Notes on Proof Outline Logic

Fred B. Schneider

Department of Computer Science, Cornell University, Ithaca, NY, 14853, U.S.A.

Abstract. Formulas of Proof Outline Logic are program texts annotated with assertions. Assertions may contain control predicates as well as terms whose values depend on previous states, making the assertion language rather expressive. The logic is complete for proving safety properties of concurrent programs. A deductive system for the logic is presented. Solutions to the mutual exclusion and readers/writers problems illustrate how the logic can be used as a tool for program development.

Keywords. Program verification, assertional reasoning, safety properties.

1. Introduction

Proof Outline Logic is a generalization of Hoare's 1969 logic for proving partial correctness of sequential programs. Generalizing from partial correctness to arbitrary safety properties requires that control state and values of variables in past states be expressible in assertions, dramatically affecting the assertion language. Generalizing from sequential programs to concurrent ones forces formulas to associate an assertion with every control point, rather than just associating assertions with the entry and exit points of the entire program as in Hoare's logic.

Like most other programming logics, Proof Outline Logic allows one to prove formally that a program satisfies a specification. In Proof Outline Logic, this is done by establishing a link between two languages: programs specified in a programming language are shown to satisfy safety properties specified in a linear-time Temporal Logic. We employ a specification language different from proof outlines to avoid having the specification bias the structure of an implementation. Had we required that specifications be given as proof outlines, the specifier would have to postulate some program structure. Of course, one is not precluded from specifying a property by giving a proof outline.

Proof Outlines link specifications and programs, because the meaning of a proof outline is formalized as a Temporal Logic formula and the meaning of a program is formalized as a set of Temporal Logic interpretations. One consequence of defining the one logic in terms of the other is that not only must Proof Outline Logic stand on its own, but it must also make sense in the context of a Temporal Logic. For example, the language of Temporal Logic must be an extension of the assertion language for proof outlines.

A goal of our work has been to deal with realistic programming language constructs. In so far as our interest is concurrent programs, this meant axiomatizing a programming language that was expressive enough to describe the various synchronization and communications structures that one finds in real programs. Guard evaluation in **if** and **do** statements, for example, define atomic actions in our programming language. Our reasoning apparatus supports this, even though such guard evaluation actions are not programming language statements per se.

2. Programs and Properties

Execution of a program S defines a set $\mathcal{H}_S$ of potentially infinite *histories*

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_i} s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \xrightarrow{\alpha_{i+2}} \cdots$$

where the s_i's denote program states, the α_i's denote atomic actions, and execution of each α_{i+1} in state s_i can terminate in state s_{i+1}. For a concurrent program, sequence $\alpha_1 \alpha_2 \ldots$ is the result of interleaving atomic actions from each of the processes in the order these actions were executed. Finite histories correspond to terminating executions; the final state of a finite history must be one in which no atomic action can execute. Note that s_0 need not be an initial state of the program.

We represent both full and partial executions as *anchored sequences*—pairs (σ, j) where σ is the finite or infinite sequence of states corresponding to a history and j is a non-negative integer satisfying $j < |\sigma|$. For $\sigma = s_0 s_1 \ldots$, we write $\sigma[..i]$ to denote prefix $s_0 s_1 \ldots s_i$, $\sigma[i]$ to denote state s_i, and $\sigma[i..]$ to denote suffix $s_i s_{i+1} \ldots$. Parameter j in (σ, j) partitions σ into

- a (possibly empty) sequence $\sigma[..j-1]$ of *past* states,
- a *current* state $\sigma[j]$, and
- a (possibly empty) sequence $\sigma[j+1..]$ of *future* states.

To the extent possible, we wish to reason compositionally. Doing so is facilitated by reasoning about executions that start in the middle of a program as well as executions that start from an initial state.

(2.1) **Program-Execution Interpretations.** Let $\mathcal{S}^\infty$ denote the set of all non-empty finite and infinite sequences of program states for S. Set $\ddot{\mathcal{H}}_S$ contains anchored sequences (σ, j) where $\sigma[j..]$ is an element in $\mathcal{H}_S$.

$$\ddot{\mathcal{H}}_S: \quad \{(\sigma, j) \mid \sigma \in \mathcal{S}^\infty \ \wedge \ \sigma[j..] \in \mathcal{H}_S\} \qquad \square$$

By including in $\ddot{\mathcal{H}}_S$ those anchored sequences (σ, j) where $\sigma[..j-1]$ is an arbitrary sequence of program states and $\sigma[j..]$ is a history of S, we remove the distinction between S comprising an entire program and S serving as a component of a program. Arbitrary prefix $\sigma[..j-1]$ models an unspecified execution that precedes execution of S.

Our specification language—Temporal Logic—is interpreted with respect to anchored sequences. For every anchored sequence (σ, j) and every Temporal Logic formula P, either (σ, j) is a model for P, denoted $(\sigma, j) \models P$, or it is not. We write $\ddot{\mathcal{H}}_S \models P$ iff every element of $\ddot{\mathcal{H}}_S$ is a model for P or, equivalently, $\ddot{\mathcal{H}}_S$ is a subset of the models for P.

For our purposes, it suffices to restrict consideration to two classes of Temporal Logic formulas: P and $\Box P$, where P is a Predicate Logic formula. When P an formula of ordinary Predicate Logic, $(\sigma, j) \models P$ holds iff P is satisfied in state $\sigma[j]$. This is consistent with identifying $\sigma[j]$ as the current state of (σ, j). We define $(\sigma, j) \models \Box P$ in terms of the suffixes of (σ, j):

$$(\sigma, j) \models \Box P \quad \text{iff} \quad \text{For all } i,\ j \le i < |\sigma|:\ (\sigma, i) \models P$$

Executions and properties are sets of anchored sequences—not simply sets of state sequences. This is unconventional, but has advantages when the language for writing specifications is sufficiently expressive. $Init_S \Rightarrow P$ asserts that P need hold only for anchored sequences $(\sigma, 0)$, where $\sigma[0]$ is an initial state, if the specification language includes a formula $Init_S$ that is satisfied only at the start of executing program S. Thus, by proving $\ddot{\mathcal{H}}_S \models (Init_S \Rightarrow P)$, we can establish that only those executions of S starting from an initial state need satisfy P. Moreover, by proving $\ddot{\mathcal{H}}_S \models P$, we can establish that all executions—including those that start in the middle of the program—satisfy P. Reasoning about executions that start in the middle of a program is particularly useful when considering concurrent programs.

3. A Programming Language

A *program* consists of declarations followed by statements. The *declarations* introduce program variables and associate a type with each. The *statements* define sets of atomic actions. Consequently, a program defines a set of program states and a set of atomic actions. Each *program state* assigns a value of the correct type to every program variable and contains control information to indicate which atomic actions might next be executed.

The syntax of a declaration is:

$$\textbf{var}\ \overline{id}_1 : type_1;\ \ \overline{id}_2 : type_2;\ \ \cdots\ \ \overline{id}_n : type_n$$

Each $\overline{id}_i$ is a list of distinct identifiers, separated by commas. Each $type_i$ gives a type for the variables in $\overline{id}_i$. This type can be Bool, Nat, Int, or Real or it can be an enumeration, set, array, or record, specified in the usual way.

3.1. Statements

Executing a statement results in execution of a sequence of atomic actions, each of which indivisibly transforms the program state. Therefore, we define the semantics of a statement S by giving its atomic actions $\mathcal{A}(S)$ and the effect of each.

The **skip** statement is a single atomic action whose execution has no effect on any program variable. Its syntax is:

(3.1) **skip**

The assignment is also a single atomic action. Execution of

(3.2) $x_1, x_2, ..., x_n := e_1, e_2, ..., e_n$

where $x_1, x_2, ..., x_n$ are called *targets* of the assignment, first computes values for all expressions appearing in the statement (including those in the targets, as in $x[e]$). If (i) any of the x_i is undefined (e.g. x_i is an array reference $x[e]$ and the value of e is outside the range of permissible subscripts) or (ii) the value computed for some expression e_i is not consistent with the type of corresponding target x_i, then execution of (3.2) is blocked. Otherwise execution proceeds by setting x_1 to the value computed for e_1, then setting x_2 to the value computed for e_2, and so on.

We assume that expressions are defined in all states, although the value of a given expression might be unspecified in some of those states. Thus, execution of $x := y/z$ will assign some value to x even if started in a state in which $z=0$ holds provided the (unspecified) value of y/z is consistent with the type of x.

Statement juxtaposition combines two statements S_1 and S_2 into a new one:

(3.3) $S_1 \ S_2$

The atomic actions of (3.3) are just the atomic actions of S_1 and S_2. Execution is performed by executing S_1 and, when (and if) it terminates, executing S_2.

The syntax of an **if** statement S is:

(3.4) S: **if** $B_1 \rightarrow S_1$ [] $B_2 \rightarrow S_2$ [] $\cdots$ [] $B_n \rightarrow S_n$ **fi**

Each $B_i \rightarrow S_i$ is called a *guarded command*. The *guard* B_i is a boolean-valued expression, and S_i is a statement. The atomic actions of **if** statement S consist of the atomic actions of S_1 through S_n and an additional *guard evaluation action*, $GEval_{if}(S)$, which selects one of S_1 through S_n for execution. Execution of (3.4) proceeds as follows. First, $GEval_{if}(S)$ is executed. This blocks until at least one of guards B_1 through B_n holds and then selects some guarded command $B_i \rightarrow S_i$ for which guard B_i holds. Next, corresponding statement S_i is executed.

The **do** statement

(3.5) S: **do** $B_1 \rightarrow S_1$ [] $B_2 \rightarrow S_2$ [] $\cdots$ [] $B_n \rightarrow S_n$ **od**

is used to specify iteration. Its atomic actions are the atomic actions of S_1 through S_n plus a guard evaluation action $GEval_{do}(S)$. Execution of (3.5) consists of repeating the following until no *true* guard is found: use $GEval_{do}(S)$ to select a guarded command $B_i \rightarrow S_i$ where B_i is *true*; then, execute S_i.

The **cobegin** statement

(3.6) S: **cobegin** $S_1 \parallel S_2 \parallel \cdots \parallel S_n$ **coend**

specifies concurrent execution of processes S_1, ..., S_n. Its atomic actions are the atomic actions of S_1 through S_n. Execution of S results in interleaving the atomic actions of its processes and terminates when all of these processes have terminated.

Placing angle brackets around a statement S defines an *atomic statement*, which is executed indivisibly as a single atomic action. Thus, $\langle S \rangle$ defines a statement whose execution is blocked unless the state satisfies $enbl(S)$, where

(3.7) $enbl(\alpha)$: $wp(\alpha, true)$.

Because $enbl(S)$ can, in general, differ from $enbl(\alpha)$ for α the first atomic action of S, the angle-bracket notation allows *condition synchronization* to be specified.

An atomic action α is defined to be *unconditional* in a program S if and only if $enbl(\alpha)$ holds in all program states; otherwise, α is *conditional* in S. Thus, a **skip** is unconditional but the guard evaluation for an **if** can be conditional[1].

Allowing arbitrary programs to appear inside angle brackets can pose implementation problems. However, if atomic statements are used only to describe synchronization mechanisms that already exist, such implementation problems need never be confronted. The question of what synchronization mechanisms are available depends on hardware and underlying support software.

Statement Labels

A label L is associated with a statement by prefixing that statement with L followed by a colon. We use indentation and sometimes a brace to indicate when a label is associated with the statement that results from a juxtaposition of two or more statements. For example, in the program of Fig. 3.1, indentation is used to indicate that S_3 labels the statement juxtaposition formed from the **if** labeled S_4 and the assignment labeled S_7.

We assume that every statement in a program has a unique label. This said, Fig. 3.1 illustrates how including such labels can result in a program texts that are cluttered and difficult to read. Therefore, wherever possible, we avoid explicitly giving statement labels. For example, when no ambiguity results, we use the text of a statement as a label for that statement.

---

[1]If the disjunction of the guards in an **if** is satisfied in all program states, then the guard evaluation action for that **if** is unconditional.

$$
\begin{array}{l}
\textbf{var } i : \text{Int};\quad m : \text{Real};\quad a : \textbf{array } [0 \,..\, n] \textbf{ of } \text{Real} \\
S_1\text{: } i, m := 0, a[0] \\
S_2\text{: } \textbf{do } i \neq n \rightarrow \; S_3\text{: } S_4\text{: } \textbf{if } a[i+1] \leq m \rightarrow \; S_5\text{: } \textbf{skip} \\
\qquad\qquad\qquad\qquad\qquad\quad [\,]\ a[i+1] > m \rightarrow \; S_6\text{: } m := a[i+1] \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{fi} \\
\qquad\qquad\qquad\qquad\quad S_7\text{: } i := i+1 \\
\quad \textbf{od}
\end{array}
$$

Fig. 3.1. Maximum Element of an Array

4. Predicate Logic

We extend ordinary first-order predicate logic so that it specifies sets of program states and sets of past state sequences. To characterize program states, we add axioms to the logic. These axioms restrict what values can be associated with variables and what values program counters can take. To characterize past state sequences, we add to the logic special terms and predicates that allow us to construct Predicate Logic formulas P for which $(\sigma, j) \models P$ depends on sequence $\sigma[..j-1]$ of past states as well as current state $\sigma[j]$.

4.1. Axioms for Program Variables

The declarations in a program S give rise to a set $VarAx(S)$ of Predicate Logic axioms called *program variable axioms*. These axioms rule out states in which variables have values that are not type-correct. Thus, the axioms characterize which values program states can associate with variables. For example, the declarations in the program of Fig. 3.1 imply that the following holds for all program states.

(4.1) $\quad i \in \text{Int} \;\wedge\; m \in \text{Real} \;\wedge\; (e \in \text{Int} \wedge 0 \leq e \leq n \Rightarrow a[e] \in \text{Real})$

Given an arbitrary program S, we construct the set $VarAx(S)$ of program variable axioms as follows.

(4.2) **Program Variable Axioms.** $VarAx(S)$ is the union of $ValAx(v, t)$ for every program variable v declared in S, where t is its type. $ValAx(v, t)$ is defined in Fig. 4.1. □

The origin of (4.1) should now be clear—each conjunct is a program variable axiom. We obtain $i \in \text{Int}$ from the declaration that i is of type Int, $m \in \text{Real}$ from the declaration that m is of type Real, and $e \in \text{Int} \wedge 0 \leq e \leq n \Rightarrow a[e] \in \text{Real}$ from the declaration that a is of type **array** $[0 \,..\, n]$ **of** Real.

type	$ValAx(v, type)$
Bool, Nat, Int, Real	$v \in type$
enum($C_1, C_2, ..., C_n$)	$v \in \{C_1, C_2, ..., C_n\}$
set of *type*	$v \subseteq type$
array $[a_1 .. b_1,$ $a_2 .. b_2,$ $\cdots$ $a_n .. b_n]$ **of** *type*	$(e_1 \in \mathrm{Int} \wedge a_1 \le e_1 \le b_1 \wedge$ $e_2 \in \mathrm{Int} \wedge a_2 \le e_2 \le b_2 \wedge$ $\cdots$ $e_n \in \mathrm{Int} \wedge a_n \le e_n \le b_n)$ $\Rightarrow ValAx(v[e_1, e_2, ..., e_n], type)$
record($id_1 : type_1;$ $id_2 : type_2;$ $\cdots$ $id_n : type_n$)	$ValAx(v.id_1, type_1),$ $ValAx(v.id_2, type_2),$ $\cdots$ $ValAx(v.id_n, type_n)$

Fig. 4.1. Definition of $ValAx(v, t)$

4.2. Control Predicates

The *control points* of a program are defined by its 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; a guard evaluation atomic action $GEval_{if}(S)$ has one entry control point and multiple exit control points—one for each guarded command.

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. The program state usually encodes which control points are active by representing this information in (implicit) variables, called *program counters*, each of which ranges over some subset of the control points.

Since a statement S defines a set $\mathcal{A}(S)$ of atomic actions, each statement also defines a set of control points. In specifying and proving properties of programs, it is useful to be able to assert that one or another control point is active. To facilitate this, we define a nullary predicate, called a *control predicate*, for each S an atomic action or statement:

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

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

In addition, it will sometimes be convenient to assert that an entry control point for an atomic action in $\mathcal{A}(S)$ is active. The following control predicate permits this, where $Parts(S)$ is a set consisting of label S and the label of any component of S.

$in(S)$: $at(T)$ holds for some $T \in Parts(S)$.

For our programming language, $Parts(S)$ is defined based on the structure of S:

(4.3) **Statement Decomposition.** $Parts(S)$ is defined by:

For S a **skip**, an assignment, a guard evaluation action, or an atomic statement

$$Parts(S) = \{S\}.$$

For S: $S_1\ S_2$,

$$Parts(S) = \{S\} \cup Parts(S_1) \cup Parts(S_2).$$

For S: **if** $B_1 \rightarrow S_1$ [] $\cdots$ [] $B_n \rightarrow S_n$ **fi**,

$$Parts(S) = \{S, GEval_{if}(S)\} \cup \bigcup_{1 \le i \le n} Parts(S_i).$$

For S: **do** $B_1 \rightarrow S_1$ [] $\cdots$ [] $B_n \rightarrow S_n$ **od**,

$$Parts(S) = \{S, GEval_{do}(S)\} \cup \bigcup_{1 \le i \le n} Parts(S_i).$$

For S: **cobegin** S_1 || $\cdots$ || S_n **coend**

$$Parts(S) = \{S\} \cup \bigcup_{1 \le i \le n} Parts(S_i).$$

□

In order to reason about formulas containing control predicates, we introduce *control predicate axioms*. These 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. The axioms also characterize the entry and exit control points for each S by defining $at(S)$ and $after(S)$. Operator $\oplus$ (with the same precedence as $\vee$) is used to denote n-way exclusive-or, so that $P_1 \oplus P_2 \oplus \cdots \oplus P_n$ is a predicate that is *true* when exactly one of P_1 through P_n is.

Four axioms are a direct consequence of how $in(S)$ and $Parts(S)$ are defined:

(4.4) *In Axioms*: (a) $at(S) \Rightarrow in(S)$

(b) For $T \in Parts(S)$: $in(T) \Rightarrow in(S)$

(c) For $T \in Parts(S)$: $after(T) \Rightarrow (after(S) \vee in(S))$

(d) For S a single atomic action: $at(S) = in(S)$

The next axiom asserts that an exit control point for T cannot be active at the same time as an entry control point for T or for any of its components.

(4.5) *Entry/Exit Axiom*: $\neg\,(in(T) \wedge after(T))$

Since all reasoning is with respect to what happens during execution of some program S, every state must satisfy one of the following: (i) S has not yet started, (ii) S has started but not yet terminated, or (iii) S has terminated. This allows us to conclude:

(4.6) *Program Control*: For S the entire program: $in(S) \oplus after(S)$

The control predicate axioms for a statements are based on control flow.

(4.7) *Statement Juxtaposition Control Axioms*: For S the juxtaposition $S_1\ S_2$:
(a) $at(S) = at(S_1)$
(b) $after(S) = after(S_2)$
(c) $after(S_1) = at(S_2)$
(d) $in(S) \ = \ (in(S_1) \vee in(S_2))$
(e) $(in(S) \vee after(S)) \ \Rightarrow \ (in(S_1) \oplus in(S_2) \oplus after(S))$

(4.8) **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) $after(S) \ = \ (after(S_1) \vee after(S_2) \vee ... \vee after(S_n))$
(c) $after(GEval_{if}(S)) \ = \ (at(S_1) \vee at(S_2) \vee ... \vee at(S_n))$
(d) $in(S) \ = \ (in(GEval_{if}(S)) \vee in(S_1) \vee in(S_2) \vee ... \vee in(S_n))$
(e) $(in(S) \vee after(S)) \ \Rightarrow \ (in(GEval_{if}(S)) \oplus in(S_1) \oplus in(S_2) \oplus ... \oplus in(S_n)$
$\oplus\ after(S_1) \oplus after(S_2) \oplus ... \oplus after(S_n))$

(4.9) **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) $at(GEval_{do}(S)) \ = \ (at(S) \vee after(S_1) \vee after(S_2) \vee ... \vee after(S_n))$
(b) $at(GEval_{do}(S)) \ \Rightarrow \ (at(S) \oplus after(S_1) \oplus after(S_2) \oplus ... \oplus after(S_n))$
(c) $after(GEval_{do}(S)) \ = \ (after(S) \vee at(S_1) \vee at(S_2) \vee ... \vee at(S_n))$
(d) $after(GEval_{do}(S)) \ \Rightarrow \ (after(S) \oplus at(S_1) \oplus at(S_2) \oplus ... \oplus at(S_n))$
(e) $in(S) \ = \ (in(GEval_{do}(S)) \vee in(S_1) \vee ... \vee in(S_n))$
(f) $(in(S) \vee after(S)) \ \Rightarrow \ (in(GEval_{if}(S)) \oplus in(S_1) \oplus in(S_2) \oplus ... \oplus in(S_n)$
$\oplus\ after(S))$

(4.10) **cobegin** *Control Axioms*: For a **cobegin** statement:

$$S\colon \textbf{cobegin } S_1 \ \| \ S_2 \ \| \ \cdots \ \| \ S_n \textbf{ coend}$$

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

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

(c) $in(S) = ((in(S_1) \vee after(S_1)) \wedge \ldots \wedge (in(S_n) \vee after(S_n)) \wedge \neg(after(S_1) \wedge \ldots \wedge after(S_n)))$

(4.11) $\langle S \rangle$ *Control Axioms*: For an atomic statement:

$$S\colon \langle T \rangle$$

(a) $at(S) = at(T)$

(b) $in(S) = at(S)$

(c) $after(S) = after(T)$

4.3. Past and Derived Terms

Proof Outline Logic is intended for proving safety properties. A safety property proscribes some "bad thing". Such a "bad thing" might be any state in some set. For example, $\neg(in(CS_1) \wedge in(CS_2))$ specifies program states in which processes concurrently execute CS_1 and CS_2. A safety property to proscribe such states in executions of a program S would be given by the Temporal Logic formula:

$$Init_S \Rightarrow \Box\neg(in(CS_1) \wedge in(CS_2))$$

This formula asserts CS_1 and CS_2 are mutually exclusive in executions of S that start with an initial state.

For some safety properties, whether a state is considered a "bad thing" depends on what states precede it. The defining characteristic of such safety properties is a set of finite sequences of states. Prescribing a program variable x to be non-decreasing is an example of such a safety property—the "bad thing" is a pair of adjacent states in which the value of x decreases.

Given a sufficiently expressive Predicate Logic for writing *Etern*, every safety property for a program S can be specified by a Temporal Logic formula $Init_S \Rightarrow \Box Etern$. Thus far, our Predicate Logic formulas could only specify those safety properties where a set of states defines the "bad thing". This is because we defined $(\sigma, j) \models P$ (for a Predicate Logic formula P) to equal the value of P in current state $\sigma[j]$. Past states $\sigma[..j-1]$ were ignored. We now enrich the language of Predicate Logic to include formulas that are sensitive to past states.

Past Terms and Predicates

Let $(\sigma, j)[\![\mathcal{T}]\!]$ denote the value of a term $\mathcal{T}$ in anchored sequence (σ, j). For terms of ordinary Predicate Logic, $(\sigma, j)[\![\mathcal{T}]\!]$ is defined as is conventional when states, rather than anchored sequences, are the interpretations.

$\mathcal{T}$	$(\sigma, j)[\![\mathcal{T}]\!]$
constant C	C
rigid variable R	value of R in state $\sigma[0]$
variable v	value of v in state $\sigma[j]$
term $\mathcal{E}(\mathcal{T}_1, ..., \mathcal{T}_n)$	$\mathcal{E}(\ (\sigma, j)[\![\mathcal{T}_1]\!], ...,\ (\sigma, j)[\![\mathcal{T}_n]\!]\ )$

A *past term* consists of a finite sequence of Θ's (each read "previous") followed by a term. We assign to Θ the same precedence as is given to the unary operators of Predicate Logic. The value of $(\sigma, j)[\![\Theta\mathcal{T}]\!]$ is essentially the same as evaluating $\mathcal{T}$ in $(\sigma, j-1)$.

$\mathcal{T}$	$(\sigma, j)[\![\Theta\mathcal{T}]\!]$
constant or rigid variable C	C if $j \geq 1$ unspecified (but fixed) if $j < 1$
variable v	$(\sigma, j-1)[\![v]\!]$ if $j \geq 1$ unspecified (but fixed) if $j < 1$
term $\mathcal{E}(\mathcal{T}_1, ..., \mathcal{T}_n)$	$(\sigma, j-1)[\![\mathcal{E}(\mathcal{T}_1, ..., \mathcal{T}_n)]\!]$ if $j \geq 1$ unspecified (but fixed) if $j < 1$

For example, the value of Θx in $(s_0\, s_1\, s_2, 2)$, is the value of x in s_1. So, the value $\Theta x \leq x$ in $(s_0\, s_1\, s_2, 2)$ is *true* iff the value of x in s_1 is no greater than the value of x in s_2.

Consistent with the view that a Predicate Logic formula is a boolean-valued term, Θ may be applied to the formulas of Predicate Logic. It has the expected meaning based on the definition just given for $(\sigma, j)[\![\Theta\mathcal{T}]\!]$.

$$(\sigma, j) \models \Theta P: \quad \begin{cases} (\sigma, j-1)[\![P]\!] = \mathit{true} & \text{if } j \geq 1 \\ \text{unspecified (but boolean)} & \text{if } j < 1 \end{cases}$$

Finally, in order to characterize those anchored sequences for which a given past term is defined, we introduce a nullary predicate def_Θ.

$$(\sigma, j)[\![\mathit{def}_\Theta]\!]: \quad j > 0$$

Predicate def_Θ allows formulas to have specified values in any anchored sequence (σ, j), regardless of $|\sigma|$. An example is $\mathit{def}_\Theta \Rightarrow \Theta x \leq x$, which has a specified value in all anchored sequences; in contrast, $\Theta x \leq x$ has an unspecified value for sequences having a single state, because the value of Θx is unspecified in these sequences.

The following rules suffice for reasoning about formulas with Θ and def_Θ.

(4.12) Θ *Expression Expansion*: For $\mathcal{E}(\mathcal{T}_1, ..., \mathcal{T}_n)$ a non-nullary term or formula that is constructed from terms $\mathcal{T}_1, ..., \mathcal{T}_n$:

$$def_\Theta \;\Rightarrow\; (\Theta\mathcal{E}(\mathcal{T}_1, ..., \mathcal{T}_n) = \mathcal{E}(\Theta\mathcal{T}_1, ..., \Theta\mathcal{T}_n))$$

(4.13) Θ *Constant Expansion*: For a rigid variable or constant C:

$$def_\Theta \;\Rightarrow\; (\Theta \mathrm{C} = \mathrm{C})$$

(4.14) *Textual Substitution* [*Past Term*]: For a past term $\Theta\mathcal{T}$:

$$(\Theta\mathcal{T})_e^x = \Theta\mathcal{T}$$

(4.15) *Trace Induction Rule*: $\dfrac{\neg\, def_\Theta \Rightarrow P,\quad (def_\Theta \wedge \Theta P) \Rightarrow P}{P}$

Derived Terms

$Init_S \Rightarrow \Box Etern$ can describe only those safety properties for which the "bad thing" is definable as $\neg Etern$. However, the "bad thing" of a safety property might be any set of finite sequences of states. Therefore, to be able to use $Init_S \Rightarrow \Box Etern$ for specifying any safety property, we must be able to characterize any set of finite sequences of states using a Predicate Logic formula $\neg Etern$.

Some sets of finite sequences of states can be characterized only by writing a formula that depends on all of the states in a sequence. An example is finite sequences of states in which x is non-decreasing. A formula whose past terms involved n Θ's can depend on at most $n+1$ of the states in an anchored sequence; but, an arbitrary finite sequence might have more than $n+1$ states. Thus, extending Predicate Logic with Θ and def_Θ does not yield a logic that is sufficiently expressive for our purposes.

A Predicate Logic with the expressiveness we seek results if we allow a form of primitive recursive definition over the sequence of past states. We do this by adding a new class of terms. To define a *derived term*, we give its name and a method for computing its (unique) value in each anchored sequence.[2] The syntax we employ for defining a derived term Z is to give a collection of *clauses*, each comprising an *expression* e_i and a *guard* B_i

$$Z\colon \begin{cases} e_1 & \text{if } B_1 \\ \quad\cdots & \\ e_n & \text{if } B_n \end{cases}$$

[2]By convention, derived terms are named by identifiers starting with an upper-case letter.

where:

- Z does not appear in guards.
- Each occurrence of Z in an expression e_i appears in the scope of Θ^{i} for $\mathrm{i} > 0$.
- Each expression e_i containing Z in the scope of Θ^{i} has an associated guard B_i containing conjunct $\Theta^{\mathrm{i}-1} def_\Theta$.

The value of Z in (σ, j) is $(\sigma, j)[\![e_i]\!]$ where e_i is the expression corresponding to the unique guard B_i that holds. If no guard holds or more than one guard holds, then the value of Z is unspecified.

An example of a derived term is M, defined below. The value of M in (σ, j) is the largest value x assumes in states $\sigma[0]$, $\sigma[1]$, ..., $\sigma[j]$.

$$M\colon \begin{cases} x \quad \text{if } \neg\, def_\Theta \\ \max(x, \Theta M) \quad \text{if } def_\Theta \end{cases}$$

Notice how the presence of ΘM in the second clause causes the value of M to depend on all states, even though only a fixed number of Θ's are mentioned in the definition.

A variant of Leibniz's law—substitution of equals for equals—allows a derived term Z in a Predicate Logic formula to be replaced by its definition. In the following, we denote a term $\mathcal{T}$ prefixed by i Θ operators by $\Theta^{\mathrm{i}}\mathcal{T}$. When i is 0, then $\Theta^{\mathrm{i}}\mathcal{T}$ is just $\mathcal{T}$.

(4.17) *Derived Term Expansion Rule*: For Z a derived term

$$Z\colon \begin{cases} e_1 \quad \text{if } B_1 \\ \quad \cdots \\ e_n \quad \text{if } B_n \end{cases}$$

and P a Predicate Logic formula where x does not occur free within the scope of Θ:

$$\frac{\bigwedge_{1 \le k \le n} (\Theta^{\mathrm{i}} B_k = \neg (\bigvee_{j \ne k} \Theta^{\mathrm{i}} B_j))}{P^x_{\Theta^{\mathrm{i}} Z} = ((\Theta^{\mathrm{i}} B_1 \wedge P^x_{\Theta^{\mathrm{i}} e_1}) \vee \cdots \vee (\Theta^{\mathrm{i}} B_n \wedge P^x_{\Theta^{\mathrm{i}} e_n}))}$$

The hypothesis of the rule ensures that exactly one of the guards $\Theta^{\mathrm{i}} B_k$ holds, thereby ensuring that the value of Z is not unspecified.

5. Syntax and Meaning of Proof Outlines

The formulas of Proof Outline Logic include Predicate Logic formulas, proof outlines for programs, and triples for guard evaluation actions. A *proof outline* $PO(S)$ for a program S is a program in which every statement is preceded and followed by an assertion enclosed in braces ("{" and "}"). Fig. 5.1 contains an example. A *triple* is a proof outline $\{P\}\ S\ \{Q\}$ in which program S is a single atomic action.

An *assertion* is a Predicate Logic formula in which all free variables[3] are program variables or rigid variables, and all predicates are control predicates or predicates defined by the types of the program variables. Assertions that depend only on the values of program variables in the current state are called *primitive*. Thus, primitive assertions may not mention control predicates, Θ, or def_Θ. For example, in the proof outline of Fig. 5.1, x is a program variable, X is a rigid variable, and all assertions except the first and last are primitive.

The assertion that immediately precedes a statement T in a proof outline is called the *precondition* of T and is denoted by $pre(T)$; the assertion that directly follows T is called the *postcondition* of T and is denoted by $post(T)$. For the proof outline in Fig. 5.1, this correspondence is summarized in Fig. 5.2. Finally, for a proof outline $PO(S)$, we write $pre(PO(S))$ to denote $pre(S)$, $post(PO(S))$ to denote $post(S)$, and write

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

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 $PO(S)$.

Meaning of Proof Outlines

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 $Parts(S)$. 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.

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

Fig. 5.1. Computing $abs(x)$

[3]Program variables are typeset in lower-case italic; rigid variables are typeset in upper-case roman.

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

Fig. 5.2. Assertions in a Proof Outline

In most cases, a control point is mapped to a single assertion. For example, the proof outline

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

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 active, and (5.2) associates P with both of these control predicates.

However, a proof outline can map a given control point to a set with more than one assertion. An example of this appears in Fig. 5.1. 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*.

Assertions in a proof outline are intended to characterize the program state as execution proceeds. The proof outline of Fig. 5.1, 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, is 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 $PO(S)$

(5.3) $$I_{PO(S)}\colon \bigwedge_{T \in Stmts(S)} ((at(T) \Rightarrow pre(T)) \wedge (after(T) \Rightarrow post(T))),$$

where $Stmts(T)$ is $Parts(T)$ with all guard evaluation actions removed.

Notice that our definition for proof outline validity requires that $I_{PO(S)}$ not be falsified by execution started in a program state satisfying $I_{PO(S)}$ that could never arise by executing S from an initial state. For example,

$$\{x=0 \wedge y=0\}\ S_1\text{:}\ \textbf{skip}\ \{x=0\}\ S_2\text{:}\ \textbf{skip}\ \{x=0 \wedge y=0\}$$

is *not* valid since execution of S_2 in a program state satisfying $at(S_2)$, $x=0$, and $y=15$ falsifies the proof outline invariant because $x=0 \wedge y=0$ will not hold when $after(S_2)$ becomes *true*.

Equating proof outline validity with invariance of $I_{PO(S)}$ leads to technical complications when a proof outline $PO(S)$ maps the entry control point of S to multiple assertions. To illustrate, consider the following concurrent program to increment x and y.

(5.4) S: **cobegin** T: $x := x+1$ || T': $y := y+1$ **coend**

According to the control predicate axioms for (5.4), $at(S) \Rightarrow at(T)$ and $at(S) \Rightarrow at(T')$ are theorems. Thus, the proof outline of Fig. 5.3 associates $pre(S)$, $pre(T)$, and $pre(T')$ with the entry control point for S. This means, however, that $pre(PO(S))$ does not characterize states in which S could be started and have $I_{PO(S)}$ hold: $at(S) \wedge pre(PO(S))$ does not imply $I_{PO(S)}$.

We avoid problems caused by associating multiple assertions with an entry control point if we also require that $pre(PO(S))$ implies $I_{PO(S)}$ in order for $PO(S)$ to be considered valid. Define a proof outline $PO(S)$ to be *self consistent* if and only if $at(S) \wedge pre(PO(S)) \Rightarrow I_{PO(S)}$ is valid. The proof outline of Fig. 5.3 is not self consistent.

We can now formalize the requirements for validity of a proof outline in terms of $\ddot{\mathcal{H}}_S$-validity of temporal logic formulas.

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

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

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

$\{true\}$
S: **cobegin**
    $\{x=\mathrm{X}\}$ T: $x := x+1$ $\{x=\mathrm{X}+1\}$
    ||
    $\{y=\mathrm{Y}\}$ T': $y := y+1$ $\{y=\mathrm{Y}+1\}$
  coend
$\{x=\mathrm{X}+1 \wedge y=\mathrm{Y}+1\}$

Figure 5.3. Incrementing x and y

From this definition of proof outline validity, we infer that rigid variables in proof outlines allow us to relate the values of program variables from one state to the next. This is because $I_{PO(S)} \Rightarrow \Box I_{PO(S)}$ is a $\ddot{\mathcal{H}}_S$-valid temporal logic formula 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 $I_{PO(S)}$.

From Proof Outlines to Safety Properties

To prove $\ddot{\mathcal{H}}_S \models Init \Rightarrow \Box Etern$, it suffices to find a Predicate Logic formula I for which the following are $\ddot{\mathcal{H}}_S$-valid:

(5.6) $Init \Rightarrow I$

(5.7) $I \Rightarrow \Box I$

(5.8) $I \Rightarrow Etern$

Thus, I is an invariant and is satisfied whenever execution cannot lead to the "bad thing" (i.e. $\neg Etern$) being proscribed. Because not all anchored sequences satisfying *Etern* are ones from which *Etern* will continue to hold, I is typically stronger than *Etern*.

The $\ddot{\mathcal{H}}_S$-validity of (5.6), (5.7) and (5.8) suffices for proving $\ddot{\mathcal{H}}_S \models Init \Rightarrow \Box Etern$ because we can use ordinary Temporal Logic (which is sound for $\ddot{\mathcal{H}}_S$-validity) as follows.

$$\begin{array}{ll}
 & Init \\
\Rightarrow & \quad «(5.6)» \\
 & I \\
\Rightarrow & \quad «(5.7)» \\
 & \Box I \\
\Rightarrow & \quad «(5.8) \text{ and rule } \dfrac{P \Rightarrow Q}{\Box P \Rightarrow \Box Q}» \\
 & \Box Etern
\end{array}$$

Predicate Logic (as extended above with program variable axioms, control predicate axioms, and axioms for Θ and def_{Θ}) can be used to prove $\ddot{\mathcal{H}}_S$-validity of (5.6) and (5.8). This is because *Init*, I, and *Etern* are formulas of that logic, and the logic is complete.

Showing that I is invariant, as required to establish that (5.7) is $\ddot{\mathcal{H}}_S$-valid, is not as simple. It involves reasoning about program execution. Proof Outline Logic was designed for just this type of reasoning. According to Valid Proof Outline (5.5), if $PO(S)$ is a theorem of Proof Outline Logic, then $I_{PO(S)} \Rightarrow \Box I_{PO(S)}$ is $\ddot{\mathcal{H}}_S$-valid. Thus, demonstrating that $I \Rightarrow \Box I$ is $\ddot{\mathcal{H}}_S$-valid is equivalent to proving a theorem of Proof Outline Logic, and we have the following rule for verifying that a program S satisfies a safety property.

$$(5.9)\quad \textit{Safety Rule}: \quad \frac{\begin{array}{l}\text{(a)}\ \ PO(S),\\ \text{(b)}\ \ Init \Rightarrow I_{PO(S)},\\ \text{(c)}\ \ I_{PO(S)} \Rightarrow Etern\end{array}}{Init \Rightarrow \Box Etern}$$

A variant of this rule involves showing that states satisfying $\neg Etern$ cannot arise during execution.

$$(5.10)\quad \textit{Exclusion of Configurations Rule}: \quad \frac{\begin{array}{l}\text{(a)}\ \ PO(S),\\ \text{(b)}\ \ Init \Rightarrow I_{PO(S)},\\ \text{(c)}\ \ \neg Etern \wedge I_{PO(S)} \Rightarrow false\end{array}}{Init \Rightarrow \Box Etern}$$

Soundness of this variant is established by proving that its hypothesis (c) implies hypothesis (c) of Safety Rule (5.9), since hypotheses (a) and (b) of Exclusion of Configurations Rule (5.10) are identical to hypotheses (a) and (b) of Safety Rule (5.9). Here is that proof.

$$\begin{array}{ll} & \neg Etern \wedge I_{PO(S)} \Rightarrow false \\ = & \qquad \text{«Law of Implication»} \\ & Etern \vee \neg I_{PO(S)} \vee false \\ = & \qquad \text{«Law of Or-simplification»} \\ & Etern \vee \neg I_{PO(S)} \\ = & \qquad \text{«Commutative Law»} \\ & \neg I_{PO(S)} \vee Etern \\ = & \qquad \text{«Law of Implication»} \\ & I_{PO(S)} \Rightarrow Etern \end{array}$$

6. Axioms and Inference Rules for Proof Outlines

There is an axiom or inference rule for **skip**, assignment, statement juxtaposition, **if**, **do**, their guard evaluation actions, and **cobegin**, because these are the statements and atomic actions of our programming language. There are also some statement-independent inference rules. The resulting logic is sound and complete relative to our Predicate Logic.

6.1. Axiomatizing Sequential Statements

The first axiom of Proof Outline Logic is for **skip**.

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

The next axiom is for an assignment $\overline{x} := \overline{e}$ where $\overline{x}$ is a list $x_1, x_2, ..., x_n$ of identifiers (i.e. not elements of records or arrays[4]) and $\overline{e}$ is a list $e_1, e_2, ..., e_n$ of expressions.

(6.2) *Assignment Axiom*: For a primitive assertion P: $\{P^{\overline{x}}_{\overline{e}}\}\ \overline{x} := \overline{e}\ \{P\}$

A proof outline for the juxtaposition of two statements can be derived from proof outlines for each of its components.

(6.3) *Statement Juxtaposition Rule*: $\dfrac{\{P\}\ PO(S_1)\ \{Q\}, \quad \{Q\}\ PO(S_2)\ \{R\}}{\{P\}\ PO(S_1)\ \{Q\}\ PO(S_2)\ \{R\}}$

The guard evaluation action for an **if** ensures that the appropriate statement is selected for execution. This is reflected in the following axiom.

(6.4) $GEval_{if}(S)$ *Axiom*: For an **if** statement

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

and a primitive assertion P:

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

The inference rule for **if** permits a valid proof outline to be inferred from valid proof outlines for its components.

(6.5) **if** *Rule*:

$$\frac{\begin{array}{l}\text{(a)}\ \ \{P\}\ GEval_{if}(S)\ \{R\},\\ \text{(b)}\ \ (R \wedge at(S_1)) \Rightarrow P_1,\ \ ...,\ \ (R \wedge at(S_n)) \Rightarrow P_n,\\ \text{(c)}\ \ \{P_1\}\ PO(S_1)\ \{Q\},\ \ ...,\ \ \{P_n\}\ PO(S_n)\ \{Q\}\end{array}}{\begin{array}{l}\{P\}\\ S:\ \mathbf{if}\ \ B_1 \rightarrow \{P_1\}\ PO(S_1)\ \{Q\}\\ \qquad []\ \ \cdots\\ \qquad []\ \ B_n \rightarrow \{P_n\}\ PO(S_n)\ \{Q\}\\ \qquad \mathbf{fi}\\ \{Q\}\end{array}}$$

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.

[4]See [10] for the extensions necessary to handle elements of records and arrays.

(6.6) *$GEval_{do}(S)$ Axiom*: For a **do** statement

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

and a primitive assertion P:

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

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.

(6.7) **do** *Rule*:

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

6.2. Axiomatizing Concurrent Statements

The inference rule for **cobegin** is based on proving interference-freedom—that execution of no atomic action invalidates an assertion in another process. Define $pre^*(\alpha)$ to be the predicate that, according to the assertions in the proof outline containing α, is satisfied just before α executes:

(6.8) **Precondition of an Action.** If α is a **skip**, assignment, or atomic statement with label S, or α is guard evaluation action $GEval_{if}(S)$ for an **if** with label S, then:

$$pre^*(\alpha):\ \ pre(S)$$

If α is guard evaluation action $GEval_{do}(S)$ for a **do**

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

then:

$$pre^*(\alpha):\ \ pre(S)\ \vee\ (\bigvee_{1 \le i \le n} post(S_i))$$

□

The condition that α does not invalidate an assertion A is then implied by the validity of the *interference freedom triple*:

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

Generalizing, we conclude that no atomic action α from one process can interfere with the proof outline invariant for any other process provided:

(6.9) **Interference Freedom Condition.** $PO(S_1)$, ..., $PO(S_n)$ are proved *interference free* by establishing:

$$\begin{array}{l} \text{For all } i,\ j,\ 1 \le i \le n,\ 1 \le j \le n,\ i \ne j: \\ \quad \text{For all atomic actions } \alpha \in \mathcal{A}(S_i): \\ \quad\quad \text{For all assertions } A \text{ in } PO(S_j):\ NI(\alpha, A) \end{array}$$

□

Constructing proof outlines for the processes in a **cobegin** and establishing that these are interference free suffices to ensure validity of proof outline for the **cobegin** constructed using these processes.

(6.10) **cobegin** *Rule*:

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

In addition, we know execution of no process can change the value of a control predicate associated with another. This gives rise to:

(6.11) *Process Independence Axiom*: If α and β are from different processes of a **cobegin** and $cp(\beta)$ denotes one of the control predicates $at(\beta)$, $in(\beta)$, $after(\beta)$, or its negation, then:

$$\{cp(\beta)\}\ \alpha\ \{cp(\beta)\}$$

Atomic Statements

If P and Q are primitive assertions and $\{P\}\ PO(S)\ \{Q\}$ is valid, then $\{P\}\ \langle S \rangle\ \{Q\}$ is also valid. The following inference rule is based on this observation.

(6.12) $\langle S \rangle$ *Rule*: For primitive assertions P and Q:

$$\frac{\{P\}\ PO(S)\ \{Q\}}{\{P\}\ \langle S \rangle\ \{Q\}}$$

Second, by definition, an atomic action α cannot execute to completion in any state satisfying $\neg enbl(\alpha)$. Since $\{P\}\ \alpha\ \{Q\}$ is valid if execution of α that starts in a state satisfying P does not terminate, we have the following (derived) Proof Outline Logic rule.

(6.13) *Blocked Atomic Action Rule*: For any assertion Q and any atomic action or atomic statement α:

$$\{\neg\, enbl(\alpha)\}\ \ \alpha\ \ \{Q\}$$

6.3. Program-independent Rules

We now turn to the statement-independent inference rules of Proof Outline Logic. Rule of Consequence (6.14) allows the precondition of a proof outline to be strengthened and the postcondition to be weakened, based on deductions possible in Predicate Logic.

(6.14) *Rule of Consequence*: $$\frac{P' \Rightarrow P,\quad \{P\}\ PO(S)\ \{Q\},\quad Q \Rightarrow Q'}{\{P'\}\ PO(S)\ \{Q'\}}$$

The presence of Predicate Logic formulas in the hypothesis of this rule and the next one forces the completeness of Proof Outline Logic to be relative to Predicate Logic.

Rule of Equivalence (6.15) allows assertions anywhere in a proof outline to be modified. In particular, the rule allows a proof outline $PO'(S)$ for a program S to be inferred from another proof outline $PO(S)$ for that program when $I_{PO(S)}$ and $I_{PO'(S)}$ are equivalent and $PO'(S)$ is self consistent.

(6.15) *Rule of Equivalence*: $$\frac{\begin{array}{l}\text{(a)}\ \ PO(S),\\ \text{(b)}\ \ I_{PO(S)} = I_{PO'(S)},\\ \text{(c)}\ \ pre(PO'(S)) \wedge at(S) \Rightarrow pre(PO(S))\end{array}}{PO'(S)}$$

Control-Predicate Deletion is a derived rule that allows certain control predicates in assertions to be deleted. It is easily derived from Rule of Equivalence (6.15).

(6.16) *Control-Predicate Deletion*: $$\frac{\{P \wedge at(S)\}\ PO(S)\ \{Q \vee \neg\, after(S)\}}{\{P\}\ PO(S)\ \{Q\}}$$

Control-Point Identity allows control predicates to be added to assertions. This rule, too, can be derived from Rule of Equivalence (6.15).

(6.17) *Control-Point Identity*: $$\frac{\{P\}\ PO(S)\ \{Q\}}{\{P \wedge at(S)\}\ PO(S)\ \{Q \wedge after(S)\}}$$

The Rigid Variable Rule allows a rigid variable to be renamed or replaced by a specific value. We write $PO(S)^{\mathrm{X}}_{\mathrm{Exp}}$ in the conclusion of the rule to denote a proof outline in which rigid variable X in every assertion is replaced by Exp, an expression only involving constants and rigid variables.

(6.18) *Rigid Variable Rule*: $\dfrac{\{P\}\ PO(S)\ \{Q\}}{\{P^{\mathrm{X}}_{\mathrm{Exp}}\}\ PO(S)^{\mathrm{X}}_{\mathrm{Exp}}\ \{Q^{\mathrm{X}}_{\mathrm{Exp}}\}}$

The Conjunction and Disjunction Rules allow two proof outlines for the same program to be combined. Given proof outlines $PO_A(S)$ and $PO_B(S)$ for S, let A_{cp} be the assertion that $PO_A(S)$ associates with control predicate cp and let B_{cp} be the assertion that $PO_B(S)$ associates with cp. Define $PO_A(S) \otimes PO_B(S)$ to be a proof outline that associates assertion $A_{cp} \wedge B_{cp}$ with each control predicate cp. The following Conjunction Rule states that $PO_A(S) \otimes PO_B(S)$ can be inferred from $PO_A(S)$ and $PO_B(S)$.

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

Define $PO_A(S) \oslash PO_B(S)$ to be a proof outline that associates assertion $A_{cp} \vee B_{cp}$ with each control predicate cp. The Disjunction Rule allows $PO_A(S) \oslash PO_B(S)$ to be inferred from $PO_A(S)$ and $PO_B(S)$.

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

Terms and predicates involving Θ or def_Θ can be introduced into assertions with the following rule. Observe that Rule of Consequence (6.14) alone would not be sufficient, as illustrated by $\{x=0\}$ **skip** $\{\Theta x=0\}$, which is valid but cannot be proved without a rule like the following.

(6.21) Θ^{i}-*Introduction Rule*: For an atomic action α, non-negative integer i, past term $\Theta^{\mathrm{i}+1}\mathcal{T}$, rigid variable X, and primitive assertions P and Q:

$$\frac{\{P\}\ \ \alpha\ \ \{Q\}}{\{P^{\mathrm{X}}_{\Theta^{\mathrm{i}}\mathcal{T}}\}\ \ \alpha\ \ \{Q^{\mathrm{X}}_{\Theta^{\mathrm{i}+1}\mathcal{T}}\ \wedge\ \Theta(P^{\mathrm{X}}_{\Theta^{\mathrm{i}}\mathcal{T}})\ \wedge\ def_\Theta\ \}}$$

Θ^{i}-Introduction Rule (6.21) is sound because a rigid variable in a proof outline denotes the same value in all assertions. Thus, rigid variable X in the pre- and postconditions of hypothesis $\{P\}\ \alpha\ \{Q\}$ can be uniformly replaced by the value of $\Theta^{\mathrm{i}}\mathcal{T}$. The value of $\Theta^{\mathrm{i}}\mathcal{T}$ before α is executed is the same as the value of $\Theta^{\mathrm{i}+1}\mathcal{T}$ after α has completed. So, if X in precondition P is replaced by $\Theta^{\mathrm{i}}\mathcal{T}$ then X in postcondition Q can be replaced by $\Theta^{\mathrm{i}+1}\mathcal{T}$. The remaining two conjuncts, $\Theta(P^{\mathrm{X}}_{\Theta^{\mathrm{i}}\mathcal{T}})$ and def_Θ, in the postcondition are satisfied if α terminates, because executing α adds one more state to a sequence that is known to have been satisfied by precondition $P^{\mathrm{X}}_{\Theta^{\mathrm{i}}\mathcal{T}}$.

7. Developing Programs for Safety Properties

It is not unusual to be asked to design a program that satisfies some given safety properties. Proof Outline Logic obviously has application in determining whether this job has been completed. Perhaps not so obvious is how the logic has application in the development of programs: By keeping in mind during construction of a program how we intend to prove that it satisfies the safety properties of interest, possible refinements can be restricted to those furthering our goal. Moreover, constructing proof and program together virtually ensures success in ultimately verifying that the final program satisfies desired safety properties.

7.1. Mutual Exclusion Protocol

We illustrate this approach to program design by deriving a solution to the mutual exclusion problem, a classical concurrent programming exercise. A *mutual exclusion protocol* ensures that execution of selected statements, called *critical sections*, exclude each other.

The mutual exclusion problem is usually posed in terms of two processes, each of which executes a critical section and a non-critical section. This situation is illustrated in Fig.7.1. For each process S_i, we must design an entry protocol $entry_i$ and an exit protocol $exit_i$ to ensure that execution of critical sections satisfy:

(7.1) *Mutual Exclusion*. In no history satisfying $Init_S$ is there a state in which control is inside both CS_1 and CS_2.

(7.2) *Entry Non Blocking*. In no history satisfying $Init_S$ is there a state where both processes are blocked executing their entry protocols.

(7.3) *NCS Non Blocking*. In no history satisfying $Init_S$ is there a state where a process becomes blocked executing its entry protocol when the other is executing outside of its entry protocol, critical section, and exit protocol.

```
S:  cobegin
    S1:  do true → entry1
                   CS1
                   exit1
                   NCS1
         od
    ||
    S2:  do true → entry2
                   CS2
                   exit2
                   NCS2
         od
    coend
```

Fig. 7.1. Mutual Exclusion Problem

(7.4) *Exit Non Blocking*. In no history satisfying $Init_S$ is there a state where a process becomes blocked executing its exit protocol.

Ensuring Mutual Exclusion

It is impossible to formalize non-blocking properties (7.2), (7.3), and (7.4) without first knowing what conditional atomic actions are in the entry and exit protocols. Therefore, we start out by constructing entry and exit protocols to ensure Mutual Exclusion (7.1), which is formalized as:

(7.5) $Init_S \Rightarrow \Box \neg (in(CS_1) \wedge in(CS_2))$

Once candidate protocols have been developed, we return to the three non-blocking properties.

We begin by devising a proof outline for the program of Fig. 7.1 with an eye towards proving (7.5). In this initial proof outline, the entry and exit protocols are **skip** statements since there is no reason to choose otherwise. A failure to prove (7.5) will then identify assertions that must be strengthened for the proof of Mutual Exclusion (7.1) to succeed. These assertions are strengthened by modifying the entry and exit protocols.

Fig. 7.2 gives an initial proof outline for the program of Fig. 7.1. $PO(S)$ of

$\{true\}$
S: **cobegin**
    $\{\neg\, in(CS_1)\}$
    S_1: **do** $true \rightarrow \{\neg\, in(CS_1)\}$
            $entry_1$: **skip**
            $\{in(CS_1)\}\ PO(CS_1)\ \{\neg\, in(CS_1)\}$
            $exit_1$: **skip**
            $\{\neg\, in(CS_1)\}\ PO(NCS_1)\ \{\neg\, in(CS_1)\}$
        od $\{false\}$
    ||
    $\{\neg\, in(CS_2)\}$
    S_2: **do** $true \rightarrow \{\neg\, in(CS_2)\}$
            $entry_2$: **skip**
            $\{in(CS_2)\}\ PO(CS_2)\ \{\neg\, in(CS_2)\}$
            $exit_2$: **skip**
            $\{\neg\, in(CS_2)\}\ PO(NCS_2)\ \{\neg\, in(CS_2)\}$
        od $\{false\}$
    coend
$\{false\}$

Fig. 7.2. Initial Proof Outline for Mutual Exclusion Problem

Fig. 7.2 is a Proof Outline Logic theorem. Hypothesis (a) of Safety Rule (5.9) is therefore satisfied for proving (7.5). To discharge hypothesis (b), it suffices that $at(S)$ be $Init_S$. And, to prove hypothesis (c), we must show

(7.6) $loc(A) \wedge A \Rightarrow \neg(in(CS_1) \wedge in(CS_2))$

for each assertion A that is associated by the proof outline with control predicate $loc(A)$. Unfortunately, (7.6) is not valid for assertions in $PO(CS_1)$ and $PO(CS_2)$. However, from this failure to prove (7.5), we have learned that assertions in $PO(CS_1)$ must be strengthened so that each implies $\neg in(CS_2)$ and assertions in $PO(CS_2)$ must be strengthened so that each implies $\neg in(CS_1)$.

To accomplish this strengthening, we alter the entry protocols. We find predicates B_1 and B_2 such that

$$I\colon \ (B_1 \Rightarrow \neg in(CS_2)) \ \wedge \ (B_2 \Rightarrow \neg in(CS_1))$$

holds throughout execution. An **if** with guard B_1 now can be used to strengthen $pre(PO(CS_1))$ with B_1 and anything $I \wedge B_1$ implies—in particular, by $\neg in(CS_2)$. We can similarly strengthen $pre(PO(CS_2))$ with B_2 and anything $I \wedge B_2$ implies.

Next, this stronger assertion is propagated to strengthen the other assertions in $PO(CS_1)$ and $PO(CS_2)$ with these same conjuncts. These strengthenings result in the following modifications to the proof outline of Fig. 7.2, where $PO(S) \otimes P$ denotes the proof outline in which every assertion of $PO(S)$ is strengthened by conjunct P.

...
S_1: ... $\{I \wedge \neg in(CS_1)\}$
  $entry_1$: **if** $B_1 \rightarrow \{I \wedge B_1\}$ T_1: **skip** $\{I \wedge B_1\}$ **fi**
  $\{I \wedge B_1\}$
  $PO(CS_1) \otimes (I \wedge B_1)$
  ...
||
S_2: ... $\{I \wedge \neg in(CS_2)\}$
  $entry_2$: **if** $B_2 \rightarrow \{I \wedge B_2\}$ T_2: **skip** $\{I \wedge B_2\}$ **fi**
  $\{I \wedge B_2\}$
  $PO(CS_2) \otimes (I \wedge B_2)$
  ...
...

Unfortunately, this new proof outline is not interference free. Executing T_2 invalidates $I \wedge B_1$ (in particular, $\neg in(CS_2)$) in the proof outline of S_1. This is because when T_2 terminates, $after(T_2)$ holds and, due to the following proof, $\neg in(CS_2)$ cannot hold as well.

  $after(T_2)$
$\Rightarrow$ «**if** Control Axiom (4.8b)»

$$
\begin{array}{ll}
 & after(entry_2) \\
\Rightarrow & \quad \text{«Statement Juxtaposition Control Axiom (4.7c)»} \\
 & at(CS_2) \\
\Rightarrow & \quad \text{«In Axiom (4.4a)»} \\
 & in(CS_2)
\end{array}
$$

Symmetrically, T_1 interferes with $I \wedge B_2$ in the assertions of $PO(S_2)$.

We can eliminate interference of T_2 with $I \wedge B_1$ by strengthening both $pre(T_2)$ and I so that $pre(T_2) \wedge I \wedge B_1$ equals *false*, making $NI(T_2, I \wedge B_1)$ valid. To accomplish this, we strengthen $pre(T_2)$ with the conjunct $at(T_2)$ and modify I so that $I \wedge B_1 \Rightarrow \neg at(T_2)$. Making symmetric modifications to eliminate interference of T_1 with $I \wedge B_2$, results in the following new definition for I

$$I: \quad (B_1 \Rightarrow \neg (in(CS_2) \vee at(T_2))) \quad \wedge \quad (B_2 \Rightarrow \neg (in(CS_1) \vee at(T_1)))$$

and the following revised proof outline.

$$
\begin{array}{lll}
\ldots & & \\
S_1: & \ldots & \{I \wedge \neg in(CS_1)\} \\
 & & entry_1\text{: } \textbf{if } B_1 \rightarrow \{I \wedge at(T_1) \wedge B_1\} \;\; T_1\text{: } \textbf{skip} \;\; \{I \wedge B_1\} \textbf{ fi} \\
 & & \{I \wedge B_1\} \\
 & & PO(CS_1) \otimes (I \wedge B_1) \\
 & & \ldots \\
\| & & \\
S_2: & \ldots & \{I \wedge \neg in(CS_2)\} \\
 & & entry_2\text{: } \textbf{if } B_2 \rightarrow \{I \wedge at(T_2) \wedge B_2\} \;\; T_2\text{: } \textbf{skip} \;\; \{I \wedge B_2\} \textbf{ fi} \\
 & & \{I \wedge B_2\} \\
 & & PO(CS_2) \otimes (I \wedge B_2) \\
 & & \ldots \\
\ldots & &
\end{array}
$$

While $NI(T_2, I \wedge B_1)$ and $NI(T_1, I \wedge B_2)$ are valid in this new proof outline, it is now possible for $GEval_{if}(entry_2)$ to interfere with $I \wedge B_1$ by invalidating $\neg at(T_2)$; similarly, $GEval_{if}(entry_1)$ can interfere with $I \wedge B_2$. One more strengthening of I solves this problem.

$$
\begin{array}{ll}
I: & (B_1 \Rightarrow \neg (in(CS_2) \vee in(entry_2))) \\
 & \wedge \;\; (B_2 \Rightarrow \neg (in(CS_1) \vee in(entry_1)))
\end{array}
$$

Finally, we must ensure that execution of the atomic action preceding $entry_2$ does not invalidate I in making $at(entry_2)$ hold (and that execution of the atomic action preceding $entry_1$ does not similarly invalidate I). We solve this problem by postulating that $pre(entry_1) \Rightarrow \neg B_2$ and $pre(entry_2) \Rightarrow \neg B_1$ are valid. Thus, we have:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\ldots} \\
S_1\text{: } \ldots & \{I \wedge \neg\, in(CS_1) \wedge \neg B_2\} \\
& entry_1\text{: } \mathbf{if}\ B_1 \rightarrow \{I \wedge at(T_1) \wedge B_1\}\ \ T_1\text{: } \mathbf{skip}\ \ \{I \wedge B_1\}\ \mathbf{fi} \\
& \{I \wedge B_1\} \\
& PO(CS_1) \otimes (I \wedge B_1) \\
& \ldots \\
\multicolumn{2}{l}{||} \\
S_2\text{: } \ldots & \{I \wedge \neg\, in(CS_2) \wedge \neg B_1\} \\
& entry_2\text{: } \mathbf{if}\ B_2 \rightarrow \{I \wedge at(T_2) \wedge B_2\}\ \ T_2\text{: } \mathbf{skip}\ \ \{I \wedge B_2\}\ \mathbf{fi} \\
& \{I \wedge B_2\} \\
& PO(CS_2) \otimes (I \wedge B_2) \\
& \ldots \\
\multicolumn{2}{l}{\ldots}
\end{array}
$$

Our next task is to define B_1 and B_2 in terms of program variables, since guards may not mention control predicates. We introduce boolean program variables *in1* and *in2* and add assignments to the entry and exit protocols so that we can replace I by:

$$
\begin{array}{ll}
I\text{: } & (\neg\, in2 \Rightarrow \neg\,(in(CS_2) \vee in(entry_2))) \\
& \wedge\ \ (\neg\, in1 \Rightarrow \neg\,(in(CS_1) \vee in(entry_1)))
\end{array}
$$

Then, $\neg\, in2$ can replace B_1 and $\neg\, in1$ can replace B_2. We have only to identify assignment statements that ensure I holds throughout execution and that ensure $pre(entry_1)$ and $pre(entry_2)$ hold when they are reached.

Execution of either $entry_i$ or CS_i causes $\neg\,(in(CS_i) \vee in(entry_i))$ to become *false*. Therefore, maintaining the truth of I requires that *in1* be *true* before $entry_1$ executes and that *in2* be *true* before $entry_2$ executes. We accomplish this by adding $in1 := true$ before $entry_1$ and $in2 := true$ before $entry_2$. Since these statements are part of the entry protocol, we redefine $entry_i$ to include the assignment (labeled $door_i$) and the **if** (labeled $gate_i$). The result is shown in the following proof outline. Notice the revised definition of I to account for the renaming of statements.

$$
\begin{array}{l}
I\colon\ (\neg\, in2 \Rightarrow \neg\,(in(CS_2) \vee in(gate_2))) \\
\quad \wedge\ (\neg\, in1 \Rightarrow \neg\,(in(CS_1) \vee in(gate_1))) \\
\ldots \\
S_1\colon\ \ldots\ \{I \wedge \neg\, in(CS_1)\} \\
\qquad entry_1\colon\ door_1\colon\ in1 := true\ \ \{I \wedge \neg\, in(CS_1) \wedge in1\} \\
\qquad\qquad gate_1\colon\ \mathbf{if}\ \neg\, in2 \rightarrow \{I \wedge at(T_1) \wedge in1 \wedge \neg\, in2\} \\
\qquad\qquad\qquad\qquad\qquad T_1\colon\ \mathbf{skip}\ \ \{I \wedge in1 \wedge \neg\, in2\}\ \mathbf{fi} \\
\qquad \{I \wedge in1 \wedge \neg\, in2\} \\
\qquad PO(CS_1) \otimes (I \wedge in1 \wedge \neg\, in2) \\
\qquad \ldots \\
\| \\
S_2\colon\ \ldots\ \{I \wedge \neg\, in(CS_2)\} \\
\qquad entry_1\colon\ door_2\colon\ in2 := true\ \ \{I \wedge \neg\, in(CS_2) \wedge in2\} \\
\qquad\qquad gate_2\colon\ \mathbf{if}\ \neg\, in1 \rightarrow \{I \wedge at(T_2) \wedge in2 \wedge \neg\, in1\} \\
\qquad\qquad\qquad\qquad\qquad T_2\colon\ \mathbf{skip}\ \ \{I \wedge in2 \wedge \neg\, in1\}\ \mathbf{fi} \\
\qquad \{I \wedge in2 \wedge \neg\, in1\} \\
\qquad PO(CS_2) \otimes (I \wedge in2 \wedge \neg\, in1) \\
\qquad \ldots \\
\ldots
\end{array}
$$

Unfortunately, these new assignments cause interference. Execution of $in1 := true$ invalidates $\neg\, in1$ in assertions of S_2, and execution of $in2 := true$ invalidates $\neg\, in2$ in assertions of S_1. However, this interference can be removed by replacing $\neg\, in1$ in assertions of S_2 with $\neg\, in1 \vee after(door_1)$ and replacing $\neg\, in2$ in assertions of S_1 with $\neg\, in2 \vee after(door_2)$. The result is shown in the proof outline of Fig. 7.3, which is interference-free. Moreover, because

$$
\begin{array}{l}
(I \wedge (\neg\, in2 \vee after(door_2)) \Rightarrow \neg\, in(CS_2) \\
(I \wedge (\neg\, in1 \vee after(door_1)) \Rightarrow \neg\, in(CS_1)
\end{array}
$$

are valid, we conclude that (7.6) is valid for each assertion A in the proof outline, so Mutual Exclusion (7.1) is satisfied.

Non Blocking

Having a candidate entry protocol, we can now check whether Entry Non Blocking (7.2) is satisfied. For our protocol, this property is formalized as

$$
\text{(7.7)}\quad at(S) \Rightarrow \Box\neg\,(at(gate_1) \wedge \neg\, enbl(GEval_{if}(gate_1)) \\
\qquad\qquad \wedge\ at(gate_2) \wedge \neg\, enbl(GEval_{if}(gate_2))),
$$

because the only conditional atomic actions in the entry protocols are $GEval_{if}(gate_1)$ and $GEval_{if}(gate_2)$.

We select Exclusion of Configurations Rule (5.10) for proving (7.7). Hypothesis (a) is satisfied by the (valid) proof outline of Fig. 7.3. Hypothesis (b) is satisfied because $at(S)$ equals $Init_S$. Hypothesis (c) requires that

$\{true\}$
S: $in1, in2 := ...$
$\{I\colon (\neg\, in2 \Rightarrow \neg\,(in(CS_2) \vee in(gate_2)))$
$\quad \wedge \ (\neg\, in1 \Rightarrow \neg\,(in(CS_1) \vee in(gate_1)))\}$
cobegin
$\{I \wedge \neg\, in(CS_1)\}$
S_1: **do** $true \rightarrow \{I \wedge \neg\, in(CS_1)\}$
$entry_1$: $door_1$: $in1 := true$ $\{I \wedge \neg\, in(CS_1) \wedge in1\}$
$gate_1$: **if** $\neg\, in2 \rightarrow \{I \wedge at(T_1) \wedge in1 \wedge (\neg\, in2 \vee after(door_2))\}$
T_1: **skip**
$\{I \wedge in1 \wedge (\neg\, in2 \vee after(door_2))\}$ **fi**
$\{I \wedge in1 \wedge (\neg\, in2 \vee after(door_2))\}$
$PO(CS_1) \circledwedge (I \wedge in1 \wedge (\neg\, in2 \vee after(door_2)))$
$\{I \wedge \neg\, in(CS_1)\}$
$exit_1$: **skip**
$\{I \wedge \neg\, in(CS_1)\}$ $PO(NCS_1) \circledwedge (I \wedge \neg\, in(CS_1))$ $\{I \wedge \neg\, in(CS_1)\}$
od $\{false\}$
$||$
$\{I \wedge \neg\, in(CS_2)\}$
S_2: **do** $true \rightarrow \{I \wedge \neg\, in(CS_2)\}$
$entry_2$: $door_2$: $in2 := true$ $\{I \wedge \neg\, in(CS_2) \wedge in2\}$
$gate_2$: **if** $\neg\, in1 \rightarrow \{I \wedge at(T_2) \wedge in2 \wedge (\neg\, in1 \vee after(door_1))\}$
T_2: **skip**
$\{I \wedge in2 \wedge (\neg\, in1 \vee after(door_1))\}$ **fi**
$\{I \wedge in2 \wedge (\neg\, in1 \vee after(door_1))\}$
$PO(CS_2) \circledwedge (I \wedge in2 \wedge (\neg\, in1 \vee after(door_1)))$
$\{I \wedge \neg\, in(CS_2)\}$
$exit_2$: **skip**
$\{I \wedge \neg\, in(CS_2)\}$ $PO(NCS_2) \circledwedge (I \wedge \neg\, in(CS_2))$ $\{I \wedge \neg\, in(CS_2)\}$
od $\{false\}$
coend
$\{false\}$

Fig. 7.3. Protocol for Mutual Exclusion (7.1)

(7.8) $\quad at(gate_1) \wedge in2 \wedge at(gate_2) \wedge in1 \wedge I_{PO(S)}$

implies *false*, because $enbl(GEval_{if}(gate_1))$ is $\neg\, in2$ and $enbl(GEval_{if}(gate_2))$ is $\neg\, in1$. Unfortunately, (7.8) does not imply *false*; it implies

(7.9) $\quad at(gate_1) \wedge in2 \wedge at(gate_2) \wedge in1 \wedge I \wedge \neg\, in(CS_1) \wedge \neg\, in(CS_2)$.

Either the proof outline of Fig. 7.3 is not strong enough to prove (7.7) or this property is not satisfied by our protocol. Working backwards from a state satisfying (7.9), we find that execution of $door_1$ followed by $door_2$ results in a state where S_1 is blocked at $gate_1$ and S_2 is blocked at $gate_2$. The entry protocol we have developed simply does not satisfy Entry Non Blocking (7.2).

To eliminate this deadlock, we use weaker guards in $gate_1$ and $gate_2$— weaker guards mean fewer states will cause blocking. Constraints on these guards can be determined by using an as yet unspecified disjunct X_i to accomplish the weakening for $gate_i$. The proof outline for S_1 with such a weaker guard would be:

$$
\begin{array}{l}
...\\
\{I \wedge \neg\, in(CS_1)\}\\
entry_1\!:\ door_1\!:\ in1 := true\ \ \{I \wedge \neg\, in(CS_1) \wedge in1\}\\
\qquad\quad gate_1\!:\ \mathbf{if}\ \neg\, in2 \vee X_1 \rightarrow\\
\qquad\qquad\qquad\qquad \{I \wedge at(T_1) \wedge in1 \wedge (\neg\, in2 \vee X_1 \vee after(door_2))\}\\
\qquad\qquad\qquad\qquad T_1\!:\ \mathbf{skip}\\
\qquad\qquad\qquad\qquad \{I \wedge in1 \wedge (\neg\, in2 \vee X_1 \vee after(door_2))\}\ \mathbf{fi}\\
\{I \wedge in1 \wedge (\neg\, in2 \vee X_1 \vee after(door_2))\}\\
PO(CS_1) \otimes (I \wedge in1 \wedge (\neg\, in2 \vee X_1 \vee after(door_2)))\\
...
\end{array}
$$

Constraints on X_1 and X_2 that ensure Entry Non Blocking (7.2) is satisfied are now obtained by using the proof outline with weaker guards and repeating the above proof for (7.7). Notice that if $\neg X_1 \wedge \neg X_2 \Rightarrow false$ is valid, then so is

$$
\begin{array}{l}
at(gate_1) \wedge \neg(\neg\, in2 \vee X_1) \wedge at(gate_2) \wedge \neg(\neg\, in1 \vee X_2)\\
\ \wedge\, I \wedge \neg\, in(CS_1) \wedge in1 \wedge \neg\, in(CS_2) \wedge in2 \qquad \Rightarrow \qquad false,
\end{array}
$$

and hypothesis (c) of Exclusion of Configurations Rule (5.10) is satisfied. Therefore, if X_1 and X_2 are predicates that cannot simultaneously be *false* then Entry Non Blocking (7.2) will hold.

An obvious choice is to define a single variable, say t. Strengthening I to be

$$
\begin{array}{l}
I\!:\ \ (\neg\, in2 \Rightarrow \neg\,(in(CS_2) \vee in(gate_2)))\\
\qquad \wedge\ (\neg\, in1 \Rightarrow \neg\,(in(CS_1) \vee in(gate_1)))\\
\qquad \wedge\ (t{=}1 \vee t{=}2),
\end{array}
$$

allows us to use $t{=}1$ for X_1 and use $t{=}2$ for X_2. We make the substitution into the proof outlines to get:

$$
\begin{array}{l}
\ldots \\
\{I \wedge \neg\, in(CS_1)\} \\
entry_1\!: \; door_1\!: \; in1 := true \;\; \{I \wedge \neg\, in(CS_1) \wedge in1\} \\
\qquad\qquad gate_1\!: \; \mathbf{if} \; \neg\, in2 \vee t{=}1 \rightarrow \{I \wedge at(T_1) \wedge in1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge (\neg\, in2 \vee t{=}1 \vee after(door_2))\} \\
\qquad\qquad\qquad\qquad\qquad\quad T_1\!: \; \mathbf{skip} \\
\qquad\qquad\qquad\qquad\qquad\quad \{I \wedge in1 \wedge (\neg\, in2 \vee t{=}1 \vee after(door_2))\} \; \mathbf{fi} \\
\{I \wedge in1 \wedge (\neg\, in2 \vee t{=}1 \vee after(door_2))\} \\
PO(CS_1) \otimes (I \wedge in1 \wedge (\neg\, in2 \vee t{=}1 \vee after(door_2))) \\
\ldots \\
\| \\
\ldots \\
\{I \wedge \neg\, in(CS_2)\} \\
entry_2\!: \; door_2\!: \; in2 := true \;\; \{I \wedge \neg\, in(CS_2) \wedge in2\} \\
\qquad\qquad gate_2\!: \; \mathbf{if} \; \neg\, in1 \vee t{=}2 \rightarrow \{I \wedge at(T_2) \wedge in2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge (\neg\, in1 \vee t{=}2 \vee after(door_1))\} \\
\qquad\qquad\qquad\qquad\qquad\quad T_2\!: \; \mathbf{skip} \\
\qquad\qquad\qquad\qquad\qquad\quad \{I \wedge in2 \wedge (\neg\, in1 \vee t{=}2 \vee after(door_1))\} \; \mathbf{fi} \\
\{I \wedge in2 \wedge (\neg\, in1 \vee t{=}2 \vee after(door_1))\} \\
PO(CS_2) \otimes (I \wedge in2 \wedge (\neg\, in1 \vee t{=}2 \vee after(door_1))) \\
\ldots
\end{array}
$$

This proof outline is not interference free. Executing $GEval_{if}(gate_2)$ invalidates $after(door_2)$ (because $after(door_2)=at(GEval_{if}(gate_2))$) without causing $\neg\, in2 \vee t{=}1$ to become *true*. We solve this problem by inserting a statement, $step_2$, between $door_2$ and $gate_2$. This statement causes $after(door_2)$ and $at(gate_2)$ to refer to different control points and makes it impossible for $gate_2$ to be executed when $after(door_2)$ holds.

To ensure that $step_2$ itself does not invalidate $\neg\, in2 \vee t{=}1 \vee after(door_2)$, we implement $step_2$ by the assignment $t := 1$. (The assignment $in2 := false$, which also does not interfere with $\neg\, in2 \vee t{=}1 \vee after(door_2)$, cannot be used because it invalidates $\neg\, in2 \Rightarrow \neg\,(in(CS_2) \vee at(T_2))$ in I.) Similarly, executing $gate_1$ can invalidate $after(door_1)$, and this interference is eliminated by adding a statement $step_1$.

The proof outline that results when $step_1$ is added to S_1 and $step_2$ is added to S_2 is given in Fig. 7.4. It is interference free and is strong enough to establish Mutual Exclusion (7.1) and Entry Non Blocking (7.2).

We next check whether NCS Non Blocking (7.3) is satisfied by the entry and exit protocols of Fig. 7.4. For our program, this property is formalized by:

$$
\begin{array}{l}
at(S) \Rightarrow \Box\neg\,(at(gate_1) \wedge \neg\, enbl(GEval_{if}(gate_1)) \\
\qquad\qquad\qquad \wedge\, (at(GEval_{do}(S_2)) \vee in(NCS_2) \vee after(NCS_2))) \\
at(S) \Rightarrow \Box\neg\,(at(gate_2) \wedge \neg\, enbl(GEval_{if}(gate_2)) \\
\qquad\qquad\qquad \wedge\, (at(GEval_{do}(S_1)) \vee in(NCS_1) \vee after(NCS_1)))
\end{array}
$$

$\{\mathit{true}\}$
$S:\ t, \mathit{in1}, \mathit{in2} := \ldots$
$\{I\!: (\neg\, \mathit{in2} \Rightarrow \neg\, (\mathit{in}(CS_2) \vee \mathit{in}(\mathit{gate}_2))) \ \wedge\ (\neg\, \mathit{in1} \Rightarrow \neg\, (\mathit{in}(CS_1) \vee \mathit{in}(\mathit{gate}_1)))$
$\quad \wedge\ (t=1 \vee t=2)\}$
cobegin
$\{I \wedge \neg\, \mathit{in}(CS_1)\}$
S_1: **do** $\mathit{true} \rightarrow \{I \wedge \neg\, \mathit{in}(CS_1)\}$
$\quad \mathit{entry}_1\!: \ \mathit{door}_1\!: \ \mathit{in1} := \mathit{true} \ \ \{I \wedge \neg\, \mathit{in}(CS_1) \wedge \mathit{in1}\}$
$\qquad \mathit{step}_1\!: \ t := 2 \ \ \{I \wedge \neg\, \mathit{in}(CS_1) \wedge \mathit{in1}\}$
$\qquad \mathit{gate}_1\!:$ **if** $\neg\, \mathit{in2} \vee t=1 \rightarrow \{I \wedge \mathit{at}(T_1) \wedge \mathit{in1}$
$\qquad\qquad \wedge (\neg\, \mathit{in2} \vee t=1 \vee \mathit{after}(\mathit{door}_2))\}$
$\qquad\qquad T_1\!:$ **skip**
$\qquad\qquad \{I \wedge \mathit{in1} \wedge (\neg\, \mathit{in2} \vee t=1 \vee \mathit{after}(\mathit{door}_2))\}$ **fi**
$\quad \{I \wedge \mathit{in1} \wedge (\neg\, \mathit{in2} \vee t=1 \vee \mathit{after}(\mathit{door}_2))\}$
$\quad PO(CS_1) \mathbin{\bigcirc\hspace{-0.72em}\triangle} (I \wedge \mathit{in1} \wedge (\neg\, \mathit{in2} \vee t=1 \vee \mathit{after}(\mathit{door}_2)))$
$\quad \{I \wedge \neg\, \mathit{in}(CS_1)\}$
$\quad \mathit{exit}_1\!:$ **skip**
$\quad \{I \wedge \neg\, \mathit{in}(CS_1)\} \ \ PO(NCS_1) \mathbin{\bigcirc\hspace{-0.72em}\triangle} (I \wedge \neg\, \mathit{in}(CS_1)) \ \ \{I \wedge \neg\, \mathit{in}(CS_1)\}$
od $\{\mathit{false}\}$
||
$\{I \wedge \neg\, \mathit{in}(CS_2)\}$
S_2: **do** $\mathit{true} \rightarrow \{I \wedge \neg\, \mathit{in}(CS_2)\}$
$\quad \mathit{entry}_2\!: \ \mathit{door}_2\!: \ \mathit{in2} := \mathit{true} \ \ \{I \wedge \neg\, \mathit{in}(CS_2) \wedge \mathit{in2}\}$
$\qquad \mathit{step}_2\!: \ t := 1 \ \ \{I \wedge \neg\, \mathit{in}(CS_2) \wedge \mathit{in2}\}$
$\qquad \mathit{gate}_2\!:$ **if** $\neg\, \mathit{in1} \vee t=2 \rightarrow \{I \wedge \mathit{at}(T_2) \wedge \mathit{in2}$
$\qquad\qquad \wedge (\neg\, \mathit{in1} \vee t=2 \vee \mathit{after}(\mathit{door}_1))\}$
$\qquad\qquad T_2\!:$ **skip**
$\qquad\qquad \{I \wedge \mathit{in2} \wedge (\neg\, \mathit{in1} \vee t=2 \vee \mathit{after}(\mathit{door}_1))\}$ **fi**
$\quad \{I \wedge \mathit{in2} \wedge (\neg\, \mathit{in1} \vee t=2 \vee \mathit{after}(\mathit{door}_1))\}$
$\quad PO(CS_2) \mathbin{\bigcirc\hspace{-0.72em}\triangle} (I \wedge \mathit{in2} \wedge (\neg\, \mathit{in1} \vee t=2 \vee \mathit{after}(\mathit{door}_1)))$
$\quad \{I \wedge \neg\, \mathit{in}(CS_2)\}$
$\quad \mathit{exit}_2\!:$ **skip**
$\quad \{I \wedge \neg\, \mathit{in}(CS_2)\} \ \ PO(NCS_2) \mathbin{\bigcirc\hspace{-0.72em}\triangle} (I \wedge \neg\, \mathit{in}(CS_2)) \ \ \{I \wedge \neg\, \mathit{in}(CS_2)\}$
od $\{\mathit{false}\}$
coend
$\{\mathit{false}\}$

Fig. 7.4. Mutual Exclusion (7.1) and Entry Non Blocking (7.2)

We again use Exclusion of Configurations Rule (5.10), this time with the proof outline of Fig. 7.4. Hypothesis (c) would be satisfied by showing that

$$(7.10)\quad at(gate_1) \wedge \neg\, enbl(GEval_{if}(gate_1)) \wedge (at(GEval_{do}(S_2)) \vee in(NCS_2) \vee after(NCS_2)) \wedge I_{PO(S)} \;\Rightarrow\; false$$

$$(7.11)\quad at(gate_2) \wedge \neg\, enbl(GEval_{if}(gate_2)) \wedge (at(GEval_{do}(S_1)) \vee in(NCS_1) \vee after(NCS_1)) \wedge I_{PO(S)} \;\Rightarrow\; false$$

are valid.

Unfortunately, neither is. This should not be surprising, because currently no program variable is changed when a process exits its critical section. Thus, the program variables provide no way for an entry protocol to determine whether a process *is* executing in its critical section or merely *was* executing in its critical section.

The obvious way to remedy this problem is for the exit protocol to change some program variable(s). Deciding exactly which variable to change is guided by unfulfilled obligations (7.10) and (7.11). In the antecedent of (7.10), $(at(GEval_{do}(S_2)) \vee in(NCS_2) \vee after(NCS_2)) \wedge I_{PO(S)}$ effectively selects assertions associated with control points at, in, and after NCS_2. Thus, if each of these assertions implied a predicate P such that $P \wedge \neg\, enbl(GEval_{if}(gate_1)) \Rightarrow false$, then obligation (7.10) would be satisfied.

Two obvious candidates for P are $\neg\, in2$ and $t=1$ because $\neg\, enbl(GEval_{if}(gate_1))$ is $in2 \wedge t \neq 1$. Of the two candidates, we reject $t=1$ because it would be invalidated by executing $step_1$. This leaves $\neg\, in2$ as our choice for P. It is not invalidated by executing S_1. Thus, to make (7.10) valid, we have only to modify $exit_2$ so that assertions in and after NCS_2 can be strengthened by $\neg\, in2$ and modify the initialization so that the assertion before $entry_2$ can be so strengthened. Assignment statement *in2* := *false* in the exit protocol does the job.

Using symmetric reasoning for process S_2, we obtain the proof outline of Fig. 7.5. Variable *t* can be initialized to either 1 or 2. The proof outline is valid and makes (7.10) and (7.11) valid, which means our protocol now satisfies NCS Non Blocking (7.3). It is wise to check that Mutual Exclusion (7.1) and Entry Non Blocking (7.2) are still satisfied as well. They are.

Finally, we check that Exit Non Blocking (7.4) is satisfied by the program of Fig. 7.5. To do so, we must verify that *S* satisfies:

$$(7.12)\quad at(S) \Rightarrow \Box \neg ((at(exit_1) \wedge \neg\, enbl(exit_1)) \vee (at(exit_2) \wedge \neg\, enbl(exit_2)))$$

Because each $exit_i$ is implemented by a single unconditional atomic action, from definition (3.7) of *enbl* we have

$$enbl(exit_1) = true$$
$$enbl(exit_1) = true$$

and therefore, by Temporal Logic, (7.12) holds.

$\{true\}$
$S:\ t, in1, in2 := 1, false, false$
$\quad \{t=1 \wedge \neg\, in1 \wedge \neg\, in2 \wedge$
$\quad\ I:\ (\neg\, in2 \Rightarrow \neg\,(in(CS_2) \vee in(gate_2)))\ \wedge\ (\neg\, in1 \Rightarrow \neg\,(in(CS_1) \vee in(gate_1)))$
$\qquad \wedge\ (t=1 \vee t=2)\}$
$\quad \mathbf{cobegin}$
$\quad \{I \wedge \neg\, in(CS_1) \wedge \neg\, in1\}$
$\quad S_1:\ \mathbf{do}\ true \rightarrow \{I \wedge \neg\, in(CS_1) \wedge \neg\, in1\}$
$\qquad entry_1:\ door_1:\ in1 := true\ \ \{I \wedge \neg\, in(CS_1) \wedge in1\}$
$\qquad\qquad step_1:\ t := 2\ \ \{I \wedge \neg\, in(CS_1) \wedge in1\}$
$\qquad\qquad gate_1:\ \mathbf{if}\ \neg\, in2 \vee t=1 \rightarrow \{I \wedge at(T_1) \wedge in1$
$\qquad\qquad\qquad\qquad \wedge (\neg\, in2 \vee t=1 \vee after(door_2))\}$
$\qquad\qquad\qquad T_1:\ \mathbf{skip}$
$\qquad\qquad\qquad \{I \wedge in1 \wedge (\neg\, in2 \vee t=1 \vee after(door_2))\}\ \mathbf{fi}$
$\qquad \{I \wedge in1 \wedge (\neg\, in2 \vee t=1 \vee after(door_2))\}$
$\qquad PO(CS_1) \owedge (I \wedge in1 \wedge (\neg\, in2 \vee t=1 \vee after(door_2)))$
$\qquad \{I \wedge \neg\, in(CS_1)\}$
$\qquad exit_1:\ in1 := false\ \ \{I \wedge \neg\, in(CS_1) \wedge \neg\, in1\}$
$\qquad PO(NCS_1) \owedge (I \wedge \neg\, in(CS_1) \wedge \neg\, in1)$
$\qquad \{I \wedge \neg\, in(CS_1) \wedge \neg\, in1\}$
$\qquad \mathbf{od}\ \ \{false\}$
$\quad \|$
$\quad \{I \wedge \neg\, in(CS_2) \wedge \neg\, in2\}$
$\quad S_2:\ \mathbf{do}\ true \rightarrow \{I \wedge \neg\, in(CS_2) \wedge \neg\, in2\}$
$\qquad entry_2:\ door_2:\ in2 := true\ \ \{I \wedge \neg\, in(CS_2) \wedge in2\}$
$\qquad\qquad step_2:\ t := 1\ \ \{I \wedge \neg\, in(CS_2) \wedge in2\}$
$\qquad\qquad gate_2:\ \mathbf{if}\ \neg\, in1 \vee t=2 \rightarrow \{I \wedge at(T_2) \wedge in2$
$\qquad\qquad\qquad\qquad \wedge (\neg\, in1 \vee t=2 \vee after(door_1))\}$
$\qquad\qquad\qquad T_2:\ \mathbf{skip}$
$\qquad\qquad\qquad \{I \wedge in2 \wedge (\neg\, in1 \vee t=2 \vee after(door_1))\}\ \mathbf{fi}$
$\qquad \{I \wedge in2 \wedge (\neg\, in1 \vee t=2 \vee after(door_1))\}$
$\qquad PO(CS_2) \owedge (I \wedge in2 \wedge (\neg\, in1 \vee t=2 \vee after(door_1)))$
$\qquad \{I \wedge \neg\, in(CS_2)\}$
$\qquad exit_2:\ in2 := false\ \ \{I \wedge \neg\, in(CS_2) \wedge \neg\, in2\}$
$\qquad PO(NCS_2) \owedge (I \wedge \neg\, in(CS_2) \wedge \neg\, in2)$
$\qquad \{I \wedge \neg\, in(CS_2) \wedge \neg\, in2\}$
$\qquad \mathbf{od}\ \ \{false\}$
$\quad \mathbf{coend}$
$\{false\}$

Fig. 7.5. Exit Protocol for NCS Non Blocking (7.3)

This completes the derivation of the solution to the mutual exclusion problem. Fig. 7.5 contains a protocol that satisfies Mutual Exclusion (7.1), Entry Non Blocking (7.2), NCS Non Blocking (7.3), and Exit Non Blocking (7.4).

Reviewing the Method

The derivation described above is based on repeated application of what is really a simple method:

(7.13) **Safety Property Methodology.** If a program does not satisfy $Init \Rightarrow \Box Etern$:

(1) Construct a valid proof outline for that program.

(2) Identify assertions that must be strengthened in order to prove that $Init \Rightarrow \Box Etern$ is satisfied.

(3) Modify the program and proof outline so that those assertions are strengthened. $\Box$

Of course, step (3) requires creativity—especially since stronger assertions are more likely to be interfered with. Therefore, strengthening an assertion in some process S_i is typically a two-phase process. First, S_i is modified ignoring other processes. This results in a proof outline that is valid in isolation and has the stronger assertions. Then, that proof outline is considered in the context of the concurrent program and any interference is eliminated.

For the mutual exclusion problem, we were given a program skeleton containing some unspecified operations and asked to refine those operations to make certain safety properties hold. The skeleton imposed constraints on the solution, and these constraints simplified our task by restricting possible design choices. Additional constraints accumulated as the derivation proceeded. Each safety property, once satisfied, imposed constraints on subsequent modifications to the entry and exit protocols. For example, maintaining a valid proof outline from which Mutual Exclusion (7.1) could be proved constrained modifications to the entry protocol so that Entry Non Blocking (7.2) could be proved.

7.2. Concurrent Reading While Writing

We next attack a problem that arises when shared variables are used for communication in a concurrent program. Suppose one process reads from these variables by executing a non-atomic operation *READ*; the other writes to them by executing a non-atomic operation *WRITE*. Desired is a protocol to synchronize *READ* and *WRITE* so that values seen by reader reflect the state of the shared variables either before a concurrent write has started or after it has completed.

We derive a statement R to control each *READ* operation and a statement W to control each *WRITE* operations. The problem description requires that R not terminate with values reflecting an in-progress *WRITE*. This is a safety property and is specified in Temporal Logic as

(7.14) $Init \Rightarrow \Box(after(R) \Rightarrow \neg BD)$

where derived term *BD* (abbreviating "Bad Data") is satisfied if the last *READ* that started overlapped with execution of *WRITE*:

$$BD:\ \begin{cases} false \quad \text{if } at(READ) \\ in(READ) \wedge in(WRITE) \quad \text{if } \neg at(READ) \wedge \neg def_{\Theta} \\ (in(READ) \wedge in(WRITE)) \ \vee\ \Theta BD \quad \text{if } \neg at(READ) \wedge def_{\Theta} \end{cases}$$

Any valid proof outline having a precondition implied by *Init* and in which $post(R)$ implies $\neg BD$ is sufficient for proving that (7.14) is satisfied, due to Safety Rule (5.9). Thus, ensuring satisfaction of (7.14) is equivalent to filling out the bodies of *R* and *W* in the following proof outline. Note that assertions not pertaining to the proof of (7.14) are being ignored and have been omitted.

(7.15) $\{Init\}$
cobegin
...
R: ... *READ* ...
$\{\neg BD\}$
...
||
...
W: ... *WRITE* ...
...
coend

One way to ensure that $\neg BD$ holds when *R* terminates is to prevent execution of *READ* while *WRITE* is executing, and vice versa. This, however, can cause execution of *WRITE* to be delayed—something that is not always desirable. For example, suppose the digits of a multi-digit clock are each implemented by a separate shared variable. If the clock is advanced by a process that periodically executes *WRITE* to store new values in these variables, then the clock's correctness depends not only on what values are written but on when those values are written. Delaying *WRITE* compromises the clock's accuracy.

WRITE will never be delayed if *W* contains no conditional atomic actions or loops. We therefore adopt this additional constraint, ruling out exclusion-based readers/writers protocols.

In order to proceed with the development of (7.15), we first construct a valid proof outline for *R* in isolation. The body of *R* is simply *READ*—there is no justification for including anything else. Moreover, because $\neg BD$ holds when $at(R)$ does, it is easy to construct a proof outline with the desired postcondition. $PO(READ)$ is a proof outline for *READ* having *true* for every assertion.

(7.16) $\{\neg BD\}$
R: $PO(READ) \oslash \neg BD$
$\{\neg BD\}$

To include this proof outline in a **cobegin**, however, requires that W not interfere with (7.16). Unfortunately, it does. Execution of atomic actions in *WRITE* invalidate conjunct $\neg BD$ in all assertions except $pre(READ)$.

To eliminate this interference, we postulate a predicate p such that for every atomic action $\alpha \in \mathcal{A}(WRITE)$, the following holds:

(7.17) $pre(\alpha) \Rightarrow p$

We then weaken those assertions that formerly were invalidated. The result is the following modification of (7.16).

$\{\neg BD\}$
R: $PO(READ) \oslash (p \vee \neg BD)$
$\{p \vee \neg BD\}$

A problem with this proof outline is that $post(R)$ is now weaker than desired. Moreover, once $\neg BD$ has been invalidated, waiting can never make $\neg BD$ hold again (due to the third clause in the definition of BD), so blocking the process containing R cannot be used to strengthen $post(R)$.

A loop can also be used to strengthen an assertion, because **do** Rule (6.7) has as its postcondition the conjunction of its precondition and another predicate, the guards negated. This suggests that *READ* be made the body of a loop with $p \vee \neg BD$ the loop invariant and p the guard, thereby allowing the postcondition of the loop to be $\neg BD$ because it is implied by $(p \vee \neg BD) \wedge \neg p$. We allow concurrent reading while writing, but prevent data read during a *WRITE* from becoming visible outside of R.

(7.18) $\{I\colon\ p \vee \neg BD\}$
R: **do** $p \rightarrow$ $\{I \wedge \neg BD\}$
$PO(READ) \oslash I$
$\{I\}$
od
$\{\neg BD\}$

An easy way to discharge obligation (7.17) is by introducing a program variable p and bracketing *WRITE* with assignments to p. This is done in the following proof outline fragment, where $PO(WRITE)$ has *true* for each of its assertions.

$$
\begin{array}{l}
W\!: \; p := \mathit{true} \\
\quad\; PO(\mathit{WRITE}) \circledwedge p \\
\quad\; p := \mathit{false}
\end{array}
$$

Unfortunately, when embedded in the **cobegin** of (7.15), final assignment $p := \mathit{false}$ interferes with I in all assertions of (7.18) except $\mathit{pre}(\mathit{READ})$. To solve this problem, we postulate a predicate q satisfying

$$\mathit{pre}(p := \mathit{false}) \Rightarrow q,$$

and use q to weaken those assertions in $PO(R)$ that could be invalidated by executing $p := \mathit{false}$. The revised proof outline for R follows. In it, the weaker loop guard, $p \vee q$, is needed in order to be able to infer $\neg BD$ when the loop terminates, given the weaker loop invariant.

$$
\begin{array}{lll}
\{I\!: \; p \vee q \vee \neg BD\} & & \\
R\!: \; \mathbf{do} \;\; p \vee q \rightarrow & \{I \wedge \neg BD\} \\
 & PO(\mathit{READ}) \circledwedge I \\
 & \{I\} \\
\quad\;\; \mathbf{od} & \\
\{\neg BD\} &
\end{array}
$$

The revised protocol for W is:

$$
\begin{array}{l}
W\!: \; p := \mathit{true} \\
\quad\; PO(\mathit{WRITE}) \circledwedge p \\
\quad\; q := \mathit{true} \;\; \{q\} \\
\quad\; p := \mathit{false}
\end{array}
$$

We have succeeded in constructing proof outlines for R and W that are interference free, satisfy the constraints in (7.15), and satisfy the constraints that ensure *WRITE* is not delayed. However, our protocol for synchronizing *READ* has two problems:

(i) Once q is set to *true* in W, the **do** in R loops forever. Useful computation by the process containing R then becomes impossible.

(ii) A suitable initialization must be devised so that loop invariant $p \vee q \vee \neg BD$ will hold at the start of the **do**.

Although infinite looping of the **do** in R cannot cause (7.14) to be violated, it can be a problem when proving termination and other liveness properties. Non-terminating loops can prevent a "bad thing" from happening, but in so doing might also prevent "good things" from happening. Thus, when liveness properties may be of interest, use of such non-terminating loops is rarely a good practice.

The loop in R will terminate if $\neg(p \vee q)$ holds when its guard evaluation action is executed. W establishes $\neg p$ before exiting, but cannot also establish $\neg q$ without causing interference with I. Therefore, in order make $\neg(p \vee q)$ hold, we investigate possible places in R to add an assignment that will establish $\neg q$.

The assignment must occur in the body of the **do** or else it will not be executed after the loop has started (and when it would be needed). Also, looking at the assertions in the body of the **do**, we see that the new assignment must leave I *true*. Thus, execution of $q := false$ must occur in a state where $p \vee \neg BD$ holds, since $p \vee \neg BD$ implies I. By definition, $\neg BD$ holds when $at(READ)$ does, so we place the assignment immediately before *READ*, obtaining the following valid proof outline.

$$
\begin{array}{l}
\{I\colon\ p \vee q \vee \neg BD\} \\
R\colon\ \textbf{do}\ \ p \vee q \rightarrow\ \ \{I\} \\
\qquad\qquad\qquad\quad q := false\ \ \{I \wedge \neg BD\} \\
\qquad\qquad\qquad\quad PO(READ) \otimes I \\
\qquad\qquad\qquad\quad \{I\} \\
\quad\ \textbf{od} \\
\{\neg BD\}
\end{array}
$$

Now, however, $q := false$ in R interferes with $pre(p := false)$ (which is q) in the proof outline for W. Recall, having q be a conjunct of $pre(p := false)$ eliminated interference by $p := false$ with I in assertions of the proof outline for R. Thus, provided $pre(p := false)$ remains strong enough for $NI(p := false, I)$ to be valid, we can use disjunct $\neg BD$ to weaken $pre(p := false)$, because executing $q := false$ establishes $at(READ)$, which implies $\neg BD$, and executing $p := false$ in a state satisfying $\neg BD$ does not invalidate $\neg BD$ (hence I). Here is the revised proof outline for W:

$$
\begin{array}{l}
W\colon\ p := true \\
\qquad PO(WRITE) \otimes p \\
\qquad q := true\ \ \{q \vee \neg BD\} \\
\qquad p := false
\end{array}
$$

Finally, we devise an initialization that establishes loop invariant $p \vee q \vee \neg BD$. Assigning *true* to either p or q will establish I. We choose an assignment to q, so that execution of R can terminate without W executing. The final protocol appears as Fig. 7.6.

8. Historical Notes

The content of this chapter is derived from [25], a forthcoming text on concurrent programming.

Hoare was the first to propose a logic for reasoning about programs [12]. His logic is based on a program verification technique described in [9]. Formulas of the logical system in [12] were of the form $P\ \{S\}\ Q$, although this notation has since been displaced by $\{P\}\ S\ \{Q\}$, which is suggestive of assertions being viewed as comments.

Hoare was also the first to address the design of a programming logic for concurrent programs. In [13], he extended the logic of [12] with inference rules for parallel composition of processes that synchronize using conditional critical

$\{Init\}$
cobegin
...
$R:\ q := true\ \ \{I\!:\ p \vee q \vee \neg BD\}$
do $p \vee q \rightarrow\ \ \{I\}$
$q := false\ \ \{I \wedge \neg BD\}$
$PO(READ) \owedge I$
$\{I\}$
od
$\{\neg BD\}$
...
||
...
$W:\ p := true$
$PO(WRITE) \owedge p$
$q := true\ \ \{q \vee \neg BD\}$
$p := false$
...
coend

Fig. 7.6. Concurrent Reading While Writing

regions.

Interference freedom and the first complete programming logic for partial correctness was developed by Owicki in a Ph.D. thesis [20] supervised by Gries [21]. The work extends Hoare's logic of triples to handle concurrent programs that synchronize and communicate using shared variables. Lamport, working independently, developed an idea (monotone assertions) similar to interference freedom as part of a more general method for proving both safety and liveness properties of concurrent programs [16]. Unfortunately, the method of [16] is described in terms of the flowchart representation of a concurrent program, and this probably accounted for its failure to attract the attention it deserved.

Lamport's Generalized Hoare Logic (GHL) is a Hoare-style programming logic for reasoning about concurrent programs, motivated by the success of the Owicki-Gries logic [17]. In contrasting the logic of [21] and GHL, the first significant difference concerns the role of proof outlines. The Owicki-Gries logic appears to be based on triples rather than proof outlines. However, this is deceptive. Had interference freedom been formalized in the logic, the need for treating proof outlines (in addition to triples) as formulas would probably have become apparent. GHL is based on proof outlines, making formulas a bit more complex but allowing a simple inference rule for **cobegin**.

The second significant difference between the Owicki-Gries logic and GHL is the use of control predicates. Instead of control predicates, the Owicki-Gries logic sometimes requires that additional variables, called auxiliary variables, be added to a program when constructing a proof. (These variables can be thought of as derived terms whose value is computed by the program rather than by a definition.)

The final distinction between the Owicki-Gries logic and GHL concerns the class of properties that can be proved. The Owicki-Gries logic was intended for proving three types of properties: partial correctness, mutual exclusion, and deadlock freedom. The logic could have been extended for proving safety properties, although doing so is subtle. GHL was originally intended for proving safety properties, even for programs where all of the atomic actions have not been specified.

Proof Outline Logic is based on GHL. The programming notation axiomatized by Proof Outline Logic has additional control structures but less flexibility about atomicity. Second, GHL cannot be used to prove safety properties defined in terms of sequences of past states; Proof Outline Logic can, because Θ can appear in its assertions. Finally, while the notation used for proof outlines in GHL is more expressive than the notation our Proof Outline Logic employs, our notation is closer to conventional annotated programs.

Our assignment statement and Assignment Axiom (6.2) are based on [10]; the **if** and **do** statements are from [6]. The angle bracket notation for specifying synchronization was invented by Lamport and formalized in [17]. However, the notation was popularized by Dijkstra, with the earliest published use in [8]. The idea that an **if** statement with no *true* guard should delay until some guard becomes *true* originated with [7].

Most methods that use Hoare-style programming logics for verifying safety properties involving past states employ variables to record relevant aspects of a computation's history. One approach is to allow such variables to appear in assertions, but not to permit them in program statements [26, 27]. A more popular approach is to augment the program with assignments to auxiliary variables that encode whatever history information is of interest. The auxiliary variables are used in a formal statement of the property as well as in a proof outline to establish that the augmented program satisfies that property. To infer that the original program also satisfies the property in question, it is asserted that the auxiliary variables can be deleted because they have no affect on program execution. Knowing just when such auxiliary variables can be deleted is rather a subtle question, however.

Although many who have written about programming logics use proof outlines, few have formalized them and even fewer have done so correctly. One of the earlier (correct) formalizations appears in [2]; a natural deduction programming logic of proof outlines is presented in [4].

Pnueli was the first to use a temporal logic for reasoning about concurrent programs [23]. Interpretations like the anchored sequences used here were first introduced in [18] and later used used in [19].

Safety properties were first defined by Lamport in [16]. The method given in [16] for proving that a program satisfies such a property is based on finding a suitable invariant. This use of invariants, however, did not originate with

Lamport. For safety properties concerning the control state (e.g. mutual exclusion, readers/writers), proofs that use invariants appear in [3, 5, 11]. For safety properties involving relationships among the control state and program variables, proof methods based on finding an invariant are discussed in [1] and [15].

Safety Rule (5.9) is based on a meta-theorem of Lamport's Generalized Hoare Logic [17]. Exclusion of Configurations Rule (5.10) is a generalization of a method that is used in [21] for proving that a program is free from deadlock and in [7] for proving mutual exclusion.

There is an extensive literature on the mutual exclusion problem. See [24] for a summary of various protocols and their properties. The solution developed in §7.1 is based on [22]. The protocol is usually presented operationally; the derivation in §7.1 is new. The reading while writing protocol in §7.2 is a variation of one developed by Jayanti [14]. Our variant is a bit simpler; we discovered it while attempting to provide an assertional derivation (and proof) of Jayanti's protocol.

Acknowledgements

I am grateful to David Gries, Leslie Lamport, Amir Pnueli, Bard Bloom, Limor Fix, and Scott Stoller for discussions and criticisms as this work developed. Cornell graduate students taking CS613 have been kind enough to provide feedback on the presentation and content of the approach described here. Funding for this work has been provided by the National Science Foundation, Office of Naval Research, Air Force Office of Scientific Research, and the Defense Advanced Research Projects Agency.

References

1. Ashcroft, E. Proving Assertions about Parallel Programs. *Journal of Computer and System Sciences 10*, 1 (Feb. 1975), 110-135.
2. Ashcroft, E. Program verification tableaus. Technical Report CS-76-01. University of Waterloo, Waterloo, Ontario, Canada, Jan. 1976.
3. Brinch Hansen, P. A comparison of two synchronizing concepts. *Acta Informatica 1*,3 (1972), 190-199.
4. Constable, R.L., and M.J. O'Donnell. *A Programming Logic*. Winthrop Publishers, Cambridge, Mass., 1978.
5. Dijkstra, E.W. Hierarchical ordering of sequential processes. *Acta Informatica 1*, (1971), 115-138.
6. Dijkstra, E.W. Guarded commands, nondeterminacy and formal derivation of programs. *CACM 18*, 8 (Aug. 1975), 453-457.
7. Dijkstra, E.W. A personal summary of the Gries-Owicki theory. EWD554, in *Selected Writings on Computing: A Personal Perspective*, E.W. Dijkstra, Springer Verlag, New York, 1982.
8. E.W. Dijkstra. On making solutions more and more fine-grained. EWD622, in *Selected Writings on Computing: A Personal Perspective*, E.W. Dijkstra, Springer Verlag, New York, 1982.
9. Floyd, R.W. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics 19*, 1967, 19-31.
10. Gries, D. The multiple assignment statement. *IEEE Trans. on Software Engineering SE-4*, 2 (March 1978), 87-93.
11. Habermann, A.N. Synchronization of communicating processes. *CACM 15*, 3 (March 1972), 171-176.
12. Hoare, C.A.R. An axiomatic basis for computer programming. *CACM 12*, 10 (Oct.

1969), 576-580.
13. Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, C.A.R. Hoare and R. Perrot (eds.), Academic Press, New York, 1972.
14. Jayanti, P., A. Sethi, and E. Lloyd. Minimal Shared Information for Concurrent Reading and Writing, Technical report 90-13, Department of Computer and Information Sciences, University of Delaware, Newark, DE, 1990.
15. Keller, R.M. Formal verification of parallel programs. *CACM 19*, 7 (July 1976), 371-384.
16. Lamport, L. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering SE-3*, 2 (March 1977), 125-143.
17. Lamport, L. The "Hoare Logic" of concurrent programs. *Acta Informatica 14* (1980), 21-37.
18. Manna, Z., and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, J.W. de Bakker, W.P. de Roever, and G. Rozenberg (eds.), Lecture Notes in Computer Science Volume 354, Springer-Verlag, New York, 1989, 201-284.
19. Manna, Z., and A. Pnueli. *Temporal Logic of Reactive Systems*. Springer-Verlag, New York, 1991.
20. Owicki, S.S. *Axiomatic Proof Techniques for Parallel Programs*. Ph.D. Thesis, Computer Science Department, Cornell University, Ithaca, New York, 1975.
21. Owicki, S.S., and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica 6* (1976), 319-340.
22. Peterson, Gary L. Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3 (June 1981), 115-116.
23. Pnueli, A. The temporal logic of programs. *Proc. Eighteenth Annual Symposium on Foundations of Computer Science*, ACM, Providence, R.I., Nov. 1977, 46-57.
24. Raynal, M. *Algorithms for Mutual Exclusion*. The MIT Press, Cambridge, Mass., 1986.
25. Schneider, F.B. *On Concurrent Programming*. Springer Verlag, Heidelberg, 1995. To appear.
26. Soundararajan, N. A proof technique for parallel programs. *Theoretical Computer Science 31* (1983), 13-29.
27. Zwiers, J. and W.P. de Roever. Predicates are predicate transformers: A unified compositional theory of concurrency. *Proc. Eighth Symposium on Principles of Distributed Computing*, ACM, Edmonton, Alberta, Canada, August 1989, 265-280.