Reasoning About Nonatomic Operations

Leslie Lamport
Computer Science Laboratory
SRI International

ABSTRACT

A method is presented that permits assertional reasoning about a concurrent program even though the atomicity of the elementary operations is left unspecified. It is based upon a generalization of the dynamic logic operator $[\alpha]$. The method is illustrated by verifying the mutual exclusion property for a two-process version of the bakery algorithm.

1. Introduction

Assertional methods for reasoning about concurrent programs allow rigorous and, in principle, machine-verifiable correctness proofs [1],[4],[6],[7],[10],[11],[12]. All of these methods are based upon a model of program execution as a sequence of atomic operations. Although one could perhaps replace the totally ordered sequence by a partial ordering, the concept of atomicity is crucial. These methods do not allow one to reason about a concurrent program unless its atomic operations are specified.

This work was supported in part by the National Science Foundation under grant number MCS-8104459.

As an example, consider the trivial program of Figure 1, containing two concurrent processes that are completely independent of each other. It terminates with x and y both having the value one, regardless of how many separate atomic operations are used to implement the assignment statements. However, this cannot be proved directly by any previous assertional method; with these methods, one must first reason that the given program is equivalent to a different one in which the two assignment statements are atomic, and then prove the property for the new program. To reason about the program, one must first change it and demonstrate that the change is not significant.

```
cobegin
    x := 1  ▯  y := 1
coend
```

Figure 1. A Trivial Example

In this example, stating that the assertional methods cannot reason about a program with nonatomic operations may seem like a philosophical quibble, since it obviously makes no difference if the operations are atomic. We now consider a nontrivial example which clearly demonstrates that the inability to reason directly about nonatomic operations is a significant drawback. The example is the two-process version of the "bakery algorithm" for achieving mutual exclusion [5], which is given in Figure 2. The symbol $\underset{i,j}{<}$ is defined by

```
declare number  array 1..2 of integers  initially 0
declare choosing array 1..2 of booleans initially false

cobegin

  process 1: repeat forever
                 noncritical section;
                 choosing[1] := true;
                 number[1]   := 1 + number[2];
                 choosing[1] := false;
                 while choosing[2] do od;
                 while 0 < number[2] <(2,1) number[1] do od;
                 critical section;
                 number[1] := 0
             end repeat
           ▯
  process 2: repeat forever
                 noncritical section;
                 choosing[2] := true;
                 number[2]   := 1 + number[1];
                 choosing[2] := false;
                 while choosing[1] do od;
                 while 0 < number[1] <(1,2) number[2] do od;
                 critical section;
                 number[2] := 0
             end repeat

coend
```

Figure 2. The Two-Process Bakery Algorithm

$$x \underset{i,j}{<} y \equiv \begin{cases} x \leq y & \text{if } i < j \\ x < y & \text{if } i > j \end{cases}$$

The correctness criterion to be considered here is *mutual exclusion* – the two processes may not be in their critical sections at the same time.

This algorithm is correct regardless of how the operations are implemented by collections of atomic operations. For example, the fetching and storing of *number*[1] can be done one bit at a time. Thus, if process 2 reads *number*[1] while it is being changed from 0 to 100, then it could obtain the value 36 – or even the value 100,000. We give here a brief informal demonstration of the algorithm's correctness, referring the reader to [5] for a more detailed proof. Process i is said to be *in the doorway* from the time it has finished setting *choosing*[i] true until it has finished choosing a nonzero value of *number*[i] (in its third statement), and to be *in the bakery* from the time it has finished choosing a nonzero value of *number*[i] until it finishes its critical section. Thus, a process enters its doorway, chooses a nonzero number, and then enters the bakery. When process 1, executing its first **while** loop, finds *choosing*[2] false (just prior to exiting the loop), process 2 is not in its doorway. If 2 subsequently enters its doorway, then it will choose a value of *number*[2] greater than *number*[1]. Thus, from the time 1 finishes its first **while** loop until it leaves the bakery, if 2 is in the bakery then either:

- process 2 chose its current value of *number*[2] before process 1 finished its first **while** loop, or
- process 2 began choosing its current value of *number*[2] after process 1 finished its first **while** loop, so *number*[2] > *number*[1].

In the first case, process 1 must read the current value of *number*[1] when executing its second **while** loop. This implies that in either case, after 1 has finished its second **while** loop until it leaves the bakery: if 2 is also in the bakery then *number*[1] $\underset{1,2}{<}$ *number*[2]. The same reasoning with 1 and 2 interchanged shows that after 2 has exited its second **while** loop until it leaves the bakery: if 1 is also in the bakery then *number*[2] $\underset{2,1}{<}$ *number*[1]. Since *number*[1] $\underset{1,2}{<}$ *number*[2] and *number*[2] $\underset{2,1}{<}$ *number*[1] cannot both be true, we see that the two processes cannot be in their critical sections at the same time.

As convincing as this proof may appear – and a very convincing version of it appears in [5] – it is wrong, and the algorithm of Figure 2 contains an error. The proof tacitly assumes that process i always chooses a positive value of *number*[i] in its third statement. If process 1 could read a value of -1 for *number*[2] while process 2 was writing it, then the algorithm would not work. The algorithm is correct only with this extra assumption about the third statement. This illustrates the need for formal proofs; in fact, we discovered the error only in the course of writing a formal proof.

Previous methods of assertional reasoning cannot be used to formalize our argument because they require a knowledge of what the atomic operations are in order to prove anything about the program. Any particular implementation in terms of explicit atomic operations changes the program by disallowing certain forms of behavior. In [6], we tried to write a very general implementation by representing each assignment statement as two operations: the first setting the variable to an indeterminate value which, when read, can return any value, and the second setting it to its final value. However, this was obviously not the most general implementation because it excluded the possibility of the error mentioned above. It is better to reason directly about the program as given, with its atomic operations left unspecified, rather than trying to translate it into an equivalent program with explicit atomic operations. In this paper, we present a method for doing this. It allows a rigorous formulation of the two-process bakery algorithm's informal correctness proof.

Our method is based upon a new kind of predicate that is closely related to the predicate $[\alpha]P$ of dynamic logic [2]. (No knowledge of dynamic logic is needed to understand our method.) We will restrict our attention to proving safety (invariance) properties. However, the same predicates can be used with the methods of [11] for proving liveness (eventuality) properties.

2. Assertional Proofs

As explained in [9], the various assertional methods for proving safety properties of concurrent programs are basically the same. We will use essentially the formulation of [6] and [10], which is called the Gries-Owicki method. A *predicate* is a boolean-valued function of the program state, where the state includes all information relevant to the program's execution, including the values of variables and "program counters". An *annotation* of a program is an assignment of a predicate to each control point. We say that a control point is *active* in a state if some program counter has that control point as its value – *i.e.*, if some process is currently at that control point, and that a state is *consistent* with an annotation if the predicate assigned to each active control point is true. An annotation is *invariant* if, starting in any consistent state, executing a single atomic action of the program results in a consistent state. If the starting state of a program is consistent with an invariant annotation, then every state reached during its execution must be consistent. Safety properties are proved by finding an invariant annotation.

Proving the invariance of an annotation involves two steps for each atomic operation α:

- *Sequential Correctness*: Showing that if α is executed with the predicate attached to its entry point true, then when it terminates the predicate attached to its exit point is true.
- *Interference Freedom*: Executing α leaves the predicate attached to any other active control point true.

Unlike sequential programs, verifying nontrivial concurrent programs requires annotations with predicates whose values depend upon the control state. For this purpose, we use the following predicates, where ρ is any program statement.

$at(\rho)$: True if control is at the entry point of ρ.

$in(\rho)$: True if control is anywhere in ρ, including its entry point but excluding its exit point(s).

$after(\rho)$: True if control is at an exit point of ρ.

Letting P_α and Q_α be the predicates assigned to the entry and exit points of an atomic operation α, the verification conditions for α can be restated more precisely as follows:

- *Sequential Correctness*: $\{P_\alpha\}\,\alpha\,\{Q_\alpha\}$
- *Interference Freedom*: For each predicate R assigned to a control point ξ that can be active at the same time as the entry point of α:

$$\{P_\alpha \wedge R \wedge at(\alpha) \wedge active(\xi)\}\ \alpha\ \{R \wedge after(\alpha) \wedge active(\xi)\}$$

where $active(\xi)$ denotes the *at* or *after* predicate which asserts that control is at ξ.

Here, $\{P\}\,\alpha\,\{Q\}$ denotes the usual Floyd-Hoare input/output conditions for the atomic operation α.

A safety property is proved by showing the invariance of an appropriate annotation. For example, to prove that the trivial program of Figure 1, modified so the assignment statements are atomic, terminates with $x = y = 1$, we use the annotation of Figure 3. (Angle brackets enclose atomic operations; control points with no explicitly indicated predicate are assigned the predicate *true*.) Its invariance is proved by verifying the following four conditions.

cobegin
a: $\langle x := 1 \rangle$ $\{x = 1\}$
[]
b: $\langle y := 1 \rangle$ $\{y = 1\}$
coend $\{x = 1 \wedge y = 1\}$

Figure 3. Annotation of a Trivial Example

- $\{true\}\ \langle x := 1 \rangle\ \{x = 1\}$
- $\{true\}\ \langle y := 1 \rangle\ \{y = 1\}$
- $\{y = 1 \wedge at(a) \wedge after(b)\}\ \langle x := 1 \rangle\ \{y = 1 \wedge after(a) \wedge after(b)\}$
- $\{x = 1 \wedge at(b) \wedge after(a)\}\ \langle y := 1 \rangle\ \{x = 1 \wedge after(b) \wedge after(a)\}$

The program's starting state is obviously consistent with the annotation, so the program will remain in a consistent state throughout its execution. This means that when it reaches the end, the predicate $x = 1 \wedge y = 1$ assigned to its exit point must be true, which is the desired result.

As a more interesting example, we verify the mutual exclusion property of the bakery algorithm when each assignment and **while** test is atomic. The annotation is given in Figure 4, where writing a predicate in front of a nonatomic statement indicates that the predicate is to be assigned to all control points in the statement, excluding the exit point, and the predicates C_i, W_i, X_{ij} and $i \prec j$ are defined as follows.

$$C_i \equiv choosing[i] = true$$
$$W_i \equiv number[i] > 0$$
$$X_{ij} \equiv W_i \wedge [(at(bk_j) \wedge C_j) \supset$$

$$(number[i] \underset{ij}{\lesssim} number[j])]$$

$$i \prec j \equiv W_i \wedge [(j \text{ in bakery} \wedge W_j) \supset number[i] \underset{ij}{\lesssim} number[j]]$$

where i *in bakery* $\equiv at(bk_i) \vee at(w1_i) \vee at(w2_i) \vee in(cs_i)$

```
declare number  array 1..2 of integers  initially 0
declare choosing array 1..2 of booleans initially false

cobegin

 process 1: repeat forever
                 noncritical section;
                 (choosing[1] := true);
     {C1}  dw1: (number[1]  := 1 + number[2]);
 {C1 ∧ W1}  bk1: (choosing[1] := false);
     {W1}  w11: while (choosing[2]) do od;
     {X12} w21: while (0 < number[2] ≲21 number[1])
                   do od;
   {1 ≺ 2} cs1: critical section;
                 (number[1] := 0)
            end repeat

 process 2: repeat forever
                 noncritical section;
                 (choosing[2] := true);
     {C2}  dw2: (number[2]  := 1 + number[1]);
 {C2 ∧ W2}  bk2: (choosing[2] := false);
     {W2}  w12: while (choosing[1]) do od;
     {X21} w22: while (0 < number[1] ≲12 number[2])
                   do od;
   {2 ≺ 1} cs2: critical section;
                 (number[2] := 0)
            end repeat

coend
```

Figure 4. An Atomized Version of the Two-Process Bakery Algorithm

We urge the reader to write down the verification conditions – especially the ones for statements dw_1 and dw_2 – and convince himself that they are satisfied. In doing so, he will make use of the tacit assumption that atomic actions in the critical and noncritical sections do not modify any element of the arrays *choosing* and *number*.

The initial state of the program, in which each process is in its noncritical section, is obviously consistent with this annotation. The invariance of the annotation therefore means that $1 \prec 2$ is true whenever process 1 is in its critical section, and $2 \prec 1$ is true whenever process 2 is in its critical section. This easily implies that the two processes cannot both be in their critical sections, which is the desired result.

The Gries-Owicki method can be derived as an application of the generalized Hoare Logic (GHL) introduced in [7]. In GHL, the assertion $\{P\}\,\pi\,\{Q\}$ means:

> If any single atomic action of π is executed with P true, then either:
>
> (i) control remains in π and P remains true, or
>
> (ii) control reaches the exit point of π and Q becomes true.

This generalizes the ordinary Hoare logic because if π consists of a single atomic operation, then $\{P\}\,\pi\,\{Q\}$ asserts the partial correctness of π with respect to the input predicate P and the output predicate Q.

It is convenient to view an annotation of a program π as the GHL formula $\{P\}\,\pi\,\{Q\}$ in which Q is the predicate written at the exit point of π and P is the conjunction of all predicates

$$(in(\rho) \supset P_\rho) \wedge (after(\rho) \supset Q_\rho) \quad ,$$

where P_ρ and Q_ρ are the predicates written in front of and at the exit point of statement ρ. This GHL formula asserts the invariance of the annotation. As explained in [9], the Gries-Owicki method and other assertional methods can be viewed as a way of breaking the verification of a GHL assertion $\{P\}\,\pi\,\{Q\}$ into simpler subproblems using the following rules:

Decomposition Principle: If each atomic operation of π is an atomic operation of one of the π_i, then

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

Locality Rule:

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

Conjunction Rule:

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

The Disjunction Rule – obtained by replacing $\wedge$ by $\vee$ in the Conjunction Rule – is also valid, as is the converse of the Locality Rule, which follows from the Conjunction Rule and the axiom $\{in(\pi)\}\,\pi\,\{after(\pi)\}$.

These rules imply that to verify the GHL assertion corresponding to an annotation of a program π, we must verify the following two conditions for every elementary operation ρ of π, where P_ρ is the conjunction of all predicates written in front of ρ or in front of some statement containing ρ, and Q_ρ is the conjunction of all predicates similarly "attached" to the exit point of ρ.

- *Sequential Correctness*: $\{P_\rho\}\,\rho\,\{Q_\rho\}$
- *Interference Freedom*: For each predicate R_ξ written in front of or after a statement ξ in a different clause of a **cobegin** that contains ρ:

$$\{P_\rho \wedge R \wedge in(\rho) \wedge active(\xi)\}\ \rho\ \{R \wedge after(\rho) \wedge active(\xi)\}$$

where $active(\xi)$ denotes either $in(\xi)$ or $after(\xi)$.

If each elementary operation ρ is atomic, then these are just the Floyd-Hoare verification conditions of the Gries-Owicki method. The method described here will permit the reduction to stop at elementary nonatomic operations, such as the entire assignment statement

$$number[1] := 1 + number[2]$$

in the bakery algorithm.

3. Generalized Dynamic Logic

The basis of our method is a new class of predicates that generalize ones used in dynamic logic [2]. For a predicate Q and a program statement ρ, we define $[\rho]\,Q$ to be the predicate that is true if and only if either:

- $Q \wedge \sim in(\rho)$ is true, or
- $in(\rho)$ is true, and any execution of ρ starting in the current state, with no other program statement being executed concurrently, either:
  - fails to reach an exit point of ρ, or
  - reaches an exit point of ρ with Q true.

It is important to realize that $[\rho]\,Q$ is a predicate – a boolean function of the program state. If s is a state for which control is in ρ, then saying that $[\rho]\,Q$ is true for s asserts what would happen if ρ were to be executed starting in the state s. It does not assert that ρ will be executed.

As an example, consider the following program statement ρ.

```
ρ: begin
     α:⟨ temp := x ⟩;
     β:⟨ x := temp + 1 ⟩;
   end
```

For this statement, $[\rho]\,(x > 5)$ is the predicate

$$(\sim in(\rho) \supset x > 5) \wedge (at(\alpha) \supset x > 4) \wedge (at(\beta) \supset temp > 4)\ .$$

The usual Floyd-Hoare input/output condition, denoted $P\,\{\rho\}\,Q$ as in [3], is equivalent to $P \wedge at(\rho) \supset [\rho]\,Q$. It follows from this that the predicate denoted $[\rho]\,Q$ in ordinary dynamic logic is equivalent to the predicate that we write $at(\rho) \wedge [\rho]\,Q$. Just as $\{P\}\,\rho\,\{Q\}$ generalizes the ordinary Hoare assertion $P\,\{\rho\}\,Q$ by considering executions of ρ started with control not at its entry point, so $[\rho]\,Q$ generalizes the corresponding dynamic logic predicate by considering states in which control is not at the entry point of ρ.

We now develop methods for proving GHL assertions $\{P\}\,\rho\,\{Q\}$ when P and Q are generalized dynamic logic predicates. First, we state some axioms that are immediate consequences of the definition of $[\rho]\,Q$.

Dynamic Logic Axioms:

1. $\{[\rho]\,Q\}\ \rho\ \{[\rho]\,Q\}$
2. $([\rho]\,Q) \wedge \sim in(\rho) \equiv Q \wedge \sim in(\rho)$
3. $[\rho]\,(P \wedge Q) \equiv ([\rho]\,P) \wedge ([\rho]\,Q)$
4. $[\rho]\,(P \supset Q) \supset ([\rho]\,P \supset [\rho]\,Q)$

Since $after(\rho) \supset \sim in(\rho)$, combining the second axiom with the Locality Rule shows that the first axiom can also be written as

$$\{[\rho]\,Q\}\ \rho\ \{Q\}\quad .$$

We also have the following obvious inference rule.

Tautology Rule:

$$\frac{Q}{[\rho]\,Q}$$

We next formally state the relation between the ordinary Hoare logic formula $P\,\{\rho\}\,Q$ and the generalized dynamic logic predicate $[\rho]\,Q$ that we mentioned above.

Hoare Rule:

$$\frac{P\,\{\rho\}\,Q}{P \wedge at(\rho) \supset [\rho]\,Q}$$

For our next rules, we need the concept of *noninterference* that is often used but is seldom formalized. We assume that a program state can be decomposed into separate components, and say that a statement ρ *does not interfere with* a predicate P if ρ does not change any state component upon which the value of P depends. For example, ρ does not interfere with the predicate $at(\sigma) \supset x > 0$ unless executing ρ can change the value of x or the value of $at(\sigma)$. For simple languages, executing ρ can change the value of $at(\sigma)$ only if σ is ρ or the statement immediately following ρ. More complicated languages may allow more exotic possibilities – for example, executing ρ might abort the process containing σ.

In general, noninterference can depend upon the state. For example, the statement

$$number[i] := 0$$

does not interfere with the predicate $number[1] > 0$ if $i \neq 1$. We therfore define $\rho \rightarrowtail P$ (read "ρ does not interfere with P") to be the predicate that is true for a state s if any atomic operation of ρ executed starting in state s does not change any component of the state upon which the value of P depends.

We also need the concept of one statement not interfering with another, where $\rho \rightarrow\!\!| \sigma$ means that ρ does not change any part of the state which influences the behavior of σ. For example, if ρ is the statement x := 0 and σ is the statement x := y, then $\rho \rightarrow\!\!| \sigma$. However, the statement y := 0 does interfere with σ. To define this more precisely, we let $s \xrightarrow{\alpha} t$ denote that executing the atomic operation α starting in state s can yield state t. We then define $\rho \rightarrow\!\!| \sigma$ to be the predicate that is true for a state s if the following condition holds.

> For any atomic operations α in ρ and β in σ, and any state s: if $s \xrightarrow{\alpha} t$, then executing β starting in state t does the same thing as executing it starting in state s.

Note that "doing the same thing" does not mean producing the same state, since t can reflect changes made by executing α that do not affect the behavior of σ – for example, if α is $\langle x := 0 \rangle$ and β is $\langle y := 0 \rangle$.

Our definitions of the predicates $\rho \rightarrow\!\!| P$ and $\rho \rightarrow\!\!| \sigma$ have been quite informal. They will suffice for our examples. In a more formal approach, one would give axioms and inference rules for deriving valid $\rightarrow\!\!|$ relations for particular programming and assertion languages. One such axiom might be

$$(\rho \rightarrow\!\!| P) \wedge (\rho \rightarrow\!\!| Q) \supset \rho \rightarrow\!\!| (P \vee Q)$$

We will not develop such a formalism, and will just use our informal definitions to justify the $\rightarrow\!\!|$ relations needed for the examples.

The concept of noninterference appears in the following rules.

Noninterference Rules:

1. $$\frac{P \supset \rho \rightarrow\!\!| P}{\{P\}\,\rho\,\{P\}}$$

2. $$\frac{\rho \rightarrow\!\!| P}{[\rho]P \equiv P \vee [\rho]\mathit{false}}$$

3. $$\frac{\rho \rightarrow\!\!| P,\ \rho \rightarrow\!\!| \sigma}{\rho \rightarrow\!\!| ([\sigma]P)}$$

The validity of the first rule is clear. It is used repeatedly when applying the Gries-Owicki method, being tacitly invoked whenever a verification condition is dismissed as being "trivial". The verification condition $\{P\}\,\rho\,\{P\}$ is trivial when ρ does not modify any of the variables or control points that appear in P – *i.e.*, when $\rho \rightarrow\!\!| P$ is a valid predicate (true for all states). In the Gries-Owicki method, the number of verification conditions that arise in checking noninterference is quadratic in the size of the program. The method is practical only because most of these verifications are trivial, so they follow from the first Noninterference Rule.

The validity of the second rule follows from the observation that $\rho \rightarrow\!\!| P$ implies that if ρ terminates when executed from state s, then the value of P for the final state equals its value for s. Using the Dynamic Logic Axioms, one can show that this rule is equivalent to the following:

2'. $$\frac{\rho \rightarrow\!\!| P}{P \wedge [\rho]Q \equiv P \wedge [\rho]P \wedge Q}$$

The validity of the third Noninterference Rule is demonstrated as follows. The hypothesis $\rho \rightarrow\!\!| \sigma$ implies that executing any atomic operation of ρ cannot change the behavior of σ, so it cannot affect whether or not a particular execution sequence reaches an exit point, and the hypothesis $\rho \rightarrow\!\!| P$ implies that executing that atomic operation cannot change the value of P when an exit point is reached. The conclusion then follows from the definition of $[\sigma]P$.

Using the Noninterference Rules, we can prove the noninterference condition $\{[\sigma]P\}\ \rho\ \{[\sigma]P\}$ when ρ does not interfere with σ and P. However, one may want to prove such a GHL formula under the weaker hypothesis that ρ leaves P invariant – that is, when $\{P\}\,\rho\,\{P\}$ holds. To infer $\{[\sigma]P\}\ \rho\ \{[\sigma]P\}$ under the hypothesis $\{P\}\,\rho\,\{P\}$, we need a stronger hypothesis than $\rho \rightarrow\!\!| \sigma$. A counterexample is provided by the following choices of ρ, σ and P:

ρ: $\langle x := y \rangle$

σ: $\langle x := 0 \rangle$

P: $(at(\rho) \supset y > 7) \wedge (after(\rho) \supset x > 7)$

In this case, both $\rho \rightarrow\!\!| \sigma$ and $\sigma \rightarrow\!\!| \rho$ as well as $\{P\}\,\rho\,\{P\}$ hold, but $\{[\sigma]P\}\ \rho\ \{[\sigma]P\}$ does not hold because

$$[\sigma]P \wedge at(\sigma) \equiv (at(\rho) \supset y > 7) \wedge (\sim at(\rho) \supset 0 > 7) \wedge at(\sigma),$$

so executing ρ in a state with $at(\rho)$, $at(\sigma)$ and $y > 7$ all true changes the value of $[\sigma]P$ from *true* to *false*.

The inference rules that we want require some new definitions. For atomic operations α and β, we say that α *right commutes* with β if the following condition is satisfied:

> For any states s and t: if $s \xrightarrow{\alpha} t$ and $t \xrightarrow{\beta} u$, then there is a state t' such that $s \xrightarrow{\beta} t'$ and $t' \xrightarrow{\alpha} u$.

In other words, if $s \xrightarrow{\alpha} t \xrightarrow{\beta} u$ then we can "move the α action to the right of the β action", obtaining $s \xrightarrow{\beta} t' \xrightarrow{\alpha} u$. The semaphore operation $\langle P(s) \rangle$ right commutes with the semaphore operation $\langle V(s) \rangle$, but $\langle V(s) \rangle$ does not right commute with $\langle P(s) \rangle$.

For arbitrary statements ρ and σ, we say that ρ right commutes with σ if every atomic operation of ρ right commutes with every atomic operation of σ. They are said simply to *commute* if each right commutes with the other. A sufficient condition for ρ and σ to commute is that neither

accesses or changes any state component accessed or changed by the other. (Note that this is stronger than the condition that neither interferes with the other.) However, this condition is not necessary, since the two atomic operations $\langle x := x+1 \rangle$ and $\langle x := x+2 \rangle$ commute.

The concept of right-commutativity is introduced for the following rule:

Right-Commutativity Rule:

$$\frac{\{P \vee in(\sigma)\}\,\rho\,\{P \vee in(\sigma)\},\ \rho \text{ right commutes with } \sigma}{\{[\sigma]P\}\,\rho\,\{[\sigma]P\}}$$

The validity of this rule is demonstrated as follows. By definition of the GHL assertion $\{Q\}\,\rho\,\{Q\}$, to prove $\{[\sigma]P\}\,\rho\,\{[\sigma]P\}$ it suffices to show that for any atomic operation α of ρ and any states s and t such that $s \xrightarrow{\alpha} t$: if $[\sigma]P$ is false for t, then it is false for s. By definition, $[\sigma]P$ is false for t if there exists a (possibly null) sequence of states t_i and atomic operations β_i of σ such that

$$t \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_n} t_n \quad ,$$

and, in state t_n, P is false and control is not in σ. (If $\sim in(\sigma)$ holds for t, then $n = 0$ and $t_0 = t$.) In other words, $\sim(P \vee in(\sigma))$ holds for t_n. By right-commutativity, we can move the α action to the right of the β_i actions to obtain

$$s \xrightarrow{\beta_1} t' \xrightarrow{\beta_2} t'_1 \cdots \xrightarrow{\beta_n} t'_{n-1} \xrightarrow{\alpha} t_n$$

Since $\sim(P \vee in(\sigma))$ holds for t_n, the hypothesis

$$\{P \vee in(\sigma)\}\,\rho\,\{P \vee in(\sigma)\}$$

implies that $\sim(P \vee in(\sigma))$ must hold for t'_{n-1}, which implies that $[\sigma]P$ is false for s, proving the validity of the rule.

When applying the Right-Commutativity Rule, ρ and σ will be in different processes, so $\{in(\sigma)\}\,\rho\,\{in(\sigma)\}$ will hold (assuming ρ cannot abort the execution of σ). Hence, the hypothesis $\{P \vee in(\sigma)\}\,\rho\,\{P \vee in(\sigma)\}$ follows from $\{P\}\,\rho\,\{P\}$ and the Disjunction Rule.

Our last inference rule is the following.

Commutativity Rule:

$$\frac{\rho \text{ and } \sigma \text{ commute}}{[\rho]([\sigma]P) \equiv [\sigma]([\rho]P)}$$

Its validity is proved by using commutativity to show that if $[\rho]([\sigma]P)$ is false for a state, then $[\sigma]([\rho]P)$ is also false, and *vice-versa*. The details are left to the reader. We will not use these commutativity rules in our examples, but we expect them to be useful for reasoning about some programs.

4. Examples

In annotating a program, we let $[\rho]\,\{Q\}$ be an abbreviation for $\{[\rho]Q\}\,\rho\,\{Q\}$. Returning to the trivial example of Figure 1, to prove that $x = 1$ and $y = 1$ when it terminates, we use the annotation of Figure 5. Applying the Decomposition Principle and the Locality and Conjunction Rules of GHL, verifying the GHL formula corresponding to this annotation is reduced to verifying the following conditions.

1. $\{[a](x = 1)\}\ x := 1\ \{x = 1\}$
2. $at(a) \supset [a](x = 1)$
3. $\{[b](y = 1)\}\ y := 1\ \{y = 1\}$
4. $at(b) \supset [b](y = 1)$
5. $\{[b](y = 1)\}\ x := 1\ \{[b](y = 1)\}$
6. $\{[a](x = 1)\}\ y := 1\ \{[a](x = 1)\}$

Conditions 1 and 3 follow from Dynamic Logic Axiom 1, conditions 2 and 4 follow from the Hoare Rule, and conditions 5 and 6 follow from the first Noninterference Rule.

Observe that in applying the Noninterference Rule, we are making the tacit assumption that neither of the two assignment statements modifies any variables used in the other. For example, the following implementation of the statement x := x+1 is disallowed, even though it would be a perfectly valid implementation in a sequential program.

$$\langle x := y+1 \rangle\ ;\ \langle x := x-y \rangle$$

cobegin
a: $[x := 1]$ $\{x = 1\}$
[]
b: $[y := 1]$ $\{y = 1\}$
coend $\{x = 1 \wedge y = 1\}$

Figure 5. Annotation of a Trivial Example

Comparing these six verification conditions with the ones for the version in Figure 3, we see that they are the same except for the two extra conditions that are immediate consequences of the Dynamic Logic Axiom 1. We used the Noninterference Rule instead of replacing the nonatomic statements by atomic ones. The same thing will happen for any program in which nonatomic operations can be replaced by equivalent atomic ones - our method allows us to apply essentially the same reasoning used for the atomic version directly to the original program. Although the actual verification conditions are the same, we find it more elegant to reason about the given program instead of changing it.

declare *number* **array** 1..2 **of** integers **initially** 0
declare *choosing* **array** 1..2 **of** booleans **initially** *false*

cobegin

process 1: **repeat forever**
noncritical section;
st_1: [*choosing*[1] := *true*] $\{C_1\}$;
$\{C_1\}$ $d1_1$: *number*[1] := 1 + *number*[2];
$\{C_1\}$ $d2_1$: [*number*[1] := maximum(1, *number*[1])]
$\{C_1 \wedge W_1\}$;
$\{W_1\}$ bk_1: *choosing*[1] := *false*;
$\{W_1\}$ $w1_1$: [**while** *choosing*[2] **do od**] $\{X_{12}\}$;
$\{X_{12}\}$ $w2_1$: [**while** $0 < number[2] \underset{21}{\leq} number[1]$ **do od**] $\{1 \prec 2\}$;
$\{1 \prec 2\}$ cs_1: *critical section*;
ex_1: *number*[1] := 0
end repeat

[]

process 2: **repeat forever**
noncritical section;
st_2: [*choosing*[2] := *true*] $\{C_2\}$;
$\{C_2\}$ $d1_2$: *number*[2] := 1 + *number*[1];
$\{C_2\}$ $d2_2$: [*number*[2] := maximum(1, *number*[2])]
$\{C_2 \wedge W_2\}$;
$\{W_2\}$ bk_2: *choosing*[2] := *false*;
$\{W_2\}$ $w1_2$: [**while** *choosing*[1] **do od**] $\{X_{21}\}$;
$\{X_{21}\}$ $w2_2$: [**while** $0 < number[1] \underset{12}{\leq} number[2]$ **do od**] $\{2 \prec 1\}$;
$\{2 \prec 1\}$ cs_2: *critical section*;
ex_2: *number*[2] := 0
end repeat

coend

Figure 6. Annotation of the Two-Process Bakery Algorithm

We now consider the nontrivial example of the two-process bakery algorithm. The algorithm and the annotation used to prove the mutual exclusion property are given in Figure 6. Notice that the original third statement of process i, which appeared as statement dw_i in the atomized version of Figure 4, has been implemented as the the concatenation of the two statements $d1_i$; $d2_i$ to remove the error mentioned in the introduction. We let dw_i denote the concatenation of these two statements. In this annotation, the definitions of C_i and W_i are the same as before, but X_{ij} and $i \prec j$ are redefined as follows.

$$X_{ij} \equiv W_i \wedge [(in(dw_j) \wedge C_j) \supset [dw_j](number[i] \underset{ij}{\leq} number[j])]$$

$$i \prec j \equiv X_{ij} \wedge [(j \text{ in bakery} \wedge W_j) \supset (number[i] \underset{ij}{\leq} number[j])]$$

where $i \text{ in bakery} \equiv in(bk_i) \vee in(w1_i) \vee in(w2_i) \vee in(cs_i)$

To prove the GHL formula corresponding to this annotation, we apply the GHL rules to reduce the problem to proving a number of individual verification conditions. The ones corresponding to the sequential correctness conditions of the Gries-Owicki method, obtained by considering the annotation of each process separately, are proved using the Dynamic Logic Axioms, the Noninterference Rules and the Hoare Rule. For example, we get the following correctness conditions for statement $w1_1$:

$$\{W_1 \wedge [w1_1]X_{12}\}\ w1_1\ \{[w1_1]X_{12}\}$$

$$(W_1 \wedge at(w1_1)) \supset [w1_1]X_{12}$$

The first condition follows from Dynamic Logic Axiom 1, the Conjunction Rule and Noninterference Rule 1, since $w1_1 \rightarrow\!| W_1$. The second follows from the Hoare Rule, since

$$W_1\,\{w1_1\}\,(W_1 \wedge [(A \wedge C_2) \supset B])$$

holds for any predicates A and B.

The annotation of process 1 contains the following predicates:

(a) $[st_1]C_1$

(b) $C_1 \wedge [d2_1](C_1 \wedge W_1)$

(c) W_1

(d) $W_1 \wedge [w1_1]X_{12}$

(e) $X_{12} \wedge [w2_1](1 \prec 2)$

(f) $1 \prec 2$

To prove interference freedom, we must show that each of the nine elementary statements of process 2 leaves these six predicates invariant. This gives 54 verification conditions to be checked - plus the symmetrical 54 conditions to show that process 1 leaves invariant the predicates in the annotation of process 2. No statement of process 2 interferes with predicates (a), (b) or (c), so the Noninterference Rules imply that they are left invariant by process 2. The only verification conditions that do not follow immediately from the Noninterference Rules are that statements st_2, $d1_2$, $d2_2$ and ex_2 leave (d), (e) and (f) invariant. We consider $d1_2$ and $d2_2$ together as the single statement dw_2. The corresponding GHL formulas that must be verified are then:

(d)-st_2: $\{W_1 \wedge [w1_1]X_{12}\}\ st_2\ \{W_1 \wedge [w1_1]X_{12}\}$

(e)-st_2: $\{X_{12} \wedge [w2_1](1 \prec 2)\}\ st_2\ \{X_{12} \wedge [w2_1](1 \prec 2)\}$

(f)-st_2: $\{1 \prec 2\}\ st_2\ \{1 \prec 2\}$

(d)-dw_2: $\{C_2 \wedge W_1 \wedge [w1_1]X_{12}\}\ dw_2\ \{W_1 \wedge [w1_1]X_{12}\}$

(e)-dw_2: $\{C_1 \wedge X_{12} \wedge [w2_1](1 \prec 2)\}\ dw_2\ \{X_{12} \wedge [w2_1](1 \prec 2)\}$

(f)-dw_2: $\{C_1 \wedge (1 \prec 2)\}\ dw_2\ \{1 \prec 2\}$

(d)-ex_2: $\{C_2 \wedge W_1 \wedge [w1_1]X_{12}\}\ ex_2\ \{W_1 \wedge [w1_1]X_{12}\}$

(e)-ex_2: $\{C_1 \wedge X_{12} \wedge [w2_1](1 \prec 2)\}\ ex_2\ \{X_{12} \wedge [w2_1](1 \prec 2)\}$

(f)-ex_2: $\{1 \prec 2\}\ ex_2\ \{1 \prec 2\}$

The reader will find it a good exercise in understanding generalized dynamic logic predicates to convince himself of the intuitive validity of these formulas. As is often the case, proving things from basic axioms and inference rules can be quite tiresome. We therefore prove only two of these formulas. The proofs of the rest are similar, and are left to the reader. The proof of the first formula is as follows.

$$
\begin{aligned}
&(W_1 \wedge [w1_1]X_{12}) \wedge in(st_2) \\
&\quad \equiv [w1_1](in(st_2) \wedge X_{12}) \\
&\qquad \text{[by Noninterference Rule 2', since } W_1 \wedge X_{12} \equiv X_{12}\text{]} \\
&\quad \equiv [w1_1](W_1 \wedge in(st_2)) \\
&\qquad \text{[since } in(st_2) \wedge in(dw_2) \equiv false\text{]} \\
&\quad \equiv (W_1 \wedge in(st_2)) \vee ([w1_1]false) \\
&\qquad \text{[by the Tautology Rule]}
\end{aligned}
$$

$$
\begin{aligned}
&(W_1 \wedge [w1_1]X_{12}) \wedge after(st_2) \\
&\quad \equiv [w1_1](after(st_2) \wedge X_{12}) \\
&\quad \equiv [w1_1](W_1 \wedge at(dw_2) \wedge (C_2 \supset \\
&\qquad\qquad [dw_2](number[1] \underset{1\ 2}{<} number[2]))) \\
&\qquad \text{[since } after(st_2) \equiv at(dw_2)\text{]} \\
&\quad \equiv [w1_1](W_1 \wedge at(dw_2)) \\
&\qquad \text{[since by the Hoare Rule,} \\
&\qquad\ at(dw_2) \supset [dw_2](number[1] \underset{1\ 2}{<} number[2])\text{]} \\
&\quad \equiv (W_1 \wedge after(st_2)) \vee [w1_1]false \\
&\qquad \text{[by Noninterference Rule 2', since } at(dw_2) \equiv after(st_2)\text{]}
\end{aligned}
$$

Applying the Locality and Disjunction Rules, we see that it suffices to verify

$$\{W_1 \wedge in(st_2)\}\ st_2\ \{W_1 \wedge after(st_2)\}$$

$$\{[w1_1]false\}\ st_2\ \{[w1_1]false\},$$

both of which follow easily from the Noninterference Rules.

We verify condition (d)$-dw_2$ as follows.

$$
\begin{aligned}
&(C_2 \wedge W_1 \wedge [w1_1]X_{12}) \wedge in(dw_2) \\
&\quad \equiv [w1_1](W_1 \wedge in(dw_2) \wedge C_2 \wedge \\
&\qquad\qquad [dw_2](number[1] \underset{1\ 2}{<} number[2])) \\
&\quad \equiv (W_1 \wedge in(dw_2) \wedge C_2 \wedge [dw_2](number[1] \underset{1\ 2}{<} \\
&\qquad\qquad number[2])) \vee [w1_1]false \\
&\qquad \text{[by Noninterference Rule 2]}
\end{aligned}
$$

$$
\begin{aligned}
&(C_2 \wedge W_1 \wedge [w1_1]X_{12}) \wedge after(dw_2) \\
&\quad \equiv [w1_1](C_2 \wedge W_1 \wedge after(dw_2)) \\
&\qquad \text{[since } after(dw_2) \wedge in(dw_2) \equiv false\text{]} \\
&\quad \equiv (C_2 \wedge W_1 \wedge after(dw_2)) \vee [w1_1]false \\
&\qquad \text{[by Noninterference Rule 2]}
\end{aligned}
$$

The proof now continues in the same way as in the proof of condition (d)$-st_2$.

5. Discussion

We have presented a method for assertional reasoning about a program whose implementation in terms of atomic operations is not specified. It is based upon a generalization of dynamic logic. If making the program's elementary operations atomic produces an equivalent program, then the verification steps are essentially the same as for the usual method of reasoning about the "atomized" program. In this case, the advantages of our method are largely aesthetic. However, previous methods cannot be applied if making the elementary operations atomic changes the program, and our method is the first one to allow rigorous assertional reasoning about such a program.

Our formal approach led us to discover a previously unrecognized assumption required for the correctness of the bakery algorithm – an algorithm we had studied very carefully and verified in two different ways. The formal proof is somewhat tedious, but tedium seems to be an inevitable byproduct of formal rigor.

Although we have presented the logical foundation for reasoning about programs with nonatomic elementary operations, we have not discussed how such operations can be specified. In the n-process bakery algorithm, process i chooses its value of $number[i]$ by the statement

$$number[i] := 1 + \text{maximum}(number[1], \ldots, number[n]).$$

For $n > 2$ we need the following additional assumption about this statement.

> If $number[j]$ does not change during execution of the statement, then $number[i]$ is chosen to be greater than $number[j]$ – regardless of the concurrent activity of other processes.

Thus, the following is not a valid implementation if the two instances of $number[k]$ represent separate fetches.

```
temp[i] := 0 ;
for k := 1 to n
    do if number[k] > temp[i]
          then temp[i] := number[k] od ;
number[i] := 1 + temp[i]
```

An obvious way of specifying this requirement assertionally is to state that process k must leave invariant the predicate $[dw_i](number[i] > number[j])$ for $k \neq i, j$. However, this condition is too strong to be met by any reasonable nonatomic implementation. For example, consider the state reached when

1. Process 1 begins choosing $number[1]$ starting in a state with $number[2] = 0$ and $number[3] = 17$.
2. Process 1 reads $number[2]$ and finds it equal to zero.
3. Process 2 sets $number[2]$ to 17.

At this point, before process 1 reads $number[3]$,

$$[dw_1](number[1] > number[2])$$

is true. However, process 3 can make it false by setting $number[3]$ to zero.

The correct specification states that there exists a predicate P_{ij} such that:

(i) $P_{ij} \supset [dw_i](number[i] > number[j])$, and

(ii) Process k leaves P_{ij} invariant.

Intuitively, the predicate P_{ij} asserts that when process i is choosing $number[i]$, if it has not yet read $number[j]$ then it will choose a value of $number[i]$ greater than the current value of $number[j]$. This illustrates the important point that to specify a statement without describing its atomic operations, one must write a second-order formula of the form: "There exist state functions ... such that ...." The general problem of specifying concurrent programs is discussed in [8].

REFERENCES

[1] E. A. Ashcroft. Proving Assertions About Parallel Programs. *J. Comput. Sys. Sci. 10* (Jan. 1975), 110-135.

[2] David Harel. *First-Order Dynamic Logic. Lecture Notes in Computer Science, No. 68,* Springer-Verlag, Berlin, 1979.

[3] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM. 12*, 10 (Oct. 1969), 576-580.

[4] R. M. Keller. Formal Verification of Parallel Programs. *Comm. ACM. 19*, 7 (July 1976), 371-384.

[5] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Comm. ACM. 17*, 8 (Aug. 1974), 453-455.

[6] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. on Soft. Eng. SE-3 2*, 2 (Mar. 1977), 125-143.

[7] L. Lamport. The 'Hoare Logic' of Concurrent Programs. *Acta Informatica 14* (1980), 21-37.

[8] L. Lamport. Specifying Concurrent Program Modules. To appear in *ACM. Trans. Prog. Lang. and Sys. 5*, 1 (Jan. 1983).

[9] L. Lamport and F.B. Schneider. The "Hoare Logic" of CSP, and All That. To appear in *ACM. Trans. Prog. Lang. and Sys.*

[10] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica 6*, 4 (1976), 319-340.

[11] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM. Trans. Prog. Lang. and Sys. 4*, 3 (July 1982), 455-495.

[12] A. Pnueli. The Temporal Logic of Programs. *Proc. of the 18th Symposium on the Foundations of Computer Science*, Nov. 1977, 46-57.