Acta Informatica 14, 21–37 (1980)

The 'Hoare Logic' of Concurrent Programs

Leslie Lamport

SRI International Computer Science Laboratory,
333 Ravenswood Avenue, Menlo Park, CA 94025, USA

Summary. Hoare's logical system for specifying and proving partial correctness properties of sequential programs is generalized to concurrent programs. The basic idea is to define the assertion $\{P\}\,S\,\{Q\}$ to mean that if execution is begun anywhere in S with P true, then P will remain true until S terminates, and Q will be true if and when S terminates. The predicates P and Q may depend upon program control locations as well as upon the values of variables. A system of inference rules and axiom schemas is given, and a formal correctness proof for a simple program is outlined. We show that by specifying certain requirements for the unimplemented parts, correctness properties can be proved without completely implementing the program. The relation to Pnueli's temporal logic formalism is also discussed.

Table of Contents

1. Introduction

In [3], Hoare described his well-known method for proving correctness properties of sequential programs. He built upon the idea, first published by Floyd [2], that an important class of properties of a program S – its 'partial correctness' properties – can be described by the condition that if an input predicate P is satisfied when the program is begun, then an output predicate Q will be true if and when S terminates. Hoare expressed this condition with the notation $P\,\{S\}\,Q$. He described a logical system for deriving, from a few basic rules,

0001-5903/80/0014/0021/$03.40

conditions of this form for arbitrary programs S. These rules can be divided into two classes: (i) general rules applicable to all programs, and (ii) rules that define the partial correctness semantics of the programming language. For example, the partial correctness properties of the programming language statement $x := x + 1$ are defined by the following axiom schema (rule for generating axioms), where P^x_{x+1} denotes the predicate obtained by substituting $x+1$ for x everywhere in the predicate P:

$$\vdash P^x_{x+1}\{x := x+1\}\, P. \tag{1}$$

To complete the specification of this statement, one must add the requirement that its executions always terminate.

Hoare's method is very useful because it allows one to specify a statement in terms of input and output predicates, and to prove the correctness of a program containing that statement, without having to describe how the statement is implemented. In particular, the implementation might require a complex subroutine, in which case we obtain a precise specification of the subroutine and a proof that the program is correct if the subroutine meets its specification.

We would like this same ability to specify a statement without describing its implementation for concurrent as well as sequential programs. Unfortunately, Hoare's method does not work for concurrent programs. To see why, observe that the statement

$$\begin{aligned} &[y := -y \ ; \\ &\ x := x+1; \\ &\ y := -y \ ] \end{aligned} \tag{2}$$

satisfies exactly the same input/output conditions (1) as the statement $x := x+1$. (We use square brackets in place of a **begin/end** pair.) However, if another process that uses the variable y may be executed concurrently, then the two statements have different meanings: statement (2) can 'interfere' with the execution of the other process because it temporarily changes the value of y, whereas the statement $x := x+1$ cannot. Hence, input and output predicates are not sufficient to specify a statement in a concurrent program.

To extend Hoare's method to concurrent programs, we introduce an assertion, which we write $\{P\}\, S\, \{Q\}$, having the following interpretation: if execution is begun *anywhere in* S with the predicate P true, then executing S will leave P true while control is inside S, and will make Q true if and when S terminates. (We consider the starting point of S to be in S, but the point at which it terminates to be outside S.) The assertion $\{P\}\, S\, \{Q\}$ is a condition constraining only what S does, and does not prohibit some concurrently executed statements from making P false while control is still in S.

Our assertion $\{P\}\, S\, \{Q\}$ differs from the Hoare assertion $P\, \{S\}\, Q$ in two ways: (i) P must be strong enough to guarantee that Q will be true upon termination even if execution is started in the middle of statement S, not just when it is started at the beginning; and (ii) P must remain true while control resides in S. However, if S is an indivisible, atomic operation, so that there are no places in S at which control can reside except at its starting point, then the assertions $\{P\}\, S\, \{Q\}$ and $P\, \{S\}\, Q$ are equivalent.

In this paper, we generalize Hoare's logical system to allow one to prove assertions of the form $\{P\}\, S\, \{Q\}$ for nonatomic as well as atomic statements S. In addition to the sequential programming constructs considered by Hoare, we will also be able to give a proof rule for the statement **cobegin** $S_1 \,[]\, \ldots \,[]\, S_n$ **coend**, which denotes that the n statements $S_1, \ldots, S_n$ are to be executed concurrently. Thus, we will extend Hoare's method to concurrent programs.

We caution the reader that the notation $\{P\}\, S\, \{Q\}$ has been widely used to be synonymous with Hoare's original notation $P\, \{S\}\, Q$. Hence, we are giving a new meaning to an existing notation, something that should never be done without good reason. We have found our new notation to be a great improvement over the current one when writing correctness proofs for concurrent programs, and we strongly urge its adoption. Hoare's assertion $P\, \{Q\}\, S$ is easily expressed in our notation as $\{P\}\, \langle S \rangle\, \{Q\}$, where $\langle S \rangle$ denotes that the statement S is to be executed as a single atomic action.

There are two types of properties one wants to prove about programs: (i) *safety properties*, which state that something bad cannot happen, and (ii) *liveness properties*, which state that something good must happen. Partial correctness is a safety property, stating that the program cannot terminate with incorrect output; and termination is a liveness property, stating that the program must terminate. Just as Hoare's method can state and prove only partial correctness properties, our generalization can state and prove only safety properties (the generalization of partial correctness).

A discussion of how liveness properties are specified and proved is beyond the scope of this paper. We merely mention that one proves safety properties in order to derive liveness properties. For example, one usually proves that a loop terminates (a liveness property) by first proving that an execution of the loop body cannot terminate without decreasing some nonnegative quantity (a safety property). Since a nonnegative quantity cannot decrease indefinitely, this shows that if the loop body always terminates, then the entire loop eventually terminates. However, some additional liveness property must be invoked to allow the formal conclusion that the loop body always terminates.

It is not our purpose to consider programming language design, so we will restrict ourselves to a limited set of program constructs. More sophisticated constructs can be handled with our method by defining the appropriate proof rules. In particular, one can apply our approach to specify the safety properties of any desired synchronization primitives. However, we will only show how this is done for the semaphore.

2. Programs and Predicates

We begin by considering what the meaning is of the statement $x := x+1$ in a concurrent program. First of all, we observe that it is impossible to decide whether even so simple an assertion as $\{x=1\}\, x := x+1\, \{x=2\}$ is true without knowing the locations of the possible internal control points of the statement $x := x+1$. Writing "$x := x+1$" does not specify a statement in a concurrent program because it does not specify the possible internal control points. Is it an

atomic statement with no internal control points? Is there a single internal control point between the evaluation of $x+1$ and the setting of x? Or are there control points inside the operations of fetching and storing x? To specify which of these possibilities we mean, we enclose atomic operations inside angle brackets $\langle\ \rangle$. For example, there are two control points in the statement $\langle x \rangle := \langle x+1 \rangle$: one at the beginning and one after the evaluation of $\langle x+1 \rangle$. (Remember our convention that the control point immediately after a statement is not in that statement.) Thus, the execution of this statement consists of two atomic actions:

1. Evaluating the expression $x+1$.
2. Setting x equal to this value.

The statement $\langle x \rangle := \langle x \rangle + 1$ has three control points: at the beginning, after the evaluation of $\langle x \rangle$, and after the evaluation of $\langle x \rangle + 1$.[1] The statement $\langle x := x + 1 \rangle$, like all atomic statements, has only one control point: at its beginning.

It is clear that the assertion $\{x=1\}\langle x := x+1 \rangle \{x=2\}$ is true, but what about the assertion $\{x=1\}\langle x \rangle := \langle x+1 \rangle \{x=2\}$? Certainly, if we execute this statement from the beginning with $x=1$, then x will remain equal to 1 while control is in the statement, and it will equal 2 when the statement terminates. However, what if the execution is begun after the evaluation of $x+1$? The statement $\langle x \rangle := \langle x+1 \rangle$ is equivalent to the following:

$$[\langle temp := x+1 \rangle; \\ \langle x := temp \rangle], \tag{3}$$

where *temp* is an internal variable that is invisible to the programmer. Starting the execution after the first assignment with $x=1$ does not guarantee that x will wind up equal to 2 unless *temp* equals 2. Thus, the assertion $\{x=1\}\langle x \rangle := \langle x+1 \rangle \{x=2\}$ is not valid. For a valid assertion, we would have to write something like the following:

$$\{x=1 \wedge [\text{control after the evaluation of } \langle x+1 \rangle \supset temp=2]\} \\ \langle x \rangle := \langle x+1 \rangle \quad \{x=2\}. \tag{4}$$

However, this doesn't make any sense as it is written, since there is no variable *temp* in the statement $\langle x \rangle := \langle x+1 \rangle$. Clearly, we need some way of referring to the value obtained by evaluating the expression $\langle x+1 \rangle$.

Evaluating an expression has a definite effect, even in the absence of 'side effects'. Hence, the meaning of expression evaluation needs to be defined just as carefully as the meaning of any complete program statement. We will need to be able to derive assertions of the form $\{P\}\, e\, \{Q\}$ for an expression e. Before doing this, we introduce the concept of the *name* of a statement or expression.

We assume that each occurrence of a statement or expression in a program has a unique name. (This includes any subexpression which is not part of a larger atomic expression.) If the statement $\langle x := x+1 \rangle$ appears twice in a program, each of these two occurrences has a unique name. In writing actual

[1] We regard the '+1' as an atomic unary function applied to the value $\langle x \rangle$

proofs, we use ordinary program labels as names. However, instead of defining a formal labeling scheme, we will simply introduce the convention of using *'S'* to denote the name of a specific occurrence of the statement or expression *S*.

To define the meaning of expressions, we give a method for rewriting them in terms of atomic assignment statements, statement concatenation (the ";"), and the **cobegin** statement. The meaning of an expression can then be derived from the meaning of its constituent parts. More precisely, the proof rules for these other language constructs will then enable one to derive assertions of the form $\{P\}\, e\, \{Q\}$ for an expression e.

We need to introduce a new class of *value* variables to refer to the value obtained by evaluating an expression. The expression $\langle x+1\rangle$ is rewritten as the atomic assignment statement $\langle value('\langle x+1\rangle') := x+1\rangle$.[2] Unlike the usual 'fictitious' variables introduced in proofs, these value variables are actually implemented. When the expression $\langle x+1\rangle$ is evaluated, its value must appear at least temporarily in some register; but ordinary programming languages do not give a name to the variable representing that register. The variable $value('\langle x+1\rangle')$ is just the variable *temp* of (3). Thus, we can now rewrite (4) as follows, removing the meaningless reference to the nonexistent variable *temp*:

$$\{x=1 \wedge [\text{control after the evaluation of } \langle x+1\rangle \supset \text{value } ('\langle x+1\rangle')=2]\}$$
$$\langle x\rangle := \langle x+1\rangle \quad \{x=2\}. \tag{5}$$

We define the meaning of an arbitrary expression by the following recursively applied rules for rewriting it, where $e, e_1, \ldots, e_n$ are expressions, and f is any standard mathematical function (such as the sum of its arguments).

(a) $\langle e\rangle \rightarrow \langle value('\langle e\rangle') := e\rangle$

(b) $f(e_1, \ldots, e_n) \rightarrow$
[cobegin $e_1 \,[]\, \ldots \,[]\, e_n$ **coend**;
$\langle value('f(e_1, \ldots, e_n)') := f(value('e_1'), \ldots, value('e_n'))\rangle$
]. (6)

This defines the meaning of any ordinary program expression in which every variable appears inside some atomic subexpression. These are the only expressions we consider in this paper. For example, applying (6) to the expression $\langle x\rangle + \langle y\rangle$ - first applying part (b), where f is the "+" function, then applying part (a) to the expressions $\langle x\rangle$ and $\langle y\rangle$ in the resulting **cobegin** statement - we obtain the following:

[cobegin $\langle value('\langle x\rangle') := x\rangle$ []
$\langle value('\langle y\rangle') := y\rangle$ **coend**;
$\langle value('\langle x\rangle + \langle y\rangle') := value('\langle x\rangle') + value('\langle y\rangle')\rangle$].

---

[2] Observe that two different occurrences of the expression $\langle x+1\rangle$ have two different *value* variables - hence the use of the name $'\langle x+1\rangle'$ rather than the expression $\langle x+1\rangle$ in the name of the variable $value('\langle x+1\rangle')$

Of course, the precise meaning of the **cobegin** statement and of the concatenation of two statements has yet to be defined.

In Hoare's method, the predicates that are used are just boolean functions of the values of program variables. For proving properties of concurrent programs, it is necessary to allow predicates that are functions of program locations as well. For any statement or expression S, we define the following three predicates, having the indicated interpretations:

$at('S')$	true if and only if control is at the beginning of S.
$in('S')$	true if and only if control resides somewhere in S - including at its beginning.
$after('S')$	true if and only if control resides at the point immediately following S.

Using these predicates, we can now express the assertion (5) more concisely as follows:

$$\{x=1 \wedge [after('\langle x+1\rangle') \supset value('\langle x+1\rangle')=2]\}$$
$$\langle x\rangle := \langle x+1\rangle \quad \{x=2\}. \tag{7}$$

We introduce the following conventions that will make it more convenient to write our assertions:

- If T is a substatement of S, then writing the assertion $\{P\}$ immediately before T is equivalent to writing $\{in('T') \supset P\}$ immediately before S; and writing $\{P\}$ immediately after T is equivalent to writing $\{after('T') \supset P\}$ immediately before S.
- $\{P\}\,\{Q\}$ is equivalent to $\{P \wedge Q\}$.

With these conventions, the assertion $[\{P\}\,S;\ \{R\}\,T]\,\{U\}$ is equivalent to

$$\{[in('S') \supset P] \wedge [in('T') \supset R\} \quad [S; T]\,\{U\}. \tag{8}$$

This is different from the assertion $\{P\}\,[S;\ \{R\}\,T]\,\{U\}$, which is equivalent to $\{P \wedge [in('T') \supset R]\}\,[S; T]\,\{U\}$.

We now introduce some relations between program statements. The relation 'S' *part of* 'T' means that the statement or expression 'S' is a substatement or subexpression of the statement or expression 'T'; hence, if $in('S')$ is true, then $in('T')$ must also be true. We write $'T' = 'S_1' \oplus \ldots \oplus 'S_n'$ to denote that control is in statement 'T' if and only if control is in exactly one of the statements 'S_i'. More precisely, $'T' = 'S_1 \oplus \ldots \oplus 'S_n'$ is an abbreviation for the following:

$$[in('T') \equiv in('S_1') \oplus \ldots \oplus in('S_n')] \wedge [\forall i\colon 'S_i' \textit{ part of } 'T'], \tag{9}$$

where the "$\oplus$" in (9) denotes 'exclusive or'.

The relation $'S' \| 'T'$ denotes that S and T lie in different substatements of a **cobegin** statement. The property of such substatements that we require is that executing S cannot affect the truth of any of the predicates $at('T')$, $in('T')$ or $after('T')$, and vice versa. Thus, we could also write $'S' \| 'T'$ for the statement $[[S; R]; T]$. However, we only need the relation $'S' \| 'T'$ when S and T are in different substatements of a **cobegin**, so those are the only "$\|$" relations we define.

To prove properties of programs, one must be able to perform logical reasoning about predicates in order to derive such theorems as $\vdash [x<7 \supset x<8]$. We assume that we are given some axiom system for deriving theorems about integers or other data types used by the program. However, since our predicates involve program control locations, we need some additional axioms for reasoning about control locations. These axioms are provided by the following four schemas.

A 1. $\vdash at('S') \supset in('S')$

A 2. $\vdash$ '*R' part of 'S*' $\wedge$ '*S' part of 'T*' $\supset$ '*R' part of 'S*'

A 3. $\vdash 'S' \| 'T' \equiv 'T' \| 'S'$

A 4. $\vdash ['S' \| 'T' \wedge$ '*R' part of 'S*'$] \supset 'R' \| 'T'$

The formal semantics we define in the next section describes a programming language in which a single statement cannot be executed concurrently by two different processes. For example, two different statements may not concurrently call the same subroutine, they must use different copies of that subroutine. (If such concurrent execution were possible, then our proof rules would not be valid.) To permit concurrent calls to the same subroutine, we would have to define a naming convention in which invocations of the same subroutine by different statements have different names. In this paper, we will avoid the problem by not considering subroutine calls at all. Of course, concurrent execution of the different substatements of a **cobegin** statement is permitted.

3. The Formal Logic

3.1. General Rules

We now define some rules of inference and axiom schemas that apply to all programs, regardless of the language in which they are written. These fall into three classes: (i) a rule to establish the equivalence of Hoare's $P\{S\}Q$ and our $\{P\}S\{Q\}$ when S is an atomic statement; (ii) the generalization of Hoare's 'rules of consequence' and (iii) rules for reasoning about program control locations. We will use the notation $\dfrac{A, \ldots, B}{C, \ldots, D}$ to indicate the inference rule that $C, \ldots,$ and D can be deduced from the truth of $A, \ldots,$ and B.[3]

3.1.1. Relation to Hoare's System. Hoare's assertion $P\{S\}Q$ is equivalent to our assertion $\{P\}S\{Q\}$ if S is atomic. However, we must remember that Hoare did not consider predicates that were functions of program control locations. Hence, we first define a *simple predicate* to be one that depends only upon the value of variables, and not upon program control locations. (A simple predicate may,

[3] This inference rule is equivalent to the metaimplication $\vdash A \wedge \ldots \wedge \vdash B \supset \vdash C \wedge \ldots \wedge \vdash D$, and is different from the simple implication $A \wedge \ldots \wedge B \supset C \wedge \ldots \wedge D$.

however, depend upon our special *value* variables.) We then have the following inference rule and axiom schema:

*P*1. (a) If P and Q are simple predicates, then:

$$\frac{P\,\{S\}\,Q}{\{P\}\,\langle S\rangle\,\{Q\}}.$$

(b) $\vdash in('\langle S\rangle') \equiv at('\langle S\rangle')$.

We use *P*1 to define the partial correctness semantics of any atomic statement. A programming language provides a way of building more complicated statements from atomic ones. (Recall that by (6), expressions are also statements.) To define the semantics of a programming language, we need to specify rules for deriving assertions about a composite statement from assertions about its components.

3.1.2. The Rules of Consequence. Hoare's system has the following two 'rules of consequence':

$$\text{(a)}\quad \frac{P\,\{S\}\,Q,\ Q \supset R}{P\,\{S\}\,R}$$

$$\text{(b)}\quad \frac{P\,\{S\}\,Q,\ R \supset P}{R\,\{S\}\,Q}. \tag{10}$$

We can generalize (10)(a) as follows:

*P*2. (a) $$\frac{\{P\}\,S\,\{Q\},\ Q \supset R}{\{P\}\,S\,\{R\}}.$$

Unfortunately, the similar generalization of (10)(b) is not true for concurrent programs. To see this, observe that from such a rule and (7) we could derive the following assertion:

$$\{x=1 \wedge [after('\langle x+1\rangle') \supset (value('\langle x+1\rangle') \triangleq 2 \wedge y=7)]\}$$
$$\langle x\rangle := \langle x+1\rangle \quad \{x=2\}.$$

However, this is an invalid assertion, since the predicate

$$x=1 \wedge [after('\langle x+1\rangle') \supset (value('\langle x+1\rangle') = 2 \wedge y=7)]$$

is true if we begin the execution with $x=1$ and $y \neq 7$, but becomes false after the expression $\langle x+1\rangle$ is evaluated.

From (10)(a) and (10)(b), Hoare could derive a number of rules that we must generalize to concurrent programs. Not having any counterpart to (10)(b), we require additional rules to derive them. We therefore add the following axiom schema and rules of inference to P2:

P 2.

(b) $\vdash \{false\}\, S\, \{false\}$

(c) $$\frac{\{P\}\, S\, \{Q\},\ \{P'\}\, S\, \{Q'\}}{\{P \wedge P'\}\, S\, \{Q \wedge Q'\},\ \{P \vee P'\}\, S\, \{Q \vee Q'\}}$$

(d) $$\frac{\{P\}\, S\, \{Q\},\ P \equiv P'}{\{P'\}\, S\, \{Q\}}.$$

3.1.3. Reasoning About Program Locations. So far, P1 is our only rule for deriving assertions of the form $\{P\}\, S\, \{Q\}$ without already having other assertions of that form. However, it can only derive such assertions for simple predicates P and Q. To derive these assertions for predicates that depend upon program control locations, we need the following axiom schemas and derivation rules. We will use P2 to combine the assertions yielded by them with those obtained from P1. In P3(d), we let "$loc('T')$" denote any one of the predicates "$in('T')$", "$at('T')$", "$after('T')$", or their negations.

P 3.

(a) $\vdash \{true\}\, S\, \{after('S')\}$

(b) $\vdash \{in('S')\}\, S\, \{true\}$

(c) $$\frac{\{in('S') \wedge P\}\, S\, \{Q\}}{\{P\}\, S\, \{Q\}}$$

(d) $$\frac{'S' \parallel 'T'}{\{loc('T')\}\, S\, \{loc('T')\}}.$$

3.2. Semantics of the Programming Language

We now give the derivation rules that define the 'safety semantics' of our programming language. We will assume the Hoare logic for sequential programs, so that the semantics of any atomic statement is defined by P1. We therefore only have to define the semantics of the constructs that combine statements into a larger single statement.

For sequential programs, the semantics of a programming language construct is defined by a derivation rule that allows one to derive an assertion of the form $P\, \{S\}\, Q$ for the entire statement S in terms of similar assertions about its components. For example, Hoare gave the following rule for the concatenation of two statements:

$$\frac{P\, \{S\}\, Q,\ Q\, \{T\}\, R}{P\, \{[S; T]\}\, R}. \tag{11}$$

For concurrent programs, each construct has such a derivation rule that is a generalization of the Hoare rule for sequential programs. However, we also need axiom schemas to specify the relation between the predicates $in('S')$, $at('S')$ and

after('S') for the entire statement S and for its component statements. For example, to specify the semantics of an atomic statement, we needed the axiom schema P1(b) in addition to the derivation rule P1(a).

The Composition Rule

P4.

(a) $$\frac{\{P\}\,S\,\{Q\},\ \{R\}\,T\,\{U\},\ Q \wedge at('T') \supset R}{[\{P\}\,S;\ \{R\}\,T]\,\{U\}}$$

(b) (i) $\vdash '[S;\ T]' = 'S' \oplus 'T'$

(ii) $\vdash at('[S;\ T]') \equiv at('S')$

(iii) $\vdash after('[S;\ T]') \equiv after('T')$

(iv) $\vdash after('S') \equiv at('T')$.

Derivation rule P4(a) is a straightforward generaliation of Hoare's rule (11). Recall that $[\{P\}\,S;\ \{R\}\,T]\,\{U\}$ is an abbreviation for the assertion (8) above. In (b)(iv), it is understood that S and T are parts of the statement $[S;\ T]$.

The While Statement

In the following rule, W denotes the statement **while** b **do** S **od**. The '$\oplus$' in (b)(i) is the operator defined by (9), while in (b)(iii) it denotes the ordinary 'exclusive or' operation. In (b)(iv), it is understood that 'S' and 'b' are parts of the **while** statement W.

P5.

(a) $$\frac{\{P\}\,b\,\{Q\},\ \{R\}\,S\,\{P\},\ [Q \wedge at('S') \wedge value('b') = true \supset R]}{\textbf{while}\ \{P\}\,b\ \textbf{do}\ \{R\}\,S\ \textbf{od}\ \{Q \wedge value('b') = false\}}$$

(b) (i) $\vdash 'W' = 'b' \oplus 'S'$

(ii) $\vdash at('W') \equiv at('b')$

(iii) $\vdash after('b') \equiv at('S') \oplus after('W')$

(iv) $\vdash after('S') \equiv at('b')$

The **cobegin** *Statement*

In the following rule, we let B denote the statement **cobegin** $S_1 \| \ldots \| S_n$ **coend**.

P6.

(a) $$\frac{\{P\}\,S_1\,\{P\},\ \ldots,\ \{P\}\,S_n\,\{P\}}{\{P\}\,B\,\{P\}}$$

(b) (i) $\vdash in('B') \equiv [[in('S_1') \vee after('S_1')] \wedge \ldots$
$\wedge\ [in('S_n') \vee after('S_n')]]$
$\wedge\ \sim[after('S_1') \wedge \ldots \wedge after('S_n')]$

(ii) $\vdash at('B') \equiv at('S_1') \wedge \ldots \wedge at('S_n')$

(iii) $\vdash after('B') \equiv after('S_1') \wedge \ldots \wedge after('S_n')$

(iv) $\forall i$: 'S_i' *part of* 'B'

(c) $\forall i \neq j$: '$S_i' \| 'S_j'$.

Rule P6(a) may seem rather weak. However, combined with the general rules P1-P3 and suitable rules for the other programming language statements, one can derive all previous assertional methods for proving safety properties of concurrent programs, such as the ones given in [6, 4], and [1]. In Sect. 4, we discuss how the methods of [6] and [4] can be derived from P6 and P1-P3.

Other Statements

We have defined the semantics of all the constructs considered by Hoare in [3] except the nonatomic assignment statement. We define the assignment statement $\langle x\rangle := e$ to be equivalent to the statement $[e;\ \langle x := value('e')\rangle]$, using our Definition (6) of the expression e. Hence, the semantics of the assignment $\langle x\rangle := e$ can be deduced from (6), P1, P4 and P6.

The semantics of a nonatomic array assignment of the form $\langle a\rangle(i) := e$ can also be defined in terms of other statements. One possible definition of this statement is that it should be equivalent to the following:

$$[\textbf{cobegin } e \,[]\, i \textbf{ coend};\ \langle a(value('i')) := value('e')\rangle].$$

(With this definition, the evaluation of e need not precede the evaluation of the subscript expression i.)

The reader should be able to write the appropriate rules for other simple statements, such as the '**if**...**then**...**else**'.

To define the partial correctness semantics of a synchronization primitive, one can either give a derivation rule and axiom schemas like P4-P6, or else simply define the primitive in terms of other statements in the language. For example, for the purpose of proving safety properties, we could define the semaphore operation $P(s)$ to be the statement

$$\langle \textbf{if } s>0 \textbf{ then } s := s-1 \textbf{ else abort fi}\rangle.$$

(Introducing the possibility of aborting the entire program does not alter its safety properties.) Alternatively, we could define the safety properties of the operation $P(s)$ by the following derivation rule.

For any simple predicate Q:

$$\{Q^s_{s-1} \wedge s \geqq 0\}\ P(s)\ \{Q \wedge s \geqq 0\}.$$

Of course, the semaphore operation $V(s)$ is equivalent to the statement $\langle s := s+1\rangle$.

3.3. Two Theorems

In any logical system, it is a tedious process to prove significant theorems directly from the axioms. Instead, one builds up a battery of theorems that can be used to simplify proofs. We present two such useful theorems here. The second one will be used in Sect. 4. We indicate how they may be derived from the axioms, but we will not bother giving formal proofs.

Theorem 1. *The following is a valid derivation rule:*

$$\frac{in('S') \supset \sim P}{\{P\}\, S\, \{Q\}}.$$

Theorem 2. *If the predicate P mentions only variables that are not changed by S, and mentions only program control locations for statements $'T'$ for which $\vdash 'S' \| 'T'$, then $\vdash \{P\}\, S\, \{P\}$.*

With our programming language, the only variables changed by a statement S are those that appear on the left-hand side of an assignment statement somewhere in S. (This includes the *value* variables in an expression.) The only way a predicate can mention control locations for $'T'$ is with logical combinations of the predicates $at('T')$, $in('T')$ and $after('T')$.

Theorem 1 follows easily from P2 and P3. To prove Theorem 2, we would first use induction on the 'size' of S to prove the theorem for simple predicates P. We would then use P3(d), P2(c) and induction on the 'size' of P to prove it for arbitrary predicates P. However, to do this we would first have to define precisely the class of well-formed predicates.

4. Proving Properties of Concurrent Programs

We now consider how our inference rules and axiom schemas are used to prove safety properties of concurrent programs, and how they are related to the previous proof method of Owicki [6] and the one we described in [4]. For simplicity we restrict ourselves to programs with a fixed number of processes, which were the only ones considered in [4]. There is no difficulty extending our remarks to the more general type of program with nested **cobegin** statements considered by Owicki.

A safety property for such a program S is expressed by the assertion that some predicate Q is always true. To further simplify things, we assume that Q is automatically satisfied after S terminates, i.e., we assume $\vdash after('S') \supset Q$. To prove that Q is always true, we must find a predicate P with the following three properties.

1. The initial condition implies that P is true. The initial condition might simply be $at('S')$, or it might specify some initial variable values.
2. $\vdash [\{P\}\, S\, \{true\}]$.
3. $\vdash P \supset Q$.

The second property guarantees that if P is true initially, then it will remain true while S is being executed. The first property means that P is true initially, so it is always true while S is being executed. The third property then implies that Q is always true, as required.

We assume that S is of the form **cobegin** $S_1 \,[]\, \ldots \,[]\, S_n$ **coend**, where the S_i are the individual processes. We denote this statement by **cobegin** $[]\, S_i$ **coend**. We also write $\bigwedge_j A_j$ to denote the conjunction of the predicates A_j, where the range of j will be clear from the context.

Our problem is to verify the second property: $\vdash \{P\}\, S\, \{\text{true}\}$. The predicate P will be of the form $\bigwedge_i [in('S_i') \supset P_i]$, so the assertion $\{P\}\, S\, \{\text{true}\}$ may be written as

$$\textbf{cobegin} \parallel \{P_i\}\, S_i\, \{\text{true}\}\ \textbf{coend}. \tag{12}$$

Combining P2 and P6, we see that to prove (12) it suffices to prove the following for all i.

$$\{P_i\}\, S_i\, \{\text{true}\} \tag{13}$$

$$\forall j \neq i\colon\ \{P_i \wedge P_j\}\, S_i\, \{\text{true}\} \tag{14}$$

Verifying condition (13) involves proving that each process S_i is 'correct' as a nonterminating sequential program. It corresponds to the verification of the 'proof figure' for each part of a **cobegin** statement in [6], and to the verification of 'consistency' for each process in [4].

We now consider Condition (14). Suppose that each P_j is of the form $\bigwedge_T [in('T') \supset P_j^T]$, where T ranges over the parts of S_j. Using P2, we can reduce (14) to

$$\forall j \neq i\colon\ \forall T\colon\ \{P_i \wedge P_j^T\}\, S_j\, \{\text{true}\}. \tag{15}$$

Proving the assertions in (15) requires decomposing S_i into its component statements, using the proof rules for the programming language constructs with which the program is composed. When this decomposition has been carried down to the level of atomic statements, the verification of (15) becomes the verification of Owicki's 'noninterference' conditions [6] or of the 'monotonicity' conditions in [4], depending upon the programming language involved. (A language similar to the one in this paper is used in [6], while [4] uses a flowchart language.)

To illustrate how our rules and axioms can be applied, we use a simple two-process mutual exclusion algorithm. There are two processes, numbered 1 and 2. Each process i has a critical section, which we label cs_i. The condition we want always to be true is $\sim(in(cs_1) \wedge in(cs_2))$. Note that this predicate is trivially true after the program has terminated. The program S and the assertion $\{P\}\, S\, \{\text{true}\}$ needed in the proof are indicated in Fig. 1. The following is the predicate P defined by Fig. 1, where we let ~ 1 equal 2 and ~ 2 equal 1:

$$\bigwedge_i [(in\, b_i \supset x_i = 1) \wedge (in\, cs_i \supset \sim in\, cs_{\sim i})]. \tag{16}$$

Observe that P has the requisite properties (1) and (3) discussed above, since it is implied by the initial condition $at('S')$, and it implies the desired predicate $\sim(in(cs_1) \wedge in(cs_2))$. Thus, to prove the desired mutual exclusion property, we need only prove $\vdash \{P\}\, S\, \{\text{true}\}$. We now sketch the proof of this.

Let W_i denote the **while** loop w_i of Fig. 1 together with its embedded predicates, and let D_i denote the body of that loop together with its predicates. Let R_i denote the assertion $in(cs_{\sim i}) \supset x_{\sim i} = 1$.

```
cobegin □
w_i: while true
     do
                noncritical section_i;
                ⟨x_i := 1⟩;
     {x_i = 1} b_i: [          u_i: while ⟨x_~i = 1⟩
                                    do ⟨skip⟩ od;
                    {~in cs_~i} cs_i: critical section_i
                ];
                ⟨x_i := 0⟩
     od {true}
coend
```

Fig. 1

We first prove $\vdash \{R_i\}\, W_i \{true\}$. To do this, P5 implies we need only prove $\vdash \{R_i\}\, D_i \{R_i\}$. Using P4 to decompose D_i, P1 and Theorem 2 to handle the atomic assignment statements, and then applying P2(c) and P3(d) to the critical and noncritical sections, we conclude that we need to prove only the following assertions.

(a) $\vdash \{x_{\sim i} = 1\}$ noncritical section$_i$ $\{x_{\sim i} = 1\}$

(b) $\vdash \{x_{\sim i} = 1\}$ critical section$_i$ $\{x_{\sim i} = 1\}$ (17)

$\vdash \{R_i \wedge x_i = 1\}\, u_i$: **while**...**od** $\{R_i\}$. (18)

We take (17) to be part of the specification of the critical and noncritical sections. Of course, (17) follows from Theorem 2 if the critical and noncritical sections of w_i do not change $x_{\sim i}$. To prove (18), we apply P5(a) and P2(a) - using (6)(a) and P1 to handle the expression $\langle x_{\sim i} = 1 \rangle$ - to obtain

$\vdash$ **while** $\{R_i$ and $x_i = 1\}$ $\langle x_{\sim i} = 1 \rangle$
do
$\{R_i$ and $x_i = 1\}$ $\langle$skip$\rangle$
od $\{R_i$ and $x_i = 1$ and $x_{\sim i} \neq 1\}$.

Using P2(a) and P5(b)(i) then yields (18).

This completes the proof of $\vdash \{R_i\}\, W_i \{true\}$. A similar (but simpler) proof yields $\vdash \{in\, b_{\sim i} \supset x_{\sim i} = 1\}\, W_i \{true\}$. Using P2(c) and P3(d) we then obtain $\vdash \{P\}\, w_i$: **while**...**od** $\{P\}$, where P is given by (16). We can then apply P3(a), P2(c) and P6(b)(iii) to obtain the desired assertion

$$\{P\}\ \textbf{cobegin}\ldots\textbf{coend}\ \{true\}.$$

The observant reader will have noticed that although our algorithm satisfies the safety property of guaranteeing mutual exclusion, it is not very useful because it lacks the desirable liveness property that a process must eventually be able to enter its critical section. It is quite possible for both processes to remain forever in their **while** loops u_i. To obtain an algorithm possessing this liveness

property, we must replace the ⟨skip⟩ statement in u_i by some statement T_i and add more statements to the noncritical section. One way of doing this yields an algorithm described in [7], but there are also other ways which yield different algorithms having the same liveness property.

Since we are not considering liveness properties in this paper, the precise algorithms do not concern us. What we will do, however, is sketch a simple proof that all these algorithms guarantee mutual exclusion, without having to specify them completely. We prove that if their implementation satisfies certain assertions of the form $\vdash \{P\} S \{Q\}$, then these algorithms all guarantee mutual exclusion.

cobegin□
w_i: **while** *true*
do
noncritical section$_i$;
$\langle x_i := 1 \rangle$;
b_i: [$\{x_i = 1\}$ u_i: **while** $\langle x_{\sim i} = 1 \rangle$
do $\{Q_i\}$ T_i **od**;
$\{\sim in\,cs_{\sim i}\}$ cs_i: critical section$_i$
];
$\langle x_i := 0 \rangle$
od {true}
coend

Fig. 2

As with the program of Fig. 1, the mutual exlusion property is demonstrated by proving the assertion described in Fig. 2, where Q_i is a predicate that must be specified along with the statement T_i. The proof is similar to that of the assertion of Fig. 1 and will be left to the reader. However, we need the following assumptions in addition to (17), where WD_i denotes the body of the **while** loop w_i:

$$\begin{aligned} &\text{(a)} \quad \vdash [at'T_i' \wedge x_i = 1] \supset Q_i \\ &\text{(b)} \quad \vdash \{Q_i\}\, T_i \{x_i = 1\} \\ &\text{(c)} \quad \vdash \{Q_{\sim i}\}\, WD_i \{Q_{\sim i}\}. \end{aligned} \tag{19}$$

The assertions (19) constitute the specification of the statements T_i. For the algorithms that we have in mind, including the one in [17], part (c) is a simple consequence of Theorem 2.

This example illustrates how, as in Hoare's method for sequential programs, assertions of the form $\vdash \{P\} S \{Q\}$ can be used to specify requirements for parts of a program without having to specify their implementation. Note that our specifications (19) involved the 'unimplemented assertions' Q_i, as well as the unimplemented statements T_i. This same phenomenon was observed in [4], where we described an informal hierarchical design procedure for multiprocess programs. There we needed to leave certain assertions unspecified in the higher-level design of a 'subroutine'. In fact, the method we have described here provides a formal basis for the informal design procedure of [4].

5. The Relation to Temporal Logic

We have given formal rules for deriving assertions of the form $\{P\}\, S\, \{Q\}$ and have informally indicated what these assertions mean. For example, we proved an assertion of the form $\{P\}\, S\, \{\text{true}\}$ and then used this result to conclude that if execution of S is begun with P true, then P will remain true while S is being executed. However, this latter reasoning is purely informal.

To formalize the concept that 'P will always remain true', we must use temporal logic, as applied to programs by Pnueli in [7] and explained in more detail in [5]. The operator $\Box$ is introduced to mean 'at all times now and in the future', so $\Box P$ means that P is true now and will always remain true.

To make the formal connection between 'Hoare logic' assertions and temporal assertions, we use the following principle:

$$\text{If } \vdash \{P\}\, S\, \{P\} \text{ then } \vdash_S P \supset \Box P, \tag{20}$$

where $\vdash_S$ denotes provability in the temporal logic defined by the program S. (This indicates that S is the entire program, not just part of some larger one.) Note that any assertion $\{P'\}\, S\, \{Q\}$ can be written in the form $\{P\}\, S\, \{P\}$, where P is the predicate

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

One uses (20) to derive temporal assertions about the program, then uses temporal logic to prove the desired properties of the program, which are expressed as temporal assertions.

This method of reasoning is not powerful enough to derive certain types of safety properties about programs, properties stating that one thing must happen before another. To state these properties, we need the more general form of temporal assertion $R \Box P$, described in [5], which states that P will be true as long as R remains true.

To prove this type of temporal assertion, we generalize our 'Hoare logic' to include assertions of the form $R \vdash \{P\}\, S\, \{Q\}$. This assertion means that if execution is begun anywhere in S with P and R true, then, as long as R remains true, P will be true while control is in S and Q will become true if and when S terminates. All of our rules are easily generalized to handle this more general type of assertion. For example, the rule P4(a) can be generalized to the following:

$$\frac{R \vdash \{P\}\, S\, \{Q\},\ R \vdash \{V\}\, T\, \{U\},\ R \wedge Q \wedge at('T') \supset V}{R \vdash [\{P\}\, S;\ \{V\}\, T]\, \{U\}}.$$

To proceed from these generalized 'Hoare logic' assertions to temporal assertions, we use the following generalization of (20):

$$\text{If } R \vdash \{P\}\, S\, \{P\} \text{ then } \vdash_S P \supset (R \Box P).$$

6. Conclusion

We have described a generalization of Hoare's logic to current programs. It provides a formal logical system for proving safety properties of such programs.

When the proof is carried out for a completely specified program, it turns out to be essentially the same as the proof one would obtain with one of the previously described proof methods [6, 4, 1]. We could thus view our method as providing a general logical framework that yields the previous methods when applied to specific programming languages. However, our method also provides a convenient way to prove the correctness of programs that are not completely implemented down to the level of indivisible atomic statements, something that was difficult to do with previous methods. One can state requirements for an unimplemented statement in terms of certain assertions, and then prove that the entire program satisfies its desired safety properties if the unimplemented statements meet their requirements. The proof that these requirements are met is then carried out when the statements are implemented.

Our system is more complicated than Hoare's in that a richer class of predicates is required: our predicates can depend upon the values of the 'program counters' of individual processes. Moreover, we cannot just restrict ourselves to preconditions and postconditions, conditions that hold only at the beginning and end of a statement's execution. We must also consider what is true *during* the execution of the statement. This added complexity reflects the fact that concurrent programs are inherently more complex than sequential ones. There is no way to reduce this complexity without limiting the kind of concurrent execution that is allowed. Such limitations can be enforced by the programming language and will yield powerful rules of inference that simplify the proofs. For example, if two sustatements of a **cobegin** cannot reference the same variable, then Theorem 2 can be used to derive a much more powerful rule of inference than P6(a). However, the programs that meet this restriction are not very interesting.

Acknowledgments. We wish to thank Sheldon Finkelstein and Amir Pnueli for their comments on earlier versions of this paper. This work was supported in part by the National Science Foundation under grant number MCS-7816783.

References

1. Ashcroft, E.A.: Proving assertions about parallel programs. J. Comput. Systm. Sci. **10**, 110–135 (1975)
2. Floyd, R.W.: Assigning meanings to programs. Proc. A.M.S. Symp. in Applied Math., Amer. Math. Soc. pp. 19–31, 1967
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM **12**, 576–583 (1969)
4. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Soft. Engrg. SE-3, **2**, 125–143 (1977)
5. Lamport, L.: Sometime is sometimes not never: On the temporal logic of programs. Proceedings of the Seventh Annual Symposium on Principles of Programming Languages, ACM SIGACT-SIGPLAN, January 1980
6. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Informat. **6**, 319–340 (1976)
7. Pnueli, A.: The temporal logic of programs. Proc. of the 18th Symposium on the Foundations of Computer Science, ACM, November 1977

Received January 23, 1979; Revised February 8, 1980