Reasoning about Programs by Exploiting the Environment*

Limor Fix

Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

October 25, 1994

ABSTRACT

A method for making aspects of a computational model explicit in the formulas of a programming logic is given. The method is based on a new notion of environment—an environment augments the state transitions defined by a program's atomic actions rather than being interleaved with them. Two simple semantic principles are presented for extending a programming logic in order to reason about executions feasible in various environments. The approach is illustrated by (i) discussing a new way to reason in TLA and Hoare-style programming logics about real-time and by (ii) deriving TLA and the first Hoare-style proof rules for reasoning about schedulers.

*This material is based on work supported in part by the Office of Naval Research under contract N00014-91-J-1219, AFOSR under proposal 93NM312, the National Science Foundation under Grant No. CCR-8701103, and DARPA/NSF Grant No. CCR-9014363. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies. Limor Fix is also supported, in part, by a Fullbright post-doctoral award.

1. Introduction

What behaviors of a concurrent program are possible may depend on the scheduler, instruction timings, and other aspects of the environment in which that program executes. For example, consider the program of Figure 1.1. Process P_1 executes an atomic action that sets y to 1 followed by one that sets y to 2. Concurrently, process P_2 executes an atomic action that sets y to 3. If all behaviors of this concurrent program were possible, then the final value of y would be 2 or 3. The environment, however, may rule out certain behaviors.

- Suppose P_1 has higher-priority than P_2 and the environment selects between executable atomic actions by using a priority scheduler. Behaviors in which actions of P_2 execute before those of P_1 are now infeasible, and the final value of y cannot be 2.
- Suppose the environment uses a first-come first-served scheduler to select between executable atomic actions. Behaviors in which P_2 executes after the second action of P_1 are now infeasible, and the final value of y cannot be 3.

Thus, changing the environment can affect what properties a program satisfies.

Programming logics usually axiomatize program behavior under certain assumptions about the environment. Logics to reason about real-time, for example, axiomatize assumptions about how time advances while the program executes. These assumptions abstract the effects of the scheduler and the execution times of various atomic actions. A logic to reason about the consequences of resource constraints would similarly have to axiomatize assumptions about resource availability.

If assumptions about an environment are made when defining a programming logic, then changes to the environment may require changes to the logic. Previously feasible behaviors could become infeasible when the assumptions are strengthened; a logic for the original environment would then be incomplete for this new environment. Weakening the assumptions could add feasible behaviors; the logic for the original environment would then become unsound. For example, any of the programming logics for shared-memory concurrency (e.g. [OG76]) could be used to prove that program of Figure 1.1 terminates with $y=2$ or $y=3$. But, these logics must be changed to prove that $y=2$ necessarily holds if a first-come first served scheduler is being used or that $y=3$ necessarily holds if a priority scheduler is used. As another example, termination of the program in Figure 1.2 depends on whether unfair behaviors are feasible. (Usually they are not.) Logics, like the temporal logic of [MP89], that assume a fair scheduler become unsound when this assumption about the environment is relaxed.

cobegin

P_1: $y := 1$; $y := 2$

//

P_2: $y := 3$

coend

Figure 1.1. A concurrent program

cobegin

P_1: $b := false$

//

P_2: **do** $b \rightarrow$ **skip od**

coend

Figure 1.2. Termination with fairness

This paper explores the design of programming logics in which assumptions about the environment can be given explicitly. Such logics allow us to prove that all feasible behaviors of a program satisfy a property, where the characterization of what is feasible is now explicit and subject to change. We give two semantic principles—program reduction and property reduction—for extending a programming logic so that explicit assumptions about an environment can be exploited in reasoning. These principles allow extant programming logics to be extended for reasoning about the effects of various fairness conditions, schedulers, and models of real-time; a new logic need not be defined every time a new model of computation is postulated. We illustrate the application of our two principles using TLA [L91] and a Hoare-style Proof Outline Logic [S94]. In TLA, programs and properties are both represented using a single language; in Proof Outline Logic these two languages are distinct.

The remainder of this paper is structured as follows. In section 2, our program and property reduction principles are derived. Then, in section 3, program reduction is applied to TLA. In section 4, property reduction is used to drive an extension to a Hoare-style logic. Section 5 puts this work in context; section 6 is a conclusion. The appendix contains a proof of relative completeness for the extended Hoare-style logic.

2. Formalizing and Exploiting the Environment

A programming logic comprises deductive system for verifying that a given program satisfies a property of interest. We write $\langle S, \Psi \rangle \in Sat$ to denote that a program S satisfies a property Ψ; each programming logic will have its own syntax for saying this. In any given programming logic, a *program language* is used to specify S and a *property language*, perhaps identical to the program language, is used to specify Ψ.

Usually, both the program S and the property Ψ define sets of behaviors, where a *behavior* is a mathematical object that encodes a sequence of state transitions resulting from program execution, and a *state* is a mapping from variables to values. Notice that

- the set $[\![S]\!]$ of behaviors for a program S constrains only the values of program variables[1], and
- the set $[\![\Psi]\!]$ of behaviors for a property Ψ may also constrain the values of variables that are not program variables.

[1]Program variables include all those declared explicitly in the program as well as others, like program counters and message buffers, concerning aspects of the state implicitly involved in executing the program.

A program S satisfies a property Ψ exactly when all of the behaviors of S are behaviors permitted by Ψ:

$$\langle S, \Psi \rangle \in \mathit{Sat} \quad \text{if and only if} \quad [\![S]\!] \subseteq [\![\Psi]\!] \tag{2.1}$$

The environment in which a program executes defines a property too. This property contains any behavior that is not precluded by one or another aspect of the environment. For example, a priority scheduler precludes behaviors in which atomic actions from low-priority processes are executed instead of those from high-priority processes. As another example, the environment might define the way a distinguished variable *time* (say) changes in successive states, taking into account the processor speed for each type of atomic action.

For E the property defined by the environment, the *feasible behaviors* of a program S under E are those behaviors of S that are also in E: $[\![S]\!] \cap [\![E]\!]$. A program S satisfies a property Ψ under an environment E, denoted $\langle S, E, \Psi \rangle \in \mathit{ESat}$, if and only if every feasible behavior of S under E is in Ψ:

$$\langle S, E, \Psi \rangle \in \mathit{ESat} \quad \text{if and only if} \quad ([\![S]\!] \cap [\![E]\!]) \subseteq [\![\Psi]\!] \tag{2.2}$$

Thus, a deductive system for verifying $\langle S, E, \Psi \rangle \in \mathit{ESat}$ would permit us to prove properties of programs under various assumptions about schedulers, execution times, and so on.

Defining a separate logic to prove $\langle S, E, \Psi \rangle \in \mathit{ESat}$ is not always necessary if a logic to prove $\langle S, \Psi \rangle \in \mathit{Sat}$ is available. For properties Φ and Ψ, let property $\Phi \cap \Psi$ be $[\![\Phi]\!] \cap [\![\Psi]\!]$, and let property $\Phi \cup \overline{\Psi}$ be $[\![\Phi]\!] \cup \overline{[\![\Psi]\!]}$ where $\overline{[\![\Psi]\!]}$ denotes the complement of $[\![\Psi]\!]$. Then, one reduction from *ESat* to *Sat* is derived as follows.

$\langle S, E, \Psi \rangle \in \mathit{ESat}$
iff «definition (2.2) of *ESat*»
$([\![S]\!] \cap [\![E]\!]) \subseteq [\![\Psi]\!]$
iff «definition (2.1) of *Sat*»
$\langle S \cap E, \Psi \rangle \in \mathit{Sat}$

Thus we have:

Program Reduction: $\langle S, E, \Psi \rangle \in \mathit{ESat}$ if and only if $\langle S \cap E, \Psi \rangle \in \mathit{Sat}$. (2.3)

Program Reduction is useful if the logic for $\langle S, \Psi \rangle \in \mathit{Sat}$ has a program language that is closed under intersection with the language used to define environments. Section 3 shows this to be the case for Lamport's TLA; it is also the case for most other temporal logics.

A second reduction from *ESat* to *Sat* is based on using the environment to modify the property (rather than the program).

$\langle S, E, \Psi \rangle \in \mathit{ESat}$
iff «definition (2.2) of *ESat*»
$([\![S]\!] \cap [\![E]\!]) \subseteq [\![\Psi]\!]$
iff «set theory»
$[\![S]\!] \subseteq ([\![\Psi]\!] \cup \overline{[\![E]\!]})$
iff «definition (2.1) of *Sat*»
$\langle S, \Psi \cup \overline{E} \rangle \in \mathit{Sat}$

This proves:

Property Reduction: $\langle S, E, \Psi \rangle \in ESat$ if and only if $\langle S, \Psi \cup \overline{E} \rangle \in Sat$. (2.4)

Property reduction imposes no requirement on the program language, but does require that the property language be closed under union with the complement of properties that might be defined by environments. An example of a logic whose property language satisfies this closure condition is CTL* [EH86]. Linear-time temporal logics, on the other hand, do not satisfy this closure condition—$\overline{P}$ is not equivalent to $\neg P$.

When neither reduction principle applies, then we can reason about the effects of an environment by extending the logic being used to establish $(S, \Psi) \in Sat$. Extensions to the program language allow Program Reduction to be applied; extensions to the property language allow Property Reduction to be applied. Section 4 illustrates how this might be done, by extending the property language of a Hoare-style logic called Proof Outline Logic.

3. Environments for TLA

The Temporal Logic of Actions (TLA) is a linear-time temporal logic in which programs and properties are represented as formulas. Thus, the program language and property language of TLA are one and the same. This single language includes the usual propositional connectives, and the TLA formula $F \wedge G$ defines a property that is the intersection of the properties defined by F and G. TLA is, therefore, an ideal candidate for Program Reduction.

3.1. TLA Overview

A TLA *state predicate* is a predicate logic formula over some variables.[2] The usual meaning is ascribed to $s \models p$ for a state s and a state predicate p: when each variable v in p is replaced by its value $s(v)$ in state s, the resulting formula is equivalent to *true*. For example, in a state s that maps y to 14 and z to 22, $s \models y+1<z$ holds because $s(y)+1<s(z)$ equals $14+1<22$, which is equivalent to *true*.

A TLA *action* is a predicate logic formula over unprimed variables and primed variables. Actions are interpreted over pairs of states. The unprimed variables are evaluated in the first state s of the pair (s, t) and the primed variables are evaluated, as if unprimed, in the second state t of the pair. For example, if $s(y)$ equals 13 and $t(y)$ equals 16 then $(s, t) \models y+1<y'$ holds because $s(y)+1<t(y)$ is equal to $13+1<16$, or, *true*.

In order to facilitate writing actions that are invariant under stuttering, TLA provides an abbreviation. For action A and list $\overline{x}$ of variables $x_1, x_2, ..., x_n$, the action[3] $[A]_{\overline{x}}$ is satisfied by any pair (s, t) of states such that $(s, t) \models A$ or the values of the x_i are unchanged between s and t. Writing $\overline{x}'$ to denote the result of priming every variable in $\overline{x}$, we get:

$[A]_{\overline{x}}$: $A \vee \overline{x}=\overline{x}'$

TLA actions define state transitions. Therefore, they can be used to describe the next-state relation of a concurrent program, a single sequential process, or any piece thereof. For this purpose, it is useful to define a state predicate satisfied by any state from which transition is possible due to an action A. That state predicate, $Enbl(A)$, is defined by:

[2]We assume that variable names do not contain the character "′" (prime).

[3]TLA actually allows subscript $\overline{x}$ to be an arbitrary state function whose value will remain unchanged.

$s \models Enbl(A)$ if and only if Exists t: $(s, t) \models A$

Each formula Φ of TLA defines a property $[\![\Phi]\!]$, which is the set of behaviors that satisfy Φ, where a behavior is represented by an infinite sequence of states. Let σ be a behavior $s_0 s_1 \ldots$, let p be a state predicate, let A be an action, and let $\bar{x}$ be a list of variables. The syntax of the *elementary formulas* of TLA, along with the property defined by each, is:

$$\begin{array}{ll} \sigma \in [\![p]\!] & \text{iff } s_0 \models p \\ \sigma \in [\![\Box[A]_{\bar{x}}]\!] & \text{iff For all } i, i \geq 0\text{: } (s_i, s_{i+1}) \models [A]_{\bar{x}} \end{array}$$

The remaining formulas of TLA are formed from these, as follows. Let Φ and Ψ be elementary TLA formulas or arbitrary TLA formulas.

$$\begin{array}{ll} \sigma \in [\![\neg\Phi]\!] & \text{iff } \sigma \notin [\![\Phi]\!] \\ \sigma \in [\![\Phi \vee \Psi]\!] & \text{iff } \sigma \in ([\![\Phi]\!] \cup [\![\Psi]\!]) \\ \sigma \in [\![\Phi \wedge \Psi]\!] & \text{iff } \sigma \in ([\![\Phi]\!] \cap [\![\Psi]\!]) \\ \sigma \in [\![\Phi \Rightarrow \Psi]\!] & \text{iff } \sigma \in [\![\neg \Phi \vee \Psi]\!] \end{array}$$

$$\begin{array}{ll} \sigma \in [\![\Box\Phi]\!] & \text{iff For all } i, i \geq 0\text{: } s_i s_{i+1} \ldots \models \Phi \\ \sigma \in [\![\Diamond\Phi]\!] & \text{iff } \sigma \in [\![\neg \Box \neg \Phi]\!] \end{array}$$

A TLA formula Φ is *valid* if and only if for every behavior σ, $\sigma \in [\![\Phi]\!]$ holds. Validity of $\Phi \Rightarrow \Psi$ implies that every behavior σ is in $[\![\Phi \Rightarrow \Psi]\!]$. From the definition above for $\sigma \in [\![\Phi \Rightarrow \Psi]\!]$, we have that if $\Phi \Rightarrow \Psi$ is valid then every σ in $[\![\Phi]\!]$ is also in $[\![\Psi]\!]$. Accordingly, we conclude:

$\Phi \Rightarrow \Psi$ is valid if and only if $\langle \Phi, \Psi \rangle \in Sat$ (3.1)

To prove that a program S satisfies a property Ψ using TLA, we

(1) construct a TLA formula Φ_S such that $[\![\Phi_S]\!]$ is the set of behaviors of S, and

(2) prove $\Phi_S \Rightarrow \Psi$ valid.

As an example, we return to the program of §1. It is reproduced in Figure 3.1, with each atomic action labeled. The TLA formula Φ_S that characterizes behaviors for this program is

Φ_S: $Init_S \wedge \Box[A_S]_{y, pc_1, pc_2}$ (3.2)

where $Init_S$ is a state predicate defining initial states of the program's behavior and A_S is a TLA

cobegin

P_1: α: $y := 1$;
β: $y := 2$

//

P_2: γ: $y := 3$

coend

Figure 3.1. A concurrent program

action that characterizes the program's next-state relation. In defining the effect of each atomic action, variable pc_i denotes the program counter for process P_i and value "$\downarrow$" is assumed to be different from the entry (control) point for any atomic action of the program.

$Init_S$: $pc_1=\alpha \wedge pc_2=\gamma$

A_S: $A_\alpha \vee A_\beta \vee A_\gamma$

A_α: $pc_1=\alpha \wedge pc_1'=\beta \wedge y'=1 \wedge pc_2=pc_2'$

A_β: $pc_1=\beta \wedge pc_1'=\downarrow \wedge y'=2 \wedge pc_2=pc_2'$

A_γ: $pc_2=\gamma \wedge pc_2'=\downarrow \wedge y'=3 \wedge pc_1=pc_1'$

3.2. Exploiting an Environment with TLA

If the property defined by an environment can be characterized in TLA, then Program Reduction can be used to reason about feasible behaviors under that environment. We prove $\Phi \wedge E \Rightarrow \Psi$ to establish that behaviors of the program characterized by Φ under the environment characterized by E are in the property characterized by Ψ:

$\Phi \wedge E \Rightarrow \Psi$ is valid
iff «definition (3.1)»
$\langle \Phi \wedge E, \Psi \rangle \in Sat$
iff «definition (2.1)»
$[\![\Phi \wedge E]\!] \subseteq [\![\Psi]\!]$
iff «$[\![F \wedge G]\!] = [\![F]\!] \cap [\![G]\!]$»
$([\![\Phi]\!] \cap [\![E]\!]) \subseteq [\![\Psi]\!]$
iff «definition (2.1)»
$\langle \Phi \cap E, \Psi \rangle \in Sat$
iff «Program Reduction (2.3)»
$\langle \Phi, E, \Psi \rangle \in ESat$

The utility of this method depends on (i) being able to prove $\Phi \wedge E \Rightarrow \Psi$ when it is valid and (ii) being able to characterize in TLA those aspects of environments that interest us. A complete[4] deductive system for TLA (see [L91], for example) will, by definition, be complete for proving $\Phi \wedge E \Rightarrow \Psi$. In fact, this is one of the advantages of using Program Reduction to extend a complete proof system for *Sat* into a proof system for *ESat*—the complete proof system for *ESat* comes at no cost. Examples in the remainder of this section convey a sense for how an environment is represented by a TLA formula.

3.3. Schedulers as TLA formulas

If there are more processes than processors in a computer system, then processors must be shared. This sharing is usually implemented by the scheduler of an operating system. To use Program Reduction with TLA and reason about execution of a program under a given scheduler, we write a TLA formula E to characterize that scheduler.

[4]Completeness here and throughout this paper is only relative to arithmetic.

Many schedulers implement safety properties—they rule out certain assignments of processors to processes. Formalizations for these schedulers have much in common. Let Π be the set of processes to be executed in a system with N processors. For each process π, two pieces of information are maintained (in some form) by a scheduler:

$active_\pi$: whether there is a processor currently allocated to π

$rank_\pi$: a value used to determine whether a processor should be allocated to π

Only a single atomic action from one process can be executed at any time by a processor. This restriction is formalized as predicate $Alloc(N)$, which bounds the number of processes to which N processors can be allocated at any time:[5]

$Alloc(N)$: $(\#\pi \in \Pi: active_\pi) \leq N$

The restriction that processes that have processors allocated are the only ones that advance is formalized in terms of A_π, the next-state relation for a process π. We assume that these next-state relations are disjoint.

$Pgrs(\pi)$: $A_\pi \Rightarrow active_\pi$

Finally, we formalize as $Run(\pi)$ the requirement that $active_\pi$ holds only for those processes with sufficiently large rank.

$Run(\pi)$: $active_\pi \Rightarrow |larger(\pi)| < N$

where:
$larger(\pi)$: $\{\pi' \mid rank_\pi < rank_{\pi'}\}$

In a *fixed-priority scheduler*, there is a fixed value v_π associated with each process π. A process that has not terminated and has higher priority is executed in preference to a process having a lower priority. This is ensured by assigning ranks as follows.

$Prio(\pi)$: $(pc_\pi \neq \downarrow \Rightarrow (rank_\pi = v_\pi)) \wedge (pc_\pi = \downarrow \Rightarrow (rank_\pi = 0))$

A fixed-priority scheduler is thus characterized by

$FixedPrio$: $\Box[Alloc(N) \ \wedge \ (\forall \pi \in \Pi: Pgrs(\pi) \wedge Run(\pi) \wedge Prio(\pi))]_{\overline{x}}$

where $\overline{x}$ is a list of all the variables in the system. For example, $\overline{x}$ for the program of Figure 3.1 would have pc_1, pc_2, y, $active_{P_1}$, $rank_{P_1}$, $active_{P_2}$, and $rank_{P_2}$.

In a *first-come first-served* scheduler, processes are ranked in accordance with elapsed time since last executed. We can model this by assigning ranks that are increased for processes that have not had an action executed.

$Age(\Pi)$: $(\forall \pi \in \Pi: (A_\pi \Rightarrow (rank_\pi' = 0)) \wedge (\neg A_\pi \Rightarrow (rank_\pi' = rank_\pi + 1)))$

A first-come, first-served scheduler is therefore characterized by

$FCFS$: $(\forall \pi \in \Pi: rank_\pi = 0) \ \wedge \ \Box[Alloc(N) \wedge (\forall \pi \in \Pi: Pgrs(\pi) \wedge Run(\pi)) \wedge Age(\Pi)]_{\overline{x}}$

where $\overline{x}$ is a list of all the variables in the system.

---

[5] We use the notation $(\#x \in P: R)$ for "the number of distinct values of x in P for which R holds".

Both of these schedulers can allocate a processor to a process, even though that process may be unable to make progress. It is wasteful to allocate a processor to process π when $Enbl(A_\pi)$ does not hold (because π has terminated or because its next atomic action is not enabled). A variant of *FixedPrio* that allocates processors only to non-terminated and enabled higher-priority processes is:

$$EnblFixedPrio: \ \Box[Alloc(N) \ \wedge \ (\forall \pi \in \Pi: \ Pgrs(\pi) \wedge Run(\pi) \wedge EnblPrio(\pi))]_{\overline{x}}$$

where

$$EnblPrio(\pi): \ (Enbl(A_\pi) \Rightarrow (rank_\pi = v_\pi)) \ \wedge \ (\neg Enbl(A_\pi) \Rightarrow (rank_\pi = 0))$$

As before, $\overline{x}$ is a list of all the variables in the system.

A difficulty with assigning fixed priorities to processes is that execution of a high-priority process can be delayed awaiting progress by processes with lower-priorities. For example, suppose a high-priority process π_H is awaiting some lock to be freed, so π_H is not enabled. If that lock is owned by a lower-priority process π_L, then execution of π_H cannot proceed until π_L executes. This is known as a *priority inversion* [SRL90][BMS93], because execution of a high-priority process depends on resources being allocated to a lower-priority process.

Priority Inheritance schedulers give preference to low-priority processes that are blocking high-priority processes. This is done by changing process priorities. The low-priority process inherits a new, higher priority from any higher-priority process it blocks. Priority inheritance schedulers exhibit improved worst-case response times in systems of tasks [SRL90], and they have become important in the design of real-time systems.

A priority inheritance scheduler must know what processes are blocked and how to unblock them. In systems where acquiring a lock is the only operation that blocks a process, deducing this information is easy: execution of the process that has acquired a lock is the only way that a process awaiting that lock becomes unblocked.

To describe systems with locks in TLA, we employ a variable $lock_i$ for each lock; TLA actions for acquiring and releasing a lock by process π are:

$$acquire(lock_i, \pi): \ \ lock_i = FREE \wedge lock_i' = \pi$$

$$release(lock_i): \ \ lock_i' = FREE$$

Notice that $lock_i = FREE$ is implied by $Enbl(A_\pi)$ when process π is waiting to acquire $lock_i$.

In a priority inheritance scheduler, each process π is assumed to have a priority v_π. The rank of a process π is the maximum of v_π and the priorities assigned to processes that are blocked by π. Thus, $rank_\pi$ is the maximum of v_p for the process p satisfying $lock_i = p$ (i.e. the priority of the current lock holder) and v_q for the process q satisfying $Enbl(q) \Rightarrow (lock_i = FREE)$ (i.e. the priority of the process attempting to acquire $lock_i$). For simplicity, we assume a system having a single lock, *lock*.

$$\begin{aligned} PrioInher(\pi): \ & [(\neg Enbl(A_\pi) \Rightarrow (rank_\pi = 0)) \ \wedge \\ & (lock = \pi \wedge Enbl(A_\pi) \\ & \qquad \Rightarrow (rank_\pi = (\max p \in \Pi: \ (Enbl(p) \Rightarrow lock = FREE) \vee lock = p: \ v_p))) \ \wedge \\ & (lock \neq \pi \wedge Enbl(A_\pi) \Rightarrow (rank_\pi = v_\pi))]_{\overline{x}} \end{aligned}$$

Again, $\overline{x}$ is a list of all the variables in the system. A priority inheritance scheduler is thus characterized by

$$InhPrio: \ \Box[Alloc(N) \ \wedge \ (\forall \pi \in \Pi: \ Pgrs(\pi) \wedge Run(\pi) \wedge PrioInher(\pi))]_{\overline{x}}$$

Not all schedulers are safety properties. Even schedulers that implement safety properties are often abstracted in programming logics as implementing (weaker) liveness properties. Such a liveness property gives conditions under which an action or process will be executed eventually. A simple example is the following, which implies that an enabled process with sufficiently high priority will execute.

$$FAIR: \quad (\forall \pi \in \Pi: \ \Diamond\Box(\pi \in TOP(n, \Pi) \wedge Enbl(\pi)): \ \neg\Diamond\Box[\neg A_\pi]_{\bar{x}})$$

Other examples of such liveness properties include *weak fairness* $\mathrm{WF}_{\bar{x}}(A)$ and *strong fairness* $\mathrm{SF}_{\bar{x}}(A)$ of TLA.

TLA Reasoning about Schedulers

In section 3.2, we showed that given TLA formulas Φ_S and E for a program and scheduler respectively, $\Phi_S \wedge E \Rightarrow \Psi$ is valid iff behaviors of S under E satisfy Ψ. Returning, for example, to the program of Figure 3.1, we prove as follows that assuming a fixed-priority scheduler, a single processor (i.e. $N=1$), $v_{P_1}=2$ and $v_{P_2}=1$ implies that $y=3$ will hold upon termination. The property that $y=3$ holds upon termination of S is formulated in TLA as:

$$\Box(\neg Enbl(A_S) \Rightarrow y=3).$$

Thus, for Φ_S as defined by (3.2), we must prove:

$$\Phi_S \wedge FixedPrio \wedge \Box(N=1 \wedge v_{P_1}=2 \wedge v_{P_2}=1 \wedge \Pi=\{P_1, P_2\}) \quad (3.3)$$
$$\Rightarrow \quad \Box(\neg Enbl(A_S) \Rightarrow y=3).$$

In general, one proves a TLA formula $init \wedge \Box[A] \wedge \Box B \Rightarrow \Box C$ by finding a predicate I, called an *invariant*, and proving[6] $init \Rightarrow I$, $I \Rightarrow C$, and $I \wedge A \wedge B \wedge B' \Rightarrow I'$. The first obligation establishes that I holds initially, the second implies that C holds whenever I does, and the third ensures that I holds throughout.

For proving (3.3), we choose[7]:

$init$: $Init_S$

$\Box[A]$: $\Box[A_s]_{y, pc_1, pc_2} \ \wedge \ FixedPrio$

B: $N=1 \wedge v_{P_1}=2 \wedge v_{P_2}=1$

For I, the following suffices—the proof is left to the reader:

I: $(\neg Enbl(A_S) \Rightarrow y=3) \wedge ((pc_{P_1} \neq \downarrow) \Rightarrow (pc_{P_2}=\gamma))$

3.4. Real time in TLA

The correlation between execution of a program and the advancement of time is largely an artifact of the environment in which that program executes. The scheduler, the number of processors, and the availability of other resources all play a role in determining when a process may take a step. To reason with TLA about properties satisfied by a program in such an environment, we simply

[6] A' denotes the formula obtained by priming each un-primed free variable in A.

[7] The choice of B is based on applying the Temporal Logic axiom $(\Box E \wedge \Box F)=\Box(E \wedge F)$.

characterize the way time advances and then use Program Reduction. Various models of real-time one finds in the literature differ only in their characterization of how time advances.

When only a single processor is assumed, then process execution is interleaved on that processor. One way to abstract this is to associate two constants with each atomic action α:

e_α: the fixed execution time of atomic action α on a bare machine

δ_α: the maximum time that can elapse from the time that the processor is allocated for execution of α until α starts executing.

Execution of α is thus correlated with the passage of between e_α and $e_\alpha+\delta_\alpha$ time units.

The following TLA formula is satisfied by such behaviors. Variable T is the current time and ATOM(S) is the set of atomic actions in S. Recall that A_α defines atomic action α.

$$\mathrm{T}=0 \ \wedge \ \Box[\bigwedge_{\alpha \in \mathrm{ATOM}(S)} (A_\alpha \Rightarrow (\mathrm{T}+e_\alpha \le \mathrm{T}' \le \mathrm{T}+e_\alpha+\delta_\alpha))]_{\bar{x}}$$

As before, $\bar{x}$ is a list of all variables in the system.

Another common model of how time advances abstracts the case where each process is executed on its own processor. We assume that the next action to be executed at process π is uniquely defined at each control point. (Other assumptions are possible, and these can be formalized also.) We formalize this environment in TLA, by using a separate variable T_π for each process π:

T_π: the time process π arrived at its current state.

System time T is the maximum of the T_π:

$$\mathit{SysTme}: \ \mathrm{T}=\max_{\pi \in \Pi}(\mathrm{T}_\pi)$$

And each individual process π must execute its next action α (say) before $e_\alpha+\delta_\alpha$ has elapsed from the time π reached its current state. Let the label on action α be "α".

$$\mathit{LclTme}: \ (\forall \pi \in \Pi: \ pc_\pi=\alpha: \ \mathrm{T}-\mathrm{T}_\pi \le e_\alpha+\delta_\alpha)$$

The range $pc_\pi=\alpha$ is satisfied by states in which the program counter for process π indicates that α is the next atomic action to be executed; the body requires α to be executed before the system's time has advanced too far.

Finally, the value of T_π changes iff an atomic action from process π is executed:

$$\mathit{LclTmeUpdt}: \ (\forall \pi \in \Pi: \ (\forall \alpha \in \mathrm{ATOM}(S): \ A_\alpha: \ \mathrm{T}_\pi+e_\alpha \le \mathrm{T}_\pi' \le \mathrm{T}_\pi+e_\alpha+\delta_\alpha \ \wedge \ (\forall \phi \in \Pi: \ \phi \ne \pi: \ \mathrm{T}_\phi'=\mathrm{T}_\phi)))$$

Here, the range is satisfied only by steps attributed to atomic action α of process π; the body causes all of the T_π to be updated.

Putting all these together, we get a TLA formula characterizing this model of real time:

$$\mathrm{T}=0 \ \wedge \ (\forall \pi \in \Pi: \ \mathrm{T}_\pi=0) \ \wedge \ \Box[\bigwedge_{\alpha \in \mathrm{ATOM}(S)} (\mathit{SysTme} \wedge \mathit{LclTme} \wedge \mathit{LclTmeUpdt})]_{\bar{x}} \qquad (3.4)$$

An Old-fashioned Recipe

The scheme just described works by restricting the transitions allowed by each action. These restrictions ensure that an action only executes when its starting and ending times are as prescribed by the real-time model. Thus, the approach regards the environment as augmenting each action of the

original system. The environment executes simultaneously with the system's actions.

A somewhat different approach to reasoning about real-time with TLA is described by Abadi and Lamport in "An old-fashioned recipe for real-time" [AL91]. That recipe is extended for handling schedulers in [LJJ93]. Like our scheme, the recipe does not require changes to the language or deductive system of TLA. However, unlike our scheme, additional actions are used to handle the passage of time. These new actions interleave with the original program actions, updating a clock and some count-down timers.

There seems to be no technical reason to prefer one approach to the other. In the examples we have checked, the old-fashioned recipe is a bit cumbersome. A variable *now* analogous to our variable T is used to keep track of the current time, and a variable, called a *timer*, is associated with each atomic action whose execution timing is constrained. Timers ensure (i) that the new actions to advance *now* are disabled when actions of the original program must progress and (ii) that actions of the original program are disabled when *now* has not advanced sufficiently. The timers, *now*, and added actions implement what amounts to a discrete-event simulation that causes time to advance and actions to be executed in an order consistent with timing constraints. To write real-time specifications, it suffices to learn the few TLA idioms in [AL91] and repeat them. However, to prove properties from these specifications, the details of this discrete event simulation must be mastered.

4. Environments for a Hoare-style Proof Outline Logic

We now turn our attention to a second programming logic—one that is quite different in character from TLA and can be used for proving safety but not for proving liveness properties. The formulas of a Hoare-style logic are imperative programs in which an assertion is associated with each control point. This rules out Program Reduction (2.3), because imperative programming languages are generally not closed under intersection of any sort.[8] Similarly, Property Reduction (2.4) is ruled out because the property language, annotated program texts, also lacks the necessary closure. However, it is not difficult to extend the property language of a Hoare-style logic and then apply Property Reduction (2.4). An example of such an extension is given in this section.

4.1. A Hoare-style Logic

Consider a simple programming language having assignment, sequential composition, and parallel composition statements.[9] An example program is given in Figure 4.1; it is equivalent to the program of Figure 1.1.

The syntax of programs in our language is given by the following grammar. There, λ is a label, x is a program variable, and E is an expression over the program variables.

$$S ::= \quad \lambda\colon [x := E] \quad | \quad \lambda\colon [S;\ S] \quad | \quad \lambda\colon [S \mathbin{//} S]$$

Every label in a program is assumed to be unique. In the discussion that follows, the label on the entire program is used to name that program. In addition, for a statement λ: [...], we call "λ: [" the *opening* of λ, call "]" the *closing*, and define *Lab*(λ) to be the set containing label λ and all labels used between the opening and closing of λ.

---

[8] Constraint-maintenance languages are the obvious exception.

[9] Handling an imperative language with **if** and **do** is not fundamentally different.

λ: [λ_1: [λ_{11}: [$y := 1$];
λ_{12}: [$y := 2$]]
//
λ_2: [$y := 3$]]

Figure 4.1. Simple Program

A program state assigns values to the program variables and to control variables. The *control variables* for a program λ are $at(\lambda')$, $in(\lambda')$, and $after(\lambda')$ for every label λ' in $Lab(\lambda)$.

The set Σ of program states contains only those states satisfying certain constraints on the values of control variables. These constraints are given in Figure 4.2. They ensure that the control variables encode plausible values of program counters. For example, the constraints rule out the possibility that control variables $at(\lambda)$ and $after(\lambda)$ are both *true* in a state. As another example, the constraints imply that any state for program λ of Figure 4.1 assigning *true* to $after(\lambda_{11})$ must also assign *true* to $at(\lambda_{12})$.

The executions of a program λ defines a set of behaviors. It will be convenient to represent a

Each state s of a program λ satisfies:

C0: $s \models (in(\lambda) \neq after(\lambda))$

C1: $s \models \neg (at(\lambda) \wedge after(\lambda))$

C2: $s \models (at(\lambda) \Rightarrow in(\lambda))$

C3: For every assignment statement λ: [$x := E$]:
$s \models (at(\lambda) = in(\lambda))$

C4: For every sequential composition λ: [λ_1: [S 1]; λ_2: [S 2]]:
$s \models (at(\lambda) = at(\lambda_1))$
$s \models (after(\lambda) = after(\lambda_2))$
$s \models (after(\lambda_1) = at(\lambda_2))$
$s \models ((in(\lambda_1) \vee in(\lambda_2)) \Rightarrow in(\lambda))$
$s \models \neg (in(\lambda_1) \wedge in(\lambda_2))$

C5: For every parallel composition λ: [λ_1: [S 1] // λ_2: [S 2]]:
$s \models (at(\lambda) = (at(\lambda_1) \wedge at(\lambda_2)))$
$s \models (after(\lambda) = (after(\lambda_1) \wedge after(\lambda_2)))$
$s \models (in(\lambda) = ((in(\lambda_1) \vee after(\lambda_1)) \wedge (in(\lambda_2) \vee after(\lambda_2)) \wedge \neg (after(\lambda_1) \wedge after(\lambda_2))))$

Figure 4.2. Constraints on control variables

behavior using a triple $\langle \sigma, i, j \rangle$, where σ is an infinite sequence[10] of states, i is a natural number, and j is a natural number satisfying $i \leq j$ or is ∞. Informally, behavior $\langle \sigma, i, j \rangle$ models a (possibly partial) execution starting in state $\sigma[i]$ that produces sequence of states $\sigma[i..j]$. Prefix $\sigma[..i-1]$ is the sequence of states that precedes the execution; suffix $\sigma[j..]$ models subsequent execution.

Formally, we define the set $[\![\lambda]\!]$ of behaviors for a program λ in terms of relations $R_{\lambda': [x := E]}$ for the assignments λ' in λ:

$$\langle s, t \rangle \in R_{\lambda': [x := E]} \quad \text{iff} \quad s \vDash at(\lambda'),\ \ t \vDash after(\lambda'),\ \ t(x) = s(E),\ \ \text{and} \quad s(v) = t(v) \text{ for all program variables } v \text{ different from } x. \tag{4.1}$$

Let $Assig(\lambda)$ be the subset of $Lab(\lambda)$ that are labels on assignment statements in λ. Behavior $\langle \sigma, i, j \rangle$ is defined to be an element of $[\![\lambda]\!]$ iff

$$\text{For all } k,\ i \leq k < j\text{: Exists } \lambda' \in Assig(\lambda)\text{: } \langle \sigma[k], \sigma[k+1] \rangle \in R_{\lambda': [x := E]} \tag{4.2}$$

Thus, each pair of adjacent states in $\sigma[i..j]$ models execution of some assignment statement and the corresponding changes to the target and control variables.

Proof Outlines

Having defined the program language, we now define the property language of Proof Outline Logic. A *proof outline* for a program λ associates an assertion with the opening and closing of each label in $Lab(\lambda)$. The assertion associated with the opening of a label λ is called the *precondition* of λ and is denoted $pre(\lambda)$; the assertion associated with its closing is called the *postcondition* of λ and is denoted $post(\lambda)$.

Here is a grammar giving a syntax of proof outlines for our simple programming language.

$$\begin{aligned} PO ::= \quad & \{p\}\ \lambda\text{: } [x := E]\ \{q\} \quad | \\ & \{p\}\ \lambda\text{: } [PO_1;\ PO_2]\ \{q\} \quad | \\ & \{p\}\ \lambda\text{: } [PO_1\ //\ PO_2]\ \{q\} \end{aligned}$$

PO_1 and PO_2 are proof outlines, and p and q are assertions. A concrete example of a proof outline is given in Figure 4.3. It contains a proof outline for the program of Figure 4.1. Easier to read notations[11] for proof outlines do exist; this format is particularly easy to define formally, so it is well suited to our purpose.

Assertions in proof outlines are formulas of a first-order predicate logic. In this logic, terms and predicates are evaluated over *traces*, finite sequences of program states. A trace $s_0\, s_1 \ldots s_n$ that is a prefix of a program behavior defines a *current* program state s_n as well as a sequence $s_0\, s_1 \ldots s_{n-1}$ of *past* states. Thus, assertions interpreted with respect to traces can not only characterize the current state of the system, but can also characterize histories leading up to that state. Such expressiveness is necessary for proving arbitrary safety properties and for describing many environments.

The terms of our assertion language include constants, variables, the usual expressions over

---

[10]For an infinite sequence $\sigma = s_0\, s_1 \ldots$ we write: $\sigma[i]$ to denote s_i; $\sigma[..i]$ to denote prefix $s_0\, s_1 \ldots s_i$; $\sigma[i..]$ to denote suffix $s_i\, s_{i+1} \ldots$; and $\sigma[i..j]$, where $i \leq j$, to denote subsequence $s_i \ldots s_j$.

[11]For example, we sometimes write $\{p\}\ PO(\lambda)\ \{q\}$ to denote a proof outline that is identical to $PO(\lambda)$ but with p replacing $pre(\lambda)$ and q replacing $post(\lambda)$.

$\{true\}$
λ: [$\{true\}$
λ_1: [$\{true\}$
λ_{11}: $[y := 1]$ $\{y=1 \vee y=3\}$;
$\{y=1 \vee y=3\}$
λ_{12}: $[y := 2]$ $\{y=2 \vee y=3\}$
] $\{y=2 \vee y=3\}$
//
$\{true\}$
λ_2: $[y := 3]$ $\{y=2 \vee y=3\}$
] $\{y=2 \vee y=3\}$

Figure 4.3. Example Proof Outline

terms, and the *past term* ΘT for T any term [S94].[12] The Θ operator allows terms to be constructed whose values depend on the past of a trace. For example, $x+\Theta y$ evaluated in a trace $s_0\, s_1\, s_2$ equals $s_2(x)+s_1(y)$. More formally, we define as follows the value $M[\![T]\!]\tau$ of a term T in trace τ, where c is a constant, v is a variable, and T_1 and T_2 are terms.

term T	$M[\![T]\!]s_0\, s_1 \ldots s_n$
c	c
v	$s_n(v)$
T_1+T_2	$M[\![T_1]\!]s_0\, s_1 \ldots s_n \quad + \quad M[\![T_2]\!]s_0\, s_1 \ldots s_n$
...	...
ΘT	$\begin{cases} M[\![T]\!]s_0\, s_1 \ldots s_{n-1} & \text{if } n>0 \\ false & \text{if } n=0 \end{cases}$

Predicates of the assertion language are formed in the usual way from predicate symbols, terms, propositional connectives, and the universal and existential quantifiers. It is also convenient to regard Boolean-valued variables as predicates. This allows control variables to be treated as predicates. It also allows $\Theta true$ to be treated as a predicate whose value is *true* in any trace having more than one state. Assertions are just predicates.

Proof outlines define properties. Informally, the property defined by a proof outline $PO(\lambda)$ includes all behaviors $\langle \sigma, i, j \rangle$ in which execution of λ starting in state $\sigma[i]$ does not cause *proof outline invariant* $I_{PO(\lambda)}$ to be invalidated. The proof outline invariant implies that the assertion associated with each control variable is *true* whenever that control variable is *true*:

$$I_{PO(\lambda)}: \quad \bigwedge_{\lambda' \in Lab(\lambda)} ((at(\lambda') \Rightarrow pre(\lambda')) \wedge (after(\lambda') \Rightarrow post(\lambda'))) \tag{4.3}$$

[12]The Proof Outline Logic of [S94] also allows recursively-defined terms using Θ. This increases the expressiveness of the assertion language, but is independent to the issues being addressed in this paper. Therefore, in the interest of simplicity, we omit such terms from the assertion language.

It is easier to reason about proof outlines when the precondition for each statement λ' summarizes what is required for $I_{PO(\lambda)}$ to hold when $at(\lambda')$ is *true*. Then, proving that $pre(\lambda)$ holds before λ is executed suffices to ensure that $I_{PO(\lambda)}$ will hold throughout execution. For a proof outline $PO(\lambda)$, this *self consistency* requirement is:

For every label $\lambda' \in Lab(\lambda)$:

If λ' labels a sequential composition λ': $[\lambda_1 : [S_1];\ \lambda_2 : [S_2]]$ then:
$pre(\lambda') \Rightarrow pre(\lambda_1)$
$post(\lambda_1) \Rightarrow pre(\lambda_2)$

If λ' labels a parallel composition λ': $[\lambda_1 : [S_1]\ //\ \lambda_2 : [S_2]]$ then:
$pre(\lambda') \Rightarrow (pre(\lambda_1) \wedge pre(\lambda_2))$

We can now formally define the set $[\![PO(\lambda)]\!]$ of behaviors in the property $PO(\lambda)$:

$$[\![PO(\lambda)]\!]: \quad \begin{cases} \varnothing & \text{if } PO(\lambda) \text{ is not self consistent} \\ \{\langle \sigma, i, j \rangle \mid \sigma[..i] \not\models I_{PO(\lambda)} \text{ or for all } k,\ i \le k \le j:\ \sigma[..k] \models I_{PO(\lambda)}\} \end{cases} \tag{4.4}$$

Thus, $[\![PO(\lambda)]\!]$ is empty if $PO(\lambda)$ is not self consistent. And, if $PO(\lambda)$ is self consistent, then $[\![PO(\lambda)]\!]$ includes a behavior $\langle \sigma, i, j \rangle$ provided either (i) $I_{PO(S)}$ is not satisfied when execution is started in state $\sigma[i]$ or (ii) $I_{PO(S)}$ is kept *true* throughout execution started in state $\sigma[i]$. In the definition, proof outline invariant $I_{PO(S)}$ is evaluated in prefixes of σ because assertions may contain terms involving Θ.

A proof outline is defined to be *valid* iff $\langle \lambda, PO(\lambda) \rangle \in Sat$ holds, where

$$\langle \lambda, PO(\lambda) \rangle \in Sat \quad \text{if and only if} \quad [\![\lambda]\!] \subseteq [\![PO(\lambda)]\!] \tag{4.5}$$

as prescribed by (2.1). Appendix A contains a sound and complete proof system for establishing that a proof outline is valid. Such logics have become commonplace since Hoare's original proposal [H69]. The particular axiomatization that we give is based on [S94], which, in turn, builds on the logic of [L80].

4.2. Exploiting an Environment with Proof Outlines

Our program language does not satisfy the closure conditions required for Program Reduction (2.3), nor does the property language (proof outlines) satisfy the closure conditions required for Property Reduction (2.4). To pursue property reduction, we define a language *EnvL* that characterizes properties imposed by environments. We then extend the property language so that it satisfies the necessary closure condition for property reduction.

We base *EnvL* on the assertion language of proof outlines. Every formula of *EnvL* is of the form $\Box A$ where A is a formula of the assertion language. $\Box A$ defines a set of behaviors as follows.

$$[\![\Box A]\!]: \quad \{\langle \sigma, i, j \rangle \mid \text{For all } k,\ i \le k \le j:\ \sigma[..k] \models A\}$$

Thus, $\Box A$ contains behaviors $\langle \sigma, i, j \rangle$ for which prefixes $\sigma[..i]$, $\sigma[..i+1]$, ..., $\sigma[..j]$ do not violate A. Formulas in *EnvL* define safety properties, and *EnvL* includes all of the scheduler and real-time examples of §3.3 and §3.4. A more expressive assertion language (e.g. the one with recursive terms in [S94]) would enable all safety properties to be defined in this manner.

In order to close the property language of Proof Outline Logic under union with the complement of $[\![\Box A]\!]$, we introduce a new form of proof outline. A *constrained proof outline* is a formula $\Box A \rightarrow PO(\lambda)$, where A is a formula of the assertion language and $PO(\lambda)$ is an ordinary proof outline. The property defined by a constrained proof outline is given by:

$$[\![\Box A \rightarrow PO(\lambda)]\!]: \ [\![PO(\lambda)]\!] \cup \overline{[\![\Box A]\!]} \tag{4.6}$$

Generalizing from ordinary proof outlines, a constrained proof outline $\Box A \rightarrow PO(\lambda)$ is considered *valid* iff $\langle \lambda, \Box A \rightarrow PO(\lambda) \rangle \in Sat$. Thus, if $\Box A \rightarrow PO(\lambda)$ is valid then $[\![\lambda]\!] \subseteq [\![\Box A \rightarrow PO(\lambda)]\!]$ holds.

The set of properties defined by constrained proof outlines and proof outlines does satisfy the necessary closure condition for property reduction. Given a program λ, let L_λ be the set of constrained proof outlines and proof outlines for λ. The required closure condition is equivalent to:

Lemma: For any assertion A and any $\Phi \in L_\lambda$, there exists a constrained proof outline Φ' in L_λ such that

$$[\![\Phi']\!] \ = \ [\![\Phi]\!] \cup \overline{[\![\Box A]\!]}$$

Proof. The proof is by cases.

Case: Φ *is an ordinary proof outline.* In this case, choose Φ' to be $\Box A \rightarrow \Phi$.

Case: Φ *is a constrained proof outline* $\Box B \rightarrow PO(\lambda)$. In this case, choose Φ' to be $\Box(A \wedge B) \rightarrow PO(\lambda)$. This choice is justified by the following.

$\langle \sigma, i, j \rangle \in [\![\Box(A \wedge B) \rightarrow PO(\lambda)]\!]$
iff «definition (4.6) of $[\![\Box(A \wedge B) \rightarrow PO(\lambda)]\!]$»
$\langle \sigma, i, j \rangle \in ([\![PO(\lambda)]\!] \cup \overline{[\![\Box(A \wedge B)]\!]})$
iff «definition of $\overline{[\![\Box(A \wedge B)]\!]}$»
$\langle \sigma, i, j \rangle \in ([\![PO(\lambda)]\!] \cup \overline{[\![\Box A]\!]} \cup \overline{[\![\Box B]\!]})$
iff «definition (4.6) of $[\![\Box B \rightarrow PO(\lambda)]\!]$»
$\langle \sigma, i, j \rangle \in ([\![\Box B \rightarrow PO(\lambda)]\!] \cup \overline{[\![\Box A]\!]})$

Q.E.D.

Logic for Constrained Proof Outlines

Our goal is to prove that a program λ satisfies a property $PO(\lambda)$ under an environment $\Box A$:

$$\langle \lambda, \Box A, PO(\lambda) \rangle \in ESat \tag{4.7}$$

Using Property Reduction (2.4), we see that to prove (4.7), it suffices to be able to prove that λ satisfies property $\Box A \rightarrow PO(\lambda)$.

$\langle \lambda, \Box A, PO(\lambda) \rangle \in ESat$
iff «Property Reduction (2.4)»
$\langle \lambda, PO(\lambda) \cup \overline{\Box A} \rangle \in Sat$
iff «definition (2.1)»
$[\![\lambda]\!] \subseteq [\![PO(\lambda) \cup \overline{\Box A}]\!]$
iff «$[\![F \cup G]\!] = [\![F]\!] \cup [\![G]\!]$ and definition (4.6) of $\Box A \rightarrow PO(\lambda)$»
$[\![\lambda]\!] \subseteq [\![\Box A \rightarrow PO(\lambda)]\!]$
iff «definition (2.1)»

$\langle \lambda, \Box A \rightarrow PO(\lambda) \rangle \in Sat$

The deductive system of Appendix A enables us to prove that $\langle \lambda, \Phi \rangle \in Sat$ holds for Φ an ordinary proof outline. Extensions are needed for the case where Φ is a constrained proof outline. We now give these; a soundness and completeness proof for them appears in Appendix B.

For reasoning about assignment statements executed under an environment $\Box A$, we can assume that A holds before execution and, because the environment precludes transition to a state satisfying $\neg A$, any postcondition asserting $\neg A$ can be strengthened.

Cnstr-Assig:
$$\frac{\{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}}{\Box A \rightarrow \{p\}\ \ \lambda\colon [x := E]\ \ \{q\}}$$

Sequential composition under an environment $\Box A$ allows a weaker postcondition for the first statement, since the environment ensures that A will hold.

Cnstr-SeqComp:
$$\frac{\begin{array}{l}\Box A \rightarrow PO(\lambda_1),\ \Box A \rightarrow PO(\lambda_2) \\ (A \wedge post(\lambda_1)) \Rightarrow pre(\lambda_2)\end{array}}{\Box A \rightarrow \{pre(\lambda_1)\}\ \ \lambda\colon [PO(\lambda_1); PO(\lambda_2)]\ \ \{post(\lambda_2)\}}$$

Parallel composition under an environment $\Box A$ also allows weaker assertions. A can be assumed in the preconditions of the interference-freedom proofs.

Cnstr-ParComp:
$$\frac{\begin{array}{l}\Box A \rightarrow PO(\lambda_1),\quad \Box A \rightarrow PO(\lambda_2), \\ \Box A \rightarrow PO(\lambda_1) \text{ and } \Box A \rightarrow PO(\lambda_2) \text{ are interference free}\end{array}}{\Box A \rightarrow \{pre(\lambda_1) \wedge pre(\lambda_2)\}\ \ \lambda\colon [\ PO(\lambda_1)\ //\ PO(\lambda_2)]\ \ \{post(\lambda_1) \wedge post(\lambda_2)\}}$$

We establish that $\Box A \rightarrow PO(\lambda_1)$ and $\Box A \rightarrow PO(\lambda_2)$ are *interference free* in much the same way as for ordinary proof outlines.

For all $\lambda_\alpha \in Assig(\lambda_1)$, where λ_α is the assignment $\lambda_\alpha\colon [x := E]$:

$$\Box A \rightarrow \{at(\lambda_\alpha) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \ \lambda_\alpha\colon [x := E]\ \ \{I_{PO(\lambda_2)}\}$$

For all $\lambda_\alpha \in Assig(\lambda_2)$, where λ_α is the assignment $\lambda_\alpha\colon [x := E]$:

$$\Box A \rightarrow \{at(\lambda_\alpha) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)}\}\ \ \lambda_\alpha\colon [x := E]\ \ \{I_{PO(\lambda_1)}\}$$

As with ordinary proof outlines, two rules allow us to modify assertions based on deductions possible in the assertion language. For a constrained proof outline $\Box A \rightarrow PO(\lambda)$, we can assume A in making those deductions.

Cnstr-Conseq:
$$\frac{\Box A \rightarrow PO(\lambda),\ (p \wedge A) \Rightarrow pre(\lambda),\ (post(\lambda) \wedge A) \Rightarrow q}{\Box A \rightarrow \{p\}\ PO(\lambda)\ \{q\}}$$

Cnstr-Equiv: $$\frac{\Box A \rightarrow PO(\lambda),\ A \Rightarrow (I_{PO(\lambda)} = I_{PO'(\lambda)}),\ PO'(\lambda) \text{ is self consistent}}{\Box A \rightarrow PO'(\lambda)}$$

Example Revisited

We illustrate the deductive system for constrained proof outlines by proving that $y=3$ holds upon termination, when the program of Figure 4.1 is executed by a single processor using a fixed-priority scheduler with process λ_1 having higher priority than λ_2.

Recall that a fixed-priority scheduler rules out allocating a processor to any but the highest-priority processes, where a fixed priority value v_π is associated with each process π. The formulation of this restriction using the assertion language of our Proof Outline Logic closely parallels our TLA formulation in §3.3.

As before, for N the number of processors, we define:

$Alloc(N)$: $(\#\pi \in \Pi:\ active_\pi) \leq N$

$Run(\pi)$: $active_\pi \Rightarrow \pi \in TOP(N, \Pi)$

These state that variable $active_\pi$ is *true* for the N highest ranked different processes π. To stipulate that $active_\pi$ be *true* in order for a process to execute an atomic action, let $Lab(\lambda_\pi)$ be the set of labels for process π. Execution of an atomic action from π causes control variables to change for some $\lambda' \in Lab(\lambda_\pi)$.

$Pgrs(\pi)$: $(\Theta true \ \wedge \bigvee_{\lambda' \in Lab(\lambda_\pi)} (at(\lambda') \neq \Theta at(\lambda'))) \ \Rightarrow \ \Theta active_\pi$

The rank $rank_\pi$ of a process depends on whether or not that process has terminated. Since we assume that process π has label λ_π, that process has not terminated if $in(\lambda_\pi)$ is *true*. We thus can assign values to $rank_\pi$ using v_π as follows.

$Prio(\pi)$: $(in(\lambda_\pi) \Rightarrow (rank_\pi = v_\pi)) \wedge (\neg in(\lambda_\pi) \Rightarrow (rank_\pi = 0))$

Combining these, we obtain an assertion *FixedPrio* which characterizes a fixed-priority scheduler.

$FixedPrio$: $Alloc(N) \ \wedge \ (\forall \pi \in \Pi:\ Run(\pi) \wedge Pgrs(\pi) \wedge Prio(\pi))$

To conclude that $y=3$ holds upon termination of program λ in Figure 4.1, we prove $\Box FixedPrio \rightarrow PO(\lambda)$ a theorem, where $post(\lambda) \Rightarrow y=3$. We assume $N=1$, $v_{\lambda_1}=2$, and $v_{\lambda_2}=1$.

Using Assig2 (of Appendix A), we get:

$$\{at(\lambda_2)\}\ \ \lambda_{11}: [y := 1]\ \ \{at(\lambda_2)\} \tag{4.8}$$

$$\{at(\lambda_2)\}\ \ \lambda_{12}: [y := 2]\ \ \{at(\lambda_2)\} \tag{4.9}$$

With Conseq (of Appendix A), we can strengthen the precondition of (4.8) and (4.9) as well as weakening the postconditions of both—in preparation for using Cnstr-Assig with $\Box FixedPrio$

$$\{at(\lambda_2) \wedge FixedPrio\}\ \ \lambda_{11}: [y := 1]\ \ \{at(\lambda_2) \vee \neg FixedPrio\} \tag{4.10}$$

$$\{at(\lambda_2) \wedge FixedPrio\}\ \ \lambda_{12}: [y := 2]\ \ \{true \vee \neg FixedPrio\} \tag{4.11}$$

Using Cnstr-Assig we now obtain:

$$\Box FixedPrio \rightarrow \{at(\lambda_2)\}\ \ \lambda_{11}: [y := 1]\ \ \{at(\lambda_2)\} \tag{4.12}$$

$$\Box FixedPrio \rightarrow \{at(\lambda_2)\}\ \ \lambda_{12}: [y := 2]\ \ \{true\} \tag{4.13}$$

We combine these, using Cnstr-SeqComp to obtain a constrained proof outline for process λ_1.

$$\begin{array}{l} \Box FixedPrio \rightarrow \{at(\lambda_2)\} \\ \qquad \lambda_1: [\ \{at(\lambda_2)\}\ \ \lambda_{11}: [y := 1]\ \ \{at(\lambda_2)\}\ ; \\ \qquad\quad \{at(\lambda_2)\}\ \ \lambda_{12}: [y := 2]\ \ \{true\}] \\ \qquad \{true\} \end{array} \tag{4.14}$$

A proof outline for process λ_2 is constructed by starting with Assig1 (of Appendix A).

$$\{3=3\}\ \ \lambda_2: [y:=3]\ \ \{y=3\} \tag{4.15}$$

In preparation for using Cnstr-Assig, the precondition is strengthened and postcondition is weakened.

$$\{true \wedge FixedPrio\}\ \ \lambda_2: [y:=3]\ \ \{y=3 \vee \neg FixedPrio\} \tag{4.16}$$

We now can use Cnstr-Assig to obtain a constrained proof outline for process λ_2.

$$\Box FixedPrio \rightarrow \{true\}\ \ \lambda_2: [y:=3]\ \ \{y=3\} \tag{4.17}$$

Finally, we use Cnstr-ParComp to combine (4.14) and (4.17):

$$\begin{array}{l} \Box FixedPrio \rightarrow \{at(\lambda_2)\} \\ \qquad \lambda: [\{at(\lambda_2)\} \\ \qquad\quad \lambda_1: [\ \{at(\lambda_2)\}\ \ \lambda_{11}: [y := 1]\ \ \{at(\lambda_2)\}\ ; \\ \qquad\qquad \{at(\lambda_2)\}\ \ \lambda_{12}: [y := 2]\ \ \{true\}] \\ \qquad\quad \{true\} \\ \qquad\quad // \\ \qquad\quad \{true\}\ \ \lambda_2: [y:=3]\ \ \{y=3\}] \\ \qquad \{y=3\} \end{array} \tag{4.18}$$

This requires that we discharge the following interference-freedom requirements:

$$\Box FixedPrio \rightarrow \{at(\lambda_{11}) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \ \lambda_{11}: [y := 1]\ \ \{I_{PO(\lambda_2)}\} \tag{4.19}$$

$$\Box FixedPrio \rightarrow \{at(\lambda_{12}) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \ \lambda_{12}: [y := 2]\ \ \{I_{PO(\lambda_2)}\} \tag{4.20}$$

$$\Box FixedPrio \rightarrow \{at(\lambda_2) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)}\}\ \ \lambda_2: [y := 3]\ \ \{I_{PO(\lambda_1)}\} \tag{4.21}$$

where:

$$\begin{array}{lll} I_{PO(\lambda_1)}: & & (at(\lambda_1) \Rightarrow at(\lambda_2)) \wedge (after(\lambda_1) \Rightarrow true) \\ & \wedge & (at(\lambda_{11}) \Rightarrow at(\lambda_2)) \wedge (after(\lambda_{11}) \Rightarrow at(\lambda_2)) \\ & \wedge & (at(\lambda_{12}) \Rightarrow at(\lambda_2)) \wedge (after(\lambda_{12}) \Rightarrow true) \end{array}$$

$$I_{PO(\lambda_2)}: \quad (at(\lambda_2) \Rightarrow true) \wedge (after(\lambda_2) \Rightarrow y=3)$$

$I_{PO(\lambda_1)}$ and $I_{PO(\lambda_2)}$ can be simplified, using ordinary Predicate Logic, resulting in:

$$I_{PO(\lambda_1)}: \quad (at(\lambda_1) \vee at(\lambda_{11}) \vee after(\lambda_{11}) \vee at(\lambda_{12})) \ \Rightarrow\ at(\lambda_2)$$

$$I_{PO(\lambda_2)}: \quad after(\lambda_2) \Rightarrow y=3$$

To prove formula (4.19), observe that according to the definitions of $I_{PO(\lambda_1)}$, $I_{PO(\lambda_2)}$, and *FixedPrio*:

$$(at(\lambda_{11}) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)} \wedge FixedPrio) \;\Rightarrow\; at(\lambda_2)$$

$$at(\lambda_2) \Rightarrow (I_{PO(\lambda_2)} \vee \neg FixedPrio)$$

Applying Conseq and then Cnstr-Assig to (4.8) we obtain (4.19). The proof of (4.20) is virtually identical.

Proving formula (4.21) illustrates the role of environment $\Box FixedPrio$. Using Assig3, Equiv, and Conseq it is not difficult to prove:

$$\{at(\lambda_2)\}\ \ \lambda_2\colon [y := 3]\ \ \{Q\colon\ \ \Theta at(\lambda_2) \wedge \neg at(\lambda_2) \wedge at(\lambda_1) = \Theta at(\lambda_1) \wedge (in(\lambda_1) \vee after(\lambda))\}$$

$$\{in(\lambda_1) \Rightarrow (active_{\lambda_1} \wedge \neg active_{\lambda_2})\}\ \ \lambda_2\colon [y := 3]\ \ \{R\colon\ \ \Theta(in(\lambda_1) \Rightarrow (active_{\lambda_1} \wedge \neg active_{\lambda_2}))\}$$

Each of these preconditions is implied by $at(\lambda_2) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)} \wedge FixedPrio$, so we can use Conseq to strengthen each and deduce:

$$\{at(\lambda_2) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)} \wedge FixedPrio\}\ \ \lambda_2\colon [y := 3]\ \ \{Q\}$$

$$\{at(\lambda_2) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)} \wedge FixedPrio\}\ \ \lambda_2\colon [y := 3]\ \ \{R\}$$

Therefore, by Conj, we obtain:

$$\{at(\lambda_2) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)} \wedge FixedPrio\}\ \ \lambda_2\colon [y := 3]\ \ \{Q \wedge R\}$$

We now use Conseq to infer that $I_{PO(\lambda_1)}$ or $\neg FixedPrio$ holds whenever $Q \wedge R$ does by proving:

$$(in(\lambda_1) \wedge Q \wedge R) \Rightarrow \neg FixedPrio$$

$$(after(\lambda_1) \wedge Q \wedge R) \Rightarrow I_{PO(\lambda_1)}$$

Using these with Conseq, we conclude:

$$\{at(\lambda_2) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)} \wedge FixedPrio\}\ \ \lambda_2\colon [y := 3]\ \ \{I_{PO(\lambda_1)} \vee \neg FixedPrio\}$$

Cnstr-Assig now allows us to conclude (4.21), as is desired.

4.3. An Even Older Recipe

The notion of a constrained proof outline is not new. In [LS85] a similar idea was discussed in connection with reasoning about aliasing and other artifacts of variable declarations. The aliasing of two variables imposes the constraint that their values are equal; the declaration of a variable imposes a constraint on the values that variable may store. Constrained proof outlines, because they provide a basis for proving properties of programs whose execution depends on constraints being preserved, are thus a way to reason about aliasing and declarations. An even earlier call for a construct like our constrained proof outlines appears in [L80]. There, Lamport claims that such proof outlines would be helpful in proving certain types of safety properties of concurrent programs.

5. Discussion

Related Work

Our work is perhaps closest in spirit to the various approaches for reasoning about open systems. An *open system* is one that interacts with its environment through shared memory or communication. The execution of such a system is commonly modeled as an interleaving of steps by the system and steps by the environment. Since an open system is not expected to function properly in an

arbitrary environment, its specification typically will contain explicit assumptions about the environment. Such specifications are called *assume-guarantee* specifications because they guarantee behavior when the environment satisfies some assumptions. Logics for verifying safety properties of assume-guarantee specifications are discussed in [FFG92], [J83], and [MC81]; liveness properties are treated in [AL91], [BKP84], and [P85]; and model-checking techniques based on assume-guarantee specifications are introduced in [CLM89] and [GL91].

Our approach differs from this open systems work both in the role played by the environment and in how state changes are made by the environment. We use the environment to represent aspects of the computation model, not as an abstraction of the behaviors for other agents that will run concurrently with the system. This is exactly what is advocated in [E83] for reasoning about fair computations in temporal logic. Second, in our approach, every state change obeys constraints defined by the environment. State changes attributed to the environment are not interleaved with system actions, as is the case with the open systems view.

Our view of the environment and the view employed for open systems are complementary. They address different problems. Both notions of environment can coexist in a single logic. Open systems and their notion of an environment are an accepted part of the verification scene. This paper explores the use of a new type of environment. Our environments allow logics to be extended for various computational models. As a result, a single principle suffices for reasoning about the effects of schedulers, real-time models, resource constraints, and fairness assumptions. Thus, one does not have to redesign a programming logic every time the computational model is changed.

In terms of program construction, our notion of an environment is closely related to the notions of superposition discussed in [BF88] [CM88] [K93]. There, the superposition of two programs S and T is a single program, whose steps involve steps of S and a steps of T. Thus, in terms of TLA, the superposition of two actions is simply their conjunction. Our work extends the domain of applicability for superposition by allowing one component of a superposition to characterize aspects of a computational model.

Redefining Feasible Behaviors

The definition of §2 for the feasible behaviors of a program S under an environment E is not the only plausible one. Every infeasible behavior of S ruled out by E has a maximal finite prefix (possibly empty) that agrees with a prefix of some behavior in E. Such a prefix can be regarded as modeling an execution of S that aborts due to the constraints of E, and this prefix might well be included in the set of feasible behaviors.

For example, consider executing the program

T: $x := 0$; **do** $i := 1$ **to** 5: $x := x+1$ **od**

in an environment that constrains x to be between 0 and 3 (i.e., x is represented using 2 bits). The alternative definition of feasible behaviors would include prefixes of behaviors of T up until the point where an attempt is made to store 4 into x. Using the definition of §2, the set of feasible behaviors would be empty.

The alternative definition of feasible behaviors for a program S under an environment E,

$$([\![S]\!] \cap [\![E]\!]) \cup (\mathit{prefix}([\![S]\!]) \cap \mathit{prefix}([\![E]\!])), \tag{5.1}$$

admits reasoning about feasible, but incomplete, executions of a program under a given environment.

Unfortunately, we have been unable to identify reduction principles for definition (5.1). It remains an open question how to extend a logic for *Sat* into a logic for *ESat* given this definition.

6. Conclusion

In this paper, we have shown that environments are a powerful device for making aspects of a computational model explicit in a programming logic. We have shown how environments can be used to formalize schedulers and real-time; a forthcoming paper will show how they can be applied to hybrid systems, where a continuous transition system governs changes to certain variables.

We have given two semantic principles, program reduction and property reduction, for extending programming logics to enable reasoning about program executions feasible under a specified environment. Having such principles means that a new logic need not be designed every time the computational model served by an extant logic is changed. For example, in this paper, we give a new way to reason about real-time in TLA and in Hoare-style programming logics. We also derive the first Hoare-style logic for reasoning about schedulers.

The basic idea of reasoning about program executions that are feasible in some environment is not new, having enjoyed widespread use in connection with open systems. The basic idea of augmenting the individual state transitions caused by the atomic actions in a program is not new, either. It underlies methods for program composition by superposition, methods for reasoning about aliasing, and proposals for verifying certain types of safety properties. What is new is our use of environments for describing aspects of a computational model and our unifying semantic principles for reasoning about environments. Extensions to a computational model can now be translated into extensions to an existing programming logic, by applying one of two simple semantic principles.

Acknowledgments

We are grateful to Leslie Lamport and Jay Misra for discussions and helpful comments on this material.

References

[AL91] Abadi, M. and L. Lamport. An old-fashioned recipe for real-time. *Real-time: Theory in Practice*, J.W. de Bakker and W.-P. de Roever and G. Rozenberg eds., Lecture Notes in Computer Science Vol. 600, Springer-Verlag, New York, 1989, 1-27.

[BMS93] Babaoglu, O, K. Marzullo, and F.B. Schneider. A formalization of priority inversion. *Real-Time Systems 5* (1993), 285-303.

[BKP84] Barringer, H., R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. *Proceedings 16th Annual ACM Symposium on Theory of Computing* (Washington, D.C., April 1984), 51-63.

[BF88] Francez, N. and L. Bouge. A Compositional Approach to Superimposition. *Proceedings 15th ACM Symp. on Principles of Programming Languages* (San Diego, Ca, Jan. 1988), 240-249.

[CM88] Chandy, M. and J. Misra. *Parallel program design*. Addison-Wesley, 1988.

[CLM89] Clarke, E.M., D.E. Long, and K.L. McMillan. Compositional model checking. *Proceedings 4th IEEE Symposium on Logic in Computer Science* (Palo Alto, CA. June 1989), 353-362.

[E83] Emerson, E.A. Alternative semantics for temporal logics. *Theoretical Computer Science 26* (1983), 121-130.

[EH86] Emerson, E.A and J.Y. Halpern. Sometimes and Not Never Revisited: On Branching Versus Linear Time. *Journal of the ACM 33*, 1 (1986), 151-178.

[FFG92] Fix, L., N. Francez, and O. Grumberg. Program composition via unification. *Automata, Languages and Programming*, Lecture Notes in Computer Science Vol. 623, Springer-Verlag, New York, 1989, 672-684.

[GL91] Grumberg, O. and D.E. Long. Model checking and modular verification. *CONCUR'91, 2nd International Conference on Concurrency Theory*, Lecture Notes in Computer Science Vol. 527, Springer-Verlag, New York, 1991, 250-265.

[H69] Hoare, C.A.R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576-580.

[J83] Jones, C.B. Specification and design of (parallel) programs. *Information Processing '83*, R.E.A. Mason ed., Elsevier Science Publishers, North Holland, The Netherlands, 321-332.

[K93] Katz, S. A Superimposition Control Construct for Distributed Systems. *ACM TOPLAS 15*, 2 (April 1993), 337-356.

[L80] Lamport, L. The "Hoare Logic" of concurrent programs. *Acta Informatica 14* (1980), 21-37.

[L91] Lamport, L. The temporal logic of actions. Technical report 79, Systems Research Center, Digital Equipment Corp. Palo Alto, CA. Dec, 1991.

[LS84] Lamport, L. and F.B. Schneider. The "Hoare Logic" of CSP and all that. *ACM TOPLAS 6*, 2 (April 1984), 281-296.

[LS85] Lamport, L. and F.B. Schneider. Constraints: A uniform approach to aliasing and typing. *Conference Record of Twelfth Annual ACM Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 1986), 205-216.

[LJJ93] Liu, Z., M. Joseph, and T. Janowski. Specifying schedulability for real-time programs. Technical report, Department of Computer Science, University of Warwick, Coventry, United Kingdom.

[MP89] Manna, Z and A. Pnueli. The anchored version of the temporal framework. *Linear time, branching time and partial order in Logics and models for concurrency*, J.W. de Bakker and W.-P. de Roever and G. Rozenberg eds., Lecture Notes in Computer Science Vol. 354, Springer-Verlag, New York, 1989, 201-284.

[MC81] Misra, J. and M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering, SE-7*, 4 (July 1981), 417-426.

[OG76] Owicki, S. and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica 6* (1976), 319-340.

[P85] Pnueli, A. In transition from global to modular temporal reasoning about programs. *Logics and models of concurrent systems*, K.R. Apt ed., NATO ASI Series, Vol. F13, Springer-Verlag, 1985, 123-144.

[S94] Schneider, F.B. *On concurrent programming*. To appear.

[SRL90] Sha, L., R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers* C-39, 1175-1185.

Appendix A: A Logic of Proof Outlines

The deductive system for reasoning about assertions includes the axioms and inference rules of first-order predicate logic. It also axiomatizes theories for the datatypes of program variables and expressions. Perhaps the only aspect of this axiomatization that might be unfamiliar concerns Θ. It will include axioms like:

$$\Theta true \Rightarrow (\Theta E(T_1, ..., T_n) \quad = \quad E(\Theta T_1, ..., \Theta T_n))$$

In order to reason about control variables in program states, each program λ gives rise to a set of axioms. These axioms characterize the constraints of Figure 4.2. For every label $\lambda' \in Lab(\lambda)$:

CP0: $in(\lambda) \neq after(\lambda)$

CP1: $\neg (at(\lambda') \wedge after(\lambda'))$

CP2: $at(\lambda') \Rightarrow in(\lambda')$

CP3: If λ' labels $[x := E]$: $\quad at(\lambda') = in(\lambda')$

CP4: If λ' labels $[\lambda_1 : [S];\ \lambda_2 : [T]]$:
- (a) $at(\lambda') = at(\lambda_1)$
- (b) $at(\lambda_2) = after(\lambda_1)$
- (c) $after(\lambda') = after(\lambda_2)$
- (d) $in(\lambda') = ((in(\lambda_1) \wedge \neg in(\lambda_2)) \ \vee \ (\neg in(\lambda_1) \wedge in(\lambda_2)))$
- (e) $(in(\lambda_1) \vee in(\lambda_2)) \Rightarrow in(\lambda')$

CP5: If λ' labels $[\lambda_1 : [S] \ // \ \lambda_2 : [T]]$:
- (a) $at(\lambda') = (at(\lambda_1) \wedge at(\lambda_2))$
- (b) $after(\lambda') = (after(\lambda_1) \wedge after(\lambda_2))$
- (d) $in(\lambda') = ((in(\lambda_1) \vee after(\lambda_1)) \wedge (in(\lambda_2) \vee after(\lambda_2)) \wedge \neg (after(\lambda_1) \wedge after(\lambda_2)))$
- (d) $(in(\lambda_1) \vee in(\lambda_2)) \Rightarrow in(\lambda')$

For reasoning about proof outlines, we have the following. First, here are the axioms for assignment statements.

Assig1: If no free variable in p is a control variable and p^x_E denotes the predicate logic formula that results from replacing every free occurrence of x in p that is not in the scope of Θ with E:

$$\{p^x_E\} \ \ \lambda : [x := E] \ \ \{p\}$$

Assig2: If λ' is a label from a program that is parallel to that containing λ, and $cp(\lambda')$ denotes any of the control variables $at(\lambda')$, $after(\lambda')$, $in(\lambda')$ or their negations:

$$\{cp(\lambda')\} \ \ \lambda : [x := E] \ \ \{cp(\lambda')\}$$

Assig3: $\{p\} \ \ \lambda : [x := E] \ \ \{\Theta p\}$

Sequential composition is handled by a single inference rule.

SeqComp:
$$\frac{PO(\lambda_1),\quad PO(\lambda_2),\quad post(\lambda_1) \Rightarrow pre(\lambda_2)}{\{pre(\lambda_1)\}\ \ \lambda\colon [\, PO(\lambda_1);\ PO(\lambda_2)]\ \ \{post(\lambda_2)\}}$$

The parallel composition rule is based on the formulation of interference freedom [OG76] of proof outlines given in [LS84]. Two proof outlines $PO(\lambda_1)$ and $PO(\lambda_2)$ are *interference free* iff

For all $\lambda_\alpha \in Assig(\lambda_1)$, where λ_α is the assignment $\lambda_\alpha\colon [x := E]$:

$$\{at(\lambda_\alpha) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \ \lambda_\alpha\colon [x := E]\ \ \{I_{PO(\lambda_2)}\}$$

For all $\lambda_\alpha \in Assig(\lambda_2)$, where λ_α is the assignment $\lambda_\alpha\colon [x := E]$:

$$\{at(\lambda_\alpha) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)}\}\ \ \lambda_\alpha\colon [x := E]\ \ \{I_{PO(\lambda_1)}\}$$

ParComp:
$$\frac{PO(\lambda_1),\quad PO(\lambda_2),\quad PO(\lambda_1) \text{ and } PO(\lambda_2) \text{ are interference free}}{\{pre(\lambda_1) \wedge pre(\lambda_2)\}\ \ \lambda\colon [\, PO(\lambda_1)\ //\ PO(\lambda_2)]\ \ \{post(\lambda_1) \wedge post(\lambda_2)\}}$$

Finally, three rules allow us to modify assertions based on the deductive system for the assertion language. Recall, $\{p\}\ PO(\lambda)\ \{q\}$ denotes a proof outline that is identical to $PO(\lambda)$ but with p replacing $pre(\lambda)$ and q replacing $post(\lambda)$.

Conj:
$$\frac{\{p_1\}\ \ \lambda\colon [x := E]\ \ \{q_1\} \qquad \{p_2\}\ \ \lambda\colon [x := E]\ \ \{q_2\}}{\{p_1 \wedge p_2\}\ \ \lambda\colon [x := E]\ \ \{q_1 \wedge q_2\}}$$

Conseq:
$$\frac{PO(\lambda),\quad p \Rightarrow pre(\lambda),\quad post(\lambda) \Rightarrow q}{\{p\}\ PO(\lambda)\ \{q\}}$$

Equiv:
$$\frac{PO(\lambda),\quad I_{PO(\lambda)} = I_{PO'(\lambda)},\quad I_{PO'(\lambda)} \text{ self consistent}}{PO'(\lambda)}$$

Appendix B: Soundness and Completeness for Constrained Proof Outlines

We now prove soundness and relative completeness for the Logic of Constrained Proof Outlines given in section 4.2. Specifically, we prove that Cnstr-Assig, Cnstr-SeqComp, Const-ParComp and Cnstr-Equiv are sound. We also prove that Cnstr-Assig, Cnstr-SeqComp and Const-ParComp comprise a complete deductive system relative to the deductive system of Appendix A for ordinary proof outlines. (Cnstr-Conseq and Cnstr-Equiv of section 4.2 are not necessary for completeness.)

Lemma (Soundness of Cnstr-Assig): The rule

$$\text{Cnstr-Assig:}\qquad \frac{\{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}}{\Box A \rightarrow \{p\}\ \ \lambda\colon [x := E]\ \ \{q\}}$$

is sound.

Proof. Assume that hypothesis $\{p \wedge A\}\ \lambda\colon [x := E]\ \{q \vee \neg A\}$ is valid. We show that if $\langle \sigma, i, j\rangle \in [\![\, \lambda \,]\!]$ holds, then $\langle \sigma, i, j\rangle \in [\![\, \Box A \rightarrow \{p\}\ \lambda\colon [x := E]\ \{q\} \,]\!]$ holds, and thus that $\Box A \rightarrow \{p\}\ \lambda\colon [x := E]\ \{q\}$ is valid.

If $\{p \wedge A\}\ \lambda\colon [x := E]\ \{q \vee \neg A\}$ is valid then, by definition, $[\![\, \lambda \,]\!] \subseteq [\![\, PO(\lambda) \,]\!]$ holds, where $PO(\lambda)$ is $\{p \wedge A\}\ \lambda\colon [x := E]\ \{q \vee \neg A\}$. This implies that for any $\langle \sigma, i, j\rangle \in [\![\, \lambda \,]\!]$ one of the following must hold:

$$\sigma[..i] \not\models I_{PO(\lambda)} \tag{B.1.1}$$

$$\text{For all } k,\ \ i \le k \le j\colon\ \sigma[..k] \models I_{PO(\lambda)} \tag{B.1.2}$$

where $I_{PO(\lambda)}$: $(at(\lambda) \Rightarrow p \wedge A)\ \ \wedge\ \ (after(\lambda) \Rightarrow q \vee \neg A)$

We consider two cases.

Case 1: Assume $\langle \sigma, i, j\rangle \in \overline{[\![\, \Box A \,]\!]}$. According to definition (4.6), $\langle \sigma, i, j\rangle$ is in $[\![\, \Box A \rightarrow \{p\}\ \lambda\colon [x := E]\ \{q\} \,]\!]$.

Case 2: Assume $\langle \sigma, i, j\rangle \in [\![\, \Box A \,]\!]$. According to the definition of $[\![\, \Box A \,]\!]$:

$$\text{For all } k,\ \ i \le k \le j\colon\ \sigma[..k] \models A \tag{B.1.3}$$

It suffices to prove that if (B.1.1) holds or (B.1.2) holds, then $\langle \sigma, i, j\rangle \in [\![\, \Box A \rightarrow \{p\}\ \lambda\colon [x := E]\ \{q\} \,]\!]$.

Case 2.1: Assume (B.1.1) holds. Thus

$$\sigma[..i] \models ((at(\lambda) \wedge (\neg p \vee \neg A)) \vee (after(\lambda) \wedge (\neg q \wedge A)))$$

holds. Conjoining (B.1.3), we conclude

$$\sigma[..i] \models ((at(\lambda) \wedge \neg p) \vee (after(\lambda) \wedge \neg q))$$

which implies $\sigma[..i] \not\models (at(\lambda) \Rightarrow p) \wedge (after(\lambda) \Rightarrow q)$. Because $\{p\}\ \lambda\colon [x := E]\ \{q\}$ is self consistent, by definition (4.4) we have that $\langle \sigma, i, j\rangle \in [\![\, \{p\}\ \lambda\colon [x := E]\ \{q\} \,]\!]$ holds. Hence, by definition (4.6) of the property defined by a constrained proof outline, $\langle \sigma, i, j\rangle \in [\![\, \Box A \rightarrow \{p\}\ \lambda\colon [x := E]\ \{q\} \,]\!]$ holds.

Case 2.2: Assume (B.1.2) holds. Conjoining (B.1.3), we conclude

$$\text{For all } k,\ i \le k \le j\colon\ \sigma[..k] \models ((at(\lambda) \Rightarrow p) \wedge (after(\lambda) \Rightarrow q)).$$

Because $\{p\}\ \lambda\colon [x := E]\ \{q\}$ is self consistent, by definition (4.4) we have that $\langle \sigma, i, j\rangle \in [\![\, \{p\}\ \lambda\colon [x := E]\ \{q\} \,]\!]$ holds, so by definition (4.6), $\langle \sigma, i, j\rangle \in [\![\, \Box A \rightarrow \{p\}\ \lambda\colon [x := E]\ \{q\} \,]\!]$ holds as well. □

Lemma (Soundness of Cnstr-SeqComp): The rule

$$\text{Cnstr-SeqComp:}\quad \frac{\begin{array}{l}\Box A \rightarrow PO(\lambda_1),\ \Box A \rightarrow PO(\lambda_2) \\ (A \wedge post(\lambda_1)) \Rightarrow pre(\lambda_2)\end{array}}{\Box A \rightarrow \{pre(\lambda_1)\}\ \ \lambda\colon [PO(\lambda_1);\ PO(\lambda_2)]\ \ \{post(\lambda_2)\}}$$

is sound.

Proof. Assume that the hypotheses are valid. Therefore, we have that $PO(\lambda_1)$ is self consistent, $PO(\lambda_2)$ is self consistent, and:

$$[\![\ \lambda_1\ ]\!] \subseteq ([\![\ PO(\lambda_1)\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]}) \tag{B.2.1}$$

$$[\![\ \lambda_2\ ]\!] \subseteq ([\![\ PO(\lambda_2)\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]}) \tag{B.2.2}$$

$$(A \wedge post(\lambda_1)) \Rightarrow pre(\lambda_2) \tag{B.2.3}$$

To establish validity of the rule's conclusion, we must prove that $[\![\ \lambda\ ]\!] \subseteq [\![\ \Box A \rightarrow PO(\lambda)\ ]\!]$, where $PO(\lambda)$ is $\{pre(\lambda_1)\}\ \ \lambda\colon [PO(\lambda_1);\ PO(\lambda_2)]\ \ \{post(\lambda_2)\}$. We do this by proving $\langle \sigma, i, j\rangle \in [\![\ \lambda\ ]\!]$ implies $\langle \sigma, i, j\rangle \in [\![\ PO(\lambda)\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]}$), where, according to definition (4.3) of $I_{PO(\lambda)}$, we have:

$$I_{PO(\lambda)}\colon\ \ I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1)) \wedge (after(\lambda) \Rightarrow post(\lambda_2))$$

Let $\langle \sigma, i, j\rangle \in [\![\ \lambda\ ]\!]$ hold. We consider two cases.

Case 1: Assume $\langle \sigma, i, j\rangle \in \overline{[\![\ \Box A\ ]\!]}$. According to definition (4.6) of a constrained proof outline, $\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow PO(\lambda)\ ]\!]$ holds.

Case 2: Assume $\langle \sigma, i, j\rangle \in [\![\ \Box A\ ]\!]$. In order to conclude $\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow PO(\lambda)\ ]\!]$ holds, we must show that $\langle \sigma, i, j\rangle \in [\![\ PO(\lambda)\ ]\!]$ holds. According to the definition of sequential composition $[\![\ \lambda\colon [\lambda_1;\ \lambda_2]\ ]\!]$, there are three cases:

(2.1) For all k, $i \le k \le j$: $\sigma[k] \not\models (in(\lambda_2) \vee after(\lambda_2))$ and $\langle \sigma, i, j\rangle \in [\![\ \lambda_1\ ]\!]$.

(2.2) For all k, $i \le k \le j$: $\sigma[k] \not\models (in(\lambda_1) \vee after(\lambda_1))$ and $\langle \sigma, i, j\rangle \in [\![\ \lambda_2\ ]\!]$.

(2.3) Exists n, $i \le n \le j$: $\langle \sigma, i, n\rangle \in [\![\ \lambda_1\ ]\!]$ and
$\langle \sigma, n, j\rangle \in [\![\ \lambda_2\ ]\!]$ and
$\sigma[n] \models (after(\lambda_1) \wedge at(\lambda_2))$ and
for all k, $1 \le k < n$: $\sigma[k] \not\models (in(\lambda_2) \vee after(\lambda_2))$ and
for all k, $n < k \le i$: $\sigma[k] \not\models (in(\lambda_1) \vee after(\lambda_1))$

Case 2.1: Assume that for all k, $i \le k \le j$: $\sigma[k] \not\models (in(\lambda_2) \vee after(\lambda_2))$ and $\langle \sigma, i, j\rangle \in [\![\ \lambda_1\ ]\!]$ hold. According to (B.2.1), $\langle \sigma, i, j\rangle \in ([\![\ PO(\lambda_1)\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]})$. Given assumption (of Case 2) $\langle \sigma, i, j\rangle \in [\![\ \Box A\ ]\!]$, we conclude $\langle \sigma, i, j\rangle \in [\![\ PO(\lambda_1)\ ]\!]$. Thus, by definition (4.4) of $[\![\ PO(\lambda_1)\ ]\!]$, we have that either $\sigma[..i] \not\models I_{PO(\lambda_1)}$ or else for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_1)}$. Since $at(\lambda_1) = at(\lambda)$ due to Control Predicate Axiom CP4(a) of Appendix A and $pre(\lambda)$ is $pre(\lambda_1)$, we conclude that either

$$\sigma[..i] \not\models (I_{PO(\lambda_1)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1))) \quad \text{or else} \tag{B.2.5}$$

for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_1)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1))$. (B.2.6)

From assumption (of Case 2.1) for all k, $i \le k \le j$: $\sigma[k] \not\models (in(\lambda_2) \vee after(\lambda_2))$, we conclude that for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_2)}$. And, because $after(\lambda_2) = after(\lambda)$ due to Control Predicate Axiom CP4(c) of Appendix A, we have for all k, $i \le k \le j$: $\sigma[..k] \models (I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2)))$. Conjoining this with (B.2.5) and (B.2.6) we get that either $\sigma[..i] \not\models I_{PO(\lambda)}$ or else for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda)}$ holds. Since $PO(\lambda)$ is self consistent (because $(A \wedge post(\lambda_1)) \Rightarrow pre(\lambda_2)$ and both $PO(\lambda_1)$ and $PO(\lambda_2)$ are self consistent), we have $\langle \sigma, i, j \rangle \in [\![PO(\lambda)]\!]$. Thus, by definition (4.6) we conclude that $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO(\lambda)]\!]$.

Case 2.2: Assume for all k, $i \le k \le j$: $\sigma[k] \not\models (in(\lambda_1) \vee after(\lambda_1))$ and $\langle \sigma, i, j \rangle \in [\![\lambda_2]\!]$ hold. According to (B.2.2), $\langle \sigma, i, j \rangle \in ([\![PO(\lambda_2)]\!] \cup \overline{[\![\Box A]\!]})$. Given assumption (of Case 2) $\langle \sigma, i, j \rangle \in [\![\Box A]\!]$, we conclude $\langle \sigma, i, j \rangle \in [\![PO(\lambda_2)]\!]$. Thus, by definition (4.4) of $[\![PO(\lambda_2)]\!]$, we have that either $\sigma[..i] \not\models I_{PO(\lambda_2)}$ or else for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_2)}$. Since $after(\lambda_2) = after(\lambda)$ due to Control Predicate Axiom CP4(c) of Appendix A, we conclude that either

$\sigma[..i] \not\models (I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2)))$ or else (B.2.7)

for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2))$. (B.2.8)

From assumption (of Case 2.2) for all k, $i \le k \le j$: $\sigma[k] \not\models (in(\lambda_1) \vee after(\lambda_1))$ we conclude that for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_1)}$. And, because $at(\lambda_1) = at(\lambda)$ due to Control Predicate Axiom CP4(a) of Appendix A, we have for all k, $i \le k \le j$: $\sigma[..k] \models (I_{PO(\lambda_1)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1)))$. Conjoining this with (B.2.7) and (B.2.8) we get $\sigma[..i] \not\models I_{PO(\lambda)}$ or else for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda)}$ holds. Since $PO(\lambda)$ is self consistent (because $(A \wedge post(\lambda_1)) \Rightarrow pre(\lambda_2)$ and both $PO(\lambda_1)$ and $PO(\lambda_2)$ are self consistent), we have $\langle \sigma, i, j \rangle \in [\![PO(\lambda)]\!]$. Thus, by definition (4.6) we conclude that $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO(\lambda)]\!]$.

Case 2.3: Assume that there exists n, $i \le n \le j$, such that:

$\langle \sigma, i, n \rangle \in [\![\lambda_1]\!]$ (B.2.9)
$\langle \sigma, n, j \rangle \in [\![\lambda_2]\!]$ (B.2.10)
$\sigma[n] \models (after(\lambda_1) \wedge at(\lambda_2))$ (B.2.11)
for all k, $1 \le k < n$: $\sigma[k] \not\models (in(\lambda_2) \vee after(\lambda_2))$ (B.2.12)
for all k, $n < k \le i$: $\sigma[k] \not\models (in(\lambda_1) \vee after(\lambda_1))$ (B.2.13)

If $\sigma[..i] \not\models I_{PO(\lambda)}$ then, by definition, $\langle \sigma, i, j \rangle \in [\![PO(\lambda)]\!]$. By definition (4.6), we conclude that $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO(\lambda)]\!]$.

Now suppose $\sigma[..i] \models I_{PO(\lambda)}$ holds.

First observe that due to Control Predicate Axiom CP4(a) and CP4(c) of Appendix A we have $at(\lambda) = at(\lambda_1)$ and $after(\lambda) = after(\lambda_2)$. Therefore, choosing $pre(\lambda_1)$ for $pre(\lambda)$ and choosing $post(\lambda_2)$ for $post(\lambda)$ we have:

$$I_{PO(\lambda_1)} \;=\; (I_{PO(\lambda_1)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1)) \tag{B.2.14}$$

$$I_{PO(\lambda_2)} \;=\; (I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2)) \tag{B.2.15}$$

Because $I_{PO(\lambda)}$ implies $I_{PO(\lambda_1)}$, $\sigma[..i] \models I_{PO(\lambda_1)}$ holds. From (B.2.9) and (B.2.1) we have $\langle \sigma, i, n \rangle \in [\![\Box A \rightarrow PO(\lambda_1)]\!]$. Given the assumption (of Case 2) that $\langle \sigma, i, j \rangle \in [\![\Box A]\!]$ we conclude that $\langle \sigma, i, n \rangle \in [\![PO(\lambda_1)]\!]$.

From $\langle \sigma, i, n \rangle \in [\![PO(\lambda_1)]\!]$ and assumption (above) $\sigma[..i] \models I_{PO(\lambda_1)}$ we have:

$$\text{For all } k,\ i \le k \le n\text{:}\ \ \sigma[..k] \models I_{PO(\lambda_1)} \tag{B.2.16}$$

By narrowing the range, we get:

$$\text{For all } k,\ i \le k < n\text{:}\ \ \sigma[..k] \models I_{PO(\lambda_1)}$$

And, from (B.2.14) we get

$$\text{For all } k,\ i \le k < n\text{:}\ \ \sigma[..k] \models I_{PO(\lambda_1)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1))$$

According to (B.2.12), $I_{PO(\lambda_2)}$ is trivially true in states $\sigma[..k]$ where $i \le k < n$. Moreover, from (B.2.15), $I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2))$ is also true in those states:

$$\text{For all } k,\ i \le k < n\text{:}\ \ \sigma[..k] \models (I_{PO(\lambda_1)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1)) \wedge I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2)))$$

Equivalently, we have for all k, $i \le k < n$: $\sigma[..k] \models I_{PO(\lambda)}$.

From (B.2.16), again by narrowing the range, we conclude $\sigma[..n] \models I_{PO(\lambda_1)}$. By definition, $I_{PO(\lambda_1)}$ implies $after(\lambda_1) \Rightarrow post(\lambda_1)$, so by conjoining (B.2.11), we infer

$$\sigma[..n] \models I_{PO(\lambda_1)} \wedge after(\lambda_1) \wedge post(\lambda_1) \wedge at(\lambda_2)$$

And, because of the assumption (Case 2) that $\langle \sigma, i, j \rangle \in [\![\Box A]\!]$, we have $\sigma[..n] \models A$. Thus we conclude:

$$\sigma[..n] \models A \wedge I_{PO(\lambda_1)} \wedge after(\lambda_1) \wedge post(\lambda_1) \wedge at(\lambda_2)$$

Using (B.2.3) we get:

$$\sigma[..n] \models I_{PO(\lambda_1)} \wedge after(\lambda_1) \wedge post(\lambda_1) \wedge at(\lambda_2) \wedge pre(\lambda_2)$$

$PO(\lambda_2)$ is self consistent, so $at(\lambda_2) \wedge pre(\lambda_2)$ implies $I_{PO(\lambda_2)}$ and we have:

$$\sigma[..n] \models I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}$$

Using (B.2.14) and (B.2.15), we get

$$\sigma[..n] \models I_{PO(\lambda)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1)) \wedge I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2)))$$

or equivalently $\sigma[..n] \models I_{PO(\lambda)}$.

From (B.2.10) and (B.2.1) we have $\langle \sigma, n, j \rangle \in [\![\Box A \rightarrow PO(\lambda_2)]\!]$. Given the assumption (of Case 2) that $\langle \sigma, i, j \rangle \in [\![\Box A]\!]$ we conclude that $\langle \sigma, n, j \rangle \in [\![PO(\lambda_2)]\!]$.

From $\langle \sigma, n, j \rangle \in [\![PO(\lambda_2)]\!]$ and $\sigma[..n] \models I_{PO(\lambda)}$ we have:

For all k, $n \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_2)}$

And, from (B.2.15) we get

For all k, $n \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2))$

According to (B.2.13), $I_{PO(\lambda_1)}$ is trivially true in traces $\sigma[..k]$ where $n \le k \le j$. Moreover, from (B.2.14), $I_{PO(\lambda_1)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1))$ is also true in those traces:

For all k, $n \le k \le j$: $\sigma[..k] \models (I_{PO(\lambda_1)} \wedge (at(\lambda) \Rightarrow pre(\lambda_1)) \wedge$
$I_{PO(\lambda_2)} \wedge (after(\lambda) \Rightarrow post(\lambda_2)))$,

or equivalently for all k, $n \le k \le j$: $\sigma[..k] \models I_{PO(\lambda)}$.

Having proved for all k, $i \le k < n$: $\sigma[..k] \models I_{PO(\lambda)}$ and for all k, $n \le k \le j$: $\sigma[..k] \models I_{PO(\lambda)}$ we conclude for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda)}$. This means that $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO(\lambda)]\!]$ holds. □

Lemma (Soundness of Cnstr-ParComp): The rule

$$\frac{\begin{array}{c}\Box A \rightarrow PO(\lambda_1), \quad \Box A \rightarrow PO(\lambda_2), \\ \Box A \rightarrow PO(\lambda_1) \text{ and } \Box A \rightarrow PO(\lambda_2) \text{ are interference free}\end{array}}{\Box A \rightarrow \{pre(\lambda_1) \wedge pre(\lambda_2)\}\ \ \lambda\colon [\ PO(\lambda_1)\ //\ PO(\lambda_2)]\ \ \{post(\lambda_1) \wedge post(\lambda_2)\}}$$

is sound.

Proof. Assume that the three hypotheses are valid. Therefore, we have that $PO(\lambda_1)$ is self consistent, $PO(\lambda_2)$ is self consistent, and:

$$[\![\ \lambda_1\ ]\!] \subseteq ([\![\ PO(\lambda_1)\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]}) \tag{B.3.1}$$

$$[\![\ \lambda_2\ ]\!] \subseteq ([\![\ PO(\lambda_2)\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]}) \tag{B.3.2}$$

For all $\lambda_\alpha \in Assig(\lambda_1)$, where λ_α is the assignment $\lambda_\alpha\colon [x := E]$: (B.3.3)

$$[\![\ \lambda_\alpha\ ]\!] \subseteq ([\![\ \{at(\lambda_\alpha) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \ \lambda_\alpha\colon [x := E]\ \ \{I_{PO(\lambda_2)}\}\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]})$$

For all $\lambda_\alpha \in Assig(\lambda_2)$, where λ_α is the assignment $\lambda_\alpha\colon [x := E]$: (B.3.4)

$$[\![\ \lambda_\alpha\ ]\!] \subseteq ([\![\ \{at(\lambda_\alpha) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \ \lambda_\alpha\colon [x := E]\ \ \{I_{PO(\lambda_2)}\}\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]})$$

To establish validity of the rule's conclusion, we must prove that $[\![\ \lambda\ ]\!] \subseteq [\![\ \Box A \rightarrow PO(\lambda)\ ]\!]$, where $PO(\lambda)$ is $\{pre(\lambda_1) \wedge pre(\lambda_2)\}\ \ \lambda\colon [\ PO(\lambda_1)\ //\ PO(\lambda_2)]\ \ \{post(\lambda_1) \wedge post(\lambda_2)\}$. We do this by proving $\langle \sigma, i, j\rangle \in [\![\ \lambda\ ]\!]$ implies $\langle \sigma, i, j\rangle \in [\![\ PO(\lambda)\ ]\!] \cup \overline{[\![\ \Box A\ ]\!]}$).

Let $\langle \sigma, i, j\rangle \in [\![\ \lambda\ ]\!]$ hold.

Case 1: Assume $\langle \sigma, i, j\rangle \in \overline{[\![\ \Box A\ ]\!]}$. According to definition (4.6) of a constrained proof outline, $\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow PO(\lambda)\ ]\!]$ holds.

Case 2: Assume $\langle \sigma, i, j\rangle \in [\![\ \Box A\ ]\!]$. In order to conclude $\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow PO(\lambda)\ ]\!]$ holds, we must show that $\langle \sigma, i, j\rangle \in [\![\ PO(\lambda)\ ]\!]$ holds. We consider two cases.

Case 2.1: Assume $\sigma[..i] \not\models I_{PO(\lambda)}$. According to definition (4.4) of the property defined by $PO(\lambda)$, we conclude $\langle \sigma, i, j\rangle \in [\![\ PO(\lambda)\ ]\!]$ holds.

Case 2.2: Assume $\sigma[..i] \models I_{PO(\lambda)}$. We prove

For all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda)}$

by induction. This establishes that $\langle \sigma, i, j\rangle \in [\![\ PO(\lambda)\ ]\!]$ holds, due to definition (4.4) of the property defined by $PO(\lambda)$.

For the induction hypothesis we use

$$P(h)\colon\ \ i \le h \quad \wedge \quad (h \le j \ \Rightarrow\ \text{for all } k,\ i \le k \le h\colon\ \sigma[..k] \models I_{PO(\lambda)})$$

Base Case: Prove $P(i)$. $P(i)$ holds because it is implied by the assumption of (this) Case 2.2.

Induction Case: Prove $P(h) \Rightarrow P(h+1)$. We assume that $P(h)$ holds and prove $P(h+1)$. If $j \le h$ then $P(h+1)$ is trivially valid, so $P(h) \Rightarrow P(h+1)$ is proved. We now consider the case

where $h < j$.

According to the definition (4.2) of $[\![\lambda]\!]$, we have $(\sigma[h], \sigma[h+1]) \in R_{\lambda': [x := E]}$ for some $\lambda' \in (Assig(\lambda_1) \cup Assig(\lambda_2))$. Without loss of generality, suppose $\lambda' \in Assig(\lambda_1)$ holds. Thus, we have $\langle \sigma, h, h+1 \rangle \in [\![\lambda']\!]$ and $\langle \sigma, h, h+1 \rangle \in [\![\lambda_1]\!]$.

Given assumption $\langle \sigma, i, j \rangle \in [\![\Box A]\!]$ of (this) Case 2 and (B.3.1), we conclude $\langle \sigma, h, h+1 \rangle \in [\![PO(\lambda_1)]\!]$ holds. From assumptions $P(h)$ and $h < j$ we have $\sigma[..h] \models I_{PO(\lambda)}$. Thus, $\sigma[..h] \models I_{PO(\lambda_1)}$ because $I_{PO(\lambda)}$ implies $I_{PO(\lambda_1)}$ by definition:

$$I_{PO(\lambda)}: \quad I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)} \wedge (at(\lambda) \Rightarrow (pre(\lambda_1) \wedge pre(\lambda_2))) \wedge (after(\lambda) \Rightarrow (post(\lambda_1) \wedge post(\lambda_2))) \tag{B.3.5}$$

Since $\sigma[..h] \models I_{PO(\lambda_1)}$, according to definition (4.4) of $[\![PO(\lambda_1)]\!]$ we have that for all k, $i \le k \le j$ $\sigma[..k] \models I_{PO(\lambda_1)}$ holds, because we know $\langle \sigma, i, j \rangle \in [\![PO(\lambda_1)]\!]$. And, since $i \le h < j$ we conclude

$$\sigma[..h+1] \models I_{PO(\lambda_1)}. \tag{B.3.6}$$

Given $\langle \sigma, h, h+1 \rangle \in [\![\lambda']\!]$, we conclude from (B.3.3) that:

$$\langle \sigma, h, h+1 \rangle \in ([\![\{at(\lambda') \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\} \ \lambda' \ \{I_{PO(\lambda_2)}\}]\!] \cup \overline{[\![\Box A]\!]})$$

Thus, from the assumption $\langle \sigma, i, j \rangle \in [\![\Box A]\!]$ of (this) Case 2, we obtain:

$$\langle \sigma, h, h+1 \rangle \in [\![\{at(\lambda') \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\} \ \lambda' \ \{I_{PO(\lambda_2)}\}]\!] \tag{B.3.7}$$

From assumptions $P(h)$ and $h < j$ we have $\sigma[..h] \models I_{PO(\lambda)}$ so, from (B.3.5), we conclude

$$\sigma[..h] \models I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}. \tag{B.3.8}$$

Using definition (4.4) of the property defined by a proof outline, we conclude from (B.3.7) that either $\sigma[..h] \not\models I_{PO(\lambda')}$ or else

$$\text{For all } k,\ h \le k \le h+1 \quad \sigma[..k] \models I_{PO(\lambda')} \tag{B.3.9}$$

where

$$I_{PO(\lambda')}: \quad (at(\lambda') \Rightarrow (at(\lambda') \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)})) \ \wedge \ (after(\lambda') \Rightarrow I_{PO(\lambda_2)})$$

Definition (4.1) and $(\sigma[h], \sigma[h+1]) \in R_{\lambda': [x := E]}$ implies $\sigma[h] \models at(\lambda')$. Conjoining $\sigma[h] \models at(\lambda')$ and (B.3.8) allows us to rule out $\sigma[..h] \not\models I_{PO(\lambda')}$ so (B.3.9) must hold.

From $(\sigma[h], \sigma[h+1]) \in R_{\lambda': [x := E]}$ and definition (4.1), we have $\sigma[h+1] \models after(\lambda')$. We, therefore, conclude from (B.3.9) that

$$\sigma[..h+1] \models I_{PO(\lambda_2)} \tag{B.3.10}$$

Finally, observe that, by construction, $I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}$ is equivalent to $I_{PO}(\lambda)$ defined in (B.3.5). Thus, proving (B.3.6) and (B.3.10)—as we have—suffices for proving $\sigma[..h+1] \models I_{PO(\lambda)}$, and $P(h+1)$ is proved. □

Lemma (Soundness of Cnstr-Equiv): The rule

$$\text{Cnstr-Equiv:}\quad \frac{\Box A \rightarrow PO(\lambda),\ A \Rightarrow (I_{PO(\lambda)} = I_{PO'(\lambda)}),\ PO'(\lambda) \text{ is self consistent}}{\Box A \rightarrow PO'(\lambda)}$$

is sound.

Proof. Assume that the three hypotheses are valid. We prove that the rule's conclusion is valid by showing $[\![\lambda]\!] \subseteq [\![\Box A \rightarrow PO'(\lambda)]\!]$.

Let $\langle \sigma, i, j \rangle \in [\![\lambda]\!]$. From the validity of $\Box A \rightarrow PO(\lambda)$, we have $[\![\lambda]\!] \subseteq [\![\Box A \rightarrow PO(\lambda)]\!]$. We consider two cases.

Case 1: Assume $\langle \sigma, i, j \rangle \in \overline{[\![\Box A]\!]}$. According to definition (4.6) for the property defined by a constrained proof outline, we have $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO'(\lambda)]\!]$.

Case 2: Assume $\langle \sigma, i, j \rangle \in [\![\Box A]\!]$. This means

$$\text{For all } k,\ i \le k \le j\text{: } \sigma[..k] \models A \tag{B.4.1}$$

Assuming $\langle \sigma, i, j \rangle \in [\![\Box A]\!]$ also implies $\langle \sigma, i, j \rangle \in [\![PO(\lambda)]\!]$, due to definition (4.6) of the property defined by a constrained proof outline. From definition (4.4) of $[\![PO(\lambda)]\!]$ we conclude $\sigma[..i] \not\models I_{PO(\lambda)}$ or else for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda)}$. By conjoining (B.4.1) with these, we infer $\sigma[..i] \not\models (A \wedge I_{PO(\lambda)})$ or else for all k, $i \le k \le j$: $\sigma[..k] \models (A \wedge I_{PO(\lambda)})$.

The second hypothesis of Cnstr-Equiv implies that $A \wedge I_{PO(\lambda)}$ equals $A \wedge I_{PO'(\lambda)}$. Thus, we conclude $\sigma[..i] \not\models (A \wedge I_{PO'(\lambda)})$ or else for all k, $i \le k \le j$: $\sigma[..k] \models (A \wedge I_{PO'(\lambda)})$ must hold. Given (B.4.1), if $\sigma[..i] \not\models (A \wedge I_{PO'(\lambda)})$ holds then $\sigma[..i] \not\models I_{PO'(\lambda)}$ must. And, in that case, $\langle \sigma, i, j \rangle \in [\![PO'(\lambda)]\!]$ according to definition (4.4) applied to $[\![PO'(\lambda)]\!]$ because, by hypothesis, $PO'(\lambda)$ is self consistent. In the case where for all k, $i \le k \le j$: $\sigma[..k] \models (A \wedge I_{PO'(\lambda)})$ holds, we conclude that for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO'(\lambda)}$ must hold, because $(A \wedge I_{PO'(\lambda)}) \Rightarrow I_{PO'(\lambda)}$. Again, $\langle \sigma, i, j \rangle \in [\![PO'(\lambda)]\!]$ according to definition (4.4) applied to $[\![PO'(\lambda)]\!]$ because, by hypothesis, $PO'(\lambda)$ is self consistent.

Having proved $\langle \sigma, i, j \rangle \in [\![PO'(\lambda)]\!]$, we conclude $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO'(\lambda)]\!]$ based on definition (4.6) for the property defined by a constrained proof outline. □

Lemma (Relative Completeness of Cnstr-Assig): The rule

$$\text{Cnstr-Assig:} \qquad \frac{\{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}}{\Box A \rightarrow \{p\}\ \ \lambda\colon [x := E]\ \ \{q\}}$$

is relatively complete.

Proof. Assume that conclusion $\Box A \rightarrow \{p\}\ \ \lambda\colon [x := E]\ \ \{q\}$ is valid. We show that hypothesis $\{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}$ is valid as well, by showing that if $\langle \sigma, i, j \rangle \in [\![\ \lambda\ ]\!]$ holds, then $\langle \sigma, i, j \rangle \in [\![\ \{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}\ ]\!]$ holds.

Let $\langle \sigma, i, j \rangle \in [\![\ \lambda\ ]\!]$ hold. We consider two cases.

Case 1: Assume $\langle \sigma, i, j \rangle \in [\![\ \Box A\ ]\!]$. According to the definition of $[\![\ \Box A\ ]\!]$:

$$\text{For all } k,\ \ i \le k \le j\colon\ \sigma[..k] \models A \tag{B.5.1}$$

Using this and the assumption that $\Box A \rightarrow \{p\}\ \ \lambda\colon [x := E]\ \ \{q\}$ is valid, we conclude that $\langle \sigma, i, j \rangle \in [\![\ PO(\lambda)\ ]\!]$ and due to (4.4) one of the following holds:

$$\sigma[..i] \not\models I_{PO(\lambda)} \tag{B.5.2}$$

$$\text{for all } k,\ \ i \le k \le j\colon\ \sigma[..k] \models I_{PO(\lambda)} \tag{B.5.3}$$

where $I_{PO(\lambda)}$: $(at(\lambda) \Rightarrow p)\ \ \wedge\ \ (after(\lambda) \Rightarrow q)$

It suffices to prove that if (B.5.2) holds or (B.5.3) holds, then $\langle \sigma, i, j \rangle \in [\![\ \{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}\ ]\!]$.

> **Case 1.1**: Assume (B.5.2) holds. Thus, $\sigma[..i] \not\models ((at(\lambda) \Rightarrow p) \wedge (after(\lambda) \Rightarrow q)$ holds. Given (B.5.1), this implies $\sigma[..i] \models ((at(\lambda) \wedge (\neg p \vee \neg A)) \vee (after(\lambda) \wedge (\neg q \wedge A))$ holds. This means $\sigma[..i] \not\models ((at(\lambda) \Rightarrow p \wedge A) \wedge (after(\lambda) \Rightarrow q \vee \neg A)$, holds so, by definition, $\langle \sigma, i, j \rangle \in [\![\ \{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}\ ]\!]$ holds.
>
> **Case 1.2**: Assume (B.5.3) holds. Given (B.5.1), this implies that
>
> $$\text{for all } k,\ \ i \le k \le j\colon\ \sigma[..k] \models ((at(\lambda) \Rightarrow p \wedge A) \wedge (after(\lambda) \Rightarrow q \vee \neg A))$$
>
> holds. Thus, by definition, $\langle \sigma, i, j \rangle \in [\![\ \{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}\ ]\!]$ holds.

Case 2: Assume $\langle \sigma, i, j \rangle \in \overline{[\![\ \Box A\ ]\!]}$. According to the definition of $\overline{[\![\ \Box A\ ]\!]}$:

$$\text{Exists } k,\ \ i \le k \le j\colon\ \sigma[..k] \models \neg A \tag{B.5.4}$$

We consider three cases, according to the definition of $[\![\ \lambda\ ]\!]$ given in (4.2). Since $\{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}$ is self consistent, it suffices to prove that $\langle \sigma, i, j \rangle \in [\![\ \{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}\ ]\!]$ holds.

> **Case 2.1**: Assume $j = i$ and $\sigma[..i] \models at(\lambda) \wedge \neg after(\lambda)$. By (B.5.4), we conclude $\sigma[..i] \not\models (at(\lambda) \Rightarrow p \wedge A)$, so $\sigma[..i] \not\models ((at(\lambda) \Rightarrow p \wedge A) \wedge (after(\lambda) \Rightarrow q \vee \neg A))$. Thus, by definition, $\langle \sigma, i, j \rangle \in [\![\ \{p \wedge A\}\ \ \lambda\colon [x := E]\ \ \{q \vee \neg A\}\ ]\!]$ holds.
>
> **Case 2.2**: Assume $j = i$ and $\sigma[..i] \models \neg at(\lambda) \wedge after(\lambda)$. By (B.5.4), we conclude $\sigma[..i] \models (after(\lambda) \Rightarrow q \vee \neg A)$, so $\sigma[..i] \models ((at(\lambda) \Rightarrow p \wedge A) \wedge (after(\lambda) \Rightarrow q \vee \neg A))$. Since $i = j$ holds, so does
>
> $$\text{for all } k,\ \ i \le k \le j\colon\ \sigma[..k] \models ((at(\lambda) \Rightarrow p \wedge A) \wedge (after(\lambda) \Rightarrow q \vee \neg A)).$$

Thus, by definition, $\langle \sigma, i, j \rangle \in [\![\{p \wedge A\}\ \lambda\colon [x := E]\ \{q \vee \neg A\}]\!]$ holds.

Case 2.3: Assume $j = i+1$ and $\sigma[..i] \models at(\lambda) \wedge \neg after(\lambda)$. If $\sigma[..i] \models (\neg p \vee \neg A)$ then $\sigma[..i] \not\models ((at(\lambda) \Rightarrow p \wedge A) \wedge (after(\lambda) \Rightarrow q \vee \neg A))$ and, by definition, $\langle \sigma, i, j \rangle \in [\![\{p \wedge A\}\ \lambda\colon [x := E]\ \{q \vee \neg A\}]\!]$ holds. If, on the other hand, $\sigma[..i] \models (p \wedge A)$, then we conclude from $j = i+1$ and (B.5.4) that $\sigma[..j] \models (q \vee \neg A)$ holds, so

$$\text{for all } k,\ \ i \le k \le j\colon\ \sigma[..k] \models ((at(\lambda) \Rightarrow p \wedge A) \wedge (after(\lambda) \Rightarrow q \vee \neg A))$$

also holds. Again, $\langle \sigma, i, j \rangle \in [\![\{p \wedge A\}\ \lambda\colon [x := E]\ \{q \vee \neg A\}]\!]$ holds. □

Lemma (Relative Completeness of Cnstr-SeqComp): Assume that all valid constrained proof outlines $\Box A^* \rightarrow PO^*(\lambda_1)$ and $\Box A^* \rightarrow PO^*(\lambda_2)$ are provable, and assume that all valid predicate logic formulas are provable. Then

$$\Box A \rightarrow \{p\}\ \ \lambda\colon [PO(\lambda_1); PO(\lambda_2)]\ \ \{q\} \tag{B.6.1}$$

is provable if valid.

Proof. Assume (B.6.1) is valid. Let

$$\Box A \rightarrow \{I\}\ \ \lambda\colon [PO'(\lambda_1); PO'(\lambda_2)]\ \ \{I\} \tag{B.6.2}$$

be a proof outline for $\lambda\colon [\lambda_1;\ \lambda_2]$ in which every assertion in $PO'(\lambda_1)$ and $PO'(\lambda_2)$ is

$$I\colon\ \ I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)} \wedge (at(\lambda) \Rightarrow p) \wedge (after(\lambda) \Rightarrow q).$$

Observe that (B.6.2) is valid by Cnstr-Equiv because the proof outline invariant for (B.6.1) equals I and (B.6.1) is valid.

We show below that

$$\Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\} \tag{B.6.3}$$

$$\Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\} \tag{B.6.4}$$

are both valid. This implies that (B.6.3) and (B.6.4) are provable by the assumptions of this lemma. By construction, $post(\lambda_1)$ and $pre(\lambda_2)$ in (B.6.2) are both I, so $A \wedge post(\lambda_1) \Rightarrow pre(\lambda_2)$ is valid and, by the lemma's assumption, provable. The three hypotheses needed for using Cnstr-SeqComp to deduce (B.6.2) are now discharged. We can thus use Cnstr-Equiv to deduce (B.6.1)

We prove that (B.6.3) is valid by showing that $[\![\ \lambda_1\ ]\!] \subseteq [\![\ \Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\}\ ]\!]$ holds. Let $\langle \sigma, i, j\rangle \in [\![\ \lambda_1\ ]\!]$ hold. This means that $\langle \sigma, i, j\rangle \in [\![\ \lambda\ ]\!]$ holds as well. We conclude

$$\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow \{I\}\ \ \lambda\colon [PO'(\lambda_1); PO'(\lambda_2)]\ \ \{I\}\ ]\!] \tag{B.6.5}$$

because (B.6.2) was proved valid above.

Case 1: Assume $\langle \sigma, i, j\rangle \in \overline{[\![\ \Box A\ ]\!]}$. According to definition (4.6) for the property defined by a constrained proof outline, we have $\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\}\ ]\!]$.

Case 2: Assume $\langle \sigma, i, j\rangle \notin \overline{[\![\ \Box A\ ]\!]}$. This assumption and (B.6.5) imply $\langle \sigma, i, j\rangle \in [\![\ \{I\}\ \ \lambda\colon [PO'(\lambda_1); PO'(\lambda_2)]\ \ \{I\}\ ]\!]$.

The proof outline invariant for (B.6.2) is

$$I'\colon \bigwedge_{\lambda' \in Lab(\lambda)} ((at(\lambda') \Rightarrow I) \wedge (after(\lambda') \Rightarrow I)) \tag{B.6.6}$$

where cp ranges over the control predicates of λ. From definition (4.4) of $[\![\ PO(\lambda)\ ]\!]$ we infer that $\sigma[..i] \not\models I'$ or else for all k, $i \le k \le j$: $\sigma[..k] \models I'$.

> **Case 2.1**: Assume $\sigma[..i] \not\models I'$. Thus, by definition, $\sigma[..i] \models \neg I'$ holds. According to definition (B.6.6) for I', we conclude $\sigma[..i] \models ((\bigvee_{cp} cp) \wedge \neg I)$ holds, which implies that $\sigma[..i] \not\models I$ holds. By definition (4.4) of $[\![\ PO(\lambda_1)\ ]\!]$, we conclude $\langle \sigma, i, j\rangle \in [\![\ \{I\}\ \ PO'(\lambda_1)\ \{I\}\ ]\!]$. And, according to definition (4.6) for the property defined by a constrained proof outline, we have $\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\}\ ]\!]$.

Case 2.2: Assume for all k, $i \le k \le j$: $\sigma[..k] \models I'$. Since $\langle \sigma, i, j \rangle \in [\![\lambda_1]\!]$ we have for all k, $i \le k \le j$: $\sigma[..k] \models (in(\lambda_1) \vee after(\lambda_1))$. By construction $(I' \wedge (in(\lambda_1) \vee after(\lambda_1))) \Rightarrow I$ is valid. So we infer that for all k, $i \le k \le j$: $\sigma[..k] \models I$. I implies $I_{PO(\lambda_1)}$, and this allows us to conclude for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_1)}$. Thus, by definition (4.4) of $[\![PO(\lambda_1)]\!]$, we conclude $\langle \sigma, i, j \rangle \in [\![\{I\}\ PO'(\lambda_1)\ \{I\}]\!]$. According to definition (4.6) for the property defined by a constrained proof outline, we conclude $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\}]\!]$.

A similar argument establishes that (B.6.4) is valid. □

Lemma (Relative Completeness of Cnstr-ParComp): Assume that all valid constrained proof outlines $\Box A^* \rightarrow PO^*(\lambda_1)$ and $\Box A^* \rightarrow PO^*(\lambda_2)$ are provable, and assume that all valid constrained proof outlines involving the individual assignment statements in λ_1 and λ_2 are provable. Then

$$\Box A \rightarrow \{pre(\lambda_1) \wedge pre(\lambda_2)\}\ \ \lambda\colon [PO(\lambda_1)\ //\ PO(\lambda_2)]\ \ \{post(\lambda_1) \wedge post(\lambda_2)\} \tag{B.7.1}$$

is provable if valid.

Proof. Assume (B.7.1) is valid. Let

$$\Box A \rightarrow \{I\}\ \ \lambda\colon [PO'(\lambda_1)\ //\ PO'(\lambda_2)]\ \ \{I\} \tag{B.7.2}$$

be a proof outline for $\lambda\colon [\lambda_1\ //\ \lambda_2]$ in which every assertion in $PO'(\lambda_1)$ and $PO'(\lambda_2)$ is

$$I\colon\ \ I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)} \wedge (at(\lambda) \Rightarrow (pre(\lambda_1) \wedge pre(\lambda_2))) \wedge (after(\lambda) \Rightarrow (post(\lambda_1) \wedge post(\lambda_2))).$$

Observe that (B.7.2) is valid by Cnstr-Equiv, because the proof outline invariant for (B.7.1) equals I and (B.7.1) is valid.

We show below that

$$\Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\} \tag{B.7.3}$$

$$\Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\} \tag{B.7.4}$$

are both valid. This implies that (B.7.3) and (B.7.4) are provable by the assumptions of this lemma. We also show:

For all $\lambda_\alpha \in Assig(\lambda_2)$, where λ_α is the assignment $\lambda_\alpha\colon [x := E]$: (B.7.5)

$$\Box A \rightarrow \{at(\lambda) \wedge I_{PO(\lambda_2)} \wedge I_{PO(\lambda_1)}\}\ \ \lambda_\alpha\colon [x := E]\ \ \{I_{PO(\lambda_1)}\}$$

is valid.

For all $\lambda_\alpha \in Assig(\lambda_1)$, where λ_α is the assignment $\lambda_\alpha\colon [x := E]$: (B.7.6)

$$\Box A \rightarrow \{at(\lambda) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \ \lambda_\alpha\colon [x := E]\ \ \{I_{PO(\lambda_2)}\}$$

is valid.

By the lemma's assumption, interference-freedom obligations (B.7.4) and (B.7.5) are thus provable. The three hypotheses needed for using Cnstr-ParComp to deduce (B.7.2) are now discharged. We can thus use Cnstr-Equiv to deduce (B.7.1).

We prove that (B.7.3) is valid by showing that $[\![\ \lambda_1\ ]\!] \subseteq [\![\ \Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\}\ ]\!]$ holds. Let $\langle \sigma, i, j\rangle \in [\![\ \lambda_1\ ]\!]$ hold. This means that $\langle \sigma, i, j\rangle \in [\![\ \lambda\ ]\!]$ holds as well. We conclude

$$\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow \{I\}\ \ \lambda\colon [PO'(\lambda_1)\ //\ PO'(\lambda_2)]\ \ \{I\}\ ]\!] \tag{B.7.7}$$

because (B.7.2) was proved valid above.

Case 1: Assume $\langle \sigma, i, j\rangle \in \overline{[\![\ \Box A\ ]\!]}$. According to definition (4.6) for the property defined by a constrained proof outline, we have $\langle \sigma, i, j\rangle \in [\![\ \Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\}\ ]\!]$.

Case 2: Assume $\langle \sigma, i, j\rangle \notin \overline{[\![\ \Box A\ ]\!]}$. This assumption and (B.7.7) imply $\langle \sigma, i, j\rangle \in [\![\ \{I\}\ \ \lambda\colon [PO'(\lambda_1); PO'(\lambda_2)]\ \ \{I\}\ ]\!]$.

The proof outline invariant for (B.7.2) is

$$I'\colon \quad \bigwedge_{cp} (cp \Rightarrow I), \tag{B.7.8}$$

where *cp* ranges over the control predicates of λ. From definition (4.4) of $[\![PO(\lambda)]\!]$ we infer that $\sigma[..i] \not\models I'$ or else for all k, $i \le k \le j$: $\sigma[..k] \models I'$.

> **Case 2.1**: Assume $\sigma[..i] \not\models I'$. Thus, by definition, $\sigma[..i] \models \neg I'$ holds. According to definition (B.7.8) for I', we conclude $\sigma[..i] \models ((\bigvee_{cp} cp) \wedge \neg I)$ holds, which implies that $\sigma[..i] \not\models I$ holds. By definition (4.4) of $[\![PO(\lambda_1)]\!]$, we conclude $\langle \sigma, i, j \rangle \in [\![\{I\}\ PO'(\lambda_1)\ \{I\}]\!]$. And, according to definition (4.6) for the property defined by a constrained proof outline, we have $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\}]\!]$.
>
> **Case 2.2**: Assume for all k, $i \le k \le j$: $\sigma[..k] \models I'$. Since $\langle \sigma, i, j \rangle \in [\![\lambda_1]\!]$ we have for all k, $i \le k \le j$: $\sigma[..k] \models (in(\lambda_1) \vee after(\lambda_1))$. By construction $(I' \wedge (in(\lambda_1) \vee after(\lambda_1))) \Rightarrow I$ is valid. So we infer that for all k, $i \le k \le j$: $\sigma[..k] \models I$. I implies $I_{PO(\lambda_1)}$, and this allows us to conclude for all k, $i \le k \le j$: $\sigma[..k] \models I_{PO(\lambda_1)}$. Thus, by definition (4.4) of $[\![PO(\lambda_1)]\!]$, we conclude $\langle \sigma, i, j \rangle \in [\![\{I\}\ PO'(\lambda_1)\ \{I\}]\!]$. According to definition (4.6) for the property defined by a constrained proof outline, we conclude $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow \{I\}\ PO'(\lambda_1)\ \{I\}]\!]$.

A similar argument establishes that (B.7.4) is valid.

We prove that (B.7.5) is valid by showing that

$$[\![\lambda_\alpha]\!] \subseteq [\![\Box A \rightarrow \{at(\lambda) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \lambda_\alpha\colon [x := E]\ \{I_{PO(\lambda_2)}\}]\!]$$

holds for all $\lambda_\alpha \in Assig(\lambda_1)$, where λ_α is an assignment $\lambda_\alpha\colon [x := E]$.

For arbitrary λ_α, let $\langle \sigma, i, j \rangle \in [\![\lambda_\alpha]\!]$ hold. This means that $\langle \sigma, i, j \rangle \in [\![\lambda]\!]$ holds as well. We conclude

$$\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow \{I\}\ \lambda\colon [PO'(\lambda_1)\ //\ PO'(\lambda_2)]\ \{I\}]\!] \tag{B.7.9}$$

because (B.7.2) was proved valid above. Define

$$PO^*(\lambda_\alpha)\colon \quad \{at(\lambda) \wedge I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}\}\ \lambda_\alpha\colon [x := E]\ \{I_{PO(\lambda_2)}\}$$

Case 1: Assume $\langle \sigma, i, j \rangle \in \overline{[\![\Box A]\!]}$. According to definition (4.6) for the property defined by a constrained proof outline, we have $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO^*(\lambda_\alpha)]\!]$.

Case 2: Assume $\langle \sigma, i, j \rangle \notin \overline{[\![\Box A]\!]}$. This assumption and (B.7.9) imply $\langle \sigma, i, j \rangle \in [\![\{I\}\ \lambda\colon [PO'(\lambda_1)\ //\ PO'(\lambda_2)]\ \{I\}]\!]$.

The proof outline invariant for (B.7.9) is

$$I'\colon \quad \bigwedge_{\lambda' \in Lab(\lambda)} ((at(\lambda') \Rightarrow I) \wedge (after(\lambda') \Rightarrow I)) \tag{B.7.10}$$

where *cp* ranges over the control predicates of λ. From definition (4.4) of $[\![PO(\lambda)]\!]$ we infer that $\sigma[..i] \not\models I'$ or else for all k, $i \le k \le j$: $\sigma[..k] \models I'$.

> **Case 2.1**: Assume $\sigma[..i] \not\models I'$. Thus, by definition, $\sigma[..i] \models \neg I'$ holds. According to definition (B.7.8) for I', we conclude $\sigma[..i] \models ((\bigvee_{cp} cp) \wedge \neg I)$ holds, which implies that $\sigma[..i] \not\models I_{PO(\lambda_1)} \wedge I_{PO(\lambda_2)}$ holds. By definition (4.4) of $[\![PO^*(\lambda_\alpha)]\!]$, we conclude $\langle \sigma, i, j \rangle \in [\![PO^*(\lambda_\alpha)]\!]$. And, according to definition (4.6) for the property defined by a

constrained proof outline, we have $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO^*(\lambda_\alpha)]\!]$.

Case 2.2: Assume for all k, $i \leq k \leq j$: $\sigma[..k] \models I'$. Since $\langle \sigma, i, j \rangle \in [\![\lambda_\alpha]\!]$ holds we have

for all k, $i \leq k \leq j$: $\sigma[..k] \models (at(\lambda_\alpha) \vee after(\lambda_\alpha))$.

By construction $(I' \wedge (at(\lambda_\alpha) \vee after(\lambda_\alpha))) \Rightarrow I$ is valid. So we infer that for all k, $i \leq k \leq j$: $\sigma[..k] \models I$. I implies $(at(\lambda_\alpha) \Rightarrow (I_{PO}(\lambda_1) \wedge I_{PO}(\lambda_2))) \wedge (after(\lambda_\alpha) \Rightarrow I_{PO}(\lambda_2))$ and this allows us to conclude for all k, $i \leq k \leq j$: $\sigma[..k] \models I_{PO^*(\lambda_\alpha)}$. Thus, by definition (4.4) of $[\![PO^*(\lambda_\alpha)]\!]$, we conclude $\langle \sigma, i, j \rangle \in [\![PO^*(\lambda_\alpha)]\!]$. According to definition (4.6) for the property defined by a constrained proof outline, we conclude $\langle \sigma, i, j \rangle \in [\![\Box A \rightarrow PO^*(\lambda_\alpha)]\!]$.

A similar argument establishes that (B.7.6) is valid. □

Theorem (Relative Completeness): Cnstr-Assig, Cnstr-SeqComp and Const-ParComp comprise a (relatively) complete deductive system.

Proof. The proof is by structural induction on programs.

Base Case: A program consisting of a single assignment statement. This case then follows by the Relative Completeness of Cnstr-Assig Lemma.

Induction Case: This case then follows by the Relative Completeness of Cnstr-SeqComp and Relative Completeness of Cnstr-ParComp Lemmas. □