Putting Time into Proof Outlines*

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

TR 93-1333
March 1993

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

---

*A preliminary version of this work appeared in *Real-time: Theory and Practice,* Proceedings of REX workshop, June 1991, Springer-Verlag Lecture Notes in Computer Science, volume 600.
**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.

Putting Time into Proof Outlines*

Fred B. Schneider†
Cornell University
fbs@cs.cornell.edu

Bard Bloom ‡
Cornell University
bard@cs.cornell.edu

Keith Marzullo §
Cornell University
marzullo@cs.cornell.edu

March 10, 1993

Abstract

A logic for reasoning about timing properties of concurrent programs is presented. The logic is based on Hoare-style proof outlines and can handle maximal parallelism as well as certain 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. A soundness proof using structural operational semantics is outlined in the appendix.

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

1 Introduction

A *safety property* [9] 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 concern externally visible events, like inputs and outputs, as well as events and data that are internal to a program, like the value of a variable or the time that a particular command starts or finishes. For example, in process control applications, the elapsed time between a stimulus and response may have to be bounded. This is a timing property where the "bad thing" is defined in terms of the time that elapses after one control point becomes active until some other control point does. Timing properties concerning internal events are useful in reasoning about ordinary

*A preliminary version of this work appeared in *Real-time: Theory and Practice*, Proceedings of REX workshop, June 1991, Springer-Verlag Lecture Notes in Computer Science, volume 600.

†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] Informally, the *active* control points at any instant are determined by the values of the program counters at that instant. See [15] for a more formal definition.

concurrent programs that exploit knowledge of command 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:

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

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

Point (a) means that in defining L', the language of L might have to be extended so that it becomes more expressive. Point (b) 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.

A logic for timing properties will, of course, depend on the execution model for programs. Like [14], we consider two kinds of actions, *ordinary* actions, which can take any amount of time to execute, and *real-time* actions, which have constant upper and lower bounds on their execution time. Real-time actions allow the modelling of schedulers typically found when multiprogramming is employed to implement processes.

This paper describes extensions to a logic of proof outlines [15] 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 (a) and refine the inference rules according to (b). The presentation is organized as follows. In Section 2 we describe Proof Outline Logic for non-real-time programs. Then, in Section 3, we describe additional mechanisms needed to handle real time. In particular, we describe changes that must be made to the Rule of Consequence and to the definition of non-interference. In Section 4, we illustrate the use of our logic on a mutual exclusion protocol. Section 5 contains discussion of related topics.

An appendix summarizes a Plotkin-style [13] structural operational semantics and soundness proof. The full details of the soundness proof will appear in [2]. Our proof builds a natural model, similar to models built by other researchers in the theory of concurrent programming languages[6, 18, 1, 13]. This method of construction argues for the reasonability of the logic and language as well as proving its soundness, in much the same way that, for example, having the integers as a model of an arithmetic system, or the Scott models for a λ-calculus, give more plausibility than a term model does.

2 Ordinary Proof Outline Logic

In order to reason about a program, we must be able to characterize 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 $y \times (x/y)$ need not equal x when y is 0 because the value of x/y is

unspecified in such states.[2]

Predicates and function symbols for the programming language's data types provide a way to express facts about program variables and expressions. Type declarations in the program induce axioms concerning the values that may be assumed by the variables they declare. For simplicity, we take these as given.

The state of a program also includes information that tells what atomic actions might be executed next. For representing this control information, we fix some predicate symbols, called *control predicates*, and give axioms to ensure that, as execution proceeds, changes in the values of the control predicates correspond to changes in 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 is a structured collection of labelled *commands*, although we omit labels in program texts unless they are needed. Each command comprises one or more *atomic actions*. The defining characteristic of an atomic action is that its execution is performed as a single indivisible state transformation. The *control points* of a program are defined by its atomic actions. Each atomic action has distinct *entry* 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 multiple exit control points, one 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 command 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.

$in(S)$: $at(S')$ holds for some atomic action comprising S.

The various commands of a programming language give rise to a set of axioms relating these control predicates. These axioms formalize how the control predicates for a command 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 our programming language these axioms are given in Figure 1. We use $\mathrm{GEval}_{\mathbf{if}}(S)$ there to denote the guard evaluation action for an **if** with label S and $\mathrm{GEval}_{\mathbf{do}}(S)$ to denote the guard evaluation action for a **do** with label S. Equality of Boolean values is logical equivalence. 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.

Our programming language is based on guarded commands [3]. Sequencing is written without the traditional semicolon separator.[3] The guard evaluation of the **if** command atomically evaluates all its tests at once and selects a branch for which the test is true; if none are true, then it will block until some test is made true by another process. A **do** command atomically evaluates its tests, selects a branch with a true test, and repeats; if none are true, then it exits. A **cobegin** command

[2] More formally, we admit any interpretation in which a/b is interpreted as a divided by b when $b \neq 0$, and has arbitrary values otherwise.

[3] We shall later see that this avoids a problem. A semicolon separator would have to occupy the same position as an assertion in the standard textual representation of the proof outline.

starts all of its component processes simultaneously, executing them concurrently. When they all finish, the **cobegin** finishes as well.

	Construct	Axioms
α	atomic	$\neg(at(\alpha) \wedge after(\alpha))$
		$in(\alpha) = at(\alpha)$
S :	$S_1 S_2$	$at(S) = at(S_1)$
		$after(S) = after(S_2)$
		$after(S_1) = at(S_2)$
		$in(S) = in(S_1) \oplus in(S_2)$
S :	**if** $B_1 \rightarrow S_1 \cdots$	$at(S) = at(\mathrm{GEval}_{\mathrm{if}}(S))$
	$[\!] B_i \rightarrow S_i \cdots$	$after(S) = (after(S_1) \oplus after(S_2) \oplus \ldots \oplus after(S_n))$
	$[\!] B_n \rightarrow S_n$	$after(\mathrm{GEval}_{\mathrm{if}}(S)) = (at(S_1) \oplus at(S_2) \oplus \ldots \oplus at(S_n))$
	fi	$in(S) = in(\mathrm{GEval}_{\mathrm{if}}(S)) \oplus in(S_1) \oplus \ldots \oplus in(S_n)$
S :	**do** $B_1 \rightarrow S_1 \cdots$	$at(\mathrm{GEval}_{\mathrm{do}}(S)) = (at(S) \oplus after(S_1) \oplus after(S_2) \oplus \ldots \oplus after(S_n))$
	$[\!] B_i \rightarrow S_i \cdots$	$after(\mathrm{GEval}_{\mathrm{do}}(S)) = (at(S_1) \oplus at(S_2) \oplus \ldots \oplus at(S_n) \oplus after(S))$
	$[\!] B_n \rightarrow S_n$	$in(S) = in(\mathrm{GEval}_{\mathrm{do}}(S)) \oplus in(S_1) \oplus \ldots \oplus in(S_n)$
	od	$at(S) \Rightarrow at(\mathrm{GEval}_{\mathrm{do}}(S))$
	$(after(\mathrm{GEval}_{\mathrm{do}}(S)) \vee \bigvee_{i=1}^{n} after(S_i))$	$= (after(S_1) \oplus \cdots \oplus after(S_n) \oplus after(\mathrm{GEval}_{\mathrm{do}}(S)))$
S :	**cobegin**	$at(S) = (at(S_1) \wedge \ldots \wedge at(S_n))$
	$S_1 /\!/ \cdots /\!/ S_n$	$after(S) = (after(S_1) \wedge \ldots \wedge after(S_n))$
	coend	

Figure 1: Control Predicate Axioms

2.2 Syntax and Meaning of Proof Outlines

A *proof outline* $PO(S)$ for a program or atomic action S is a mapping from the control predicates of S to assertions. Each *assertion* is a Predicate Logic formula in which

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

Assertions in which all terms are constructed from program variables, rigid variables, and predicates involving those variables are called *primitive assertions*.

If T is a subprogram of S and some $PO(S)$ is fixed, then we write pre(T) (resp. post(T)) for the assertion(s) that $PO(S)$ associates with $at(T)$ (resp. $after(T)$); these are called the *precondition* and *postcondition* of T. For the proof outline in Figure 2, this correspondence is summarized in Figure 3. In it, x is a program variable and X is a rigid variable. All assertions except pre(S) and post(S) are primitive.

Proof outlines are generally represented as texts containing the program S and an assertion in braces before and after every subprogram of S. For our programming language, this will associate

$\{x = \mathrm{X} \wedge at(S)\}$
S: **if** $x \geq 0 \rightarrow \{x = \mathrm{X} \wedge x \geq 0\}$
S_1 : **skip**
$\{x = \mathrm{X} \wedge x \geq 0\}$
$[\!]\ x \leq 0 \rightarrow \{x = \mathrm{X} \wedge x \leq 0\}$
$S_2 : x := -x$
$\{-x = \mathrm{X} \wedge -x \leq 0\}$
fi
$\{x = |\mathrm{X}| \wedge after(S)\}$

Figure 2: Computing $|x|$

at least one assertion with every control predicate of a program. Multiple control predicates may be mapped to the same assertion, as illustrated by the proof outline

$$\{P\}\ x := 1\ \{Q\}\ y := 2\ \{R\}. \tag{1}$$

Here, the entry control point for program $x := 1\ y := 2$, and the entry control point for $x := 1$ map to the same assertion, P. This is reasonable, because $at(\text{“}x := 1\ y := 2\text{”})$ and $at(\text{“}x := 1\text{”})$ are equivalent; if one control point is active, so will be the other.

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* $\{P\}\ PO(S)\ \{Q\}$ 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 $PO(S)$.

Assertion	Assertion Text
pre(S)	$x = \mathrm{X} \wedge at(S)$
post(S)	$x = \lvert\mathrm{X}\rvert \wedge after(S)$
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 3: Assertions in a Proof Outline

Validity of Proof Outlines

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, 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 validity of assertions and proof outlines. A *valid* assertion P is one that holds in all program states, *viz.*, states satisfying the datatype and control predicate

```
{true}
a:  cobegin
        {x = 1}
        b:  skip
        {...}
    //
        {...}
        c:  skip
        {...}
coend
{...}
```

Figure 4: Multiple Active Control Points (Partial proof outline)

axioms. A valid proof outline $PO(S)$ will be one that 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 predicate *cp* is true, then so is every assertion that $PO(S)$ associates with *cp*" and is formalized as the *proof outline invariant* for $PO(S)$:

$$I_{PO(S)}: \bigwedge_{T} \Big((at(T) \Rightarrow \text{pre}(T)) \wedge (after(T) \Rightarrow \text{post}(T)) \Big)$$

where T ranges over the subprograms of S. For example, the proof outline invariant defined by $PO(S)$ of Figure 2 is:

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

Notice that proof outline validity is with respect to executions starting in any state, even a state in the middle of the program. For example, the proof outline

```
        {x = 0 ∧ y = 1}
S:  a:  x := x + 2
        {x = 2}
    b:  y := y + 2
        {x = 2 ∧ y = 3}
```

is not valid. This is because a state where $at(b)$, $x = 2$, and $y = 20$ hold satisfies $I_{PO(S)}$, but starting execution in this state leads to one that does not satisfy $I_{PO(S)}$.

In order to make the inference rules of our logic as easy to use as possible, we have designed them so that hypotheses concerning proof outlines do not delve too deeply into the structure of those proof outlines. Allowing multiple control predicates to be true simultaneously – as we do – complicates the realization of this goal. Consider the concurrent program of Figure 4. There, $at(a)$ is true iff both $at(b)$ and $at(c)$ are true. Thus, in Figure 4, if $at(a)$ and $I_{PO(S)}$ are both true, then $x = 1$ holds. However, pre(a) gives no warning of this requirement; it is the trivial assertion "true".

To avoid such anomalies, we insist that a proof outline $PO(S)$ give its requirements in its precondition. If $at(S)$ and $\text{pre}(PO(S))$ hold, then $I_{PO(S)}$ must hold as well. We call such proof outlines *self-consistent*, and require self-consistency of a valid proof outline along with invariance of $I_{PO(S)}$.

These invariance and self-consistency requirements for proof outline validity can be formalized in terms of $\mathcal{H}_S^+$-validity of Temporal Logic formulas, where $\mathcal{H}_S^+$ contains all infinite state sequences constructed by executing programs $C[S]$ that contain a copy of S. The elements σ of $\mathcal{H}_S^+$ are those infinite sequences of states such that:

- σ_0 is a state reached by executing some program $C[S]$ zero or more steps from its initial state, and
- each state σ_{i+1} is a possible result of performing an atomic action of S from state σ_i, or, if no action of S is enabled, repeating σ_i.

Note that $\mathcal{H}_S^+ \models P$ denotes that a Predicate Logic formula P is valid, because every program state is the first state of some sequence in $\mathcal{H}_S^+$. We now define $PO(S)$ to be *valid* if and only if

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

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

From this definition we see that values of rigid variables are the same throughout any execution, because free rigid variables in a temporal logic formula are implicitly universally quantified. This means that rigid variables in proof outlines can be used relate the values of program variables from one state to the next. For example, the proof outline of Figure 2 is valid and employs 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 X is 41 rather than -23 because then $I_{PO(S)}$ is not satisfied (causing $I_{PO(S)} \Rightarrow \Box I_{PO(S)}$ to be trivially satisfied).

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, commands, 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 command-independent inference rules as well as an axiom or inference rule for each type of command and atomic action. The command-independent rules appear in Figure 5. With one minor exception, they will apply in our real-time setting as well. We now turn to the axiomatization for a guarded-command concurrent programming language.

The **skip** command is a single atomic action whose execution has no effect on any program variable. Thus, it leaves primitive assertions—which only mention program variables and terms that by their nature cannot change—unchanged.

skip Axiom: For a primitive assertion P: $\{P\}$ **skip** $\{P\}$ (7)

The familiar Hoare [7] assignment rule applies:

Rule of Consequence:

$$\frac{P' \Rightarrow P, \quad \{P\}\ PO(S)\ \{Q\}, \quad Q \Rightarrow Q'}{\{P'\}\ PO(S)\ \{Q'\}} \tag{2}$$

Rule of Equivalence:

$$\frac{PO(S), \quad I_{PO(S)} = I_{PO'(S)}, \quad \text{pre}(PO'(S)) \wedge at(S) \Rightarrow \text{pre}(PO(S))}{PO'(S)} \tag{3}$$

Rigid Variable Rule: $PO(S)^{\mathrm{X}}_{\mathrm{Exp}}$ denotes a proof outline in which rigid variable X in every assertion is replaced by Exp, an expression involving constants and rigid variables (only).

$$\frac{\{P\}\ PO(S)\ \{Q\}}{\left\{P^{\mathrm{X}}_{\mathrm{Exp}}\right\}\ PO(S)^{\mathrm{X}}_{\mathrm{Exp}}\ \left\{Q^{\mathrm{X}}_{\mathrm{Exp}}\right\}} \tag{4}$$

Conjunction Rule: $PO_A(S) \textcircled{\wedge} PO_B(S)$ denotes 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.

$$\frac{PO_A(S), \quad PO_B(S)}{PO_A(S) \textcircled{\wedge} PO_B(S)} \tag{5}$$

Disjunction Rule: $PO_A(S) \textcircled{\vee} PO_B(S)$ denotes the proof outline that associates $A_{cp} \vee B_{cp}$ with each control predicate cp.

$$\frac{PO_A(S), \quad PO_B(S)}{PO_A(S) \textcircled{\vee} PO_B(S)} \tag{6}$$

Figure 5: Command-Independent Rules of Ordinary Proof Outline Logic

Assignment Axiom: For a primitive assertion P: $\{P^x_e\}\ x := e\ \{P\}$ (8)

Sequential composition of commands is like Hoare's rule without deletion of the intermediate assertion:

Command Composition Rule:

$$\frac{\{P\}\ S_1\ \{Q\}, \quad \{Q\}\ S_2\ \{R\}}{\{P\}S_1\{Q\}S_2\{R\}} \tag{9}$$

In order to reason about an **if** command, we must reason about its guard evaluation action as well as the commands it guards. The following axiom allows us to derive proof outlines for guard evaluation actions.

$\text{GEval}_{\text{if}}(S)$ **Axiom:** For an **if** command

$$S : \textbf{if } B_1 \rightarrow S_1 \,[\!]\, \cdots \,[\!]\, B_n \rightarrow S_n \textbf{ fi}$$

and a primitive assertion P,

$$\{P\}\ \text{GEval}_{\text{if}}(S)\ \{P \wedge (\textstyle\bigwedge_i at(S_i) \Rightarrow B_i)\} \tag{10}$$

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:

$$\frac{\begin{array}{ll}(a) & \{P\}\ \text{GEval}_{\text{if}}(S)\ \{R\} \\ (b) & (R \wedge at(S_1)) \Rightarrow P_1, \quad \ldots, \quad (R \wedge at(S_n)) \Rightarrow P_n \\ (c) & \{P_1\}\ PO(S_1)\ \{Q\}, \quad \ldots, \quad \{P_n\}\ PO(S_n)\ \{Q\}\end{array}}{\begin{array}{l}\{P\} \\ S: \quad \textbf{if } B_1 \rightarrow \{P_1\} PO(S_1) \{Q\} \\ \qquad [\!] \ \ldots \\ \qquad [\!]\, B_n \rightarrow \{P_n\} PO(S_n) \{Q\} \\ \qquad \textbf{fi} \\ \{Q\}\end{array}} \tag{11}$$

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

$\text{GEval}_{\text{do}}(S)$ **Axiom:** For a **do** command

$$S : \textbf{do } B_1 \rightarrow S_1 \,[\!]\, \cdots \,[\!]\, B_n \rightarrow S_n \textbf{ od}$$

and a primitive assertion P,

$$\{P\}\ \text{GEval}_{\text{do}}(S)\ \{P \wedge (\textstyle\bigwedge_i at(S_i) \Rightarrow B_i) \wedge (after(S) \Rightarrow (\neg B_1 \wedge \cdots \wedge \neg B_n))\} \tag{12}$$

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:

$$\frac{\begin{array}{ll}(a) & \{I\}\ \text{GEval}_{\text{do}}(S)\ \{R\} \\ (b) & (R \wedge at(S_1)) \Rightarrow P_1, \quad \ldots, \quad (R \wedge at(S_n)) \Rightarrow P_n \\ (c) & \{P_1\}\ PO(S_1)\ \{I\}, \quad \ldots, \quad \{P_n\}\ PO(S_n)\ \{I\} \\ (d) & (R \wedge after(S)) \Rightarrow (I \wedge \neg B_1 \wedge \cdots \wedge \neg B_n)\end{array}}{\begin{array}{l}\{I\} \\ S: \quad \textbf{do } B_1 \rightarrow \{P_1\} PO(S_1) \{I\} \\ \qquad [\!] \ \ldots \\ \qquad [\!]\, B_n \rightarrow \{P_n\} PO(S_n) \{I\} \\ \qquad \textbf{od} \\ \{I \wedge \neg B_1 \wedge \cdots \wedge \neg B_n\}\end{array}} \tag{13}$$

The inference rule for a **cobegin** combines proof outlines for its component processes. An interference-freedom test [12] 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,

$$NI(\alpha, A)\text{: } \{\text{pre}(\alpha) \wedge A\}\ \alpha\ \{A\}$$

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:

Interference Freedom (14)
For all $i, j, 1 \leq i \leq n, 1 \leq j \leq n, i \neq j$:
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 characterizes when a valid proof outline for a **cobegin** will result from combining valid proof outlines for its component processes:

cobegin rule:

$$\frac{\begin{array}{ll}(a) & PO(S_1), \quad \ldots, \quad PO(S_n) \\ (b) & P \Rightarrow (\bigwedge_i \text{pre}(PO(S_i))) \\ (c) & \left(\bigwedge_i \text{post}(PO(S_i))\right) \Rightarrow Q \\ (d) & PO(S_1), \ldots, PO(S_n) \text{ are interference free}\end{array}}{\{P\}\ \textbf{cobegin}\ PO(S_1) /\!/ \cdots /\!/ PO(S_n)\ \textbf{coend}\ \{Q\}} \quad (15)$$

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,

$$\{cp = \text{X}\}\ \alpha\ \{cp = \text{X}\} \quad (16)$$

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 in verifying safety properties because of the way that proof outline validity is defined. If a proof outline $PO(S)$ is valid then $I_{PO(S)}$ must be an invariant. Suppose that $I_{PO(S)}$ is an invariant. Then according to the method of Section 1 for proving safety properties, we can prove that executions of S starting with pre$(PO(S))$ true will satisfy the safety property proscribing $\neg Q$ by proving $(at(S) \wedge \text{pre}(PO(S))) \Rightarrow I_{PO(S)}$ and $I_{PO(S)} \Rightarrow \neg Q$. The proof of $(at(S) \wedge \text{pre}PO(S)) \Rightarrow I_{PO(S)}$ follows trivially from the way $I_{PO(S)}$ is defined. And, to prove $I_{PO(S)} \Rightarrow \neg Q$, we simply prove

$$(cp \wedge A_{cp}) \Rightarrow Q \quad (17)$$

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

$after(S) \Rightarrow (x = |X|)$ holds during executions started in a state satisfying $x = X \wedge at(S)$: For the case where cp does not imply $after(S)$, (17) is trivially valid. The remaining cases are when cp is $after(S_1)$, $after(S_2)$ and $after(S)$. Here, we must show

$$\begin{aligned} after(S_1) \wedge \text{post}(S_1) &\Rightarrow (after(S) \Rightarrow (x = |x|)) \\ after(S_2) \wedge \text{post}(S_2) &\Rightarrow (after(S) \Rightarrow (x = |x|)) \\ after(S) \wedge \text{post}(S) &\Rightarrow (after(S) \Rightarrow (x = |x|)) \end{aligned}$$

All are valid.

3 Real-time Logic

3.1 A View of Real Time

In taking into account real-time, our universe of discourse comprises processes executing parts of some program along with an external world. We thus are forced to consider three kinds of actions:

Ordinary actions: Atomic actions without timing constraints are called *ordinary actions*. They may execute whenever they are enabled, or wait arbitrarily long.

Real-time actions: A *real-time action* is an atomic action whose execution time is constrained.

Idles: Execution time may pass without the program doing anything. Such a passage of time can be attributed to the external world, and we model this by an *idle*.

Ordinary actions are familiar, and the Proof Outline Logic of Section 2 works fine for them. Real-time actions cause no logical difficulties, as they have the same effects on variables and program state as ordinary actions, but their execution is more constrained. Adding axioms to Proof Outline Logic suffices for reasoning about the execution time of real-time actions. Idles, strangely enough, are more troublesome because their presence causes some matters of logical concern (viz., the time) to change without program execution. For example, Rule of Consequence (2) is unsound when idles are present.

In defining our logic, we consider an extremely powerful real-time language, allowing constructs that may be impractical or impossible to implement. Programmers using actual languages will not necessarily have access to all the features we allow. However, we believe that most actual programs in our intended domain can be translated into our language. Thus, expressive power is an advantage: the more powerful our language, the greater the number of real programs that can be expressed in it.

Our programming language is the one of Section 2 with additional real-time actions. In particular, for each unconditional atomic action[4] α, we define corresponding real-time action $\langle\alpha\rangle_{[\delta,\epsilon]}$ where δ and ϵ are real-valued, non-negative constants. Execution of $\langle\alpha\rangle_{[\delta,\epsilon]}$ causes the same indivisible state transformation as α does, but constrains it to occur at some instant between ϵ and $\epsilon + \delta$ time units after the entry control point for $\langle\alpha\rangle_{[\delta,\epsilon]}$ becomes active.

We have elected to characterize the execution time for a real-time action in terms of two parameters (δ and ϵ), following [14], in order to gain flexibility in modelling various execution environments.

[4] An atomic action is *unconditional* if it is executable whenever its entry control point becomes active. In the programming notation of Section 2, **skip**, assignment, and the guard evaluation action for **do** are unconditional. The guard evaluation for **if** is not unconditional.

Parameter ϵ describes the fixed execution time of the 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.

As an illustration, suppose we know that assignment commands take one time-unit, and that there is a single processor. If three assignment commands are started concurrently, they will be executed in some order. The first one to run will be started immediately and take one time-unit; the last one will be started two units after it was issued, and it also takes one time-unit. Thus we could model this with real-time actions having an execution time of 1 and a possible delay of 2.

$$\textbf{cobegin } \langle x1 := 1\rangle_{[2,1]} /\!/ \langle x2 := 2\rangle_{[2,1]} /\!/ \langle x3 := 3\rangle_{[2,1]} \textbf{ coend}$$

We do not allow α in a real-time action $\langle\alpha\rangle_{[\delta,\epsilon]}$ to be a conditional atomic action, such as an **if** guard evaluation action, because it is not clear what such a construct would mean. The delay in a conditional atomic action is already dependent on something else – changes to the program state. Lower bounds on conditional atomic actions (*e.g.*, an **if** command that requires at least one time-unit to evaluate its condition) can be implemented with a real-time **skip** command followed by an ordinary **if**. Timeouts on conditional atomic actions can be implemented by parallel processes and shared variables.

When writing a real-time program, it is sometimes necessary to program a loop whose iterations have fixed or bounded execution time. All of the atomic actions in such a loop must be real-time actions. We therefore introduce the following syntax to specify that $\langle \mathrm{GEval}_{\mathrm{do}}(S)\rangle_{[\delta,\epsilon]}$ be used as the guard evaluation action in a **do** command:

$$\textbf{do}_{\,[\delta,\epsilon]} B_1 \rightarrow S_1 \,[\!]\, \cdots \,[\!]\, B_n \rightarrow S_n \,\textbf{od} \tag{18}$$

3.2 Reasoning About Real-time Actions

To reason about timing properties, terms are added to the assertion language and additional information is included in the program state. This is because the method of Section 2 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, by definition, concern the instants at which control predicates become active, so we define a term $\uparrow cp$ for each control predicate cp:

$$\uparrow cp = \begin{cases} t & t \text{ is the time that } cp \text{ last became true} \\ -\infty & cp \text{ has never been true} \end{cases} \tag{19}$$

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

Only certain assignments of values to these terms are sensible in a program state. In particular, if two control points become active as part of the same event, then they must be assigned the same time. For example, since S is the first subcommand of $S\ T$, we require that $\uparrow at(S) = \uparrow at(S\ T)$. Similarly, the subcommands S_1 and S_2 in S: **cobegin** $S_1 /\!/ S_2$ **coend** start at the same time, so we require $\uparrow at(S_1) = \uparrow at(S_2) = \uparrow at(S)$.

Notice what effect adding these terms to the state has on the definition of proof outline validity. Recall that a proof outline $PO(S)$ is valid if execution starting in any state that satisfies $I_{PO(S)}$ leaves $I_{PO(S)}$ invariant. Now, a state includes a time, and so we must consider starting states with arbitrary times as well as arbitrary values for program variables and control.

Let $C[S]$ be a program containing a copy of S. An initial state of $C[S]$ not only must satisfy $at(C[S])$, but must be one in which the value of $\mathcal{T}$ is some non-negative number, and the values of $\uparrow at(T)$ and $\uparrow after(T)$ (for every command T) are initialized properly. That is, for any control predicate cp, if $at(C[S]) \Rightarrow cp$ is valid, then $\uparrow cp$ has the same value as $\mathcal{T}$; otherwise it is $-\infty$. The elements σ of $\mathcal{H}_S^{\mathcal{T}}$ are those infinite sequences of states such that:

- σ_0 is a state reached by executing some program $C[S]$ zero or more steps from an initial state, updating $\mathcal{T}$, $\uparrow at(T)$, and $\uparrow after(T)$ appropriately on each step.
- Each state σ_{i+1} is the result of performing an ordinary action, a real-time action, or an idle from σ_i. Execution of an ordinary or real-time action updates $\mathcal{T}$, $\uparrow at(T)$, and $\uparrow after(T)$ appropriately. Execution of an idle only updates $\mathcal{T}$, and in a way that does not violate the bounds δ and ϵ associated with any enabled real-time action. If no action of S is enabled, then the only transition permitted is an idle that does not increase $\mathcal{T}$.

Validity of proof outlines in our real-time logic is defined as before (Section 2.2), using $\mathcal{H}_S^{\mathcal{T}}$ in place of $\mathcal{H}_S^+$:

Real-Time Invariance:

$$\mathcal{H}_S^{\mathcal{T}} \models I_{PO(S)} \Rightarrow \Box I_{PO(S)} \tag{20}$$

Real-Time Self-Consistency: $\mathcal{H}_S^{\mathcal{T}} \models (at(S) \wedge \mathrm{pre}(S)) \Rightarrow I_{PO(S)}$

Axioms and Rules for Real Time

Execution of a real-time action $\langle \alpha \rangle_{[\delta,\epsilon]}$ affects the program variables and control predicates in the same ways as the ordinary 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 $0 \leq \epsilon$:

$$\frac{\{P\}\ \alpha\ \{Q\}}{\{P\}\ \langle \alpha \rangle_{[\delta,\epsilon]}\ \{Q\}} \tag{21}$$

Some additional axioms and inference rule allow us to reason about formulas of our more expressive assertion language. First, the various non-atomic commands of our programming language give rise to axioms based on the way they equate their components' control points. These axioms are similar to the control-predicate axioms. For our programming language, these axioms are given in Figure 6.

Next, there are the additional axioms given in Figure 7 for the assertion language. In these, cp can denote any control predicate, including those not associated with entry or exit control points for real-time actions; S is the label for a real-time action $\langle \alpha \rangle_{[\delta,\epsilon]}$. Axioms (22) and (23) follow directly from the definition of $\uparrow cp$. Axioms (24) and (25) capture the 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 the following corollary of (24):

$$(\uparrow at(S) + \delta + \epsilon < \mathcal{T}) \Rightarrow \neg at(S) \tag{26}$$

For S the sequential composition S_1S_2		
$\uparrow at(S)$	$=$	$\uparrow at(S_1)$
$\uparrow after(S)$	$=$	$\uparrow after(S_2)$
$\uparrow after(S_1)$	$=$	$\uparrow at(S_2)$
For an **if** command:		
S:	**if** $B_1 \rightarrow S_1 \,[]\, \cdots \,[]\, B_n \rightarrow S_n$ **fi**	
$\uparrow at(S)$	$=$	$\uparrow at(\mathrm{GEval}_{\mathrm{if}}(S))$
$\uparrow after(S)$	$=$	$\max(\uparrow after(S_1), \ldots, \uparrow after(S_n))$
$\uparrow after(\mathrm{GEval}_{\mathrm{if}}(S))$	$=$	$\max(\uparrow at(S_1), \ldots, \uparrow at(S_n))$
For a **do** command:		
S:	**do** $B_1 \rightarrow S_1 \,[]\, \cdots \,[]\, B_n \rightarrow S_n$ **od**	
$\uparrow at(\mathrm{GEval}_{\mathrm{do}}(S))$	$=$	$\max(\uparrow at(S), \uparrow after(S_1), \ldots, \uparrow after(S_n))$
$\uparrow after(\mathrm{GEval}_{\mathrm{do}}(S))$	$=$	$\max(\uparrow after(S), \uparrow at(S_1), \ldots, \uparrow at(S_n))$
$\neg in(S)$	$\Rightarrow$	$\uparrow after(S) = \uparrow after(\mathrm{GEval}_{\mathrm{do}}(S))$
$at(S)$	$\Rightarrow$	$\uparrow at(S) = \uparrow at(\mathrm{GEval}_{\mathrm{do}}(S))$
$in(S_i)$	$\Rightarrow$	$\uparrow at(S_i) = \uparrow after(\mathrm{GEval}_{\mathrm{do}}(S))$
$after(S)$	$\Rightarrow$	$\uparrow after(S_i) = \uparrow at(\mathrm{GEval}_{\mathrm{do}}(S))$
For a **cobegin** command:		
S:	**cobegin** $S_1 // \cdots // S_n$ **coend**	
$\uparrow at(S)$	$=$	$\uparrow at(S_1) = \uparrow at(S_2) = \cdots = \uparrow at(S_n)$
$\uparrow after(S)$	$=$	$\max(\uparrow after(S_1), \ldots, \uparrow after(S_n))$

Figure 6: Control Time Axioms

The way these new terms change value when atomic actions execute is captured by new axioms. For any ordinary or real-time atomic action α and control predicate cp, we have:

$$cp \textbf{ Invariance} \qquad \{cp = \mathrm{C} \wedge \uparrow cp = \mathrm{V}\}\ \alpha\ \{(cp \Rightarrow \mathrm{C}) \Rightarrow (\uparrow cp = \mathrm{V})\} \tag{27}$$

The antecedent in the postcondition is necessary for the case where cp could become true when α finishes, *e.g.*, $cp = after(\alpha)$.

Next, for any ordinary action, we have:

Action Time Axioms:

$$\{\mathrm{K} \leq \uparrow at(S)\} \quad S:\ \alpha \quad \{\mathrm{K} \leq \uparrow after(S)\} \tag{28}$$

$$\{\mathrm{K} \leq \mathcal{T}\} \quad S:\ \alpha \quad \{\mathrm{K} \leq \uparrow after(S)\} \tag{29}$$

Action Time Axiom (28) asserts that the exit control point for S becomes active after the entry control point for S last became active. Action-time Axiom (29) makes the subtly different assertion

$$\uparrow cp \leq \mathcal{T} \tag{22}$$

$$(\uparrow cp = -\infty) \Rightarrow \neg cp \tag{23}$$

$$at(S) \Rightarrow \uparrow at(S) \leq \mathcal{T} \leq \uparrow at(S) + \delta + \epsilon \tag{24}$$

$$\uparrow at(S) \neq -\infty \Rightarrow \uparrow after(S) \leq \uparrow at(S) + \delta + \epsilon \tag{25}$$

Figure 7: General Real-Time Axioms

that the exit control point for S becomes active after every time that the entry control point for S was active.

For a real-time action $\langle\alpha\rangle_{[\delta,\epsilon]}$, the following axiom characterizes how execution changes the $\uparrow cp$-terms.

$$\textbf{Real-time Action Axiom} \qquad \{\mathrm{K} \leq \uparrow at(S)\}\ S : \langle\alpha\rangle_{[\delta,\epsilon]}\ \{\mathrm{K} + \epsilon \leq \uparrow after(S)\} \tag{30}$$

This axiom is analogous to Action-time Axiom (28), except that 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 (30) 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; Axiom (24) serves that role. Second, one might expect the following triple to be valid—its precondition being similar to that of (29).

$$\{\mathrm{K} \leq \mathcal{T}\}\ S:\ \langle\alpha\rangle_{[\delta,\epsilon]}\ \{\mathrm{K} + \epsilon \leq \mathcal{T}\} \qquad \text{(invalid)} \tag{31}$$

Unfortunately, (31) is not sound. Execution of S started in a state such that $\uparrow at(\alpha) < \mathrm{K} \leq \mathcal{T}$ would satisfy the precondition but might terminate before $\mathrm{K} + \epsilon$. For example, consider an execution of $\langle\alpha\rangle_{[0,2]}$ that is started at time 0. Thus, at time $\mathcal{T} = 1$ the state satisfies $\mathrm{K} \leq \mathcal{T}$ if we choose $\mathrm{K} = 1$, and so precondition $\mathrm{K} \leq \mathcal{T}$ is satisfied by that state. When execution of $\langle\alpha\rangle_{[0,2]}$ terminates—2 units after it is started—at time $\mathcal{T} = 2$, the postcondition $\mathrm{K} + \epsilon \leq \mathcal{T}$ is $1 + 2 \leq 2$, which is false.

Finally, the following rule allows rigid variables to be instantiated with expressions involving $\uparrow cp$-terms. (Rigid Variable Rule (4) only allows rigid variables to be instantiated by constants, rigid variables, or expressions constructed from these.)

$$\uparrow cp\textbf{-Instantiation} \qquad \frac{\{\uparrow cp = \mathrm{V}\}\ \alpha\ \{\uparrow cp = \mathrm{V}\},\quad \{P\}\ \alpha\ \{Q\}}{\left\{P^{\mathrm{X}}_{\uparrow cp}\right\}\ \alpha\ \left\{Q^{\mathrm{X}}_{\uparrow cp}\right\}} \tag{32}$$

This rule is typically used along with cp Invariance (27). 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 due to Process Independence Axioms (16). Thus, we obtain a derived rule of inference:

$$\textbf{Derived } \uparrow cp\textbf{-Instantiation} \qquad \frac{\begin{array}{c}\alpha \text{ and } cp \text{ are in different processes,}\\ \{P\}\ \alpha\ \{Q\}\end{array}}{\left\{P^{\mathrm{X}}_{\uparrow cp}\right\}\ \alpha\ \left\{Q^{\mathrm{X}}_{\uparrow cp}\right\}} \tag{33}$$

3.3 Rule of Consequence Revisited

Most of the axioms and proof rules of Proof Outline Logic are sound in our real-time setting. However, the Rule of Consequence (2) is unsound and needs revision. We also need to revise the notion of interference freedom. While the Owicki-Gries **cobegin** rule [12] is sound, when the assertion language concerns real time, the rule is is no longer complete and, in particular, not powerful enough for even simple examples of concurrent real-time programs.

Rule of Consequence (2) is invalid in any setting where some aspect of the state is not under program control. Recall, the rule is:

$\text{MaxIdle}(\alpha)$	$=$	∞ $\quad\quad$ α ordinary
$\text{MaxIdle}(\langle\alpha\rangle_{[\delta,\epsilon]})$	$=$	$\delta+\epsilon$
$\text{MaxIdle}(S_1 S_2)$	$=$	$\text{MaxIdle}(S_1)$
$\text{MaxIdle}(\textbf{if}\cdots\textbf{fi})$	$=$	∞
$\text{MaxIdle}(\textbf{do}\cdots\textbf{od})$	$=$	∞
$\text{MaxIdle}(\textbf{do}_{[\delta,\epsilon]}\cdots\textbf{od})$	$=$	$\delta+\epsilon$
$\text{MaxIdle}(\textbf{cobegin}\, S_1 /\!/ \cdots /\!/ S_n \textbf{coend})$	$=$	$\min\{\text{MaxIdle}(S_i) \mid i=1,\ldots,n\}$

Figure 8: Definition of MaxIdle(S), the maximum idle of S

$$\frac{P' \Rightarrow P, \quad \{P\}\ PO(S)\ \{Q\}, \quad Q \Rightarrow Q'}{\{P'\}\ PO(S)\ \{Q'\}}$$

The difficulty is illustrated by the following example. Consider the following proof outline, which is valid in our model:

$$\{\mathcal{T} \geq 4\}\ S\text{:}\ \ \textbf{skip}\ \{\text{true}\} \tag{34}$$

Furthermore, note that

$$(\mathcal{T} = 4) \Rightarrow (\mathcal{T} \geq 4) \tag{35}$$

is valid. However, if we apply the Rule of Consequence to (34) and (35), we obtain the following proof outline:

$$\{\mathcal{T} = 4\}\ S\text{:}\ \ \textbf{skip}\ \{\text{true}\} \tag{36}$$

It is invalid because an idle by the environment invalidates its precondition, $\mathcal{T} = 4$. In particular, let γ be a state in which $at(S) \wedge \mathcal{T} = 4$ is true. Therefore, the precondition of (36) is satisfied by γ, and so is $I_{PO(S)}$. An idle can lead to a state γ' in which $at(S) \wedge \mathcal{T} = 4.01$, invalidating $I_{PO(S)}$. Thus the proof outline does not satisfy Real-Time Invariance (20), and hence is not valid.

We eliminate problems of this sort by modifying Rule of Consequence (2) so that idles cannot invalidate a strengthened precondition P'. In light of (36), an obvious approach is to rule out any strengthening of preconditions achieved by placing an upper bound on $\mathcal{T}$. However, that restriction would prevent us from deriving the valid triple

$$\{\uparrow at(S) = 4 \wedge 4 \leq \mathcal{T} \leq 6\}\ S\text{:}\ \ \textbf{skip}_{[0,2]}\ \{\text{true}\}. \tag{37}$$

We therefore characterize the interval over which a strengthened precondition P' must not be invalidated by an idle. For any program S, define MaxIdle(S) (maximum idle time for S) to be the longest real time interval that can elapse after $at(S)$ becomes true but before some program action of S must be executed. If S may idle arbitrarily long, then $\text{MaxIdle}(S) = \infty$. Figure 8 gives a way to calculate MaxIdle(S) by induction on the structure of S.

For example,

$$\text{MaxIdle}(\textbf{skip}) = \infty$$

as **skip** can wait arbitrarily long before taking a step, and

$$\text{MaxIdle}(\textbf{cobegin skip} /\!/ \langle\textbf{skip}\rangle_{[0,2]} \textbf{coend}) = 2$$

as that program will necessarily take a step at or before time 2. In order for Rule of Consequence (2) to be sound, not only must $P' \Rightarrow P$ hold but P' must remain true until time $\uparrow at(S) + \text{MaxIdle}(S)$.

We say that an assertion P is *patient* for S if

$$(P \wedge at(S)) \Rightarrow \left(\forall d.(\mathcal{T} \leq d \leq \uparrow at(S) + \text{MaxIdle}(S)) \Rightarrow P_d^{\mathcal{T}}\right) \tag{38}$$

Thus, if P' is patient for S, then P' can be a precondition for S and no idle by S can invalidate P'. For example, $4 \leq \mathcal{T} \leq 6$ is patient for $\langle \mathbf{skip} \rangle_{[0,2]}$, but $4 \leq \mathcal{T} \leq 5$ is not. A corollary of the way $\mathcal{H}_S^{\mathcal{T}}$ is constructed is that the precondition of any valid proof outline $PO(S)$ is patient for S.

Note that under some circumstances P' is easily demonstrated to be patient for S:

- If P' does not mention $\mathcal{T}$.
- If P' only gives lower bounds on $\mathcal{T}$.

However, even assertions involving upper bounds on $\mathcal{T}$ can be patient. For example,

$$\uparrow at(S) = 4 \ \wedge \ 4 \leq \mathcal{T} \leq 6$$

is patient for $S : \mathbf{skip}_{[0,2]}$.

A sound Rule of Consequence for our real-time logic can be formulated in terms of patient assertions:

Rule of Consequence:

$$\frac{P' \Rightarrow P, \quad P' \text{ is patient for } S, \quad \{P\}\ PO(S)\ \{Q\}, \quad Q \Rightarrow Q'}{\{P'\}\ PO(S)\ \{Q'\}} \tag{39}$$

So, because $\mathcal{T} = 4$ is not patient for **skip**, it is not possible to deduce (36) from (34) and (35) using this new Rule of Consequence. On the other hand, because

$$(\uparrow at(S) = 4 \ \wedge \ 4 \leq \mathcal{T} \leq 6) \Rightarrow \mathcal{T} \geq 4 \tag{40}$$

is valid and $\uparrow at(S) = 4 \wedge 4 \leq \mathcal{T} \leq 6$ is patient for $\mathbf{skip}_{[0,2]}$, we can use the Rule of Consequence (39) to infer

$$\{\uparrow at(S) = 4 \ \wedge \ 4 \leq \mathcal{T} \leq 6\}\ \mathbf{skip}_{[0,2]}\ \{\text{true}\} \tag{41}$$

Note that we do not need to place similar restrictions on Q'. The interpretation of $\{P\}\ PO(S)\ \{Q\}$ is that, when S finishes, Q remains true indefinitely. If $Q \Rightarrow Q'$ in predicate logic, and Q remains true forever, then Q' will also remain true forever.

The following derived rule, the Simple Rule of Consequence, handles most of the bookkeeping uses of the Rule of Consequence (39):

Simple Rule of Consequence

$$\frac{P' = P, \quad \{P\}\ PO(S)\ \{Q\}, \quad Q \Rightarrow Q'}{\{P'\}\ PO(S)\ \{Q'\}} \tag{42}$$

3.4 Interference Freedom Revisited

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

$\{x = 0\}$
cobegin
$\quad \{x = 0\} \quad \alpha\colon \ \langle x := x + 1\rangle_{[0,2]} \ \{x = 1\}$
//
$\quad \{x = 0\} \quad \beta\colon \ \langle y := x + 1\rangle_{[0,1]} \ \{y = 1\}$
coend
$\{x = 1 \wedge y = 1\}$

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

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

Using operational reasoning, however, it is not difficult to see that executing α cannot invalidate $\text{pre}(\beta)$, so $PO(\alpha)$ and $PO(\beta)$ should be considered interference free. This is because according to Figure 6, 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. Thus, it is simply not possible for α to change the value of x while $at(\beta)$ is active.

The ordinary **cobegin** Rule (15) 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 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.[5]

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

Returning to the example above, we now have:

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

And, this obligation can be discharged as follows. We present the proof in detail, to show how the axioms and rules fit together. Steps that involve standard logic (including arithmetic) are listed as simply "predicate logic."

[5]This triple is not specific to real time; see [11] for example. It arises naturally when one attempts to construct a proof rule for **cobegin**. In many cases, it can be simplified to the Owicki-Gries condition, but here it cannot.

1. Real-time Action Axiom (30)
   $\{\mathrm{K} \leq \uparrow at(\alpha)\}\ \ \alpha :\ \ \langle x := x + 1\rangle_{[0,2]}\ \ \{\mathrm{K} + 2 \leq \uparrow after(\alpha)\}$

2. Derived $\uparrow cp$-Instantiation (33) with 1, substituting $\uparrow at(\beta)$ for K
   $\{\uparrow at(\beta) \leq \uparrow at(\alpha)\}\ \ \alpha :\ \ \langle x := x + 1\rangle_{[0,2]}\ \ \{\uparrow at(\beta) + 2 \leq \uparrow after(\alpha)\}$

3. Axiom (22) with $\uparrow after(\alpha)$ for $\uparrow cp$, and predicate logic
   $(\uparrow at(\beta) + 2 \leq \uparrow after(\alpha)) \Rightarrow (\uparrow at(\beta) + 2 \leq \mathcal{T})$

4. Simple Rule of Consequence, (42) with 2 and 3
   $\{\uparrow at(\beta) \leq \uparrow at(\alpha)\}\ \ \alpha :\ \ \langle x := x + 1\rangle_{[0,2]}\ \ \{\uparrow at(\beta) + 2 \leq \mathcal{T}\}$

5. Axiom (24) and predicate logic
   $at(\beta) \Rightarrow \uparrow at(\beta) \leq \mathcal{T} \leq \uparrow at(\beta) + 1$

6. Predicate logic and arithmetic:
   $(at(\beta) \Rightarrow \uparrow at(\beta) \leq \mathcal{T} \leq \uparrow at(\beta) + 1) \Rightarrow ((\uparrow at(\beta) + 2 \leq \mathcal{T}) \Rightarrow \neg at(\beta))$

7. Modus Ponens from 5 and 6
   $(\uparrow at(\beta) + 2 \leq \mathcal{T}) \Rightarrow \neg at(\beta)$

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

9. A Control Time Axiom from Figure 6
   $\uparrow at(\alpha) = \uparrow at(\beta)$

10. Predicate logic from 9
    $\uparrow at(\beta) \leq \uparrow at(\alpha)$

11. Predicate logic from 10, since anything implies true
    $\mathrm{pre}(NI_{rt}(\alpha, \mathrm{pre}(\beta))) \Rightarrow \uparrow at(\beta) \leq \uparrow at(\alpha)$

12. Predicate logic
    $\neg at(\beta) \Rightarrow \mathrm{post}(NI_{rt}(\alpha, \mathrm{pre}(\beta)))$

13. Rule of Consequence with 8, 11, and 12; patience is vacuous
    $NI_{rt}(\alpha, \mathrm{pre}(\beta))$

Notice that information about the time after α is accumulated in steps 4 and 5, and used in step 6 to reach the same conclusion that operational reasoning gave: that, once α finishes, β has also finished.

4 Example: A Mutual Exclusion Protocol

cobegin
b : **if** $x = 0 \rightarrow c$: $\langle x := 1 \rangle_{[\delta(c),\epsilon(c)]}$ **fi**
d : $\langle \textbf{skip} \rangle_{[\delta(d),\epsilon(d)]}$
e : **if** $x = 1 \rightarrow f$: Critical Section **fi**
//
b' : **if** $x = 0 \rightarrow c'$: $\langle x := 2 \rangle_{[\delta(c'),\epsilon(c')]}$ **fi**
d' : $\langle \textbf{skip} \rangle_{[\delta(d'),\epsilon(d')]}$
e' : **if** $x = 2 \rightarrow f'$: Critical Section **fi**
coend

Figure 9: Core of Fischer's Mutex Protocol

Knowledge of execution times can be exploited to synchronize processes. A mutual exclusion protocol attributed in [10] to Mike Fischer [4] illustrates this point. The core of this protocol appears in Figure 9. There, c, d, c' and d' are real-time actions. Provided the parameters defining these real-time actions satisfy

$$\delta(c') + \epsilon(c') < \epsilon(d) \tag{43}$$

and

$$\delta(c) + \epsilon(c) < \epsilon(d') \tag{44}$$

this protocol implements mutual exclusion of the marked critical sections, as we now show.

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

A proof outline for the first process is given in Figure 10; the proof outline for the other process is symmetric, with "1" everywhere replaced by "2" and the primed labels interchanged with unprimed ones. Notice that $\text{pre}(f) \Rightarrow x = 1$ and $\text{pre}(f') \Rightarrow x = 2$. Thus, the proof outlines satisfy the conditions just outlined for ensuring that states satisfying $at(f) \wedge at(f')$ cannot occur.

It is not difficult to derive the proof outline of Figure 10 using the axiomatization of real-time actions given above. The proofs of $\{\text{pre}(c)\}\ c\ \{\text{post}(c)\}$ and $\{\text{pre}(d)\}\ d\ \{\text{post}(d)\}$ are the most enlightening, as they expose the role of assumptions (43) and (44) in the correctness of the protocol. Here is the proof of $\{\text{pre}(c)\}\ c\ \{\text{post}(c)\}$:

Let

$$\text{M} = \delta(c') + \epsilon(c') - \epsilon(d)$$

1. Axiom (29)
   $\{\text{K} \leq \mathcal{T}\}$ c: $\langle x := 1 \rangle_{[\delta(c),\epsilon(c)]}$ $\{\text{K} \leq \uparrow after(c)\}$

$\{\text{true}\}$
b: **if** $x = 0 \rightarrow \{\uparrow at(c') \leq \mathcal{T}\}$
c: $\langle x := 1 \rangle_{[\delta(c),\epsilon(c)]}$
$\{x \neq 0 \wedge (at(c') \Rightarrow \uparrow at(c') + M < \uparrow at(d))\}$
fi
$\{x \neq 0 \wedge (at(c') \Rightarrow \uparrow at(c') + M < \uparrow at(d))\}$
d: $\langle \textbf{skip} \rangle_{[\delta(d),\epsilon(d)]}$
$\{x \neq 0 \wedge \neg at(c')\}$
e: **if** $x = 1 \rightarrow \{x = 1 \wedge \neg at(c')\}$
f: Critical Section 1
$\{\text{true}\}$
fi
$\{\text{true}\}$

Figure 10: Proof Outline for Fischer's Algorithm

2. Derived $\uparrow cp$-Instantiation, (33), on step 1, to substitute $\uparrow at(c')$ for K
   $\{\uparrow at(c') \leq \mathcal{T}\}$ c: $\langle x := 1 \rangle_{[\delta(c),\epsilon(c)]}$ $\{\uparrow at(c') \leq \uparrow after(c)\}$

3. Control Time Axiom (Figure 6) for if and sequencing
   $\uparrow after(c) = \uparrow at(d)$

4. Predicate logic on step 3, and M < 0 by (43)
   $(\uparrow at(c') \leq \uparrow after(c)) \Rightarrow (\uparrow at(c') + \text{M} < \uparrow at(d))$

5. Predicate logic on step 4
   $\uparrow at(c') \leq \uparrow after(c) \Rightarrow (at(c') \Rightarrow (\uparrow at(c') + \text{M} < \uparrow at(d)))$

6. Simple Rule of Consequence (39) on steps 2 and 5
   $\{\uparrow at(c') \leq \mathcal{T}\}$ c: $\langle x := 1 \rangle_{[\delta(c),\epsilon(c)]}$ $\{at(c') \Rightarrow (\uparrow at(c') + \text{M} < \uparrow at(d))\}$

7. Assignment Axiom (8)
   $\{1 = 1\}$ c: $\langle x := 1 \rangle_{[\delta(c),\epsilon(c)]}$ $\{x = 1\}$

8. Predicate logic
   $\text{true} \Rightarrow (1 = 1)$

9. Predicate logic
   $x = 1 \Rightarrow x \neq 0$

10. Rule of Consequence, (39) on steps 7, 8, and 9, since true is patient for c.
    $\{\text{true}\}\ c\colon\ \langle x := 1\rangle_{[\delta(c),\epsilon(c)]}\ \{x \neq 0\}$

11. Conjunction Rule (5) on steps 10 and 6, plus a trivial use of the Rule of Equivalence (3).
    $\{\uparrow at(c') \leq \mathcal{T}\}\ c\colon\ \langle x := 1\rangle_{[\delta(c),\epsilon(c)]}\ \{x \neq 0 \wedge (at(c') \Rightarrow \uparrow at(c') + \mathrm{M} < \uparrow at(d))\}$

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

1. Real-time Action Axiom (30)
   $\{\mathrm{K} \leq \uparrow at(d)\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{\mathrm{K} + \epsilon(d) \leq \uparrow after(d)\}$

2. skip Axiom (7)
   $\{\mathrm{L} < \mathrm{K}\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{\mathrm{L} < \mathrm{K}\}$

3. Conjunction rule (5) on steps 1 and 2
   $\{\mathrm{L} < \mathrm{K} \wedge \mathrm{K} \leq \uparrow at(d)\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{\mathrm{L} < \mathrm{K} \wedge \mathrm{K} + \epsilon(d) \leq \uparrow after(d)\}$

4. Predicate logic
   $(\mathrm{L} < \mathrm{K} \wedge (\mathrm{K} + \epsilon(d) \leq \uparrow after(d))) \Rightarrow (\mathrm{L} + \epsilon(d) \leq \uparrow after(d))$

5. Rule of Consequence (39) on 3 and 4, noting that the precondition does not mention time and is thus patient.
   $\{\mathrm{L} < \mathrm{K} \leq \uparrow at(d)\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{\mathrm{L} + \epsilon(d) < \uparrow after(d)\}$

6. *cp* Invariance axiom, (27)
   $\{at(d) = \mathrm{C} \wedge \uparrow at(d) = \mathrm{V}\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{(at(d) \Rightarrow \mathrm{C}) \Rightarrow (\uparrow at(d) = \mathrm{V})\}$

7. Rigid Variable Rule (4) on 6, replacing C by true
   $\{at(d) = \text{true} \wedge \uparrow at(d) = \mathrm{V}\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{(at(d) \Rightarrow \text{true}) \Rightarrow (\uparrow at(d) = \mathrm{V})\}$

8. Rule of Equivalence (3) on 7
   $\{\uparrow at(d) = \mathrm{V}\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{\uparrow at(d) = \mathrm{V}\}$

9. $\uparrow cp$-Instantiation (32) using 8 and 5 to substitute $\uparrow at(d)$ for K.
   $\{\mathrm{L} < \uparrow at(d) \leq \uparrow at(d)\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{\mathrm{L} + \epsilon(d) < \uparrow after(d)\}$

10. Rule of Equivalence (3) on 9
    $\{\mathrm{L} < \uparrow at(d)\}\ d\colon\ \langle \mathbf{skip}\rangle_{[\delta(d),\epsilon(d)]}\ \{\mathrm{L} + \epsilon(d) < \uparrow after(d)\}$

11. Derived $\uparrow cp$-Instantiation (33), and Rigid Variable Rule (4), to substitute $\uparrow at(c') + \mathrm{M}$ for L in 10

    $\{\uparrow at(c') + \mathrm{M} < \uparrow at(d)\}\ d\text{:}\ \langle \mathbf{skip} \rangle_{[\delta(d),\epsilon(d)]}\ \{\uparrow at(c') + \mathrm{M} + \epsilon(d) < \uparrow after(d)\}$

12. Process Independence Axiom (16), Rigid Variable Rule (4), and Rule of Equivalence (3)
    $\{\neg at(c')\}\ d\text{:}\ \langle \mathbf{skip} \rangle_{[\delta(d),\epsilon(d)]}\ \{\neg at(c')\}$

13. Disjunction rule (6) on steps 11 and 12
    $\{\neg at(c') \vee (\uparrow at(c') + \mathrm{M} < \uparrow at(d))\}\ d\text{:}\ \langle \mathbf{skip} \rangle_{[\delta(d),\epsilon(d)]}\ \{\neg at(c') \vee (\uparrow at(c') + \mathrm{M} + \epsilon(d) < \uparrow after(d))\}$

14. Rule of Equivalence (3) on 13
    $\{at(c') \Rightarrow (\uparrow at(c') + \mathrm{M} < \uparrow at(d))\}\ d\text{:}\ \langle \mathbf{skip} \rangle_{[\delta(d),\epsilon(d)]}\ \{\neg at(c') \vee (\uparrow at(c') + \mathrm{M} + \epsilon(d) < \uparrow after(d))\}$

15. Axiom (22)
    $\uparrow after(d) \leq \mathcal{T}$

16.
    $\neg at(c') \vee \uparrow at(c') + \epsilon(c') + \delta(c') < \uparrow after(d)$
    $\Rightarrow$ Predicate Logic from step 15
    $\neg at(c') \vee (\uparrow at(c') + \epsilon(c') + \delta(c') < \mathcal{T})$
    $\Rightarrow$ Equation (26)
    $\neg at(c') \vee \neg at(c')$
    $\Rightarrow$ Predicate Logic
    $\neg at(c')$

17. Simple Rule of Consequence (42) on steps 14 and 16
    $\{at(c') \Rightarrow (\uparrow at(c') + \mathrm{M} < \uparrow at(d))\}\ d\text{:}\ \langle \mathbf{skip} \rangle_{[\delta(d),\epsilon(d)]}\ \{\neg at(c')\}$

18. skip Axiom (7)
    $\{x \neq 0\}\ d\text{:}\ \langle \mathbf{skip} \rangle_{[\delta(d),\epsilon(d)]}\ \{x \neq 0\}$

19. Conjunction Rule (5) on steps 18 and 17
    $\left\{x \neq 0 \wedge \left(at(c') \Rightarrow (\uparrow at(c') + \mathrm{M} < \uparrow at(d))\right)\right\}\ d\text{:}\ \langle \mathbf{skip} \rangle_{[\delta(d),\epsilon(d)]}\ \{x \neq 0 \wedge \neg at(c')\}$

Notice how timing information is used in step 16 to infer that a particular control point cannot be active.

5 Related Work

It is instructive to compare our logic with that of [17], another Hoare-style logic [7] for reasoning about execution of real-time programs. In [17], 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 equivalent of our Command Composition Rule (9) and the Assignment Axiom (8) would then be used to derive rules for reasoning about these augmented atomic actions.[6] In contrast, 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 cannot derive rules for real-time actions simply by using the original logic, because we do not employ assignment commands to model the passage of time.

Although our logic is more complex, by augmenting the axioms rather than the atomic actions we are led 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 [17] 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 [17] does not permit such reasoning. For example, because of the way command composition is handled in [17], the logic produces overly-conservative intervals for time bounds. This is illustrated by the following sequential program, which takes at least 10 time units to execute.

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

This fact can be proved in our logic; the logic of [17] can prove only that execution requires at least 18 time units.

A Hoare-style programming logic for reasoning about real-time is also discussed in [8]. That work is incomparable to ours. First, the programming language axiomatized in [8] is different, having synchronous message-passing and no shared variables. This is symptomatic of a fundamental difference in the two approaches. The emphasis in [8] is on the design of compositional proof systems. Shared variables could not be handled compositionally and so they are excluded from programs. In contrast, we do not require that our proof system be compositional, and we do handle shared variables.[7] Moreover, it would not be 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 set of properties handled in [8] is also incomparable to what can be proved using our logic. Our 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 [8], therefore, may only be concerned with 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 allow proofs of certain liveness properties as well as certain safety properties. Our logic cannot be used to prove any liveness properties other than those implied by the progress of time.

[6] The idea of augmenting actions with assignment commands in order to reason about the passage of time is also discussed in [5], where it is used to extend Dijkstra's *wp* [3] for reasoning about elapsed execution time. A more recent effort to augment a *wp* calculus for real time is reported in [16].

[7] The **cobegin** Rule of Proof Outline Logic (15) is not compositional because its interference-freedom test depends on the internal structure of the processes being composed.

6 Concerns

A concern when designing a logic is expressive completeness. Our timing properties include many, but not all, safety properties of interest for 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 over time. These are safety properties, but neither is a timing property (according to our definition). In general, safety properties can be partitioned into *invariance properties* and *history properties* [15]. 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 [15]. 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.

A Outline of the Soundness Proof

A.1 Scheme of the Proof

Our soundness proof has a straightforward structure. First we build a model, using structural operational semantics (SOS) [13, 18, 6]. We then show how to interpret expressions and formulae of the logic in this model. Using the model, we define the set of execution sequences $\mathcal{H}_S^{\mathcal{T}}$ used to define validity in Section 3.2. We prove a series of "sanity lemmas," showing that the intuitive definitions presented earlier match the formal definitions. Finally, we check each of the axioms and proof rules against the model.

The most subtle part of the construction is in building the model. Checking the axioms and proof rules is long but straightforward. In our model, we give a structural operational semantics for our programming language. States γ include all of the information necessary to interpret Proof Outline Logic assertions. And, the operational semantics define the relation $\gamma \hookrightarrow \gamma'$, stating that a program in state γ can perform a single atomic action, or can idle, and enter state γ'.

Using $\hookrightarrow$, we construct a linear-time temporal-logic model of a program S; that is, a set of infinite sequences Γ of states. We define a notion of "γ is a suitable initial state for S". We get an arbitrary consistent state γ_0 by running an arbitrary suitable initial state γ for S arbitrarily long, $\gamma \hookrightarrow^* \gamma_0$. $\mathcal{H}_S^{\mathcal{T}}$ is then the set of executions of S started in arbitrary consistent states γ_0.

Having defined $\mathcal{H}_S^{\mathcal{T}}$, we have have enough information to use the definition of Section 3.2 that $\models PO(S)$ when $\mathcal{H}_S^{\mathcal{T}} \models I_{PO(S)} \Rightarrow \Box I_{PO(S)}$. This puts us in a position to check the soundness of the logic, which is tedious but not difficult.

A.2 Defining the transition relation

Defining relation $\hookrightarrow$ is nontrivial. The values of control predicates, especially at the beginning and end of concurrent segments, change in fairly complicated ways. Consider the program of Figure 11. When the **cobegin** labelled a finishes, $after(c)$ or $after(d)$ will hold as well as $after(g)$ and $after(h)$; and $at(j)$, $at(l)$, and $at(m)$. A single atomic action — say, g finishing — may cause other actions at faraway points in the program to start. Or it may not.

```
a:  cobegin
    b:  if
            B1 → c:  skip
            []
            B2 → d:  skip
        fi
    //
    e:  skip
    f:  cobegin
            g:  skip
            //
            h:  skip
        coend
    coend
i:  cobegin
    j:  skip
    //
    k:  cobegin
            l:  skip
            //
            m:  skip
        coend
    coend
```

Figure 11: Control flow example

Any description of the state transition function will, at some level, involve an inductive analysis of the structure of the original program. We thus chose to define the operational semantics of processes directly, using SOS. In this style, the behavior of a composite program is defined in terms of the behavior of its subterms. Since the state γ must contain enough information to interpret all assertions in Proof Outline Logic, it must include:

- $at(l)$ and $after(l)$ for each label l
- values of program variables
- values of rigid variables
- $\uparrow at(l)$ and $\uparrow after(l)$ for each label l
- $\mathcal{T}$

All save the first can be encoded directly as components of a tuple. For example, if γ is a state, then $\gamma.\uparrow at(\cdot)$ is a function from program labels to times, and $\gamma.\sigma(\cdot)$ is a function from program variables to values. Following the usual SOS methodology, program counters in a state γ are represented by the partial program $\gamma.S$ that remains to be executed. We assume that programs are completely labelled; that is, each command (simple and composite) has a unique label, as in the example above. We also introduce a new command, **done**, indicating that a thread of computation has finished.

We now define the relation $\gamma \hookrightarrow \gamma'$ by induction on $\gamma.S$. By in large, if $\gamma \hookrightarrow \gamma'$, then γ and γ' are almost the same; *e.g.*, most variables won't change values. We therefore present the SOS rules by explaining the differences between γ and γ'.[8] For example, if $\gamma.S$ is the program l : **skip**, then the operational rule which applies to γ is:

If
  $\gamma.S = l$: **skip**
  $\gamma'.S = l$: **done**
  $\gamma'.\mathcal{T} \geq \gamma.\mathcal{T}$
  γ' is otherwise the same as γ
Then $\gamma \hookrightarrow \gamma'$

The behavior of a composite process is determined inductively by the behavior of its subprocesses. For example, **cobegin** $S_1 /\!/ \cdots /\!/ S_n$ **coend** can act if one of the S_i's can act without exceeding the time bounds of the other S_j's. This happens if there is a state γ_0 in which the program is simply the S_i that performs the the transition. Thus, the rule for **cobegin** is roughly:

If
  $\gamma.S =$ **cobegin** $S_1 /\!/ \cdots /\!/ S_i /\!/ \cdots /\!/ S_n$ **coend**,
  $\gamma.S' =$ **cobegin** $S_1 /\!/ \cdots /\!/ S_i' /\!/ \cdots /\!/ S_n$ **coend**,
  where $\exists \gamma_0, \gamma_0'$ such that:
    $\gamma_0.S = S_i$
    γ_0 is otherwise the same as γ
    $\gamma_0 \hookrightarrow \gamma_0'$
    No other component of $\gamma.S$ is required to act before $\gamma_0'.\mathcal{T}$
    $S_i' = \gamma_0'.S$
    γ' is otherwise the same as γ_0'
Then $\gamma \hookrightarrow \gamma'$

Of course, the English antecedents are made formal.

Idle actions are described by:

If
  $\gamma'.\mathcal{T} \geq \gamma.\mathcal{T}$
  $\gamma.S$ is not required to act before $\gamma'.\mathcal{T}$
  γ' is otherwise the same as γ
Then $\gamma \hookrightarrow \gamma'$

One important consequence of using a structural operational semantics is that $\gamma \hookrightarrow \gamma'$ iff there is a proof of $\gamma \hookrightarrow \gamma'$ from the operational rules. These proofs can be regarded as formal objects, and, in particular, we can do induction on the proof that $\gamma \hookrightarrow \gamma'$. Many of the basic lemmas used for soundness proceed by such inductions.

This discussion omits subtleties that are essential to the proof. For the model construction, the actual proof rules assign responsibility for the transition, so that when we define $\mathcal{H}_S^{\mathcal{T}}$ we can ensure that S takes all the transitions in $\mathcal{H}_S^{\mathcal{T}}$. Roughly, a subterm S' of the program $\gamma.S$ is *responsible* for the transition $\gamma \hookrightarrow \gamma'$ if S' appears as $\gamma_0.S$ in the proof of $\gamma \hookrightarrow \gamma'$ or if the transition is idle. The determination of which processes are finished requires some care. For example, **done** is a finished atomic process, but there are also others, such as **cobegin done//done coend**. While the intuition is reasonably straightforward, the details are delicate.

[8] In the full proof, we use a more standard SOS notation, which makes the structural induction clearer but requires extra notation.

A.3 Interpreting Expressions and Formulae

The meanings of most expressions and formulae can be read directly from the state γ. For example, the value of the expression $x + y$ in state γ is the sum of $\gamma.\sigma(x)$ and $\gamma.\sigma(y)$. However, the values of control predicates $at(l)$ and $after(l)$ depend on the control state $\gamma.S$ of the program. In this section, we sketch the interpretations of these predicates for the case when l is the label of an atomic action.

Let γ be a state in an element of $\mathcal{H}_S^{\mathcal{T}}$. Interpreting $after(l)$ where l is the label of an atomic action is straightforward; $\gamma.S$ includes a subterm of the form l : **done** if and only if the atomic action labelled l in $\gamma.S$ is finished.

We define the set of *active atomic actions* of a program S inductively; *e.g.*, if S is atomic, $\mathbf{act}(S) = \{S\}$, and *e.g.*

$$\begin{array}{lcl} \mathbf{act}(\mathbf{cobegin} S_1 /\!/ \cdots /\!/ S_n \mathbf{coend}) & = & \bigcup_i \mathbf{act}(S_i) \\ \text{if } S_1 \text{ is not finished, } \mathbf{act}(S_1 S_2) & = & \mathbf{act}(S_1) \\ \text{if } S_1 \text{ is finished, } \mathbf{act}(S_1 S_2) & = & \mathbf{act}(S_2) \end{array}$$

Then $\gamma \models at(l)$ if l is the label of an atomic action in $\mathbf{act}(\gamma.S)$. In the actual proof, we allow $at(l)$ when l is the label of any program, not just the label of an atomic action. This complicates the definition of **act** somewhat and requires use of an extra set of markers in the operational semantics. However, it allows us to verify the Proof Outline Logic control predicate axioms without having built them directly into the definition of $\gamma \models at(l)$.

A.4 Sanity Checking

We demonstrate that the operational semantics and notion of interpreting processes are reasonable by proving a series of *sanity lemmas*. These are lemmas that are not necessarily used in the soundness proof proper but show that the formal definitions derived from the operational semantics agree with the less formal ones used in the body of this paper. The following sanity lemma, for example, shows that the intuitive definition of $\uparrow at(l)$ as the last time that $at(l)$ became true agrees with the formal definition of $\uparrow at(l)$ as it appears in the operational semantics:

> Let γ_0 be a suitable initial state, and $\gamma_0 \hookrightarrow \gamma_1 \hookrightarrow \cdots$. Then, for any index i and label l, $\gamma_i.\uparrow at(l)$ is:
>
> - $-\infty$ if $(\forall 0 \leq j \leq i : \gamma_j \not\models at(l))$
> - $\gamma_j.\mathcal{T}$ if j is the largest $0 \leq j \leq i$ such that $(\gamma_{j-1} \not\models at(l)$ and $\gamma_j \models at(l))$.
> - $\gamma_0.\mathcal{T}$, if neither of the preceding conditions obtains; that is, if $\gamma_0 \models at(l)$ but $(\nexists j : \gamma_j \not\models at(l) \wedge \gamma_{j+1} \models at(l))$.

That is, the time that the bookkeeping mechanism gives for $\uparrow at(l)$ is in fact the value of the clock on the most recent instant that $at(l)$ became true.

A.5 Soundness

Once the SOS rules have been constructed and their sanity checked, it is a routine matter to show soundness of all the Real-Time Proof Outline Logic axioms and proof rules. We define the model $\mathcal{H}_S^{\mathcal{T}}$ in the following way. We consider executions starting with S in an arbitrary state of control and memory. We allow other processes in this initial state as well. We thus consider executions that start with some program that includes S; we run this program for a time (which gives an arbitrary state, perhaps with S partially executed). However, in the sequences in $\mathcal{H}_S^{\mathcal{T}}$, only S is allowed to take steps, so S must be responsible for each transition.

- γ is an *initial state* for S iff $\gamma.S$ includes S as a subprogram, and $\gamma.\uparrow at(l)$ and $\gamma.\uparrow after(l)$ are initialized properly. That is, $\gamma.\uparrow cp = \gamma.\mathcal{T}$ if $\gamma.S \models cp$, and to $-\infty$ otherwise, for cp a control predicate.

- $\mathcal{H}_S^{\mathcal{T}}$ is the set of all sequences $\langle \gamma_0, \gamma_1, \ldots \rangle$ such that $\gamma \hookrightarrow^* \gamma_0$ for some initial state γ, and $\gamma_i \hookrightarrow \gamma_{i+1}$ for every i, and S is responsible for each transition.

Note that $\mathcal{H}_S^{\mathcal{T}}$ is suffix-closed; that is, if $\langle \gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots \rangle \in \mathcal{H}_S^{\mathcal{T}}$, then also $\langle \gamma_i, \gamma_{i+1}, \ldots \rangle \in \mathcal{H}_S^{\mathcal{T}}$.

To prove the Control Predicate Axioms of Figure 1 sound, we characterize possible control states of a program and possible executions of a program. For example, omitting labels, **done** S is a possible subterm of a control state (as **skip** S can evolve into it); but S **done** is not possible (the first in a sequence of commands is executed before the second).

Similarly, we say that program fragment S' is a descendant of S if (roughly) there are states $\gamma \hookrightarrow^* \gamma'$ such that $\gamma.S = S$ and $\gamma'.S = S'$; it is a proper descendant if $S \neq S'$. We characterize the descendants of all terms. For example, let S be a command sequence $l : ((m : S_1)(n : S_2))$, and S' be a descendant of S. Then there is at most one subterm of S' labelled l, and it must be:

- $l : ((m : S_1')(n : S_2))$ where S_1' is a descendant of S_1, or

- $l : (n : S_2')$ where S_2' is a proper descendant of S_2.

Soundness of the Control Predicate Axioms follow easily from these characterizations. For example, for one direction of the axiom for sequencing, $after(m) = at(n)$, we calculate that if $\gamma \models after(m)$, then there must be a subterm m : **done** of $\gamma.S$. From the preceding characterization, this means that $n : S_2$ must also be a subterm of $\gamma.S$; and, by the definition of $at(\cdot)$, this implies that $at(n)$ holds as well.

The other axioms and proof rules are proved similarly. For example, to show (22), we show by induction on the proof of $\gamma \hookrightarrow \gamma'$ that

$$(\gamma \hookrightarrow \gamma') \Rightarrow (\gamma.\mathcal{T} \leq \gamma'.\mathcal{T}) \tag{45}$$

Suppose that $\gamma_0 \hookrightarrow \gamma_1 \hookrightarrow \cdots$ is a sequence of transitions from an initial state. By induction on i and the definition of initial state, we show that $\gamma_i.\uparrow at(l)$ and $\gamma_i.\uparrow after(l)$ are either $-\infty$, or $\gamma_j.\mathcal{T}$ for some $j \leq i$. This and (45) suffices to show (22).

Axioms and rules involving proof outlines require verifying statements of the form $\mathcal{H}_S^{\mathcal{T}} \models I \Rightarrow \Box I$. From temporal logic, we know that, if $I \Rightarrow \bigcirc I$ is valid, so is $I \Rightarrow \Box I$. I is a predicate logic formula rather than a temporal one, hence it is true or false in a single state. It thus suffices to show that, for each $\gamma \hookrightarrow^* \gamma_0 \hookrightarrow \gamma_1$ where γ is an initial state for S, if $\gamma_0 \models I$, then $\gamma_1 \models I$.

We use this method to verify each proof outline axiom and proof rule. All these verifications proceed by induction on the proof of the transition $\gamma_0 \hookrightarrow \gamma_1$. For example, to check **skip** Axiom (7), let $S = l$: **skip**. $I_{PO(S)} = (at(l) \Rightarrow P) \wedge (after(l) \Rightarrow P)$, where P is primitive. Suppose $\gamma_0 \models I_{PO(S)}$. It is easy to show that $\gamma \models P$ is independent of $\gamma.S$ and $\gamma.\mathcal{T}$ if P is primitive. The proof comprises the following cases:

1. The transition is idle, and $\gamma_0 \models at(l)$. As $\gamma_0 \models I_{PO(S)}$, we conclude $\gamma_0 \models P$. In this case, the only component of γ_1 that is different from γ_0 is $\gamma_1.\mathcal{T}$. Since the value of a primitive formula does not depend on $\gamma.\mathcal{T}$, and since $\gamma_0 \models P$, we conclude $\gamma_1 \models P$. This suffices to show $\gamma_1 \models I_{PO(S)}$.

2. The transition is idle, and $\gamma_0 \models after(l)$. The proof proceeds as above.

3. The transition is not idle, in which case it proceeds by SOS rule for **skip** above. That is, $\gamma_0.S = l$: **skip**, $\gamma_1.S = l$: **done**, and γ_0 and γ_1 are otherwise identical except possibly for $\gamma_i.\mathcal{T}$. Note that $\gamma_0 \models P$, because $\gamma_0 \models at(l)$ and $\gamma_0 \models I_{PO(S)}$. Primitive formulas do not depend on the changed components. Hence $\gamma_1 \models P$ and, thus, $\gamma_1 \models I_{PO(S)}$ as desired.

4. $\gamma_0 \models \neg(at(l) \wedge after(l))$. In this case, S cannot be responsible for any non-idle transitions, and all transitions are thus idle. In particular, $\gamma_1 \models \neg(at(l) \wedge after(l))$, and hence $\gamma_1 \models I_{PO(S)}$ vacuously.

Each of the other axioms and rules of Real-Time Proof Outline Logic is handled in a similar manner, and thus we establish soundness. The subtle part of the proof is not checking these rules, so those details are omitted here. The subtle part is the definition of the real-time execution model that is explained above.

Acknowledgements

We are grateful to Limor Fix for extensive comments, and Georges Lauri for preliminary work on the soundness proof.

References

[1] K. Apt and G. Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, 1986.

[2] B. Bloom and F. B. Schneider. Soundness of a real-time proof outline logic. In preparation.

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

[4] M. Fischer. **Re: Where are you?** Arpanet [electronic mail system], June 1985. Message to: Leslie Lamport. Message No.: 8506252257.AA07636@YALE-BULLDOG.YALE.ARPA.

[5] V. Haase. Specification and compositional verification of real-time systems. *IEEE Transactions on software engineering*, 12(10):494–501, Oct 1981.

[6] M. Hennessy. *The Semantics of Programming Languages: An elementary introduction using structured operational semantics.* Wiley, 1990.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[8] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* PhD thesis, Eindhoven University of Technology, May 1991.

[9] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 3(2):125–143, 1977.

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

[11] L. Lamport and F. B. Schneider. The "Hoare logic" of CSP and all that. *ACM Trans. on Programming Languages and Systems*, 6(2):281–296, 1984.

[12] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1981.

[13] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, Denmark, 1981.

[14] A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems (extended abstract). In M. Joseph, editor, *Formal Techniques in Real-time and Fault-tolerant Systems*, pages 84–98. Springer-Verlag, New York, 1988. LNCS Volume 331.

[15] F. B. Schneider. On concurrent programming. (in progress), Jan 1993.

[16] D. Scholefield and H. S. M. Zedan. Weakest precondition semantics for time and concurrency. *Information Processing Letters*, 42:301–308, 1992.

[17] A. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. on Software Engineering*, 7:875–889, 1989.

[18] S. Weber, B. Bloom, and G. Brown. Compiling Joy to silicon: A verified silicon compilation scheme. In T. Knight and J. Savage, editors, *Proceedings of the Advanced Research in VLSI and Parallel Systems Conference*, pages 79–98. 1992.