Putting Time Into Proof Outlines

Fred B. Schneider*
Bard Bloom**
Keith Marzullo***

TR 91-1238
September 1991

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

*Supported in part by the Office of Naval Research under contract N00014-91-J-1219 the National Science Foundation under grant No. CCR-8701103, DARPA/NSF Grant No. CCR-9014363 and grant from IBM Endicott Programming Laboratory.
**Supported in part by NSF Grant CCR-9003441.
***Supported in part by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593 and by grants from IBM and Siemen

Putting Time into Proof Outlines

Fred B. Schneider*
Bard Bloom†
Keith Marzullo‡

Department of Computer Science
Cornell University
Ithaca, New York 14853

Abstract. A logic for reasoning about timing properties of concurrent programs is presented. The logic is based on proof outlines and can handle maximal parallelism as well as resource-constrained execution environments. The correctness proof for a mutual exclusion protocol that uses execution timings in a subtle way illustrates the logic in action.

Key words: concurrent program verification, timing properties, safety properties, real-time programming, real-time actions, proof outlines.

Contents

---

*Supported in part by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under Grant No. CCR-8701103, DARPA/NSF Grant No. CCR-9014363, and a grant from IBM Endicott Programming Laboratory.

†Supported in part by NSF grant CCR-9003441.

‡Supported in part by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, and by grants from IBM and Siemens.

1. Introduction

A safety property of a program asserts that some proscribed "bad thing" does not occur during execution. To prove that a program satisfies a safety property, one typically employs an *invariant*, a characterization of current (and possibly past) program states that is not invalidated by execution. If an invariant I holds in the initial state of the program and $I \Rightarrow Q$ is valid for some Q, then $\neg Q$ cannot occur during execution. Thus, to establish that a program satisfies the safety property asserting that $\neg Q$ does not occur, it suffices to find such an invariant I.

Timing properties are safety properties where the "bad thing" involves the time and program state at the instants that various specified control points in a program become active.[1] Timing properties can restrict externally visible events, like inputs and outputs, as well as things that are internal to a program, like the value of a variable or the time that a particular statement starts or finishes. For example, in a process control system, the elapsed time between a stimulus and response must be bounded. This is a timing property where the "bad thing" is defined in terms of the time that passes after one control point becomes active until some other control point does. Timing properties concerning internal events are useful in reasoning about ordinary concurrent programs that exploit knowledge of statement execution times to coordinate processes. One such protocol—for mutual exclusion—is given in section 4.

Because timing properties are safety properties, the invariant-based method outlined above for reasoning about safety properties can be used to reason about timing properties. This means that a programming logic L to verify (ordinary) safety properties can form the basis for a logic L' to verify timing properties. It suffices that in L' we are able to

(1) specify in I and Q information about the times at which events of interest occur and

(2) establish that program execution does not invalidate such an I.

Point (1) means that in defining L', the language of L might have to be extended so that it becomes more expressive. Point (2) means that the inferencing apparatus of L might have to be refined so that I can be proved an invariant for a program whose semantics includes information about execution timings.

This paper describes extensions to a logic of proof outlines [Schneider 92] to enable verification of timing properties for concurrent programs. The approach taken is the one just outlined: we start with a logic for proving ordinary safety properties, augment the language according to (1) and refine the inference rules according to (2). The presentation is organized as follows. In section 2, we describe a logic of proof outlines. Section 3 introduces and axiomatizes a new type of atomic action, called a real-time action. The correctness proof for a mutual exclusion protocol in section 4 illustrates the use of our logic. Related work and some unresolved technical issues are discussed in section 5.

[1]Informally, the *active* control points at any instant are determined by the values of the program counters at that instant. See §2 for a more formal definition.

2. Proof Outlines

In order to reason about a program, we must be able to define sets of program states and reason about them. First-order predicate logic is an obvious choice for this task, and we employ the usual correspondence between the formulas of the logic and the programming language of interest—each variable and expression of the programming language is made a term of the logic and each Boolean expression of the programming language is made a predicate of the logic. It will be convenient to assume that predicates and terms are always defined, although the value of a term may be unspecified in some states. For example, we will assume that the term x/y has a value whatever value y has, but that $y*(x/y)$ need not equal x when y is 0 because the value of x/y is unspecified in such states.

Predicates and function symbols for the programming language's data types provide a way to express facts about program variables and expressions. The state of a program, however, also includes information that tells what atomic actions might be executed next. For representing this control information, we will find it convenient to fix some predicate symbols, called *control predicates*, and give axioms to ensure that, as execution proceeds, changes in the values of these correspond to changes to program counters. (An alternative representation would have been to define a "program counter" variable and a data type for the values it can assume.)

2.1. Control Predicates

A program consists of a set of atomic actions, each of which executes as a single indivisible state transformation. The *control points* of the program are defined by these atomic actions. Each atomic action has distinct *entry control points* and *exit control points*. For example, the atomic action that implements **skip** has a single entry control point and a single exit control point; the test for an **if** has one entry control point and one exit control point for each alternative. Execution of an atomic action α can occur only when an entry control point for α is *active*. Among other things, execution causes that active entry control point to become inactive and an exit control point of α to become active.

For each statement or atomic action S, we define the following control predicates:

at(S): an entry control point for S is active.

after(S): an exit control point from S is active.

The various statements in a programming language give rise to axioms relating these control predicates. The axioms formalize how the control predicates for a statement or atomic action S relate to the control predicates for constructs comprising S and constructs containing S, based on the control flow defined by S. For a guarded-command programming language [Dijkstra 75], these axioms are given in Figure 2.1. We use $GEval_{if}(S)$ there to denote the guard evaluation action for an **if** and $GEval_{do}(S)$ to denote the guard evaluation action for a **do**. And, we write $P_1 \oplus P_2 \oplus \ldots \oplus P_n$ to denote that exactly one of P_1 through P_n holds.

Atomic action: For S a **skip**, guard evaluation action, or assignment:

$$\neg(at(S) \wedge after(S))$$

Sequential composition: For S the sequential composition $S_1\ S_2$:

(a) $at(S) = at(S_1)$

(b) $after(S) = after(S_2)$

(c) $after(S_1) = at(S_2)$

if *Control Axioms*: For an **if** statement:

$$S\text{: } \mathbf{if}\ B_1 \rightarrow S_1\ [\,]\ B_2 \rightarrow S_2\ [\,]\ \cdots\ [\,]\ B_n \rightarrow S_n\ \mathbf{fi}$$

(a) $at(S) = at(GEval_{if}(S))$

(b) $after(S) = (after(S_1) \oplus after(S_2) \oplus ... \oplus after(S_n))$

(c) $after(GEval_{if}(S)) = (at(S_1) \oplus at(S_2) \oplus ... \oplus at(S_n))$

do *Control Axioms*: For a **do** statement:

$$S\text{: } \mathbf{do}\ B_1 \rightarrow S_1\ [\,]\ B_2 \rightarrow S_2\ [\,]\ \cdots\ [\,]\ B_n \rightarrow S_n\ \mathbf{od}$$

(a) $at(GEval_{do}(S)) = (at(S) \oplus after(S_1) \oplus after(S_2) \oplus ... \oplus after(S_n))$

(b) $after(GEval_{do}(S)) = (at(S_1) \oplus at(S_2) \oplus ... \oplus at(S_n) \oplus after(S))$

cobegin *Control Axioms*: For a **cobegin** statement:

$$S\text{: } \mathbf{cobegin}\ S_1\ //\ S_2\ //\ \cdots\ //\ S_n\ \mathbf{coend}$$

(a) $at(S) = (at(S_1) \wedge ... \wedge at(S_n))$

(b) $after(S) = (after(S_1) \wedge ... \wedge after(S_n))$

Figure 2.1. Control Predicate Axioms

2.2. Syntax and Meaning of Proof Outlines

A *proof outline* $PO(S)$ for a program S is a text in which every atomic action of S is preceded and followed by an *assertion* enclosed in braces ("{" and "}"). Each assertion is a Predicate Logic formula in which

- the free variables are *program variables* (typeset in italics) or *rigid variables*, (typeset in upper-case roman), and
- the predicate symbols are control predicates or the predicates of the programming language's expressions.

Assertions in which all terms are constructed from program variables, rigid variables, and predicates involving those variables are called *primitive assertions*. An example of a proof outline appears in Figure 2.2. In it, x is a program variable and X is a rigid variable. All assertions except the first and last are primitive.

$$
\begin{array}{l}
\{x{=}\mathrm{X} \wedge at(S)\} \\
S\text{: } \mathbf{if}\ x{\geq}0 \rightarrow \{x{=}\mathrm{X} \wedge x{\geq}0\} \\
\qquad\qquad\quad S_1\text{: } \mathbf{skip} \\
\qquad\qquad\quad \{x{=}\mathrm{X} \wedge x{\geq}0\} \\
\quad [\,]\ x{\leq}0 \rightarrow \{x{=}\mathrm{X} \wedge x{\leq}0\} \\
\qquad\qquad\quad S_2\text{: } x := -x \\
\qquad\qquad\quad \{-x{=}\mathrm{X} \wedge -x{\leq}0\} \\
\quad \mathbf{fi} \\
\{x{=}abs(\mathrm{X}) \wedge after(S)\}
\end{array}
$$

Figure 2.2. Computing $abs(x)$

The assertion that immediately precedes a statement or atomic action T in a proof outline $PO(S)$ is called the *precondition* of T and is denoted $pre(T)$; the assertion that directly follows T is called the *postcondition* of T and is denoted by $post(T)$. For the proof outline in Figure 2.2, this correspondence is summarized in Figure 2.3. Finally, for a proof outline $PO(S)$, we write $pre(PO(S))$ to denote $pre(S)$, $post(PO(S))$ to denote $post(S)$, and use a *triple*

(2.1) $\{P\}\ PO(S)\ \{Q\}$

to specify the proof outline in which $pre(S)$ is P, $post(S)$ is Q, and all other pre- and postconditions are the same as in $PO(S)$.

A proof outline $PO(S)$ can be regarded as associating an assertion $pre(T)$ with control predicate $at(T)$ and an assertion $post(T)$ with $after(T)$ for each statement T in a program fragment S. Consequently, a proof outline defines a mapping from each control point λ of a program to a set of assertions—those assertions associated with control predicates that are *true* whenever λ is active. In most cases, a control point is mapped to a single assertion. For example, the proof outline

(2.2) $\{P\}\ S_1\ \{Q\}\ S_2\ \{R\}$

maps the entry control point for program $S_1\,S_2$ to the single assertion P. This is because $at(S_1)$ and $at(S_1\,S_2)$ are the only control predicates that are *true* if and only if the entry control point for $S_1\,S_2$ is

Assertion	Assertion Text
$pre(S)$	$x{=}\mathrm{X} \wedge at(S)$
$post(S)$	$x{=}abs(\mathrm{X}) \wedge after(S)$
$pre(S_1)$	$x{=}\mathrm{X} \wedge x{\geq}0$
$post(S_1)$	$x{=}\mathrm{X} \wedge x{\geq}0$
$pre(S_2)$	$x{=}\mathrm{X} \wedge x{\leq}0$
$post(S_2)$	$-x{=}\mathrm{X} \wedge -x{\leq}0$

Figure 2.3. Assertions in a Proof Outline

active, and (2.2) associates P with both of these control predicates. However, a proof outline can map a given control point to multiple assertions. An example of this appears in Figure 2.2. There, the exit control point for S_1 is mapped to two assertions—$post(S_1)$ and $post(S)$—because whenever the exit control point of S_1 is active both $after(S_1)$ and $after(S)$ are *true*.

The assertions in a proof outline are intended to document what can be expected to hold of the program state as execution proceeds. The proof outline of Figure 2.2, for example, implies that if execution is started at the beginning of S_1 with $x=23$ (a state that satisfies $pre(S_1)$), then if S_1 completes, $post(S_1)$ will be satisfied by the resulting program state, as will $post(S)$. And if execution is started at the beginning of S with $x=\mathrm{X}$, then whatever assertion is next reached—be it $pre(S_1)$ because $\mathrm{X}\geq 0$ or $pre(S_2)$ because $\mathrm{X}\leq 0$—that assertion will hold when reached, and the next assertion will hold when it is reached, and so on.

With this in mind, we define a proof outline $PO(S)$ to be *valid* if it describes a relationship among the program variables and control predicates of S that is invariant and, therefore, not falsified by execution of S. The invariant defined by a proof outline $PO(S)$ is "if a control point λ is active, then all assertions that λ is mapped to by $PO(S)$ are satisfied" and is formalized as the *proof outline invariant* for $PO(S)$:

(2.3) $\quad I_{PO(S)}\colon \bigwedge_{T} ((at(T) \Rightarrow pre(T)) \wedge (after(T) \Rightarrow post(T)))$

For example, the proof outline invariant defined by $PO(S)$ of Figure 2.2 is

$$\begin{array}{lll} & at(S) \Rightarrow (x=\mathrm{X} \wedge at(S)) & \wedge\ after(S) \Rightarrow (x=abs(\mathrm{X}) \wedge after(S)) \\ \wedge & at(S_1) \Rightarrow (x=\mathrm{X} \wedge x\geq 0) & \wedge\ after(S_1) \Rightarrow (x=\mathrm{X} \wedge x\geq 0) \\ \wedge & at(S_2) \Rightarrow (x=\mathrm{X} \wedge x\leq 0) & \wedge\ after(S_2) \Rightarrow (-x=\mathrm{X} \wedge -x\leq 0). \end{array}$$

Equating proof outline validity with invariance of $I_{PO(S)}$ can have disturbing consequences for proof outlines that map a single control point to multiple assertions. The following valid proof outline illustrates this.

(2.4) $\quad \{false\}$
$\quad S\colon$ **if** $true \rightarrow \{false\}\ S'\colon\ x := 3\ \{x=1\}$ **fi**
$\quad \{x=2\}$

This proof outline maps the exit control point for S' to two assertions, $post(S')$ and $post(S)$. The proof outline is valid because $I_{PO(S)}$

$$\begin{array}{lll} & at(S) \Rightarrow false & \wedge\ after(S) \Rightarrow x=2 \\ \wedge & at(S') \Rightarrow false & \wedge\ after(S') \Rightarrow x=1 \end{array}$$

is equivalent to *false* (since $after(S')=after(S)$ is valid) and therefore $I_{PO(S)}$ cannot be falsified by execution of any statement. The problem with (2.4) is that $post(S)$, the assertion associated with the exit control point of S, is not implied by $post(S')$, the assertion associated with the exit control point for the last atomic action in S (i.e S'). As a result, what (2.4) really associates with the exit control point for S' (viz. $post(S') \wedge post(S)$) is not accurately characterized by $post(S)$. Given a valid proof outline $PO(S)$, it seems reasonable to expect $post(S)$ to hold whenever an exit control point of S is active. Similarly, $pre(S)$ should be constrained so that if it holds and an entry control point of S is active, then assertions that $PO(S)$ associates with that entry control point also hold. To formalize these constraints, we define a proof outline $PO(S)$ to be *self consistent* if and only if

(2.5) $\quad at(S) \wedge pre(S) \Rightarrow II_{PO(S)}$

(2.6) $\quad after(S) \wedge II_{PO(S)} \Rightarrow post(S)$

where

$$II_{PO(S)}: \bigwedge_{T \neq S}((at(T) \Rightarrow pre(T)) \wedge (after(T) \Rightarrow post(T)))$$

$II_{PO(S)}$ is just $I_{PO(S)}$ with the two conjuncts concerning $pre(S)$ and $post(S)$ (i.e. "$at(S) \Rightarrow pre(S)$" and "$after(S) \Rightarrow post(S)$") omitted.[2] Thus, (2.5) ensures that whenever any entry control point λ for S is active, if $pre(S)$ holds then so does the assertion that $PO(S)$ associates with λ. And (2.6) ensures that whenever any exit control point λ of S is active, if the assertion associated with that control point holds then $post(S)$ will hold as well. Together, (2.5) and (2.6) mean that $pre(S)$ and $post(S)$ constitute a reasonably complete interface to S: provided $pre(S)$ holds when execution of S is started, the assertions of $PO(S)$ will characterize any states that arise as execution proceeds and $post(S)$ will hold if an exit control point for S is ever reached. It should come as no surprise that the proof outline of (2.4) is not self consistent—(2.6) is violated.

The requirements for validity of a proof outline—invariance of $I_{PO(S)}$ and self-consistency—can be formalized in terms of $\mathcal{H}_S^+$-validity of Temporal Logic formulas, where $\mathcal{H}_S^+$ is the set of infinite state sequences that model execution of S started from any program state [Owicki-Lamport 82]. In this formalization, we are able to write $\mathcal{H}_S^+ \models P$ in order to denote that a Predicate Logic formula P is valid because every program state is the first state of some interpretation in $\mathcal{H}_S^+$.

(2.7) **Valid Proof Outline.** A proof outline $PO(S)$ is *valid* if and only if:

Self Consistency: $\mathcal{H}_S^+ \models (at(S) \wedge pre(S) \Rightarrow II_{PO(S)})$

$\mathcal{H}_S^+ \models (after(S) \wedge II_{PO(S)} \Rightarrow post(S))$

Invariance: $\mathcal{H}_S^+ \models (I_{PO(S)} \Rightarrow \Box I_{PO(S)})$ $\Box$

Notice that according to Valid Proof Outline (2.7), rigid variables in proof outlines can be used relate the values of program variables from one state to the next. This is because free rigid variables in a temporal logic formula are implicitly universally quantified. Thus, $I_{PO(S)} \Rightarrow \Box I_{PO(S)}$ is $\mathcal{H}_S^+$-valid if and only if for any assignment of values to the proof outline's rigid variables, execution of S

(i) starts in a state that does not satisfy $I_{PO(S)}$ or

(ii) results in a sequence of states that each satisfy $I_{PO(S)}$.

For example, the proof outline of Figure 2.2 is valid and contains a rigid variable X to record the initial value of x. Starting execution in a state where $at(S_2)$ and $x=-23$ holds will satisfy $I_{PO(S)} \Rightarrow \Box I_{PO(S)}$ even if -23 is not assigned to X because then $I_{PO(S)}$ is not satisfied (causing $I_{PO(S)} \Rightarrow \Box I_{PO(S)}$ to be trivially satisfied).

[2] *II* is an acronym for ***internal invariant.***

2.3. Axiomatization for a Proof Outline Logic

Proof Outline Logic is an extension of Predicate Logic. The language of Predicate Logic is extended with proof outlines for all atomic actions, statements, and programs. The axioms and inference rules of Predicate Logic are extended with axioms and inference rules that allow only valid proof outlines to be proved theorems. In particular, there are some statement-independent inference rules as well as an axiom or inference rule for each type of statement and atomic action.

The statement-independent inference rules for Proof Outline Logic are given in Figure 2.4. Rule of Consequence allows the precondition of a proof outline to be strengthened and the postcondition to be weakened, based on deductions possible in Predicate Logic. Rule of Equivalence allows assertions anywhere in a proof outline to be modified, provided a self consistent proof outline having an equivalent proof outline invariant results. A rigid variable can be renamed or instantiated by using the Rigid Variable Rule; $PO(S)^{\mathrm{X}}_{\mathrm{Exp}}$ in the conclusion of that rule denotes a proof outline in which rigid variable X in every assertion is replaced by Exp, an expression involving constants and rigid variables (only). Finally, the Conjunction and Disjunction Rules allow two proof outlines for the same program to be combined. $PO_A(S) \otimes PO_B(S)$ is used to denote the proof outline that associates assertion $A_{cp} \wedge B_{cp}$ with each control predicate cp, where X_{cp} is the assertion that $PO_X(S)$ associates with control predicate cp; $PO_A(S) \oslash PO_B(S)$ denotes the proof outline that associates $A_{cp} \vee B_{cp}$ with each control predicate cp

Rule of Consequence: $\dfrac{P' \Rightarrow P,\ \ \{P\}\ PO(S)\ \{Q\},\ \ Q \Rightarrow Q'}{\{P'\}\ PO(S)\ \{Q'\}}$

Rule of Equivalence: $\dfrac{PO(S),\ \ I_{PO(S)} = I_{PO'(S)},\ \ PO'(S) \text{ self consistent}}{PO'(S)}$

Rigid Variable Rule: For Exp an expression involving only constants and rigid varibles:

$$\frac{\{P\}\ PO(S)\ \{Q\}}{\{P^{\mathrm{X}}_{\mathrm{Exp}}\}\ PO(S)^{\mathrm{X}}_{\mathrm{Exp}}\ \{Q^{\mathrm{X}}_{\mathrm{Exp}}\}}$$

Conjunction Rule: $\dfrac{PO_A(S),\ \ PO_B(S)}{PO_A(S) \otimes PO_B(S)}$

Disjunction Rule: $\dfrac{PO_A(S),\ \ PO_B(S)}{PO_A(S) \oslash PO_B(S)}$

Figure 2.4. Proof Outline Logic: Statement-independent Rules

We now turn to the axiomatization for a concurrent programming language. The **skip** statement is a single atomic action whose execution has no effect on any program variable.

skip *Axiom*: For a primitive assertion P: $\{P\}$ **skip** $\{P\}$

The assignment statement $x := E$ is also a single atomic action. Its execution involves evaluating E and then storing that value in x.[3]

Assignment Axiom: For a primitive assertion P: $\{P^{\bar{x}}_{\bar{e}}\}\ \bar{x} := \bar{e}\ \{P\}$

Sequential composition of statements is denoted by juxtaposition (without the traditional semicolon separator).

Statement Composition Rule: $\dfrac{\{P\}\, PO(S_1)\, \{Q\},\ \{Q\}\, PO(S_2)\, \{R\}}{\{P\}\, PO(S_1)\, \{Q\}\, PO(S_2)\, \{R\}}$

An **if** statement consists of an atomic guard evaluation action that selects for execution an alternative whose guard is *true*; if no guard is *true*, then the guard evaluation action blocks. We use the following rule for reasoning about a guard evaluation action.

$GEval_{if}(S)$ Axiom: For an **if** statement
S: **if** $B_1 \rightarrow S_1\ [\!]\ B_2 \rightarrow S_2\ [\!]\ \cdots\ [\!]\ B_n \rightarrow S_n$ **fi**
and a primitive assertion P:

$$\{P\}\ GEval_{if}(S)\ \{P \wedge ((at(S_1) \Rightarrow B_1) \wedge \ldots \wedge (at(S_n) \Rightarrow B_n))\}$$

A proof outline for an **if** is then constructed by combining a proof outline for its guard evaluation action with a proof outline for each alternative.

if *Rule*: (a) $\{P\}\ GEval_{if}(S)\ \{R\}$,
(b) $(R \wedge at(S_1)) \Rightarrow P_1,\ \ldots,\ (R \wedge at(S_n)) \Rightarrow P_n$,
(c) $\{P_1\}\ PO(S_1)\ \{Q\},\ \ldots,\ \{P_n\}\ PO(S_n)\ \{Q\}$,

---

$\{P\}$
S: **if** $B_1 \rightarrow \{P_1\}\ PO(S_1)\ \{Q\}$
$[\!]\ \cdots$
$[\!]\ B_n \rightarrow \{P_n\}\ PO(S_n)\ \{Q\}$
fi
$\{Q\}$

Since the guard evaluation action for an **if** blocks when no guard is *true*, we can use an **if** to implement conditional waiting. For example,

---

[3]For simplicity, we restrict consideration to the case where x is a simple identifier and not an array. See [Gries-Levin 80] for the more general rule.

if $B \rightarrow S$ **fi**

blocks until the program state satisfies B and only then starts executing S.

The guard evaluation action for **do** selects a statement S_i for which corresponding guard B_i holds and if no guard is *true*, then the control point following the **do** becomes active.

GEval$_{do}(S)$ *Axiom*: For a **do** statement

$$S\colon \textbf{do } B_1 \rightarrow S_1 \;\; [] \;\; B_2 \rightarrow S_2 \;\; [] \;\; \cdots \;\; [] \;\; B_n \rightarrow S_n \textbf{ od}$$

and a primitive assertion P:

$$\{P\} \;\; GEval_{do}(S) \;\; \{P \wedge ((at(S_1) \Rightarrow B_1) \wedge \;...\; \wedge (at(S_n) \Rightarrow B_n) \\ \wedge (after(S) \Rightarrow (\neg B_1 \wedge \;...\; \wedge \neg B_n)))\}$$

The inference rule for **do** is based on a *loop invariant*, an assertion I that holds before and after every iteration of a loop and, therefore, is guaranteed to hold when **do** terminates—no matter how many iterations occur.

do *Rule*:

$$\frac{\begin{array}{l}\text{(a) } \{I\}\; GEval_{do}(S)\; \{R\}, \\ \text{(b) } (R \wedge at(S_1)) \Rightarrow P_1, \;\; ..., \;\; (R \wedge at(S_n)) \Rightarrow P_n, \\ \text{(c) } \{P_1\}\; PO(S_1)\; \{I\}, \;\; ..., \;\; \{P_n\}\; PO(S_n)\; \{I\} \\ \text{(d) } (R \wedge after(S)) \Rightarrow (I \wedge \neg B_1 \wedge \;...\; \wedge \neg B_n)\end{array}}{\begin{array}{l}\{I\} \\ S\colon \textbf{do } B_1 \rightarrow \{P_1\}\; PO(S_1)\; \{I\} \\ \quad\quad [] \;\cdots \\ \quad\quad [] \; B_n \rightarrow \{P_n\}\; PO(S_n)\; \{I\} \\ \quad\quad \textbf{od} \\ \{I \wedge \neg B_1 \wedge \;...\; \wedge \neg B_n\}\end{array}}$$

The inference rule for a **cobegin** is based on combining proof outlines for its component processes. An interference-freedom test [Owicki-Gries 76] ensures that execution of an atomic action in one process does not invalidate the proof outline invariant for another. This interference-freedom test is formulated in terms of triples,

$$NI(\alpha, A)\colon \;\; \{pre(\alpha) \wedge A\} \;\; \alpha \;\; \{A\},$$

that are valid if and only if α does not invalidate assertion A. If no assertion in $PO(S_i)$ is invalidated by an atomic action α then, by definition, $I_{PO(S_i)}$ also cannot be invalidated by α. Therefore, we can prove that a collection of proof outlines $PO(S_1), ..., PO(S_n)$ are *interference free* by establishing:

For all $i, j, 1 \le i \le n, 1 \le j \le n, i \ne j$:
  For all atomic actions α in S_i :
    For all assertions A in $PO(S_j)$: $NI(\alpha, A)$ is valid.

The following inference rule determines when a valid proof outline for a **cobegin** will result from combining valid proof outlines for its component processes:

cobegin *Rule*:

$$\frac{\begin{array}{l}\text{(a) } PO(S_1),\ \ldots,\ PO(S_n)\\ \text{(b) } P \Rightarrow pre(PO(S_1)) \wedge \ldots \wedge pre(PO(S_n)),\\ \text{(c) } post(PO(S_1)) \wedge \ldots \wedge post(PO(S_n)) \Rightarrow Q,\\ \text{(d) } PO(S_1),\ \ldots,\ PO(S_n) \text{ are interference free.}\end{array}}{\{P\}\ \textbf{cobegin}\ PO(S_1)\ //\ \cdots\ //\ PO(S_n)\ \textbf{coend}\ \{Q\}}$$

Since execution of an atomic action α in one process never interferes with a control predicate cp in another, certain interference-freedom triples follow axiomatically.

> *Process Independence Axiom*: For a control predicate cp in one process and an atomic action α in another:
>
> $$\{cp{=}\mathrm{C}\}\ \alpha\ \{cp{=}\mathrm{C}\}$$

Notice that $NI(\alpha, cp)$ follows directly from this axiom when α and cp are from different processes.

2.4. From Proof Outlines to Safety Properties

Theorems of Proof Outline Logic can be used to verify safety properties because of the way proof outline validity is defined. If a proof outline $PO(S)$ is valid then $I_{PO(S)}$ must be an invariant. And, if $I_{PO(S)}$ is an invariant, then according to the method of §1 for proving safety properties we can prove that executions of S starting with $pre(PO(S))$ *true* will satisfy the safety property proscribing $\neg Q$. We simply prove

(2.8) $\quad (cp \wedge A_{cp}) \Rightarrow Q$

for every assertion A_{cp} in $PO(S)$, where A_{cp} is the assertion that $PO(S)$ associates with control predicate cp. For example, we prove as follows that for the absolute value program in Figure 2.2, $after(S) \Rightarrow x{=}abs(\mathrm{X})$ holds during executions started in a state satisfying $at(S) \wedge x{=}\mathrm{X}$: First, because $post(S) \Rightarrow x{=}abs(\mathrm{X})$ is valid, for the case where cp is $after(S)$, (2.8), which is

$$after(S) \wedge post(S) \Rightarrow (after(S) \Rightarrow x{=}abs(\mathrm{X})),$$

is valid. Second, for the case where cp is not implied by $after(S)$, (2.8) is trivially valid.

3. Real-time Actions

We must know something about the execution times of atomic actions in order to reason about timing properties of programs. Therefore, for each unconditional atomic action[4] α in our programming language, we define corresponding *real-time actions* $\alpha_{[\delta,\,\varepsilon]}$ where δ and ε are real-valued, non-negative constants. Execution of a real-time action $\alpha_{[\delta,\,\varepsilon]}$ causes the same indivisible state transformation as α does, but constrains it to occur at some instant between ε and $\varepsilon+\delta$ time units after the entry control point for $\alpha_{[\delta,\,\varepsilon]}$ becomes active.

---

[4]An atomic action is *unconditional* if it is executable whenever its entry control point becomes active. In the programming notation of §2.3, **skip**, assignment, and the guard evaluation action for **do** are unconditional. The guard evaluation for **if** is not unconditional.

We have elected to characterize the execution time for a real-time action in terms of two parameters (δ and ε) in order to have flexibility in modeling various execution environments. Parameter ε describes the fixed execution time of the atomic action on a bare machine; δ models execution delays attributable to multiprogramming and other resource contention. A system where each process is assigned its own processor is modeled by choosing 0 for δ; a system where processors are shared is modeled by choosing a value for δ based on the length of time that a runnable process might have to wait for a processor to become available.

3.1. Reasoning About Real-time Actions

Execution of a real-time action $\alpha_{[\delta,\,\varepsilon]}$ affects the program variables and control predicates in the same ways as the atomic action α from which it was derived. Therefore, we have the following inference rule:

> *Real-time Action Transformation*: For α an unconditional atomic action, P and Q primitive assertions, and $0\leq\delta$ and $0\leq\varepsilon$:
>
> $$\frac{\{P\}\ \alpha\ \{Q\}}{\{P\}\ \alpha_{[\delta,\,\varepsilon]}\ \{Q\}}$$

To reason about timing properties, additional terms must be added the assertion language. This is because the method of §2.4 for reasoning about safety properties can only be used to prove safety properties for which the negation of the proscribed $\neg Q$ is implied by each of a proof outline's assertions. Timing properties concern the instants at which control predicates become active and so we define a term for each control predicate cp:

$$\uparrow cp \begin{cases} \text{the time that } cp \text{ last became } \mathit{true} \text{ or} \\ -\infty \text{ if } cp \text{ has never been } \mathit{true} \end{cases}$$

We also define a new real-valued term $\mathcal{T}$ to be equal to the current time.

Some additional axioms and inference rule allow us to reason about formulas of our more expressive assertion language. First, the various non-atomic statements of our programming language give rise to axioms based on the way they equate their components' control points. For our programming language, these axioms are given in Figure 3.1. Second, there are some language-independent axioms. In these, cp and cp' can denote any control predicates, including those not associated with entry or exit control points for real-time actions.

(3.1) $\uparrow cp \leq \mathcal{T}$

(3.2) $(\uparrow cp = -\infty) \Rightarrow \neg cp$

(3.3) For a real time action $\alpha_{[\delta,\,\varepsilon]}$ with label S: (a) $at(S) \Rightarrow \uparrow at(S) \leq \mathcal{T} \leq \uparrow at(S)+\delta+\varepsilon$
(b) $\uparrow at(S) \neq -\infty \Rightarrow \uparrow after(S) \leq \uparrow at(S)+\delta+\varepsilon$

Axioms (3.1) and (3.2) follow directly from the definition of $\uparrow cp$. Axiom (3.3) captures that essence of a real-time action—that its entry control point cannot stay active too long. This, in turn, allows us to infer that a control point is not active by using

Sequential Composition Axioms: For S the sequential composition $S_1\ S_2$:

(a) $\uparrow at(S) = \uparrow at(S_1)$

(b) $\uparrow after(S) = \uparrow after(S_2)$

(c) $\uparrow after(S_1) = \uparrow at(S_2)$

if *Axioms*: For an **if** statement:

$$S\text{: }\mathbf{if}\ B_1 \rightarrow S_1\ \ [\!]\ \ B_2 \rightarrow S_2\ \ [\!]\ \ \cdots\ \ [\!]\ \ B_n \rightarrow S_n\ \mathbf{fi}$$

(a) $\uparrow at(S) = \uparrow at(GEval_{if}(S))$

(b) $\uparrow after(S) = \max(\uparrow after(S_1), \uparrow after(S_2), ..., \uparrow after(S_n))$

(c) $\uparrow after(GEval_{if}(S)) = \max(\uparrow at(S_1), \uparrow at(S_2), ...\uparrow at(S_n))$

do *Axioms*: For a **do** statement:

$$S\text{: }\mathbf{do}\ B_1 \rightarrow S_1\ \ [\!]\ \ B_2 \rightarrow S_2\ \ [\!]\ \ \cdots\ \ [\!]\ \ B_n \rightarrow S_n\ \mathbf{od}$$

(a) $\uparrow at(GEval_{do}(S)) = \max(\uparrow at(S), \uparrow after(S_1), \uparrow after(S_2), ..., \uparrow after(S_n))$

(b) $\uparrow after(GEval_{do}(S)) = \max(\uparrow at(S_1), \uparrow at(S_2), ..., \uparrow at(S_n), \uparrow after(S))$

cobegin *Axioms*: For a **cobegin** statement:

$$S\text{: }\mathbf{cobegin}\ S_1\ //\ S_2\ //\ \cdots\ //\ S_n\ \mathbf{coend}$$

(a) $\uparrow at(S) = \uparrow at(S_1) = \cdots = \uparrow at(S_n))$

(b) $\uparrow after(S) = \max(\uparrow after(S_1), ..., \uparrow after(S_n))$

Figure 3.1. $\uparrow cp$ Axioms

(3.4) $\quad \mathcal{T} > \uparrow at(S)+\delta+\varepsilon \Rightarrow \neg\, at(S)$

because from (3.3a) we have:

$$\begin{array}{ll}
 & at(S) \Rightarrow \uparrow at(S) \le \mathcal{T} \le \uparrow at(S)+\delta+\varepsilon \\
= & \text{« Predicate Logic»} \\
 & at(S) \Rightarrow ((\uparrow at(S) \le \mathcal{T}) \wedge (\mathcal{T} \le \uparrow at(S)+\delta+\varepsilon) \\
= & \text{« Predicate Logic»} \\
 & ((\uparrow at(S) > \mathcal{T}) \vee (\mathcal{T} > \uparrow at(S)+\delta+\varepsilon)) \Rightarrow \neg\, at(S) \\
= & \text{«Axiom (3.1)»} \\
 & \mathcal{T} > \uparrow at(S)+\delta+\varepsilon \Rightarrow \neg\, at(S)
\end{array}$$

The effect on these new terms of executing atomic actions is captured by the following axioms of Proof Outline Logic. First, for any ordinary or real-time atomic action, we have:

$\uparrow cp$ *Invariance*: $\{cp = \mathrm{C} \wedge \uparrow cp = \mathrm{V}\}\ S\text{: }\alpha\ \{(cp \Rightarrow \mathrm{C}) \Rightarrow (\uparrow cp = \mathrm{V})\}$

The antecedent in the postcondition is necessary for the case where cp is $after(S)$, since executing S does change the value of $after(S)$.

Next, for any ordinary atomic action:

Action-time Axioms: (a) $\{K \le \uparrow at(S)\}\ S: \alpha\ \{K \le \uparrow after(S)\}$

(b) $\{K \le \mathcal{T}\}\ S: \alpha\ \{K \le \uparrow after(S)\}$

Action-time Axiom (a) asserts that the exit control point for S becomes active after any of its entry control points last became active. Action-time Axiom (b) asserts that the exit control point of S becomes active later than any time that the entry control point for S was last active.

For a real-time action $\alpha_{[\delta,\varepsilon]}$, the following axiom characterizes how execution changes $\mathcal{T}$ and the $\uparrow cp$-terms.

Real-time Action Axiom $\{K \le \uparrow at(S)\}\ S: \alpha_{[\delta,\varepsilon]}\ \{K+\varepsilon \le \uparrow after(S)\}$

This axiom is analogous to Action-time Axiom (a), except now the postcondition has been strengthened to give a tighter lower bound on when the exit control point for S first becomes active.

Two things that the Real-time Action Axiom does not say are worthy of note. First, this axiom does not bound the interval during which the entry control point for S is active. This is because that bound already can be derived using axiom (3.3a), since $at(S)$ holds whenever the entry control point for S does. Second, one might expect to be able to prove the following triple—its precondition being similar to that of Action-time Axiom (b).

(3.5) $\{K \le \mathcal{T}\}\ S: \alpha_{[\delta,\varepsilon]}\ \{K+\varepsilon \le \mathcal{T}\}$

Unfortunately, (3.5) is not sound. Execution of S started in a state such that $\uparrow at(\alpha) < K \le \mathcal{T}$ would satisfy the precondition but might terminate before $K+\varepsilon$. For example, consider an execution of $\alpha_{[0,2]}$ that is started at time 0. Thus, at time $\mathcal{T}=1$ the state would satisfy $K \le \mathcal{T}$ for $K=1$, and so precondition $K \le \mathcal{T}$ would be satisfied by that state. When execution of $\alpha_{[0,2]}$ terminates—2 units after it is started—at time $\mathcal{T}=2$, the postcondition $K+\varepsilon \le \mathcal{T}$ is $1+2 \le 2$, which is *false*.

Finally, the following rule allows rigid variables to be instantiated with expressions involving $\uparrow cp$-terms. (Rigid Variable Rule only allows rigid variables to be instantiated by constants, rigid variables, or expressions constructed from these.)

$$\uparrow cp\text{-}Instantiation \quad \frac{\{\uparrow cp = V\}\ \alpha\ \{\uparrow cp = V\},\quad \{P\}\ \alpha\ \{Q\}}{\{P^{X}_{\uparrow cp}\}\ \alpha\ \{Q^{X}_{\uparrow cp}\}}$$

This rule is typically used along with one of the Action-time Axioms or the Real-time Action Axiom. For the case where real-time action α and control predicate cp are in different processes, the first hypothesis of $\uparrow cp$-Instantiation is automatically satisfied, as the following proof of $\{\uparrow cp = V\}\ \alpha\ \{\uparrow cp = V\}$ demonstrates.

Process Independence Axiom:
1. $\{at(\beta)=C\}\ \alpha\ \{at(\beta)=C\}$

$\uparrow cp$ Invariance:
2. $\{at(\beta)=C \wedge \uparrow at(\beta)=V\}\ \alpha\ \{(at(\beta) \Rightarrow C) \Rightarrow (\uparrow at(\beta)=V)\}$

Conjunction Rule with 1 and 2:

3. $\{at(\beta)=C \wedge \uparrow at(\beta)=V\}$ α $\{at(\beta)=C \wedge ((at(\beta) \Rightarrow C) \Rightarrow (\uparrow at(\beta)=V))\}$

4. $at(\beta)=C \wedge ((at(\beta) \Rightarrow C) \Rightarrow (\uparrow at(\beta)=V))$
   $\Rightarrow$ «Predicate Logic»
   $\uparrow at(\beta)=V$

Rule of Consequence with 3 and 4:
5. $\{at(\beta)=C \wedge \uparrow at(\beta)=V\}$ α $\{\uparrow at(\beta)=V\}$

Rigid Variable Rule with 5, replacing C by *true* and then by *false*:
6. $\{at(\beta) \wedge \uparrow at(\beta)=V\}$ α $\{\uparrow at(\beta)=V\}$
7. $\{\neg at(\beta) \wedge \uparrow at(\beta)=V\}$ α $\{\uparrow at(\beta)=V\}$

Disjunction Rule with 6 and 7:
8. $\{(at(\beta) \vee \neg at(\beta)) \wedge \uparrow at(\beta)=V\}$ α $\{\uparrow at(\beta)=V\}$

Equivalence Rule with 8:
9. $\{\uparrow at(\beta)=V\}$ α $\{at(\beta)=V\}$

Thus, we obtain a derived rule of inference:

Derived $\uparrow$cp-Instantiation: If atomic action α and control predicate cp are in different processes:

$$\frac{\{P\}\ \alpha\ \{Q\}}{\{P^{V}_{\uparrow cp}\}\ \alpha\ \{Q^{V}_{\uparrow cp}\}}$$

3.2. Interference Freedom Revisited

When the execution times of atomic actions are bounded, certain forms of interference cannot occur. This is illustrated by the proof outline

$\{x=0\}$
cobegin
  $\{x=0\}$ α: $\langle x := x+1 \rangle_{[0,2]}$ $\{x=1\}$
//
  $\{x=0\}$ β: $\langle y := x+1 \rangle_{[0,1]}$ $\{y=1\}$
coend
$\{x=1 \wedge y=1\}$

which is valid but cannot be derived using the **cobegin** Rule because $PO(\alpha)$ and $PO(\beta)$ are not interference free. In particular, $NI(\alpha, pre(\beta))$ is not valid.

$NI(\alpha, pre(\beta))$
$= \{pre(\alpha) \wedge pre(\beta)\}$ $\langle x := x+1 \rangle_{[0,2]}$ $\{pre(\beta)\}$
$= \{x=0\}$ $\langle x := x+1 \rangle_{[0,2]}$ $\{x=0\}$

Using operational reasoning, however, it is not difficult to argue that execution of α cannot invalidate $pre(\beta)$ and so $PO(\alpha)$ and $PO(\beta)$ should be considered interference free. This is because according to **cobegin** Axiom (b) in Figure 3.1 both $at(\alpha)$ and $at(\beta)$ become active at the same instant, say time 0. By definition, α completes at time 2, and so x remains 0 until this time. Real-time action β completes at time 1 and, therefore, must find x to be 0. It is simply not possible for α to change the value of x while $at(\beta)$ is active.

Our **cobegin** Rule is based on a form of interference freedom that does not take into account execution-time bounds of real-time actions. In particular, $NI(\alpha, A_{cp})$ does not account for the fact that although A_{cp} might be associated with an active control point cp when α is started, if A is the precondition of a real-time action then we may be able to prove that cp cannot be active when α completes. The remedy is to refine $NI(\alpha, A_{cp})$ taking into account the time bounds for how long an entry control point for a real-time action can remain active. The following triple accomplishes this.

$$NI_{rt}(\alpha, A_{cp}):\ \{at(\alpha) \wedge pre(\alpha) \wedge cp \wedge A_{cp}\}\ \ \alpha\ \{cp \Rightarrow A_{cp}\}$$

Returning to the example above, we have:

$$\begin{aligned} & NI_{rt}(\alpha, pre(\beta)) \\ = \ & \{at(\alpha) \wedge pre(\alpha) \wedge at(\beta) \wedge pre(\beta)\}\ \langle x := x+1\rangle_{[0,2]}\ \{at(\beta) \Rightarrow pre(\beta)\} \\ = \ & \{at(\alpha) \wedge at(\beta) \wedge x=0\}\ \langle x := x+1\rangle_{[0,2]}\ \{at(\beta) \Rightarrow x=0\} \end{aligned}$$

And, this obligation can be discharged as follows.

Real-time Action Axiom:
1. $\{K \le \uparrow at(\alpha)\}\ \alpha{:}\ \langle x := x+1\rangle_{[0,2]}\ \{K+2 \le \uparrow after(\alpha)\}$

Derived $\uparrow cp$-Instantiation with 1:
2. $\{\uparrow at(\beta) \le \uparrow at(\alpha)\}\ \alpha{:}\ \langle x := x+1\rangle_{[0,2]}\ \{\uparrow at(\beta)+2 \le \uparrow after(\alpha)\}$

Axiom (3.1):
3. $\uparrow after(\alpha) \le \mathcal{T}$

Rule of Consequence with 2 and 3:
4. $\{\uparrow at(\beta) \le \uparrow at(\alpha)\}\ \alpha{:}\ \langle x := x+1\rangle_{[0,2]}\ \{\uparrow at(\beta)+2 \le \mathcal{T}\}$

Axiom (3.3a):
5. $at(\beta) \Rightarrow \uparrow at(\beta) \le \mathcal{T} \le \uparrow at(\beta)+1$

Predicate Logic:
6. $((\uparrow at(\beta)+2 \le \mathcal{T})\ \wedge\ (at(\beta) \Rightarrow \uparrow at(\beta) \le \mathcal{T} \le \uparrow at(\beta)+1)) \Rightarrow \neg at(\beta)$

Rule of Consequence with 4, 5, and 6:
7. $\{\uparrow at(\beta) \le \uparrow at(\alpha)\}\ \alpha{:}\ \langle x := x+1\rangle_{[0,2]}\ \{\neg at(\beta)\}$

Predicate Logic and $\uparrow at(a) = \uparrow at(b)$ from **cobegin** $\uparrow cp$ Axiom (a):
8. $pre(NI_{rt}(\alpha, pre(\beta))) \Rightarrow \uparrow at(\beta) \le \uparrow at(\alpha)$
9. $\neg at(\beta) \Rightarrow post(NI_{rt}(\alpha, pre(\beta)))$

Rule of Consequence with 7, 8, and 9:
10. $NI_{rt}(\alpha, pre(\beta))$

4. Example: A Mutual Exclusion Protocol

Knowledge of execution times can be exploited to synchronize processes. A mutual exclusion protocol attributed in [Lamport 87] to Mike Fischer illustrates this point. The core of this protocol appears in Figure 4.1. There, c, d, c' and d' are real-time actions. Provided the parameters of these

$$
\begin{array}{l}
x := 0 \\
\textbf{cobegin} \\
\quad b\colon\ \textbf{if}\ x{=}0 \ \rightarrow\ c\colon \langle x := 1\rangle_{[\delta(c),\,\varepsilon(c)]}\ \textbf{fi} \\
\quad d\colon\ \langle\textbf{skip}\rangle_{[\delta(d),\,\varepsilon(d)]} \\
\quad e\colon\ \textbf{if}\ x{=}1 \ \rightarrow\ f\colon \textbf{skip fi} \\
\qquad \text{Critical Section 1} \\
// \\
\quad b'\colon\ \textbf{if}\ x{=}0 \ \rightarrow\ c'\colon \langle x := 2\rangle_{[\delta(c'),\,\varepsilon(c')]}\ \textbf{fi} \\
\quad d'\colon\ \langle\textbf{skip}\rangle_{[\delta(d'),\,\varepsilon(d')]} \\
\quad e'\colon\ \textbf{if}\ x{=}2 \ \rightarrow\ f'\colon \textbf{skip fi} \\
\qquad \text{Critical Section 2} \\
\textbf{coend}
\end{array}
$$

Figure 4.1. Mutual Exclusion Protocol

real-time actions satisfy

(4.1) $\delta(c')+\varepsilon(c')<\varepsilon(d)$

(4.2) $\delta(c)+\varepsilon(c)<\varepsilon(d')$

this protocol implements mutual exclusion of the marked critical sections.

Mutual exclusion of *after*(e) and *after*(e') is a safety property. It can be proved by constructing a valid proof outline in which $post(e) \Rightarrow \neg\, after(e')$ and $post(e') \Rightarrow \neg\, after(e)$. A standard approach for this is to construct a valid proof outline in which $\neg\,(post(e) \wedge post(e'))$ is valid. It is thus impossible for $after(e) \wedge after(e')$ to hold because that would imply $post(e) \wedge post(e')$.

A proof outline for one process is given in Figure 4.2; the proof outline for the other process is

$$
\begin{array}{l}
\quad \{\mathit{true}\} \\
b\colon \textbf{if}\ x{=}0\ \rightarrow\ \{\uparrow at(c') \le \mathcal{T}\} \\
\qquad\qquad\qquad c\colon \langle x := 1\rangle_{[\delta(c),\,\varepsilon(c)]} \\
\qquad\qquad\qquad \{x\neq 0 \wedge (at(c') \Rightarrow \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \uparrow at(d))\} \\
\quad \textbf{fi}\ \{x\neq 0 \wedge (at(c') \Rightarrow \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \uparrow at(d))\} \\
d\colon \langle\textbf{skip}\rangle_{[\delta(d),\,\varepsilon(d)]} \\
\quad \{x\neq 0 \wedge \neg\, at(c')\} \\
e\colon \textbf{if}\ x{=}1\ \rightarrow\ \{x{=}1 \wedge \neg\, at(c')\} \\
\qquad\qquad\qquad f\colon \textbf{skip} \\
\qquad\qquad\qquad \{x{=}1 \wedge \neg\, at(c')\} \\
\quad \textbf{fi}\ \{x{=}1 \wedge \neg\, at(c')\} \\
\quad \text{Critical Section 1}
\end{array}
$$

Figure 4.2. Proof Outline for Mutual Exclusion Protocol

symmetric, with "1" everywhere replaced by "2" and the primed labels interchanged with unprimed ones. Notice that $post(e) \Rightarrow x=1$ and $post(e') \Rightarrow x=2$. Thus, the proof outlines satisfy the conditions just outlined for ensuring that states satisfying $after(e) \wedge after(e')$ cannot occur.

It is not difficult to derive the proof outline of Figure 4.2 using the axiomatization of real-time actions given above. The proofs of $\{pre(c)\}\ c\ \{post(c)\}$ and $\{pre(d)\}\ d\ \{post(d)\}$ are the most enlightening, as they expose the role of assumptions (4.1) and (4.2) in the correctness of the protocol. Here is the proof of $\{pre(c)\}\ c\ \{post(c)\}$:

Assignment Axiom:

1. $\{true\}\ c: \langle x := 1\rangle_{[\delta(c),\, \varepsilon(c)]}\ \{x=1\}$

2. $x=1$
   $\Rightarrow$ «Axiom (3.1)»
   $x=1 \wedge \uparrow at(c') \leq \mathcal{T}$
   $\Rightarrow$ «assumption (4.1)»
   $x=1 \wedge \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \mathcal{T}$
   $\Rightarrow$ «Predicate Logic»
   $x \neq 0 \wedge \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \mathcal{T}$

Rule of Consequence with 1 and 2:

3. $\{true\}\ c: \langle x := 1\rangle_{[\delta(c),\, \varepsilon(c)]}\ \{x \neq 0 \wedge \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \mathcal{T}\}$

Action-time Axiom (b):

4. $\{\mathrm{K} \leq \mathcal{T}\}\ c: \langle x := 1\rangle_{[\delta(c),\, \varepsilon(c)]}\ \{\mathrm{K} \leq \uparrow after(c)\}$

Derived $\uparrow cp$-Instantiation with 4:

5. $\{\uparrow at(c') \leq \mathcal{T}\}\ c: \langle x := 1\rangle_{[\delta(c),\, \varepsilon(c)]}\ \{\uparrow at(c') \leq \uparrow after(c)\}$

Conjunction Rule with 3 and 5:

6. $\{\uparrow at(c') \leq \mathcal{T}\}$
   $c: \langle x := 1\rangle_{[\delta(c),\, \varepsilon(c)]}$
   $\{x \neq 0 \wedge \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \mathcal{T} \wedge \uparrow at(c') \leq \uparrow after(c)\}$

7. $\uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \mathcal{T} \wedge \uparrow at(c') \leq \uparrow after(c)$
   $\Rightarrow$ «assumption (4.1) and $\uparrow after(c) = \uparrow at(d)$»
   $\uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \uparrow at(d)$
   $\Rightarrow$ «Predicate Logic »
   $at(c') \Rightarrow \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \uparrow at(d)$

Rule of Consequence with 6 and 7:

8. $\{\uparrow at(c') \leq \mathcal{T}\}\ c: \langle x := 1\rangle_{[\delta(c),\, \varepsilon(c)]}\ \{x \neq 0 \wedge (at(c') \Rightarrow \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d) < \uparrow at(d))\}$

And, here is the proof of $\{pre(d)\}\ d\ \{post(d)\}$.

skip Axiom:

1. $\{x \neq 0\}\ d: \langle \mathbf{skip}\rangle_{[\delta(d),\, \varepsilon(d)]}\ \{x \neq 0\}$

Real-time Action Axiom:

2. $\{\mathrm{K} \leq \uparrow at(d)\}\ d: \langle \mathbf{skip}\rangle_{[\delta(d),\, \varepsilon(d)]}\ \{\mathrm{K}+\varepsilon(d) \leq \uparrow after(d)\}$

Rigid Variable Rule with 2, instantiating K with $L+\delta(c')+\varepsilon(c')-\varepsilon(d)+\kappa$ where $0<\kappa$:

3. $\{L+\delta(c')+\varepsilon(c')-\varepsilon(d)+\kappa \le \uparrow at(d)\}$
   $d: \langle \mathbf{skip} \rangle_{[\delta(d),\, \varepsilon(d)]}$
   $\{L+\delta(c')+\varepsilon(c')-\varepsilon(d)+\kappa+\varepsilon(d) \le \uparrow after(d)\}$

Predicate Logic, since $0<\kappa$:

4. $L+\delta(c')+\varepsilon(c')-\varepsilon(d)<\uparrow at(d) \;\Rightarrow\; L+\delta(c')+\varepsilon(c')-\varepsilon(d)+\kappa \le \uparrow at(d)$
5. $L+\delta(c')+\varepsilon(c')-\varepsilon(d)+\kappa+\varepsilon(d) \le \uparrow after(d) \;\Rightarrow\; L+\delta(c')+\varepsilon(c')<\uparrow after(d)$

Rule of Consequence with 3, 4, and 5:

6. $\{L+\delta(c')+\varepsilon(c')-\varepsilon(d)<\uparrow at(d)\}$
   $d: \langle \mathbf{skip} \rangle_{[\delta(d),\, \varepsilon(d)]}$
   $\{L+\delta(c')+\varepsilon(c')<\uparrow after(d)\}$

Derived $\uparrow cp$-Instantiation, replacing L by $\uparrow at(c')$:

7. $\{\uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d)<\uparrow at(d)\}$
   $d: \langle \mathbf{skip} \rangle_{[\delta(d),\, \varepsilon(d)]}$
   $\{\uparrow at(c')+\delta(c')+\varepsilon(c')<\uparrow after(d)\}$

8. $\uparrow at(c')+\delta(c')+\varepsilon(c')<\uparrow after(d)$
   $\Rightarrow$ «Axiom (3.1) applied to $after(d)$»
   $\uparrow at(c')+\delta(c')+\varepsilon(c')<\uparrow after(d) \le \mathcal{T}$
   $\Rightarrow$ «theorem (3.4) applied to $at(c')$»
   $\neg\, at(c')$

Rule of Consequence with 7 and 8:

9. $\{\uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d)<\uparrow at(d)\}\ d: \langle \mathbf{skip} \rangle_{[\delta(d),\, \varepsilon(d)]}\ \{\neg\, at(c')\}$

Process Independence Axiom:

10. $\{\neg\, at(c')\}\ d: \langle \mathbf{skip} \rangle_{[\delta(d),\, \varepsilon(d)]}\ \{\neg\, at(c')\}$

Disjunction Rule with 9 and 10:

11. $\{at(c') \Rightarrow \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d)<\uparrow at(d)\}\ d: \langle \mathbf{skip} \rangle_{[\delta(d),\, \varepsilon(d)]}\ \{\neg\, at(c')\}$

Conjunction Rule with 1 and 11:

12. $\{x \ne 0 \wedge (at(c') \Rightarrow \uparrow at(c')+\delta(c')+\varepsilon(c')-\varepsilon(d)<\uparrow at(d))\}$
    $d: \langle \mathbf{skip} \rangle_{[\delta(d),\, \varepsilon(d)]}$
    $\{x \ne 0 \wedge \neg\, at(c')\}$

Notice how timing information is used in step 7 to infer that a particular control point cannot be active.

5. Discussion

5.1. Other Work based on Proof Outlines

It is instructive to compare our logic with that of [Shaw 89], another Hoare-style logic [Hoare 69] for reasoning about execution of real-time programs. In [Shaw 89], the passage of time is modeled by augmenting each atomic action with an assignment to an interval-valued variable *RT* so that RT contains lower and upper bounds for the program's elapsed execution time. The Statement Composition Rule and the Assignment Axiom are then used to derive rules for reasoning about these augmented atomic actions.[5] Our logic is obtained by augmenting the assertion language (of an underlying logic of proof outlines) with additional terms ($\uparrow cp$ and $\mathcal{T}$) and devising new axioms for reasoning about these terms. We are not able to derive rules for real-time actions by using the original logic because we do not employ assignment statements to model the passage of time.

Although more complex, augmenting the axioms rather than the atomic actions has led us to a more powerful logic. First, having the $\uparrow cp$-terms allows the logic to be more expressive. These terms permit the definition of properties involving historical information—information that is not part of the current state of the program. Timing properties that constrain the elapsed time between events can only be formulated in terms of such historical information. The logic of [Shaw 89] has no way to express historical information and, consequently, can be employed to reason about only certain timing properties.

Second, our axiomatization allows reasoning about programs whose timing behavior is data-dependent. The logic of [Shaw 89] does not permit such reasoning. For example, because of the way statement composition is handled in [Shaw 89], the logic produces overly-conservative intervals for time bounds. This is illustrated by the following program, which takes exactly 10 time units to execute.

if $B \rightarrow \mathbf{skip}_{[0,9]}$ $[]$ $\neg B \rightarrow \mathbf{skip}_{[0,1]}$ **fi**
if $B \rightarrow \mathbf{skip}_{[0,1]}$ $[]$ $\neg B \rightarrow \mathbf{skip}_{[0,9]}$ **fi**

This fact can be proved in our logic; the logic of [Shaw 89] can prove only that execution requires between 2 and 18 time units.

A Hoare-style programming logic for reasoning about real-time is also discussed in [Hooman 91]. That work is largely incomparable to ours. First, the programming language axiomatized in [Hooman 91] is different, having synchronous message-passing and no shared variables. This is symptomatic of a fundamental difference in the two approaches. The emphasis in [Hooman 91] is on the design of compositional proof systems. Shared variables cannot (at present) be handled compositionally and so they are excluded from programs. In contrast, we do not require that our proof system be compositional.[6] Relaxing this compositionality requirement means that it is not difficult to extend our logic for reasoning (non-compositionally) about programs that employ synchronous message-passing or any of the other communication/synchronization mechanisms for which Hoare-style axioms have been proposed.

The types of properties handled in [Hooman 91] is also incomparable to what can be proved using our logic. Timing properties make visible the times at which control points become active through $\uparrow cp$-terms. A compositional proof system cannot include information about control points in its formulas because they betray the internal structure of a component. The logic of [Hooman 91], therefore, may only be concerned with the times at which externally visible events occur: the time of communications events and the time that program execution starts and terminates. This turns out to

[5]The idea of augmenting actions with assignment statements in order to reason about the passage of time is discussed in [Haase 81], where it is used to extend Dijkstra's *wp* [Dijkstra 75] for reasoning about elapsed execution time.

[6]The **cobegin** Rule of Proof Outline Logic is not compositional because its interference-freedom test depends on the internal structure of the processes being composed.

allow proofs of certain liveness properties as well as certain safety properties. Our logic cannot be used to prove any liveness properties.

5.2. Incompleteness Concerns

A soundness proof for the logic of this paper will appear elsewhere. The issue of completeness, however, is a bit problematic. The following proof outline illustrates the difficulties. It is valid, but is not provable with our logic.

(5.1) $\{\mathcal{T}=0\}$ a: $\mathbf{skip}_{[0,2]}$ $\{\mathcal{T}=2\}$ b: $\mathbf{skip}_{[0,2]}$ $\{\mathcal{T}=4\}$

A related proof outline is provable:

(5.2) $\{0 \leq \uparrow at(a) \leq \mathcal{T} \leq 2\}$ a: $\mathbf{skip}_{[0,2]}$ $\{2 \leq \uparrow at(b) \leq \mathcal{T} \leq 4\}$ b: $\mathbf{skip}_{[0,2]}$ $\{4 \leq \uparrow after(b) \leq \mathcal{T}\}$

Notice that the assertions of (5.2) characterize system states that would exist "during" the execution of a and b; the assertions of (5.1) do not.

A deficiency in our logic is one explanation for this situation; a deficiency in the definition of proof outline validity is another. Proof outline validity is defined in terms of a set ($\mathcal{H}_S^{+}$) of infinite state sequences that model execution of S started from any program state. This set contains no sequence whose successive states differ only in their values of $\mathcal{T}$, the states that assertions in (5.2) characterize and those in (5.1) do not. Certainly such states exist during program execution; we have simply chosen to define $\mathcal{H}_S^{+}$ so that states are recorded only when the value of some $\uparrow cp$-term changes. Now consider a set $\mathcal{H}_S^{++}$ that does contain sequences having such *temporal interpolation* states. If we replace $\mathcal{H}_S^{+}$ in Valid Proof Outline (2.7) by $\mathcal{H}_S^{++}$, then (5.2) remains valid and (5.1) becomes invalid. The incompleteness problem is gone.

There are also other reasons to prefer $\mathcal{H}_S^{++}$ in defining proof outline validity. Invariance under temporal interpolation seems to be the real-time analog of invariance under stuttering, something that is critical when proving that one specification or a program implements another. Unfortunately, the logic of this paper is unsound when $\mathcal{H}_S^{++}$ is used in place of $\mathcal{H}_S^{++}$. The existence of temporal interpolation states causes a new form of interference. This interference is easily dealt with by extending the definition of interference freedom.

Another concern when designing a logic is expressive completeness. Timing properties include many, but not all, safety properties of concern when reasoning about the behavior of real-time programs. This is because the historical information in a timing property is limited to times that control points become active. One might also be concerned with the elapsed time since the program variables last satisfied a given predicate or with satisfying constraints about how the program variables change as a function of time. Both are safety properties but neither is a timing property (according to our definition in §1). In general, safety properties can be partitioned into *invariance properties* and *history properties*. The invariant used in proving an invariance property need only refer to the current state; the invariant used in proving a history property may need to refer to the sequence of states up to the current state. Timing properties are a type of history property.

A version of Proof Outline Logic does exist for reasoning about history properties [Schneider 92]. It extends ordinary Proof Outline Logic by augmenting the assertion language with a "past state" operator and a function-definition facility. In this logic, our $\uparrow cp$-terms can be constructed explicitly; they need not be primitive. And, the more general class of safety properties involving times—be it times that predicates hold or times that control predicates hold—can be handled.

Acknowledgments

We are grateful to Nancy Lynch for pointing out an error in an earlier version of this paper.

References

[Dijkstra 75] Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs. *CACM 18*, 8 (Aug. 1975), 453-457.

[Gries-Levin 80] Gries, D., and G. Levin. Assignment and procedure call proof rules. *ACM TOPLAS 2*, 4 (Oct. 1980), 564-579.

[Haase 81] Haase, V. Real-time Behavior of Programs. *IEEE Transactions on Software Engineering SE-7*, 5 (Sept. 1981), 494-501.

[Hoare 69] Hoare, C.A.R. An axiomatic basis for computer programming. *CACM 12*, 10 (Oct. 1969), 576-580.

[Hooman 91] Hooman, J. *Specification and Compositional Verification of Real-time Systems*. Ph.D. Thesis. Technische Universiteit Eindhoven. May 1991.

[Lamport 87] Lamport, L. A fast mutual exclusion algorithm. *ACM TOCS 5*, 1 (Feb. 1987), 1-11.

[Owicki-Gries 76] Owicki, S.S., and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica 6*, (1976), 319-340.

[Owicki-Lamport 82] Owicki, S.S., and L. Lamport. Proving liveness properties of concurrent programs. *ACM TOPLAS 4*, 3 (July 1982), 455-495.

[Schneider 92] Schneider, F.B. *On concurrent programming*. In preparation.

[Shaw 89] Shaw, A. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering SE-15*, 7 (July 1989), 875-899.