Reasoning about Programs by Exploiting the Environment

Extended Abstract

Limor Fix and Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

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.

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

cobegin P_1: $y := 1$; $y := 2$ // P_2: $y := 3$ **coend**

Figure 1.1. A concurrent program

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.

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 a program can 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.

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.

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. A program S satisfies a property Ψ exactly when $[\![S]\!]$, the set of behaviors of S, is in $[\![\Psi]\!]$, the set of behaviors permitted by Ψ:

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

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 $\langle S, E, \Psi \rangle \in ESat$, if and only if every feasible behavior of S under E is in Ψ:

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

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

Defining a separate logic to prove $\langle S, E, \Psi \rangle \in ESat$ is not always necessary if a logic to prove $\langle S, \Psi \rangle \in Sat$ 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.

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

Thus we have:

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

Program Reduction is useful if the logic for $\langle S, \Psi \rangle \in Sat$ 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).

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

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 $\langle S, \Psi \rangle \in Sat$. 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.

A TLA *state predicate* is a predicate logic formula over some variables. 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 $\mathcal{A}$ and list $\bar{x}$ of variables $x_1, x_2, ..., x_n$, the action[1] $[\mathcal{A}]_{\bar{x}}$ is satisfied by any pair (s, t) of states such that $(s, t) \models \mathcal{A}$ or the values of the x_i are unchanged between s and t. Writing $\bar{x}'$ to denote the result of priming every variable in $\bar{x}$, we get $[\mathcal{A}]_{\bar{x}}$ is defined to be $\mathcal{A} \vee \bar{x}=\bar{x}'$.

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 $\mathcal{A}$. That state predicate, $Enbl(\mathcal{A})$, is equivalent to Exists t: $(s, t) \models \mathcal{A}$.

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 $\mathcal{A}$ 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:

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

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

$$\begin{array}{ll} \sigma \in [\![\neg\Phi]\!] & \text{iff } \sigma \notin [\![\Phi]\!] \\ \sigma \in [\![\Phi \vee \Psi]\!] & \text{iff } \sigma \in ([\![\Phi]\!] \cup [\![\Psi]\!]) \\ \sigma \in [\![\Phi \wedge \Psi]\!] & \text{iff } \sigma \in ([\![\Phi]\!] \cap [\![\Psi]\!]) \\ \sigma \in [\![\Phi \Rightarrow \Psi]\!] & \text{iff } \sigma \in [\![\neg\Phi \vee \Psi]\!] \\ \sigma \in [\![\Box\Phi]\!] & \text{iff For all } i, i \geq 0\text{: } s_i s_{i+1} \ldots \models \Phi \\ \sigma \in [\![\Diamond\Phi]\!] & \text{iff } \sigma \in [\![\neg\Box\neg\Phi]\!] \end{array}$$

A TLA formula Φ is *valid* iff for every behavior σ, $\sigma \in [\![\Phi]\!]$ holds. Validity of $\Phi \Rightarrow \Psi$ implies every behavior σ is in $[\![\Phi \Rightarrow \Psi]\!]$. From the definition 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:

$$\Phi \Rightarrow \Psi \text{ is valid} \quad \text{if and only if} \quad \langle \Phi, \Psi \rangle \in Sat \tag{3.1}$$

3.1. 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[2] deductive system for TLA (see [L91], for example) will, by definition, be complete for

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

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

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.

3.2. 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.

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:[3]

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

The restriction that processes that have processors allocated are the only ones that advance is formalized in terms of $\mathcal{A}_\pi$, the next-state relation for a process π. We assume that these next-state relations are disjoint.

$$Pgrs(\pi): \quad \mathcal{A}_\pi \Rightarrow active_\pi$$

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

$$Run(\pi): \quad active_\pi \Rightarrow |larger(\pi)| < N \quad \text{where} \quad larger(\pi): \ \{\pi' \mid rank_\pi < rank_{\pi'}\}$$

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.

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

Letting $\bar{x}$ be a list of all the variables in the system, a fixed-priority scheduler is thus characterized by

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

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.

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

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 $Enbl(\mathcal{A}_\pi)$ 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 $lock_i = p$ (p is the current lock holder) and v_q for the process q satisfying $Enbl(q) \Rightarrow (lock_i = FREE)$ (q is attempting to acquire $lock_i$). For simplicity, we assume a system having a single lock, $lock$; $\overline{x}$ is a list of all the variables in the system.

$$PrioInher(\pi): \ [(\neg Enbl(\mathcal{A}_\pi) \Rightarrow (rank_\pi = 0)) \ \wedge \ (lock = \pi \wedge Enbl(\mathcal{A}_\pi) \Rightarrow (rank_\pi = (\max p \in \Pi: \ (Enbl(p) \Rightarrow lock = FREE) \vee lock = p: \ v_p))) \ \wedge \ (lock \neq \pi \wedge Enbl(\mathcal{A}_\pi) \Rightarrow (rank_\pi = v_\pi))]_{\overline{x}}$$

A priority inheritance scheduler is thus characterized by

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

3.3. 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 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 $\mathcal{A}_\alpha$ defines atomic action α.

$$T = 0 \ \wedge \ \Box[\bigwedge_{\alpha \in ATOM(S)} (\mathcal{A}_\alpha \Rightarrow (T + e_\alpha \leq T' \leq T + e_\alpha + \delta_\alpha))]_{\overline{x}}$$

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. 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. 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 $Lab(\lambda)$ to be the set containing label λ and all labels used between the opening and closing of λ.

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 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. See [FS94] for details.

The executions of a program λ defines a set of behaviors. It will be convenient to represent a behavior using a triple $\langle \sigma, i, j \rangle$, where σ is an infinite sequence[4] 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 $R_{\lambda'\colon [x := E]}$ for the assignments λ' in λ:

[4] 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 $s_i \ldots s_j$.

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

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

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

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.

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.

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

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 4.1. Easier to read notations[5] 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. 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 terms, and the *past term* $\Theta\mathcal{T}$ for $\mathcal{T}$ any term [S94].[6] 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 $s_2(x)+s_1(y)$. See [FS94] for a formalization.

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

```
{true}
λ: [ {true}
     λ1: [ {true}
           λ11: [y := 1] {y=1 ∨ y=3};
           {y=1 ∨ y=3}
           λ12: [y := 2] {y=2 ∨ y=3}
         ] {y=2 ∨ y=3}
     //
     {true}
     λ2: [y := 3] {y=2 ∨ y=3}
   ] {y=2 ∨ y=3}
```

Figure 4.1. Example Proof Outline

[5]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 $post(\lambda)$.

[6]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.

associated with each control variable is *true* whenever that control variable is *true*:

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

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:
$pre(\lambda') \Rightarrow pre(\lambda_1)$ and $post(\lambda_1) \Rightarrow pre(\lambda_2)$

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 $PO(\lambda)$:

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

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). A sound and complete proof system for establishing that a proof outline is valid is given in [FS94]. Such logics have become commonplace since Hoare's original proposal [H69].

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 $[\![\Box A]\!]$: $\{\langle \sigma, i, j \rangle \mid \text{For all } k, i \le k \le j:\ \sigma[..k] \models A\}$ 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, $\Box A \rightarrow PO(\lambda)$ 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 $\mathcal{L}_\lambda$ be the set of constrained proof outlines and proof outlines for λ. The required closure condition is:

Lemma: For any assertion A and any $\Phi \in \mathcal{L}_\lambda$, there exists a constrained proof outline Φ' in $\mathcal{L}_\lambda$ such that $[\![\Phi']\!] = [\![\Phi]\!] \cup \overline{[\![\Box A]\!]}$.

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 $\Box A \rightarrow PO(\lambda)$.

The deductive system for ordinary Proof Outline Logic 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 [FS94].

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

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

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.

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. 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.

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.

[FS94] Fix, L. and F.B. Schneider. Reasoning about programs by exploiting the environment. Technical report TR 94-1409, Department of Computer Science, Cornell University, Ithaca, New York, Feb. 1994.

[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.