The "Hoare Logic" of CSP, and All That*

Leslie Lamport
Fred B. Schneider

TR 82-490
December 1982

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work is supported in part by National Science Foundation grant MCS-81-04459 at SRI International and MCS-81-03605 at Cornell University.

The "Hoare Logic" of CSP, and All That

LESLIE LAMPORT
SRI International
and
FRED B. SCHNEIDER
Cornell University

Generalized Hoare Logic is a formal logical system for deriving invariance properties of programs. It provides a uniform way to describe a variety of methods for reasoning about concurrent programs, including noninterference, satisfaction, and cooperation proofs. We describe a simple meta-rule of the Generalized Hoare Logic—the Decomposition Principle—and show how all these methods can be derived using it.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*control primitives*;

General Terms: Languages, Verification

Additional Key Words and Phrases: Invariance, safety properties, Generalized Hoare Logic, decomposition principle, noninterference, message-passing, communicating sequential processes

1. INTRODUCTION

A variety of methods have been proposed for reasoning about concurrent programs. Most of these are for proving *safety properties*—properties asserting that the program never enters an unacceptable state. Some methods deal with concurrent programs that use shared variables [2, 4, 11, 12, 16, 18]; more recently, the absence of shared variables in CSP [10] has led to other techniques for reasoning about such programs [1, 5, 15, 19]. This diversity of methods has obscured the fact that there is really a simple principle involved in all of them: proving the invariance of an assertion.

The *Generalized Hoare Logic* (GHL) [13], a formal logical system for deriving invariance properties of programs from the properties of their components,

This work is supported in part by National Science Foundation Grant MCS 81-04459 at SRI International and MCS 81-03605 at Cornell University.
Authors' addresses: L. Lamport, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025; F. B. Schneider, Department of Computer Science, Cornell University, Ithaca, NY 14853.

provides a uniform way to describe these other assertional methods for reasoning about concurrent programs. This allows a comparison of the methods, which can help in understanding them.

GHL is a generalization of the programming logic proposed by Hoare [9] for reasoning about sequential programs. In Hoare's logic, one views a program as a *partial correctness* relation between two predicates, a *precondition* and a *postcondition*, meaning that if the program is started in a state that satisfies the precondition and it terminates, then the final state satisfies the postcondition. Using the logic, one derives partial correctness properties of sequential programs from properties of their components. This provides an elegant formulation of Floyd's method [8] for proving the correctness of a program. Although one must still perform the same basic verification steps, by providing a new way of viewing programs, Hoare's approach has led to improved ways for specifying and constructing programs, such as Dijkstra's programming calculus [7].

In the same vein, GHL can be viewed as another formulation of the methods for reasoning about concurrent programs. It leads to a new way of viewing a concurrent program—as maintaining the invariance of a predicate. This, in turn, has led to new specification methods for concurrent programs [14], and we hope it will lead to improved techniques for constructing concurrent programs.

Using invariance to reason about concurrent programs is not new, having been proposed by Ashcroft [2] and Keller [11]. What GHL does is provide a logic for deriving invariance properties of a program. In this paper, we show how various techniques for establishing safety properties of concurrent programs can be formulated in GHL in terms of invariance. Section 2 contains an overview of GHL and its principles; Section 3 applies the principles to some popular proof systems.

2. PROVING SAFETY PROPERTIES WITH GHL

2.1 Atomic Actions

During execution of a program, there will be times when the system is in undefined or pathological states. For example, when the contents of a bit of memory is changed, it will pass through an intermediate state in which its value is neither zero nor one. Since a safety property asserts that a predicate is always true, it is unlikely to hold if such a transient intermediate state were visible. We therefore assume the existence of *atomic actions*, which transform the system from one state to another without passing through any visible intermediate states. An operation whose execution is atomic will be enclosed in angle brackets.

The notion of an atomic operation is irrelevant for sequential programs. However, to specify a concurrent program, one must specify which operations are atomic. To see this, consider the following two programs π and ρ:

π: **cobegin** $\langle x := x + 1 \rangle$ // $\langle x := x + 1 \rangle$ **coend**
ρ: **cobegin** $\langle x \rangle := \langle x + 1 \rangle$ // $\langle x \rangle := \langle x + 1 \rangle$ **coend**

Execution of π consists of two atomic actions, each incrementing x by one. Hence, the program will terminate with x two greater than its initial value. Execution of ρ consists of four atomic actions: two that fetch the value of x and increment it, and two that store a value in x. If both fetches precede both stores, then ρ will increment x by one; otherwise it will increment x by two. Thus, executing π and ρ can produce different results, so they are different programs. If we simply write

cobegin $x := x + 1$ // $x := x + 1$ **coend**

without specifying what operations are atomic, then no assertion can be made about the final value of x. (Consider what could happen if fetching or storing each bit is a separate atomic operation, two's complement representation is used, and the program is started with $x = -1$.)

We place no restrictions on what can appear inside angle brackets, thereby allowing a complicated statement or sequence of statements to be atomic. This allows one to write atomic operations that need not terminate, raising the question of how a nonterminating atomic action can be executed, since, being atomic, it cannot be interrupted before completion. We answer this by requiring that an atomic operation not be executed unless it will terminate. Conditionally terminating atomic operations can then be used to represent synchronization primitives. For example, a $P(s)$ semaphore operation can be represented by[1]

$\langle s := s - 1;$ **while** $s < 0$ **do skip od**$\rangle$.

We do not advocate allowing a programmer to put anything inside angle brackets; that would be impossible to implement. We are simply observing that nothing has to be added to GHL in order to reason about the synchronization primitives provided by concurrent programming languages, since these primitives can be represented as conditionally terminating atomic operations.

2.2 Proving Safety Properties

A safety property has the following form, for some predicates *Init* and *Etern*:

SP: If the program is started in any state satisfying *Init*, then every state reached during its execution satisfies *Etern*.

Examples of safety properties are deadlock freedom, where *Etern* asserts that the program is not in a deadlock state, and mutual exclusion, where *Etern* asserts that two processes are not both in their critical sections. The general method for proving SP is to find a predicate I such that:[2]

S1. $Init \Rightarrow I$.
S2. If the program is started in any state satisfying I, then every state reached during its execution satisfies I.
S3. $I \Rightarrow Etern$.

S1–S3 trivially imply SP. A predicate I satisfying S2 is called an *invariant* of the program. Properties S1 and S3 are usually easy to verify, since they are static properties of the predicates. Property S2, invariance, is the core of the proof, since it is a property of the program's behavior—a dynamic property.

The invariance of a predicate I is proved by showing that each atomic action, if started in a state satisfying I, will terminate in a state with I true. By a trivial

[1] This representation is consistent with the safety properties of the semaphore operation, but not its liveness properties. Since GHL deals only with safety properties, that is sufficient.

[2] We use $\Rightarrow$ to denote logical implication.

induction argument, this implies that I is invariant. The virtue of this approach is that it considers each atomic action *in isolation* and ignores the history of the computation.

Viewing atomic actions in isolation requires that the state of the system include control information—the values of "program counters"—to determine what atomic actions can occur next. The control state is just as much a part of a program's state as the values of its variables. Indeed, every programmer knows that variables can often be eliminated by encoding their values in the control state and that control structure can often be simplified by adding extra variables. Moreover, it is well known that one cannot verify concurrent programs without reasoning about their control state [17]. Whether one reasons about control state directly or by introducing "dummy variables" (sometimes called *ghost* or *auxiliary variables*) is a matter of taste. We find it inelegant to add dummy variables to a program when their values can easily be defined as functions of the real program state.

When reasoning about invariance properties, we need to consider what the program does when started from *any* state.[3] We therefore do not assume any preferred starting state, allowing execution to begin in any state—even one with control in the middle of the program. If, as is usually the case, we are only interested in properties that hold when the program is started with control at the beginning, then the predicate *Init* of SP will state that control is at the program's entry point.

2.3 GHL: A Logic of Invariance

A program is made up of declarations, which we ignore, and executable *program fragments*. A program fragment may itself be composed of smaller program fragments. For example, an **if–then–else** statement is a program fragment composed of three smaller fragments: the conditional test, the **then** clause, and the **else** clause. The set of atomic operations that make up program fragment π is denoted $\alpha[\pi]$. If π is composed of subfragments $\pi_1, \ldots, \pi_n$, then

$$\alpha[\pi] = \alpha[\pi_1] \cup \cdots \cup \alpha[\pi_n].$$

Formulas of GHL have the form

$$\{I\}\ \pi\ \{I\}$$

where I is a predicate and π a program fragment. This formula means that executing any atomic action in π starting in a state in which I is true leaves I true. A simple induction argument shows that if π is the entire program, then this is equivalent to S2; so $\{I\}\ \pi\ \{I\}$ means that I is an invariant of π.

In [13], GHL is described for a simple programming language. An inference rule for each language construct is given, enabling invariance properties of statements to be derived from invariance properties of their components. All these inference rules are based on the following principle:

[3] A *state* consists of a value for the program's control state and, for each program variable, a value that is consistent with its declared type.

Decomposition Principle. If $\alpha[\pi] = \alpha[\pi_1] \cup \cdots \cup \alpha[\pi_n]$, then

$$\frac{\{I\}\ \pi_1\ \{I\},\ \ldots,\ \{I\}\ \pi_n \{I\}}{\{I\}\ \pi\ \{I\}}.$$

For example, the atomic operations of $S;T$ are just the atomic operations of S together with the atomic operations of T:

$$\alpha[S;T] = \alpha[S] \cup \alpha[T],$$

so the Decomposition Principle yields the following inference rule for statement concatenation:

$$\frac{\{I\}\ S\ \{I\},\ \{I\}\ T\ \{I\}}{\{I\}\ S;T\ \{I\}}.$$

In this paper, we use the Decomposition Principle to derive inference rules for some programming constructs not considered in [13].

To describe how GHL is used, we shall also require some inference rules from [13] that apply to all programming constructs. The first rule allows invariance properties of the same program fragment to be combined. In light of the meaning of $\{I\}\ \pi\ \{I\}$, it is obviously valid.

Conjunction Rule:

$$\frac{\{I_1\}\ \pi\ \{I_1\},\ \ldots,\ \{I_n\}\ \pi\ \{I_n\}}{\{I_1 \wedge \cdots \wedge I_n\}\ \pi\ \{I_1 \wedge \cdots \wedge I_n\}}.$$

As mentioned earlier, the state of the system must include control information. GHL uses the following predicates to describe the control state:

$at(\pi)$ ≡ "control resides at an entry point of π".

$in(\pi)$ ≡ "control resides somewhere in π, including at its entry point".

$after(\pi)$ ≡ "control resides at a point immediately following π".

In GHL the definition of a language construct includes a specification of its *at*, *in*, and *after* predicates, which serve to define its control flow semantics.

We define $\{P\}\ \pi\ \{Q\}$ to be an abbreviation for[4]

$$\{in(\pi) \Rightarrow P \wedge after(\pi) \Rightarrow Q\}\ \pi\ \{in(\pi) \Rightarrow P \wedge after(\pi) \Rightarrow Q\}.$$

If π is an atomic operation, then $\{P\}\ \pi\ \{Q\}$ means that executing π starting in a state in which P holds produces a state in which Q is true. (Recall that an atomic operation π cannot be executed unless it will terminate.) When π is, in addition, a complete statement, this is the same meaning as in Hoare's programming logic.[5]

[4] Since $\{I\}\ \pi\ \{I\}$ is a special case of $\{P\}\ \pi\ \{Q\}$, we have seemingly defined it to have two different meanings. However, the following Locality Rule implies that the meanings are equivalent.

[5] In Hoare's original notation this would be written as $P\ \{\pi\}\ Q$.

For a program fragment π that might not be atomic, $\{P\}\ \pi\ \{Q\}$ means that if control is anywhere inside π and P holds, then executing the next atomic operation in π will either

(1) leave control in π with P true, or
(2) leave control at an exit point of π with Q true.

If π is atomic, then there is only one control point inside π—the one at its entry point.

The only other GHL inference rule we need follows from the observation that it is possible to execute an atomic operation π only if $in(\pi)$ is true, and that control is at an exit point of π only if $after(\pi)$ is true.

Locality Rule:

$$\frac{\{in(\pi)\ \wedge\ I\}\ \pi\ \{after(\pi)\ \wedge\ I\}.}{\{I\}\ \pi\ \{I\}}.$$

The Locality Rule can be derived from P2 and P3 of [13]. Note that from the definition of $\{P\}\ \pi\ \{Q\}$, it follows immediately that $\{in(\pi)\ \wedge\ I\}\ \pi\ \{after(\pi)\ \wedge\ I\}$ means π leaves $(in(\pi) \vee after(\pi)) \Rightarrow I$ invariant.

3. FORMULATIONS IN GHL

We turn now to some specific methods for proving programs and show how they can be formulated in GHL. These methods are for proving one specific safety property: partial correctness. Partial correctness of program π with respect to precondition P and postcondition Q is an instance of SP with the predicates

$$Init\text{: } at(\pi)\ \wedge\ P$$

$$Etern\text{: } after(\pi) \Rightarrow Q.$$

Note that because we do not assume any preferred starting point, we have to include the conjunct $at(\pi)$ in *Init* to specify that the program is started at the beginning.

3.1 Floyd's Method Revisited

Floyd's method [8] uses the flowchart representation of the program. The program's flowchart consists of a set of nodes connected by directed arcs. The nodes represent program fragments; the arcs represent control flow. There is an arc from node π to node ϕ if the entry point of ϕ is an exit point of π. For example, consider Figure 1, where boxes are labeled $\pi_1 - \pi_3$ and control points are labeled a_1–a_4. The node π_2 corresponds to a conditional test. It has two exit points, a_3 and a_4, the former being the entry point of π_3 and the latter the exit point of the entire program. Which one is reached depends on the outcome of the test. Note also that a_2 is the exit point of both π_1 and π_3, as well as the entry point of π_2.

In Floyd's method, partial correctness with respect to a precondition P and a postcondition Q is proved by associating a predicate with each control point. P is associated with the entry point, Q is associated with the exit point, and the predicates associated with the other control points are chosen so that the

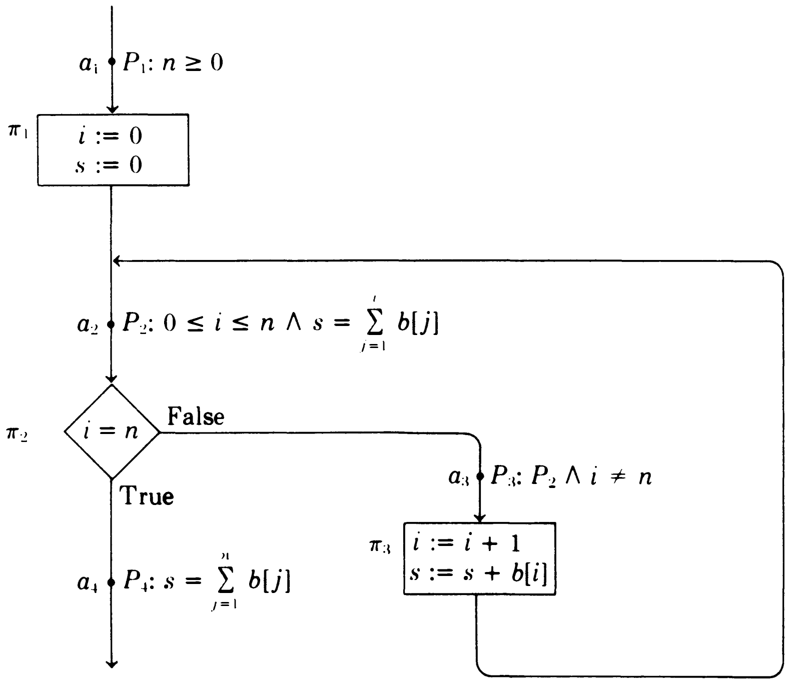

Figure 1

following condition holds:

AI: If the program is started at any control point and the predicate associated with that control point is true, then throughout execution the predicate associated with the current control point is true.

This implies that if the program is started at its entry point with the precondition P true and it reaches its exit point, then its final state satisfies the postcondition Q. To prove *AI*, it is necessary to prove the following verification condition for each box π_i:

VC: If π_i is executed starting in a state that satisfies the predicate associated with its entry point, then the predicate associated with the control point it reaches after its execution will be true.

A simple induction argument shows that the truth of VC for every box π_i implies the truth of *AI*. This reduces the problem of proving partial correctness to proving a verification condition for each box. Floyd gave simple rules for proving these verification conditions.

Figure 1 illustrates the use of Floyd's method to prove a partial correctness condition for a simple program that sums the first n elements of a one-dimensional array b and leaves the result in s. Our goal is to prove that if execution is begun with $n \geq 0$, then it can terminate only with $s = \sum_{j=1}^{n} b[j]$. In the figure, the predicate associated with each control point a_i is labeled P_i; the precondition $n \geq 0$ is associated with control point a_1 and the postcondition $s = \sum_{j=1}^{n} b[j]$ with a_4. There are three verifications conditions—one for each box—which are easily checked. For example, the verification condition for π_2 asserts that if testing $i = n$ with P_2 true yields false, then P_3 must hold, and if it yields true then P_4 must hold. This verification condition follows immediately from the definitions of P_2, P_3, and P_4.

Using the program π of Figure 1, we now illustrate that the Floyd method is an instance of the use of GHL to prove a safety property, in which AI is the invariant I of S1–S3.

To apply GHL, a program's atomic operations must be specified. For a program represented by a flowchart, we assume that each box is an atomic operation. Let pc be the control state, so $pc = a_i$ means that the program is at control point a_i. Since for program π of Figure 1 $at(\pi) \equiv pc = a_1$ and $after(\pi) \equiv pc = a_4$, we have

$$\begin{aligned} Init &\equiv (pc = a_1) \wedge P_1 \\ Etern &\equiv (pc = a_4) \Rightarrow P_4 \\ AI &\equiv \bigwedge_{i=1}^{4} (pc = a_i \Rightarrow P_i). \end{aligned}$$

Clearly, $Init \Rightarrow AI$ and $AI \Rightarrow Etern$, so S1 and S3 are satisfied. To prove S2, the invariance of AI, we must prove the GHL formula $\{AI\}\ \pi\ \{AI\}$. By the Decomposition Principle, this can be reduced to proving the three formulas $\{AI\}\ \pi_1\ \{AI\}$, $\{AI\}\ \pi_2\ \{AI\}$, and $\{AI\}\ \pi_3\ \{AI\}$.

To prove $\{AI\}\ \pi_2\ \{AI\}$, we apply the Locality Rule, reducing it to the problem of proving $\{in(\pi_2) \wedge AI\}\ \pi_2\ \{after(\pi_2) \wedge AI\}$. We have

$$\begin{aligned} in(\pi_2) &\equiv pc = a_2 \\ after(\pi_2) &\equiv (pc = a_3 \vee pc = a_4), \end{aligned}$$

so $\{in(\pi_2) \wedge AI\}\ \pi_2\ \{after(\pi_2) \wedge AI\}$ is equivalent to

$$\{pc = a_2 \wedge P_2\}\ \pi_2\ \{(pc = a_3 \wedge P_3) \vee (pc = a_4 \wedge P_4)\}.$$

Since π_2 is atomic, this formula means that if executing the test π_2 moves control to a_3 (the "false" exit) then P_3 must hold, and if control moves to a_4 (the "true" exit) then P_4 must hold. Recall that this is just the verification condition for box π_2 in the Floyd method. Therefore, proving $\{AI\}\ \pi_2\ \{AI\}$ is equivalent to establishing the verification condition for box π_2. Similarly, proving $\{AI\}\ \pi_1\ \{AI\}$ and $\{AI\}\ \pi_3\ \{AI\}$ are equivalent to establishing the verification conditions for boxes π_1 and π_3. These three verification conditions can be proved formally in GHL using the axioms and inference rules in [13].

In this way, a proof by the Floyd method of the partial correctness of any program π with respect to precondition P and postcondition Q can be expressed in GHL. The GHL proof ultimately requires proving the same verification conditions as the Floyd method. We expect that every assertional method for reasoning about a program will reduce to proving these same verification conditions. However, this should not be surprising—it is the verification conditions that capture the semantics of the program.

3.2 The Owicki–Gries Method Reviewed

The Owicki–Gries method is a way of proving partial correctness for a concurrent program π of the form

π: **cobegin** π_1 // π // $\cdots$ // π_n **coend**, where the processes π_k communicate only by using shared memory [18]. Partial correctness of π with respect to a precondition P and a postcondition Q is proved in two steps.

(1) It is proved for each process π_k in isolation.
(2) These proofs are combined by establishing *noninterference*—execution of one process does not invalidate assertions in the proof of another.

Such a proof can be formulated in GHL as follows. The partial correctness proof for each process π_k in step (1) of the Owicki–Gries method is done essentially by the Floyd method. As shown above, this means that there is a predicate I_k such that

$$at(\pi_k) \wedge P \Rightarrow I_k \tag{3.2.1}$$

$$\{I_k\}\ \pi_k\ \{I_k\} \tag{3.2.2}$$

$$I_k \Rightarrow (after(\pi_k) \Rightarrow Q). \tag{3.2.3}$$

(These are just S1–S3 for π_k.)

In the GHL proof, the invariant I is the predicate $I_1 \wedge \cdots \wedge I_n$. Conditions S1–S3 for π are

$$at(\pi) \wedge P \Rightarrow I \tag{3.2.4}$$

$$\{I\}\ \pi\ \{I\} \tag{3.2.5}$$

$$I \Rightarrow (after(\pi) \Rightarrow Q). \tag{3.2.6}$$

We assume that at the entry point of π, control is at the entry points of all the π_k; and control reaches the exit of π when it is at the exits of all the π_k.[6] This means that

$$\begin{aligned} at(\pi) &\equiv at(\pi_1) \wedge \cdots \wedge at(\pi_n) \\ after(\pi) &\equiv after(\pi_1) \wedge \cdots \wedge after(\pi_n). \end{aligned}$$

From these relations, and the fact that (3.2.1) and (3.2.3) hold for all k, we obtain (3.2.4) and (3.2.6) directly.

To prove (3.2.5), we apply the Decomposition Principle, reducing the problem to showing

$$\{I\}\ \pi_k\ \{I\} \tag{3.2.7}$$

for all k. By the Conjunction Rule, (3.2.7) is established by proving the following for all i:

$$\{I_k \wedge I_i\}\ \pi_k\ \{I_k \wedge I_i\}. \tag{3.2.8}$$

For $k = i$, this is just (3.2.2). For $k \neq i$, (3.2.8) states that execution of π_k does not invalidate assertions in the proof of π_i, which is exactly the noninterference condition proved in step (2) of the Owicki–Gries method. This completes our formulation of the Owicki–Gries method in GHL.

[6] We could have introduced a more complicated control structure for the **cobegin**—for example, having a separate entry point at the beginning of π, before control "forks" to the beginning of the processes π_i. However, this would have added extra details without providing any further insight.

Again, the GHL proof involves the same verification conditions as the Owicki–Gries method. However, we feel that invariance is clearer than noninterference of proofs, so formulating a proof in GHL makes it easier to understand. Also, unlike the Owicki–Gries method, synchronization primitives do not require special proof rules in GHL. This is because they are represented by conditionally terminating atomic operations, so Floyd's method handles them quite easily.

3.3 CSP Reexplained

CSP [10] is intended for describing distributed programs. A CSP program π has the form

$$\pi :: [\pi_1 \parallel \pi_2 \parallel \cdots \parallel \pi_n]$$

where each π_i is a sequential process. Processes synchronize and communicate by using synchronous *input* and *output* commands. For notational simplicity, we consider a variant of CSP in which communication commands reference *channels* instead of other processes. Thus, to model CSP as described by Hoare, two unidirectional channels ξ_{ij} and ξ_{ji} connect each pair of processes π_i and π_j. Channel ξ_{ij} can be named only in output commands in π_i and input commands in π_j. For a channel ξ, an expression *expr*, and a variable *var*, execution of the output command

$$s\colon \xi\,!\,expr$$

by some process causes it to be delayed until some other process executes a *matching* input command

$$r\colon \xi\,?\,var.$$

The two commands are executed simultaneously as a single atomic action, causing the value of *expr* to be assigned to *var*. Thus, this matching pair of communications commands can be viewed as the "decentralized" atomic assignment statement

$$\langle var := expr \rangle,$$

which we label $r \leftarrow_\xi s$. Communication commands can occur as free-standing statements or in the guards of guarded commands.

There are two types of guarded commands in CSP: the alternative command and the repetitive command. Here, we consider only the repetitive command; treatment of alternative commands is similar. The syntax of a repetitive command *rpt* is

$$rpt\colon *[G_1 \rightarrow C_1 \,\square\, \cdots \,\square\, G_n \rightarrow C_n]$$

The command *rpt* contains an atomic operation geval_{rpt} that can perform the following action:

If control is at the entry point of *rpt*, then
- (a) if the value of some G_i is *true*, then it moves control to the entry point of C_i;
- (b) if the values of all the G_i are *false*, then it moves control to the exit point of *rpt*.

If more than one G_i has the value *true*, then the choice of i in (a) is nondeterministic.

A *Boolean guard* G_i consists of a Boolean expression, and its value is defined to be the value of that expression. If all the G_i are Boolean guards, then *rpt* is the same as Dijkstra's **do** statement [7]. However, we also allow *communication guards* of the form

$$B_i; comm_i$$

where B_i is a Boolean expression and $comm_i$ is a communication command.[7] The value of this guard is defined to equal *false* if B_i is false, and *maybe* if B_i is true. Thus, geval_{rpt} cannot be executed if some guards have the value *maybe* and all the rest have the value *false*. A communication guard G_i whose value is *maybe* may be executed by executing its communication command $comm_i$, causing control to move to the beginning of statement C_i. Of course, this requires simultaneously executing a matching communication command in another process.

In GHL, an invariance property $\{I\}\ \pi\ \{I\}$ for a CSP program π is proved as follows. We regard each channel ξ as a program fragment. The atomic operations of ξ consist of all operations $r \leftarrow_\xi s$ for matching communication commands s and r using channel ξ. Observe that $\alpha[\pi]$, the set of atomic operations of π, is given by

$$\alpha[\pi] = \alpha[\pi_1] \cup \cdots \cup \alpha[\pi_n] \cup \alpha[\xi_1] \cup \cdots \cup \alpha[\xi_m]$$

for processes $\pi_1 - \pi_n$ and channels $\xi_1 - \xi_m$. We partition π into two parts, π^{pr} and π^{ch} with

$$\alpha[\pi^{\text{pr}}] = \alpha[\pi_1] \cup \cdots \cup \alpha[\pi_n]$$

$$\alpha[\pi^{\text{ch}}] = \alpha[\xi_1] \cup \cdots \cup \alpha[\xi_m].$$

The Decomposition Principle can now be applied, so we need only prove

$$\{I\}\ \pi^{\text{pr}}\ \{I\} \tag{3.3.1}$$

$$\{I\}\ \pi^{\text{ch}}\ \{I\}. \tag{3.3.2}$$

The program fragment π^{pr} represents the concurrent program π without any communication over channels—that is, where processes can communicate only using shared variables. (The use of shared variables is prohibited in CSP, but that makes no difference in this discussion.) Therefore, (3.3.1) can be verified by using ordinary methods, such as the Owicki–Gries method. In π^{pr}, there are no atomic operations corresponding to communication commands ("!" and "?"). The operations that perform channel communications are in π^{ch}. In proving (3.3.1), these operations can be regarded as "halts"—when control reaches the entry point of such an operation, no further progress is possible. This explains the "strange and astonishing" rules A1 and A2′ of [1], the "miraculous" Receive Axiom of [19], and the corresponding Communication Axiom in [15].

To prove (3.3.1), we apply the Decomposition Principle to π^{pr}. In doing this, we decompose a repetitive command *rpt* as follows.

[7] CSP allows only input commands in guards, but that restriction is irrelevant here.

$$\alpha[rpt] \equiv \alpha[\text{geval}_{rpt}] \cup \alpha[C_1] \cup \cdots \cup \alpha[C_n].$$

To prove that *rpt* leaves I invariant, the Decomposition Principle requires that we prove $\{I\}\ C_1\ \{I\}, \ldots, \{I\}\ C_n\ \{I\}$, and $\{I\}\ \text{geval}_{rpt}\ \{I\}$. We already know how to prove $\{I\}\ C_i\ \{I\}$. To prove $\{I\}\ \text{geval}_{rpt}\ \{I\}$, we use the following GHL proof rules, where the term *pure predicate* refers to a predicate that does not contain references to *at*, *in* or *after*.

—If P is a pure predicate then $\{P\}\ \text{geval}_{rpt}\ \{P\}$.

—$\{true\}\ \text{geval}_{rpt}\ \{(\exists i\colon G_i = true \wedge at(C_i)) \vee (\forall i\colon G_i = false \wedge after(rpt))\}$.

The first proof rule states that evaluating the guards does not change the value of any program variable. The second proof rule is just a formal description of the operational semantics of the repetitive guarded command that was given above. Note that *maybe*'s do not appear there because control passes a communication guard only by an action of a channel, not by an action of geval_{rpt}.

To apply the Locality Rule for a repetitive command *rpt*, we must also know the following relations among the *at*, *in*, and *after* predicates of its components.

$$\begin{aligned} at(rpt) &\equiv at(\text{geval}_{rpt}) \\ after(\text{geval}_{rpt}) &\equiv at(C_1) \vee at(C_2) \vee \cdots \vee after(rpt) \\ after(C_i) &\equiv at(\text{geval}_{rpt}). \end{aligned}$$

The last rule says that the command iterates.

To prove (3.3.2), we apply the Decomposition Principle to π^{ch}, which requires proving

$$\{I\}\ \xi_j\ \{I\} \tag{3.3.3}$$

for each channel ξ_j. The atomic operations of ξ_j consist of all operations $r \leftarrow_{\xi_j} s$ for matching communication commands r and s using channel ξ_j. Applying the Decomposition Principle once again, proving (3.3.2) is reduced to proving

$$\{I\}\ r \leftarrow_{\xi} s\colon \langle var := expr\rangle\ \{I\} \tag{3.3.4}$$

for all matching commands r and s and channels ξ. The final reduction is to apply the Locality Rule to (3.3.4). To do this, we must understand the meaning of $in(r \leftarrow_{\xi} s)$ and $after(r \leftarrow_{\xi} s)$. For an atomic operation ρ, $in(\rho)$ means that the control state is such that ρ could be the next operation executed. Similarly, $after(\rho)$ means that the control state is such that ρ could have been the last operation executed. For free-standing communication commands r and s, the operation $r \leftarrow_{\xi} s$ can be executed only when control is at the entry point of both r and s, and its execution moves control to the exit points of r and s. Therefore,

$$\begin{aligned} in(r \leftarrow_{\xi} s) &\equiv at(r) \wedge at(s) \\ after(r \leftarrow_{\xi} s) &\equiv after(r) \wedge after(s), \end{aligned}$$

and we can apply the Locality Rule to (3.3.4), reducing it to

$$\{at(r) \wedge at(s) \wedge I\}\ r \leftarrow_{\xi} s\colon \langle var := expr\rangle\ \{after(r) \wedge after(s) \wedge I\} \tag{3.3.5}$$

For the case where r and/or s appears in a guard, the definition of $r \leftarrow_\xi s$ must be changed to reflect the fact that the Boolean part of a guard must be true for that guard's communication operation to be executed. For example, if r is in G_i of *rpt* and s is a free-standing output command, then, in the definition of $in(r \leftarrow_\xi s)$, $at(r)$ is replaced by $at(rpt) \wedge B_i = true$. Of course, $after(r) \equiv at(C_i)$, and (3.3.5) is changed accordingly.

These are the verification conditions for channel communications. They correspond to the cooperation proof in [1], the satisfaction proof and condition (3.4.2) in [15], and the similar conditions in [19]. The formal proof of (3.3.5) in GHL uses axioms and inference rules given in [13], where the operation $r \leftarrow_\xi s$ is treated as the simple atomic assignment $\langle var := expr \rangle$.

On the basis of the semantics given thus far, a repetitive command will terminate only when all Booleans in its guards are false. In Hoare's CSP, there is a weaker condition for termination, namely, that every guard is either false or names a channel connected to a process that has terminated. To capture this, we simply define the value of the guard

$$B_i; x_{ij}?\, val$$

to be *false* if $after(\pi_i)$, and define the value of the guard

$$B_i; x_{ij}!\, val$$

to be *false* if $after(\pi_j)$.

To illustrate this approach to CSP, consider the following trivial program, which sets variable u to the value of variable x.

$$\begin{aligned} \pi :: [\pi_1 &:: s_{12}\!:\xi_{12}!x \;\| \\ \pi_2 &:: rpt\!:*[r_{12}\!:\xi_{12}?y \rightarrow s_{23}\!:\xi_{23}!y] \;\| \\ \pi_3 &:: r_{23}\!:\xi_{23}?u] \end{aligned}$$

Our goal is to prove that if execution is started at the beginning of π_1, π_2, and π_3 then it can terminate only with $u = x$. We choose

$$\begin{aligned} Init &\equiv at(s_{12}) \wedge at(rpt) \wedge at(r_{23}) \\ I &\equiv (after(s_{12}) \Rightarrow x = y) \wedge (after(r_{23}) \Rightarrow y = u) \\ Etern &\equiv (after(s_{12}) \wedge after(rpt) \wedge after(r_{23})) \Rightarrow x = u. \end{aligned}$$

Note that $Init \Rightarrow I$, since $at(s_{12}) \Rightarrow \neg after(s_{12})$ and $at(r_{23}) \Rightarrow \neg after(r_{23})$; so S1 is satisfied. Also, $I \Rightarrow Etern$, so S3 is satisfied. To prove S2, the GHL formula $\{I\}\ \pi\ \{I\}$ must be proved. By the Decomposition Principle, this can be reduced to proving

$$\{I\}\ \pi_1\ \{I\} \tag{3.3.6}$$

$$\{I\}\ \pi_2\ \{I\} \tag{3.3.7}$$

$$\{I\}\ \pi_3\ \{I\} \tag{3.3.8}$$

$$\{I\}\ \xi_{12}\ \{I\} \tag{3.3.9}$$

$$\{I\}\ \xi_{23}\ \{I\}. \tag{3.3.10}$$

Formula (3.3.6) is obviously true because π_1 contains no atomic actions—the communications action is considered to be part of channel ξ_{12}. The proof of formula (3.3.7) follows because the only action in π_2 is geval$_{rpt}$. Executing geval$_{rpt}$ does not change the value of x, y, or u. Thus, $x = y$ and $y = u$ are not changed by π_2. The GHL proof rule for **cobegin** (rule P3(d) of [13]) states that no action in one process can affect the value of *at*, *in*, and *after* for other processes. Hence, the antecedents of the implications in I are also unaffected by executing geval$_{rpt}$ and so (3.3.7) is proved. In fact, maintaining this property of the **cobegin** rule is one reason why we have placed communications operations in channels instead of in processes. The proof of (3.3.8) is similar to that of (3.3.6).

To prove (3.3.9), first observe that channel ξ_{12} contains only one operation: $r_{12} \leftarrow_{\xi_{12}} s_{12}$. Thus, by applying the Decomposition Principle it is sufficient to prove

$$\{I\}\ r_{12} \leftarrow_{\xi_{12}} s_{12}\text{: } \langle x := y \rangle\ \{I\}.$$

The Locality Rule reduces this to

$$\{at(r_{12}) \land at(s_{12}) \land I\} r_{12} \leftarrow_{\xi_{12}} s_{12}\text{: } \langle x := y \rangle\ \{after(r_{12}) \land after(s_{12}) \land I\}$$

which follows directly from the GHL axiom for communication actions. The proof of (3.3.10) is similar to that of (3.3.9).

3.4 Other Methods

GHL can be used to reason about other interprocess communication mechanisms as well. For example, consider asynchronous message-passing, which can be viewed as a modification of CSP in which an output command ("!") does not wait for execution of a matching input command ("?"). In this case, the channel is a shared variable, rather than a program fragment. The asynchronous output command is an atomic operation to add a message to the channel and the corresponding input command is an atomic operation that removes a message.

The Decomposition Principle and GHL inference rules can be used to reduce the proof of invariance to elementary verification conditions, just as in the Owicki–Gries method. A proof method for programs that use such asynchronous message-passing is given in [19]. There, the state of the communications network is captured by using additional state variables (σ_D and ρ_D), not unlike the way we have used *at*, *in* and *after* to capture the state of program counters. In general, it may be necessary to add such variables when axiomatizing a programming construct.

4. CONCLUSIONS

In this paper, we have considered proof systems for reasoning about shared-memory communication and for reasoning about the synchronous message passing of CSP. In so doing, we hope to have removed the mystery from this plethora of proof systems. Our thesis is that concurrent programs should be understood in terms of invariance. We believe that it is better to think of a concurrent program as an "invariance maintainer" than as a "predicate transformer". To that end, GHL, a simple formal system for reasoning about invariance, was developed.

Viewing proof methods in terms of invariance is not new. Many researchers have realized that different proof techniques resulted from different decompositions of a global invariant, an idea first expressed explicitly by Cousot and Cousot [3, 4]. In [5], they suggested the decomposition of CSP programs into processes and channels. In contrast to the Cousots' work, which is based on program execution, our Decomposition Principle is based on syntactic decomposition of statements. It yields proof methods that are well suited to particular programming constructs.

The Decomposition Principle also suggests two principles for the design of a programming language. First, the syntactic components of the language should correspond to the logical components into which one decomposes a program when reasoning about it. The process construct is an example of such a syntactic component, processes being the logical components into which a program is decomposed when reasoning about a concurrent program. Secondly, the language should permit many properties of a program to follow naturally from the syntax. For example, the juxtaposition of statements s_1 and s_2 denotes $after(s_1) \Rightarrow at(s_2)$ in most programming notations.

Using GHL and the Decomposition Principle, which is really a meta-rule of GHL, it is possible to derive and compare other proof methods for reasoning about concurrent programs. Recent work of Cousot and Cousot [6] has shown that for any programming language in which flow of control can be expressed in terms of *at*, *in*, and *after*, GHL can be used to provide a sound and relatively complete axiom system for proving invariance properties. The *at*, *in*, and *after* predicates suffice to express the flow of control semantics in any concurrent programming language we know of. In addition, the Decomposition Principle provides useful insight for language designers.

ACKNOWLEDGMENTS

David Gries provided helpful comments on an early draft of this paper and Rebecca Bennett assisted in the preparation of the manuscript. Discussions with Patrick Cousot, Rick Hehner, and other members of IFIP W.G. 2.3 have been very helpful. The comments of the referees are also appreciated.

REFERENCES

1. APT, K. R., FRANCEZ, N., AND DE ROEVER, W. P. A proof system for communicating sequential processes. *ACM Trans. Prog. Lang. Syst. 2*, 3 (July 1980), 359–385.
2. ASHCROFT, E.A. Proving assertions about parallel programs. *J. Comput. Syst. Sci. 10* (Jan. 1975), 110–135.
3. COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (San Antonio, Tex. Jan. 29–31). ACM, New York, 1979, pp. 269–282.
4. COUSOT, P., AND COUSOT, R. Reasoning about program invariance proof methods. Tech. Rep. CRIN-80-P0, Centre de Recherche en Informatique de Nancy, Nancy, France, July 1980.
5. COUSOT, P., AND COUSOT, R. Semantic analysis of communicating sequential processes. In *Proceedings of the 7th International Colloquium on Automata, Languages and Programming, ICALP80*. Lecture Notes in Computer Science, vol. 85. Springer-Verlag, New York, 1980, pp. 119–133.
6. COUSOT, P., AND COUSOT, R. On the soundness and completeness of generalized Hoare logic.

Tech. Rep. CRIN-80-P093. Centre de Recherche en Informatique de Nancy, Nancy, France, Dec. 1982.

7. DIJKSTRA, E.W. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.
8. FLOYD, R.W. Assigning Meanings to Programs. In *Proceedings of the Symposium on Applied Mathematics*, vol. 19. American Mathematical Society, Providence, R.I., 1967, pp. 19–32.
9. HOARE, C.A.R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (Oct. 1969), 576–580.
10. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
11. KELLER, R.M. Formal verification of parallel programs. *Commun. ACM 19*, 7 (July 1976), 371–384.
12. LAMPORT, L. Proving correctness of multiprocess programs. *IEEE Trans. Softw. Eng. SE-3*, 2 (Mar. 1977), 125–143.
13. LAMPORT, L. The 'Hoare Logic' of concurrent programs. *Acta Inf. 14* (1980), 21–37.
14. LAMPORT, L. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr. 1983), 190–222.
15. LEVIN, G.M., AND GRIES, D. A proof technique for communicating sequential processes. *Acta Inf. 15* (1981), 281–302.
16. MANNA, Z., AND PNUELI, A. Verification of concurrent programs: temporal proof principles. In *Proceedings of the Workshop on Logics of Programs*, Lecture Notes in Computer Science, vol. 131. Springer-Verlag, New York, 1981, pp. 200–252.
17. OWICKI, S.S. Axiomatic proof techniques for parallel programs. Ph.D. dissertation, Dep. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1975.
18. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs. *Acta Inf. 6* (1976), 319–340.
19. SCHLICHTING, R.D., AND SCHNEIDER, F.B. Using message passing for distributed programming: Proof rules and disciplines. Tech. Rep., Dep. of Computer Science, Cornell Univ., Ithaca, N.Y.

Received May 1982; revised December 1982, July 1983; accepted August 1983