Concepts for Concurrent Programming

Fred B. Schneider[1]

Department of Computer Science
Cornell University
Ithaca, New York, U.S.A 14853

Gregory R. Andrews[2]

Department of Computer Science
University of Arizona
Tucson, Arizona, U.S.A. 85721

Abstract. Techniques for reasoning about safety properties of concurrent programs are discussed and implications for program design noted. The relationship between interference freedom and synchronization mechanisms is described. The methods are illustrated with a number of examples, including partial correctness of a bank simulation, and mutual exclusion, non-blocking, and deadlock freedom of a solution to the critical section problem.

Keywords: auxiliary variables, concurrency, condition synchronization, conditional and unconditional atomic actions, critical section problem, deadlock freedom, interference freedom, invariant, fairness, liveness property, mutual exclusion, proof outline logic, safety property, scheduling policy, synchronization.

Contents

These notes are excerpted from *Concurrent Programming: Centralized and Distributed*, by G.R. Andrews and F.B. Schneider, in preparation.

[1]Supported by NSF grant DCR-8320274, a grant from the Office of Naval Research, and an IBM Faculty Development Award.

[2]Supported by NSF grant DCR-8402090.

A *concurrent program* consists of a collection of processes and shared objects. Each process is defined by a sequential program; the shared objects allow these programs to cooperate in accomplishing some task. The processes can be implemented by *multiprogramming*, where all share a single processor and are executed one at a time in an interleaved manner, by *multiprocessing*, where each is executed on its own processor in parallel with the others, or by some combination of these approaches. The shared objects can be implemented in shared memory or might simply be a computer-communications network.

Operating systems are among the best known examples of concurrent programs. An operating system manages execution of user tasks and controls processors and input/output devices that operate in parallel. Consequently, it has a natural specification as a concurrent program, with a separate process controlling each user task and hardware resource. And, like any large system, an operating system must be organized so that it can be understood and modified. Structuring a system as a collection of processes and shared objects has proved to be an effective way to achieve this.

Concurrent programs are not the sole province of those who design operating systems, however. They are useful whenever an application involves real or apparent parallelism, as the following examples show.

- In order for a database system to service many terminals in a timely manner, transactions are processed concurrently. Implementing transactions so that they function correctly despite other transactions manipulating the database is a concurrent programming problem.
- The availability of inexpensive microprocessors has made possible construction of computing systems that previously were not economically feasible. For example, such systems are being employed to control nuclear reactors, chemical plants, and aircraft. The programs for these applications frequently are concurrent programs.
- Computer networks are becoming widespread. Such networks consist of a collection of processors interconnected by communications lines. The protocols that enable processors to exchange data are concurrent programs.

In order to cooperate, processes must communicate and synchronize. *Communication* allows one process to influence execution of another and can be accomplished using *shared variables* or *message passing*. When shared variables are used, a process writes to a variable that is read by another process; when message passing is used, a process sends a message to another process. Both modes of communication involve a delay between the sending of a piece of information and its receipt. This delay has profound consequences because information obtained by a process can reflect a past state of the sender. Orchestrating cooperation among processes when the exact state of the system is unavailable can make designing a concurrent program rather difficult.

To communicate, one process sets the state of a shared object and the other reads it. This works only if the shared object is read after it has been written—reading the object before it is written can return a meaningful, but erroneous, value. Thus, communication between asynchronous processes cannot occur without synchronization. Two forms of synchronization are useful in concurrent programs. The first, *mutual exclusion*, involves grouping actions into *critical sections* that are never interleaved during execution, thereby ensuring that inconsistent states of a given process are not visible to other processes. The second form, *condition synchronization*, delays a process until the system state satisfies some specified condition. Both forms of synchronization restrict interleavings of processes. Mutual exclusion restricts interleavings by eliminating control points in a process; condition synchronization restricts interleavings by causing a process to be delayed at a given control point.

A simple example illustrates these types of synchronization. Communication between a sender process and receiver process is often implemented using a shared buffer. The sender writes into the buffer; the receiver reads from the buffer. Mutual exclusion is used to ensure that a partially written message is not read—access to the buffer by the sender and receiver is made mutually exclusive. Condition synchronization is used to ensure that a message is not overwritten or read twice—the sender is prevented from writing into the buffer until the last message written has been read, and the receiver is prevented from rereading the buffer until a new message has been written.

A program *state* associates a value with each variable. Variables include those explicitly defined by the programmer and those, like the program counter for each process, that are hidden. Execution of a sequential program results in a sequence of *atomic actions*, each of which transforms the state indivisibly. Execution of a concurrent program results in an interleaving of the sequences of atomic actions for each component process and can be viewed as a *history*

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_i} s_i \xrightarrow{\alpha_{i+1}} \cdots$$

where the s_i's denote states, the α_i's denote atomic actions, and the sequence $\alpha_1 \alpha_2 \ldots$ is an interleaving of the sequences of atomic actions resulting from execution of the processes. Note that even multiprocessing can be modeled in this way; the effects of executing a set of atomic actions in parallel is equivalent to executing them in some arbitrary, serial order, because the state transformation caused by an atomic action is indivisible and therefore cannot be affected by atomic actions executed in parallel with it.

In order to describe a history, it suffices to use either the sequence of states or the initial state s_0 and the sequence of atomic actions. Given a sequence of states, we can construct the sequence of atomic actions by looking at the program counters in pairs of adjacent states to see which atomic action was scheduled between them; given a sequence of atomic actions, we simulate execution to obtain the sequence of states.

There is good reason to prefer sequences of states to sequences of actions. Given a state, it is possible to determine the set of possible future states. We can determine which atomic actions are eligible for execution by looking at the values of the program counters in the state, and we can determine what each atomic action will do by looking at the values of variables. Thus, in a formalism based on sequences of states, only the last element in a sequence is needed to determine possible next states in the sequence. The future is defined by the present. In a formalism based on sequences of atomic actions, a state is defined by a sequence of atomic actions that leave the system in that state. Therefore, a sequence of atomic actions is required to determine possible next atomic actions in the sequence. This can be notationally burdensome, to say the least.

The effect of executing a concurrent program is defined by a set of histories, each history corresponding to one possible interleaving of the sequences of atomic actions that result from execution of its processes. For all but trivial concurrent programs, this set is apt to be quite large—so large that it might be impossible to enumerate, much less inspect, each of its elements in order to ascertain aspects of the behavior of the program. Therefore, the approach we will take for developing and analyzing concurrent programs is based on the use of abstraction; it is sometimes called *assertional reasoning*. Instead of enumerating sets of sequences of states, we characterize the elements (histories) in the set by describing their properties of interest. Instead of enumerating program states, we use *assertions*—

formulas of predicate logic—to characterize sets of states. Finally, use of a *programming logic* allows programs to be understood as implementing a relation between assertions, rather than as an object that is executed.

When reasoning about concurrent programs, only properties that concern *all* executions of the program are of interest. This rules out properties that are described using "sometimes", such as "the program sometimes terminates". In practice, one is rarely interested in a program that exhibits desired behavior only sometimes, so this is not a real limitation. It also rules out properties of the program's structure, such as the number of modules or lines of code, but then these can be checked in other ways (e.g. by a compiler).

Any property of all executions of a concurrent program can be formulated in terms of safety and liveness.

(1.1) **Safety.** A *safety property* asserts that nothing "bad" happens throughout execution.
(1.2) **Liveness.** A *liveness property* asserts that something "good" eventually does happen.

For example, the property that a program always produces the correct answer can be formulated using one safety property and one liveness property. The safety property is that the program never terminates with the wrong answer—terminating with the wrong answer is the "bad thing". The liveness property is that the program eventually does terminate—termination is the "good thing". We might also desire that a program generate answers in a timely manner. This is also a safety property, where the "bad thing" is that the clock (an hidden variable) has a certain value and the program counter (also an hidden variable) has not reached the statement that generates the answer.

The key attribute of safety properties is that once the proscribed "bad thing" happens, no subsequent execution can cause the safety property to hold. On the other hand, the key attribute of liveness properties is that no partial execution is irremediable: it always remains possible for the "good thing" to occur during subsequent execution.

A *program* consists of *declarations* followed by *statements*. Declarations define variables and associate a data type, and perhaps an initial value, with each; statements change the values of variables.

The syntax of a declaration in our programming notation is:

var $\overline{id_1} : type_1; \quad \cdots \quad \overline{id_n} : type_n$

Each $\overline{id_i}$ is a list of identifiers separated by commas and each $type_i$ denotes a data type.

Simple data types, like integers and booleans are denoted simply by giving their names. An array data type is specified using

array $[\, lb_1 \,..\, ub_1, \quad \cdots \quad lb_n \,..\, ub_n \,]$ **of** *type*

where *type* can be any data type and each *subscript specifier* $lb_i \,..\, ub_i$ gives the range of permissible subscript values for the i^{th} dimension of the array. If $lb_i > ub_i$ then no subscript values are permissible for that dimension and the array is empty. Examples of array declarations are

var a, b : **array** $[1 \,..\, n]$ **of** *integer*;
    matrix : **array** $[1 \,..\, rows, 1 \,..\, cols]$ **of** *boolean*

which define a and b to be one-dimensional arrays of integers and *matrix* to be a two-dimensional array of booleans. To shorten array declarations, we place the range specifiers immediately after the variable name and omit the keywords **array** and **of**. Thus, the above array declarations would be shortened to

var $a[1 \,..\, n], b[1 \,..\, n] : integer;$
    $matrix[1 \,..\, rows, 1 \,..\, cols] : booleans.$

To refer to an individual element of an array, the variable name is given, followed by an expression enclosed in brackets. For example, $a[1]$, $b[i+j]$, and $matrix[rows, j+k]$ refer to elements of the arrays declared above.

Variables can be initialized by using the optional **initial** clause. An **initial** clause follows the data type in a declaration and specifies the value of each identifier that precedes it. For example,

var i, j : *integer* **initial** 1, 1;

specifies that i and j are initialized to 1.

Execution of the **skip** statement has no effect on any program variable, but terminates promptly. Its syntax is:

skip

Execution of the assignment statement

$x_1, x_2, ..., x_n := e_1, e_2, ..., e_n$

where the *targets* of the assignment $x_1, ..., x_n$ agree in number and data type with expressions $e_1, ..., e_n$, is performed as follows. The values of any expressions in the targets are computed. (An expression can appear in a target that denotes an element of a composite variable. For example, the target $x[e]$ denotes an element of array x, and the value of e must be computed to determine which one.) Then, the values of $e_1, ..., e_n$ are computed. Finally, x_1 is set to the value computed for e_1, then x_2 is set to the value computed for e_2, and so on. If any of the x_i is undefined (e.g. x_i is an array reference $x[e]$ and the value of e is outside the range of permissible subscripts) or any of the e_i is undefined (e.g. evaluating a/b with $b = 0$), then the assignment statement does not terminate.

This form of assignment statement is called the *multiple assignment* because it allows the value of more than one variable to be changed at a time. It is more elegant than the *single-assignment* statement found in most programming notations. For example,

$x, y := y, x$

interchanges the values of x and y. When single-assignment statements are used, three assignments and an additional variable are required.

The statement composition operator ";" allows two statements to be combined to form a new one. The new statement is executed by executing the first and, when (and if) it terminates, executing the second. For example, sequential execution of S_1 followed by S_2 is specified by

S_1; S_2.

The syntax of the **if** statement is

if $G_1 \rightarrow S_1$ [] $G_2 \rightarrow S_2$ [] ... [] $G_n \rightarrow S_n$ **fi**

where each *guard* G_i is a boolean-valued expression and each S_i is a statement. Execution of the **if** proceeds as follows. If no guard is *true*, then the **if** does not terminate. Otherwise, one of the *guarded statements*[3] $G_i \rightarrow S_i$ where G_i is *true* is selected, and S_i is executed.

Unlike the other statements discussed thus far, the **if** is *non-deterministic*—its execution may not be completely determined by the state in which it is started. This is because it is unspecified which guarded statement among those with true guards is actually selected for execution, and more than one guard might be *true*. An advantage of using non-deterministic control structures is that they do not force the programmer to overspecify a computation. For example, in

if $x \le y \rightarrow maxval := y$ [] $x \ge y \rightarrow maxval := x$ **fi**

which sets $maxval$ to the maximum of x and y, either of the guarded statements can be selected when $x = y$.

The **do** statement is used to specify iteration. Whereas in an **if** one guarded statement with a true guard is selected and executed, in a **do**, this process is repeated until all the guards are *false*. Execution of

---

[3]In the literature, these are sometimes called *guarded commands*.

$$\textbf{do}\ \ G_1 \rightarrow S_1 \quad [] \quad G_2 \rightarrow S_2 \quad [] \quad ... \quad [] \quad G_n \rightarrow S_n \ \ \textbf{od}$$

proceeds by performing the following, until no longer possible: Select a guarded statement $G_i \rightarrow S_i$ where G_i is *true*, and execute S_i. Different guards might be *true* on different iterations, and therefore a different statement might be selected for execution in each iteration. Note that **do** is non-deterministic if two or more guarded statements have true guards at the beginning of some iteration.

An example program written in our notation is given in Figure 2.1. The program stores in t the sum of the values in array $b[0\,..\,n-1]$, assuming $n \geq 0$.

Recall from §1.2 that the behavior of a program is described by a set of histories. Each history contains a sequence of states corresponding to a particular execution of the program. Computing the set of histories for a program is often an intractable task. Fortunately, we are usually interested only in establishing that the histories all satisfy some given property, rather than in the details of each history. Such reasoning can be accomplished using a programming logic to derive properties of program histories directly from the program text.

Proof Outline Logic is a programming logic for reasoning about safety properties. It is an extension of Predicate Logic, and thus contains all the formulas, axioms, and inference rules of Predicate Logic. In addition, it contains formulas called *proof outlines* of the form

$$PO\colon\ \{P\}\ \hat{S}\ \{Q\}$$

where PO is an optional label, $\hat{S}$ is an *annotated program*—a program S in which each statement is preceded and followed by zero or more assertions—and P and Q are *assertions*. Assertions are Predicate Logic formulas and describe the program state at various points during execution. Thus, a formula of Proof Outline Logic describes the initial, final, and possibly some intermediate states of a program.

In proof outline PO above, P is called the *precondition*, denoted by $pre(PO)$, and Q is called the *postcondition*, denoted by $post(PO)$. In addition, the assertion immediately preceding a statement S' in PO is considered the precondition of S', denoted by $pre_{PO}(S')$, and the assertion immediately following S' is considered the postcondition of S' and denoted by $post_{PO}(S')$. When the subscript PO is clear from the context, it is omitted. This terminology is illustrated in the following example proof outlines, not all of which are theorems of Proof Outline Logic:

$PO1$: $\{x=0 \quad y=3\}$ **skip** $\{x=21 \quad y=3\}$

$PO2$: $\{x=\mathrm{X}\}$
  if $x \geq 0 \rightarrow \{x \geq 0\}$ **skip** $\{x = abs(\mathrm{X})\}$
  $[]\ x \leq 0 \rightarrow x := -x\ \{x \geq 0\}\ \{x = abs(\mathrm{X})\}$
  fi
  $\{x = abs(\mathrm{X})\}$

In them, we have:

$$pre(PO1) \ \equiv\ x=0 \quad y=3$$
$$post(PO1) \ \equiv\ x=21 \quad y=3$$

var $t,\ i,\ b[0\,..\,n-1]$: *integer*;

$t,\ i := 0,\ 0;$
do $i \neq n \rightarrow t,\ i := t+b[i],\ i+1$ **od**

Figure 2.1. Summing Elements of an Array

$pre_{PO1}(\mathbf{skip}) \ \equiv \ x=0 \quad y=3$
$post_{PO1}(\mathbf{skip}) \ \equiv \ x=21 \quad y=3$

$pre(PO2) \ \equiv \ x=\mathrm{X}$
$post(PO2) \ \equiv \ x=abs(\mathrm{X})$
$pre_{PO2}(\mathbf{skip}) \ \equiv \ x \geq 0$
$post_{PO2}(\mathbf{skip}) \ \equiv \ x=abs(\mathrm{X})$
$pre_{PO2}(x := -x) \ \equiv$ undefined
$post_{PO2}(x := -x) \ \equiv \ x \geq 0$

To give a formal interpretation to proof outlines, we distinguish between two types of free variables.[4] *Logical variables* are designated, free variables in assertions and do not appear in program statements. In these notes, logical variables are typeset using uppercase roman letters. All other free variables are called *program variables*. In *PO1* and *PO2* above, X is a logical variable; x and y are program variables. Program variables obtain their values from the state. Logical variables do not; instead they are implicitly universally quantified over the entire proof outline. This permits the following

(3.1) **Interpretation for Proof Outlines.** Let $\bar{v}$ be a list of values, one for each of the logical variables. For a state s, let $A(\bar{v}, s)$ denote the value of assertion A with every logical variable replaced by its value from $\bar{v}$ and every program variable replaced by its value from state s.

The interpretation of a proof outline

$PO\colon \ \{P\} \ \hat{S} \ \{Q\}$

is that for all $\bar{v}$, if execution of annotated program $\hat{S}$ is started in some state s at the statement following any assertion A (including P or Q) in PO and $A(\bar{v}, s)=true$, then if the state is s' when the next assertion A' in PO is encountered, $A'(\bar{v}, s')=true$. □

This interpretation of proof outlines is based on *partial correctness*, which is a safety property requiring that a program never reaches a "wrong" state when started in a "right" state. In this case, a "right" state is one that satisfies an assertion—usually, but not always—the precondition of the proof outline; a "wrong" state is one that does not satisfy an assertion encountered during execution—usually, but not always—the postcondition of the proof outline. Notice that partial correctness does not stipulate that the program terminates or that any given assertion is actually reached.

In addition to the axioms and inference rules of Predicate Logic, Proof Outline Logic has one axiom or inference rule for each type of statement, as well as some statement-independent inference rules. Each rule is defined so that it is sound with respect to interpretation of proof outlines (3.1) and the operational semantics of our programming notation.

The **skip** Axiom is simple, since **skip** has no effect on any program or logical variable.

(3.2) **skip** *Axiom*: $\{P\}$ **skip** $\{P\}$

The axiom states that anything about the program and logical variables that holds before executing **skip** also holds after it has terminated.

To understand the Assignment Axiom, consider a multiple-assignment $\bar{x} := \bar{e}$ where $\bar{x}$ is a list x_1, x_2, ..., x_n of identifiers (i.e. not elements of arrays) and $\bar{e}$ is a list e_1, e_2, ..., e_n of expressions. If execution of this multiple assignment does not terminate, then the axiom is valid for any choice of postcondition P. If execution terminates, then its only effect is to change the value denoted by each target x_i to that of the value denoted by the corresponding expression e_i before execution was begun. Thus, to be able to conclude that P is *true* when the multiple assignment terminates, execution must begin in a state in which the assertion obtained by replacing each occurrence of x_i in P by e_i holds. This means that if $P^{\bar{x}}_{\bar{e}}$ is *true* before the multiple assignment is executed and execution terminates, then P will be *true* of

[4]In addition to free variables, assertions can contain bound variables in quantified expressions.

the resulting state. Thus, we have the

(3.3) *Assignment Axiom*: $\{P^{\bar{x}}_{\bar{e}}\}\ \bar{x} := \bar{e}\ \{P\}$.

A proof outline for the composition of two statements can be derived from proof outlines for each of its components.

(3.4) *Statement Composition Rule*: $$\frac{\{P\}\ \hat{S}_1\ \{Q\}, \qquad \{Q\}\ \hat{S}_2\ \{R\}}{\{P\}\ \hat{S}_1;\ \{Q\}\ \hat{S}_2\ \{R\}}$$

When executing $S_1; S_2$, anything that is *true* when S_1 terminates will hold when S_2 starts. From the first hypothesis of the Statement Composition Rule, we conclude that if execution is started at the beginning of S_1 in a state satisfying P then every assertion in $\hat{S}_1$ encountered during execution will be *true*. Thus, if S_1 terminates, Q will be *true*. From the second hypothesis, we conclude that if execution of S_2 is begun in a state satisfying Q, then every assertion in $\hat{S}_2$ encountered during execution will be *true*, and if S_2 terminates, R will hold. Therefore, if $S_1; S_2$ is started in a state satisfying P, then every assertion in $\hat{S}_1;\ \{Q\}\,\hat{S}_2$ encountered will be *true*, and the Statement Composition Rule is sound.

Execution of **if** ensures that a statement S_i is executed only when its guard G_i is *true*. Thus, if the **if** is executed in a state satisfying P, then $P \;\; G_i$ will hold just before S_i is executed. Knowing that under these circumstances every assertion in $\hat{S}_i$ encountered will be *true*—due to the i^{th} hypothesis of the rule—is sufficient to establish that Q will hold should the **if** terminate. Thus, we have

(3.5) **if** *Rule*:

$$\frac{\{P \;\; G_1\}\ \hat{S}_1\ \{Q\}, \qquad \ldots, \qquad \{P \;\; G_n\}\ \hat{S}_n\ \{Q\}}{\{P\}\ \ \mathbf{if}\ \ G_1 \rightarrow \{P \;\; G_1\}\ \hat{S}_1\ \{Q\}\ [\,]\ \ \cdots\ \ [\,]\ G_n \rightarrow \{P \;\; G_n\}\ \hat{S}_n\ \{Q\}\ \mathbf{fi}\ \ \{Q\}}$$

The inference rule for **do** is based on a *loop invariant*—an assertion that holds both before and after every iteration of a loop.

(3.6) **do** *Rule*:

$$\frac{\{I \;\; G_1\}\ \hat{S}_1\ \{I\}, \qquad \ldots, \qquad \{I \;\; G_n\}\ \hat{S}_n\ \{I\}}{\begin{array}{l} \{I\} \\ \mathbf{do}\ G_1 \rightarrow \{I \;\; G_1\}\ \hat{S}_1\ \{I\} \\ \quad [\,] \quad \cdots \\ \quad [\,]\ G_n \rightarrow \{I \;\; G_n\}\ \hat{S}_n\ \{I\} \\ \mathbf{od} \\ \{I \;\; \neg G_1 \;\; \ldots \;\; \neg G_n\} \end{array}}$$

The hypotheses of the rule require that if execution of S_i is begun in a state in which I and G_i are *true*, every assertion in $\hat{S}_i$ encountered will hold, and if execution terminates, I will again be *true*. Hence, if a **do** statement is executed starting in a state satisfying I, then every assertion encountered in $\hat{S}_i$ will hold, and I will be *true* at the beginning and end of each iteration. Thus, I will hold if the **do** terminates. The **do** terminates when no guard is *true*, so $\neg G_1 \;\; \ldots \;\; \neg G_n$ will also hold at that time. Therefore, the inference rule is sound with respect to the interpretation of proof outlines (3.1) and the execution of **do**.

The Rule of Consequence allows the precondition of a proof outline to be strengthened and the postcondition to be weakened, based on deductions possible in Predicate Logic.

(3.7) *Rule of Consequence*: $$\frac{P' \mathit{implies}\ P, \qquad \{P\}\ \hat{S}\ \{Q\}, \qquad Q \;\; Q'}{\{P'\}\,\{P\}\ \hat{S}\ \{Q\}\,\{Q'\}}$$

Hypothesis $\{P\}\ \hat{S}\ \{Q\}$ requires that every assertion in $\hat{S}$ encountered during execution holds and that Q is *true* if S terminates. If P holds whenever P' does, which is the case if P' *implies* P, then every assertion encountered during execution will hold if S is begun in a state satisfying P'. Thus, Q will be *true* should S terminate, so $\{P'\}\,\{P\}\ \hat{S}\ \{Q\}$ is valid. Similarly, from $Q \;\; Q'$, whenever Q is *true*, so is Q'. Therefore, if S terminates and Q is *true*, Q' will also hold, and the conclusion of the rule follows.

The Rule of Consequence is the only way to introduce adjacent assertions in a theorem of Proof Outline Logic, and therefore a pair of adjacent assertions $\{A\}\ \{A'\}$ in a proof outline always means $A\ \ A'$. Since hypotheses of the Rule of Consequence involve theorems of Predicate Logic, which is incomplete, Proof Outline Logic is incomplete. Proof Outline Logic, however, is relatively complete with respect to Predicate Logic. Failure to prove a valid formula in Proof Outline Logic must be attributed to the inabilities of the prover or to valid but unprovable formulas in Predicate Logic, rather than to a weakness in the axioms or inference rules of Proof Outline Logic.

The Logical Variable Rule allows a logical variable to be renamed or replaced by a specific value.[5]

(3.8) *Logical Variable Rule*: $\{P\}\ \hat{S}\ \{Q\}$,
X a logical variable,
Y a constant or a logical variable
$\{P^{\mathrm{X}}_{\mathrm{Y}}\}\ \hat{S}^{\mathrm{X}}_{\mathrm{Y}}\ \{Q^{\mathrm{X}}_{\mathrm{Y}}\}$

The soundness of this rule is based on the interpretation of proof outlines, where a logical variable X becomes a bound variable whose scope is the entire proof outline.

The Deletion Rule allows assertions in the annotated program of a proof outline to be deleted.

(3.9) *Deletion Rule*: Let $\hat{S}'$ be the result of deleting one or more assertions from annotated program $\hat{S}$.

$$\frac{\{P\}\ \hat{S}\ \{Q\}}{\{P\}\ \hat{S}'\ \{Q\}}$$

To see that the rule is sound, notice that every assertion in $\hat{S}'$ also appears in $\hat{S}$ and if an assertion encountered in $\hat{S}$ is *true*, then that assertion, if present in $\hat{S}'$, must also be *true* when encountered.

We will frequently be concerned with a special class of proof outlines, called *triples*, that contain only two assertions: a single precondition and a single postcondition. The Deletion Rule provides a way to infer a triple $\{P\}\ S\ \{Q\}$ from a proof outline $\{P\}\ \hat{S}\ \{Q\}$.

The Conjunction and Disjunction Rules (3.10) and (3.11) allow proof outlines for the same program with assertions in corresponding positions to be combined into a single proof outline. Given two proof outlines for a program S, $PO1(S)$ and $PO2(S)$, in which a precondition (postcondition) for each statement S' appears in $PO1(S)$ if and only if a precondition (postcondition) for S' appears in $PO2(S)$, define $PO1(S)\ PO2(S)$ to be the proof outline obtained by taking the point-wise conjunction of corresponding assertions in $PO1(S)$ and $PO2(S)$.[6] For example, given

$PO1$: $\{x=5\}\ \ x := x+1;\ \ \{x=6\}\ \ y := y+1\ \ \{x=6\}$

$PO2$: $\{y=2\}\ \ x := x+1;\ \ \{y=2\}\ \ y := y+1\ \ \{y=3\}$

then $PO1\ PO2$ is:

$\{x=5\ \ y=2\}\ \ x := x+1;\ \ \{x=6\ \ y=2\}\ \ y := y+1\ \ \{x=6\ \ y=3\}$

The following Conjunction Rule states that $PO1(S)\ PO2(S)$ can be inferred from $PO1(S)$ and $PO2(S)$.

(3.10) *Conjunction Rule*: $\dfrac{PO1(S),\quad PO2(S)}{PO1(S)\ PO2(S)}$

If execution is started at a statement S' in S with $pre_{PO1\ PO2}(S')=true$ then by construction both $pre_{PO1}(S')$ and $pre_{PO2}(S')$ will hold. By the hypotheses, every subsequent assertion in both $PO1$ and

[5] Even though logical variables cannot appear in program statements, they can appear in the assertions in an annotated program. This explains the reason for substituting Y for X in $\hat{S}$ of the conclusion of the Logical Variable Rule.

[6] Thus, for each statement S' in S: $pre_{PO1(S)\ PO2(S)}(S') = pre_{PO1}(S')\ \ pre_{PO2}(S')$ and $post_{PO1(S)\ PO2(S)}(S') = post_{PO1}(S')\ \ post_{PO2}(S')$.

PO2 will hold when encountered. Thus, by construction, every subsequent assertion in *PO1 PO2* will hold when encountered, and the rule is sound.

Define $PO1(S)\ PO2(S)$ to be the proof outline that results from the point-wise disjunction of corresponding assertions in $PO1(S)$ and $PO2(S)$. The Disjunction Rule allows $PO1(S)\ PO2(S)$ to be inferred from $PO1(S)$ and $PO2(S)$.

(3.11) *Disjunction Rule*: $\dfrac{PO1(S),\quad PO2(S)}{PO1(S)\ PO2(S)}$

To see that this rule is sound, suppose execution is started at a statement S' in S with the state satisfying $pre_{PO1\ PO2}(S')$. By construction, this means that either $pre_{P01}(S')$ or $pre_{P02}(S')$ must hold. If $pre_{P01}(S')$ holds, then by the first hypothesis we conclude that the disjunct from *PO1* will hold in next assertion in *PO1 PO2* that is encountered. The second case follows in the same way from the other hypothesis. Thus, the inference rule is sound.

Tables 3.1 and 3.2 summarize the axioms and inference rules of Proof Outline Logic.

Since a proof in Proof Outline Logic is merely a sequence of proof outlines and Predicate Logic formulas, where each is an axiom or can be derived from previous lines by using inference rules, the logic provides a mechanical way to check partial correctness properties of programs. In the following, we illustrate the use of Proof Outline Logic by proving partial correctness of the program of Figure 2.1. We prove that if the program is started in a state where $0 \le n$ holds and execution terminates, then t will contain the sum of the values in $b[0]$ through $b[n-1]$.[7]

skip *Axiom*: $\{P\}$ **skip** $\{P\}$

Assignment Axiom: $\{P^{\bar{x}}_{\bar{e}}\}\ \bar{x} := \bar{e}\ \{P\}$

Statement Composition Rule: $\dfrac{\{P\}\ \hat{S}_1\ \{Q\},\qquad \{Q\}\ \hat{S}_2\ \{R\}}{\{P\}\ \hat{S}_1;\ \{Q\}\ \hat{S}_2\ \{R\}}$

$\{P\ \ G_1\}\ \hat{S}_1\ \{Q\},\quad \cdots \quad \{P\ \ G_n\}\ \hat{S}_n\ \{Q\}\ \{P\}$
if $G_1 \rightarrow \{P\ \ G_1\}\ \hat{S}_1\ \{Q\}$ [] $\cdots$ **if** *Rule*:
[] $G_n \rightarrow \{P\ \ G_n\}\ \hat{S}_n\ \{Q\}$ **fi** $\{Q\}$

$\{I\ \ G_1\}\ \hat{S}_1\ \{I\},\quad \cdots \quad \{I\ \ G_n\}\ \hat{S}_n\ \{I\}\ \{I\}$
do $G_1 \rightarrow \{I\ \ G_1\}\ \hat{S}_1\ \{I\}$ [] $\cdots$ **do** *Rule*:
[] $G_n \rightarrow \{I\ \ G_n\}\ \hat{S}_n\ \{I\}$ **od** $\{I\ \ \neg G_1\ \ ...\ \ \neg G_n\}$

Table 3.1. Proof Outline Logic:
Axioms and Inference Rules for Statements

[7] Throughout, we use the notation $(\Sigma j:\ 1 \le j \le n:\ b[j])$ in place of $\sum_{j=1,n} b[j]$.

Rule of Consequence: $\dfrac{P' \textit{implies } P,\ \ \{P\}\ \hat{S}\ \{Q\},\ \ Q\ \ Q'}{\{P'\}\{P\}\ \hat{S}\ \{Q\}\{Q'\}}$

Logical Variable Rule: $\dfrac{\{P\}\ \hat{S}\ \{Q\},\ \ \mathrm{X} \text{ a logical variable, } \mathrm{Y} \text{ a constant or a logical variable}}{\{P^{\mathrm{X}}_{\mathrm{Y}}\}\ \hat{S}^{\mathrm{X}}_{\mathrm{Y}}\ \{Q^{\mathrm{X}}_{\mathrm{Y}}\}}$

Deletion Rule: Let $\hat{S}'$ be the result of deleting one or more assertions from annotated program $\hat{S}$. Then,

$$\frac{\{P\}\ \hat{S}\ \{Q\}}{\{P\}\ \hat{S}'\ \{Q\}}$$

Conjunction Rule: $\dfrac{PO1(S),\quad PO2(S)}{PO1(S)\ \ PO2(S)}$

Disjunction Rule: $\dfrac{PO1(S),\quad PO2(S)}{PO1(S)\ \ PO2(S)}$

Table 3.2. Proof Outline Logic:
General Axioms and Inference Rules

By the Assignment Axiom (3.3):

1. $\{0 \le i+1 \le n \quad t+b[i] = (\Sigma j: 0 \le j \le i: b[j])\}$
   $t, i\ := t+b[i],\ i+1$
   $\{0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j])\}$

By Predicate Logic:

2. $(0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j]) \quad i \ne n)$
   $\qquad 0 \le i+1 \le n \quad t+b[i] = (\Sigma j: 0 \le j \le i: b[j])$

By the Rule of Consequence (3.7) with lines 1 and 2:

3. $\{0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j]) \quad i \ne n\}$
   $\{0 \le i+1 \le n \quad t+b[i] = (\Sigma j: 0 \le j \le i: b[j])\}$
   $t, i\ := t+b[i],\ i+1$
   $\{0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j])\}$

By the Deletion Rule (3.9) with line 3:

4. $\{0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j]) \quad i \ne n\}$
   $t, i\ := t+b[i],\ i+1$
   $\{0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j])\}$

By the **do** Rule (3.6) with line 4:

5. $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$
   do $i \ne n \rightarrow \{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j]) \quad i \ne n\}$
   $\qquad\qquad t, i := t + b[i], i+1$
   $\qquad\qquad \{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$
   od
   $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j]) \quad i = n\}$

By the Assignment Axiom (3.3):

6. $\{0 \le 0 \le n \quad 0 = (\Sigma j : 0 \le j \le -1 : b[j])\}$
   $t, i := 0, 0$
   $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$

By Predicate Logic:

7. $0 \le n \quad (0 \le 0 \le n \quad 0 = (\Sigma j : 0 \le j \le -1 : b[j]))$

By the Rule of Consequence (3.7) with lines 6 and 7:

8. $\{0 \le n\}$
   $\{0 \le 0 \le n \quad 0 = (\Sigma j : 0 \le j \le -1 : b[j])\}$
   $t, i := 0, 0$
   $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$

By the Deletion Rule (3.9) with line 8:

9. $\{0 \le n\}$
   $t, i := 0, 0$
   $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$

By the Statement Composition Rule (3.4) with lines 5 and 9:

10. $\{0 \le n\}$
    $t, i := 0, 0$
    $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$
    do $i \ne n \rightarrow \{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j]) \quad i \ne n\}$
    $\qquad\qquad t, i := t + b[i], i+1$
    $\qquad\qquad \{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$
    od
    $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j]) \quad i = n\}$

By Predicate Logic:

11. $(0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j]) \quad i = n)$
    $\qquad t = (\Sigma j : 0 \le j \le n-1 : b[j])$

By the Rule of Consequence (3.7) with lines 10 and 11:

12. $\{0 \le n\}$
    $t, i := 0, 0$
    $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$
    do $i \ne n \rightarrow \{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j]) \quad i \ne n\}$
    $\qquad\qquad t, i := t + b[i], i+1$
    $\qquad\qquad \{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j])\}$
    od
    $\{0 \le i \le n \quad t = (\Sigma j : 0 \le j \le i-1 : b[j]) \quad i = n\}$
    $\{t = (\Sigma j : 0 \le j \le n-1 : b[j])\}$

Proving theorems of Proof Outline Logic can be a tedious task, as illustrated above. Fortunately, it is not necessary to show that a proof outline is a theorem in order to establish that it is valid. The following test can be used instead. The test is based on the interpretation of proof outlines and involves showing that for any statement S', if $pre_{PO}(S')$ is present and execution of S' is begun in a state where $pre_{PO}(S')$ holds, then when the next assertion is encountered, it too will hold. Validity then follows by induction on the number of assertions that are encountered during execution.

(3.12) **Simple Proof Outline Validity Test.**

(1) For each assertion Q in the proof outline, identify the preceding assertion on every execution path to Q. Let this set of assertions be $pred(Q)$.

(2) For each assertion P in $pred(Q)$, identify the code that is executed between the time control is at P and control is next at Q. Let this be $exec(P, Q)$.

(a) If $exec(P, Q)$ does not change any variable or involve evaluating any guard then check that $P \quad Q$ is valid.

(b) If $exec(P, Q)$ involves executing some statement S then check that $\{P\}\ S\ \{Q\}$ is a theorem of Proof Outline Logic. For example, if $exec(P, Q)$ is an assignment $\bar{x} := \bar{e}$ then check that $P \quad Q_{\bar{e}}^{\bar{x}}$ is valid.

(c) If $exec(P, Q)$ is evaluation of guard G that is *true* then check that $P \quad G \quad Q$ is valid.

(d) If $exec(P, Q)$ is evaluation of guards $G_1, G_2, \ldots, G_n$ that are all *false* then check that

$$(P \quad \neg G_1 \quad \neg G_2 \quad \ldots \quad \neg G_n) \quad Q$$

is valid. □

Checking the validity of a proof outline is quite simple and can be done by inspection provided the program is annotated with a loop invariant for each loop. We now illustrate validity test (3.12) on the proof outline of Figure 3.1, which is based on line 12 of the proof above. All the (gory) details are worked out only so you can be sure you understand how to perform the test.

P1: Assertion *P1* has no predecessors. Therefore, no code is executed to reach it and no obligations need be satisfied.

P2: Assertion *P2* has one predecessor, *P1*. To go from *P1* to *P2*, $t, i := 0, 0$ is executed. According to 2(b) of (3.12), $P1 \quad P2_{0,0}^{t,i}$ must be valid. It is, because

$$(0 \le n) \quad 0 \le 0 \le n \quad 0 = (\Sigma j: 0 \le j \le -1: b[j])$$

is valid.

P3: Assertion *P3* has two predecessors, *P2* and *P4*.

To go from *P2* to *P3*, the guard in the **do** must be *true*. According to 2(c), $(P2 \quad i \ne n)\ P3$ must be valid. It is.

$\{P1\colon\ 0 \le n\}$
$t, i := 0, 0;$
$\{P2\colon\ 0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j])\}$
do $i \ne n \rightarrow \{P3\colon\ 0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j]) \quad i \ne n\}$
    $t, i := t + b[i], i+1$
    $\{P4\colon\ 0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j])\}$
od
$\{P5\colon\ 0 \le i \le n \quad t = (\Sigma j: 0 \le j \le i-1: b[j]) \quad i = n\}$
$\{P6\colon\ t = (\Sigma j: 0 \le j \le n-1: b[j])\}$

Figure 3.1. Example Proof Outline

To go from $P4$ to $P3$, $i \neq n$ must hold. Again by 2(c), $(P4 \;\; i \neq n) \; P3$ must be valid. It is.

P4: Assertion $P4$ has one predecessor, $P3$. To go from $P3$ to $P4$, $t, i := t+b[i], i+1$ is executed. According to 2(b), $P3 \; P4^{t,i}_{t+b[i],i+1}$ must be valid. It is, because

$$(0 \leq i \leq n \quad t = (\Sigma j: 0 \leq j \leq i-1: b[j]) \quad i \neq n) \qquad 0 \leq i+1 \leq n \quad t+b[i] = (\Sigma j: 0 \leq j \leq i: b[j])$$

is valid.

P5: Assertion $P5$ has two predecessors, $P2$ and $P4$. To go from $P2$ or $P4$ to $P5$, guard $i \neq n$ must be *false*. According to 2(d), $(P2 \;\; i = n) \; P5$ and $(P4 \;\; i = n) \; P5$ must be valid. Both are.

P6: Assertion $P6$ has one predecessor, $P5$, and no code between them. According to 2(a), $P5 \; P6$ must be valid. This follows by substituting n for i in $P5$.

Execution of the **cobegin** statement

$$\textbf{cobegin} \;\; S_1 \;/\!\!/\; S_2 \;/\!\!/\; \cdots \;/\!\!/\; S_n \;\; \textbf{coend} \tag{4.1}$$

where S_1, ..., S_n are sequential programs is equivalent to some interleaving of the atomic actions in processes S_1 through S_n and terminates after each process has terminated. Thus, to understand the behavior of (4.1), we must identify the atomic actions of its processes.

One way to identify the atomic actions in a program is by specifying the *control points* of that program—points in the program text that delimit its atomic actions. Unless otherwise stated, we assume

(4.2) **Control Point Assumption.** There is a control point before and after every statement. □

For example, we assume that execution of an assignment statement is a single atomic action, independent of the number of target variables. In §6, we relax this assumption and introduce a notation to permit assignments to appear inside larger atomic actions.

Experience has shown that decomposition often provides an effective way to reason about a complex object. The object is first partitioned into its components, then the properties of each component are derived in isolation, and finally these properties are combined to obtain the properties of the whole. In this section, we show how to derive certain properties of a concurrent program from partial correctness properties of the sequential programs it comprises. In particular, we give an inference rule for Proof Outline Logic that permits derivation of a valid proof outline for a concurrent program from proof outlines for its processes.

Define a *complete proof outline* to be a proof outline that has an assertion at every control point. Given complete proof outlines PO_1, PO_2, ..., PO_n for sequential programs S_1 through S_n, we desire conditions to ensure that

$$\begin{array}{l} \{P\} \\ \textbf{cobegin} \;\; PO_1 \;/\!\!/\; PO_2 \;/\!\!/\; \cdots \;/\!\!/\; PO_n \;\; \textbf{coend} \\ \{Q\} \end{array} \tag{5.1}$$

is a valid and complete proof outline. Establishing that (5.1) is a complete proof outline is straightforward because PO_1, ..., PO_n are complete proof outlines. Establishing that (5.1) is valid is more complex.

During execution, a control point in a process is considered *eligible* if it defines the next atomic action to be executed by that process. According to interpretation for proof outlines (3.1), establishing the validity of (5.1) is equivalent to showing

(5.2) Whenever a control point is eligible, the state satisfies the assertion associated with that control point.

Thus, to ensure validity of (5.1), it suffices to develop conditions on P, Q, and the assertions in PO_1, ..., PO_n that ensure (5.2) holds initially and that prevent any process from invalidating it.

The first condition ensures that (5.2) is maintained when each process is started. If execution of (5.1) is started at the beginning, then due to the operational semantics of **cobegin** the control point at the start of each process S_i is eligible. Since PO_i is a complete proof outline, there is an assertion $pre(PO_i)$ associated with the control point at the start of S_i. Thus, if P holds when (5.1) is started—

and therefore (5.2) holds—then $pre(PO_i)$ must also hold in order for (5.2) to be maintained. Therefore, we must have

$$P \quad (pre(PO_1) \quad pre(PO_2) \quad \cdots \quad pre(PO_n)). \tag{5.3}$$

The next condition ensures that (5.2) is maintained when each process terminates. The control point following the **cobegin** in (5.1) is eligible only after the last atomic action in every process has executed. PO_i is a complete proof outline, so there is an assertion $post(PO_i)$ associated with the control point following the last atomic action in S_i. According to (5.2), $post(PO_i)$ will hold after the last atomic action in process S_i has executed. Thus, for (5.2) to remain valid, we must have

$$(post(PO_1) \quad post(PO_2) \quad \cdots \quad post(PO_n)) \quad Q, \tag{5.4}$$

so that Q will hold when the control point at the end of the **cobegin** becomes eligible.

Finally, we give conditions to ensure that (5.2) holds while S_1, ..., S_n execute. Suppose (5.2) holds and there is an eligible control point within one or more processes. We must ensure that if one atomic action of S_i is executed, the assertion associated with the next eligible control point will then hold. By requiring that PO_i be a complete proof outline, the existence of an assertion at the next control point is assured; by requiring that PO_i is a valid proof outline, the invariance of (5.2) follows due to interpretation of proof outlines (3.1). Thus, one condition for preserving (5.2) is

$$\text{For all } i,\ 1 \le i \le n:\ PO_i \text{ is complete and } \textit{turnstile } PO_i \tag{5.5}$$

where *turnstile* PO_i means that PO_i is a theorem of Proof Outline Logic.

We must also ensure that executing the next atomic action in S_i does not invalidate an assertion associated with an eligible control point in another process. This is called *interference freedom* and is established by proving that if an assertion A in one process holds, then execution of any other process leaves A *true*. Establishing interference freedom is simplified if we assume

(5.6) **Assertion Restriction.**

(a) Assertions depend only on the values of program and logical variables.

(b) Executing an assignment statement is the only way a process can change the value of a program variable. □

Part (a) implies that when checking interference freedom we need only consider execution of statements that change the values of program variables—for example, it is unnecessary to consider statements, like **skip**, that change only the program counter. Part (b) implies that only assignment statements can invalidate an assertion in another process.

To show that an assignment statement α cannot invalidate A, it suffices to prove that

$$NI(\alpha, A): \quad \{pre_{PO_i}(\alpha) \quad A\}\ \alpha\ \{A\}$$

is a theorem of Proof Outline Logic. Thus, we have

(5.7) **Interference freedom.** Proof outlines PO_1, PO_2, ..., PO_n are interference free if

For all i, $1 \le i \le n$:
  For all j, $i \ne j \quad 1 \le j \le n$:
    For every assignment statement α in PO_i:
      For every assertion A in PO_j: *turnstile* $NI(\alpha, A)$. □

We have derived four conditions, (5.3)−(5.5) and (5.7), that together allow proof outlines for sequential programs to be combined to form a valid proof outline for a **cobegin**. Treating these conditions as hypotheses, we get our first inference rule in Proof Outline Logic for **cobegin**.

(5.8) **cobegin** *Rule*:

(a) For all i, $1 \le i \le n$: PO_i is complete and *turnstile* PO_i,
(b) $P \quad (pre(PO_1) \quad pre(PO_2) \quad \cdots \quad pre(PO_n))$,
(c) $(post(PO_1) \quad post(PO_2) \quad \cdots \quad post(PO_n)) \quad Q$,
(d) For every assignment statement α in PO_i
For every assertion A in PO_j, $j \ne i$: *turnstile* $NI(\alpha, A)$

$\{P\}$ **cobegin** PO_1 // PO_2 // $\cdots$ // PO_n **coend** $\{Q\}$

To illustrate the use of **cobegin** Rule (5.8), consider a concurrent program to increment x by 3:[8]

cobegin $x := x+1$ // $x := x+2$ **coend** (5.9)

Obviously, if initially $x=0$, then when the **cobegin** terminates, $x=3$.

To prove this, we first construct a proof outline for each of the processes in isolation.

$\{x=0\} \ \ x := x+1 \ \ \{x=1\}$

$\{x=0\} \ \ x := x+2 \ \ \{x=2\}$

Each of these is a complete proof outline and a theorem, since each is an instance of Assignment Axiom (3.3). Hypothesis (a) of **cobegin** Rule (5.8) is now satisfied.

Combining these, we obtain a (not necessarily interference-free) complete proof outline for (5.9).

$\{P\text{: } x=0\}$
cobegin
$\{P1\text{: } x=0\} \quad \alpha_1\text{: } x := x+1 \quad \{Q1\text{: } x=1\}$
//
$\{P2\text{: } x=0\} \quad \alpha_2\text{: } x := x+2 \quad \{Q2\text{: } x=2\}$
coend
$\{Q\text{: } x=3\}$

We must now check hypotheses (b), (c), and (d) of **cobegin** Rule (5.8). Hypothesis (b) is satisfied, since $P \quad (P1 \quad P2)$. Hypothesis (d) requires that $NI(\alpha_1, P2)$, $NI(\alpha_2, P1)$, $NI(\alpha_1, Q2)$, and $NI(\alpha_2, Q1)$ are theorems.

Expanding $NI(\alpha_1, P2)$ we get

$\{pre(\alpha_1) \quad P2\} \quad \alpha_1 \quad \{P2\}$

$= \{x=0\} \ \ x := x+1 \ \ \{x=0\}$,

which is not a theorem—executing α_1 interferes with $P2$. Somehow, this interference must be eliminated.

There are two general approaches to eliminating interference in a proof outline.

(5.10) **Eliminating Interference by Strengthening Assertions.** To eliminate interference of assignment statement α with assertion A, strengthen $pre(\alpha)$ making $NI(\alpha, A)$ a theorem because $pre(NI(\alpha, A)) = \mathit{false}$. □

(5.11) **Eliminating Interference by Weakening Assertions.** To eliminate interference of assignment statement α with assertion A, weaken A making $NI(\alpha, A)$ a theorem. □

We reject (5.10) for $NI(\alpha_1, P2)$ because there is no way to strengthen $pre(\alpha_1)$ and still have it be implied by P, as required by hypothesis (b) of (5.8). So, we use (5.11), and weaken $P2$ by adding $x=1$ as a disjunct, then weaken $Q2$ in accordance with hypothesis (a), and obtain the following proof outline.

[8] Recall, according to (4.2) assignment statements are executed atomically.

$\{P\colon\ x=0\}$
cobegin
$\quad \{P1\colon\ x=0\} \quad \alpha_1\colon\ x := x+1 \quad \{Q1\colon\ x=1\}$
$//$
$\quad \{P2\colon\ x=0 \quad x=1\} \quad \alpha_2\colon\ x := x+2 \quad \{Q2\colon\ x=2 \quad x=3\}$
coend
$\{Q\colon\ x=3\}$

Expanding $NI(\alpha_1, P2)$, we now get

$$\{x=0 \quad (x=0 \quad x=1)\}\ \ x := x+1\ \ \{x=0 \quad x=1\}$$

$$= \{x=0\}\ \ x := x+1\ \ \{x=0 \quad x=1\}$$

which is a theorem. Moreover, despite weakening $P2$, we still have $P \ (P1 \ P2)$, as required by hypothesis (b) of **cobegin** Rule (5.8). And, by construction, hypothesis (a) also remains satisfied.

The next part of the interference-freedom proof is to check whether $NI(\alpha_2, P1)$ is a theorem. $NI(\alpha_2, P1)$ is

$$\{pre(\alpha_2) \quad P1\}\ \ \alpha_2\ \ \{P2\}$$

$$= \{(x=0 \quad x=1) \quad x=0\}\ \ x := x+2\ \ \{x=0\}$$

$$= \{x=0\}\ \ x := x+2\ \ \{x=0\}$$

which is not a theorem. As above, we employ (5.11) and weaken $P1$ and then recompute $Q1$ so that hypothesis (a) is again satisfied.

$\{P\colon\ x=0\}$
cobegin
$\quad \{P1\colon\ x=0 \quad x=2\} \quad \alpha_1\colon\ x := x+1 \quad \{Q1\colon\ x=1 \quad x=3\}$
$//$ (5.12)
$\quad \{P2\colon\ x=0 \quad x=1\} \quad \alpha_2\colon\ x := x+2 \quad \{Q2\colon\ x=2 \quad x=3\}$
coend
$\{Q\colon\ x=3\}$

Now, $NI(\alpha_2, P1)$ is

$$\{(x=0 \quad x=1) \quad (x=0 \quad x=2)\}\ \ x := x+2\ \ \{x=0 \quad x=2\}$$

$$= \{x=0\}\ \ x := x+2\ \ \{x=0 \quad x=2\}$$

which is a theorem. Furthermore, weakening $P1$ leaves hypothesis (b) satisfied and leaves $NI(\alpha_1, P2)$ a theorem.

Expanding the last two interference-freedom formulas, we get:

$NI(\alpha_1, Q2)$: $\quad \{x=2\}\ \ x := x+1\ \ \{x=2 \quad x=3\}$

$NI(\alpha_2, Q1)$: $\quad \{x=1\}\ \ x := x+2\ \ \{x=1 \quad x=3\}$

Both are theorems.

Finally, we check hypothesis (c) of the **cobegin** Rule (5.8).

$$(Q1 \quad Q2) \quad Q$$

$$= ((x=1 \quad x=3) \quad (x=2 \quad x=3)) \quad x=3$$

It is valid, so we have proved that (5.12) is a theorem of Proof Outline Logic and can conclude that (5.9) does increment x by 3.

While it might seem like an extraordinary amount of formal manipulation was required to deduce a simple fact about a trivial program, we shall see that the method is quite powerful and will enable us to deduce properties of concurrent programs too complex to understand simply by enumerating execution interleavings. And, the pattern followed in this simple example is typical. First, to satisfy hypothesis (a) of **cobegin** Rule (5.8), a proof outline for each process in isolation is constructed. Then, to satisfy hypothesis (b), we check that the precondition of each of these proof outlines is implied by the precondition of the **cobegin**. Next, to satisfy hypothesis (d), interference-freedom formulas are enumerated and checked. If interference is detected then proof outlines for processes are changed. When such a change is made, we check that the new proof outline for the process (in isolation) is a theorem, check that its precondition is still implied by the precondition of the **cobegin**, and recheck the other interference-freedom formulas. Finally, to satisfy hypothesis (b), the postcondition for the **cobegin** is formed from the conjunction of the postconditions of each proof outline.

Interference cannot always be eliminated simply by strengthening or weakening assertions in a proof outline, as described in (5.10) and (5.11). Some interference is an inevitable consequence of certain execution interleavings and can be prevented only by restricting the interleaving of processes. Synchronization mechanisms permit such control.

Synchronization mechanisms for both mutual exclusion and condition synchronization can be specified using angle brackets "" and "". A program within angle brackets is treated as a single atomic action—it has no internal control points and is executed to completion before a process switch is permitted.

By removing the internal control points of a program, angle brackets specify mutual exclusion. For example,

$$x := x+1;\quad x := x-1$$

defines a single atomic action that increments x by 1 and then decrements x by 1. It is not possible for a process that reads x concurrently with execution of this atomic action to obtain the value of x after it has been incremented but before it has been decremented.

Since, by definition, an atomic action cannot be interrupted before completion, it must not be started unless it will terminate. Thus, S must delay until the state is one that will lead to its termination. An atomic action that can cause delay is called a *conditional atomic action* and one that cannot an *unconditional atomic action*.

Conditional atomic actions permit mechanisms that implement condition synchronization to be specified. For example,

$$\textbf{if } sem > 0 \rightarrow sem := sem - 1 \textbf{ fi} \qquad (6.1)$$

delays until decrementing *sem* would leave it non-negative and then does the decrement. The delay is a consequence of the **if**, which would not terminate when started in a state satisfying none of its guards. A somewhat more pathological statement having the same effect as (6.1) is

$$sem := sem - 1;\ \textbf{do } sem < 0 \rightarrow \textbf{ skip od}.$$

Allowing anything to appear inside angle brackets could pose difficult implementation problems. Provided angle brackets are used only to describe synchronization mechanisms already available, such implementation problems need never be confronted. The question of what synchronization mechanisms are available depends on hardware and underlying support software.

An atomic action α is *eligible* for execution if the control point at the beginning of α is eligible. More than one atomic action might be eligible at any point during the execution of a **cobegin**. A *scheduling policy* defines which among them is executed next.

The scheduling policy implemented by a **cobegin** is important when analyzing termination and other liveness properties. For example, if there is no scheduling policy then

```
ok := true;
cobegin
    Loop:  do ok  →  skip od
  //                                                    (6.2)
    Stop:  ok := false
coend
```

might never terminate, because the scheduler is not obliged to execute an atomic action from *Stop*, even though one is always eligible. A **cobegin** is *unconditionally fair* if every unconditional atomic action that becomes eligible is eventually executed. Clearly, (6.2) would terminate eventually with an unconditionally fair **cobegin**.

Even stronger assumptions about scheduling are necessary with conditional atomic actions. This is because two things are required for a conditional atomic action to be executed.

- It must be eligible.
- The state must be one that will lead to termination.

A **cobegin** is considered *weakly fair* if it is unconditionally fair and no eligible conditional atomic action awaiting a condition G is forever delayed even though G becomes *true* and thereafter remains *true*. In contrast, a **cobegin** is considered *strongly fair* provided it is unconditionally fair and no eligible conditional atomic action awaiting a condition G is forever delayed even though G is is infinitely-often *true*. The difference between weakly-fair and strongly-fair scheduling policies is illustrated by the following program.

```
continue := true;   proceed := false;
cobegin
    Loop:  do continue → proceed := true;
                         proceed := false
           od
  //
    Stop:  if proceed → continue := false fi
coend
```

If the **cobegin** is strongly-fair, then *Loop* will terminate; if it is only weakly-fair, then *Loop* need not terminate because the condition on which *Stop* is waiting is not continuously enabled.

When angle brackets are permitted in programs, an assignment statement that is part of an angle-bracketed statement can be ignored in proving interference freedom because it is only part of an atomic action; it suffices to establish that the entire segment of code enclosed in angle brackets does not cause interference. For example, if α is an assignment statement in S, then instead of checking that $NI(\alpha, A)$ is a theorem, it suffices to check that $NI(S, A)$ is one.[9] Enclosing groups of statements in angle brackets reduces the number of interference freedom formulas that must be checked, but eliminates concurrency.

Of course, checking $NI(S, A)$ requires proving theorems of Proof Outline Logic about programs enclosed in angle brackets. The following inference rule permits this.

(6.3) *Synchronization Rule*: $\dfrac{\{P\}\ \hat{S}\ \{Q\}}{\{P\}\ \hat{S}\ \{Q\}}$

Enclosing a program in angle brackets also simplifies establishing interference freedom by eliminating some control points. S contains no internal control points, no matter how many control points S contains. Because S contains no internal control points, assertions appearing inside S cannot be invalidated by execution of other processes. This means that when showing interference freedom, it is never necessary to show $NI(\alpha, A)$ for an assertion A appearing within angle-brackets. Also note that since

[9] Since only syntactically valid programs can be enclosed in angle brackets, $NI(S, A)$ will be a formula of Proof Outline Logic.

assertions appearing inside S are not associated with control points, they are, by definition, not part of a complete proof outline. Thus, we omit them in constructing a proof outline for a concurrent program.

We can now reformulate hypothesis (d) of **cobegin** Rule (5.8) to take advantage of these simplifications to the interference freedom obligations. Define an *assignment action* to be any assignment statement not in angle brackets or any angle-bracketed program containing an assignment statement. Then, we have

(6.4) **cobegin** *Rule*:

(a) For all i, $1 \le i \le n$: PO_i is complete and $turnstile\ PO_i$.

(b) $P \quad (pre(PO_1) \quad pre(PO_2) \quad \cdots \quad pre(PO_n))$,

(c) $(post(PO_1) \quad post(PO_2) \quad \cdots \quad post(PO_n)) \quad Q$,

(d) For every assignment action α in PO_i
For every assertion A in PO_j, $j \ne i$: $turnstile\ NI(\alpha, A)$

$\{P\}$ **cobegin** $PO_1 \mathbin{/\!/} PO_2 \mathbin{/\!/} \cdots \mathbin{/\!/} PO_n$ **coend** $\{Q\}$

In order to understand how synchronization mechanisms can be used to prevent interference, suppose that interference freedom cannot be established in the following program because $NI(\alpha, A)$ is not a theorem.

cobegin
  $Process_1$: ... $\{pre(\alpha)\}\ \alpha$ ...
//
  $Process_2$: ... $S1\ \{A\}\ S2$ ...
coend

Moreover, suppose $pre(\alpha)$ cannot be strengthened nor A weakened so that interference is eliminated as prescribed by (5.10) and (5.11). Thus, α interferes with A and the only way to avoid that interference is to ensure that α is not executed when the control point between *S1* and *S2* is eligible.

Either mutual exclusion or condition synchronization can be used to prevent such undesirable interleavings. Mutual exclusion can be used to eliminate the control point between *S1* and *S2* by constructing a single atomic action,

S1; *S2*.

Alternatively, condition synchronization can be used to delay α until the state is such that executing α could not interfere with A,

if $\neg A \quad wp(\alpha, A) \rightarrow \alpha$ **fi**

where $wp(\alpha, A)$ is the set of states in which executing α leads to termination in a state satisfying A. These two techniques are summarized by:

(6.5) **Eliminating Interference by Mutual Exclusion.** To eliminate interference of assignment action α with an assertion A, include the program text surrounding A in angle brackets. □

(6.6) **Eliminating Interference by Condition Synchronization.** To eliminate interference of assignment action α with an assertion A, construct a conditional atomic action that includes α and delays α when its execution would invalidate A. □

To illustrate these techniques, consider a concurrent program to model a bank. The bank manages a collection of accounts

var $acnt[1\,..\,n]$: *integer*,

supports transactions to transfer money from one account to another, and has an auditor to check for embezzlement.

A transaction to transfer $20 from account *gra* to account *fbs* does not change the total deposits at the bank. Assuming $1 \le gra \le n$, $1 \le fbs \le n$, and that $acnt[gra] \ge 20$, a proof outline for a program to implement such a transfer of funds is given by

Transfer: $\{T=(\Sigma i: 1 \le i \le n: acnt[i]) \quad acnt[gra]=G \quad acnt[fbs]=F\}$
α: $acnt[gra], acnt[fbs] := acnt[gra]-20, acnt[fbs]+20$
$\{T=(\Sigma i: 1 \le i \le n: acnt[i]) \quad acnt[gra]=G-20 \quad acnt[fbs]=F+20\}$

An auditor checks each account, accumulating the bank's total deposits. If this total is not equal to *totdeps*, which we assume is initialized to the sum of the accounts, then funds have been embezzled and boolean program variable *embzl* is set accordingly. This is implemented by:

Auditor: $\{totdeps=T=(\Sigma i: 1 \le i \le n: acnt[i])\}$
$j, cash := 0, 0;$
$\{A1: \ j=0 \quad cash=(\Sigma i: 1 \le i \le j: acnt[i]) \quad totdeps=T\}$
do $j \ne n \ \rightarrow \ \{A2: \ 0 \le j < n \quad cash=(\Sigma i: 1 \le i \le j: acnt[i]) \quad totdeps=T\}$
$cash, j := cash+acnt[j+1], j+1$
$\{A3: \ 0 \le j \le n \quad cash=(\Sigma i: 1 \le i \le j: acnt[i]) \quad totdeps=T\}$
od;
$\{A4: \ cash=(\Sigma i: 1 \le i \le n: acnt[i]) \quad totdeps=T\}$
$embzl := (cash \ne totdeps)$
$\{A5: \ embzl \quad cash \ne T\}$

The proof outlines for *Transfer* and *Auditor* are not interference free: $NI(\alpha, A2)$ and $NI(\alpha, A3)$ are not theorems. The problem is that whenever $gra \le j \le fbs$ or $fbs \le j \le gra$, executing α can change the value of $(\Sigma i: 1 \le i \le j: acnt[i])$ without changing the value of *cash*, thereby invalidating *A2* and/or *A3*.

This interference can be avoided by using mutual exclusion as described by (6.5) to eliminate the control points at *A2* and *A3*. We do this by enclosing the **do** in *Auditor* within angle brackets.

Auditor: $\{totdeps=T=(\Sigma i: 1 \le i \le n: acnt[i])\}$
$j, cash := 0, 0;$
$\{A1: \ j=0 \quad cash=(\Sigma i: 1 \le i \le j: acnt[i]) \quad totdeps=T\}$
do $j \ne n \ \rightarrow \ cash, j := cash+acnt[j+1], j+1$ **od**;
$\{A4: \ cash=(\Sigma i: 1 \le i \le n: acnt[i]) \quad totdeps=T\}$
$embzl := (cash \ne totdeps)$
$\{A5: \ embzl \quad cash \ne T\}$

The result is a proof outline that is interference free and satisfies all the other requirements of **cobegin** Rule (6.4). Unfortunately, this change causes almost all of *Auditor* to be executed without interruption. This can delay execution of *Transfer*, which is undesirable.

Another way to prevent interference of α with *A2* and *A3* is by using condition synchronization, as described by (6.6). We prevent α from executing unless $(gra<j \quad fbs<j) \quad (gra>j \quad fbs>j)$ holds by defining a conditional atomic action:

α': **if** $(gra<j \quad fbs<j) \quad (gra>j \quad fbs>j) \ \rightarrow \ \alpha$ **fi**

Now, α is no longer an assignment action, although α' is. Showing interference freedom requires that $NI(\alpha', A1)$ and $NI(\alpha', A2)$ be theorems; they are. *Transfer* and *Auditor* are interference free and the resulting theorem of Proof Outline Logic, including initialization and declarations, is given in Figure 6.1.

The logic we have presented for reasoning about programs containing **cobegin** statements is not complete. There exist proof outlines that are valid but not provable using Proof Outline Logic. To illustrate the problem, consider a program to increment x by 2.

cobegin α_1: $x := x+1 \ /\!/ \ \alpha_2$: $x := x+1$ **coend** (7.1)

We start by constructing complete valid proof outlines for each process in isolation.

var $acnt[1..n]$: $integer$;
  j, $cash$: $integer$ **initial** 0;
  $totdeps$: $integer$ **initial** $(\Sigma i: 1 \le i \le n: acnt[i])$;
  $embzl$: $boolean$

$\{j=0 \quad totdeps=\mathrm{T}=(\Sigma i: 1 \le i \le n: acnt[i])\}$

cobegin
  Transfer: $\{\mathrm{T}=(\Sigma i: 1 \le i \le n: acnt[i]) \quad acnt[gra]=\mathrm{G} \quad acnt[fbs]=\mathrm{F}\}$
    if $(gra<j \quad fbs<j) \quad (gra>j \quad fbs>j) \rightarrow$
      $acnt[gra], acnt[fbs] := acnt[gra]-20, acnt[fbs]+20$ **fi**
    $\{\mathrm{T}=(\Sigma i: 1 \le i \le n: acnt[i]) \quad acnt[gra]=\mathrm{G}-20 \quad acnt[fbs]=\mathrm{F}+20\}$

  //

  Auditor: $\{j=0 \quad cash=(\Sigma i: 1 \le i \le j: acnt[i]) \quad totdeps=\mathrm{T}\}$
    do $j \ne n \rightarrow \{0 \le j < n \quad cash=(\Sigma i: 1 \le i \le j: acnt[i])$
        land $totdeps=\mathrm{T}\}$
      $cash, j := cash+acnt[j+1], j+1$
      $\{0 \le j \le n \quad cash=(\Sigma i: 1 \le i \le j: acnt[i])$
        land $totdeps=\mathrm{T}\}$
    od;
    $\{cash=(\Sigma i: 1 \le i \le n: acnt[i]) \quad totdeps=\mathrm{T}\}$
    $embzl := (cash \ne totdeps)$
    $\{embzl \quad (cash \ne \mathrm{T})\}$
coend
$\{acnt[gra]=\mathrm{G}-20 \quad acnt[fbs]=\mathrm{F}+20 \quad embzl\,(cash \ne \mathrm{T})\}$

Figure 6.1. Bank Proof Outline

$\{x=\mathrm{X}\} \quad \alpha_1\colon\ x := x+1 \quad \{x=\mathrm{X}+1\}$

$\{x=\mathrm{X}\} \quad \alpha_2\colon\ x := x+1 \quad \{x=\mathrm{X}+1\}$

Combining these yields a complete proof outline for the **cobegin**:

$\{P\colon\ x=\mathrm{X}\}$
cobegin
  $\{P1\colon\ x=\mathrm{X}\} \quad \alpha_1\colon\ x := x+1 \quad \{Q1\colon\ x=\mathrm{X}+1\}$
  //
  $\{P2\colon\ x=\mathrm{X}\} \quad \alpha_2\colon\ x := x+1 \quad \{Q2\colon\ x=\mathrm{X}+1\}$
coend
$\{Q\colon\ x=\mathrm{X}+1\}$

Hypothesis (d) of **cobegin** Rule (6.4) requires that four interference-freedom formulas be proved: $NI(\alpha_1, P2)$, $NI(\alpha_1, Q2)$, $NI(\alpha_2, P1)$, and $NI(\alpha_2, Q1)$.

The first, $NI(\alpha_1, P2)$, is not a theorem. Since there is no way to strengthen $pre(\alpha_1)$ and still have it implied by P, we use (5.11) and weaken $P2$ to $x=\mathrm{X} \quad x=\mathrm{X}+1$. This particular weakening was selected to account for possible execution of α_1 when $x=\mathrm{X}$. We then recompute $Q2$ accordingly. A similar argument suggests that $P1$ and $Q1$ be weakened, resulting in a revised proof outline:

$\{P\colon\ x = \mathrm{X}\}$
cobegin
$\qquad \{P1\colon\ x = \mathrm{X} \quad x = \mathrm{X}{+}1\}$
$\qquad \alpha_1\colon\ x := x + 1$
$\qquad \{Q1\colon\ x = \mathrm{X}{+}1 \quad x = \mathrm{X}{+}2\}$
$/\!/$ (7.2)
$\qquad \{P2\colon\ x = \mathrm{X} \quad x = \mathrm{X}{+}1\}$
$\qquad \alpha_2\colon\ x := x + 1$
$\qquad \{Q2\colon\ x = \mathrm{X}{+}1 \quad x = \mathrm{X}{+}2\}$
coend
$\{Q\colon\ x = \mathrm{X}{+}1 \quad x = \mathrm{X}{+}2\}$

Q is still not as expected. Moreover, there is still interference! Weakening *P1*, *Q1*, *P2*, and *Q2* again results in another proof outline:

$\{P\colon\ x = \mathrm{X}\}$
cobegin
$\qquad \{P1\colon\ x = \mathrm{X} \quad x = \mathrm{X}{+}1 \quad x = \mathrm{X}{+}2\}$
$\qquad \alpha_1\colon\ x := x + 1$
$\qquad \{Q1\colon\ x = \mathrm{X}{+}1 \quad x = \mathrm{X}{+}2 \quad x = \mathrm{X}{+}3\}$
$/\!/$
$\qquad \{P2\colon\ x = \mathrm{X} \quad x = \mathrm{X}{+}1 \quad x = \mathrm{X}{+}2\}$
$\qquad \alpha_2\colon\ x := x + 1$
$\qquad \{Q2\colon\ x = \mathrm{X}{+}1 \quad x = \mathrm{X}{+}2 \quad x = \mathrm{X}{+}3\}$
coend
$\{Q\colon\ x = \mathrm{X}{+}1 \quad x = \mathrm{X}{+}2 \quad x = \mathrm{X}{+}3\}$

Q is still not $x = \mathrm{X}{+}2$ and there is still interference.

By now, it should be clear that additional weakening will not result in an interference-free proof outline. In fact, the strongest theorem of Proof Outline Logic that can be proved about (7.1) is:

$\{P\colon\ x = \mathrm{X}\}$
cobegin
$\qquad \{P1\colon\ x \geq \mathrm{X}\}\ \ x := x + 1\ \ \{Q1\colon\ x > \mathrm{X}\}$
$/\!/$ (7.3)
$\qquad \{P2\colon\ x \geq \mathrm{X}\}\ \ x := x + 1\ \ \{Q2\colon\ x > \mathrm{X}\}$
coend
$\{Q\colon\ x > \mathrm{X}\}$

Clearly, this is not satisfactory.

To understand the problem, return to (7.2) where we were required, but unable, to prove $NI(\alpha_1, P2)$. One might reason as follows that α_1 cannot interfere with *P2*.

> If α_1 is eligible then it cannot have executed. Execution of α_1 interferes with *P2* only if the control point at *P2* is eligible, from which we conclude that α_1 can interfere with *P2* only if α_2 has not yet executed. Thus, α_1 can interfere with *P2* only if $x = \mathrm{X}$. Executing α_1 in a state satisfying $x = \mathrm{X}$ terminates in a state satisfying $x = \mathrm{X}{+}1$. So, α_1 cannot invalidate *P2* of (7.2).

The crux of this operational argument is that in (7.2), $x = \mathrm{X}$ whenever α_1 is eligible and the control point with which *P2* is associated is also eligible. By contrast, the precondition of $NI(\alpha_1, P2)$, which should characterize states in which α_1 can be executed while the control point with which *P2* is associated is eligible, is $x = \mathrm{X} \quad x = \mathrm{X}{+}1$, and this is too weak due to the second disjunct. We must strengthen $pre(NI(\alpha_1, P2))$.

Reasoning about (7.1) requires information about the values of program counters. In retrospect, it should not be surprising that program counters are required for reasoning about some programs, since the program counter is an integral part of the program state—it determines what can be executed next.

We use the term *hidden state* for that portion of the program state not stored in program variables. The hidden state of a concurrent program includes the program counters of its processes. It can include other information, as well. For example, when message passing is used for communication, the hidden state includes information about messages that have been sent but not yet received, since that information is not stored in the program variable of any process, yet effects execution of the receiver.

The question, then, is how information about the hidden state can be included in assertions. One obvious approach is to define some predicates for this purpose. There are difficulties with this approach. First, inference rules must be devised for reasoning about assertions containing such predicates. Second, Assertion Restriction (5.6a) would no longer hold. Interference Freedom (5.7) would then require that $NI(\alpha, A)$ be proved for every text α between assertions in a proof outline, instead of just for assignment actions. Since $NI(\alpha, A)$ might not even be a formula of Proof Outline Logic—for example, if α were a fragment of a statement, such as a guard in a **do**—we would then have to extend Proof Outline Logic.

Another approach for including hidden-state information in assertions is based on auxiliary variables. An *auxiliary variable* is one that is added to a program solely for use in the assertions of a proof outline. The value of an auxiliary variable is altered by statements that are added to the program so that the auxiliary variable reflects the hidden state. This approach does not suffer the disadvantages of the previous one. Proof Outline Logic can be used to reason about the values of auxiliary variables, hence additional inference rules are not required to reason about the hidden state. And, since the value of an auxiliary variable can be changed only by executing an assignment statement, Assertion Restriction (5.6) remains satisfied. Therefore, the interference freedom obligations are unaltered.

It is important to be able to distinguish the auxiliary variables in a proof outline from the program variables. In these notes, names of auxiliary variables start with an upper-case letter and are typeset in italics; this distinguishes them from program variables, which always start with a lower-case letter, and from logical variables, which are typeset in roman.

Unrestricted use of auxiliary variables in proof outlines would destroy the soundness of Proof Outline Logic. Although we might use auxiliary variables when constructing a proof, a program is executed without the auxiliary variables present and therefore auxiliary variables must not influence the behavior of the program in which they are embedded. To ensure this, we require

(7.4) **Auxiliary Variable Restriction.** Auxiliary variables appear only in assignment statements $\bar{x} := \bar{e}$ where if the i^{th} expression in $\bar{e}$ references an auxiliary variable, then the i^{th} target in $\bar{x}$ is an auxiliary variable. □

This prevents program variables from obtaining their values from auxiliary variables and therefore prevents auxiliary variables from influencing execution. Note that (7.4) is not really a restriction, since we use auxiliary variables only to make hidden-state information visible in assertions.

Since auxiliary variables are added to a program solely for purposes of constructing a proof outline, we must be able to delete them when extracting the program that was the subject of the proof. The following inference rule of Proof Outline Logic permits this.

(7.5) *Auxiliary Variable Deletion Rule*: Let AV be a set of auxiliary variables in annotated program $\hat{S}$ and let $\hat{S}\,|_{AV}$ be the annotated program that results when all assignments to variables in AV are deleted from S. If P, Q, and the assertions in $\hat{S}$ do not mention any variable in AV, then

$$\frac{\{P\}\ \hat{S}\ \{Q\}}{\{P\}\ \ \hat{S}\,|_{AV}\ \ \{Q\}}$$

The inference rule does not permit deletion of an auxiliary variable mentioned in an assertion because it would then be possible to infer the invalid proof outline

$\{Aux{=}0\}$ **skip** $\{Aux{=}1\}$

from the theorem

$\{Aux{=}0\}\ \ Aux := 1\ \ \{Aux{=}1\}$.

Using auxiliary variables, it is a simple matter to prove that (7.1) increments x by 2. We define auxiliary variables *Done1* and *Done2* and modify (7.1) so that *Done1* (*Done2*) is *true* only after the first (second) process has incremented x.

$$
\begin{array}{ll}
& \{x = \mathrm{X}\} \\
& \mathit{Done1},\ \mathit{Done2} := \mathit{false},\ \mathit{false}; \\
& \{P\colon\ x = \mathrm{X}\ \ \neg \mathit{Done1}\ \ \neg \mathit{Done2}\} \\
& \textbf{cobegin} \\
& \qquad \{P1\colon\ (\neg \mathit{Done2}\ x = \mathrm{X})\ \ (\mathit{Done2}\ x = \mathrm{X}{+}1)\ \ \neg \mathit{Done1}\} \\
& \ \ \alpha_1\colon\ x,\ \mathit{Done1} := x+1,\ \mathit{true} \\
& \qquad \{Q1\colon\ (\neg \mathit{Done2}\ x = \mathrm{X}{+}1)\ \ (\mathit{Done2}\ x = \mathrm{X}{+}2)\ \ \mathit{Done1}\} \qquad\qquad\qquad\qquad (7.6) \\
& /\!/ \\
& \qquad \{P2\colon\ (\neg \mathit{Done1}\ x = \mathrm{X})\ \ (\mathit{Done1}\ x = \mathrm{X}{+}1)\ \ \neg \mathit{Done2}\} \\
& \ \ \alpha_2\colon\ x,\ \mathit{Done2} := x+1,\ \mathit{true} \\
& \qquad \{Q2\colon\ (\neg \mathit{Done1}\ x = \mathrm{X}{+}1)\ \ (\mathit{Done1}\ x = \mathrm{X}{+}2)\ \ \mathit{Done2}\} \\
& \textbf{coend} \\
& \{Q\colon\ x = \mathrm{X}{+}2\}
\end{array}
$$

We now establish that this proof outline is a theorem. The proof outline

$$\{x = \mathrm{X}\}\ \ \mathit{Done1},\ \mathit{Done2} := \mathit{false},\ \mathit{false}\ \ \{P\}$$

is a theorem due to Assignment Axiom (3.3). Both $P\ (P1\ \ P2)$ and $(Q1\ \ Q2)\ Q$ are valid as required by hypotheses (b) and (c) of **cobegin** Rule (6.4). Hypothesis (a) requires that the proof outline for each process in isolation be a theorem. This is easily established using Assignment Axiom (3.3) and then Rule of Consequence (3.7) and Deletion Rule (3.9). Finally, to satisfy hypothesis (d), we must show interference freedom. Expanding and simplifying $NI(\alpha_1, P2)$ yields,

$$
\begin{array}{l}
\{x = \mathrm{X}\ \ \neg \mathit{Done1}\ \ \neg \mathit{Done2}\} \\
x,\ \mathit{Done1} := x+1,\ \mathit{true} \\
\{(\neg \mathit{Done1}\ x = \mathrm{X})\ \ (\mathit{Done1}\ x = \mathrm{X}{+}1)\ \ \neg \mathit{Done2}\}
\end{array}
$$

which is a theorem. $NI(\alpha_1, Q2)$, $NI(\alpha_2, P1)$, and $NI(\alpha_2, Q1)$ are also theorems. Thus, the proof is completed.

Checking interference freedom involves proving that a collection of proof outlines are theorems. This can be a tedious task. For example, establishing interference freedom for

$$\{P\}\ \ \textbf{cobegin}\ \ PO_1\ /\!/\ PO_2\ /\!/\ \cdots\ /\!/\ PO_n\ \ \textbf{coend}\ \ \{Q\}$$

requires that

$$\sum_{i=1,n}\ \sum_{j=1,n} \mathit{assig}(PO_i) \times \mathit{assert}(PO_j)$$

theorems be proved, where $\mathit{assig}(PO)$ is the number of assignment actions in PO and $\mathit{assert}(PO)$ is the number of assertions in PO. The value of this expression grows rapidly with the number and size of processes.

Fortunately, there is an easier way to prove interference freedom. The technique is based on using a *proof outline invariant*—an assertion that is *true* initially and remains *true* throughout execution of a program.

(8.1) **Proof Outline Invariant.** An assertion I is an invariant of a proof outline PO provided

(a) $\mathit{pre}(PO)\ \ I$, and

(b) for every assignment action α in PO: $\{\mathit{pre}_{PO}(\alpha)\ \ I\}\ \alpha\ \{I\}$. □

Observe that a proof outline invariant can be conjoined to every assertion in a proof outline without creating interference.

Suppose every assertion A in the proof outline for a concurrent program is of the form $I\ \ L_A$, where all variables mentioned in L_A are altered only by the process containing A. Moreover, suppose we have established that the proof outline for each process (in isolation) is a theorem of Proof Outline

Logic and that I is an invariant of each. Let α be an assignment action in one process. Thus, we have

$$\{I \quad L_{pre(\alpha)}\} \;\; \alpha \;\; \{I \quad L_{post(\alpha)}\}. \tag{8.2}$$

Let A be an assertion in a different process from the one containing α. By assumption, A can be partitioned into the conjunction of invariant I and an assertion L_A involving only variables altered by the process containing A. Moreover, no variable in L_A affected by executing α. Therefore,

$$\{L_A\} \;\; \alpha \;\; \{L_A\} \tag{8.3}$$

is valid and, due to the relative completeness of Proof Outline Logic, is a theorem.

Proving $NI(\alpha, A)$ is equivalent to proving $NI(\alpha, I \;\; L_A)$,

$$\{I \quad L_{pre(\alpha)} \quad I \quad L_A\} \;\; \alpha \;\; \{I \quad L_A\}. \tag{8.4}$$

However, (8.4) can be inferred directly from (8.2) and (8.3) using Conjunction Rule (3.10) and then Rule of Consequence (3.7) (and Deletion Rule (3.9)) to weaken the postcondition. Thus, there is no need to prove Proof Outline Logic theorems when establishing interference freedom if shared variables are related by an invariant. This allows **cobegin** Rule (6.4) to be simplified by changing hypothesis (d).

(8.5) **cobegin** *Rule*:

(a) For all i, $1 \le i \le n$: PO_i is complete and *turnstile* PO_i,
(b) $P \quad (pre(PO_1) \quad pre(PO_2) \quad \cdots \quad pre(PO_n))$,
(c) $(post(PO_1) \quad post(PO_2) \quad \cdots \quad post(PO_n)) \quad Q$,
(d) For all i, $1 \le i \le n$, every assertion A in PO_i is of the form
$I \quad L_A$, where
i. L_A mentions variables changed only in process i
ii. I is an invariant of PO_i

$\{P\}$ **cobegin** $PO_1 \;/\!/\; PO_2 \;/\!/\; \cdots \;/\!/\; PO_n$ **coend** $\{Q\}$

When (8.5) is used, establishing interference freedom is trivial. However, structuring an assertion A in terms of an invariant I and a local part L_A can lead to somewhat longer and more complex assertions than might otherwise be required. This, in turn, results in more complex sequential proofs for hypothesis (a) of the new **cobegin** Rule.

To illustrate **cobegin** Rule (8.5), we return to program (7.1) to increment x by 2. During execution of (7.1), x contains the sum of three quantities: its initial value, the amount it has been incremented by the first process, and the amount it has been incremented by the second process. This can be formalized as an assertion,

$$I: \; x = \mathrm{X} + Z1 + Z2$$

where logical variable X is the initial value of x, auxiliary variable $Z1$ is the amount x has been incremented by the first process, and auxiliary variable $Z2$ is the amount x has been incremented by the second process. We now endeavor to construct a proof outline with I as an invariant.

Initially, neither process has incremented x, so we should have $Z1 = Z2 = 0$. This is easily established by the assignment statement $Z1, Z2 := 0, 0$ which we can add to the program because $Z1$ and $Z2$ are auxiliary variables. Next, we ensure that $Z1$ is incremented whenever the first process increments x and that $Z2$ is incremented whenever the second process increments x. This is achieved by changing α_1 (α_2) into an assignment that increments $Z1$ ($Z2$) as x is incremented. A proof outline for the resulting program follows. Notice that assertions in the first process are structured as a conjunction of a proof outline invariant I and an assertion about $Z1$ and that $Z1$ is changed only in the first process. Assertions in the second process are structured similarly.

$\{P\!:\ x = \mathrm{X}\}$
$Z1,\ Z2 := 0,\ 0;$
$\{P'\!:\ I \quad Z1 = Z2 = 0\}$
cobegin
  $\{I \quad Z1{=}0\} \quad \alpha_1\!:\ x,\ Z1 := x+1,\ Z1{+}1 \quad \{I \quad Z1{=}1\}$
 //
  $\{I \quad Z2{=}0\} \quad \alpha_2\!:\ x,\ Z2 := x+1,\ Z2{+}1 \quad \{I \quad Z2{=}1\}$
coend
$\{Q'\!:\ I \quad Z1 = Z2 = 1\}$
$\{Q\!:\ x = \mathrm{X}{+}2\}$

Using **cobegin** Rule (8.5), it is easy to establish that this proof outline is a theorem.

This example also illustrates how auxiliary variables can encode hidden-state information in a way that simplifies rather than complicates construction of a proof outline. Auxiliary integer variables *Z1* and *Z2* encode the values of program counters for the processes in (7.1). In §7.2, auxiliary boolean variables *Done1* and *Done2* served the same purpose:

$Done1 \quad Z1{=}1 \qquad \neg Done1 \quad Z1{=}0$

$Done2 \quad Z2{=}1 \qquad \neg Done2 \quad Z2{=}0.$

However, encoding the information in *Z1* and *Z2* permitted formulation of a simple invariant that related the value of x and the hidden state. It encoded just the right amount of information in just the right way.

A safety property stipulates that some "bad thing" does not happen during execution. In addition to partial correctness, important safety properties of concurrent programs include mutual exclusion and deadlock freedom. In *mutual exclusion* the "bad thing" is more than one process executing designated program segments, called *critical sections*, at the same time. In *deadlock freedom* it is deadlock, a state where some subset of the processes are delayed awaiting conditions that will never occur.

Without loss of generality, we need only consider safety properties of the form

$SP\!:\ P_{Bad}$ never holds during execution.

where P_{Bad} is a predicate on the program state. This is because any action a program performs must be based on its state. (Even actions performed in response to reading input can be viewed as being based on the state, since the sequence of input values available during a particular execution can be thought of as part of the initial state of the program.) So, showing that a state satisfying some P_{Bad} never arises during execution is equivalent to showing that a "bad thing" cannot happen.

In order to prove that SP holds for a program S, we must show that for each possible execution, P_{Bad} is not *true* initially, nor after the first atomic action, nor after the second, and so on. In short, we show that P_{Bad} is *true* of no intermediate state of S. Since the assertions in a complete valid proof outline for a program characterize possible intermediate states of that program, SP can be proved by constructing a complete and valid proof outline for S and then checking that each assertion in that proof outline implies $\neg P_{Bad}$. Theorems of Proof Outline Logic are valid proof outlines, so we have the following method for proving that a program satisfies a given safety property.

(9.1) **Proving a Safety Property.** To prove that SP holds for a program S, construct a theorem PO of Proof Outline Logic and show that for every assertion A in PO, $A \quad \neg P_{Bad}$. □

Since, by definition, an invariant holds on every state of a program, we also have the following method, which is sometimes simpler to apply.

(9.2) **Proving a Safety Property using an Invariant.** To prove that SP holds for a program S, construct a theorem PO of Proof Outline Logic with invariant I such that $I \quad \neg P_{Bad}$. □

We illustrate the techniques for proving safety properties by investigating a protocol that solves a classic concurrent programming exercise—the critical section problem. Consider a concurrent program with two processes, each of which repeatedly executes in a critical section and then a non-critical section. Desired is a protocol satisfying three properties.

(9.3) *Mutual Exclusion*. At most one process is executing in its critical section at a time.

(9.4) *Non Blocking*. A process executing in its non-critical section does not prevent another from entering its critical section.

(9.5) *Deadlock Freedom*. If multiple processes attempt entry to their critical sections at the same time, then one will gain entry.

If angle brackets can be used to make arbitrary statements atomic, devising a protocol that satisfies (9.3)−(9.5) is trivial. In the solution of Figure 9.1, angle brackets are used in a disciplined manner that makes the program amenable to translation into actual machine instructions.

To establish that this proof outline is a theorem of Proof Outline Logic, we use **cobegin** Rule (6.4). Hypothesis (a) is easily demonstrated. Hypothesis (b) is satisfied, since $P \;\; (P1 \;\; Q1)$. Hypothesis (c) is satisfied, since $(P9 \;\; Q9) \;\; Q$. Finally, hypothesis (d) is proved as follows. All assertions in the proof outline of the first process except *P5* and *P6* mention variables that are changed only by the first process. Thus, no assignment statement in the second process can interfere with those assertions. The only assignments in the second process that can interfere with *P5* or *P6* are *T1*, *T2*, and *T4*. Thus, since *P5* and *P6* are identical, we must prove:

$$
\begin{array}{ll}
NI(T1, P5): & \{\neg enter2 \;\; enter1 \;\; \neg AtP3 \\
 & \qquad (\neg enter2 \;\; turn{=}1 \;\; AtQ3)\} \\
 & AtQ3,\, enter2 := true,\, true \\
 & \{enter1 \;\; \neg AtP3 \;\; (\neg enter2 \;\; turn{=}1 \;\; AtQ3)\}
\end{array}
$$

$$
\begin{array}{ll}
NI(T2, P5): & \{enter2 \;\; enter1 \;\; \neg AtP3 \\
 & \qquad (\neg enter2 \;\; turn{=}1 \;\; AtQ3)\} \\
 & AtQ3,\, turn := false,\, 1 \\
 & \{enter1 \;\; \neg AtP3 \;\; (\neg enter2 \;\; turn{=}1 \;\; AtQ3)\}
\end{array}
$$

$$
\begin{array}{ll}
NI(T4, P5): & \{enter2 \;\; \neg AtQ3 \;\; (\neg enter1 \;\; turn{=}2 \;\; AtP3) \\
 & \qquad enter1 \;\; \neg AtP3 \;\; (\neg enter2 \;\; turn{=}1 \;\; AtQ3)\} \\
 & enter2 := false \\
 & \{enter1 \;\; \neg AtP3 \;\; (\neg enter2 \;\; turn{=}1 \;\; AtQ3\}
\end{array}
$$

Each of these proof outlines is a theorem, so we can infer that the second process does not interfere with the first. Similar arguments establish that the first process does not interfere with the second, hence the proof outline of Figure 9.1 is a theorem of Proof Outline Logic.

Mutual Exclusion (9.3) involves the hidden state. Thus, auxiliary variables are required in order to put (9.3) in the same form as *SP*.

In_1 ≡ the first process is executing in its critical section.

In_2 ≡ the second process is executing in its critical section.

By using $In\_1 \;\; In\_2$ as P_{Bad}, (9.3) can be put in the same form as *SP*. Figure 9.2 is a proof outline for the program of Figure 9.1 with assignments to auxiliary variables *In_1* and *In_2* included. It is a theorem of Proof Outline Logic; the proof is similar to the one used for the proof outline of Figure 9.1.

To establish that (9.3) holds, first note that

$$I1\!: \; \neg In\_1 \;\; (enter1 \;\; \neg AtP3 \;\; (\neg enter2 \;\; turn{=}1 \;\; AtQ3))$$

$$I2\!: \; \neg In\_2 \;\; (enter2 \;\; \neg AtQ3 \;\; (\neg enter1 \;\; turn{=}2 \;\; AtP3))$$

are invariants of the proof outline of Figure 9.2. Since $I1 \;\; I2 \; \neg(In\_1 \;\; In\_2)$, we can construct a proof outline invariant $I1 \;\; I2$ and use Proving a Safety Property using an Invariant (9.2) to conclude that Mutual Exclusion (9.3) holds.

```
var enter1, enter2 : boolean  initial false, false;
      AtP3, AtQ3 : boolean ;
      turn : integer  initial 1
{P:  ¬enter1  ¬enter2}
cobegin
    {P1:  ¬enter1}
    do true →  {P2:  ¬enter1}
              S1:  AtP3, enter1 := true, true;
                   {P3:  enter1}
              S2:  AtP3, turn := false, 2;
                   {P4:  enter1  ¬AtP3}
              S3:  if ¬enter2  turn=1  →  skip fi
                   {P5:  enter1  ¬AtP3  (¬enter2  turn=1  AtQ3)}
                   ... Critical Section ...
                   {P6:  enter1  ¬AtP3  (¬enter2  turn=1  AtQ3)}
              S4:  enter1 := false;
                   {P7:  ¬enter1}
                   ... Non-critical Section ...
                   {P8:  ¬enter1}
    od    {P9:  false}
  //
    {Q1:  ¬enter2}
    do true →  {Q2:  ¬enter2}
              T1:  AtQ3, enter2 := true, true;
                   {Q3:  enter2}
              T2:  AtQ3, turn := false, 1;
                   {Q4:  enter2  ¬AtQ3}
              T3:  if ¬enter1  turn=2  →  skip fi
                   {Q5:  enter2  ¬AtQ3  (¬enter1  turn=2  AtP3)}
                   ... Critical Section ...
                   {Q6:  enter2  ¬AtQ3  (¬enter1  turn=2  AtP3)}
              T4:  enter2 := false;
                   {Q7:  ¬enter2}
                   ... Non-critical Section ...
                   {Q8:  ¬enter2}

    od    {Q9:  false}
coend
{Q:  false}
```

Figure 9.1. Mutual Exclusion Protocol

```
var enter1, enter2 : boolean initial false, false;
    AtP3, AtQ3 : boolean;
    turn : integer initial 1;
    In_1, In_2 : boolean initial false, false
{P:  ¬enter1  ¬enter2  ¬In_1  ¬In_2}
cobegin
   {P1:  ¬enter1  ¬In_1}
   do true →  {P2:  ¬enter1  ¬In_1}
          S1:  AtP3, enter1 := true, true;
               {P3:  enter1  ¬In_1}
          S2:  AtP3, turn := false, 2;
               {P4:  enter1  ¬AtP3  ¬In_1}
          S3:  if ¬enter2  turn=1  →  In_1 := true fi
               {P5:  enter1  ¬AtP3  (¬enter2  turn=1  AtQ3)  In_1}
               ... Critical Section ...
               {P6:  enter1  ¬AtP3  (¬enter2  turn=1  AtQ3)  In_1}
          S4:  enter1, In_1 := false, false;
               {P7:  ¬enter1  ¬In_1}
               ... Non-critical Section ...
               {P8:  ¬enter1  ¬In_1}
   od    {P9:  false}
 //
   {Q1:  ¬enter2  ¬In_2}
   do true →  {Q2:  ¬enter2  ¬In_2}
          T1:  AtQ3, enter2 := true, true;
               {Q3:  enter2  ¬In_2}
          T2:  AtQ3, turn := false, 1;
               {Q4:  enter2  ¬AtQ3  ¬In_2}
          T3:  if ¬enter1  turn=2  →  In_2 := true fi
               {Q5:  enter2  ¬AtQ3  (¬enter1  turn=2  AtP3)  In_2}
               ... Critical Section ...
               {Q6:  enter2  ¬AtQ3  (¬enter1  turn=2  AtP3)  In_2}
          T4:  enter2, In_2 := false, false;
               {Q7:  ¬enter2  ¬In_2}
               ... Non-critical Section ...
               {Q8:  ¬enter2  ¬In_2}
   od    {Q9:  false}
coend
{Q:  false}
```

Figure 9.2. After Adding Auxiliary Variables

We now turn to Non Blocking (9.4). Here, the proscribed "bad thing" is a state where one process is executing in its non-critical section and the other is blocked attempting to enter its critical section. Whenever the first process is in its non-critical section, *¬enter1 ¬In_1* holds; whenever the second process is blocked attempting to enter its critical section, the guard of *T3* is *false*, so $\neg(\neg enter1 \ \ turn{=}2)$ holds. Thus, whenever the first process is in its non-critical section and the second process is blocked attempting to enter its critical section,

$$\neg enter1 \ \ \neg In\_1 \ \ \neg(\neg enter1 \ \ turn{=}2)$$

holds. Similarly, if the second process is in its non-critical section and the first is blocked attempting to enter its critical section

$$\neg enter2 \ \ \neg In\_2 \ \ \neg(\neg enter2 \ \ turn{=}1)$$

will hold. To put (9.4) in the same form as SP, we choose for P_{Bad}:

$$\begin{aligned}&(\neg enter1 \ \ \neg In\_1 \ \neg(\neg enter1 \ \ turn{=}2))\\ &\quad(\neg enter2 \ \ \neg In\_2 \ \ \neg(\neg enter2 \ \ turn{=}1))\\ &= false\end{aligned}$$

According to Proving a Safety Property (9.1), we can prove (9.4) by showing that $\neg P_{Bad}$ is implied by every assertion in the proof outline of Figure 9.2. Every assertion implies $\neg$*false*, so we are finished.

Finally, Deadlock Freedom (9.5) requires showing that it is not possible for both processes to be waiting to enter their critical sections. If both processes are waiting, it is because the guards of *S3* and *T3* are *false*. This state is characterized by

$$pre(S3) \ \ \neg(\neg enter2 \ \ turn{=}1) \ \ pre(T3) \ \ \neg(\neg enter1 \ \ turn{=}2) \qquad (9.6)$$

and is P_{Bad} for (9.5). Simplifying, we have (9.6)=*false* because *turn* cannot be both 1 and 2 at the same time. Clearly, $\neg$*false* is implied by every assertion in the proof outline of Figure 9.2, so according to Proving a Safety Property (9.1) the proof for Deadlock Freedom (9.5) is completed.

Most safety properties that arise in practice can be formulated in terms of restrictions on states processes should not occupy simultaneously. If assertion P describes the state of one process at some instant and Q describes the state of another process at that instant, then $P \ \ Q$ describes the state at that instant of the concurrent program containing both processes. The constant *false* is satisfied by no state, so if $P \ \ Q = false$, then it is not possible for one process to be in a state satisfying P while the other is in a state satisfying by Q.

(9.7) **Exclusion of Configurations.** To establish that one process cannot be in a state satisfying an assertion P while another process is in a state satisfying an assertion Q, show that $(P \ \ Q){=}false$. □

The technique can be justified formally, as follows. Proving exclusion of configurations is equivalent to proving SP, where P_{Bad} is $P \ \ Q$. According to (9.1), to show that a program S satisfies SP we establish that $\neg P_{Bad}$ is implied by every assertion in a valid proof outline PO for S. Any assertion implies *true*, so when $\neg(P \ \ Q){=}\neg P_{Bad}{=}true$—as required by (9.7)—(9.1) follows trivially.

Exclusion of Configurations (9.7) can be used to prove Mutual Exclusion (9.3), Non Blocking (9.4), and Deadlock Freedom (9.5) for the protocol of Figure 9.1. Mutual Exclusion (9.3) is proved by observing that whenever the first process is executing in its critical section, *P5* holds; and whenever the second process is in its critical section *Q5* holds. Thus, we can use Exclusion of Configurations (9.7) to prove (9.3) by showing $P5 \ \ Q5{=}false$. Substituting and simplifying from Figure 9.1 (since we have no need for the auxiliary variables of Figure 9.2),

$$\begin{aligned}P5 \ \ Q5 &= enter1 \ \ \neg AtP3 \ \ (\neg enter2 \ \ turn{=}1 \ \ AtQ3)\\ &\quad\ land \ \ enter2 \ \ \neg AtQ3 \ \ (\neg enter1 \ \ turn{=}2 \ \ AtP3)\\ &= (turn{=}1 \ \ turn{=}2)\\ &= false.\end{aligned}$$

To establish Non Blocking (9.4) by using Exclusion of Configurations (9.7), we first show that $\neg enter1$, which holds whenever the first process is in its non-critical section, and $Q4\ \ \neg(\neg enter1\ \ turn{=}2)$, which holds whenever the second process is delayed from entering its critical section, together imply *false*. Expanding, we get

$$\neg enter1\ \ enter2\ \ \neg AtQ3\ \ \neg(\neg enter1\ \ turn{=}2)$$

$$= \neg enter1\ \ enter2\ \ \neg AtQ3\ \ enter1\ \ turn{=}2$$

$$= false.$$

A similar argument shows that whenever the second process is in its non-critical section, the first process cannot be delayed from entering its critical section, and (9.4) follows.

Finally, to show Deadlock Freedom (9.5) by using Exclusion of Configurations (9.7), it suffices to show that both processes cannot simultaneously be waiting to enter their critical sections. When the first process is waiting, $P4\ \ \neg(\neg enter2\ \ turn{=}1)$ is *true*; when the second is waiting, $Q4\ \ \neg(\neg enter1\ \ turn{=}2)$ is *true*. Since

$$P4\ \ \neg(\neg enter2\ \ turn{=}1)\ \ Q4\ \ \neg(\neg enter1\ \ turn{=}2)$$

$$= \ enter1\ \ \neg AtP3\ \ enter2\ \ turn{=}1$$
$$land\ \ enter2\ \ \neg AtQ3\ \ enter1\ \ turn{=}2$$

$$= \ false$$

(9.5) holds.

Hoare was the first to propose a logic for reasoning about partial correctness [Hoare 69]. His logic is based on a program verification technique described in [Floyd 67]. Floyd associates a predicate with each arc in a flowchart such that if execution is started on an arc with the corresponding predicate *true* then as each subsequent arc is traversed the associated predicate will be *true*. Floyd credits Perlis and Gorn for the idea, mentioning an unpublished paper by Gorn as its earliest appearance. A similar approach was independently developed by Naur [Naur 66]. There, predicates called *general snapshots* are interspersed in the program text in a way that satisfies Interpretation for Proof Outlines (3.1). Other early investigations into formal systems for proving things about programs are reported in [Yanov 58], [Igarashi 64], and [de Bakker 68] for proving the equivalence of programs, and in [McCarthy 62] and [Burstall 68] for programs specified as recursive functions. Program verification is almost as old as programming itself, however. Early techniques are given in [Goldstine & von Neumann 47] and [Turing 49]; the Turing paper is reprinted and discussed in [Morris & Jones 84]

Formulas of the logical system in [Hoare 69] are of the form $P\ \{S\}\ Q$. It is a logic for reasoning about triples rather than proof outlines. For this reason, the logic does not require an inference rule like our Deletion Rule (3.9). In addition, the logic differs from Proof Outline Logic by not having rules analogous to our Logical Variable Rule (3.8), Conjunction Rule (3.10), or Disjunction Rule (3.11). Finally, the programming language axiomatized in [Hoare 69] contains a single-assignment statement; Assignment Axiom (3.3) for multiple-assignment statements is defined in [Gries 78].

Although many who have written about programming logics use proof outlines, few have formalized them and even fewer have done so correctly. One of the earlier (correct) formalizations appears in [Ashcroft 76]; a natural deduction programming logic of proof outlines is presented in [Constable & O'Donnell 78]. Proof Outline Logic is a straightforward generalization of the logic in [Hoare 69], resulting in a Hilbert-style logic in which formulas are (valid) proof outlines. In the logic of [Hoare 69], the conclusion of each inference rule is constructed by combining fragments of its hypothesis. Hoare's logic deletes the pre- and postconditions of the hypothesis, Proof Outline Logic makes them assertions in an annotated program. Our Simple Proof Outline Validity Test (3.12) has long been a folk-theorem among those who practice program verification.

The first assertional method for proving properties of concurrent programs was described in [Ashcroft & Manna 71]. It is based on converting the flowchart representation of a concurrent program into a non-deterministic, sequential one to which known techniques could then be applied. The method is not practical because the non-deterministic program could be large and awkward. A second assertional verification method based on transforming the flowchart representation of a concurrent program is described in [Levitt 72]. There, flowcharts for processes that synchronize using semaphores are combined by adding the flow of control implied by process switches at semaphore operations. An extension of Floyd's method allows verification conditions to be obtained from such a flowchart.

Subsequently, Ashcroft developed an approach for extracting verification conditions directly from the flowchart of a concurrent program [Ashcroft 75]. Ashcroft associated an assertion with each control point in the program by defining an assertion for each edge in the flowchart, just as Floyd had proposed for sequential programs. By including the program counter in the state, it is possible to define an invariant equivalent to our (5.2). (In Proof Outline Logic, the assertions in a valid complete proof outline define such an invariant.) To show that (5.2) is preserved by execution, Ashcroft required that each atomic action, if started in a state with (5.2) *true*, leave it *true*. Thus, a concurrent program was decomposed into its atomic actions, and each atomic action was shown not to interfere with (5.2).

Not surprisingly, Hoare was the first to address the design of a programming logic for concurrent programs. In [Hoare 72], proof rules for parallel composition of processes that synchronize using conditional critical regions are given. The proof rules extend Hoare's partial correctness logic to concurrent programs, but are inadequate for proving programs in which processes communicate because assertions appearing in the proof of one process are not allowed to mention variables local to another and an invariant can mention only specially designated shared variables. Some of these restrictions are relaxed in [Hoare 75], but the proof system is still not complete and process interaction is limited to sequences, similar to unbounded message queues.

Interference freedom and the first complete programming logic for partial correctness was developed by Owicki in a Ph.D. thesis [Owicki 75] supervised by Gries [Owicki & Gries 76]. The work extends Hoare's logic of triples to handle concurrent programs that synchronize and communicate using shared variables. Our Proof Outline Logic rules for reasoning about **cobegin** are based on rules in [Owicki & Gries 76], although the explanation of **cobegin** Rule (5.8) parallels that in [Dijkstra 76]. One significant difference between Proof Outline Logic and the Owicki-Gries logic concerns the role of proof outlines. The Owicki-Gries logic appears to be based on triples rather than proof outlines, but this is deceptive—had the logic been formalized, the need for treating proof outlines as formulas would probably have become apparent.

Auxiliary variables were first introduced in [Clint 73] to facilitate partial correctness proofs of programs using coroutines and were later used in [Hoare 75]. Owicki was the first to formalize inference rules to delete them from a proof.

The Owicki-Gries work addressed only three types of properties: partial correctness, mutual exclusion, and deadlock freedom. Lamport, working independently, developed an idea similar to interference freedom (monotone assertions) as part of a more general method for proving both safety and liveness properties of concurrent programs [Lamport 77]. (In fact, the terms safety and liveness originated in [Lamport 77], but were only recently formalized [Lamport 85] [Alpern & Schneider 86].) Both (9.1) and (9.2) for proving a safety property are based on Lamport's method.

Lamport then went on to develop Generalized Hoare Logic (GHL) to permit arbitrary safety properties to be verified using a Hoare-style programming logic [Lamport 80]. GHL Formulas resemble Hoare's triples, but have a very different interpretation. A GHL formula describes a program in terms of an invariant, rather than as a relation between a precondition and postcondition. (Hoare's triples are a special case of GHL formulas.) GHL permits reasoning about concurrent programs with arbitrary atomic actions as well as programs for which the atomic actions are not known but the invariants they maintain are. Also, GHL does not use auxiliary variables; instead, it uses predicates $at(S)$, $after(S)$ and $in(S)$ on the hidden state. The relationship between GHL, Floyd's method, and the Owicki-Gries method is explained in [Lamport & Schneider 84].

The angle bracket notation for specifying synchronization was invented by Lamport and formalized in [Lamport 80], but popularized by Dijkstra. (The earliest published use is [Dijkstra 77].) The various scheduling policies were first defined and formalized in [Lehman et al. 81], though with somewhat different terminology—*impartial* for what we call unconditionally fair, *just* for what we call weakly fair, and *fair* for what we call strongly fair. Our terminology is taken from [Francez 86]. Condition synchronization and mutual exclusion and their definitions in terms of removing interference were first described in [Andrews & Schneider 83].

The Bank Example of §6 first appeared in [Lamport 76]. The solution to the critical section problem in §9 was proposed by Peterson [Peterson 81], although our assertional correctness proof is based on [Dijkstra 81].

Concurrent programming has only recently emerged as a discipline. Recent surveys include [Brinch Hansen 73], which illustrates the use of various synchronization primitives by giving solutions to some standard concurrent programming problems, and [Andrews & Schneider 83], which describes a variety of language notations. A number of textbooks have been written about various facets of concurrent programming. Brinch Hansen wrote the first [Brinch Hansen 77]; it describes the design of three operating systems and contains Concurrent Pascal programs for them. [Holt et al. 78] and its successor [Holt 83] covers those aspects of concurrent programming most closely related to operating systems. Another undergraduate text [Ben-Ari 82] covers important synchronization mechanisms and shows how to construct informal correctness proofs for concurrent programs. In [Fillman & Friedman 84], models, languages, and heuristics for concurrent programming are treated. Advanced texts that discuss logics and programming methodology for concurrent programs include [Barringer 85], [Halpern 82], [Hoare 85], and [Paul & Siegert 85].

D. Gries, E Hartikainen, and P. Panangaden provided helpful comments on earlier drafts of this material. A. Demers was an indispensable source of information about logics.

Alpern, B., and F.B. Schneider. Defining liveness. *Information Processing Letters 21*, (Oct. 1985), 181-185. Andrews, G. and F.B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys 15*, 1 (March 1983), 3-43. Ashcroft, E. Proving Assertions about Parallel Programs. *Journal of Computer and System Sciences 10*, 1 (Feb. 1975), 110-135. Ashcroft, E. Program verification tableaus. Technical Report CS-76-01. University of Waterloo, Waterloo, Ontario, Canada, Jan. 1976. Ashcroft, E., and Z. Manna. Formalization of properties of parallel programs. *Machine Intelligence 6*, (1971), Edinburgh University Press, 17-41. Barringer, Howard. *A Survey of Verification Techniques for Parallel Programs*. Lecture Notes in Computer Science Volume 191, Springer-Verlag, New York, 1985. Ben Ari, M. *Principles of Concurrent Programming*. Prentice Hall International, Englewood Cliffs, N.J., 1982. Brinch Hansen, P. Concurrent programming concepts. *ACM Computing Surveys 5*, 4 (Dec. 1973), 223-245. Brinch Hansen, P. *The Architecture of Concurrent Programs*, Prentice Hall, Englewood Cliffs, N.J., 1977. Burstall, R.M. Proving properties of programs by structural induction.

Experimental Programming Reports, No. 17. DMIP, Edinburgh, 1968. Clint, M. Program proving: Coroutines. *Acta Informatica 2*, 1 (1973), 50-63. Constable, R.L. and M.J. O'Donnell. *A Programming Logic*. Winthrop Publishers, Cambridge, Mass., 1978. de Bakker, J.W. Axiomatics of simple assignment statements. M.R.94, Mathematisch Centrum, Amsterdam, 1968. Dijkstra, E.W. A personal summary of the Gries-Owicki theory. EWD554, in *Selected Writings on Computing: A Personal Perspective*, E.W. Dijkstra, Springer Verlag, New York, 1982. On making solutions more and more fine-grained. EWD622, in *Selected Writings on Computing: A Personal Perspective*, E.W. Dijkstra, Springer Verlag, New York, 1982. Dijkstra, E.W. An Assertional Proof of a Protocol by G.L. Peterson. Technical Report EWD779, Burroughs, Corp. Feb. 1981. Fillman, R.E. and D.P. Friedman. *Coordinated Computing Tools and Techniques for Distributed Software*. McGraw Hill, New York, New York, 1983. Floyd, R.W. Assigning meanings to programs. *Proc. of Symposia in Applied Mathematics 19*, 1967, 19-31. Francez, N. *Fairness*. To appear. Springer Verlag, New York, 1986. Goldstine, H.H. and J. von Neumann. Planning and coding of problems for an electronic computing instrument. Report for U.S. Ord. Dept. In A. Taub (ed.) *Collected Works* of J. von Neumann, New York, Pergamon, Vol. 5, 1965, 80-151. Gries, D. The multiple assignment statement. *IEEE Trans. on Software Engineering SE-4*, 2 (March 1978), 87-93. Halpern, Brent T. *Verifying Concurrent Processes Using Temporal Logic*. Lecture Notes in Computer Science Volume 129, Springer-Verlag, New York, 1982. Hoare, C.A.R. An axiomatic basis for computer programming. *CACM 12*, 10 (Oct. 1969), 576-580. Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, C.A.R. Hoare and R. Perrot (eds.), Academic Press, New York, 1972. Hoare, C.A.R. Parallel programming: An axiomatic approach. *Computer Languages 1*, 1975, Pergamon Press, 151-160. Hoare, C.A.R. *Communicating Sequential Processes*. Prentice Hall, International, New Jersey, 1985. *Concurrent Euclid, The UNIX System, and Tunis*. Addison-Wesley, Reading, Mass., 1983. Holt, R.C., G.S. Graham, E.D. Lazowska, and M.A. Scott. *Structured Concurrent Programming with Operating Systems Applications*. Addison-Wesley, Reading, Mass., 1978. Igarashi, S. An axiomatic approach to equivalence problems of algorithms with applications. Ph.D. thesis, University of Tokoyo, 1964. Lamport, L. Towards a theory of correctness for multi-user data base systems. Technical Report CA-7610-

0712, Massachusetts Computer Associates, Wakefield, Mass., Oct. 1976. Lamport, L. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering SE-3*, 2 (March 1977), 125-143. Lamport, L. The "Hoare Logic" of concurrent programs. *Acta Informatica 14*, (1980), 21-37. The "Hoare Logic" of CSP and all that. *ACM TOPLAS 6*, 2 (April 1984), 281-296. Lamport, L. Logical foundation. In *Distributed Systems—Methods and Tools for Specification*. (M. Paul and H.J. Siegert eds.), Lecture Notes in Computer Science Volume 190, Springer-Verlag, New York, 1985. Lehman, D., A. Pnueli, and J. Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. *Proc. Eighth Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science Volume 115, Springer-Verlag, 1981, 264-277. Levitt, K.N. The application of program-proving techniques to the verification of synchronization processes. *Proc. AFIPS Fall Joint Computer Conference*, AFIPS Press, 1972, 33-47. McCarthy, J. Towards a mathematical science of computation. *Proc IFIP Congress 1962*, North Holland, Amsterdam, 1962, 21-28. Morris, F.L. and C.B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing 6*, 2 (April 1984), 139-143. Naur, P. Proof of algorithms by general snapshots. *BIT 6*, 4, 310-316. Axiomatic proof techniques for parallel programs. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, New York, 1975. An axiomatic proof technique for parallel programs I. *Acta Informatica 6*, (1976), 319-340. Paul, M. and H.J. Siegert (eds.). *Distributed Systems—Methods and Tools for Specification*. Lecture Notes in Computer Science Volume 190, Springer-Verlag, New York, 1985. Peterson, G.L. Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3 (June 1981), 115-116. Turing, A.M. Checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines*, University Mathematics Laboratory, Cambridge, 67-69. Yanov, Yu I. Logical operator schemes. *Kybernetika 1* (1958).