17

win and *sin*: Predicate Transformers for Currency

Leslie Lamport

May 1, 1987, Revised December 8, 1989

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in l984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

win and *sin*: Predicate Transformers for Concurrency

Leslie Lamport

1 May 1987
Revised December 8, 1989

Author's Abstract

The *weakest liberal precondition* and *strongest postcondition* predicate transformers are generalized to the *weakest invariant* and *strongest invariant*. These new predicate transformers are useful for reasoning about concurrent programs containing operations in which the grain of atomicity is unspecified. They can also be used to replace behavioral arguments with more rigorous assertional ones.

Capsule Review

It is widely recognized that reasoning, either formally or informally, about concurrent programs is harder than reasoning about ordinary sequential programs. At any point in the execution of any thread of control it is potentially possible for shared variables to be written by another thread, invalidating conditions that have just been established by the first thread.

The standard approach to verifying concurrent programs is to divide the execution of each thread into a series of atomic actions, and to show that all possible interleavings of the atomic actions of the various threads are guaranteed to produce correct results. This report introduces a new method for verifying concurrent programs without specifying the grain of atomicity of operations. It requires instead only that certain invariants of the operations be known. For example, a statement like $a := b + c$ typically consists of several atomic actions (particularly if a, b, and c are long integers and cannot be read or written atomically by the hardware), but it may be assumed that execution of the entire statement or any part of it leaves invariant the value of any variable d distinct from a, b, and c.

From a theoretical standpoint, the verification method introduced in this report is interesting in that it makes it possible to verify concurrent programs without precisely specifying the decomposition of statements into atomic operations. From a practical standpoint, this means that programs can be analyzed at a coarser grain than that of atomic operations.

The tools developed here are by no means a panacea. Verification of concurrent algorithms is still a tricky business, requiring careful attention to detail, as study of the examples in the text will indicate. However, by allowing the analysis to be done at a coarser grain, these tools can reduce the number of steps (and consequently the temptation to skip some steps) needed for verification, making the process somewhat less arduous (and error-prone) than it has been in the past.

Jim Saxe

Contents

List of Figures

1 Introduction

Partial correctness is a relation between the program states before and after execution of an entire program. For reasoning about concurrent programs, the appropriate generalization of partial correctness is invariance, which is a relation between the program states before and after the execution of each atomic operation of a program. The appropriate generalization of the Hoare triple $\{P\}\, S\, \{Q\}$ is the assertion that S leaves a predicate I invariant [13]. Because the invariant I describes the program state during execution, it must depend upon the control state as well as on the values of ordinary program variables.

The predicate transformers *wlp* (the weakest liberal precondition) and *sp* (the strongest postcondition) for proving partial correctness properties of sequential programs were developed in the early 1970's by de Bakker and others [3, 4] and popularized by Dijkstra [5]. Here, we generalize them to the predicate transformers *win* (the weakest invariant) and *sin* (the strongest invariant) for proving safety properties of concurrent programs. Some of the ideas presented here originally appeared in [12], but with a different notation.

The *wlp* and *sp* operators are useful because they allow one to encode partial correctness information in a predicate. A predicate containing the *wlp* or *sp* operator can be used in a program annotation to prove a partial correctness property. While it is well known that the ability to express such predicates is necessary for a logic of Hoare triples to be complete [1], the practical utility of these predicates in proving partial correctness properties is not widely appreciated.

In an analogous fashion, the predicate transformers *win* and *sin* are useful for proving invariance properties of concurrent programs because predicates they can appear in an invariant. We have discovered two applications of these predicate transformers: reasoning about programs that are not decomposed into their atomic operations, and transforming certain behavioral reasoning into more rigorous assertional reasoning.

We give two examples of reasoning about nonatomic operations. The first shows that, when the atomicity of an operation is obviously irrelevant, we can reason directly about the nonatomic operation instead of pretending that it is atomic. While not having to introduce unnecessary atomicity is aesthetically pleasing, it offers little practical benefit. The second example, a correctness proof of the bakery algorithm [9], is more compelling. The bakery algorithm is a mutual exclusion algorithm that makes no atomicity assumptions about its operations. Our proof reveals that the algorithm has a subtle bug—more precisely, its correctness depends upon unstated assumptions. Correctness proofs of the bakery algorithm have appeared in [9] and [10], and a proof of a variant, requiring the same assumptions, appeared in [11]. The fact that none of these other proofs revealed the hidden assumption indicates the utility of the approach presented here.

Our final example illustrates a different use of the predicate transformers. Assertional reasoning, based upon invariance, has proved to be more reliable than behavioral reasoning, which argues directly about the sequence of operations executed by the program. However, there have been examples in which a purely assertional proof was more complicated than a hybrid proof—one using a behavioral argument to show that the given algorithm is equivalent to a simpler one whose correctness is proved assertionally. It appears that the *win* and *sin* operators can be used in these examples to replace the hybrid proof with a simple, assertional one. This is illustrated by a distributed algorithm abstracted from part of a well-known algorithm for computing a minimum spanning tree [6].

This paper is primarily concerned with applications of *win* and *sin* rather than with their formal properties. The treatment of the formalism is brief, and no attempt is made to develop a complete proof system. We hope to present completeness results in a future paper.

Our approach is semantic rather than syntactic, meaning that we deal not with pieces of program text but with the mathematical objects represented by those pieces of text. For example, we view the expression $x > 0$ as a boolean-valued function on the program state (a function that depends only on the value of the variable x) rather than as a string of characters generated by some grammar. By eschewing syntax, we hope to focus attention on the underlying concepts.

The definitions and properties of the predicate transformers *win* and *sin* are independent of a programming language. They can be applied to concurrent programs written in any imperative language, regardless of whether processes communicate through shared memory, synchronous or asynchronous message passing, or remote procedure call. However, our major examples involve a generalization of the Owicki-Gries method [10, 14], and we describe this method only for programs that can be written in a very simple language.

2 Assertional Reasoning

We begin with a review of the traditional approach to concurrent program verification that will serve to introduce some notation and describe our view of concurrent programs. We take as an example the program of Figure 1. In this program, the body of the outer **cobegin** is executed concurrently as n separate processes, each with a different value substituted for i, and the body of the inner **cobegin** similarly "forks" $n-1$ subprocesses. (Here and throughout this paper, the range of values of the variables i and j is assumed to be the set $\{1, \ldots, n\}$. To avoid having to define the meaning of an empty **cobegin** statement, we assume that $n > 1$ for this program and its variants that appear later.) The **await** operation can be executed only when its condition is true, in which case it is equivalent to a **skip**. Angle brackets enclose atomic operations, and the predicate

var *num*: **array** 1 . . . n **of nonnegative integer**;
cobegin $\square_{i=1\ldots n}$
  loop ncs_i: $\langle$*noncritical section*$\rangle$;
    β_i: $\langle num[i] := 1 + \max\{num[j] : j \neq i\}\rangle$;
    δ_i: **cobegin** $\square_{j \neq i}$
      η_{ij}: $\langle$**await** $i \ll j\rangle$
    coend;
    cs_i: $\langle$*critical section*$\rangle$;
    ρ_i: $\langle num[i] := 0\rangle$
  endloop
coend

Figure 1: A simplified version of the bakery algorithm.

$i \ll j$ is defined to equal

$$(num[j] = 0) \lor (num[i] < num[j]) \lor (num[i] = num[j] \land i < j) \qquad (1)$$

Since we are concerned only with safety properties [10], it does not matter what fairness assumptions are made about when an operation *must* be executed. Thus, the inner **cobegin** could be implemented by a **for** loop, with the subprocesses executed one after the other.

This program is a simplified version of the bakery algorithm—a mutual exclusion algorithm described in [9]. The critical and noncritical sections are represented by atomic operations, which are assumed not to modify the variables $num[i]$, and the original bakery algorithm is trivialized by making the operations β_i and η_{ij} atomic.

2.1 States and Predicates

In our semantic approach, a program consists of a set **S** of states and a set Π of atomic operations.[1] Here, we describe the set of states; atomic operations are defined in Section 2.2.

States

A *state* of a program is a mapping from the set of program variables to some set of values—in other words, a state consists of an assignment of values to the program's variables. In addition to ordinary program variables, we also introduce *control* variables that describe the control state of the program.

For simple **cobegin** programs, such as the simplified bakery algorithm of Figure 1, the control variables consist of variables *at*(ξ), for every atomic operation ξ in Π.

[1] If we were considering liveness properties as well as safety properties, a program would also have to include fairness conditions.

The variable *at(*ξ*)* is a boolean-valued variable whose value is true iff (if and only if) control is at operation ξ. For the simplified bakery algorithm, the variables are $num[i]$, *at(*ncs_i*)*, *at(*β_i*)*, *at(*η_{ij}*)*, *at(*cs_i*)*, and *at(*ρ_i*)*, for all i, $j = 1, \ldots, n$ with $i \neq j$. A state of this program is an assignment of nonnegative integers to the variables $num[i]$ and booleans to the *at* variables.

We restrict the set **S** of states to allow only valid assignments of values to the control variables. For simple **cobegin** programs, we require that the values of the *at* variables do not declare control to be at two places in the same process—except where a nested **cobegin** splits the process into subprocesses. For example, in the simplified bakery algorithm, *at(*η_{ij}*)* and *at(*cs_i*)* are not both assigned the value *true* in any state.

The set **S** of program states may include ones we don't expect to occur during an execution. For example, the simplified bakery algorithm contains states with *at(*η_{ij}*)* true and $num[i] = 0$, even though β_i sets $num[i]$ to a nonzero value. Similarly, there are states in which *at(*cs_1*)* and *at(*cs_2*)* both have the value *true*, even though this is a correct mutual exclusion algorithm, and control will never be simultaneously at the critical sections of two different processes while executing the program.

Definition of $s^{x_1 \cdots x_m}_{v_1 \cdots v_m}$

Let $x_1, \ldots, x_m$ be distinct variables, and let $v_1, \ldots, v_m$ be values. For any state s, we define $s^{x_1 \cdots x_m}_{v_1 \cdots v_m}$ to be the assignment of values to variables that is the same as s except that each x_p is assigned the value v_p. Note that $s^{x_1 \cdots x_m}_{v_1 \cdots v_m}$ need not be a state if one or more of the x_i are control variables.

State Functions and Predicates

A *state function* is a function whose domain is the set of states, and a *predicate* is a boolean-valued state function. If P is a predicate, we write $s \models P$ instead of $P(s)$ and define $\models P$ to equal $\forall s \in S\colon s \models P$. Thus, $\models P$ asserts that P is true for all program states.

A variable is a state function whose value on a state is the value of the variable in that state. In particular, a boolean-valued variable is a predicate.

State Function Not Accessing a Set of Variables

We say that a state function f *does not access* a set $\{x_1, \ldots, x_m\}$ of variables iff $f(s) = f(s^{x_1 \cdots x_m}_{v_1 \cdots v_m})$ for every state s and all values $v_1, \ldots, v_m$ such that $s^{x_1 \cdots x_m}_{v_1 \cdots v_m}$ is a state. Intuitively, f does not access a set of variables iff the value of f can be computed without knowing the values of those variables.[2]

[2]One might expect that a state function does not access a set $\{x_1, \ldots, x_m\}$ of variables iff it does not access each singleton set $\{x_i\}$. However, this is not true. For example, in the simplified bakery algorithm, taking any state and changing the value of either *at(*cs_i*)* or *at(*ρ_i*)* by itself cannot yield a valid control state. Hence, every state function does not access the set $\{$*at(*cs_i*)*$\}$ and does not access the set $\{$*at(*ρ_i*)*$\}$. However, *at(*cs_i*)* is a state function that accesses (does not not access) the set $\{$*at(*cs_i*)*, *at(*ρ_i*)*$\}$. What all this means is

A *control predicate* is a predicate that does not access the set of all variables other than control variables.

2.2 Actions and Atomic Operations

Actions

An *action* is a relation on the set of states—that is, a set of pairs of states. The possible executions of an atomic operation are represented by an action ξ, where $(s, t) \in \xi$ means that executing the atomic operation starting in state s can produce state t.

An action ξ is *deterministic* iff for each state s there is at most one t such that $(s, t) \in \xi$. Any deterministic action can be written in the following form, where the x_p are distinct program variables, b is a predicate, and the e_p are state functions:

$$b \qquad \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} := \begin{pmatrix} e_1 \\ \vdots \\ e_m \end{pmatrix} \tag{2}$$

This describes the set of all pairs $(s, s^{x_1 \; \cdots \; x_m}_{e_1(s) \cdots e_m(s)})$ such that $s \models b$ equals *true*. In other words, it is an action that can be executed only if b is true, and it has the effect of first evaluating the expressions e_p and then setting the x_p, all in one step. Although we do not assume that actions are deterministic, we will not discuss the representation of nondeterministic actions.

For the simplified bakery algorithm of Figure 1, statement β_i describes the action

$$at(\beta_i) \qquad \begin{pmatrix} num[i] \\ at(\beta_i) \\ at(\eta_{ij}) \end{pmatrix} := \begin{pmatrix} 1 + \max\{num[j] : j \neq i\} \\ false \\ true, \;\; for\ all\ j \neq i \end{pmatrix}$$

and statement η_{ij} describes the action

$$at(\eta_{ij}) \wedge i \ll j \qquad \begin{pmatrix} at(\eta_{ij}) \\ at(cs_i) \end{pmatrix} := \begin{pmatrix} false \\ \bigwedge_{k \neq i,j} \neg at(\eta_{ik}) \end{pmatrix}$$

Action Modifying or Not Accessing Variables

We say that an action ξ *modifies* a variable x iff there exists a pair (s, t) in ξ such that x has different values in states s and t. We say that ξ *does not access* the set $\{x_1, \ldots, x_m\}$ of variables iff ξ does not modify any of the x_p and for any $(s, t) \in \xi$ and any values v_1, $\ldots$, v_m, if $s^{x_1 \cdots x_m}_{v_1 \cdots v_m}$ is a state then $(s^{x_1 \cdots x_m}_{v_1 \cdots v_m}, t^{x_1 \cdots x_m}_{v_1 \cdots v_m}) \in \xi$. Intuitively, x does not access a set of variables iff ξ can be executed without reading or writing any of those variables.

that there is no unique definition of the set of variables that *are* accessed by a state function.

The action (2) does not modify any variables other than the x_p; it does not access any set of variables that does not contain the x_p and is not accessed by any of the e_p. In the simplified bakery algorithm, the action described by β_2 modifies only the variables $num[2]$, $at(\beta_2)$, and $at(\eta_{2j})$ for all $j \neq 2$; it does not access the set $\{at(\beta_1), at(\eta_{12})\}$ (as well as many other sets of variables).

Atomic Operations

An atomic operation ξ of a program consists of an action together with control predicates $at(\xi)$ and $after(\xi)$. Intuitively, $at(\xi)$ asserts that control is at a point where ξ can be executed, and $after(\xi)$ asserts that control is at a point that can be reached by executing ξ. In the simplified bakery algorithm,

$$
\begin{aligned}
after(ncs_i) &\equiv at(\beta_i) \\
after(\beta_i) &\equiv \bigwedge_{j \neq i} at(\eta_{ij}) \\
after(\eta_{ij}) &\equiv at(cs_i) \vee \left(\neg at(\eta_{ij}) \wedge \bigvee_{k \neq i,j} at(\eta_{ik}) \right) \\
after(\rho_i) &\equiv at(ncs_i)
\end{aligned}
$$

The *at* predicates are program variables and are not defined in terms of anything else.

We will identify an atomic operation with its action. Thus, if ξ is an atomic operation, $(s, t) \in \xi$ means that the pair of states (s, t) is an element of the action of ξ. Similarly, we say that an atomic operation does not modify a variable iff its action does not modify the variable.

Our informal statement, that $at(\xi)$ holds iff control is at ξ and $after(\xi)$ holds iff control is immediately after ξ, is formalized as the following assumption about atomic operations.

CTL1. For any atomic operation ξ: if $(s, t) \in \xi$ then $s \models at(\xi)$ and $t \models after(\xi)$.

For simple **cobegin** programs like the simplified bakery algorithm, there is a variable $at(\xi)$ for each atomic operation ξ in the set Π of the program's atomic operations. For programs written in a different language, the *at* predicates might be defined in terms of other control variables.

2.3 The Hoare Logic of Actions

Definition of Hoare Triples

Let ξ be an action and let P and Q be predicates. We define the Hoare triple $\{P\}\,\xi\,\{Q\}$ to mean $\forall (s, t) \in \xi : (s \models P) \Rightarrow (t \models Q)$. In other words, $\{P\}\,\xi\,\{Q\}$ asserts that if P is true in state s and executing ξ in state s can yield state t, then Q is true in state t. While this definition is superficially the same as the usual one for ordinary Hoare triples, it is different in two respects: (i) ξ is an action (a set of pairs of states), not a program statement, and (ii) the state includes control variables, not just ordinary program variables.

Proving Hoare Triples

The language-independent rules for reasoning about ordinary Hoare triples [8] apply to our Hoare triples as well. Because our states include control variables, we do not need a separate axiom or proof rule for every language construct. Instead, we can use the simple rule that, if ξ is the action (2), then $\{P\}\,\xi\,\{Q\}$ is equivalent to $\models (P \wedge b) \Rightarrow Q^{x_1\,\cdots\,x_m}_{e_1\,\cdots\,e_m}$, where $Q^{x_1\,\cdots\,x_m}_{e_1\,\cdots\,e_m}$ is the predicate defined by letting $s \models Q^{x_1\,\cdots\,x_m}_{e_1\,\cdots\,e_m}$ equal $s^{x_1\quad\cdots\,x_m}_{e_1(s)\cdots\,e_m(s)} \models Q$, for any state s.[3] This rule follows from the definitions of $\{P\}\,\xi\,\{Q\}$ and of action (2). As an example, the reader can derive $\{(num[i] > 0) \vee \neg at(\eta_{ij})\}\,\eta_{ij}\,\{i \ll j\}$ from this rule and the definition of $i \ll j$.

Action Leaving a Predicate Invariant or Unchanged

We say that a predicate P is an *invariant* of an action ξ, or that ξ *leaves* P *invariant*, iff $\{P\}\,\xi\,\{P\}$ holds. In other words, P is an invariant of ξ iff any execution of ξ from a state in which P is true yields a state in which P is true.

We say that ξ *leaves* P *unchanged* iff it leaves both P and $\neg P$ invariant, which is true iff $(s \models P) \equiv (t \models P)$ for all $(s, t) \in \xi$.

Properties of Invariance

We now list some simple properties that are useful for reasoning about invariance, where ξ is an arbitrary atomic operation and P and the P_h are predicates.

AC1. If P does not access the set of variables modified by ξ, then ξ leaves P unchanged.

AC2. If ξ leaves each P_h invariant, then it leaves $\bigwedge_h P_h$ and $\bigvee_h P_h$ invariant.

AC3. If $\models P \Rightarrow \neg at(\xi)$ then ξ leaves P invariant.

AC4. ξ leaves P invariant iff it leaves $(at(\xi) \vee after(\xi)) \wedge P$ invariant.

Properties AC1 and AC2 follow from the definitions of what it means for an action to leave a predicate invariant or unchanged. Properties AC3 and AC4 follow from the definition of invariance and assumption CTL1.

Remember that an atomic operation ξ consists of an action together with the control predicates $at(\xi)$ and $after(\xi)$. Properties of atomic operations that do not mention control predicates, such as properties AC1 and AC2, hold for any action.

[3] In a syntactic approach, one would define $Q^{x_1\,\cdots\,x_m}_{e_1\,\cdots\,e_m}$ when Q and the e_p are formulas rather than state functions. Given formulas for Q and the e_p, the formula for $Q^{x_1\,\cdots\,x_m}_{e_1\,\cdots\,e_m}$ is obtained by simultaneously substituting e_p for x_p, for $p = 1, \ldots, m$.

2.4 Properties of a Program

Executions

An *execution* of the program consists of a finite or infinite sequence $s_0, s_1, \ldots$ of states such that each pair (s_m, s_{m+1}) is in some action of Π.[4] In other words, an execution is any sequence of states obtained by starting in an arbitrary state and executing program actions. Properties of the program are expressed as assertions about the set of executions.

We do not assume any particular starting state for the execution, so the simplified bakery algorithm has executions beginning in a state with all processes at their critical sections. In our formalism, the usual assumption that the program starts in a proper initial state appears as a hypothesis in the property to be proved.

We can consider two programs to be equivalent if they have the same set of executions. A pair of states is in an action of Π iff it is in the union of all the actions of Π. (Since actions are sets of pairs, the union of actions is just ordinary set union.) The set of executions of a program depends only on the set **S** of states and the union of the actions in Π. Thus, two programs may be considered equivalent if they have the same set of states and the unions of their atomic operations are the same.

There can be many different sets Π that have the same union and thus define equivalent programs. For example, suppose a program has an atomic operation ξ that sends a message to some process p and an atomic operation μ that sends a message to some other process q. Replacing these two atomic operations by the single atomic operation $\xi \cup \mu$ that sends a message to either p or q results in a new set Π that defines an equivalent program. (We define $at(\xi \cup \mu)$ to be $at(\xi) \vee at(\mu)$ and $after(\xi \cup \mu)$ to be $after(\xi) \vee after(\mu)$.) The action $\xi \cup \mu$ will be nondeterministic if there exists a state in which the program can send a message to either p or q.

Properties

A *property* is a boolean-valued function on the set of sequences of states. The program is said to *satisfy* a property $\mathcal{P}$, written $\models \mathcal{P}$, iff $\mathcal{P}$ is true for every program execution.

If P and Q are predicates, we define $P \Rightarrow \Box Q$ to be the property that is true of a sequence $s_0, s_1, \ldots$ iff $\neg(s_0 \models P) \vee (\forall m\colon s_m \models Q)$. Thus, $\models P \Rightarrow \Box Q$ asserts that Q is true for every state of every program execution that starts in a state with P true.

We consider only properties of the form $P \Rightarrow \Box Q$. Partial correctness is expressed in this form by letting P be the initial condition and Q the predicate asserting that the termination condition (which is a control predicate) implies that the answer is correct. The mutual exclusion property of the simplified bakery algorithm is expressed as $P \Rightarrow \Box Q$ where P is $\bigwedge_i at(ncs_i)$ and Q is $\bigwedge_{i \neq j} \neg(at(cs_i) \wedge at(cs_j))$.

[4] Since we are concerned only with safety properties, we need not disallow finite sequences that end in nonhalting states.

Program Invariants

A predicate is said to be a *program invariant* iff it is an invariant of every action of Π, or, equivalently, iff it is an invariant of the union of all actions of Π. A predicate I is a program invariant iff $\models I \Rightarrow \Box I$. It is clear that $\models P \Rightarrow I$, $\models I \Rightarrow \Box I$, and $\models I \Rightarrow Q$ together imply $\models P \Rightarrow \Box Q$. Hence, to prove $\models P \Rightarrow \Box Q$, it suffices to find a program invariant I such that $\models P \Rightarrow I$ and $\models I \Rightarrow Q$. This reduces the proof of a safety property, which is an assertion about executions, to reasoning about predicates and individual actions.

2.5 Simple cobegin Programs

We will describe the Owicki-Gries method only for programs that can be written in a simple language of nested **cobegin**s. We now describe these programs and make some definitions that pertain only to them and not to arbitrary programs.

The Programs and Their Control Predicates

A simple **cobegin** program is one that can be written in a language consisting of elementary statements (such as assignment and **await** statements), concatenation (";"), nonterminating **loop—endloop** statements, and **cobegin—coend** statements. We require that any "**loop**" keywords must precede every ";". Each elementary statement is enclosed in angle brackets, indicating that it represents an atomic operation.

The control variables of a simple **cobegin** program consist of the variables *at*(ξ) for all its atomic operations ξ. The *after* predicates can be defined in terms of the *at* variables by a simple recursion on the program structure; we will not bother giving the general definition.

Atomic Operations Belonging to Different Processes

We say that two atomic operations *belong to different processes* iff they occur in different clauses of the same **cobegin** statement. For example, in the simplified bakery algorithm of Figure 1, η_{ij} and η_{ik} belong to different processes if $j \neq k$, while β_i and η_{ij} do not belong to different processes. The Owicki-Gries method is based upon the following property of simple **cobegin** programs.

CTL2. If atomic operations ξ and μ in Π belong to different processes, then ξ leaves *at*(μ) and *after*(μ) unchanged.

Predecessors

We say that an atomic operation μ is a *predecessor* of an atomic operation ξ iff control can reach ξ by executing μ. In the simplified bakery algorithm, β_i is the only predecessor of each η_{ij}, and each η_{ij} is the only predecessor of cs_i. Our restriction

var *num*: **array** 1. . . n **of nonnegative integer**;
cobegin $\Box_{i=1\ldots n}$
  loop ncs_i: $\langle$*noncritical section*$\rangle$;
    β_i: $\langle num[i] := 1 + \max\{num[j] : j \neq i\}\rangle$ $\{num[i] > 0\}$;
    δ_i: **cobegin** $\Box_{j \neq i}$
 $\{num[i] > 0\}$ η_{ij}: $\langle$**await** $i \ll j\rangle$ $\{(num[i] > 0) \wedge (i \ll j)\}$
      coend;
   $\{(num[i] > 0) \wedge \bigwedge_{j \neq i}(i \ll j)\}$
    cs_i: $\langle$*critical section*$\rangle$;
    ρ_i: $\langle num[i] := 0\rangle$
  endloop
coend

Figure 2: An annotation of the simplified bakery algorithm.

that a "**loop**" cannot follow a ";" implies that an atomic operation has more than one predecessor only if it immediately follows a "**coend**". If the body of a **loop** statement consists of a single atomic operation ξ, then ξ is its own predecessor.

2.6 The Owicki-Gries Method

Decomposing the Invariant

One can prove directly that a predicate I is a program invariant by proving $\{I\}\,\xi\,\{I\}$ for every atomic operation ξ, as proposed by Ashcroft [2]. However, in the Owicki-Gries method [10, 14], the proof is decomposed into smaller steps by writing I as a conjunction of simpler predicates. For our **cobegin** programs, I is written in the form

$$\bigwedge_{\xi \in \Pi} (at(\xi) \Rightarrow I_\xi) \wedge (after(\xi) \Rightarrow I'_\xi) \tag{3}$$

for predicates I_ξ and I'_ξ. Intuitively, I is the predicate asserting that, for every atomic operation ξ, if control is at ξ then I_ξ is true, and if control is immediately after ξ then I'_ξ is true. We represent I as a *program annotation*, where $\{I_\xi\}$ is written immediately before and $\{I'_\xi\}$ immediately after ξ, omitting predicates that are identically *true*. We say that the *annotation is invariant* iff the predicate I represented by the annotation is a program invariant.

Figure 2 shows such an annotation for the simplified bakery algorithm. For the predicate I defined by this annotation, it is easy to see that $\models \bigwedge_i at(ncs_i) \Rightarrow I$, and some predicate calculus reasoning shows that the definition of $i \ll j$ implies $\models I \Rightarrow \bigwedge_{i \neq j} \neg(at(cs_i) \wedge at(cs_j))$. Hence, to prove the mutual exclusion property for this algorithm, we need prove only the invariance of I.

The Owicki-Gries Conditions

One proves the invariance of an annotation by proving the following two *Owicki-Gries conditions*.

> *Sequential Correctness*:
>
> (a) For every action $\xi \in \Pi$: $\{I_\xi\}\,\xi\,\{I'_\xi\}$.
>
> (b) For every action $\xi \in \Pi$: if $\mu_1, \ldots, \mu_m$ are the predecessors of ξ, then $\models (at(\xi) \wedge \bigwedge_p I'_{\mu_p}) \Rightarrow I_\xi$.
>
> *Interference Freedom*: For every pair of distinct atomic operations ξ, μ in Π that belong to different processes: $\{I_\mu \wedge I_\xi\}\,\mu\,\{I_\xi\}$ and $\{I_\mu \wedge I'_\xi\}\,\mu\,\{I'_\xi\}$.

The proof that these conditions imply the invariance of (3) uses properties CTL1, CTL2, and AC2, the definition of a Hoare triple, and properties of the control structure of simple **cobegin** programs.

We urge the reader who is not familiar with the Owicki-Gries method to use it to prove the invariance of the annotation of Figure 2.

3 The Weakest and Strongest Invariants

3.1 More About Actions

The Composition of Actions

Let $\xi\mu$ denote the composition of the actions ξ and μ, which is defined to be the action $\{(s,u) : \exists t : ((s,t) \in \xi) \wedge ((t,u) \in \mu)\}$. Thus, $\xi\mu$ is executed by first executing ξ then executing μ, all as a single action. The composition of two actions in Π, the set of atomic operations of the program, is usually not an element of Π.

The composition $\xi_1 \cdots \xi_m$ of any finite, nonempty sequence of actions is defined in the obvious way, and the composition of the null sequence of actions is defined to be the identity action $\{(s,s) : s \in S\}$. Thus, any element in Π^*, the set of finite sequences of atomic operations in Π, is defined to be an action.

Commutativity of Actions

We say that ξ *right commutes* with μ (or that μ *left commutes* with ξ) iff $\xi\mu \subseteq \mu\xi$. Hence, ξ right commutes with μ iff $(s,t) \in \xi$ and $(t,u) \in \mu$ imply that there exists a state t' with $(s,t') \in \mu$ and $(t',u) \in \xi$. Intuitively, ξ right commutes with μ iff any state reachable from state s by first executing ξ and then executing μ is also reachable from s by first executing μ then executing ξ.

Two actions are said to *commute* iff each of them right commutes with the other—in other words, iff executing them in either order has the same effect. A semaphore action $P(s)$ right commutes with a semaphore action $V(s)$ in a different process, but these two actions do not commute.

The following property is a consequence of the definitions of commutativity and of what it means for an action not to access a set of variables.

AC5. Two actions commute if each of them does not access the set of variables modified by the other.

3.2 The Weakest Liberal Precondition

For any action ξ and predicate Q, we define the predicate $wlp(\xi, Q)$ by letting $s \models wlp(\xi, Q)$ equal $\forall t \in S: ((s, t) \in \xi) \Rightarrow (t \models Q)$. The operator *wlp* is the *weakest liberal precondition* operator [5]. The predicate $wlp(\xi, Q)$ is the weakest one satisfying $\{wlp(\xi, Q)\}\, \xi\, \{Q\}$. Thus, $\{P\}\, \xi\, \{Q\}$ is equivalent to $\models P \Rightarrow wlp(\xi, Q)$, so ξ leaves I invariant iff $\models I \Rightarrow wlp(\xi, I)$. If ξ is the action defined by (2), then $wlp(\xi, Q) \equiv Q^{x_1 \cdots x_m}_{e_1 \cdots e_m} \vee \neg b$.

Our definition of $wlp(\xi, Q)$ differs from the usual definition in that (i) ξ is an action rather than a program statement, and (ii) our predicates may be functions of control variables, rather than just of ordinary variables. For example, CTL1 and the definition of *wlp* imply $\models (\neg at(\xi)) \Rightarrow wlp(\xi, Q)$ for any atomic operation ξ and predicate Q. This result has no counterpart for the usual definition of *wlp*.

We will use the following properties of *wlp*, where P, Q, and the Q_h are any predicates, and ξ and μ are any actions.

WLP0. $\models wlp(\xi\mu, Q) \equiv wlp(\xi, wlp(\mu, Q))$

WLP1. $\models \bigwedge_h wlp(\xi, Q_h) \equiv wlp(\xi, \bigwedge_h Q_h)$

WLP2. If $\models P \Rightarrow Q$ then $\models wlp(\xi, P) \Rightarrow wlp(\xi, Q)$.

WLP3. If ξ leaves I invariant and ξ right commutes with μ, then ξ leaves $wlp(\mu, I)$ invariant.

WLP4. If ξ leaves P unchanged, then $\models wlp(\xi, P \vee Q) \equiv P \vee wlp(\xi, Q)$.

WLP5. If a set of variables is not accessed by ξ and not accessed by Q, then it is not accessed by $wlp(\xi, Q)$.

Properties WLP0–WLP2 follow easily from the definition of *wlp* and are well known. Note that in WLP1, h can range over an infinite set of indices. Property WLP3 follows from WLP0 and the easily derived property that $\alpha \subseteq \beta$ implies $\models wlp(\beta, Q) \Rightarrow wlp(\alpha, Q)$. Property WLP4 can be derived from WLP1 and WLP2, although it is easier to prove it directly from the definition of *wlp*. Property WLP5 follows from the definitions of *wlp* and of what it means for a predicate or an action not to access a set of variables.

3.3 The Strongest Postcondition

The *strongest postcondition* operator, *sp*, is defined by letting $t \models sp(\xi, P)$ equal $\exists s \in S: ((s, t) \in \xi) \wedge (s \models P)$. It follows from this definition that $\{P\}\,\xi\,\{Q\}$ is equivalent to $\models sp(\xi, P) \Rightarrow Q$.

As observed by de Bakker and Meertens [4], the operator *sp* is a dual of *wlp*; for every property of *wlp* there is a corresponding dual property of *sp*. For example, the following are the duals of WLP2 and WLP3.

SP2. If $\models P \Rightarrow Q$ then $\models sp(\xi, P) \Rightarrow sp(\xi, Q)$.

SP3. If ξ leaves I invariant and ξ left commutes with μ, then ξ leaves $sp(\mu, I)$ invariant.

The interested reader can derive these and the duals of the other properties of *wlp*.

3.4 Nonatomic Operations

Operations and Their Control Predicates

An *operation* σ consists of a set of atomic operations and two control predicates, $at(\sigma)$ and $after(\sigma)$. The set of operations of σ contains all the atomic operations that constitute σ, and the predicates $at(\sigma)$ and $after(\sigma)$ assert that control is at the entry and exit point of σ, respectively. For example, in the simplified bakery algorithm, the operation δ_i has $\{\eta_{ij} : j \neq i\}$ as its set of operations, $at(\delta_i) \equiv \bigwedge_j at(\eta_{ij})$, and $after(\delta_i) \equiv at(cs_i)$.

We identify an operation σ with its set of atomic operations, writing $\xi \in \sigma$ to denote that ξ is an element of σ's set of atomic operations. We can view an operation as a set of actions plus certain control information, so any concept defined for sets of actions is also defined for operations. Any property of operations that does not mention control predicates holds for an arbitrary set of actions.

If σ is an operation, we define the control predicate $in(\sigma)$ to equal $\bigvee_{\xi\in\sigma} at(\xi)$, so $in(\sigma)$ asserts that control is inside σ or at its entry point. We make the following assumption about the relation between $in(\sigma)$, $after(\sigma)$, and the control predicates for the atomic operations in σ.

CTL3. $\models (in(\sigma) \vee after(\sigma)) \equiv \bigvee_{\xi\in\sigma}(at(\xi) \vee after(\xi))$

We identify an atomic operation ξ with the singleton set $\{\xi\}$, so an atomic operation is an operation consisting of a single action. If ξ is an atomic operation, then $in(\xi)$

is equivalent to *at(ξ)*. Therefore, any rules for reasoning about nonatomic operations should reduce to rules for atomic operations when *in* is replaced by *at*.

The Action $\langle\sigma\rangle$

For an operation σ, we let $\langle\sigma\rangle$ denote the action consisting of all pairs (s, t) such that an execution of σ starting from state s can terminate in state t. In other words, $\langle\sigma\rangle$ is the action obtained by considering σ to be an atomic operation, where nonterminating executions are disallowed. If $\models$ *after(σ)* $\Rightarrow \neg$*in(σ)* holds, so σ is not a "self-looping" operation, then we can define the action $\langle\sigma\rangle$ in terms of σ, *at(σ)*, and *after(σ)* by

$$\langle\sigma\rangle = \bigcup_{\lambda\in\sigma^*} \{(s, t) \in \lambda : (s \models \mathit{at}(\sigma)) \wedge (t \models \mathit{after}(\sigma))\} \tag{4}$$

When self-looping operations are allowed, the definition of $\langle\sigma\rangle$ is more complicated and is omitted.

Hoare Triples, *wlp*, and *sp* for Operations

We have defined Hoare triples, *wlp*, and *sp* for actions. We extend these definitions to operations by defining $\{P\}\,\sigma\,\{Q\}$ to equal $\{P\}\,\langle\sigma\rangle\,\{Q\}$, defining $wlp(\sigma, Q)$ to equal $wlp(\langle\sigma\rangle, Q)$, etc.

These concepts are traditionally defined for program statements. If we view a program statement as an operation, then our definitions are essentially the same as the conventional ones—except that our program state includes control information. More precisely, if operation σ represents a program statement S, and the predicate Q does not access the set of control variables, then $wlp(\sigma, Q)$ equals $wlp(S, Q) \vee \neg$*at(σ)*, where $wlp(S, -)$ denotes the traditional weakest liberal precondition operator for statement S.

Some Definitions for Sets of Actions

We now extend the definitions of some properties of individual actions to properties of sets of actions (and hence of operations) by defining them to hold for a set of actions iff they hold for each action in the set. A set σ of actions is said to leave a predicate P *invariant* iff each action in σ leaves P invariant, and to leave P *unchanged* iff each action in σ leaves P unchanged. We say that σ *modifies* a variable iff some action in σ modifies the variable, and that it *does not access* a set of variables iff each of its actions does not access the set of variables. We say that σ *right commutes* with a set of actions τ iff every action of σ right commutes with every action of τ; the definitions of *left commutes* and *commutes* are analogous.

Properties of Operations

We will use the following general properties of operations, where σ and τ are any operations and P, Q, and the P_h are any predicates. Note that OP1, OP2, and OP5 hold for arbitrary sets of actions, not just for operations.

OP1. If P does not access the set of variables modified by σ, then σ leaves P unchanged.

OP2. If σ leaves each P_h invariant, then it leaves $\bigwedge_h P_h$ and $\bigvee_h P_h$ invariant.

OP3. σ leaves $P \land \neg$*in(*σ*)* invariant.

OP4. σ leaves P invariant iff it leaves (*in(*σ*)* $\lor$ *after(*σ*)*) $\land P$ invariant.

OP5. Operations σ and τ commute if each of them does not access the set of variables modified by the other.

Properties OP1, OP2, and OP5 are immediate consequences of the correspondingly-numbered properties of actions. Property OP3 follows from AC3 and the definition of *in(*σ*)*. Property OP4 follows from AC3, AC4, the definition of *in(*σ*)*, and assumption CTL3.

3.5 The Weakest Invariant

Definition of *win*

Let σ be a set of actions and let Q be a predicate. The predicate $win(\sigma, Q)$ is defined to equal the disjunction of all predicates I such that $\models I \Rightarrow Q$ and σ leaves I invariant. The operator *win* is called the *weakest invariant* operator. By OP2, $win(\sigma, Q)$ is an invariant of σ; it is the weakest invariant of σ that implies Q. The set of actions σ leaves Q invariant iff $\models Q \equiv win(\sigma, Q)$. (Since $\models win(\sigma, Q) \Rightarrow Q$ always holds, σ leaves Q invariant iff $\models Q \Rightarrow win(\sigma, Q)$.)

Expressing *win* in Terms of *wlp*

The *win* operator can be expressed in terms of *wlp* as follows.

$$win(\sigma, Q) \equiv \bigwedge_{\lambda \in \sigma^*} wlp(\lambda, Q) \tag{5}$$

Let R denote the right-hand side of (5). To verify (5), we must prove that (i) $\models R \Rightarrow Q$, (ii) R is an invariant of σ, and (iii) R is implied by every invariant of σ. Property (i) holds because the empty sequence, which is in σ^*, is the identity action ι, and $wlp(\iota, Q) = Q$. To prove (ii), observe that for any action ξ of σ, WLP0 and WLP1 imply $\models wlp(\xi, R) \equiv \bigwedge_\lambda wlp(\xi\lambda, Q)$. Hence $\models R \Rightarrow wlp(\xi, R)$, so ξ leaves R invariant. Finally, it follows from WLP0 and WLP2 that $\models I \Rightarrow wlp(\xi, I)$ and $\models I \Rightarrow wlp(\lambda, Q)$ imply $\models I \Rightarrow wlp(\xi\lambda, Q)$. A simple induction argument then shows that if σ leaves I invariant and $\models I \Rightarrow Q$, then $\models I \Rightarrow wlp(\lambda, Q)$ for all $\lambda \in \sigma^*$, which proves (iii).

Let $\langle\sigma^*\rangle$ be $\bigcup_{\lambda\in\sigma^*}\lambda$, the action consisting of all (s, t) such that executing some finite number of actions of σ starting in s yields t. It is easy to show that $\models win(\sigma, Q) \equiv wlp(\langle\sigma^*\rangle, Q)$. If σ is an operation, so $\langle\sigma\rangle$ is defined, then $\langle\sigma^*\rangle$ is a superset of $\langle\sigma\rangle$. While $\langle\sigma\rangle$ contains pairs of states obtained only from complete executions of σ, the action $\langle\sigma^*\rangle$ includes pairs obtained from incomplete executions as well.

Properties of *win*

We will use the following properties of the *win* operator, where P, Q, and the Q_h are any predicates and σ and τ are any sets of actions. They follow easily from equation (5) and the corresponding properties of *wlp*.

WIN1. $\models \bigwedge_h win(\sigma, Q_h) \equiv win(\sigma, \bigwedge_h Q_h)$

WIN2. If $\models P \Rightarrow Q$ then $\models win(\sigma, P) \Rightarrow win(\sigma, Q)$.

WIN3. If σ leaves I invariant and σ right commutes with τ, then σ leaves $win(\tau, I)$ invariant.

WIN4. If σ leaves P unchanged, then $\models win(\sigma, P \vee Q) \equiv P \vee win(\sigma, Q)$.

WIN5. If a set of variables is not accessed by σ and not accessed by Q, then it is not accessed by $win(\sigma, Q)$.

The Predicate Transformer *winp*

Of particular importance in verifying programs are formulas of the form $win(\sigma, \mathit{after}(\sigma) \Rightarrow Q)$, where σ is an operation. We denote this formula by $winp(\sigma, Q)$, where *winp* stands for *weakest invariant of a postcondition*. The predicate $winp(\sigma, Q)$ asserts of a state s that if control is anywhere in σ, then any terminating execution of σ starting in state s terminates with Q true. Contrast $winp(\sigma, Q)$ with $wlp(\sigma, Q)$, which makes this assertion only for a state s with control at the beginning of σ. We will use the following properties of *winp*.

WINP1. $\models \mathit{at}(\sigma) \wedge winp(\sigma, Q) \equiv \mathit{at}(\sigma) \wedge wlp(\sigma, Q)$

WINP2. If $\models \mathit{after}(\sigma) \Rightarrow \neg \mathit{in}(\sigma)$ then $\models \mathit{after}(\sigma) \wedge winp(\sigma, Q) \equiv \mathit{after}(\sigma) \wedge Q$.

WINP3. If σ leaves P invariant, then $\models P \wedge winp(\sigma, Q) \equiv P \wedge winp(\sigma, P \wedge Q)$.

WINP4. If $\models \mathit{after}(\sigma) \Rightarrow \neg \mathit{in}(\sigma)$ and σ leaves P invariant, then σ leaves $(\mathit{in}(\sigma) \wedge P \wedge winp(\sigma, Q)) \vee (\mathit{after}(\sigma) \wedge P \wedge Q)$ invariant.

The validity of WINP1 should be obvious from our discussion of the relation between *winp* and *wlp*. It can be derived from (4), (5), and the observation that $\models wlp(\bigcup_h \xi_h, Q) \equiv \bigwedge_h wlp(\xi_h, Q)$. Property WINP2 is proved as follows.[5]

[5]Complicated proofs are broken down into numbered steps. Boxed numbers indicate the statement or statements that immediately imply the desired conclusion.

1. $\models$ *after(σ)* $\equiv win(\sigma,$ *after(σ)*$)$

   PROOF: OP3 and the hypothesis imply that σ leaves *after(σ)* invariant.

2. $\models$ *after(σ)* $\wedge\ winp(\sigma, Q) \equiv win(\sigma,$ *after(σ)* $\wedge\ Q)$

   PROOF: By 1, WIN1, and the definition of *winp*, since $\models$ (*after(σ)* $\wedge$ (*after(σ)* $\Rightarrow$ Q)) $\equiv$ *after(σ)* $\wedge\ Q$.

$\boxed{3}$ $\models$ *after(σ)* $\wedge\ winp(\sigma, Q) \equiv$ *after(σ)* $\wedge\ Q$.

   PROOF: By 2 and the definition of *win*, since OP3 and the hypothesis imply that σ leaves *after(σ)* $\wedge\ Q$ invariant, so $win(\sigma,$ *after(σ)* $\wedge\ Q)$ equals *after(σ)* $\wedge\ Q$.

Property WINP3 is proved as follows.

1. $\models winp(\sigma, P \wedge Q) \equiv winp(\sigma, P) \wedge winp(\sigma, Q)$

   PROOF: By the definition of *winp* and WIN1, since (*after(σ)* $\Rightarrow P) \wedge$ (*after(σ)* $\Rightarrow Q$) equals *after(σ)* $\Rightarrow (P \wedge Q)$.

2. $\models P \Rightarrow winp(\sigma, P)$

   PROOF: Since $\models P \Rightarrow$ (*after(σ)* $\Rightarrow P$), WIN2 implies $\models win(\sigma, P) \Rightarrow winp(\sigma, P)$. But σ leaves P invariant, so $\models win(\sigma, P) \equiv P$.

$\boxed{3}$ $\models (P \wedge winp(\sigma, Q)) \Rightarrow (P \wedge winp(\sigma, P \wedge Q))$

   PROOF: By 1 and 2.

$\boxed{4}$ $\models (P \wedge winp(\sigma, P \wedge Q)) \Rightarrow (P \wedge winp(\sigma, Q))$

   PROOF: By WIN2, $\models winp(\sigma, P \wedge Q) \Rightarrow winp(\sigma, Q)$.

To prove WINP4, we apply WINP2 to rewrite (*in(σ)* $\wedge\ P \wedge winp(\sigma, Q)) \vee$ (*after(σ)* $\wedge$ $P \wedge Q$) as (*in(σ)* $\vee$ *after(σ)*) $\wedge\ P \wedge winp(\sigma, Q)$ and then apply OP2.

3.6 The Strongest Invariant

Just as *sp* is the dual of *wlp*, we can define an operator *sin*, the *strongest invariant*, that is dual to *win*. For any set of actions σ and predicate P, $sin(\sigma, P)$ is defined to be the conjunction of all invariants I of σ that are implied by P. Corresponding to (5), we have

$$sin(\sigma, P) \equiv \bigwedge_{\lambda \in \sigma^*} sp(\lambda, P) \tag{6}$$

The dual of *winp* is $sinp(\sigma, P)$, defined to be $sin(\sigma,$ *at(σ)* $\wedge\ P)$, where σ is an operation. We will use the following properties, dual to WIN2 and WIN3, which can be derived from (6), SP2, and SP3.

SIN2. If $\models P \Rightarrow Q$ then $\models sin(\sigma, P) \Rightarrow sin(\sigma, Q)$.

```
array num[1...n] of nonnegative integer
array c[1...n]     of boolean
cobegin □_{i=1...n}
  loop ncs_i: noncritical section;
        α_i: c[i] := true;
        β_i: num[i] := 1 + max{num[j] : j ≠ i};
        γ_i: c[i] := false;
        δ_i: cobegin □_{j≠i}
               ε_ij: await ¬c[j];
               η_ij: await i ≪ j
             coend;
       cs_i: critical section;
        ρ_i: num[i] := 0
  endloop
coend
```

Figure 3: The bakery algorithm.

SIN3. If σ leaves I invariant and σ left commutes with τ, then σ leaves $sin(\tau, I)$ invariant.

3.7 Simple cobegin Programs with Unspecified Atomicity

The Programs and Their Control Predicates

We now consider simple **cobegin** programs containing elementary statements that are not atomic operations. These are programs that can be written in the same simple language considered above, except without the requirement that every elementary statement be enclosed in angle brackets. An example of such a program is the bakery algorithm, given in Figure 3. This is essentially the same as the original version in [9], though with different notation. It is an extreme example because *no* atomic operations are specified.

Figure 3 says nothing about the grain of atomicity of the program's operations. Statement β_i could be executed by reading each $num[j]$ one bit at a time, and writing $num[i]$ one bit at a time. The individual bits could even be read and written several times. Thus, Figure 3 does not describe a single program; it is a specification of a class of programs that are valid implementations of the bakery algorithm. Proving a property of the bakery algorithm means proving that property for any valid implementation.

In addition to the ordinary variables $num[i]$ and $c[i]$, an implementation of the bakery algorithm will contain *hidden variables*—variables not explicitly mentioned in Figure 3. For example, hidden variables are needed to hold the values of intermediate computations when executing β_i. In the bakery algorithm, the control variables are hidden variables. We can't write an explicit expression for the predicate $in(\beta_i)$ in terms of variables $at(\xi)$ for atomic operations ξ because Figure 3 does not specify what those atomic operations are. Such an expression can be written only for a particular implementation, in which the atomic operations are given.

We let Ω denote the set of operations that correspond to the elementary statements and tests of the program. For the bakery algorithm, $\Omega = \{ncs_i, \alpha_i, \beta_i, \gamma_i, \epsilon_{ij}, \eta_{ij}, cs_i, \rho_i : i \neq j\}$. The set Ω is a partition of the set Π of atomic operations, since each atomic operation of the program belongs to exactly one operation in Ω. Of course, the actual atomic operations that constitute an element of Ω depend upon the implementation.

We can deduce certain relations between the *at* and *after* predicates from the program control structure. For example, in the bakery algorithm, we have $\models at(cs_i) \equiv \bigwedge_{j \neq i} after(\eta_{ij})$ and $\models (after(\eta_{ij}) \wedge in(cs_i)) \Rightarrow at(cs_i)$. We will assume these obvious relations without giving a formal method for deriving them.

Operations Belonging to Different Processes

The definition of what it means for two arbitrary operations to belong to different processes is the same as the definition for atomic operations—namely, that σ and τ belong to different processes iff they occur in different clauses of the same **cobegin** statement. We make the following assumption, which is the generalization of CTL2 to arbitrary operations.

CTL4. If operations σ and τ in Ω belong to different processes, then τ leaves $at(\sigma)$, $in(\sigma)$, and $after(\sigma)$ unchanged.

Predecessors

The definition of one operation being a predecessor of another is essentially the same as the definition for atomic operations—namely, an operation ρ in Ω is a predecessor of an operation σ in Ω iff control can reach σ by completing the execution of ρ. In the bakery algorithm, ρ_i is the only predecessor of ncs_i, and each η_{ij} is a predecessor of cs_i.

The Semantics of Nonatomic Operations

To reason formally about programs with nonatomic operations, we must make some assumptions about those operations. Our first assumption is that, in the absence of concurrent execution of other operations, a nonatomic operation has the expected meaning. For example, executing a nonatomic assignment $x := 2 * y$ when y equals 1 sets x to 2. Formally, this means that we assume the validity of ordinary rules for manipulating *wlp* formulas involving nonatomic operations. Thus, if σ is a nonatomic assignment $x := 2 * y$, then $wlp(\sigma, x = 2)$ equals $(y = 1) \vee \neg at(\sigma)$.

What it means to execute a nonatomic operation in the presence of concurrent activity is a subtle issue. Consider again a nonatomic assignment $x := 2 * y$. If x is not concurrently modified by another operation, must execution of this assignment set x to an even value? One can argue that the answer is "yes", since regardless of what value is obtained when reading y, multiplying it by 2 yields an even number. On the other hand, one can argue that the answer is "no", since $\langle x := y \rangle; \langle x := x + y \rangle$ is a valid implementation of $x := 2 * y$ whose execution could set x to an odd value—for example, if another process increments y by 1 in the middle of the execution.

Deciding what the semantics of $x := 2 * y$ should be is a problem in language design—a topic we wish to avoid. Instead, we just assume that this operation does not modify or access any variables we don't expect it to. We can make the obvious assumption that x is the only nonhidden variable modified by this operation, and the operation does not access any set of nonhidden variables that does not contain x or y. However, we also need some assumption about the hidden variables that the operation may modify or access.

Intuitively, we assume that each process has its own local variables that are not accessed or modified by any other process. More precisely, we assume that, for each operation σ in Ω, there is a set of variables that are *local* to σ. If σ and τ are operations in different processes, we assume that they have disjoint sets of local variables. We then assume the following rules for reasoning about nonatomic assignment and **await** statements.

Assignment Rule A nonatomic operation $x := exp(y_1, \ldots, y_m)$ modifies only x and variables local to the operation. The operation does not access any set of variables that contains neither x, nor any y_p, nor any variable local to the operation.

Await Rule A nonatomic operation **await** $exp(y_1, \ldots, y_m)$ modifies only variables local to it. The operation does not access any set of variables that contains neither any y_p nor any variable local to the operation.

3.8 The Owicki-Gries Method with Unspecified Atomicity

Decomposing the Invariant

We now extend the Owicki-Gries method to permit reasoning about simple **cobegin** programs like the bakery algorithm with nonatomic elementary statements. A safety property is still proved by finding the appropriate invariant I, where I is written as an annotation. However, the annotation now denotes the predicate

$$\bigwedge_{\sigma \in \Omega} (\mathit{in}(\sigma) \Rightarrow I_\sigma) \wedge (\mathit{after}(\sigma) \Rightarrow I'_\sigma) \tag{7}$$

Intuitively, this predicate asserts that, for each operation σ, if control is in σ then I_σ is true, and if control is immediately after σ then I'_σ is true. Since *in*(σ) is equivalent to *at*(σ) if σ is an atomic operation, (7) is the same as (3) if every operation σ is atomic.

The Owicki-Gries Conditions

To prove the invariance of an annotation, one proves the following *nonatomic Owicki-Gries conditions*, where J_σ is defined to be $(\mathit{in}(\sigma) \wedge I_\sigma) \vee (\mathit{after}(\sigma) \wedge I'_\sigma)$.

> *Sequential Correctness*:
>
> (a) Every operation $\sigma \in \Omega$ leaves J_σ invariant.
>
> (b) For every operation $\sigma \in \Omega$ and every predecessor set $\rho_1, \ldots, \rho_m$ of σ: $\models (\mathit{at}(\sigma) \wedge \bigwedge_p I'_{\rho_p}) \Rightarrow I_\sigma$.
>
> *Interference Freedom*: For every pair of distinct operations σ, τ in Ω that belong to different processes: τ leaves $\mathit{in}(\sigma) \wedge I_\sigma \wedge J_\tau$ and $\mathit{after}(\sigma) \wedge I'_\sigma \wedge J_\tau$ invariant.

The proof that these conditions imply the invariance of I is similar to the proof for the atomic Owicki-Gries conditions.

By part (a) of the sequential correctness condition, each operation τ in Ω leaves J_τ invariant. Therefore, OP2 implies that to prove the interference-freedom condition for the pair σ, τ, it suffices to prove that τ leaves $\mathit{in}(\sigma) \wedge I_\sigma$ and $\mathit{after}(\sigma) \wedge I'_\sigma$ invariant. Since σ and τ are in different processes, τ leaves $\mathit{in}(\sigma)$ and $\mathit{after}(\sigma)$ invariant (by CTL4). Hence by OP2, to prove this interference-freedom condition, it also suffices to prove that τ leaves I_σ and I'_σ invariant.

For an atomic operation ξ, the formula $\{I_\xi\}\,\xi\,\{I'_\xi\}$ is equivalent to the assertion that ξ leaves J_ξ invariant. Hence, if all operations are atomic, the nonatomic sequential-correctness condition is equivalent to the atomic Owicki-Gries condition. If σ and τ are atomic operations, the presence of the $\mathit{in}(\sigma)$ and $\mathit{after}(\sigma)$ conjuncts makes this nonatomic interference-freedom condition somewhat weaker than the atomic Owicki-Gries condition.

4 Applications

4.1 The Single-Access Rule

It is usually assumed that an operation may be treated as atomic if it contains at most one access to a shared variable. We call this assumption the *single-access rule*. It was first published by Owicki and Gries in [14], but probably qualifies as a folk theorem [7]. In the traditional method of reasoning about a concurrent program, one first applies the single-access rule to replace the program with one containing larger atomic operations and then applies the atomic Owicki-Gries method to the new program. We will indicate with an example how the *win* formalism allows one to use the nonatomic Owicki-Gries method to reason about the original program without using the single-access rule to change the grain of atomicity.

$$
\begin{array}{l}
\textbf{var}\ x,\ y\text{: }\textbf{array}\ 1\ldots\text{n}\ \textbf{of integer};\\
\quad\ m\text{:}\quad \textbf{integer};\\
\textbf{cobegin}\,\Box_{i=1\ldots n}\ \alpha_i\text{: } \langle\, m \quad := \max(m, x[i])\,;\\
\qquad\qquad\qquad\qquad\ \ y[i] := x[i] \qquad\qquad \rangle\ \{m \geq y[i]\}\\
\textbf{coend}
\end{array}
$$

Figure 4: Annotation of a program obtained with the single-action rule.

$$
\begin{array}{rl}
& \textbf{var}\ x,\ y\text{: }\textbf{array}\ 1\ldots\text{n}\ \textbf{of integer};\\
& \quad\ m\text{:}\quad \textbf{integer};\\
& \textbf{cobegin}\,\Box_{i=1\ldots n}\ \xi_i\text{: } \langle m \quad := \max(m, x[i])\,\rangle\ \{m \geq x[i]\};\\
\{winp(\psi_i, m \geq y[i])\} & \psi_i\text{: } y[i]\ := x[i] \qquad\qquad \{m \geq y[i]\}\\
& \textbf{coend}
\end{array}
$$

Figure 5: Annotation of the original program.

The single-access rule is based upon the assumption that any access to a shared variable is atomic, which may not always be the case. (For example, the variable may be implemented as two words of memory, with access to each word being a separate action.) A more precise formulation of the single-access rule is that if $\theta;\, \xi;\, \psi$ appears in a program, ξ is atomic, and θ and ψ are operations that do not access any set of variables that are not local to the process containing them, then $\theta;\, \xi;\, \psi$ may be considered a single atomic operation.

Any Owicki-Gries method proof of a program transformed with the single-access rule can be turned into a proof of the original program. However, proving this result in general is rather tedious and requires properties of *win* and *sin* that we have not introduced. Instead, we illustrate the result with an example—namely, the annotated program of Figure 4, which is obtained by applying the single-action rule to combine

$$
\begin{array}{l}
\xi_i\text{: } \langle\, m \quad := \max(m, x[i])\,\rangle\\
\psi_i\text{: } \quad y[i] := x[i]
\end{array}
$$

into the one atomic operation α_i. (In this program, m is the only nonlocal variable.) It is easy to prove the invariance of this annotation, from which one can deduce that $m \geq \max(y[1], \ldots, y[n])$ holds upon termination.

Instead of applying the single-action rule, we apply the nonatomic Owicki-Gries method directly to the annotated program of Figure 5. We give a more detailed proof than is warranted by the example in order to illustrate the decomposition into simple steps that is the hallmark of the Owicki-Gries method.

Proof of Sequential Correctness—(a)

We must show that every operation σ leaves J_σ invariant. (Recall that J_σ equals $(in(\sigma) \wedge I_\sigma) \vee (after(\sigma) \wedge I'_\sigma)$.) There are two cases to check: $\sigma = \xi_i$ and $\sigma = \psi_i$.

ξ_i: An atomic action σ leaves J_σ invariant iff $\{I_\sigma\}\, \sigma\, \{I'_\sigma\}$. We must therefore prove $\{true\}\, \xi_i\, \{m \geq x[i]\}$, which follows from the usual rules for Hoare triples.

ψ_i: The invariance of J_{ψ_i} follows immediately from WINP4 (substituting *true* for P).

Proof of Sequential Correctness—(b)

We must show that for every operation σ: if $\rho_1, \ldots, \rho_m$ are the predecessors of σ, then $\models at(\sigma) \wedge \bigwedge_p I'_{\rho_p} \Rightarrow I_\sigma$. Again, there are two choices of σ to consider.

ξ_i: This condition is vacuous, since ξ_i has no predecessors. (Formally, the condition holds because the conjunction of an empty set of predicates equals *false*.)

ψ_i: Since ξ_i is the only predecessor of ψ_i, we must prove

$$\models (at(\psi_i) \wedge (m \geq x[i])) \;\Rightarrow\; winp(\psi_i, m \geq y[i])$$

This formula follows from WINP1, since a simple *wlp* calculation shows that $at(\psi_i) \wedge wlp(\psi_i, m \geq y[i])$ equals $at(\psi_i) \wedge (m \geq x[i])$.

Proof of Interference Freedom

For each operation τ and each operation σ in a different process from τ, we must prove that τ leaves $in(\sigma) \wedge I_\sigma \wedge J_\tau$ and $after(\sigma) \wedge I'_\sigma \wedge J_\tau$ invariant. As we observed in Section 3.8, it suffices to prove that τ leaves I_σ and I'_σ invariant.

Proof for $\tau = \xi_k$. There are two choices of σ to be checked—namely, ξ_i and ψ_i, with $i \neq k$.

ξ_i: Operation ξ_k obviously leaves I_{ξ_i} invariant, since I_{ξ_i} equals *true*. (Formally, this follows from OP1.) To prove that ξ_k leaves I'_{ξ_i} invariant, we must show that $\{m \geq x[i]\}\, \xi_k\, \{m \geq x[i]\}$ holds, which follows from the usual rules for reasoning about Hoare triples.

ψ_i: $\boxed{1}$ ξ_k leaves I'_{ψ_i} invariant.

PROOF: We must show that $\{m \geq y[i]\}\, \xi_k\, \{m \geq y[i]\}$ holds, which follows by ordinary reasoning about Hoare triples.

2. ξ_k commutes with ψ_i.

PROOF: By the Assignment Rule and OP5.

3. ξ_k leaves $\neg$*after*(ψ_i) invariant.
   PROOF: By CTL4.
4. ξ_k leaves *after*$(\psi_i) \Rightarrow I'_{\psi_i}$ invariant.
   PROOF: By 1, 3, and OP2.
5. $\boxed{5}$ ξ_k leaves I_{ψ_i} invariant.
   PROOF: By 2, 4, and WIN3, since I_{ψ_i} equals $win(\psi_i, \textit{after}(\psi_i) \Rightarrow I'_{\psi_i})$.

Proof for $\tau = \psi_k$. We have the same two choices for σ.

ξ_i: Operation ψ_k obviously leaves I_{ξ_i} invariant, since $I_{\xi_i} \equiv true$. By the Assignment Rule and OP1, it leaves I'_{ξ_i} invariant

ψ_i: The Assignment Rule and OP1 imply that ψ_k leaves I'_{ψ_i} invariant (since $i \neq k$). The proof that it leaves I_{ψ_i} invariant is similar to the proof for $\tau = \xi_k$.

4.2 The Bakery Algorithm

We now prove the correctness of the original bakery algorithm, shown in Figure 3. More precisely, we prove that this algorithm is correct if two additional assumptions are made about it. Our inability to verify the correctness of the original algorithm will lead to the discovery of the necessary assumptions. These assumptions will be discussed later, after the proof.

We have already given rules for reasoning about nonatomic assignment and **await** statements. The bakery algorithm also contains the nonatomic critical and noncritical sections, for which we make the following obvious assumption.

Section Hypothesis In the bakery algorithm of Figure 3, a cs_i or ncs_i operation neither modifies nor accesses any set of variables that contains neither any $num[j]$, nor any $c[j]$, nor any variable local to the operation.

4.2.1 Almost a Proof

In the Owicki-Gries method, the key to the proof is finding an invariant annotation. In practice, the annotation is obtained by a method of trial and error that can be viewed as an attempt to approximate a weakest invariant. We begin with an informal derivation of an invariant annotation for the bakery algorithm. After obtaining the annotation, we use the Owicki-Gries method to prove its invariance. This is an idealized presentation; in reality, derivation and proof of the annotation go hand in hand.

We start with the predicate I_{cs_i}, which is true when control is in process i's critical section. The truth of I_{cs_i} must imply that no other process j is in its critical section. The structure of the program suggests that we let I_{cs_i} equal $\bigwedge_{j \neq i} I'_{\eta_{ij}}$, so we look next at $I'_{\eta_{ij}}$.

The basic idea of the algorithm is that process i enters its critical section only when $num[i] > 0$ and $i \ll j$. Mutual exclusion is guaranteed because $num[i] > 0$ and $i \ll j$ imply that $j \not\ll i$. Letting N_i denote the predicate $num[i] > 0$, our first guess for $I'_{\eta_{ij}}$ is $N_i \wedge (i \ll j)$.

This choice of $I'_{\eta_{ij}}$ does not satisfy the interference-freedom condition for β_j or ρ_j (the condition with σ equal to η_{ij}, and τ equal to β_j or ρ_j), since $num[j]$ can assume arbitrary values during execution of the operations β_j and ρ_j. In such a case, the standard approach is either to strengthen $I'_{\eta_{ij}}$ to imply that control is not in β_j or ρ_j, or else to weaken it to be true whenever control is in those operations. Since process j can execute β_j after process i has executed η_{ij}, strengthening $I'_{\eta_{ij}}$ won't work; we must weaken it. We weaken $I'_{\eta_{ij}}$ to require only that $i \ll j$ hold while control in process j is after β_j and before ρ_j. This is still strong enough to guarantee mutual exclusion when we take I_{cs_i} to be $\bigwedge_{j \neq i} I'_{\eta_{ij}}$. Let Q_{ij} be the predicate asserting that if control is in γ_j, δ_j, or cs_j, then $i \ll j$. Our next guess at $I'_{\eta_{ij}}$ is $N_i \wedge Q_{ij}$.

Our choice of $I'_{\eta_{ij}}$ still does not satisfy the interference-freedom condition for β_j because β_j puts control at γ_j without necessarily ensuring that $i \ll j$. We must strengthen $I'_{\eta_{ij}}$ by conjoining a predicate to ensure that $i \ll j$ if executing β_j leaves control at γ_j. Since $winp(\beta_j, i \ll j)$ is the predicate asserting that $i \ll j$ holds upon completion of β_j, we conjoin the predicate $in(\beta_j) \Rightarrow winp(\beta_j, i \ll j)$, which we denote by P'_{ij}. We thus choose $N_i \wedge P'_{ij} \wedge Q_{ij}$ for $I'_{\eta_{ij}}$. A quick check shows that this $I'_{\eta_{ij}}$ seems to be left invariant by every operation of process j.

The standard approach is to work backwards through the program, so we now choose $I_{\eta_{ij}}$. Since we know nothing about the atomic operations that constitute η_{ij}, we are forced to let $I_{\eta_{ij}}$ equal $winp(\eta_{ij}, I'_{\eta_{ij}})$ in order to satisfy part (a) of the Sequential Correctness Condition. We continue working backwards and now try to find $I'_{\epsilon_{ij}}$.

Part (b) of the Sequential Correctness Condition states that $at(\eta_{ij}) \wedge I'_{\epsilon_{ij}}$ implies $I_{\eta_{ij}}$, which equals $winp(\eta_{ij}, N_i \wedge P'_{ij} \wedge Q_{ij})$. By WINP1, $I'_{\epsilon_{ij}}$ must therefore imply $wlp(\eta_{ij}, N_i \wedge P'_{ij} \wedge Q_{ij})$. In the absence of concurrent activity, $i \ll j$ must hold upon completion of η_{ij}, so executing η_{ij} makes Q_{ij} true. In other words, $wlp(\eta_{ij}, Q_{ij})$ is identically true. Since executing η_{ij} doesn't change N_i or P'_{ij}, we see that $wlp(\eta_{ij}, N_i \wedge P'_{ij} \wedge Q_{ij})$ equals $N_i \wedge P'_{ij}$, which becomes our natural choice for $I'_{\epsilon_{ij}}$.

Continuing backwards in this way, we let $I_{\epsilon_{ij}}$ equal $winp(\epsilon_{ij}, I'_{\epsilon_{ij}})$ and choose I'_{γ_i} so it implies $wlp(\epsilon_{ij}, N_i \wedge P'_{ij})$. Since ϵ_{ij} does not change $num[i]$, we see that $wlp(\epsilon_{ij}, N_i \wedge P'_{ij})$ equals $N_i \wedge wlp(\epsilon_{ij}, P'_{ij})$. If $wlp(\epsilon_{ij}, P'_{ij})$ were identically true, then we could let I'_{γ_i} equal N_i, which obviously holds after process i has executed β_i and γ_i. Unfortunately, $wlp(\epsilon_{ij}, P'_{ij})$ is not identically true; just looking at ϵ_{ij} gives us no reason to believe that P'_{ij} will be true after executing it.

Simply manipulating formulas will take us no further; we must think about why the algorithm works. The predicate P'_{ij} asserts that if β_j is currently executing, then running it to completion will set $num[j]$ to a value that makes $i \ll j$ true. We expect P'_{ij} to be true after executing ϵ_{ij} because ϵ_{ij} terminates only when it finds $c[j]$ false, and $c[j]$ is true when control is in statement β_j. This suggests replacing P'_{ij} by the weaker predicate $(in(\beta_j) \wedge c[j]) \Rightarrow winp(\beta_j, i \ll j)$, which we denote P_{ij}. A complete execution of ϵ_{ij} terminates only when $c[j]$ is false, so $wlp(\epsilon_{ij}, P_{ij})$ is identically true and we can satisfy the requirement that I'_{γ_i} implies $wlp(\epsilon_{ij}, N_i \wedge P_{ij})$ by letting I'_{γ_i} equal N_i. Of course, we must also make sure that replacing P'_{ij} by P_{ij} does not invalidate any of the conditions we have already checked.

The rest of the derivation is straightforward, so we stop now and define the complete annotation. First, recall that the predicates N_i, P_{ij}, and Q_{ij}, for $i \neq j$, are defined as follows:

$$
\begin{array}{rcl}
N_i & \equiv & num[i] > 0 \\
P_{ij} & \equiv & (in(\beta_j) \wedge c[j]) \Rightarrow winp(\beta_j, i \ll j) \\
Q_{ij} & \equiv & (in(\gamma_j) \vee in(\delta_j) \vee in(cs_j)) \Rightarrow i \ll j
\end{array}
$$

where $in(\delta_j)$ is defined to equal $\bigvee_l in(\epsilon_{kl}) \vee in(\eta_{kl})$. The predicates of the annotation are defined below. Each I_σ that contains a *winp* is equal to $winp(\sigma, I'_\sigma)$, but WINP3 has been used to write some of these predicates in a more convenient form.

$$
\begin{array}{ll}
I_{ncs_i} \equiv true & I'_{ncs_i} \equiv true \\
I_{\alpha_i} \equiv winp(\alpha_i, c[i]) & I'_{\alpha_i} \equiv c[i] \\
I_{\beta_i} \equiv c[i] \wedge winp(\beta_i, N_i) & I'_{\beta_i} \equiv c[i] \wedge N_i \\
I_{\gamma_i} \equiv N_i & I'_{\gamma_i} \equiv N_i \\
I_{\epsilon_{ij}} \equiv N_i \wedge winp(\epsilon_{ij}, P_{ij}) & I'_{\epsilon_{ij}} \equiv N_i \wedge P_{ij} \\
I_{\eta_{ij}} \equiv N_i \wedge P_{ij} \wedge winp(\eta_{ij}, Q_{ij}) & I'_{\eta_{ij}} \equiv N_i \wedge P_{ij} \wedge Q_{ij} \\
I_{cs_i} \equiv N_i \wedge \bigwedge_{j \neq i} P_{ij} \wedge Q_{ij} & I'_{cs_i} \equiv N_i \wedge \bigwedge_{j \neq i} P_{ij} \wedge Q_{ij} \\
I_{\rho_i} \equiv true & I'_{\rho_i} \equiv true
\end{array}
$$

The predicate defined by the annotation is clearly true in the initial state and, since I_{cs_i} and I_{cs_j} cannot both be true if $i \neq j$, it implies the mutual exclusion condition. We now attempt to prove the invariance of this annotation using the nonatomic Owicki-Gries method.

Proof of Sequential Correctness—(a)

We must prove that each operation σ leaves J_σ invariant.

ncs_i: Since I_{ncs_i} and I'_{ncs_i} both equal *true*, OP4 implies that J_{ncs_i} is left invariant by ncs_i.

α_i: WINP4 implies that α_i leaves J_{α_i} invariant.

β_i: The Assignment Rule and OP1 imply that β_i leaves $c[i]$ invariant, and WINP4 then implies that β_i leaves J_{β_i} invariant.

γ_i: The Assignment Rule and OP1 imply that γ_i leaves N_i invariant, so OP4 implies that γ_i leaves J_{γ_i} invariant.

ϵ_{ij}: The Await Rule and OP1 imply that ϵ_{ij} leaves N_i invariant, so WINP4 implies that ϵ_{ij} leaves $J_{\epsilon_{ij}}$ invariant.

η_{ij}: 1. η_{ij} leaves N_i invariant.
PROOF: By the Await Rule and OP1.

2. η_{ij} leaves $winp(\beta_j, i \ll j)$ invariant.
PROOF: The Await and Assignment Rules and WIN5 imply that $winp(\beta_j, i \ll j)$ does not access the set of variables modified by η_{ij}, so OP1 implies that η_{ij} leaves $winp(\beta_j, i \ll j)$ invariant.

3. η_{ij} leaves $\neg(in(\beta_j) \wedge c[j])$ invariant.
PROOF: CTL4 implies that η_{ij} leaves $\neg in(\beta_{ij})$ invariant. The Await Rule and OP1 imply that it leaves $\neg c[j]$ invariant. Rule OP2 then implies that η_{ij} leaves $\neg(in(\beta_j) \wedge c[j])$ invariant.

4. η_{ij} leaves P_{ij} invariant.
PROOF: By 2, 3, and OP2.

$\boxed{5}$ η_{ij} leaves $J_{\eta_{ij}}$ invariant.
PROOF: By 1, 4, OP2, and WINP4.

cs_i: The Section Hypothesis and OP1 imply that cs_i leaves I_{cs_i} invariant, so OP4 implies that it leaves J_{cs_i} invariant.

ρ_i: By OP4, since I_{ρ_i} and I'_{ρ_i} both equal *true*.

Proof of Sequential Correctness—(b)

We must show that for every operation σ: if $\rho_1, \ldots, \rho_m$ are the predecessors of σ, then $\models at(\sigma) \wedge \bigwedge_p I'_{\rho_p} \Rightarrow I_\sigma$. There are eight choices of σ to consider.

ncs_i: $\models (at(ncs_i) \wedge I'_{\rho_i}) \Rightarrow I_{ncs_i}$, is trivially true, since $I_{ncs_i} \equiv true$.

α_i : 1. $\models at(\alpha_i) \wedge winp(\alpha_i, c[i]) \equiv at(\alpha_i) \wedge wlp(\alpha_i, c[i])$
PROOF: By WINP1.

2. $\models wlp(\alpha_i, c[i]) \equiv true$
PROOF: By an elementary *wlp* calculation.

$\boxed{3}$ $\models (at(\alpha_i) \wedge I'_{ncs_i}) \Rightarrow I_{\alpha_i}$
PROOF: By 1, 2 and the definition of I_{α_i}

β_i: Similar to the proof for α_i.

γ_i: $\models I'_{\beta_i} \Rightarrow I_{\gamma_i}$ follows immediately from the definitions of I'_{β_i} and I_{γ_i}.

ϵ_{ij}: 1. $\models at(\epsilon_{ij}) \wedge winp(\epsilon_{ij}, P_{ij}) \equiv at(\epsilon_{ij}) \wedge wlp(\epsilon_{ij}, P_{ij})$
PROOF: By WINP1.

2. $\models wlp(\epsilon_{ij}, \neg c[j]) \Rightarrow wlp(\epsilon_{ij}, P_{ij})$
PROOF: By WLP2, since the definition of P_{ij} implies $\models (\neg c[j]) \Rightarrow P_{ij}$.

3. $\models wlp(\epsilon_{ij}, \neg c[j]) \equiv true$
PROOF: By an elementary *wlp* calculation.

4. $\models at(\epsilon_{ij}) \Rightarrow winp(\epsilon_{ij}, P_{ij})$
PROOF: By 1, 2, and 3.

$\boxed{5}$ $\models (at(\epsilon_{ij}) \wedge I'_{\gamma_i}) \Rightarrow I_{\epsilon_{ij}}$
PROOF: By 4 and the definitions of I'_{γ_i} and $I_{\epsilon_{ij}}$.

η_{ij}: 1. $\models at(\eta_{ij}) \wedge winp(\eta_{ij}, Q_{ij}) \equiv at(\eta_{ij}) \wedge wlp(\eta_{ij}, Q_{ij})$
PROOF: By WINP1.

2. $\models wlp(\eta_{ij}, i \ll j) \Rightarrow wlp(\eta_{ij}, Q_{ij})$
PROOF: By WLP2, since $\models (i \ll j) \Rightarrow Q_{ij}$.

3. $\models wlp(\eta_{ij}, i \ll j) \equiv true$
PROOF: By an elementary *wlp* calculation.

4. $\models at(\eta_{ij}) \Rightarrow winp(\eta_{ij}, Q_{ij})$
PROOF: By 1, 2, and 3.

$\boxed{5}$ $\models (at(\eta_{ij}) \wedge I'_{\epsilon_{ij}}) \Rightarrow I_{\eta_{ij}}$
PROOF: By 4 and the definitions of $I'_{\epsilon_{ij}}$ and $I_{\eta_{ij}}$.

cs_i: $\models (at(cs_i) \wedge \bigwedge_{j \neq i} I'_{\eta_{ij}}) \Rightarrow I_{cs_i}$ follows immediately from the definitions of $I'_{\eta_{ij}}$ and I_{cs_i}.

ρ_i: $\models (at(\rho_i) \wedge I'_{cs_i}) \Rightarrow I_{\rho_i}$ obviously holds, since I_{ρ_i} equals *true*.

Proof of Interference Freedom

For each operation τ and each operation σ in a different process from τ, we must prove that τ leaves both $in(\sigma) \wedge I_\sigma \wedge J_\tau$ and $after(\sigma) \wedge I'_\sigma \wedge J_\tau$ invariant. As observed in Section 3.8, to prove that τ leaves $in(\sigma) \wedge I_\sigma \wedge J_\tau$ invariant, it suffices to prove that it leaves either $I_\sigma \wedge J_\tau$ or simply I_σ invariant, and similarly for $after(\sigma) \wedge I'_\sigma \wedge J_\tau$.

Proof for $\tau = ncs_k$. We begin by proving that ncs_k leaves invariant the "primitive" predicates, such as P_{ij}, that appear in the annotation. Predicate P_{ij} is a little trickier than the rest because it contains a *winp* formula. Also, since P_{ij} and Q_{ij} mention the control state of process j, which is changed by ncs_j, the case $k = j$ requires special consideration.

NC1. Operation ncs_k leaves $c[i]$, N_i, and Q_{ij} invariant, for $i \neq k$ and $j \neq k$.

PROOF: This follows from the Section Hypothesis and OP1.

NC2. Operation ncs_k leaves P_{ij} invariant, for $i \neq k$ and $j \neq k$.

PROOF: Operation ncs_k leaves $winp(\beta_j, i \ll j)$ invariant by the Section Hypothesis, the Assignment Rule, and WIN5. It leaves $\neg in(\beta_j)$ invariant by CTL4, and $\neg c[j]$ invariant by the Section Hypothesis and OP1. Rule OP2 then implies that ncs_k leaves P_{ij} invariant.

NC3. Operation ncs_j leaves Q_{ij} invariant, for $i \neq j$.

PROOF: Reasoning about control predicates implies

$$\models (in(ncs_j) \lor after(ncs_j)) \land (in(\gamma_j) \lor in(\delta_j) \lor in(cs_j)) \equiv false$$

Hence, $(in(ncs_j) \lor after(ncs_j)) \land Q_{ij}$ is identically *true*, so OP4 implies that ncs_j leaves Q_{ij} invariant.

NC4. Operation ncs_j leaves P_{ij} invariant, for $i \neq j$.

PROOF: The proof is similar to that of NC3.

Using these four results, we can prove that ncs_k leaves I_σ and I'_σ invariant, for each operation σ in process i, where $i \neq k$. If I_σ contains no *winp* expression, then invariance follows easily from NC1–NC4. The proofs for all σ containing a *winp* expression are similar to the proof for $\sigma = \epsilon_{ij}$, which is given below.

ϵ_{ij}: 1. ncs_k leaves P_{ij} invariant.

PROOF: By NC2 and NC4.

2. ncs_k leaves $after(\epsilon_{ij}) \Rightarrow P_{ij}$ invariant.

PROOF: By 1, CTL4, and OP2.

3. ncs_k commutes with ϵ_{ij}.

PROOF: By the Section Hypothesis, the Await Rule, and OP5.

4. ncs_k leaves $winp(\epsilon_{ij}, P_{ij})$ invariant.

PROOF: By 2, 3, WIN3, and the definition of *winp*.

$\boxed{5}$ ncs_k leaves $I_{\epsilon_{ij}}$ invariant.

PROOF: By 4, NC1, and OP2.

$\boxed{6}$ ncs_k leaves $I'_{\epsilon_{ij}}$ invariant.

PROOF: By 1, NC1, and OP2.

Proof for $\tau = \alpha_k$. We begin by proving the invariance results for α_k that are the analogs of NC1–NC4. The proofs of $\alpha 1$–$\alpha 3$ are similar to the proofs of NC1–NC3 and are omitted. The strict analog of NC4 does not hold, since α_j does not leave P_{ij} invariant. However, in the annotation, P_{ij} always appears conjoined with N_i, so it suffices to prove that α_j leaves $N_i \wedge P_{ij}$ invariant.

$\alpha 1$. Operation α_k leaves $c[i]$, N_i, and Q_{ij} invariant, for $i \neq k$ and $j \neq k$.

$\alpha 2$. Operation α_k leaves P_{ij} invariant, for $i \neq k$ and $j \neq k$.

$\alpha 3$. Operation α_j leaves Q_{ij} invariant, for $i \neq j$.

$\alpha 4$. Operation α_j leaves $N_i \wedge P_{ij}$ invariant, for $i \neq j$.

1. $\models (\mathit{in}(\alpha_j) \vee \mathit{after}(\alpha_j)) \wedge \mathit{in}(\beta_j) \equiv \mathit{at}(\beta_j)$
   PROOF: By reasoning about the control state.
2. $\models N_i \Rightarrow wlp(\beta_j, i \ll j)$
   PROOF: By elementary reasoning about *wlp*.
3. $\models (N_i \wedge \mathit{at}(\beta_j)) \Rightarrow winp(\beta_j, i \ll j)$
   PROOF: By 2 and WINP1.
4. $\models ((\mathit{in}(\alpha_j) \vee \mathit{after}(\alpha_j)) \wedge N_i \wedge \mathit{in}(\beta_j) \wedge c[j]) \Rightarrow winp(\beta_j, i \ll j)$
   PROOF: By 1 and 3.
5. $\models (\mathit{in}(\alpha_j) \vee \mathit{after}(\alpha_j)) \wedge N_i \wedge P_{ij} \equiv (\mathit{in}(\alpha_j) \vee \mathit{after}(\alpha_j)) \wedge N_i$
   PROOF: By 4 and the definition of P_{ij}, since $\models (A \wedge B) \Rightarrow C$ implies $\models A \wedge (B \Rightarrow C) \equiv A$. (Substitute P_{ij} for $B \Rightarrow C$.)
6. α_j leaves $(\mathit{in}(\alpha_j) \vee \mathit{after}(\alpha_j)) \wedge N_i$ invariant.
   PROOF: By $\alpha 1$ and OP4.

$\boxed{7}$ α_j leaves $N_i \wedge P_{ij}$ invariant.
   PROOF: By 5, 6, and OP4.

We can now prove that α_k leaves I_σ and I'_σ invariant for all the operations σ in process i, where $i \neq k$. Only the proofs for σ equal to β_i and ϵ_{ij} are given; the rest are similar or else follow easily from $\alpha 1$–$\alpha 4$.

β_i: $\boxed{1}$ α_k leaves I'_{β_i} invariant.
   PROOF: By $\alpha 1$ and OP2.

2. α_k leaves $\mathit{after}(\beta_i) \Rightarrow N_i$ invariant.
   PROOF: By $\alpha 1$, CTL4, and OP2.
3. α_k and β_i commute.
   PROOF: By the Assignment Rule and OP5.

4. α_k leaves $winp(\beta_i, N_i)$ invariant.
   PROOF: By 2, 3, WIN3, and the definition of *winp*.

$\boxed{5}$ α_k leaves I_{β_i} invariant.
   PROOF: By 4, $\alpha 1$, and OP2.

ϵ_{ij}: We consider separately the two cases $j \neq k$ and $j = k$. The proof for $j \neq k$ is as follows.

$\boxed{1}$ α_k leaves $I'_{\epsilon_{ij}}$ invariant.
   PROOF: α_k leaves N_i invariant by $\alpha 1$, and it leaves P_{ij} invariant by $\alpha 2$, so OP2 implies that it leaves $N_i \wedge P_{ij}$ invariant.

2. α_k leaves $after(\epsilon_{ij}) \Rightarrow P_{ij}$ invariant.
   PROOF: By $\alpha 2$, CTL4, and OP2.

3. α_k commutes with ϵ_{ij}.
   PROOF: By the Assignment and Await Rules (since $j \neq k$) and OP5.

4. α_k leaves $winp(\epsilon_{ij}, P_{ij})$ invariant.
   PROOF: By 2, 3, WIN3, and the definition of *winp*.

$\boxed{5}$ α_k leaves $I_{\epsilon_{ij}}$ invariant.
   PROOF: By 4, $\alpha 1$, and OP2.

We now consider the case $j = k$.

$\boxed{1}$ α_j leaves $I'_{\epsilon_{ij}}$ invariant.
   PROOF: By $\alpha 4$.

2. ϵ_{ij} leaves P_{ij} unchanged.
   PROOF: By the Await Rule and OP1.

3. $\models win(\epsilon_{ij}, P_{ij} \vee \neg after(\epsilon_{ij})) \equiv P_{ij} \vee win(\epsilon_{ij}, \neg after(\epsilon_{ij}))$
   PROOF: By 2 and WIN4.

4. $\models winp(\epsilon_{ij}, P_{ij}) \equiv P_{ij} \vee winp(\epsilon_{ij}, true)$
   PROOF: By 3 and the definition of *winp*.

5. $\models winp(\epsilon_{ij}, P_{ij}) \equiv P_{ij} \vee (\neg in(\alpha_j) \wedge winp(\epsilon_{ij}, true))$
   PROOF: By 4 and propositional logic, since $\models in(\alpha_j) \Rightarrow \neg in(\beta_j)$ implies $\models in(\alpha_j) \Rightarrow P_{ij}$.

6. $\models I_{\epsilon_{ij}} \equiv (N_i \wedge P_{ij}) \vee (\neg in(\alpha_j) \wedge \ldots)$
   PROOF: By 5.

$\boxed{7}$ α_j leaves $I_{\epsilon_{ij}}$ invariant.
   PROOF: By 6, $\alpha 4$, OP3, and OP2.

Proof for $\tau = \beta_k$. We begin with the analogs of NC1–NC4. The analog of NC3 isn't valid because β_j does not leave Q_{ij} invariant. Since Q_{ij} always appears in conjunction with P_{ij}, it would suffice to prove that β_j leaves $P_{ij} \wedge Q_{ij}$ invariant—but it doesn't. However, to prove interference freedom, it suffices to show that β_j leaves $J_{\beta_j} \wedge P_{ij} \wedge Q_{ij}$ invariant.

$\beta 1$. Operation β_k leaves $c[i]$, N_i, and Q_{ij} invariant, for $i \neq k$ and $j \neq k$.

PROOF: Follows from the Assignment Rule and OP1.

$\beta 2$. Operation β_k leaves P_{ij} invariant, for $i \neq k$ and $j \neq k$.

1. β_k leaves $\neg(\mathit{in}(\beta_j) \wedge c[j])$ invariant.
   PROOF: β_k leaves $\neg \mathit{in}(\beta_j)$ invariant by CTL4 and it leaves $\neg c[j]$ invariant by the Assignment Rule and OP1, so OP2 implies that it leaves $\neg(\mathit{in}(\beta_j) \wedge c[j])$ invariant.
2. β_k leaves $winp(\beta_j, i \ll j)$ invariant.

The crucial fact that β_k leaves $winp(\beta_j, i \ll j)$ invariant cannot be proved. It must be assumed as an additional hypothesis. This assumption is discussed later.

3. β_k leaves P_{ij} invariant.
   PROOF: By 1, 2, and OP2.

$\beta 3$. Operation β_j leaves $J_{\beta_j} \wedge P_{ij} \wedge Q_{ij}$ invariant, for $i \neq j$.

1. $\models J_{\beta_j} \equiv (\mathit{in}(\beta_j) \vee \mathit{after}(\beta_j)) \wedge c[j] \wedge winp(\beta_j, N_j)$
   PROOF: By WINP2 and the definition of J_{β_j}.
2. $\models (\mathit{in}(\beta_j) \vee \mathit{after}(\beta_j)) \wedge Q_{ij} \equiv (\mathit{in}(\beta_j) \vee \mathit{after}(\beta_j)) \wedge (\neg \mathit{after}(\beta_j) \vee (\mathit{after}(\beta_j) \wedge winp(\beta_j, i \ll j)))$
   PROOF: By WINP2 and propositional logic, since
   $$\models (\mathit{in}(\beta_j) \vee \mathit{after}(\beta_j)) \wedge (\mathit{in}(\gamma_j) \vee \mathit{in}(\delta_j) \vee \mathit{in}(cs_j)) \equiv \mathit{after}(\beta_j)$$
3. $\models (\mathit{in}(\beta_j) \vee \mathit{after}(\beta_j)) \wedge c[j] \wedge P_{ij} \wedge Q_{ij} \equiv (\mathit{in}(\beta_j) \vee \mathit{after}(\beta_j)) \wedge c[j] \wedge winp(\beta_j, i \ll j)$
   PROOF: By 2 and propositional logic, using $\models \mathit{after}(\beta_j) \Rightarrow \neg \mathit{in}(\beta_j)$.
4. $\models J_{\beta_j} \wedge P_{ij} \wedge Q_{ij} \equiv J_{\beta_j} \wedge winp(\beta_j, i \ll j)$
   PROOF: By 1, 3, and propositional logic.

$\boxed{5}$ β_j leaves $J_{\beta_j} \wedge P_{ij} \wedge Q_{ij}$ invariant.
   PROOF: By 4 and OP2, since the sequential correctness proof showed that β_j leaves J_{β_j} invariant, and the definition of *winp* implies that β_j leaves $winp(\beta_j, i \ll j)$ invariant.

$\beta 4$. Operation β_j leaves P_{ij} invariant.

1. β_j leaves $\neg \mathit{in}(\beta_j)$ invariant.
   PROOF: By OP3.
2. β_j leaves $\neg c[j]$ invariant.
   PROOF: By the Assignment Rule and OP1.
3. β_j leaves $winp(\beta_j, i \ll j)$ invariant.
   PROOF: By the definition of *win*.

$\boxed{4}$ β_j leaves P_{ij} invariant.
   PROOF: By 1, 2, 3, OP2, and the definition of P_{ij}.

We must now prove the interference-freedom condition for $\tau = \beta_k$ and all σ in process i, with $i \neq k$. For most operations σ, the proof is essentially the same as for $\tau = \alpha_k$. When $\sigma = \epsilon_{ij}$, the proof for $\tau = \beta_k$ is simpler than the proof for $\tau = \alpha_k$, since β_j commutes with ϵ_{ij} and α_j does not. However, the proof for $\sigma = \eta_{ij}$ is is trickier because β_j does not commute with η_{ij}. We consider the interference-freedom proofs for $\tau = \beta_k$ only when σ equals β_i and η_{ij}, with $i \neq k$.

β_i: We must prove that β_k leaves I_{β_i} and I'_{β_i} invariant. The invariance of I'_{β_i} follows immediately from $\beta 1$. However, to prove that β_k leaves I_{β_i} invariant, we must show that it leaves $winp(\beta_i, num[i] > 0)$ invariant. This cannot be done. **We must assume that β_k leaves $winp(\beta_i, num[i] > 0)$ invariant.** This assumption is discussed later.

η_{ij}: We must prove that β_k leaves $J_{\beta_j} \wedge I_{\eta_{ij}}$ and $J_{\beta_j} \wedge I'_{\eta_{ij}}$ invariant. The proof when $k \neq j$ is similar to the proof for $\tau = \alpha_k$ and $\sigma = \epsilon_{ij}$ given above. We consider only the case when $k = j$.

$\boxed{1}$ β_j leaves $J_{\beta_j} \wedge I'_{\eta_{ij}}$ invariant.
   PROOF: By $\beta 1$, $\beta 3$, and OP2.

2. η_{ij} leaves Q_{ij} unchanged.
   PROOF: By the Await Rule and OP1.
3. $\models winp(\eta_{ij}, Q_{ij}) \equiv Q_{ij} \vee winp(\eta_{ij}, true)$
   PROOF: By 2 and WIN4, which imply that $win(\eta_{ij}, Q_{ij} \vee \neg \mathit{after}(\eta_{ij}))$ equals $Q_{ij} \vee win(\eta_{ij}, \neg \mathit{after}(\eta_{ij}))$.
4. $\models winp(\eta_{ij}, Q_{ij}) \equiv Q_{ij} \vee (\neg \mathit{in}(\beta_j) \wedge winp(\eta_{ij}, true))$
   PROOF: By 3 and predicate calculus reasoning, since $\models \mathit{in}(\beta_j) \Rightarrow \neg(\mathit{in}(\gamma_j) \vee \mathit{in}(\delta_j) \vee \mathit{in}(cs_j))$, so $\models \mathit{in}(\beta_j) \Rightarrow Q_{ij}$.
5. $\models J_{\beta_j} \wedge I_{\eta_{ij}} \equiv (J_{\beta_j} \wedge N_i \wedge P_{ij} \wedge Q_{ij}) \vee (\neg \mathit{in}(\beta_j) \wedge \ldots)$
   PROOF: By 4 and the definition of $J_{\eta_{ij}}$.

$\boxed{6}$ β_j leaves $J_{\beta_j} \wedge I_{\eta_{ij}}$ invariant.

PROOF: By 5 and OP2, since $\beta 1$ implies that it leaves N_i invariant, $\beta 4$ implies that it leaves $J_{\beta_j} \wedge P_{ij} \wedge Q_{ij}$ invariant, and OP3 implies that it leaves $\neg in(\beta_j) \wedge \ldots$ invariant.

Proof for $\tau = \gamma_k$. As usual, we begin with the analogs of NC1–NC4. The statements and proofs of $\gamma 1$ and $\gamma 2$ are similar to the ones for NC1 and NC2 and are omitted.

$\gamma 3$. Operation γ_j leaves Q_{ij} invariant, for $i \neq j$.

PROOF: Follows from OP4, the Assignment Rule, OP1, and OP2, since $\models$ *after*$(\gamma_j) \Rightarrow$ *in*(δ_j) implies that $(in(\gamma_j) \vee after(\gamma_j)) \wedge Q_{ij}$ equals $(in(\gamma_j) \vee after(\gamma_j)) \wedge (i \ll j)$.

$\gamma 4$. Operation γ_j leaves P_{ij} invariant.

PROOF: By OP4, since $\models (in(\gamma_j) \vee after(\gamma_j)) \Rightarrow \neg in(\beta_j)$ implies that $(in(\gamma_j) \vee after(\gamma_j)) \wedge P_{ij}$ equals $in(\gamma_j) \vee after(\gamma_j)$.

The proof of the individual interference conditions for $\tau = \gamma_k$ are similar to the proofs for $\tau = \alpha_k$ and are omitted.

Proof for $\tau = \epsilon_{kl}$ ($k \neq l$). The proof begins, as usual, by stating and proving $\epsilon 1$–$\epsilon 4$, the analogs of NC1–NC4. Their proofs are essentially the same as the proofs of $\gamma 1$–$\gamma 4$. The interference-freedom conditions follow easily from $\epsilon 1$–$\epsilon 4$, WIN3, and OP2, using the Assignment and Await Rules and WIN5 to show that ϵ_{kl} commutes with α_i, β_i, ϵ_{ij}, and η_{ij}, for $i \neq k$.

Proofs for $\tau = \eta_{kl}$ ($k \neq l$) and $\tau = cs_k$. These proofs are similar to the proofs for ϵ_{kl} and ncs_k, respectively, and are omitted.

Proof for $\tau = \rho_k$. This proof is similar to, but simpler than, the proof for β_k. Like that proof, it requires two additional assumptions—namely, **we must assume that ρ_k leaves $winp(\beta_j, i \ll j)$ and $winp(\beta_i, num[i] > 0)$ invariant, for $i \neq k$ and $j \neq k$.**

4.2.2 Correcting the Proof

The proof above relied upon two extra assumptions:

- β_k and ρ_k leave $winp(\beta_i, num[i] > 0)$ invariant, for $k \neq i$.
- β_k and ρ_k leave $winp(\beta_j, i \ll j)$ invariant, for $k \neq i$ and $k \neq j$.

The first assumption is satisfied if β_i always sets $num[i]$ greater than 0, regardless of how the value of $num[k]$ changes while executing the operation. One can devise a “legal” implementation of β_i that does not satisfy this assumption, but such an implementation would be contrived. It seems quite reasonable to incorporate the assumption into the definition of statement β_i.

For the second assumption to be satisfied, modifying the value of $num[k]$ must leave $winp(\beta_j, i \ll j)$ invariant. However, there is no reason why it should. Here is a perfectly reasonable implementation of β_j, where the variables t_{jl} are local to process j:

$$\textbf{cobegin} \,\square_{l \neq j}\; \beta 1_{jl}\colon\; t_{jl} := num[l] \;\textbf{coend};$$
$$\beta 2_j\colon\; num[j] := 1 + \max\{t_{jl} : l \neq j\}$$

Consider a state s in which all the t_{jl} equal 0, all the $num[l]$ equal 0 except $num[k] = num[i] > 1$, and control is at $\beta 1_{jk}$ and after $\beta 1_{ji}$. (It is possible to reach such a state in a normal execution of the bakery algorithm.) Completing the execution of β_j starting in state s will set $num[j]$ to $1 + num[k]$, which equals $1 + num[i]$, making $i \ll j$ true. Therefore, $s \models winp(\beta_j, i \ll j)$ is true. However, if the state is changed by setting $num[k]$ to 0, completing the execution of β_j will set $num[j]$ to 1, making $i \ll j$ false. Hence executing ρ_k makes $winp(\beta_j, i \ll j)$ false, so this predicate is not left invariant by ρ_k. Moreover, since β_k could temporarily change $num[k]$ from a nonzero to a zero value, β_k need not leave $winp(\beta_j, i \ll j)$ invariant either.

We can fix the proof by replacing $winp(\beta_j, i \ll j)$ with a predicate R_{ij} having the following properties.

(i) R_{ij} is left invariant by β_j.

(ii) $\models (\mathit{after}(\beta_j) \wedge R_{ij}) \Rightarrow i \ll j$.

(iii) $\models (\mathit{at}(\beta_j) \wedge num[i] > 0) \;\Rightarrow\; R_{ij}$.

(iv) R_{ij} does not access any set of variables that contains neither $num[i]$, nor $num[j]$, nor any variable local to β_j.

We leave it to the reader to check that if R_{ij} satisfies these properties, then the invariance proof above works with R_{ij} substituted for $winp(\beta_j, i \ll j)$ in the definition of P_{ij}. (Perhaps the most difficult part of this proof is verifying $\beta 3$, which is done by using property (ii) to show that $\models J_{\beta_j} \wedge P_{ij} \wedge Q_{ij} \;\equiv\; (J_{\beta_j} \wedge R_{ij}) \vee (\mathit{after}(\beta_j) \wedge \ldots)$.) We also leave it to the reader to check that, for the implementation of β_j given above, we can define R_{ij} to be

$$\begin{gathered}[(\mathit{in}(\beta 1_{ji}) \vee \mathit{after}(\beta 1_{ji})) \Rightarrow winp(\beta 1_{ji}, t_{ji} = num[i] > 0)] \\ \wedge\; [(\mathit{in}(\beta 2_j) \vee \mathit{after}(\beta 2_j)) \Rightarrow winp(\beta 2_j, i \ll j)]\end{gathered}$$

thereby proving that the algorithm is correct with this implementation of β_j. (Property (i) is proved by applying the nonatomic Owicki-Gries method to the one-process program β_j.)

The following is an example of a valid implementation of β_j for which there is no such predicate R_{ij}, and for which the bakery algorithm is incorrect.

$\langle num[j] := 0 \rangle$;
$\langle m_j := j \rangle$;
for $k_j := 1$ **to** n **do** $\langle$**if** $num[k_j] > num[m_j]$ **then** $m_j := k_j \rangle$;
$\langle num[j] := 1 + num[m_j] \rangle$

There is nothing in Figure 3 to prohibit such an implementation of statement β_j; it would be a fine implementation if β_j appeared in a sequential program. We leave it to the reader to construct a scenario demonstrating that the bakery algorithm does not satisfy the mutual exclusion property with this implementation of β_j.

4.3 Another Example

Thus far, we have used *win* to reason about statements with an unspecified grain of atomicity. In our final example, we use *sin* to replace behavioral reasoning with assertional reasoning. The example may seem contrived, but it is abstracted from the part of the minimum spanning tree algorithm of Gallager et al. [6] that computes the minimum-weight external edge of a fragment. For this example, we just sketch the programs and proofs, omitting details.

Consider a tree of processes, each one communicating with its parent and its children by sending messages. Each node p has a value $val[p]$, and the goal of the algorithm is for the root process, denoted by r, to compute the minimum of all these values. The algorithm is obvious—every process finds the minimum of its value and that of its descendants, and reports that value to its parent. Each process p maintains three variables:

$Q[p]$: a queue of received messages.

$mini[p]$: the minimum of $val[p]$ and the values reported by p's children.

$cnt[p]$: the number of children of p who have not yet reported

For simplicity, assume that another process sends a message to process p by simply inserting the message in $Q[p]$. All queues are initially empty except for $Q[r]$, which contains a *find* message. The initial values of the other variables are unspecified. Each process p executes the following two actions.

find(p)**:** If there is a *find* message in $Q[p]$, then remove it from $Q[p]$ and set $mini[p]$ to $val[p]$. Set $cnt[p]$ to the number of children p has, and add a *find* message to every child's queue. If p has no children and $p \neq r$, then add a *report*$(val[p])$ message to the queue of p's parent.

receive(p)**:** If there is a $report(v)$ message in $Q[p]$, then remove it from $Q[p]$, set $mini[p]$ to the minimum of itself and v, and decrease $cnt[p]$ by one. If this makes $cnt[p]$ zero and $p \neq r$, then add a $report(mini[p])$ message to the queue of p's parent.

The algorithm terminates when $cnt[r] = 0$ and $Q[r]$ is empty, at which time $mini[r]$ is the result. We wish to prove the partial correctness property $\models P \Rightarrow \Box Q$ for this algorithm, where P asserts the initial condition on the queues and Q asserts that if the termination condition holds then $mini[r]$ has the correct value.

Define a process to be *active* if there is a *report* message in its queue or any message in the queue of any descendant, and to be *finished* if it is not active and there is no *find* message in its queue or in the queue of any ancestor. Let I be the predicate asserting that for every process p:

1. If there is a *find* message in $Q[p]$, then (i) it is the only message in $Q[p]$ and (ii) the queue of every descendant of p is empty.
2. If p is active, then (i) $cnt[p]$ equals the number of unfinished descendants of p plus the number of *report* messages in $Q[p]$, and (ii) the minimum of $mini[p]$ and all v for which there is a $report(v)$ message in $Q[p]$ equals the minimum of all $val[p']$ with p' equal to p or a finished descendant of p.
3. If p is finished, then $mini[p]$ is the minimum of all $val[p']$ for p' equal to p or a descendant of p.

The reader can check that $\models P \Rightarrow I$, $\models I \Rightarrow Q$, and I is left invariant by every program action, proving that $\models P \Rightarrow \Box Q$.

Thus far, our example has been a simple exercise in assertional reasoning. We now complicate matters by allowing the tree of processes to grow dynamically. We assume a larger collection of processes, only some of which are initially in the process tree, and add a new action $addchild(p, q)$ that makes process q a child of process p. This action may be executed only when the following conditions hold: q is not the parent or child of any process, $Q[q]$ is empty, and $val[q]$ is greater than the minimum of all $val[p']$ for processes p' currently in the tree.

The following simple operational argument shows that the modified algorithm, with the *addchild* actions, satisfies the same correctness property $P \Rightarrow \Box Q$. If an $addchild(p, q)$ action is executed before the $find(p)$ action, then the effect is the same as if q were part of the original process tree. On the other hand, if the action is executed after the $find(p)$ action, then the effect is the same as if q were added to the process tree after the algorithm had terminated. Hence, we may pretend that each *addchild* action occurs either before or after the algorithm is executed. It is clear that executing an $addchild(p, q)$ action at the beginning does not change P, and, since the action is executed only if $val[q]$ is greater than the minimum value among existing tree processes, executing it at the end does not change Q. Hence, the modified algorithm satisfies $P \Rightarrow \Box Q$.

Although the modified algorithm satisfies the same partial correctness property as the original algorithm, a different proof is required because the modified algorithm does not leave I invariant. For example, an $addchild(p, q)$ action can make condition 3 of I false. One can find a new invariant for the modified algorithm, but it would be nice to reason directly from the correctness of the original algorithm, as in the behavioral argument.

Let us write each $addchild(p, q)$ action as the union of the two actions $preadd(p, q)$ and $postadd(p, q)$, where a pair (s, t) is in $preadd(p, q)$ if process p is neither active nor finished in state s, otherwise it is in $postadd(p, q)$. (Formally, we modify the set Π of actions but leave the union of all actions unchanged, so we obtain an equivalent program.) The reader can check that every *preadd* action leaves I invariant; it is the *postadd* actions that may falsify I.

Let σ denote the set of all *postadd* actions. We show that $sin(\sigma, I)$ is the invariant that proves the correctness of the modified algorithm. To do this, we must prove $\models P \Rightarrow sin(\sigma, I)$, $\models sin(\sigma, I) \Rightarrow Q$, and the invariance of $sin(\sigma, I)$.

As we observed above, every *addchild* action leaves P and Q invariant, so every action of σ does also. Hence, $\models sin(\sigma, P) \equiv P$ and $\models sin(\sigma, Q) \equiv Q$. By SIN2, $\models P \Rightarrow sin(\sigma, I)$ and $\models sin(\sigma, I) \Rightarrow Q$ then follow from $\models P \Rightarrow I$ and $\models I \Rightarrow Q$.

Finally, we show that $sin(\sigma, I)$ is an invariant. It is obviously left invariant by any action in σ, so we must show that it is left invariant by every other action. By SIN3, it suffices to show that every action not in σ left commutes with a $postadd(p, q)$ action. It is clear that the action $postadd(p, q)$ commutes with every action not in σ except for the following: $preadd(p', p)$, $preadd(q, q')$, $find(p)$, and $find(q)$. We prove left commutativity by showing that, if ξ is any one of these four actions, then $postadd(p, q)\,\xi$ is the empty action. (Recall that ξ left commutes with μ iff $\mu\xi \subseteq \xi\mu$.)

preadd(p', q)**:** The composition $postadd(p, q)\,preadd(p', p)$ is empty because an $addchild(p, q)$ action can be executed only if p is already in the process tree, in which case the $preadd(p', p)$ action cannot be executed.

preadd(q, q')**:** $postadd(p, q)\,preadd(q, q')$ is empty because the composition $addchild(p, q)\,addchild(q, q')$ can be nonempty only if the two *addchild* actions are either both *postadd* or both *preadd* actions.

find(p)**:** $postadd(p, q)\,find(p)$ is empty because an $addchild(p, q)$ action cannot be a *postadd* action if $Q[p]$ contains a *find* message.

find(q)**:** $postadd(p, q)\,find(q)$ is empty because an $addchild(p, q)$ action can occur only if $Q[q]$ is empty.

This completes the proof of invariance of $sin(\sigma, I)$.

5 Discussion

Although we have provided rigorous, step-by-step proofs in our first two examples, we have not tried to be completely formal. We did not give the rules for reasoning about control predicates needed to prove such obvious relations as $\models (in(\epsilon_{jl}) \lor after(\epsilon_{jl})) \Rightarrow \neg in(\beta_j)$ for the bakery algorithm. We believe that if a formalism is to be useful, it must be possible to use it rigorously but informally, without having to prove obvious properties. Experience with the atomic Owicki-Gries method indicates that it can be used in this way, and we believe that the same is true of the nonatomic version employing *win* and *sin*.

In judging the utility of *win* and *sin*, it is instructive to consider why previous correctness proofs of the bakery algorithm did not discover its hidden assumptions. The original proof in [9] is an informal behavioral one, so it is not surprising that it is incorrect. The proof in [11] utilizes a set of axioms for reasoning about behaviors involving nonatomic operations. While the use of axioms gives an appearance of extreme rigor, the method ultimately reduces to the unstructured, informal reasoning of ordinary mathematics. The undetected assumptions in the bakery algorithm provide one more example of the unreliability of such reasoning.

A rigorous Owicki-Gries method proof is given in [10]. However, since the original Owicki-Gries method requires that all atomic operations be specified, it was necessary to translate the bakery algorithm into one with explicit atomic operations. The translation effectively specified a particular class of implementations of the algorithm—a class that includes only implementations satisfying the hidden assumptions. This proof illustrates the danger in trying to replace one program with an equivalent one, if the equivalence has not been proved formally. Without a formal justification of the single-action rule, even *its* use should be regarded with suspicion.

The bakery algorithm's two hidden assumptions are that β_i sets $num[i]$ to be (i) positive and (ii) greater than $num[j]$, even if it is executed while the value of $num[k]$ is being changed, for $k \neq i, j$. Although the algorithm has been rather widely studied, we know of only one other person who independently discovered assumption (ii). We discovered assumption (i) only when expanding an earlier version of our *win* proof to its present, more rigorous, form. We knew about assumption (ii) before writing this article, but we are confident that attempting the proof would have led to its discovery anyway.

Assertional methods, including the Owicki-Gries method, reduce a proof of correctness to a collection of small steps—each of which involves reasoning about a single operation. Previous assertional methods require that each operation be atomic. The *win* and *sin* operators permit the generalization of these methods to allow nonatomic operations. However, much work remains in assessing the practical utility of these operators and developing their formal theory. We believe that our rules for reasoning about *win* provide a relatively complete method for proving $P \Rightarrow \Box Q$ formulas for simple **cobegin** programs, where the semantics of nonatomic operations are defined by the Assignment and Await Rules, but a detailed proof of this result has not yet been written. Moreover, we expect a formal system for reasoning about nonatomic operations to be much more sensitive to the semantics of the particular language constructs than one for reasoning about atomic operations, so no far-reaching conclusions can be drawn from a single completeness result. In particular, nonatomic communication primitives have yet to be studied.

Acknowledgments

I wish to thank Fred Schneider, Frank Anger, Jim Saxe, and Martín Abadi for their comments on earlier versions of this report.

References

[1] Krzysztof R. Apt. Ten years of Hoare's logic: A survey—part one. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[2] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[3] J. W. de Bakker and W. P. de Roever. A calculus for recursive program schemes. In *Automata, Langages, and Programming*, pages 167–196, Amsterdam, July 1972. North-Holland.

[4] J. W. de Bakker and L. G. L. T. Meertens. On the completeness of the inductive assertion method. *Journal of Computer and System Sciences*, 11(3):323–357, 1975.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[6] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[7] David Harel. On folk theorems. *Communications of the ACM*, 23(7):379–389, July 1980.

[8] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[9] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[10] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[11] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[12] Leslie Lamport. Reasoning about nonatomic operations. In *Proceedings of the Tenth Annual Symposium on Principles of Programming Languages*, pages 28–37. ACM SIGACT-SIGPLAN, January 1983.

[13] Leslie Lamport and Fred B. Schneider. The "Hoare logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984.

[14] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

Index