A Temporal Logic of Actions

Leslie Lamport

April 1, 1990

©Digital Equipment Corporation 1990

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the Systems Research Center of Digital Equipment Corporation in Palo Alto, California; an acknowledgment of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of fee to the Systems Research Center. All rights reserved.

Author's Abstract

In 1977, Pnueli introduced to computer scientists a temporal logic for reasoning about concurrent programs. His logic was simple and elegant, based on the single temporal modality "forever", but it was not expressive enough to completely describe programs. Since then, a plethora of more expressive logics have been proposed, all with additional temporal modalities such as "next", "until", and "since". Here, a temporal logic is introduced based only on Pnueli's original modality "forever", but with predicates (assertions about a single state) generalized to actions—assertions about pairs of states. This logic has all the expressive power needed to describe and reason about concurrent programs. Much of the temporal reasoning required with other logics is replaced by nontemporal reasoning about actions.

Perspective by Kevin D. Jones

It is generally accepted by the software engineering community that some means of formally specifying software is an important tool in increasing confidence in its correctness, as this allows the possibility of formally proving important properties of the system with respect to the semantics of the specification. Much work has been done in this area, with the usual route being, unsurprisingly, a passing of the torch from theoreticians to engineers.

The current state of the art for sequential systems illustrates this well. From the early "theoretical" work done in the late 60s by Floyd and Hoare, formal specification and verification of sequential programs has matured into tools like VDM and Z that are now being used in industry. The most successful of these are based on the concept of modelling state transformations in some suitable logic, usually first-order predicate calculus (or a variant thereof). This approach is both sufficiently expressive and of manageable complexity.

Concurrent programs have posed more of a challenge, and specifying concurrent systems is not yet practical. One approach has been to extend the state-based methods mentioned above by replacing simple predicate calculus with a temporal logic. Early attempts suffered from a lack of expressiveness. The common approach to this problem has been to increase the power of the temporal operators. Whilst this gives the required expressiveness, it is at the price of increased logical complexity. This raises the sophistication of the reasoning involved in verification and has resulted in these techniques being difficult to translate into practice.

In this work, the author has taken a different direction. Rather than extending the logical connectives, he has extended the base terms to include predicates on pairs of states (*actions*). Much of the complexity in verification now involves reasoning about actions, rather than about the temporal system. The logic has been shown to be applicable to practical problems in the verification of concurrent algorithms, and the author is continuing to work on such examples.

His approach is more in the spirit of what has been shown to be successful in the sequential world. As stated in the report, the current area of application is the verification of concurrent algorithms, rather than complete programs. This allows the possibility of machine-checked verification of important parts of any concurrent system. Eventually, one hopes that this method may be extended to permit the practical application of formal specification and verification to complete concurrent systems.

Contents

1 Introduction

Classical program verification, begun by Floyd and Hoare, employs two languages. The program is written in a programming language, and properties of the program are written in the language of formulas of some logic. Properties are derived from the program text by special proof rules, and from other properties by reasoning within the logic. Hoare's popular method for sequential programs [Hoa69] uses an Algol-like programming language and a logic of Hoare triples.

A program logic expresses both programs and properties with a single language. Program Π satisfies property P if and only if $\Pi \Rightarrow P$ is a valid formula of the logic. Pratt's dynamic logic [Pra76] is a well-known program logic.

The temporal logic of actions is a new logic for expressing concurrent programs and their properties. It is a slightly extended version of the simplest form of temporal logic. Formulas can be built from elementary formulas using only logical operators ($\neg$, $\vee$, ...) and the one temporal operator $\Box$, which means "forever" [Pnu77]. There are two types of elementary formulas: ones of the form $[\mathcal{A}]$, where $\mathcal{A}$ is an *action*, and *state predicates*. An action is a boolean-valued expression containing primed and unprimed variables, such as $x' = x + 1$, and a state predicate is an action containing only unprimed variables. For example,[1]

$$(x = 0 \wedge y = 0) \;\wedge\; \Box[x' = x + 1 \;\wedge\; y' = y + 2x + 1] \quad\Rightarrow\quad \Box(y = x^2)$$

asserts that if x and y are initially equal to zero, and each individual step of the program increments x by 1 and y by $2x + 1$, then y always equals x^2.

The logic's action-based formulas provide fairly natural representations of programs. The action $x' = x + 1 \;\wedge\; y' = y + 2x + 1$ represents a multiple assignment statement $x, y := x + 1, y + 2x + 1$. But, unlike the assignment statement, the action can be manipulated algebraically. For example, a simple calculation shows that this action equals $y' - y = x' + x \;\wedge\; y' - y = 3x - x' + 2$.

The actions used to describe a program bear a striking resemblance to statements in a Unity program [CM88]. It is easy to translate a Unity program into a formula in the temporal logic of actions. The translation in the other direction would also be easy, except that the logic permits a

[1]We take $\Rightarrow$ to have lower precedence than any other operator except $\forall$, $\exists$, and $\models$, so $A \wedge B \Rightarrow C$ means $(A \wedge B) \Rightarrow C$.

rich class of progress (fairness) conditions, many of which are not easily expressed with Unity. Giving up any pretense of having a "real" programming language allows us to merge program and logic.

Our logic may be viewed as a generalization of Hehner's [Heh84]. While Hehner's logic deals only with partial correctness, the temporal logic of actions can express invariance properties, which generalize partial correctness, and simple fairness properties that generalize termination. We also present, in Section 5, a more general logic that can express an even richer class of properties.

A program logic offers the choice of writing programs directly in the logic, or in a more conventional language that is then translated into the logic. Our choice is based on the pragmatic view that one reasons about algorithms, not programs. An algorithm, described in perhaps a few dozen lines, may be abstracted from a ten thousand line program. The algorithm is written as a program in any convenient language, using existing synchronization primitives such as semaphores or CSP communication, or new ones invented for the occasion. This program provides an informal description of the algorithm; the formal description is obtained by translating the program into the logic.

We consider the problem of verification—proving that a given algorithm has a desired property. This is a practical problem for designers of today's concurrent systems. Any verification method can, in principle, be applied in reverse to derive a program from its desired properties. However, we have had little practical experience with program derivation, and we refer the reader interested in this topic to Chandy and Misra [CM88].

The primary goal of this paper is to explain the simple temporal logic of actions. Although this logic is new, it offers no really new method of reasoning about concurrent programs. We feel that existing "assertional" proof methods are adequate: safety properties are proved by invariance arguments [LS84, OG76], and liveness properties are proved by counting-down arguments or proof lattices [OL82]. The logic just provides a convenient way of formalizing the proofs.

Since these proof methods are not new, there is no need for realistic examples. We illustrate the logic with the simple program of Figure 1. The program has integer-valued variables x and y initialized to 0, and an (integer-valued) semaphore variable *sem* initialized to 1. There are two processes, each consisting of a loop with three atomic operations (enclosed in angle brackets), where **P** and **V** are the usual semaphore operations. The atomic actions have been labeled for easy reference. Execution of the

```
var  integer  $x$, $y$          $= 0$;
     semaphore  $sem$   $= 1$;

cobegin  loop  $\alpha_1$: $\langle\, \mathbf{P}(sem)\, \rangle$;
               $\beta_1$:  $\langle\, x := x + 1\, \rangle$;
               $\gamma_1$: $\langle\, \mathbf{V}(sem)\, \rangle$
         end loop
   □
         loop  $\alpha_2$: $\langle\, \mathbf{P}(sem)\, \rangle$;
               $\beta_2$:  $\langle\, y := y + 1\, \rangle$;
               $\gamma_2$: $\langle\, \mathbf{V}(sem)\, \rangle$
         end loop
coend
```

Figure 1: An example program.

program consists of an infinite sequence of steps, where a step is performed by executing an atomic operation of either process. Fairness is discussed later.

A more general logic is introduced in Section 5 to permit proofs that one program implements another. This logic makes possible the verification of properties that can be expressed with abstract programs—as large a class of properties as one can hope to prove by assertional techniques.

2 States, Actions, and Temporal Formulas

We now present the logic. Its semantics are defined by assigning to every syntactic formula F a meaning $[\![F]\!]$, which is an object in a semantic domain. The semantic domains are defined in terms of a set $\mathbf{S}$ of *states*. It may help to think of an element of $\mathbf{S}$ as describing the complete state of a computing device that goes through a sequence of state changes when executing a program. But formally, $\mathbf{S}$ is just an abstract set.

2.1 States and State Functions

We assume a set $\mathbf{V}$ of *values*. This set includes all data elements of interest, such as integers, booleans, and strings. All operators are assumed to be defined on all values. Thus, "abc" + 7 is some value—perhaps "fred" or 42. (Since this value is left unspecified, the correctness of a program cannot depend upon what result is obtained by adding "abc" and 7.) Operators such as $\wedge$ and $<$ are assumed to be boolean-valued, so "abc" $<$ 7 equals either *true* or *false*.

State variables are primitive terms in the logic. They represent quantities that can change during execution of a program. The meaning $[\![x]\!]$ of a state variable x is a function from $\mathbf{S}$ to $\mathbf{V}$. Intuitively, $[\![x]\!](s)$ is the value of x when the computing device is in state s.

The formal description of the program in Figure 1 uses state variables x, y, and sem to represent the corresponding program variables. Two additional state variables pc_1 and pc_2 represent the control state of the two processes. For example, $pc_1 = \beta$ iff (if and only if) control in the first process is at control point β_1, which is reached after executing statement α_1. We assume that α, β, and γ are three distinct values. (Control at β_i could be denoted by $pc_i = \beta_i$, with β_1 and β_2 being distinct values, but using a single value β eliminates some subscripts.)

A *state function* is an expression made from state variables. Its meaning is a function from $\mathbf{S}$ to $\mathbf{V}$. For example, if u and v are state variables, then $2u - v$ is the state function whose meaning $[\![2u - v]\!]$ is the function defined by[2] $[\![2u - v]\!](s) \triangleq 2[\![u]\!](s) - [\![v]\!](s)$, for any state s in $\mathbf{S}$.

A *state predicate* is a boolean-valued state function. A state predicate P is *valid*, written $\models P$, iff P is true in all states. Formally, $[\![\models P]\!]$ equals $\forall s \in \mathbf{S} : [\![P]\!](s)$.

State variables represent quantities that can change during program execution. They should not be confused with ordinary logical variables, which represent constants. If x is a state variable and n a logical variable, then $x - n^2 = 2$ is a state predicate having n as a free logical variable, and $\exists n : x - n^2 = 2$ is the state predicate whose meaning is defined by

$$[\![\exists n : x - n^2 = 2]\!](s) \;\triangleq\; \exists n : [\![x]\!](s) - n^2 = 2$$

In the simple temporal logic of actions, quantification is over logical variables only.

2.2 Actions

2.2.1 Actions and their Meaning

An *action* is a boolean-valued expression containing primed and unprimed state variables. Its meaning is a boolean-valued function on $\mathbf{S} \times \mathbf{S}$, where

[2]The symbol $\triangleq$ means *equals by definition*.

unprimed state variables are applied to the first component and primed variables to the second component. For example, the meaning of the action $y' - x > 1$ is defined by $[\![y' - x > 1]\!](s,t) \stackrel{\Delta}{=} [\![y]\!](t) - [\![x]\!](s) > 1$.

We think of an action as specifying a set of allowed state transitions. Action $\mathcal{A}$ allows the transition $s \rightarrow t$ from state s to state t iff $[\![\mathcal{A}]\!](s,t)$ equals *true*. A state transition allowed by $\mathcal{A}$ is called an $\mathcal{A}$ *transition.*

A program's atomic operations are represented by actions. The operation labeled β_2 in our example program is represented by the action β_2 defined as follows.

$$\begin{array}{lllll} \beta_2 & \stackrel{\Delta}{=} & pc_1' = pc_1 & \wedge & x' = x & \wedge \\ & & pc_2 = \beta & \wedge & y' = y + 1 & \wedge \\ & & pc_2' = \gamma & \wedge & sem' = sem & \end{array}$$

Action β_2 describes the changes to the program variables x, y, and *sem* and to the control variables pc_1 and pc_2 that can be caused by executing the atomic operation labeled β_2. We can write this somewhat more compactly as

$$\begin{array}{lllll} \beta_2 & \stackrel{\Delta}{=} & pc_2 = \beta & \wedge & y' = y + 1 \quad \wedge \\ & & pc_2' = \gamma & \wedge & \textbf{unchanged } \{x,\ sem,\ pc_1\} \end{array} \tag{1}$$

where **unchanged w** denotes the conjunction of the actions $w' = w$ for each w in the set $\mathbf{w}$ of state variables.

The actions that represent the other atomic program operations are defined similarly—for example:

$$\begin{array}{lllll} \gamma_2 & \stackrel{\Delta}{=} & pc_2 = \gamma & \wedge & sem' = sem + 1 \quad \wedge \\ & & pc_2' = \alpha & \wedge & \textbf{unchanged } \{x,\ y,\ pc_1\} \end{array} \tag{2}$$

$$\begin{array}{lllll} \alpha_1 & \stackrel{\Delta}{=} & pc_1 = \alpha & \wedge & sem > 0 \quad \wedge \\ & & pc_1' = \beta & \wedge & sem' = sem - 1 \quad \wedge \\ & & \multicolumn{3}{l}{\textbf{unchanged } \{x,\ y,\ pc_2\}} \end{array} \tag{3}$$

A state predicate, such as $x = y + 1$, is also an action—one that can be written with no primed variables. In general, a state predicate P is an action that allows a state transition $s \rightarrow t$ iff P is true in state s. When viewing P as a state predicate, $[\![P]\!]$ is a boolean-valued function on states; when viewing P as an action, $[\![P]\!]$ is a boolean-valued function on pairs of states. It will be clear from context which function we mean when we write $[\![P]\!]$.

2.2.2 The *Enabled* Predicate

For any action $\mathcal{A}$, the state predicate $\mathit{Enabled}(\mathcal{A})$ is defined to be true in state s iff there exists some state t such that $s \rightarrow t$ is an $\mathcal{A}$ transition. Formally, $[\![\mathit{Enabled}(\mathcal{A})]\!](s) \;\triangleq\; \exists t : [\![\mathcal{A}]\!](s, t)$. For the actions defined by (1)–(3),

$$\begin{array}{rcl} \mathit{Enabled}(\beta_2) & = & pc_2 = \beta \\ \mathit{Enabled}(\gamma_2) & = & pc_2 = \gamma \\ \mathit{Enabled}(\alpha_1) & = & pc_1 = \alpha \;\wedge\; \mathit{sem} > 0 \end{array}$$

If $x'_1, \ldots, x'_n$ are the primed state variables that appear in action $\mathcal{A}$, then

$$\mathit{Enabled}(\mathcal{A}) \;=\; \exists x'_1 \in \mathbf{V} : \;\ldots\; \exists x'_n \in \mathbf{V} : \mathcal{A}$$

where, in the right-hand expression, the x'_i are taken to be logical variables rather than primed state variables.

2.2.3 The Logic of Actions

Boolean combinations of actions are defined in the obvious way. Thus, $s \rightarrow t$ is an $\mathcal{A} \wedge \mathcal{B}$ transition iff it is both an $\mathcal{A}$ transition and a $\mathcal{B}$ transition. The constants *true* and *false* are also actions: *true* allows any state transition, and *false* allows no state transition. We write $\models \mathcal{A}$ to denote that $\mathcal{A}$ equals *true*. In other words, $[\![\models \mathcal{A}]\!] \;\triangleq\; \forall s, t \in \mathbf{S} : \mathcal{A}(s, t)$.

If P is a state predicate, then P' denotes the action obtained by replacing every state variable x in P by its primed version x'. For example, $(x > y + 1)' \triangleq x' > y' + 1$.

For any action $\mathcal{A}$ and state predicates P and Q, we define the Hoare triple $\{P\}\mathcal{A}\{Q\}$ to be the action $P \wedge \mathcal{A} \;\Rightarrow\; Q'$. The formula $\models \{P\}\mathcal{A}\{Q\}$ asserts that for all states s and t, if P is true in state s and $\mathcal{A}$ allows the transition $s \rightarrow t$, then Q is true in state t. A Hoare triple is an action—an expression involving primed and unprimed variables—and its validity is proved by ordinary mathematical reasoning. For example, simple algebra proves $x, y \in \mathbf{Int} \Rightarrow \{x + y > 0\}\beta_2\{x + y > 1\}$, where $\mathbf{Int}$ is the set of integers and β_2 is defined by (1).

The language-independent proof rules of Hoare logic are immediate consequences of our definition. For example, Hoare's first "Rule of Consequence" [Hoa69]

$$\frac{\{P\}\mathcal{A}\{Q\}, \quad (Q \Rightarrow R)}{\{P\}\mathcal{A}\{R\}}$$

is obvious when we substitute $P \wedge \mathcal{A} \Rightarrow Q'$ for $\{P\}\mathcal{A}\{Q\}$ and $P \wedge \mathcal{A} \Rightarrow R'$ for $\{P\}\mathcal{A}\{R\}$, since $Q \Rightarrow R$ implies $Q' \Rightarrow R'$.[3]

Language-based proof rules, such as Hoare's "Rule of Iteration" for a **while** statement, are replaced by the "Decomposition Rule" [LS84]

$$\models \ \{P\}\ \mathcal{A} \vee \mathcal{B}\ \{Q\} \ = \ \{P\}\ \mathcal{A}\ \{Q\} \ \wedge \ \{P\}\ \mathcal{B}\ \{Q\} \tag{4}$$

which follows from the definition of a Hoare triple as an action.

2.3 Temporal Logic

2.3.1 Behaviors

A *behavior* is an infinite sequence of states; the set of all behaviors is denoted by $\mathbf{S}^\omega$. If σ is the behavior $s_0, s_1, \ldots$, then σ_i denotes the i^{th} state s_i. The i^{th} *step* of σ is the state transition $\sigma_{i-1} \rightarrow \sigma_i$. It is called a *stuttering step* iff states σ_{i-1} and σ_i are equal.

An execution of a program is represented by the behavior consisting of the sequence of states assumed by the computing device. A terminating execution is represented by a behavior that ends with an infinite sequence of stuttering steps. (The computing device has stopped when it no longer changes state.) Remember that $\mathbf{S}^\omega$ contains all sequences of states; most of them do not represent executions of any program.

In a linear-time temporal logic [Lam80], the meaning of a formula is a boolean-valued function on the set $\mathbf{S}^\omega$ of behaviors. The meaning of a logical combination of temporal formulas is defined in the obvious way—for example,

$$\begin{aligned} [\![\neg F]\!](\sigma) &\triangleq \neg[\![F]\!](\sigma) \\ [\![F \wedge G]\!](\sigma) &\triangleq [\![F]\!](\sigma) \wedge [\![G]\!](\sigma) \end{aligned}$$

A formula F is valid iff it is true for all behaviors:

$$[\![\models F]\!] \ \triangleq \ \forall \sigma \in \mathbf{S}^\omega : [\![F]\!](\sigma)$$

2.3.2 $\Box$ and Friends

The temporal logic operator $\Box$ (usually read "always") is defined as follows. If σ is a behavior, let σ^{+i} denote the behavior $\sigma_i, \sigma_{i+1}, \ldots$ obtained by

[3]A proof rule $\frac{A,B}{C}$ means $(\models A) \wedge (\models B) \Rightarrow (\models C)$, which is not the same as $\models (A \wedge B \Rightarrow C)$.

cutting off the first i states in the sequence σ. For any formula F and behavior σ,

$$[\![\Box F]\!](\sigma) \;\triangleq\; \forall i \geq 0 : [\![F]\!](\sigma^{+i}) \tag{5}$$

Intuitively, a temporal formula F holds at a certain time iff F holds for the infinite behavior starting at that time. The formula $\Box F$ asserts that F holds at all times—now and in the future.

The operator $\Diamond$ (read "eventually") is defined by $\Diamond F \triangleq \neg\Box\neg F$. Intuitively, $\Diamond F$ asserts that F holds now or at some time in the future.

The operators $\Box$ and $\Diamond$ can be nested and combined with logical operators to provide more complicated temporal modalities. For example, $\Box\Diamond F$ asserts that at all times, F must be true then or at some future time. In other words, $\Box\Diamond F$ asserts that F is true infinitely often. Of particularly interest is the operator $\leadsto$ (read "leads to"), where $F \leadsto G \triangleq \Box(F \Rightarrow \Diamond G)$. Intuitively, $F \leadsto G$ asserts that whenever F is true, G is true then or at some later time.

2.3.3 Elementary Temporal Formulas

Logical operators, $\Box$, and derived operators like $\Diamond$ build formulas from other formulas. To get anywhere, we must start with elementary formulas. We now define two types of elementary formula.

If P is a state predicate, the temporal formula $\textsc{Act}(P)$ is defined by

$$[\![\,\textsc{Act}(P)\,]\!](\sigma) \;\triangleq\; [\![P]\!](\sigma_0)$$

for any behavior σ. Thus, $\textsc{Act}(P)$ is true for a behavior iff the state predicate P is true in the first state of the behavior. As usual in temporal logic, we write the temporal formula $\textsc{Act}(P)$ simply as P. The same symbol P therefore denotes both a state predicate (which is also an action) and a temporal formula. It will be clear from context which is meant.

If $\mathcal{A}$ is an action, the temporal formula $[\mathcal{A}]$ is defined by

$$[\![\,[\mathcal{A}]\,]\!](\sigma) \;\triangleq\; \forall i > 0 \,:\, \sigma_0 = \sigma_1 = \ldots = \sigma_{i-1} \neq \sigma_i \;\Rightarrow\; [\![\mathcal{A}]\!](\sigma_{i-1}, \sigma_i) \tag{6}$$

for any behavior σ. Thus, $[\mathcal{A}]$ is true for a behavior σ iff either σ consists only of stuttering steps, or else the first nonstuttering step of σ is an $\mathcal{A}$ transition. Observe that $[\mathit{false}]$ asserts that there are no nonstuttering steps.

The temporal formula $\langle\mathcal{A}\rangle$ is defined to equal $\neg[\neg\mathcal{A}]$, for any action $\mathcal{A}$. It follows from (6) that

$$[\![\langle\mathcal{A}\rangle]\!](\sigma) \;\triangleq\; \exists i > 0 \,:\, \sigma_0 = \sigma_1 = \ldots = \sigma_{i-1} \neq \sigma_i \wedge [\![\mathcal{A}]\!](\sigma_{i-1}, \sigma_i)$$

Thus, $\langle \mathcal{A} \rangle$ is true for a behavior σ iff there exists a nonstuttering step of σ and the first such step is an $\mathcal{A}$ transition.

2.3.4 Temporal Reasoning

The temporal reasoning with $\Box$ and its derived operators used in program verification is simple. With a little practice, it becomes quite natural. Much has been written on the subject [OL82, Pnu77], so we will take this kind of reasoning for granted. In addition to the usual rules for $\Box$, we need laws governing the operator [ ]. We make no attempt to formulate a comprehensive set of proof rules; we just present the ones that are used in practice.

Traditional invariance proofs of safety properties are based on the tautology

$$\Box P \ \wedge\ \Box[\mathcal{A}] \ \Rightarrow\ \Box[P \wedge \mathcal{A}] \tag{7}$$

and the following proof rule, where P is a predicate and $\mathcal{A}$ an action.

Invariance Rule $\dfrac{\{P\}\mathcal{A}\{P\}}{\Box[\mathcal{A}] \ \Rightarrow\ (P \Rightarrow \Box P)}$

The hypothesis asserts that any $\mathcal{A}$ transition with P true in the starting state has P true in the ending state. The conclusion asserts that if every nonstuttering step is an $\mathcal{A}$ transition, then P true initially implies that it remains true forever. Observe that the hypothesis is an action, while the conclusion is a temporal formula.

Proofs of leads-to formulas are based on two additional rules that are given in section 3.3.

3 Expressing Programs as Temporal Formulas

3.1 The Parts of a Program

A program Π is described by four things:

- *A collection of state variables.* The state variables for the program of Figure 1 are x, y, sem, pc_1, and pc_2.

- *A state predicate $Init_\Pi$ specifying the initial state.* For the program of Figure 1,

$$\begin{aligned} Init_\Pi \ \stackrel{\Delta}{=}\ \ & sem = 1 \ \wedge\ x = 0 \ \wedge\ pc_1 = \alpha \ \wedge \\ & \qquad\qquad\ \ y = 0 \ \wedge\ pc_2 = \alpha \end{aligned}$$

- *An action $\mathcal{N}_\Pi$ specifying the state transitions allowed by the program.* A state transition is allowed iff it can be produced by an atomic operation of the program. Thus, $\mathcal{N}_\Pi$ is the disjunction of the actions that represent the program's atomic operations. For the program of Figure 1,
$$\mathcal{N}_\Pi \;\triangleq\; \alpha_1 \lor \beta_1 \lor \gamma_1 \lor \alpha_2 \lor \beta_2 \lor \gamma_2$$
where α_1, β_2, and γ_2 are defined by (1)–(3), and α_2, β_1, and γ_1 are defined similarly.

- *A temporal formula L_Π specifying the program's progress condition.* The program action $\mathcal{N}_\Pi$ specifies what the program *may* do, but it doesn't require it to do anything. The progress condition L_Π describes what the program eventually *must* do. It is discussed in Section 3.2 below.

The program itself is the temporal logic formula Π defined by

$$\Pi \;\triangleq\; \mathit{Init}_\Pi \land \Box[\mathcal{N}_\Pi] \land L_\Pi \tag{8}$$

Viewed as an assertion about a behavior σ, the first conjunct of (8) states that Init_Π holds in the initial state σ_0; the second conjunct states that every step of σ is a stuttering step or a transition allowed by $\mathcal{N}_\Pi$; and the third conjunct states that σ satisfies the progress condition.

3.2 The Progress Condition

A program's progress condition is usually expressed in terms of fairness conditions for actions. The *weak* and *strong fairness* conditions $\mathrm{WF}(\mathcal{A})$ and $\mathrm{SF}(\mathcal{A})$ for an action $\mathcal{A}$ are defined by

$$\begin{aligned}\mathrm{WF}(\mathcal{A}) &\;\triangleq\; \Box \mathit{Enabled}(\mathcal{A}) \leadsto \langle\mathcal{A}\rangle \\ \mathrm{SF}(\mathcal{A}) &\;\triangleq\; \Box\Diamond \mathit{Enabled}(\mathcal{A}) \leadsto \langle\mathcal{A}\rangle\end{aligned}$$

The formula $\mathrm{WF}(\mathcal{A})$ asserts that if $\mathcal{A}$ becomes forever enabled, then a nonstuttering $\mathcal{A}$ transition must eventually occur—in other words, $\mathcal{A}$ cannot be continuously enabled without a nonstuttering $\mathcal{A}$ transition occurring. The formula $\mathrm{SF}(\mathcal{A})$ asserts that whenever $\mathcal{A}$ is enabled infinitely often, a nonstuttering $\mathcal{A}$ transition must eventually occur. Since $\Box F$ implies $\Box\Diamond F$ for any F, we see that $\mathrm{SF}(\mathcal{A})$ implies $\mathrm{WF}(\mathcal{A})$, for any action $\mathcal{A}$.

The weakest progress condition that occurs in practice asserts that the program never halts if some step is possible. This condition is expressed by the formula $\mathrm{WF}(\mathcal{N}_\Pi)$.

Progress conditions stronger than $\mathrm{WF}(\mathcal{N}_\Pi)$ are generally called fairness conditions. The customary "weak fairness" condition for the program of Figure 1 asserts that neither process can remain forever ready to take a step without ever taking one. This condition is expressed by $\mathrm{WF}(\mathcal{N}_\Pi^1) \wedge \mathrm{WF}(\mathcal{N}_\Pi^2)$, where $\mathcal{N}_\Pi^i$ is the action $\alpha_i \vee \beta_i \vee \gamma_i$ describing the state transitions allowed by process i.

Once an action in our example program is enabled, no other action in the same process can become enabled until the first action has been executed. It follows from this that $\mathrm{WF}(\mathcal{N}_\Pi^i)$ equals $\mathrm{WF}(\alpha_i) \wedge \mathrm{WF}(\beta_i) \wedge \mathrm{WF}(\gamma_i)$. More precisely, we can prove

$$\Box[\mathcal{N}_\Pi] \;\Rightarrow\; (\mathrm{WF}(\mathcal{N}_\Pi^i) \;=\; \mathrm{WF}(\alpha_i) \wedge \mathrm{WF}(\beta_i) \wedge \mathrm{WF}(\gamma_i))$$

Thus, weak fairness can be expressed as the conjunction of weak fairness conditions either on the processes or on the individual atomic operations.

In the program of Figure 1, weak fairness allows an individual process to wait forever at its **P** operation. Such starvation is forbidden by the customary "strong fairness" condition, which asserts that a process must eventually take infinitely many steps if it can do so infinitely often. This condition is expressed by $\mathrm{SF}(\mathcal{N}_\Pi^1) \wedge \mathrm{SF}(\mathcal{N}_\Pi^2)$. As with weak fairness, $\mathrm{SF}(\mathcal{N}_\Pi^i)$ equals $\mathrm{SF}(\alpha_i) \wedge \mathrm{SF}(\beta_i) \wedge \mathrm{SF}(\gamma_i)$, so strong fairness can be expressed by conditions on either the processes or their individual operations.

3.3 Reasoning with Fairness

Leads-to properties are derived from weak fairness assumptions by using the following rule, where P and Q are predicates and $\mathcal{N}$ and $\mathcal{A}$ are actions.

WF Rule

$$\frac{\begin{array}{l}\{P\}\;\mathcal{A}\;\{Q\}\\ \{P\}\;\mathcal{N} \wedge \neg\mathcal{A}\;\{P \vee Q\}\\ P \Rightarrow \mathit{Enabled}(\mathcal{A})\end{array}}{\Box[\mathcal{N}] \wedge \mathrm{WF}(\mathcal{A}) \;\Rightarrow\; (P \leadsto Q)}$$

The soundness of this rule is demonstrated by the following informal argument. The first two hypotheses imply that, starting from a state with P true, any $\mathcal{A}$ transition makes Q true, and any $\mathcal{N}$ transition either leaves P true or makes Q true. Hence, if every step is an $\mathcal{N}$ transition, then whenever

P becomes true, either it remains true forever or Q eventually becomes true. But if P remains true forever, then the third hypothesis implies that $\mathcal{A}$ is enabled forever, so $\text{WF}(\mathcal{A})$ implies that an $\mathcal{A}$ transition eventually occurs to make Q true.

Leads-to properties are derived from strong fairness assumptions by using the following rule, where P, Q, $\mathcal{N}$, and $\mathcal{A}$ are as before, and F is any temporal formula.

SF Rule
$$\frac{\begin{array}{l}\{P\}\ \mathcal{A}\ \{Q\} \\ \{P\}\ \mathcal{N} \wedge \neg\mathcal{A}\ \{P \vee Q\} \\ \Box F \wedge \Box[\mathcal{N} \wedge \neg\mathcal{A}] \Rightarrow (P \wedge \neg \mathit{Enabled}(\mathcal{A}) \leadsto \mathit{Enabled}(\mathcal{A}))\end{array}}{\Box F \wedge \Box[\mathcal{N}] \wedge \text{SF}(\mathcal{A}) \ \Rightarrow\ (P \leadsto Q)}$$

As in the WF Rule, the first two hypotheses together with $\Box[\mathcal{N}]$ imply that whenever P becomes true, either it remains true forever or eventually Q is true. The third hypothesis ensures that, if P is true forever, $\Box F$ holds, and an $\mathcal{A}$ transition never occurs, then whenever $\mathit{Enabled}(\mathcal{A})$ is false it eventually becomes true again. This implies that $\mathit{Enabled}(\mathcal{A})$ is true infinitely often, so $\text{SF}(\mathcal{A})$ implies that an $\mathcal{A}$ transition must eventually occur to make Q true, proving the soundness of the rule.

A formula of the form $\Box F$ is said to be *permanent.* Since $F \leadsto G$ equals $\Box(F \Rightarrow \Diamond G)$, the properties $\text{WF}(\mathcal{A})$ and $\text{SF}(\mathcal{A})$ are permanent, for any action $\mathcal{A}$. Since $\Box F \wedge \Box G$ equals $\Box(F \wedge G)$, the conjunction of permanent properties is permanent. In applying the SF Rule, $\Box F$ will be the conjunction of progress properties other than $\text{SF}(\mathcal{A})$ that are assumed of the program, perhaps conjoined with a formula $\Box I$, where I is an invariant.

4 Reasoning About Programs

Traditional state-based methods prove two types of properties: safety properties, which assert that something bad does not happen, and liveness properties, which assert that something good eventually does happen. (Safety and liveness are defined formally by Alpern and Schneider [AS85].) Although the proofs of both types of properties can be carried out in the same temporal logic of actions, they use different styles of reasoning.

4.1 Safety Properties

Assertional methods for proving safety properties, including Floyd's method for sequential programs, and the Ashcroft [Ash75] and Owicki-Gries [OG76] methods for concurrent programs, all have the same logical basis: the Invariance Rule is used to prove a formula $\models \Pi \Rightarrow \Box P$, for some predicate P. We give two simple examples of how such proofs are expressed in the temporal logic of actions.

The first safety property one usually proves about a program is that it is *type-correct*, meaning that the values of all variables are of the expected "type". Type-correctness for the program of Figure 1 is expressed by the formula $\Pi \Rightarrow \Box T$, where

$$\begin{array}{rcl} T & \triangleq & pc_1 \in \{\alpha, \beta, \gamma\} \quad \wedge \quad x \in \mathbf{Int} \quad \wedge \quad sem \in \mathbf{Nat} \quad \wedge \\ & & pc_2 \in \{\alpha, \beta, \gamma\} \quad \wedge \quad y \in \mathbf{Int} \end{array} \qquad (9)$$

and **Int** and **Nat** denote the sets of integers and naturals.

We now prove $\Pi \Rightarrow \Box T$. Since Π equals $Init_\Pi \wedge \Box[\mathcal{N}_\Pi] \wedge L_\Pi$, it suffices to prove $Init_\Pi \Rightarrow T$ and $T \wedge \Box[\mathcal{N}_\Pi] \Rightarrow \Box T$. (The progress condition L_Π, which describes only what must eventually happen, is not needed for proving safety properties.) By the Invariance Rule, $\{T\}\mathcal{N}_\Pi\{T\}$ implies $T \wedge \Box[\mathcal{N}_\Pi] \Rightarrow \Box T$. Hence, it suffices to prove $Init_\Pi \Rightarrow T$ and $\{T\}\mathcal{N}_\Pi\{T\}$.

The proof of $Init_\Pi \Rightarrow T$ is immediate. To prove $\{T\}\mathcal{N}_\Pi\{T\}$, the decomposition rule (4) implies that we need only verify the six triples $\{T\}\alpha_1\{T\}$, ... , $\{T\}\gamma_2\{T\}$. This is an exercise in triviality, the hardest step being the proof of

$$(sem \in \mathbf{Nat}) \wedge (sem > 0) \wedge (sem' = sem - 1) \Rightarrow (sem' \in \mathbf{Nat})$$

which is used in proving $\{T\}\alpha_1\{T\}$ and $\{T\}\alpha_2\{T\}$.

As a slightly less trivial example, we prove the obvious mutual exclusion property for this program: at most one process at a time can have control at β or γ. Formally, the property to be proved is $\Pi \Rightarrow \Box P$, where P equals $\neg(pc_1 \in \{\beta, \gamma\} \wedge pc_2 \in \{\beta, \gamma\})$. To prove it, we find an *invariant* I satisfying three properties: (i) $Init_\Pi \Rightarrow I$, (ii) $T \wedge I \Rightarrow P$, and (iii) $\{I\}T \wedge \mathcal{N}_\Pi\{I\}$. The following proof shows that these three properties imply $\Pi \Rightarrow \Box P$.

1. $\Pi \Rightarrow \Box[T \wedge N_\Pi]$
   Proof: $\Pi \Rightarrow \Box T$ (proved above), $\Pi \Rightarrow \Box[\mathcal{N}_\Pi]$ (by definition of Π), and (7).
2. $I \wedge \Box[T \wedge N_\Pi] \Rightarrow \Box I$
   Proof: Property (iii) and the Invariance Rule.
3. $\Pi \Rightarrow \Box I$
   Proof: 1, $\Pi \Rightarrow Init_\Pi$ (by definition of Π), property (i), and 2.
4. $\Pi \Rightarrow \Box P$
   Proof: $\Pi \Rightarrow \Box T$, 3, property (ii), and simple temporal reasoning.

We define the invariant I by

$$I \;\triangleq\; \mathit{sem} + \Upsilon(pc_1 \in \{\beta, \gamma\}) + \Upsilon(pc_2 \in \{\beta, \gamma\}) = 1 \qquad (10)$$

where $\Upsilon(\mathit{true}) \triangleq 1$ and $\Upsilon(\mathit{false}) \triangleq 0$. The proofs of $\mathit{Init}_\Pi \Rightarrow I$ and $T \wedge I \Rightarrow P$ are immediate. Applying (4), $\{I\}T \wedge \mathcal{N}_\Pi\{I\}$ is proved by verifying the six triples $\{I\}T \wedge \alpha_1\{I\}, \ldots, \{I\}T \wedge \gamma_2\{I\}$. For example, $\{I\}T \wedge \alpha_1\{I\}$ equals $T \wedge \alpha_1 \wedge I \Rightarrow I'$ which by the definition (10) of I equals

$$\begin{array}{l} T \;\wedge\; \alpha_1 \;\wedge\; \mathit{sem} + \Upsilon(pc_1 \in \{\beta, \gamma\}) + \Upsilon(pc_2 \in \{\beta, \gamma\}) = 1 \\ \qquad \Rightarrow \;\; \mathit{sem}' + \Upsilon(pc_1' \in \{\beta, \gamma\}) + \Upsilon(pc_2' \in \{\beta, \gamma\}) = 1 \end{array}$$

This implication follows by simple algebra from the definitions (9) of T and (3) of α_1.

4.2 Liveness Properties

We illustrate the proof of liveness properties by showing that the program of Figure 1, under the strong fairness assumption, increases x without bound. More precisely, we assume that the progress condition L_Π equals $\mathrm{SF}(\mathcal{N}_\Pi^1) \wedge \mathrm{SF}(\mathcal{N}_\Pi^2)$, and we prove

$$\Pi \;\Rightarrow\; (x = n \leadsto x = n + 1) \qquad (11)$$

which asserts that if x ever equals n, then at some later time it will equal $n + 1$. (Here, n is a logical variable.)

An informal proof of (11) involves four steps:

A. Control in process 1 is either at α_1, β_1, or γ_1.

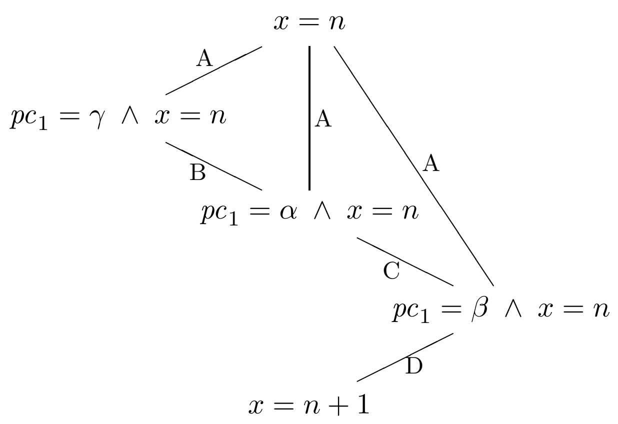
B. If control is at γ_1 with x equal to n, then eventually (after executing γ_1) control will be at α_1 with x equal to n.

C. If control is at α_1 with x equal to n, then eventually (after executing α_1) control will be at β_1 with x equal to n.

D. If control is at β_1, then eventually (after executing β_1) x will equal $n + 1$.

These steps imply that if x equals n, then it will eventually equal $n + 1$.

This line of reasoning is formalized with the proof lattice [OL82] of Figure 2. The lattice denotes the following four formulas, where letters on the lines indicate the correspondence between lattice and formula.



$x = n$
A
$pc_1 = \gamma \ \wedge \ x = n$
A
A
B
$pc_1 = \alpha \ \wedge \ x = n$
C
$pc_1 = \beta \ \wedge \ x = n$
D
$x = n + 1$

Figure 2: A proof lattice for the program of Figure 1.

A. $\Pi \ \Rightarrow \ (x = n) \leadsto$
$((pc_1 = \gamma \wedge x = n) \ \vee \ (pc_1 = \alpha \wedge x = n) \ \vee \ (pc_1 = \beta \wedge x = n))$

B. $\Pi \ \Rightarrow \ (pc_1 = \gamma \wedge x = n) \leadsto (pc_1 = \alpha \wedge x = n)$

C. $\Pi \ \Rightarrow \ (pc_1 = \alpha \wedge x = n) \leadsto (pc_1 = \beta \wedge x = n)$

D. $\Pi \ \Rightarrow \ (pc_1 = \beta \wedge x = n) \leadsto (x = n + 1)$

Property (11) is deduced from A–D by simple temporal reasoning. Detailed proofs of A–C follow. The proof of D is similar to that of B and is omitted.

Proof of A

Formula A follows from $\Pi \Rightarrow \Box T$, which was proved above, and simple temporal reasoning, since T implies $pc_1 \in \{\alpha, \beta, \gamma\}$.

Proof of B

The proof is given below in excruciating detail. Steps 1, 2, and 3 verify the three hypotheses of the WF Rule. Step 2 is actually stronger than the second hypothesis of the rule, since $\{P\}\mathcal{N} \wedge \neg\mathcal{A}\{P\}$ implies $\{P\}\mathcal{N} \wedge \neg\mathcal{A}\{P \vee Q\}$.

1. $\{pc_1 = \gamma \wedge x = n\} \, \mathcal{N}^1_\Pi \, \{pc_1 = \alpha \wedge x = n\}$
   1.1. $pc_1 = \gamma \wedge \mathcal{N}^1_\Pi \ \Rightarrow \ \gamma_1$
   Proof: $\mathcal{N}^1_\Pi = \alpha_1 \vee \beta_1 \vee \gamma_1$ (definition of $\mathcal{N}^1_\Pi$) and $pc_1 = \gamma \ \Rightarrow \ \neg(\alpha_1 \vee \beta_1)$ (definitions of α_1 and β_1).

1.2. $\gamma_1 \;\Rightarrow\; pc_1' = \alpha \wedge x' = x$
Proof: Definition of γ_1.

1.3. $\{pc_1 = \gamma \wedge x = n\}\, \mathcal{N}_\Pi^1\, \{pc_1 = \alpha \wedge x = n\}$
Proof: 1.1, 1.2, and the definition of a Hoare triple.

2. $\{pc_1 = \gamma \wedge x = n\}\, \mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1\, \{pc_1 = \gamma \wedge x = n\}$

2.1. $\mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1 = \mathcal{N}_\Pi^2$
Proof: $\mathcal{N}_\Pi = \mathcal{N}_\Pi^1 \vee \mathcal{N}_\Pi^2$ (definition of $\mathcal{N}_\Pi$).

2.2. $\mathcal{N}_\Pi^2 \;\Rightarrow\; x' = x \wedge pc_1' = pc_1$
Proof: $\mathcal{N}_\Pi^2 = \alpha_2 \vee \beta_2 \vee \gamma_2$ (definition of $\mathcal{N}_\Pi^2$) and definitions of α_2, β_2, and γ_2.

2.3. $\{pc_1 = \gamma \wedge x = n\}\, \mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1\, \{pc_1 = \gamma \wedge x = n\}$
Proof: 2.1, 2.2, and the definition of a Hoare triple.

3. $pc_1 = \gamma \wedge x = n \;\Rightarrow\; \mathit{Enabled}(\mathcal{N}_\Pi^1)$

3.1. $\mathit{Enabled}(\gamma_1) \;=\; (pc_1 = \gamma)$
Proof: Definitions of γ_1 and *Enabled*.

3.2. $\gamma_1 \Rightarrow \mathcal{N}_\Pi^1$
Proof: Definition of $\mathcal{N}_\Pi^1$.

3.3. $\mathit{Enabled}(\gamma_1) \Rightarrow \mathit{Enabled}(\mathcal{N}_\Pi^1)$
Proof: 3.2 and definition of *Enabled*.

3.4. $pc_1 = \gamma \wedge x = n \;\Rightarrow\; \mathit{Enabled}(\mathcal{N}_\Pi^1)$
Proof: 3.1 and 3.3.

4. $\Pi \;\Rightarrow\; (pc_1 = \gamma \wedge x = n) \leadsto (pc_1 = \alpha \wedge x = n)$

4.1. $\Box[\mathcal{N}_\Pi] \wedge \mathrm{WF}(\mathcal{N}_\Pi^1) \;\Rightarrow\; (pc_1 = \gamma \wedge x = n) \leadsto (pc_1 = \alpha \wedge x = n)$
Proof: 1–3 and the WF Rule.

4.2. $\Pi \;\Rightarrow\; \Box[\mathcal{N}_\Pi] \wedge \mathrm{WF}(\mathcal{N}_\Pi^1)$
Proof: Definition of Π and $\mathrm{SF}(\mathcal{N}_\Pi^1) \Rightarrow \mathrm{WF}(\mathcal{N}_\Pi^1)$.

4.3. $\Pi \;\Rightarrow\; (pc_1 = \gamma \wedge x = n) \leadsto (pc_1 = \alpha \wedge x = n)$
Proof: 4.1 and 4.2.

Proof of C

Let $P \triangleq (pc_1 = \alpha \wedge x = n)$ and $Q \triangleq (pc_1 = \beta \wedge x = n)$, so property C is $\Pi \Rightarrow (P \leadsto Q)$. This property is proved using the SF Rule and the properties $\Pi \Rightarrow \Box T$ and $\Pi \Rightarrow \Box I$, which were proved in Section 4.1. Steps 1–3 in the following proof verify the hypotheses of the SF Rule, with $\Box F$ replaced by $\Box(T \wedge I) \wedge \mathrm{WF}(\mathcal{N}_\Pi^1)$. Steps 1 and 2 are similar to the corresponding steps in the proof of property B, and the proofs are omitted.

1. $\{P\}\, \mathcal{N}_\Pi^1\, \{Q\}$

2. $\{P\}\, \mathcal{N}_P \wedge \neg\mathcal{N}_\Pi^1\, \{P\}$

3. $\Box(T \wedge I) \;\wedge\; \mathrm{WF}(\mathcal{N}_\Pi^2) \;\wedge\; \Box[\mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1] \;\;\Rightarrow$
   $(P \wedge \neg \mathit{Enabled}(\mathcal{N}_\Pi^1) \;\leadsto\; \mathit{Enabled}(\mathcal{N}_\Pi^1))$
   Proof: Let $P_\beta \triangleq (P \wedge pc_2 = \beta \wedge \mathit{sem} = 0)$ and $P_\gamma \triangleq (P \wedge pc_2 = \gamma \wedge \mathit{sem} = 0)$. Steps 3.1–3.4 represent a proof that $\Box(T \wedge I) \wedge \mathrm{WF}(\mathcal{N}_\Pi^2) \wedge \Box[\mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1]$ implies $P \wedge \mathit{sem} = 0 \leadsto P \wedge \mathit{sem} = 1$, based on the following proof lattice.

$$\begin{tikzcd}
 & P \wedge \mathit{sem} = 0 \arrow[dl, dash] \arrow[dd, dash] & \\
P_\beta \arrow[dr, dash] & & \\
 & P_\gamma \arrow[dr, dash] & \\
 & & P \wedge \mathit{sem} = 1
\end{tikzcd}$$

   Formula 3 then follows (steps 3.5–3.7) because P implies that $\mathcal{N}_\Pi^1$ is enabled iff *sem* is greater than 0.

   3.1. $\Box(T \wedge I) \;\Rightarrow\; (P \wedge \mathit{sem} = 0 \leadsto P_\beta \vee P_\gamma)$
   Proof: $T \wedge I \wedge P \wedge \mathit{sem} = 0 \;\Rightarrow\; P_\beta \vee P_\gamma$ and temporal reasoning.

   3.2. $\Box[\mathcal{N}_\Pi \;\wedge\; \neg\mathcal{N}_\Pi^1] \wedge \mathrm{WF}(\mathcal{N}_\Pi^2) \;\Rightarrow\; (P_\beta \leadsto P_\gamma)$
   Proof: By the WF Rule with $\mathcal{N}_\Pi \;\wedge\; \neg\mathcal{N}_\Pi^1$ substituted for $\mathcal{N}$ and $\mathcal{N}_\Pi^2$ substituted for $\mathcal{A}$. (The second hypothesis of the rule holds trivially because $\mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1 \wedge \neg\mathcal{N}_\Pi^2$ equals *false*.)

   3.3. $\Box[\mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1] \wedge \mathrm{WF}(\mathcal{N}_\Pi^2) \;\Rightarrow\; (P_\gamma \leadsto P \wedge \mathit{sem} = 1)$
   Proof: Similar to proof of 3.2.

   3.4. $\Box(T \wedge I) \wedge \Box[\mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1] \wedge \mathrm{WF}(\mathcal{N}_\Pi^2) \;\Rightarrow\; (P \wedge \mathit{sem} = 0 \;\leadsto\; P \wedge \mathit{sem} = 1)$
   Proof: 3.1–3.3 and temporal reasoning.

   3.5. $T \wedge I \;\Rightarrow\; (P \wedge \neg\mathit{Enabled}(\mathcal{N}_\Pi^1) \;\Rightarrow\; P \wedge \mathit{sem} = 0)$
   Proof: $P \Rightarrow (\mathcal{N}_\Pi^1 = \alpha_1)$, and $\mathit{Enabled}(\alpha_1)$ equals $pc_1 = \alpha \;\wedge\; \mathit{sem} > 0$.

   3.6. $P \wedge \mathit{sem} = 1 \;\Rightarrow\; \mathit{Enabled}(\mathcal{N}_\Pi^1)$
   Proof: Same as proof of 3.5.

   3.7. $\Box(T \wedge I) \;\wedge\; \mathrm{WF}(\mathcal{N}_\Pi^2) \;\wedge\; \Box[\mathcal{N}_\Pi \wedge \neg\mathcal{N}_\Pi^1] \;\;\Rightarrow$
   $(P \wedge \neg\mathit{Enabled}(\mathcal{N}_\Pi^1) \;\leadsto\; \mathit{Enabled}(\mathcal{N}_\Pi^1))$
   Proof: 3.4–3.6 and temporal reasoning.

4. $\Pi \;\Rightarrow\; (P \leadsto Q)$

   4.1. $\Box(T \wedge I) \;\wedge\; \mathrm{WF}(\mathcal{N}_\Pi^2) \;\wedge\; \Box[\mathcal{N}_\Pi] \;\wedge\; \mathrm{SF}(\mathcal{N}_\Pi^1) \;\Rightarrow\; (P \leadsto Q)$
   Proof: 1–3 and the SF Rule.

   4.2. $\Pi \;\Rightarrow\; \Box(T \wedge I) \;\wedge\; \mathrm{WF}(\mathcal{N}_\Pi^2) \;\wedge\; \Box[\mathcal{N}_\Pi] \;\wedge\; \mathrm{SF}(\mathcal{N}_\Pi^1)$
   Proof: $\Pi \;\Rightarrow\; \Box T$ and $\Pi \;\Rightarrow\; \Box I$ (proved above), temporal reasoning, the definition of Π, and $\mathrm{SF}(\mathcal{N}_\Pi^2) \Rightarrow \mathrm{WF}(\mathcal{N}_\Pi^2)$.

   4.3. $\Pi \;\Rightarrow\; (P \leadsto Q)$
   Proof: 4.1 and 4.2.

This completes a tiresome proof of a rather obvious property. Unfortunately, we do not know how to write shorter proofs of liveness properties without sacrificing rigor. Fortunately, with a little practice, deriving the detailed proof from a lattice like Figure 2 becomes a routine task requiring little ingenuity. People with limited time or patience will just draw the proof lattice and argue informally that it is correct. Perhaps some day these informal arguments will be mechanically translated into complete, verified proofs.

5 The General Logic

The temporal logic of actions defined above is adequate for describing and proving simple safety and liveness properties of a single program. However, the logic is deficient in three ways: there are useful properties that it cannot express, it does not support proofs that one program implements another, and it does not permit a simple compositional semantics of programs.

To illustrate the problems with the simple logic, let us try to specify how the program of Figure 1, described by the formula Π, changes the values of variables x and y. We might write such a specification as a formula Φ describing a very simple program that repeatedly increments x or y, nondeterministically but fairly choosing which to increment next. Such a formula Φ is defined as follows.

$$\begin{aligned}
\mathit{Init}_\Phi &\triangleq x = 0 \land y = 0 \\
\mathcal{N}_\Phi^1 &\triangleq x' = x + 1 \land y' = y \\
\mathcal{N}_\Phi^2 &\triangleq y' = y + 1 \land x' = x \\
\mathcal{N}_\Phi &\triangleq \mathcal{N}_\Phi^1 \lor \mathcal{N}_\Phi^2 \\
\Phi &\triangleq \mathit{Init}_\Phi \land \Box[\mathcal{N}_\Phi] \land \mathrm{WF}(\mathcal{N}_\Phi^1) \land \mathrm{WF}(\mathcal{N}_\Phi^2)
\end{aligned}$$

To show that the program satisfies property Φ, we must prove $\Pi \Rightarrow \Phi$. However, this implication is not valid. The formula $\Box[\mathcal{N}_\Phi]$ asserts that every nonstuttering step is an $\mathcal{N}_\Phi$ transition, but Π allows nonstuttering steps that do not change x or y—for example, α_1 transitions. Therefore, Π cannot imply $\Box[\mathcal{N}_\Phi]$, so it cannot imply Φ. To make $\Pi \Rightarrow \Phi$ valid, we must replace $\Box[\mathcal{N}_\Phi]$ in the definition of Φ by a formula asserting that every step *that changes x or y* is an $\mathcal{N}_\Phi$ transition—a formula written $\Box[\mathcal{N}_\Phi]_{\{x,y\}}$.

In general, let $\mathbf{w}$ be a finite set of state variables, and define a **w**-*stuttering step* to be one that does not change the value of any variable in $\mathbf{w}$. The temporal formula $[\mathcal{A}]_\mathbf{w}$ is defined to be true of a behavior σ iff the first step of σ that is not a $\mathbf{w}$-stuttering step, if such a step exists, is an $\mathcal{A}$ transition. Formally, let $s =_{\mathbf{w}} t$ denote $\forall w \in \mathbf{w} : [\![w]\!](s) = [\![w]\!](t)$ and define $[A]_{\mathbf{w}}$ by

$$[\![[\mathcal{A}]_{\mathbf{w}}]\!](\sigma) \quad = \quad \forall i > 0 : \sigma_0 =_{\mathbf{w}} \ldots =_{\mathbf{w}} \sigma_{i-1} \neq_{\mathbf{w}} \sigma_i \;\Rightarrow\; [\![\mathcal{A}]\!](\sigma_{i-1}, \sigma_i)$$

This is the same as (6), the definition of $[\mathcal{A}]$, except with $=$ replaced by $=_{\mathbf{w}}$.

The more general temporal logic of actions has, as its elementary formulas, state predicates and formulas of the form $[\mathcal{A}]_{\mathbf{w}}$. It has the same temporal operators as the simple logic, all derived from the single operator $\Box$. We define $\langle\mathcal{A}\rangle_{\mathbf{w}}$ to equal $\neg[\neg\mathcal{A}]_{\mathbf{w}}$, which asserts that there is a step changing some variable in $\mathbf{w}$, and the first such step is an $\mathcal{A}$ transition. The formulas $\mathrm{WF}_{\mathbf{w}}(\mathcal{A})$ and $\mathrm{SF}_{\mathbf{w}}(\mathcal{A})$ are defined to be the obvious analogs of $\mathrm{WF}(\mathcal{A})$ and $\mathrm{SF}(\mathcal{A})$.

The simple temporal logic of actions is derived as a special case of this new logic by fixing a finite set $\mathbf{w}$ of state variables, allowing only predicates and actions whose variables are in $\mathbf{w}$, and letting $\mathbf{w}$ be the only "subscript". The subscript $\mathbf{w}$'s are then redundant and can be dropped, yielding the simple logic.

We now redefine the formula Π describing the program of Figure 1 and the formula Φ describing how x and y are changed. In these definitions, $\mathbf{w}$ denotes the set $\{x, y, \mathit{sem}, \mathit{pc}_1, \mathit{pc}_2\}$.

$$\begin{aligned} \Pi &\triangleq \mathit{Init}_\Pi \wedge \Box[\mathcal{N}_\Pi]_{\mathbf{w}} \wedge \mathrm{SF}_{\mathbf{w}}(\mathcal{N}_\Pi^1) \wedge \mathrm{SF}_{\mathbf{w}}(\mathcal{N}_\Pi^2) \\ \Phi &\triangleq \mathit{Init}_\Phi \wedge \Box[\mathcal{N}_\Phi]_{\{x,y\}} \wedge \mathrm{WF}_{\{x,y\}}(\mathcal{N}_\Phi^1) \wedge \mathrm{WF}_{\{x,y\}}(\mathcal{N}_\Phi^2) \end{aligned}$$

With these definitions, the formula $\Pi \Rightarrow \Phi$ is valid. It can be deduced from: generalizations of (7) and of the Invariance, WF, and SF rules; some rules relating $[\mathcal{A}]_{\mathbf{u}}$ and $[\mathcal{A}]_{\mathbf{v}}$ for different $\mathbf{u}$ and $\mathbf{v}$; and simple temporal reasoning.

The general logic is completed by adding quantification over state variables, as defined in [AL88]. Intuitively, if formula Ψ represents a program with a variable w, then $\exists w : \Psi$ represents the same program except with w hidden (made internal). In our example, the formula $\exists \mathit{sem}, \mathit{pc}_1, \mathit{pc}_2 : \Pi$ describes how the program of Figure 1 changes x and y, with its effect on sem, pc_1, and pc_2 hidden. This formula is equivalent to Φ. We deduce that $\exists \mathit{sem}, \mathit{pc}_1, \mathit{pc}_2 : \Pi$ implies Φ from $\Pi \Rightarrow \Phi$ and simple properties of quantification. Proving the converse implication requires the addition of an auxiliary variable v to transform Φ into an equivalent formula $\exists v : \Phi^v$ [AL88].

The general temporal logic of actions, with quantification over state variables, can serve as the logical foundation for the transition-axiom method—a hierarchical method of specifying and verifying concurrent systems [Lam89].

6 Conclusion

Pnueli [Pnu77] introduced a temporal logic for reasoning about programs that was based on predicates, logical operators, and the single temporal operator $\Box$. It was soon recognized that Pnueli's logic is not expressive enough, and a plethora of new logics were introduced—all obtained by adding more powerful temporal operators. Instead of adding temporal operators, we have added a new kind of "action predicate".

The simple temporal logic of actions is less expressive than other extensions to Pnueli's simple logic. Its formulas can be expressed in any temporal logic that includes an *until* operator and quantification over logical variables. Our logic is just a little more expressive than Pnueli's. However, this extra expressiveness is all that we need to reason about concurrent programs. Deductions that in other logics require more powerful temporal reasoning are performed in our logic by nontemporal reasoning about actions. Although not formally complete, the Invariance, WF, and SF rules combined with (7) and simple temporal reasoning seem adequate for proving invariance and leads-to properties. Martín Abadi has developed a complete proof system for the propositional case, where predicates and actions are just uninterpreted symbols [Aba90].

The style of reasoning used in the simple logic can be extended to the general temporal logic of actions. In the absence of quantification over state variables (hiding of internal variables), such reasoning seems sufficient for proving that one program implements another. Proofs use refinement mappings, in which the implemented program's variables are expressed as functions of the implementing program's variables [AL88]. A complete proof system should exist for the propositional logic without quantification, but it has yet to be discovered.

The general logic with quantification over state variables has all the useful expressive power of any linear-time temporal logic. Other logics are more expressive only in their ability to write properties that are not invariant under stuttering [AL88]—for example, by mentioning "the next state". Such properties raise problems in proving that a lower-level program, whose next state may come after executing one machine-language instruction, implements a higher-level program, whose next state may come after executing one assignment statement. Invariance under stuttering is crucial to the hierarchical approach of the transition-axiom method [Lam89]. With quantification over state variables, additional proof rules for adding auxiliary variables are needed. The general logic may be too expressive to have a complete deduction system. However, the refinement mapping method seems to work in practice, and its semantic completeness is proved in [AL88].

Acknowledgments

Martín Abadi helped to develop the temporal logic of actions and is the source of many of the ideas presented here. Fred Schneider suggested expository improvements.

References

[Aba90] Martín Abadi. An axiomatization of Lamport's temporal logic of actions. To appear, 1990.

[AL88] Martín Abadi and Leslie Lamport. The existence of refinement mappings. Research Report 29, Digital Systems Research Center, 1988. To appear in *Theoretical Computer Science*. A preliminary version appeared in *Proceedings of the Third Annual Symposium on Logic In Computer Science*, pages 165-177, IEEE Computer Society, Edinburgh, Scotland, July 1988.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison-Wesley, Reading, Massachusetts, 1988.

[Heh84] Eric C. R. Hehner. Predicative programming. *Communications of the ACM*, 27(2):134–151, February 1984.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Lam80] Leslie Lamport. 'Sometime' is sometimes 'not never': A tutorial on the temporal logic of programs. In *Proceedings of the Seventh Annual Symposium on Principles of Programming Languages*, pages 174–185. ACM SIGACT-SIGPLAN, January 1980.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[LS84] Leslie Lamport and Fred B. Schneider. The "Hoare logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Symposium on the Foundations of Computer Science*, pages 46–57. ACM, November 1977.

[Pra76] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Symposium on Foundations of Computer Science*, pages 109–121. IEEE, October 1976.