79

The Temporal Logic of Actions

Leslie Lamport

December 25, 1991

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

The Temporal Logic of Actions

Leslie Lamport

December 25, 1991

Author's Abstract

The temporal logic of actions (TLA) is a logic for specifying and reasoning about concurrent systems. Systems and their properties are represented in the same logic, so the assertion that a system meets its specification and the assertion that one system implements another are both expressed by logical implication. TLA is very simple; its syntax and complete formal semantics are summarized in a little over a page. Yet, TLA is not just a logician's toy; it is extremely powerful, both in principle and in practice. This report introduces TLA and describes how it is used to specify and verify concurrent algorithms. The use of TLA to specify and reason about open systems will be described elsewhere.

Contents

1 Logic versus Programming

A concurrent algorithm is usually specified by a program. Correctness of the algorithm means that the program satisfies the desired property. *Program*, *property*, and *satisfies* are three separate concepts, so there are three things to define: a programming language, a language for expressing properties, and a *satisfies* relation.

We propose a simpler approach in which both the algorithm and the property are specified by formulas in a single logic. Correctness of the algorithm means that the formula specifying the algorithm implies the formula specifying the property, where *implies* is ordinary logical implication. The three concepts *program*, *property*, and *satisfies* are replaced by the single concept *logical formula*. One concept is simpler than three.

We are motivated not by an abstract ideal of elegance, but by the practical problem of reasoning about real algorithms. Rigorous reasoning is the only way to avoid subtle errors in concurrent algorithms, and such reasoning is practical only if the underlying formalism is simple.

How can we abandon conventional programming languages in favor of logic if the algorithm must be coded as a program to be executed? The answer is that we almost always reason about the abstract algorithm, not the concurrent program that is actually executed. A typical example is the distributed spanning-tree algorithm used in the Autonet local area network [22]. The algorithm can be described in about one page of pseudo-code, but its implementation required about 5000 lines of C code and 500 lines of assembly code.[1] Reasoning about 5000 lines of C would be a herculean task, but we can reason about a one-page abstract algorithm. By starting from a correct algorithm, we can avoid the timing-dependent synchronization errors that are the bane of concurrent programming. If the algorithms we reason about are not real, compilable programs, then they do not have to be written in a programming language.

But, why replace a programming language by logic? Aren't programs simpler than logical formulas? The answer is no. Programs are complicated; logical formulas are simple. For example, consider the following three-line Pascal statement and the approximately equivalent three-line logical formula, which uses the convention that x' denotes the new value of x.

[1] Assembly code was needed because C has no primitives for sending messages across wires.

if $a > b$ **then if** $a > c$ **then** $x := a$ **else** $x := c$ **else if** $b > c$ **then** $x := b$ **else** $x := c$

$$
\begin{aligned}
& x' = \max(a, \max(b, c)) \\
& \textbf{where} \;\; \max(v, w) \; = \; \text{if } v > w \text{ then } v \\
& \hspace{10em} \text{else } \; w
\end{aligned}
$$

Most readers will find the logical formula much easier to understand than the Pascal statement.

Of course, we cheated. The logical formula is easier to understand because it is formated better. But, we cheated to make a point. The reason people think programs are simpler than logical formulas is that programmers are better at formating than logicians are. When properly formated, logical formulas are no harder to read than programs.

Let us rewrite the Pascal statement and the logical formula in equally readable forms.

$$x \;:=\; \max(a, \max(b, c)) \qquad\qquad x' \;=\; \max(a, \max(b, c))$$

Many readers may think that the statement and the formula are equally simple. They are wrong. The formula is much simpler than the Pascal statement. Equality is a simple concept that five-year-old children understand. Assignment (:=) is a complicated concept that university students find difficult. Equality obeys simple algebraic laws; assignment doesn't. If we assume that all variables are integer-valued, we can subtract the left-hand side from both sides of the formula to obtain the equivalent formula

$$0 \;=\; \max(a, \max(b, c)) - x'$$

Trying this with the Pascal statement yields the absurdity

$$0 \;\overset{?}{:=}\; \max(a, \max(b, c)) - x$$

The right-hand sides of the Pascal assignment and the logical formula may look the same, but they are quite different. In the logical formula, "max" represents a mathematical function. Mathematical functions are simple; American children learn about them at the age of twelve. In the Pascal formula, "max" is a Pascal function. Pascal functions are complicated, involving concepts like call by reference, call by value, and aliasing; it is unlikely that many university students understand them well.

Since real languages like Pascal are so complicated, methods for reasoning about programs are usually based on toy languages. Although simpler than real programming languages, toy languages are more complicated than simple logic. Moreover, their resemblance to real languages can be dangerously misleading. Consider the naive programmer who wants a Pascal procedure that reverses the order of 100-element arrays, so $reverse(a, b)$ sets $a(0)$ to $b(99)$, sets $a(1)$ to $b(98)$, and so on. He writes the procedure body

$$\textbf{for } i := 0, 99 \textbf{ do } a(i) := b(99 - i)$$

and uses his toy-language system to verify that it has the desired post-condition, believing that this means his Pascal procedure is correct. He will be surprised to discover that his program doesn't work because it contains the procedure call $reverse(c, c)$.

We do not mean to belittle programming languages. They are complicated because they have a difficult job to do. Logic can be based on simple concepts like mathematical functions. Programming languages cannot, because they must allow reasonably simple compilers to translate programs into reasonably efficient code for complex computers. Real languages must embrace difficult concepts like the distinction between values and locations, which leads to call-by-reference arguments and aliasing—complications that have no counterpart in simple mathematics. Programming languages are necessary for writing real programs; but logic offers a simpler alternative for reasoning about concurrent algorithms.

To offer a practical alternative to programming languages, a logic must be both simple and expressive. There is no point trading a programming language for a logic that is just as complicated and hard to understand. Furthermore, a logic that achieves simplicity at the expense of needed expressiveness will be impractical because the formulas describing real algorithms will be too long and complicated to understand.

The logic that we propose for reasoning about concurrent algorithms is the temporal logic of actions, abbreviated as TLA. It is simple enough that its syntax and complete formal semantics can be written in less than a page. Almost all of TLA—syntax, formal semantics, all derived notation used to express algorithms, and the axioms and proof rules used to reason about algorithms—appears in Figures 4 (page 21), 5 (page 22), and 9 (page 45). (All that is missing from those figures are the rules for adding dummy variables, mentioned in Section 9.3.)

Logic is a tool. Its true test comes with use. Although TLA can be defined formally in a few pages, such a definition would tell you nothing about how it is used. In this report, we develop TLA as a method of describing and reasoning about concurrent algorithms. We limit ourselves to simple examples, so we can only hint at how TLA works with real algorithms.

2 Closed versus Open Systems

In this report, we consider closed systems. A closed system is one that is completely self-contained—in contrast to an open system, which interacts with its environment. Any real system is open; it does not eternally contemplate its navel, oblivious to the outside world. But for many purposes, one can model the actual system together with its environment as a single closed system. Such an approach is usually adequate for reasoning about algorithms. However, some problems can be studied only in the context of open systems. For example, composing component systems to form one large system makes sense only for components that are open systems.

TLA can be used to describe and reason about open as well as closed systems. But closed systems are simpler, and they provide a necessary foundation for the study of open systems. This report develops TLA and applies it to closed systems. Open systems will be discussed elsewhere.

3 The Logic of Actions

TLA is the combination of two logics: a logic of actions and a simple temporal logic. This section describes the logic of actions, Section 4 describes simple temporal logic, and Section 5 combines them into a logic called RTLA, which is refined in Section 6 to TLA.

3.1 Values, Variables, and States

Algorithms manipulate data. We assume a set Val of *values*, where a value is a data item. The set Val includes the booleans true and false, numbers such as 1, 7, and -14, strings like "abc", and sets like the set Bool of booleans and the set Nat of natural numbers. We won't bother to define Val precisely, but
Note[2] 1 will simply assume that it contains all the values needed for our examples.

We think of algorithms as assigning values to variables. We assume an infinite set **Var** of variable names. We won't bother describing a precise syntax for generating variable names, but will use names like x and sem.

A logic consists of a set of rules for manipulating formulas. To understand what the formulas and their manipulation mean, we need a semantics. A semantics is given by defining a semantic meaning $[\![F]\!]$ to syntactic objects F in the logic.

[2]End notes appear on page 61.

The semantics of our logic will be defined in terms of *states*. A state is an assignment of values to variables—that is, a mapping from the set **Var** of variable names to the set Val of values. Thus a state s assigns a value $s(x)$ to a variable x.

We will write $s[\![x]\!]$ to denote $s(x)$. Thus, we consider the meaning $[\![x]\!]$ of the variable x to be a mapping from states to values, using a postfix notation for function application. States are a purely semantic concept; they are not part of the logic.

3.2 State Functions and Predicates

A *state function* is an expression built from variables and values—for example, $x^2 + y - 3$. The meaning $[\![f]\!]$ of a state function f is a mapping from the set St of states to the set Val of values. For example, $[\![x^2 + y - 3]\!]$ is the mapping that assigns to a state s the value $(s[\![x]\!])^2 + s[\![y]\!] - 3$. We use a postfix functional notation, letting $s[\![f]\!]$ denote the value that $[\![f]\!]$ assigns to state s. The semantic definition is Note 2

$$s[\![f]\!] \quad \triangleq \quad f(\forall \text{ `}v\text{'} : s[\![v]\!]/v) \tag{1}$$

where $f(\forall \text{ `}v\text{'} : s[\![v]\!]/v)$ denotes the value obtained from f by substituting $s[\![v]\!]$ for v, for all variables v. (The symbol $\triangleq$ means *equals by definition.*)

A variable x is a state function—the state function that assigns the value $s[\![x]\!]$ to the state s. The definition of $[\![f]\!]$ for a state function f therefore extends the definition of $[\![x]\!]$ for a variable x.

A *predicate* is a boolean-valued state function—for example, $x^2 = y - 3$ and $x \in \mathsf{Nat}$. In other words, a predicate P is a state function such that $s[\![P]\!]$ equals true or false for every state s. We say that a state s *satisfies* a predicate P iff (if and only if) $s[\![P]\!]$ equals true. Note 3

State functions correspond both to expressions in ordinary programming languages and to subexpressions of the assertions used in ordinary program verification. Predicates correspond both to boolean-valued expressions in programming languages and to assertions.

3.3 Actions

An *action* is any boolean-valued expression formed from variables, primed variables, and values—for example, $x' + 1 = y$ and $x - 1 \notin z'$ are actions, where x, y, and z are variables.

An action represents a relation between old states and new states, where the unprimed variables refer to the old state and the primed variables refer to the new state. Thus, $y = x' + 1$ is the relation asserting that the value of y in the old state is one greater than the value of x in the new state. An atomic operation of a concurrent program will be represented in TLA by an action.

Formally, the meaning $[\![\mathcal{A}]\!]$ of an action $\mathcal{A}$ is a relation between states—a function that assigns a boolean $s[\![\mathcal{A}]\!]t$ to a pair of states s, t. We define $s[\![\mathcal{A}]\!]t$ by considering s to be the "old state" and t the "new state", so $s[\![\mathcal{A}]\!]t$ is obtained from $\mathcal{A}$ by replacing each unprimed variable v by $s[\![v]\!]$ and each primed variable v' by $t[\![v]\!]$:

$$s[\![\mathcal{A}]\!]t \;\triangleq\; \mathcal{A}(\forall \text{`}v\text{'} : s[\![v]\!]/v, t[\![v]\!]/v') \tag{2}$$

Thus, $s[\![y = x' + 1]\!]t$ equals the boolean value $s[\![y]\!] = t[\![x]\!] + 1$.

The pair of states s, t is called an "$\mathcal{A}$ step" iff $s[\![\mathcal{A}]\!]t$ equals true. If action $\mathcal{A}$ represents an atomic operation of a program, then s, t is an $\mathcal{A}$ step iff executing the operation in state s can produce state t.

3.4 Predicates as Actions

We have defined a predicate P to be a boolean-valued state function, so $s[\![P]\!]$ is a boolean, for any state s. The predicate P can also be viewed as an action (a boolean-valued expression involving primed and unprimed variables) that contains no primed variables. Thus, $s[\![P]\!]t$ is a boolean, which equals $s[\![P]\!]$, for any states s and t. A pair of states s, t is a P step iff s satisfies P.

Recall that a state function is an expression built from variables and values. For any state function f, we define f' to be the expression obtained by replacing each variable v in f by the primed variable v':

$$f' \;\triangleq\; f(\forall \text{`}v\text{'} : v'/v) \tag{3}$$

If P is a predicate, then P' is an action, and $s[\![P']\!]t$ equals $t[\![P]\!]$ for any states s and t.

3.5 Validity and Provability

An action $\mathcal{A}$ is said to be *valid*, written $\models \mathcal{A}$, iff every step is an $\mathcal{A}$ step. Formally,

$$\models \mathcal{A} \;\triangleq\; \forall s, t \in \mathbf{St} : s[\![\mathcal{A}]\!]t$$

As a special case of this definition, if P is a predicate, then

$$\models P \;\triangleq\; \forall s \in \mathbf{St} : s[\![P]\!]$$

A valid action is one that is true regardless of what values one substitutes for the primed and unprimed variables. For example, the action

$$(x' + y \in \mathsf{Nat}) \;\Rightarrow\; (2(x' + y) \geq x' + y) \qquad (4)$$

is valid. The validity of an action thus expresses a theorem about values.

A logic contains rules for proving formulas. It is customary to write $\vdash F$ to denote that formula F is provable by the rules of the logic. Soundness of the logic means that every provable formula is valid—in other words, that $\vdash \mathcal{A}$ implies $\models \mathcal{A}$. The validity of actions such as (4) is proved by ordinary mathematical reasoning. How this reasoning is formalized does not concern us here, so we will not bother to define a logic for proving the validity of actions. But, this omission does not mean such reasoning is unimportant. When verifying the validity of TLA formulas, most of the work goes into proving the validity of actions (and of predicates, a special class of actions). The practical success of any TLA verification system will depend primarily on how good it is at ordinary mathematical reasoning.

Note 4

3.6 Rigid Variables and Quantifiers

Consider a program that is described in terms of a parameter n—for example an n-process mutual exclusion algorithm. An action representing an atomic operation of that program may contain the symbol n. This symbol does not represent a known value like 1 or "abc". But unlike the variables we have considered so far, the value of n does not change; it must be the same in the old and new state.

The symbol n denotes some fixed but unknown value. A programmer would call it a *constant* because its value doesn't change during execution of the program, while a mathematician would call it a *variable* because it is an "unknown". We call such a symbol n a *rigid variable*. The variables introduced above will be called *program variables*, or simply *variables*.

Rigid variables like n and values like the string "abc" are called *constants*. A *constant expression* is an expression like "abc" $\neq \mathsf{n}$ that is composed of constants. Constant expressions are ordinary mathematical formulas, and can be manipulated by the ordinary rules of mathematics. In particular, we can quantify over rigid variables. For example

$$(\mathsf{n} \in \mathsf{Nat}) \wedge (\forall \mathsf{m} \in \mathsf{Nat} : \mathsf{m} \geq \mathsf{n}) \qquad (5)$$

is a constant expression containing the free rigid variable n and the bound rigid variable m. This expression is equal to the constant expression $\mathsf{n} = 0$.

We could give a formal semantics for constant expressions, defining the meaning of an expression like (5) in terms of other mathematical objects. However, the semantics of constant expressions is the basis of ordinary predicate calculus and is well known. We will therefore assume that the meaning of a constant expression is understood, and we will use constant expressions in defining the semantics of TLA.

We generalize state functions and actions to allow arbitrary constants instead of just values. For example, if x is a (program) variable and n a rigid variable, then $x' = x + \mathsf{n}$ is the action asserting that the value of x in the new state is n greater than its value in the old state. More precisely, the meaning $[\![x' = x + n]\!]$ of this action is defined by

$$s[\![x' = x + \mathsf{n}]\!]t \;\stackrel{\Delta}{=}\; t[\![x]\!] = s[\![x]\!] + \mathsf{n}$$

Thus, the semantics of state functions and actions are now given in terms of constant expressions rather than values. However, a state is still an assignment of values to variables.

If $\mathcal{A}$ is an action and n a rigid variable, then $\exists \mathsf{n} \in \mathsf{Nat} : \mathcal{A}$ is also an action—a boolean-valued expression composed of constants, variables, and primed variables. The definition of $[\![\mathcal{A}]\!]$ in Section 3.3 implies

$$[\![\exists \mathsf{n} \in \mathsf{Nat} : \mathcal{A}]\!] \;\equiv\; \exists \mathsf{n} \in \mathsf{Nat} : [\![\mathcal{A}]\!]$$

A constant expression exp is *valid*, denoted $\models \mathsf{exp}$, iff it equals true when any values are substituted for its rigid variables—for example, $\models (\mathsf{n} \in \mathsf{Nat}) \Rightarrow (\mathsf{n}+1 > \mathsf{n})$. Since $s[\![\mathcal{A}]\!]t$ is now a constant expression, not a boolean, we must generalize our definition of validity to

$$\models \mathcal{A} \;\stackrel{\Delta}{=}\; \forall s, t \in \mathbf{St} : \models s[\![\mathcal{A}]\!]t$$

for any action $\mathcal{A}$.

3.7 The *Enabled* Predicate

For any action $\mathcal{A}$, we define *Enabled* $\mathcal{A}$ to be the predicate that is true for a state iff it is possible to take an $\mathcal{A}$ step starting in that state. Semantically, *Enabled* $\mathcal{A}$ is defined by

$$s[\![\mathit{Enabled}\ \mathcal{A}]\!] \stackrel{\Delta}{=} \exists t \in \mathbf{St} : s[\![\mathcal{A}]\!]t \tag{6}$$

for any state s. The predicate *Enabled* $\mathcal{A}$ can be defined syntactically as follows. If $v_1, \ldots, v_n$ are all the (program) variables that occur in $\mathcal{A}$, then

$$\mathit{Enabled}\ \mathcal{A} \ \triangleq\ \exists \mathsf{c}_1, \ldots, \mathsf{c}_n : \mathcal{A}(\mathsf{c}_1/v_1', \ldots, \mathsf{c}_n/v_n')$$

where $\mathcal{A}(\mathsf{c}_1/v_1', \ldots, \mathsf{c}_n/v_n')$ denotes the formula obtained by substituting the rigid variables c_i for all occurrences of the v_i' in $\mathcal{A}$. For example,

$$\mathit{Enabled}\ (y = (x')^2 + \mathsf{n}) \quad \equiv \quad \exists \mathsf{c} : y = \mathsf{c}^2 + \mathsf{n}$$

If action $\mathcal{A}$ represents an atomic operation of a program, then *Enabled* $\mathcal{A}$ is true for those states in which it is possible to perform the operation.

4 Simple Temporal Logic

An execution of an algorithm is often thought of as a sequence of steps, each producing a new state by changing the values of one or more variables. We will consider an execution to be the resulting sequence of states, and will take the semantic meaning of an algorithm to be the set of all its possible executions. Reasoning about algorithms will therefore require reasoning about sequences of states. Such reasoning is the province of temporal logic.

4.1 Temporal Formulas

A temporal formula is built from elementary formulas using boolean operators and the unary operator $\Box$ (read *always*). For example, if E_1 and E_2 are elementary formulas, then $\neg E_1 \wedge \Box(\neg E_2)$ and $\Box(E_1 \Rightarrow \Box(E_1 \vee E_2))$ are temporal formulas. We define simple temporal logic for an arbitrary class of elementary formulas. TLA will be defined later as a special case of simple temporal logic by specifying its elementary formulas.

The semantics of temporal logic is based on *behaviors*, where a behavior is an infinite sequence of states. Think of a behavior as the sequence of states that a computing device might go through when executing an algorithm. (It might seem that a terminating execution would be represented by a finite sequence of states, but we will see in Section 6.5 why infinite sequences are enough.)

We will define the meaning of a temporal formula in terms of the meanings of the elementary formulas it contains. Since an arbitrary temporal formula is built up from elementary formulas using boolean operators and the $\Box$ operator, and all the boolean operators can be defined in terms of $\wedge$

and $\neg$, it suffices to define $[\![F \wedge G]\!]$, $[\![\neg F]\!]$, and $[\![\Box F]\!]$ in terms of $[\![F]\!]$ and $[\![G]\!]$.

We interpret a temporal formula as an assertion about behaviors. Formally, the meaning $[\![F]\!]$ of a formula F is a boolean-valued function on behaviors. We let $\sigma[\![F]\!]$ denote the boolean value that formula F assigns to behavior σ, and we say that σ *satisfies* F iff $\sigma[\![F]\!]$ equals true.

The definitions of $[\![F \wedge G]\!]$ and $[\![\neg F]\!]$ are simple:

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

In other words, a behavior satisfies $F \wedge G$ iff it satisfies both F and G; and a behavior satisfies $\neg F$ iff it does not satisfy F. One can derive similar formulas for the other boolean operators. For example, since $F \Rightarrow G$ equals $\neg(F \wedge \neg G)$, a straightforward calculation proves that $\sigma[\![F \Rightarrow G]\!]$ equals $\sigma[\![F]\!] \Rightarrow \sigma[\![G]\!]$.

We now define $[\![\Box F]\!]$ in terms of $[\![F]\!]$. Let $\langle\!\langle s_0, s_1, s_2, \ldots\rangle\!\rangle$ denote the behavior whose first state is s_0, second state is s_1, and so on. Then

$$\langle\!\langle s_0, s_1, s_2, \ldots\rangle\!\rangle[\![\Box F]\!] \triangleq \forall n \in \mathsf{Nat} : \langle\!\langle s_n, s_{n+1}, s_{n+2}, \ldots\rangle\!\rangle[\![F]\!] \tag{7}$$

Think of the behavior $\langle\!\langle s_0, \ldots\rangle\!\rangle$ as representing the evolution of the universe, where s_n is the state of the universe at "time" n. The formula $\langle\!\langle s_0, \ldots\rangle\!\rangle[\![F]\!]$ asserts that F is true at time 0 of this behavior, and $\langle\!\langle s_n, \ldots\rangle\!\rangle[\![F]\!]$ asserts that it is true at time n. Thus, $\langle\!\langle s_0, \ldots\rangle\!\rangle[\![\Box F]\!]$ asserts that F is true at all times during the behavior $\langle\!\langle s_0, \ldots\rangle\!\rangle$. In other words, $\Box F$ asserts that F is *always* true.

4.2 Some Useful Temporal Formulas

4.2.1 Eventually

For any temporal formula F, let $\Diamond F$ be defined by

$$\Diamond F \triangleq \neg\Box\neg F \tag{8}$$

This formula asserts that it is not the case that F is always false. In other words, $\Diamond F$ asserts that F is *eventually* true. Since $\neg\forall\neg$ is the same as $\exists$,

$$\langle\!\langle s_0, s_1, s_2, \ldots\rangle\!\rangle[\![\Diamond F]\!] \equiv \exists n \in \mathsf{Nat} : \langle\!\langle s_n, s_{n+1}, s_{n+2}, \ldots\rangle\!\rangle[\![F]\!]$$

for any behavior $\langle\!\langle s_0, s_1, \ldots\rangle\!\rangle$. Therefore, a behavior satisfies $\Diamond F$ iff F is true at some time during the behavior.

4.2.2 Infinitely Often and Eventually Always

The formula $\Box\Diamond F$ is true for a behavior iff $\Diamond F$ is true at all times n during that behavior, and $\Diamond F$ is true at time n iff F is true at some time m greater than or equal to n. Formally,

$$\langle\langle s_0, s_1, \ldots\rangle\rangle[\![\Box\Diamond F]\!] \;\equiv\; \forall n \in \mathsf{Nat} : \exists m \in \mathsf{Nat} : \langle\langle s_{n+m}, s_{n+m+1}, \ldots\rangle\rangle[\![F]\!]$$

A formula of the form $\forall n : \exists m : g(n+m)$ asserts that $g(i)$ is true for infinitely many values of i. Thus, a behavior satisfies $\Box\Diamond F$ iff F is true at infinitely many times during the behavior. In other words, $\Box\Diamond F$ asserts that F is true *infinitely often*.

The formula $\Diamond\Box F$ asserts that eventually F is always true. Thus, a behavior satisfies $\Diamond\Box F$ iff there is some time such that F is true from that time on.

4.2.3 Leads To

For any temporal formulas F and G, we define $F \leadsto G$ to equal $\Box(F \Rightarrow \Diamond G)$. This formula asserts that any time F is true, G is true then or at some later time. The operator $\leadsto$ (read *leads to*) is transitive, meaning that any behavior satisfying $F \leadsto G$ and $G \leadsto H$ also satisfies $F \leadsto H$. We suggest that the reader convince himself both that $\leadsto$ is transitive, and that it would not be had $F \leadsto G$ been defined to equal $F \Rightarrow \Diamond G$.

4.3 Validity and Provability

A temporal formula F is said to be *valid*, written $\models F$, iff it is satisfied by all possible behaviors. More precisely,

$$\models F \;\stackrel{\Delta}{=}\; \forall \sigma \in \mathbf{St}^\infty : \sigma[\![F]\!] \tag{9}$$

where $\mathbf{St}^\infty$ denotes the set of all behaviors (infinite sequences of elements of $\mathbf{St}$).

We will represent both algorithms and properties as temporal formulas. An algorithm is represented by a temporal formula F such that $\sigma[\![F]\!]$ equals true iff σ represents a possible execution of the algorithm. If G is a temporal formula, then $F \Rightarrow G$ is valid iff $\sigma[\![F \Rightarrow G]\!]$ equals true for every behavior σ. Since $\sigma[\![F \Rightarrow G]\!]$ equals $\sigma[\![F]\!] \Rightarrow \sigma[\![G]\!]$, validity of $F \Rightarrow G$ means that every behavior representing a possible execution of the algorithm satisfies G. In other words, $\models F \Rightarrow G$ asserts that the algorithm represented by F satisfies property G.

In Section 6.6, we give rules for proving temporal formulas. As usual, soundness of the rules means that every provable formula is valid—that is, $\vdash F$ implies $\models F$ for any temporal formula F.

5 The Raw Logic

5.1 Actions as Temporal Formulas

The *Raw Temporal Logic of Actions*, or RTLA, is obtained by letting the elementary temporal formulas be actions. To define the semantics of RTLA formulas, we must define what it means for an action to be true on a behavior.

In Section 3.3, we defined the meaning $[\![\mathcal{A}]\!]$ of an action $\mathcal{A}$ to be a boolean-valued function that assigns the value $s[\![\mathcal{A}]\!]t$ to the pair of states s, t. We defined s, t to be an $\mathcal{A}$ step iff $s[\![\mathcal{A}]\!]t$ equals **true**. We now define $[\![\mathcal{A}]\!]$ to be true for a behavior iff the first pair of states in the behavior is an

Note 5 $\mathcal{A}$ step. Formally,

$$\langle\!\langle s_0, s_1, s_2, \ldots \rangle\!\rangle[\![\mathcal{A}]\!] \;\triangleq\; s_0[\![\mathcal{A}]\!]s_1 \tag{10}$$

RTLA formulas are built up from actions using logical operators and the temporal operator $\Box$. Thus, if $\mathcal{A}$ is an action, then $\Box\mathcal{A}$ is an RTLA formula. Its meaning is computed as follows.

$$\begin{aligned}
&\langle\!\langle s_0, s_1, s_2, \ldots \rangle\!\rangle[\![\Box\mathcal{A}]\!] \\
&\quad \equiv \forall n \in \mathsf{Nat} : \langle\!\langle s_n, s_{n+1}, s_{n+2}, \ldots \rangle\!\rangle[\![\mathcal{A}]\!] && \{\text{by } (7)\} \\
&\quad \equiv \forall n \in \mathsf{Nat} : s_n[\![\mathcal{A}]\!]s_{n+1} && \{\text{by } (10)\}
\end{aligned}$$

In other words, a behavior satisfies $\Box\mathcal{A}$ iff every step of the behavior is an $\mathcal{A}$ step.

In Section 3.4, we observed that if P is a predicate, then $s[\![P]\!]t$ equals $s[\![P]\!]$. Therefore,

$$\begin{aligned}
\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle[\![P]\!] &\;\equiv\; s_0[\![P]\!] \\
\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle[\![\Box P]\!] &\;\equiv\; \forall n \in \mathsf{Nat} : s_n[\![P]\!]
\end{aligned}$$

In other words, a behavior satisfies a predicate P iff the first state of the behavior satisfies P. A behavior satisfies $\Box P$ iff all states in the behavior satisfy P.

We will see that the raw logic RTLA is too powerful; it allows one to make assertions about behaviors that should not be assertable. We will define the formulas of TLA to be a subset of RTLA formulas.

$$
\begin{array}{l}
\textbf{var natural } x, y = 0 \; ; \\
\textbf{do } \langle \; \mathsf{true} \rightarrow x := x + 1 \; \rangle \\
\quad \Box \\
\quad\quad \langle \; \mathsf{true} \rightarrow y := y + 1 \; \rangle \quad \textbf{od}
\end{array}
$$

Figure 1: Program 1—a simple program, written in a conventional language.

5.2 Describing Programs with RTLA Formulas

We have defined the syntax and semantics of RTLA formulas, but have given no idea what RTLA is good for. We illustrate how RTLA can be used by describing the simple Program 1 of Figure 1 on this page as an RTLA formula. This program is written in a conventional language, using Dijkstra's **do** construct [7], with angle brackets enclosing operations that are assumed to be atomic. An execution of this program begins with x and y both zero, and repeatedly increments either x or y (in a single operation), choosing nondeterministically between them. We will define an RTLA formula Φ that represents this program, meaning that $\sigma[\![\Phi]\!]$ will equal true iff the behavior σ represents a possible execution of Program 1.

The formula Φ is defined in Figure 2 on the next page. The predicate $Init_\Phi$ asserts the initial condition, that x and y are both zero. The semantic meaning of action $\mathcal{M}_1$ is a relation between states asserting that the value of x in the new state is one greater than its value in the old state, and the value of y is the same in the old and new states. Thus, an $\mathcal{M}_1$ step represents an execution of the program's atomic operation of incrementing x. Similarly, an $\mathcal{M}_2$ step represents an execution of the program's other atomic operation, which increments y. The action $\mathcal{M}$ is defined to be the disjunction of $\mathcal{M}_1$ and $\mathcal{M}_2$, so an $\mathcal{M}$ step represents an execution of one program operation. The formula Φ is true of a behavior iff $Init_\Phi$ is true of the first state and every step is an $\mathcal{M}$ step. In other words, Φ asserts that the initial condition is true initially, and that every step of the behavior represents the execution of an atomic operation of the program. Clearly, a behavior satisfies Φ iff it represents a possible execution of Program 1. Note 6

There is nothing special about our choice of names, or in the particular way of writing Φ. There are many ways of writing equivalent logical formulas. Here are a couple of formulas that are equivalent to Φ.

$$
\begin{array}{l}
(x = 0) \;\wedge\; \Box(\mathcal{M}_1 \vee \mathcal{M}_2) \;\wedge\; (y = 0) \\
Init_\Phi \;\wedge\; \Box((x' = x + 1) \vee (y' = y + 1)) \;\wedge\; \Box((x' = x) \vee (y' = y))
\end{array}
$$

$$Init_\Phi \triangleq (x = 0) \land (y = 0)$$
$$\mathcal{M}_1 \triangleq (x' = x + 1) \land (y' = y) \qquad \mathcal{M}_2 \triangleq (y' = y + 1) \land (x' = x)$$
$$\mathcal{M} \triangleq \mathcal{M}_1 \lor \mathcal{M}_2$$
$$\Phi \triangleq Init_\Phi \land \Box\mathcal{M}$$

Figure 2: An RTLA formula Φ describing Program 1.

The particular way of defining Φ in Figure 2 was chosen to make the correspondence with Figure 1 obvious.

6 TLA

6.1 The Example Program Revisited

Formula Φ of Figure 2 is very simple. Unfortunately, it is too simple. In addition to steps in which x or y is incremented, a formula describing Program 1 should allow "stuttering" steps that leave both x and y unchanged. Allowing these stuttering steps may seem strange now, but later it will become clear why we need them.

It is easy to modify Φ so it asserts that every step is either an $\mathcal{M}$ step or a step that leaves x and y unchanged; the new definition is

$$\Phi \triangleq Init_\Phi \land \Box(\mathcal{M} \lor ((x' = x) \land (y' = y))) \tag{11}$$

We now introduce notation to make such formulas easier to write. Two ordered pairs are equal iff their components are equal, so the conjunction $(x' = x) \land (y' = y)$ is equivalent to the single equality $(x', y') = (x, y)$. The definition of priming a state function (formula (3) on page 6) allows us to
Note 7 write (x', y') as $(x, y)'$. For any action $\mathcal{A}$ and state function f, we let

$$[\mathcal{A}]_f \triangleq \mathcal{A} \lor (f' = f) \tag{12}$$

(The action $[\mathcal{A}]_f$ is read *square $\mathcal{A}$ sub f*.) Then

$$\begin{aligned}[\mathcal{M}]_{(x,y)} &\equiv \mathcal{M} \lor ((x, y)' = (x, y)) \\ &\equiv \mathcal{M} \lor ((x' = x) \land (y' = y))\end{aligned}$$

and we can rewrite (11) as

$$\Phi \triangleq Init_\Phi \land \Box[\mathcal{M}]_{(x,y)} \tag{13}$$

We define TLA to be the temporal logic whose elementary formulas are predicates and formulas of the form $\Box[\mathcal{A}]_f$, where $\mathcal{A}$ is an action and f a state function. Since these formulas are RTLA formulas, we have already defined their semantic meanings.

6.2 Adding Liveness

The formula Φ defined by (13) allows behaviors that start with $Init_\Phi$ true (x and y both zero) and never change x or y. Such behaviors do not represent acceptable executions of Program 1, so we must strengthen Φ to disallow them.

Formula Φ of (13) asserts that a behavior may not start in any state other than one satisfying $Init_\Phi$ and may never take any step other than a $[\mathcal{M}]_{(x,y)}$ step. An assertion that something may never happen is called a *safety* property. An assertion that something eventually does happen is called a *liveness* property. (Safety and liveness have been defined formally by Alpern and Schneider [3].) The formula $Init_\Phi \wedge \Box[\mathcal{M}]_{(x,y)}$ is a safety property. To complete the description of Program 1, we need an additional liveness property asserting that the program keeps going.

By Dijkstra's semantics for his **do** construct, the liveness property for Program 1 should assert only that the program never terminates. In other words, Dijkstra would require that a behavior must contain infinitely many steps that increment x or y. This property is expressed by the RTLA formula $\Box\Diamond\mathcal{M}$, which asserts that there are infinitely many $\mathcal{M}$ steps. Dijkstra would have us define Φ by

$$\Phi \triangleq Init_\Phi \wedge \Box[\mathcal{M}]_{(x,y)} \wedge \Box\Diamond\mathcal{M} \tag{14}$$

However, the example becomes more interesting if we add the fairness requirement that both x and y must be incremented infinitely often. (Dijkstra's definition would allow an execution in which one variable is incremented infinitely often while the other is incremented only a finite number of times.) Since we are not fettered by the dictates of conventional programming languages, we will adopt this stronger liveness requirement. The formula Φ representing the program with this fairness requirement is

$$\Phi \triangleq Init_\Phi \wedge \Box[\mathcal{M}]_{(x,y)} \wedge \Box\Diamond\mathcal{M}_1 \wedge \Box\Diamond\mathcal{M}_2 \tag{15}$$

Formulas (14) and (15) are RTLA formulas, but not TLA formulas. An action $\mathcal{A}$ can appear in a TLA formula only in the form $\Box[\mathcal{A}]_f$, so $\Box\Diamond\mathcal{M}_1$ and $\Box\Diamond\mathcal{M}_2$ are not TLA formulas. We now rewrite them as TLA formulas.

Let $\mathcal{A}$ be any action and f any state function. Then $\neg\mathcal{A}$ is also an action, so $\neg\Box[\neg\mathcal{A}]_f$ is a TLA formula. Applying our definitions gives

$$\begin{array}{rcll}\neg\Box[\neg\mathcal{A}]_f & \equiv & \Diamond\neg[\neg\mathcal{A}]_f & \{\text{by (8), which implies } \neg\Box\ldots\equiv\Diamond\neg\ldots\} \\ & \equiv & \Diamond\neg(\neg\mathcal{A}\vee(f'=f)) & \{\text{by (12)}\} \\ & \equiv & \Diamond(\mathcal{A}\wedge(f'\neq f)) & \{\text{by simple logic}\}\end{array}$$

We define the action $\langle\mathcal{A}\rangle_f$ (read *angle* $\mathcal{A}$ *sub* f) by

$$\langle\mathcal{A}\rangle_f \triangleq \mathcal{A}\wedge(f'\neq f) \tag{16}$$

The calculation above shows that $\Diamond\langle\mathcal{A}\rangle_f$ equals $\neg\Box[\neg\mathcal{A}]_f$, so it is a TLA formula.

Since incrementing a variable changes its value, both $\mathcal{M}_1$ and $\mathcal{M}_2$ imply
Note 8 $(x,y)'\neq(x,y)$. Hence, $\mathcal{M}_1$ is equivalent to $\langle\mathcal{M}_1\rangle_{(x,y)}$, and $\mathcal{M}_2$ is equivalent to $\langle\mathcal{M}_2\rangle_{(x,y)}$. We can therefore rewrite Φ as a TLA formula as follows.

$$\Phi \triangleq Init_\Phi \wedge \Box[\mathcal{M}]_{(x,y)} \wedge \Box\Diamond\langle\mathcal{M}_1\rangle_{(x,y)} \wedge \Box\Diamond\langle\mathcal{M}_2\rangle_{(x,y)} \tag{17}$$

6.3 Fairness

The liveness condition $\Box\Diamond\langle\mathcal{M}_1\rangle_{(x,y)}\wedge\Box\Diamond\langle\mathcal{M}_2\rangle_{(x,y)}$ for Program 1 is very simple. In realistic examples, liveness conditions are not always so simple. We now rewrite this condition in a more general form that can be used to express most of the liveness conditions that arise in practice.

Liveness conditions of concurrent algorithms are expressed by fairness properties. Fairness means that if a certain operation is possible, then the program must eventually execute it. To our knowledge, all fairness conditions that have been proposed fall into one of two classes: *weak fairness* and *strong fairness*. We first define weak and strong fairness informally, then translate the informal definitions into TLA formulas.

Weak fairness asserts that an operation must be executed if it remains possible to do so for a long enough time. "Long enough" means until the operation is executed, so weak fairness asserts that eventually the operation must either be executed or become impossible to execute—perhaps only briefly. A naive temporal-logic translation is

weak fairness: $(\Diamond\ \text{executed})\vee(\Diamond\ \text{impossible})$

Strong fairness asserts that the operation must be executed if it is often enough possible to do so. Interpreting "often enough" to mean infinitely often, strong fairness asserts that either the operation is eventually executed, or its execution is not infinitely often possible. Not infinitely often possible is the same as eventually always impossible (because (8) implies $\neg\Box\Diamond \ldots \equiv \Diamond\Box\neg \ldots$), so we get

$$\textit{strong fairness}: \quad (\Diamond \text{ executed}) \lor (\Diamond\Box \text{ impossible})$$

These two temporal formulas assert fairness at "time zero", but we want fairness to hold at all times. The correct formulas are therefore

$$\begin{array}{ll} \textit{weak fairness}: & \Box((\Diamond \text{ executed}) \lor (\Diamond \text{ impossible})) \\ \textit{strong fairness}: & \Box((\Diamond \text{ executed}) \lor (\Diamond\Box \text{ impossible})) \end{array}$$

Temporal-logic reasoning, using either the axioms in Section 6.6 or the semantic definitions of $\Box$ and $\Diamond$, shows that these conditions are equivalent to

$$\begin{array}{ll} \textit{weak fairness}: & (\Box\Diamond \text{ executed}) \lor (\Box\Diamond \text{ impossible}) \\ \textit{strong fairness}: & (\Box\Diamond \text{ executed}) \lor (\Diamond\Box \text{ impossible}) \end{array}$$

To formalize these definitions, we must define "executed" and "impossible".

In Program 1, execution of the operation $x := x + 1$ corresponds to an $\mathcal{M}_1$ step in the behavior. To obtain a TLA formula, the "$\Diamond$ executed" for this operation must be expressed as $\Diamond\langle\mathcal{M}_1\rangle_{(x,y)}$. In general, "$\Diamond$ executed" will be expressed as $\Diamond\langle\mathcal{A}\rangle_f$, where $\mathcal{A}$ is the action that corresponds to an execution of the operation, and f is an n-tuple of relevant variables. Recall that an $\langle\mathcal{A}\rangle_f$ step is an $\mathcal{A}$ step that changes the value of f. Steps that do not change the values of any relevant variables might as well not have occurred, so there is no need to consider them as representing operation executions.

We now define "impossible". Executing an operation means taking an $\langle\mathcal{A}\rangle_f$ step for some action $\mathcal{A}$ and state function f. It is possible to take such a step iff $\mathit{Enabled}\ \langle\mathcal{A}\rangle_f$ is true. Thus, $\mathit{Enabled}\ \langle\mathcal{A}\rangle_f$ asserts that it is possible to execute the operation represented by the action $\langle\mathcal{A}\rangle_f$, so "impossible" is $\neg\mathit{Enabled}\ \langle\mathcal{A}\rangle_f$.

Weak fairness and strong fairness are therefore expressed by the two formulas

$$\mathrm{WF}_f(\mathcal{A}) \;\triangleq\; (\Box\Diamond\langle\mathcal{A}\rangle_f) \lor (\Box\Diamond\neg\mathit{Enabled}\ \langle\mathcal{A}\rangle_f) \tag{18}$$

$$\mathrm{SF}_f(\mathcal{A}) \;\triangleq\; (\Box\Diamond\langle\mathcal{A}\rangle_f) \lor (\Diamond\Box\neg\mathit{Enabled}\ \langle\mathcal{A}\rangle_f) \tag{19}$$

$$
\begin{array}{l}
Init_{\Phi} \;\triangleq\; (x = 0) \;\wedge\; (y = 0) \\
\mathcal{M}_1 \;\triangleq\; (x' = x + 1) \;\wedge\; (y' = y) \qquad \mathcal{M}_2 \;\triangleq\; (y' = y + 1) \;\wedge\; (x' = x) \\
\mathcal{M} \;\triangleq\; \mathcal{M}_1 \vee \mathcal{M}_2 \\
\Phi \;\triangleq\; Init_{\Phi} \;\wedge\; \Box[\mathcal{M}]_{(x,y)} \;\wedge\; \mathrm{WF}_{(x,y)}(\mathcal{M}_1) \;\wedge\; \mathrm{WF}_{(x,y)}(\mathcal{M}_2)
\end{array}
$$

Figure 3: The TLA formula Φ describing Program 1.

6.4 Rewriting the Fairness Condition

We now rewrite the property $\Box\Diamond\langle\mathcal{M}_1\rangle_{(x,y)} \wedge \Box\Diamond\langle\mathcal{M}_2\rangle_{(x,y)}$ in terms of fairness conditions. An $\langle\mathcal{M}_1\rangle_{(x,y)}$ step is one that increments x by one, leaves y unchanged, and changes the value of (x, y). It is always possible to take a step that adds one to x and leaves y unchanged, and adding one to a number changes it. Hence, *Enabled* $\langle\mathcal{M}_1\rangle_{(x,y)}$ equals **true** throughout any
Note 9 execution of Program 1. Since $\Box\neg$**true** equals **false**, both $\mathrm{WF}_{(x,y)}(\mathcal{M}_1)$ and $\mathrm{SF}_{(x,y)}(\mathcal{M}_1)$ equal $\Box\Diamond\langle\mathcal{M}_1\rangle_{(x,y)}$. Similarly, $\mathrm{WF}_{(x,y)}(\mathcal{M}_2)$ and $\mathrm{SF}_{(x,y)}(\mathcal{M}_2)$ both equal $\Box\Diamond\langle\mathcal{M}_2\rangle_{(x,y)}$. We can therefore rewrite the definition (17) of Φ as shown in Figure 3 on this page.

Suppose we wanted the weaker liveness condition that execution never terminates, so the program is described by the RTLA formula (14) on page 15. The same argument as for $\mathcal{M}_1$ and $\mathcal{M}_2$ shows that $\Box\Diamond\langle\mathcal{M}\rangle_{(x,y)}$ equals $\mathrm{WF}_{(x,y)}(\mathcal{M})$. Therefore, Program 1 with this weaker liveness condition is described by the TLA formula $Init_{\Phi} \wedge \Box[\mathcal{M}]_{(x,y)} \wedge \mathrm{WF}_{(x,y)}(\mathcal{M})$.

6.5 Examining Formula Φ

TLA formulas that represent programs can always be written in the same form as Φ of Figure 3—that is, as a conjunction $Init \wedge \Box[\mathcal{N}]_f \wedge F$, where

Init is a predicate specifying the initial values of variables.

$\mathcal{N}$ is the program's *next-state relation*, the action whose steps represent executions of individual atomic operations.

f is the n-tuple of all program variables.

F is the conjunction of formulas of the form $\mathrm{WF}_f(\mathcal{A})$ and/or $\mathrm{SF}_f(\mathcal{A})$, where $\mathcal{A}$ is an action representing some subset of the program's atomic operations.

We now examine the behaviors that satisfy formula Φ. Let

$$\langle\!\langle x \triangleq 7,\ y \triangleq -10,\ z \triangleq \text{“}\mathsf{abc}\text{”},\ \ldots \rangle\!\rangle$$

denote a state s such that $s[\![x]\!] = 7$, $s[\![y]\!] = -10$, and $s[\![z]\!] = \text{“}\mathsf{abc}\text{”}$. (The "…" indicates that the value of $s[\![v]\!]$ is left unspecified for all other variables v.) A behavior that satisfies Φ begins in a state satisfying $Init_\Phi$, and consists of a sequence of $[\mathcal{M}]_{\langle x,y \rangle}$ steps—ones that are either $\mathcal{M}$ steps or else leave x and y unchanged. One such behavior is

$$\begin{array}{l}
\langle\!\langle\ x \triangleq 0,\ y \triangleq 0,\ z \triangleq \text{“}\mathsf{abc}\text{”}\ \ldots\ \rangle\!\rangle \\
\langle\!\langle\ x \triangleq 1,\ y \triangleq 0,\ z \triangleq \mathsf{true}\ \ \ldots\ \rangle\!\rangle \\
\langle\!\langle\ x \triangleq 2,\ y \triangleq 0,\ z \triangleq \mathsf{true}\ \ \ldots\ \rangle\!\rangle \\
\langle\!\langle\ x \triangleq 2,\ y \triangleq 0,\ z \triangleq -20\ \ \ldots\ \rangle\!\rangle \\
\langle\!\langle\ x \triangleq 2,\ y \triangleq 1,\ z \triangleq \mathsf{Nat}\ \ \ldots\ \rangle\!\rangle \\
\quad \vdots
\end{array}$$

Observe that Φ constrains only the values of x and y; it allows all other variables such as z to assume completely arbitrary values. Suppose Ψ is a formula describing a program that has no variables in common with Φ. Then a behavior satisfies $\Phi \wedge \Psi$ iff it represents an execution of both programs—that is, iff it describes a universe in which both Φ and Ψ are executed concurrently. Thus, $\Phi \wedge \Psi$ is the TLA formula representing the parallel composition of the two programs. In the realm of closed systems, conjunction represents parallel composition only in special cases—for example, when the programs do not interact, so they are represented by TLA formulas with no variables in common. The desire to make conjunction represent parallel composition Note 10
for interacting programs guides our approach to open systems, but that is a topic to be discussed elsewhere.

The observation that a single behavior can represent an execution of two or more noninteracting programs explains why we represent terminating as well as nonterminating executions by infinite behaviors. Termination of a program means that it has stopped; it does not mean that the entire universe has come to a halt. A terminating execution is represented by a behavior in which eventually all of the program's variables stop changing.

It is unusual in computer science for the semantics of a formula describing a program with variables x and y to involve other variables such as z. One of the keys to TLA's simplicity is that its semantics rests on a single, infinite set of variables—not on a different set of variables for each program. Thus, in TLA as in elementary logic, we can take the conjunction $F \wedge G$ and negation $\neg F$ of any formulas F and G—not just of formulas with properly matching variable declarations.

6.6 Simple TLA

We now complete the definition of *Simple TLA* by adding one more bit of notation. (The full logic, containing quantification, is introduced in Section 9.) It is convenient to define the action *Unchanged* f, for f a state function, by

$$\mathit{Unchanged}\ f \ \stackrel{\Delta}{=}\ f' = f$$

Thus, an *Unchanged* f step is one in which the value of f does not change.

The syntax and semantics of Simple TLA, along with the additional notation we use to write TLA formulas, are all summarized in Figure 4 on the next page. This figure explains all you need to know to understand TLA formulas such as formula Φ of Figure 3.

A logic contains not only syntax and semantics, but also rules for proving theorems. Figure 5 on page 22 lists all the axioms and proof rules we need for proving simple TLA formulas.[3] We will not prove the soundness of these rules here.

The rules of simple temporal logic are used to derive temporal tautologies—formulas that are true regardless of the meanings of their elementary

Note 11 formulas. Rule STL1 encompasses the rules of ordinary logic, such as *modus ponens*. The Lattice Rule assumes a (possibly infinite) nonempty set of formulas H_{c} indexed by elements of a set S. A partial order $\succ$ on S is *well-founded* iff there exists no infinite descending chain $\mathsf{c}_1 \succ \mathsf{c}_2 \succ \ldots$ with all the c_i in S. This rule permits the formalization of counting-down arguments, such as the ones traditionally used to prove termination of sequential programs.

Rules STL1–STL6, the Lattice Rule, and the basic rules TLA1 and TLA2 form a relatively complete proof system for reasoning about algorithms in TLA. Roughly speaking, this means that every valid TLA formula that we must prove to verify properties of algorithms would be provable from these rules if we could prove all valid action formulas. (This is analogous to the traditional relative completeness results for program verification, which

[3] A proof rule $\frac{F}{G}$ asserts that $\vdash F$ implies $\vdash G$. We use the term "rule" for both axioms and proof rules, since an axiom may be viewed as a proof rule with no hypotheses.

Syntax

$$
\begin{array}{lcl}
\langle \mathit{formula} \rangle & \triangleq & \langle \mathit{predicate} \rangle \;\mid\; \Box[\langle \mathit{action} \rangle]_{\langle \mathit{state\ function} \rangle} \;\mid\; \neg\langle \mathit{formula} \rangle \\
 & & \quad \mid\; \langle \mathit{formula} \rangle \wedge \langle \mathit{formula} \rangle \;\mid\; \Box\, \langle \mathit{formula} \rangle \\
\langle \mathit{action} \rangle & \triangleq & \text{boolean-valued expression containing constants,} \\
 & & \quad \text{variables, and primed variables} \\
\langle \mathit{predicate} \rangle & \triangleq & \text{boolean-valued } \langle \mathit{state\ function} \rangle \;\mid\; \mathit{Enabled}\ \langle \mathit{action} \rangle \\
\langle \mathit{state\ function} \rangle & \triangleq & \text{expression containing constants and variables}
\end{array}
$$

Semantics

$$
\begin{array}{lcllcl}
s[\![f]\!] & \triangleq & f(\forall\, `v\text{'} : s[\![v]\!]/v) & \sigma[\![F \wedge G]\!] & \triangleq & \sigma[\![F]\!] \wedge \sigma[\![G]\!] \\
s[\![\mathcal{A}]\!]t & \triangleq & \mathcal{A}(\forall\, `v\text{'} : s[\![v]\!]/v,\, t[\![v]\!]/v') & \sigma[\![\neg F]\!] & \triangleq & \neg\sigma[\![F]\!] \\
\models \mathcal{A} & \triangleq & \forall s, t \in \mathbf{St} : \models s[\![\mathcal{A}]\!]t & \models F & \triangleq & \forall \sigma \in \mathbf{St}^{\infty} : \models \sigma[\![F]\!]
\end{array}
$$

$$
\begin{array}{lcl}
s[\![\mathit{Enabled}\ \mathcal{A}]\!] & \triangleq & \exists t \in \mathbf{St} : s[\![\mathcal{A}]\!]t \\
\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle[\![\Box F]\!] & \triangleq & \forall n \in \mathsf{Nat} : \langle\!\langle s_n, s_{n+1}, \ldots \rangle\!\rangle[\![F]\!] \\
\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle[\![\mathcal{A}]\!] & \triangleq & s_0[\![\mathcal{A}]\!]s_1
\end{array}
$$

Additional notation

$$
\begin{array}{lcllcl}
f' & \triangleq & f(\forall\, `v\text{'} : v'/v) & \Diamond F & \triangleq & \neg\Box\neg F \\
[\mathcal{A}]_f & \triangleq & \mathcal{A} \vee (f' = f) & F \leadsto G & \triangleq & \Box(F \Rightarrow \Diamond G) \\
\langle \mathcal{A} \rangle_f & \triangleq & \mathcal{A} \wedge (f' \neq f) & \mathrm{WF}_f(\mathcal{A}) & \triangleq & \Box\Diamond\langle \mathcal{A} \rangle_f \vee \Box\Diamond\neg \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \\
\mathit{Unchanged}\ f & \triangleq & f' = f & \mathrm{SF}_f(\mathcal{A}) & \triangleq & \Box\Diamond\langle \mathcal{A} \rangle_f \vee \Diamond\Box\neg \mathit{Enabled}\ \langle \mathcal{A} \rangle_f
\end{array}
$$

where f is a $\langle \mathit{state\ function} \rangle$ — s, s_0, s_1, $\ldots$ are states
$\mathcal{A}$ is an $\langle \mathit{action} \rangle$ — σ is a behavior
F, G are $\langle \mathit{formula} \rangle$s — $(\forall\, `v\text{'} : \ldots/v,\, \ldots/v')$ denotes substitution for all variables v

Figure 4: Simple TLA.

The Rules of Simple Temporal Logic

STL1. $\dfrac{F \text{ provable by propositional logic}}{F}$

STL2. $\vdash \Box F \;\Rightarrow\; F$

STL3. $\vdash \Box\Box F \;\equiv\; \Box F$

STL4. $\dfrac{F \Rightarrow G}{\Box F \;\Rightarrow\; \Box G}$

STL5. $\vdash \Box(F \wedge G) \;\equiv\; (\Box F) \wedge (\Box G)$

STL6. $\vdash (\Diamond\Box F) \wedge (\Diamond\Box G) \;\equiv\; \Diamond\Box(F \wedge G)$

LATTICE. $\succ$ well-founded partial order on nonempty set S

$$\frac{F \wedge (\mathsf{c} \in \mathsf{S}) \;\Rightarrow\; (H_{\mathsf{c}} \leadsto (G \vee \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}}))}{F \;\Rightarrow\; ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \leadsto G)}$$

The Basic Rules of TLA

TLA1. $\vdash \Box P \;\equiv\; P \wedge \Box[P \Rightarrow P']_P$

TLA2. $\dfrac{P \wedge [\mathcal{A}]_f \;\Rightarrow\; Q \wedge [\mathcal{B}]_g}{\Box P \wedge \Box[\mathcal{A}]_f \;\Rightarrow\; \Box Q \wedge \Box[\mathcal{B}]_g}$

Additional Rules

INV1. $\dfrac{I \wedge [\mathcal{N}]_f \Rightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$

INV2. $\vdash \Box I \;\Rightarrow\; (\Box[\mathcal{N}]_f \;\equiv\; \Box[\mathcal{N} \wedge I \wedge I']_f)$

WF1.
$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A}\rangle_f \;\Rightarrow\; Q' \\ P \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A}\rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \;\Rightarrow\; (P \leadsto Q)}$$

WF2.
$$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B}\rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}}\rangle_{\overline{g}} \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A}\rangle_f \;\Rightarrow\; \mathcal{B} \\ P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M}\rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A}\rangle_f \\ \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \Diamond\Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathrm{WF}_g(\mathcal{M})}}$$

SF1.
$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A}\rangle_f \;\Rightarrow\; Q' \\ \Box P \wedge \Box[\mathcal{N}]_f \wedge \Box F \;\Rightarrow\; \Diamond \mathit{Enabled}\ \langle \mathcal{A}\rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{SF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; (P \leadsto Q)}$$

SF2.
$$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B}\rangle_f \;\Rightarrow\; \langle \overline{M}\rangle_{\overline{g}} \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A}\rangle_f \;\Rightarrow\; \mathcal{B} \\ P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M}\rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A}\rangle_f \\ \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathrm{SF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \Diamond\Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{SF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathrm{SF}_g(\mathcal{M})}}$$

where F, G, H_{c} are TLA formulas; P, Q, I are predicates; $\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{M}$ are actions; f, g are state functions

Figure 5: The axioms and proof rules of Simple TLA.

assume provability of all valid predicates [4].) The precise statement and proof of the completeness theorem will appear elsewhere.

A complete proof system is not necessarily a convenient one. There are a number of rules that are useful in practice but are either difficult or impossible to derive from STL1–TLA2. For a practical system, STL1–STL6 should be expanded to include some useful temporal tautologies like

$$\vdash (\Box F) \wedge (\Diamond G) \;\Rightarrow\; \Diamond(F \wedge G)$$

With practice, such simple tautologies become as obvious as the ordinary laws of propositional logic, and they are usually assumed in proofs that are not machine-checked. We will not bother to provide a complete list of the useful rules and theorems of simple temporal logic.

Assuming simple temporal reasoning, we have found that TLA2 and the "additional rules" INV1–SF2 of Figure 5 provide a convenient system for all the proofs that arise in reasoning about programs with TLA. The overbars in rules WF2 and SF2 are explained in Section 9.3; for now, the reader can pretend that they are not there, obtaining special cases of the rules.

We do not know how to prove "practical completeness", and there are probably examples in which TLA2 and INV1–SF2 are inadequate, complicated reasoning with TLA1 and TLA2 being necessary. However, we have not yet encountered such examples and we believe they will be rare.

7 Proving Simple Properties of Programs

Having expressed Program 1 of Figure 1 as the TLA formula Φ of Figure 3, we now consider how to express and prove properties of such a program. A property is expressed by a TLA formula F. The assertion "program Φ has property F" is expressed in TLA by the validity of the formula $\Phi \Rightarrow F$, which asserts that every behavior satisfying Φ satisfies F. We consider two popular classes of properties, invariance and eventuality.

7.1 Invariance Properties

7.1.1 Definition

An invariance property is expressed by a TLA formula $\Box P$, where P is a predicate. Examples of invariance properties include

partial correctness P asserts that if the program has terminated, then the answer is correct.

deadlock freedom P asserts that the program is not deadlocked.

mutual exclusion P asserts that at most one process is in its critical section.

Invariance properties are proved with rule INV1 of Figure 5. Its hypothesis asserts that any $[\mathcal{N}]_f$ step starting with I true in the old state leaves I true in the new state. The conclusion asserts that, for any behavior that starts in a state with I true and has only $[\mathcal{N}]_f$ steps, I is true in all states. The soundness of this deduction is proved by a simple induction argument.

7.1.2 An Example: Type Correctness

One part of the program in Figure 1 does not correspond to anything in Figure 3—the type declaration of the variables x and y. Such a declaration is not needed because type-correctness is an invariance property of the program, asserting that x and y are always natural numbers. We illustrate invariance proofs by proving type correctness of Program 1. Type correctness is expressed formally as $\Phi \Rightarrow \Box T$, where

$$T \;\triangleq\; (x \in \mathsf{Nat}) \wedge (y \in \mathsf{Nat}) \tag{20}$$

Rule INV1 tells us that we must prove

$$Init_\Phi \Rightarrow T \tag{21}$$

$$T \wedge [\mathcal{M}]_{(x,y)} \Rightarrow T' \tag{22}$$

from which we deduce $\Phi \Rightarrow \Box T$ as follows

$$\begin{array}{rcll} \Phi & \Rightarrow & Init_\Phi \wedge \Box[\mathcal{M}]_{(x,y)} & \{\text{by definition of } \Phi \text{ (Figure 3)}\} \\ & \Rightarrow & T \wedge \Box[\mathcal{M}]_{(x,y)} & \{\text{by (21)}\} \\ & \Rightarrow & \Box T & \{\text{by (22) and INV1}\} \end{array}$$

The proof of (21) is trivial. The proof of (22) is quite simple, but we will sketch it to show how the structure of the formulas leads to a natural decomposition of the proof. First, we expand the definition of $[\mathcal{M}]_{(x,y)}$.

$$\begin{array}{rcll} [\mathcal{M}]_{(x,y)} & \equiv & \mathcal{M} \vee ((x,y)' = (x,y)) & \{\text{by (12)}\} \\ & \equiv & \mathcal{M}_1 \vee \mathcal{M}_2 \vee ((x,y)' = (x,y)) & \{\text{by definition of } \mathcal{M}\} \end{array}$$

Since $[\mathcal{M}]_{(x,y)}$ is the disjunction of three actions, the proof of (22) decomposes into the proof of three simpler formulas:

$$T \wedge \mathcal{M}_1 \;\Rightarrow\; T' \tag{23}$$
$$T \wedge \mathcal{M}_2 \;\Rightarrow\; T' \tag{24}$$
$$T \wedge ((x, y)' = (x, y)) \;\Rightarrow\; T' \tag{25}$$

We consider the proof of (23); the others are equally simple. First, we expand the definition of T'.

$$\begin{array}{rcll} T' & \equiv & ((x \in \mathsf{Nat}) \wedge (y \in \mathsf{Nat}))' & \{\text{by (20)}\} \\ & \equiv & (x' \in \mathsf{Nat}) \wedge (y' \in \mathsf{Nat}) & \{\text{by (3)}\} \end{array}$$

The structure of T' as the conjunction of two actions leads to the decomposition of the proof of (23) into the proof of the two simpler formulas

$$T \wedge \mathcal{M}_1 \;\Rightarrow\; x' \in \mathsf{Nat} \tag{26}$$
$$T \wedge \mathcal{M}_1 \;\Rightarrow\; y' \in \mathsf{Nat} \tag{27}$$

The proof of (26) is

$$\begin{array}{rcll} T \wedge \mathcal{M}_1 & \Rightarrow & (x \in \mathsf{Nat}) \wedge (x' = x + 1) & \{\text{by definition of } T \text{ and } \mathcal{M}_1\} \\ & \Rightarrow & x' \in \mathsf{Nat} & \{\text{by properties of natural numbers}\} \end{array}$$

and the proof of (27) is equally trivial.

The purpose of this exercise in simple mathematics is to illustrate how "mechanical" the proof of this invariance property is. Rule INV1 tells us we must prove (21) and (22), and the structure of $[\mathcal{M}]_{(x,y)}$ and T leads to the decomposition of those proofs into the verification of simple facts about natural numbers, such as $(x \in \mathsf{Nat}) \Rightarrow (x + 1 \in \mathsf{Nat})$.

7.1.3 General Invariance Proofs

The proof of $\Phi \Rightarrow \Box T$ was simple because T is an *invariant* of the action $[\mathcal{M}]_{(x,y)}$, meaning that $T \wedge [\mathcal{M}]_{(x,y)}$ implies T'. Therefore, $\Phi \Rightarrow \Box T$ could be proved by simply substituting T for I in rule INV1. For invariance properties $\Box P$ other than simple type correctness, P is usually not an invariant. In general, one proves that $\Box P$ is an invariance property of the program represented by the TLA formula $\mathit{Init} \wedge \Box[\mathcal{N}]_f \wedge F$ by finding a predicate I (the invariant) satisfying the three conditions

$$\mathit{Init} \Rightarrow I \tag{28}$$
$$I \Rightarrow P \tag{29}$$
$$I \wedge [\mathcal{N}]_f \Rightarrow I' \tag{30}$$

Rule INV1 and some simple temporal reasoning shows that (28)–(30) imply $Init \wedge \Box[\mathcal{N}]_f \Rightarrow \Box P$.

Creative thought is needed to find the invariant I. Once I is found, verifying (28)–(30) is a matter of mechanically applying the definitions and using the structure of the formulas to decompose the proofs, just as in the proof of $\Phi \Rightarrow \Box T$ above. The formulas $Init$, I, and $\mathcal{N}$ will usually be much more complicated than in the example, but the principle is the same.

Formulas (28)–(30) are assertions about predicates and actions; they are not temporal formulas. All the work in proving an invariance property is done in the realm of predicates and actions—expressions involving variables and primed variables that can be manipulated by ordinary mathematics. Temporal reasoning is used only to deduce $Init \wedge \Box[\mathcal{N}]_f \Rightarrow \Box P$ from (28)–(30). It is this reduction of temporal properties to ordinary, nontemporal reasoning that makes TLA practical.

7.1.4 More About Invariance Proofs

Over the years, many methods have been proposed to prove invariance properties of programs, including Floyd's method [10], Hoare logic [13], and the Owicki-Gries method [19]. All of these methods are essentially the same—when applied to the same program, they involve the same proof steps, though perhaps in different orders and with different notation. These methods can be described formally in TLA as applications of rule INV1. The advantage of TLA is that the proof method arises directly from the logic, without the need for proof rules based on a particular programming language.

We illustrate the advantage of working in a simple logic by considering the use of one invariance property to prove another. We have just proved $\Phi \Rightarrow \Box T$, the assertion that the program satisfies the invariance property $\Box T$. How can we use this fact when proving that the program satisfies a second invariance property $\Box P$? Some methods have a special rule saying that if a program satisfies $\Box T$, then one can pretend that T is true when reasoning about the program. (The "substitution axiom" of Unity [6] is such a rule.) In TLA, we use Rule INV2 of Figure 5. This rule implies that having proved $\Phi \Rightarrow \Box T$, we can rewrite the definition of Φ in Figure 3 as

$$\Phi \;\triangleq\; Init_\Phi \;\wedge\; \Box[\mathcal{M} \wedge T \wedge T']_{(x,y)} \;\wedge\; \mathrm{WF}_{(x,y)}(\mathcal{M}_1) \;\wedge\; \mathrm{WF}_{(x,y)}(\mathcal{M}_2)$$

It follows that in proving Φ implies $\Box P$, instead of assuming every step to be a $[\mathcal{M}]_{(x,y)}$ step, we can make the stronger assumption that every step is a $[\mathcal{M} \wedge T \wedge T']_{(x,y)}$ step. More precisely, we can substitute $\mathcal{M} \wedge T \wedge T'$ instead of $\mathcal{M}$ for $\mathcal{N}$ in rule INV1, giving a stronger proof rule. This stronger rule is tantamount to "pretending T is true". The validity of this pretense follows directly from the logic; it is not an *ad hoc* rule.

7.1.5 More About Types

In TLA, variables have no types. Any variable can assume any value. Type-correctness of a program is a provable property, not a syntactic requirement as in strongly-typed programming languages. This has some subtle consequences. Consider the action $x' = x + 1$. Its meaning is a boolean-valued function on pairs of states. Suppose s and t are states that assign the values "abc" and true to x, respectively—that is, so $s[\![x]\!]$ equals "abc" and $t[\![x]\!]$ equals true. Then $s[\![x' = x + 1]\!]t$ equals $\mathsf{true} =$ "abc" $+ 1$. But what is "abc" $+ 1$? Does it equal true?

We don't know the answers to these questions, and we don't care. All we know is that "abc" $+ 1$ is some value. Since that value is either equal to or unequal to true, the expression $\mathsf{true} =$ "abc" $+ 1$ is equal to either true or false. More precisely, we assume that $\mathsf{m} + \mathsf{n}$ is an element of the set Val of values, for any m and n in Val. However, we have no rules for deducing anything about the value of $\mathsf{m} + \mathsf{n}$ except when m and n are numbers. In general, we assume that all functions such as "+" are *total*—their domains include all elements of Val, and their range is (a subset of) Val. What we usually think of as the domain of a function is just the set of values for which we know how to evaluate the function. We know how to evaluate $\mathsf{p} \wedge \mathsf{q}$ only when p and q are elements of the set Bool of booleans, but it is defined (in the mathematical sense of being a meaningful expression) for all values p and q.

Since we can't deduce anything about the value "abc" $+ 1$, whatever we prove about an algorithm is true regardless of what that value is. If we can prove that the program is correct, then either it will never add 1 to "abc" (as in the case of Program 1), or else correctness does not depend on the result of that addition.

This approach may seem strange to computer scientists used to types in programming languages, but it captures the way mathematicians have reasoned for thousands of years. For example, mathematicians would say that the formula

$$(\mathsf{n} \in \mathsf{Nat}) \Rightarrow (\mathsf{n} + 1 > \mathsf{n})$$

is true for all n. Substituting "abc" for n yields

$$(\text{"abc"} \in \mathsf{Nat}) \Rightarrow (\text{"abc"} + 1 > \text{"abc"})$$

This formula equals **true** regardless of what "abc" $+1$ equals, and whether or not that value is greater than "abc", because "abc" $\in \mathsf{Nat}$ equals **false**. The formula is not meaningless or "type-incorrect" just because we don't know the value of "abc" $+ 1$; we don't need any complicated three-valued logic to understand the formula.

Note 12

Types are a useful feature in programming languages, but they are a needless complication when reasoning about algorithms. Logic should be simple, and one of the ways we keep TLA simple is by not complicating it with any notion of types.

We have found computer scientists to be skeptical of our eliminating types from logic. They seem to feel that type-checking is easier than proving a property like $\Box T$. This is nonsense. Type-checking the program of Figure 1 is logically equivalent to proving the TLA formula $\Phi \Rightarrow \Box T$, so a type-checker must perform exactly the same reasoning that we did. Adding strong typing to the logic would mean that a TLA formula would be type-correct only if the proof of the corresponding invariance property were so trivial that it could be done automatically. If type-correctness of the program were not completely trivial, so its proof required even the tiniest bit of intelligence, then strong typing would prohibit us from expressing it in the logic.

We can assure the reader that, just as mathematics thrived for thousands of years without type theory, logic works fine without types. Experience has taught us that adding types would only make TLA less convenient to use. The absence of types in TLA does not prevent mechanized type checking. One can still write a program that tries to prove the TLA formula representing type correctness, and complains if it fails. But only an intelligent human can determine if the type checker failed because of its stupidity, or because the TLA formula is invalid. If he expects to verify interesting properties of concurrent algorithms, our human had better be intelligent enough to prove simple type correctness.

7.2 Eventuality Properties

The second class of properties we consider are eventuality properties—ones asserting that something eventually happens. Here are some traditional eventuality properties and their expressions in temporal logic:

termination The program eventually terminates: $\Diamond$ *terminated*.

service If a process has requested service, then it eventually is served: $requested \leadsto served$.

message delivery If a message is sent often enough, then it is eventually delivered: $(\Box\Diamond\, sent) \Rightarrow \Diamond\, delivered$.

Although eventuality properties are expressed by a variety of temporal formulas, their proofs are always reduced to the proof of leads-to properties—formulas of the form $P \leadsto Q$. For example, suppose we want to prove that Program 1 increases the value of x without bound. The TLA formula to be proved is

$$\Phi \wedge (\mathsf{n} \in \mathsf{Nat}) \;\Rightarrow\; \Diamond(x > \mathsf{n}) \tag{31}$$

The Lattice Rule of Figure 5 together with some simple temporal reasoning shows that (31) follows from

$$\Phi \;\Rightarrow\; ((\mathsf{n} \in \mathsf{Nat} \,\wedge\, x = \mathsf{n}) \leadsto (x = \mathsf{n} + 1)) \tag{32}$$

To illustrate the use of TLA in proving leads-to properties, we now sketch the proof of (32).

Since safety properties don't imply that anything ever happens, leads-to properties must be derived from the program's fairness condition. Examining Figure 5 leads us to try rule WF1, with the following substitutions:

$$\begin{array}{lll} P \leftarrow \mathsf{n} \in \mathsf{Nat} \,\wedge\, x = \mathsf{n} & \mathcal{N} \leftarrow \mathcal{M} & f \leftarrow (x, y) \\ Q \leftarrow x = \mathsf{n} + 1 & \mathcal{A} \leftarrow \mathcal{M}1 & \end{array}$$

The rule's hypotheses become

$$\begin{array}{l} (\mathsf{n} \in \mathsf{Nat} \,\wedge\, x = \mathsf{n}) \wedge [\mathcal{M}]_{(x,y)} \;\Rightarrow\; ((\mathsf{n} \in \mathsf{Nat} \,\wedge\, x' = \mathsf{n}) \vee (x' = \mathsf{n} + 1)) \\ (\mathsf{n} \in \mathsf{Nat} \,\wedge\, x = \mathsf{n}) \wedge \langle \mathcal{M}1 \rangle_{(x,y)} \;\Rightarrow\; (x' = \mathsf{n} + 1) \\ (\mathsf{n} \in \mathsf{Nat} \,\wedge\, x = \mathsf{n}) \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{M}1 \rangle_{(x,y)} \end{array}$$

which follow easily from the definitions of $\mathcal{M}1$ and $\mathcal{M}$ in Figure 3. The rule's conclusion becomes

$$\Box[\mathcal{M}]_{(x,y)} \wedge \mathrm{WF}_{(x,y)}(\mathcal{M}1) \;\Rightarrow\; ((\mathsf{n} \in \mathsf{Nat} \,\wedge\, x = \mathsf{n}) \leadsto (x = \mathsf{n} + 1))$$

which, by definition of Φ, implies (32).

7.3 Other Properties

We have seen how invariance properties and eventuality properties are expressed as TLA formulas and proved. But, what about more complicated properties? How would one state the following property as a TLA formula?

> A behavior begins with x and y both zero, and repeatedly increments either x or y (in a single operation), choosing nondeterministically between them, but choosing each infinitely many times.

The answer, of course, it that we already have expressed this property in TLA. It is formula Φ of Figure 3.

In TLA, there is no distinction between a program and a property. Instead of viewing Φ as a description of a program, we can just as well consider it to be a property that we want a program to satisfy. The formula Φ, like the program of Figure 1 that it represents, is so simple that we can regard it as a specification of how we want a program to behave. As our next example, we consider a program that implements property Φ. That is, we will give a program represented by a TLA formula Ψ that implies Φ.

8 Another Example

8.1 Program 2

Our next example is Program 2 of Figure 6 on the next page, written in a language invented for this program. (Since its only purpose is to help us write the TLA formula, the programming-language description of the program can be written with any convenient notation.) The program consists of two processes, each repeatedly executing a loop that contains three atomic operations. The variable sem is an integer semaphore, and P and V are the standard semaphore operations [8]. Since Figure 6 is an informal description, it doesn't matter whether or not you understand it. The real definition of Program 2 is the TLA formula Ψ defined below.

Describing the execution of Program 2 as a sequence of states requires each state to specify not only the values of the variables x, y, and sem, but also the control state of each process. Control in process 1 can be at one of the three "control points" α_1, β_1, or γ_1. We introduce the variable pc_1 that will assume the values "a", "b", and "g", denoting that control is at

```
var  integer  x, y       = 0 ;
     semaphore sem       = 1 ;
cobegin  loop  α1: ⟨ P(sem) ⟩ ;
               β1: ⟨ x := x + 1 ⟩ ;
               γ1: ⟨ V(sem) ⟩        endloop
   □
         loop  α2: ⟨ P(sem) ⟩ ;
               β2: ⟨ y := y + 1 ⟩ ;
               γ2: ⟨ V(sem) ⟩        endloop
coend
```

Figure 6: Program 2—our second example program.

α_1, β_1, and γ_1, respectively. A similar variable pc_2 denotes the control state of process 2.

The definition of the TLA formula Ψ that represents Program 2 is given in Figure 7 on the next page.[4] We have used the following convention to make our formulas easier to read: a list of formulas preceded by "$\wedge$"s denotes the conjunction of those formulas. Thus, Figure 7 defines the predicate $Init_\Psi$ to be the conjunction of three formulas, the second of which is $(x = 0) \wedge (y = 0)$.

As we explained in Section 6.5 (page 18), a program is represented by a formula $Init \wedge \Box[\mathcal{N}]_f \wedge F$. In this example, the formulas $Init$ and f are fairly obvious: $Init$ is the predicate $Init_\Psi$ that specifies the initial values of the variables, and f is the 5-tuple w consisting of all the program's variables. The next-state relation $\mathcal{N}$ and the fairness condition F are less obvious and merit some discussion.

8.1.1 The Next-State Relation

Corresponding to the six atomic operations in Figure 6 are the six actions $\alpha_1, \ldots, \gamma_2$ defined in Figure 7. The four conjuncts in the definition of α_1 assert that an α_1 step:

1. Starts in a state with $pc_1 =$ "a" (control in the first process is at control point α_1) and $0 < sem$ (the semaphore is positive).

2. Ends in a state with $pc_1 =$ "b" (control in the first process is at control point β_1).

[4] Section 10.2 discusses why Figure 7 is longer and seems more complex than Figure 6.

$$
\begin{array}{lcl}
Init_{\Psi} & \triangleq & \land\ (pc_1 = \text{“}\mathsf{a}\text{”}) \land (pc_2 = \text{“}\mathsf{a}\text{”}) \\
 & & \land\ (x = 0) \land (y = 0) \\
 & & \land\ sem = 1
\end{array}
$$

$$
\begin{array}{lcl}
\alpha_1 & \triangleq & \land\ (pc_1 = \text{“}\mathsf{a}\text{”}) \land (0 < sem) \\
 & & \land\ pc_1' = \text{“}\mathsf{b}\text{”} \\
 & & \land\ sem' = sem - 1 \\
 & & \land\ \mathit{Unchanged}\ (x, y, pc_2) \\
\beta_1 & \triangleq & \land\ pc_1 = \text{“}\mathsf{b}\text{”} \\
 & & \land\ pc_1' = \text{“}\mathsf{g}\text{”} \\
 & & \land\ x' = x + 1 \\
 & & \land\ \mathit{Unchanged}\ (y, sem, pc_2) \\
\gamma_1 & \triangleq & \land\ pc_1 = \text{“}\mathsf{g}\text{”} \\
 & & \land\ pc_1' = \text{“}\mathsf{a}\text{”} \\
 & & \land\ sem' = sem + 1 \\
 & & \land\ \mathit{Unchanged}\ (x, y, pc_2) \\
\mathcal{N}_1 & \triangleq & \alpha_1 \lor \beta_1 \lor \gamma_1
\end{array}
\qquad
\begin{array}{lcl}
\alpha_2 & \triangleq & \land\ (pc_2 = \text{“}\mathsf{a}\text{”}) \land (0 < sem) \\
 & & \land\ pc_2' = \text{“}\mathsf{b}\text{”} \\
 & & \land\ sem' = sem - 1 \\
 & & \land\ \mathit{Unchanged}\ (x, y, pc_1) \\
\beta_2 & \triangleq & \land\ pc_2 = \text{“}\mathsf{b}\text{”} \\
 & & \land\ pc_2' = \text{“}\mathsf{g}\text{”} \\
 & & \land\ y' = y + 1 \\
 & & \land\ \mathit{Unchanged}\ (x, sem, pc_1) \\
\gamma_2 & \triangleq & \land\ pc_2' = \text{“}\mathsf{a}\text{”} \\
 & & \land\ pc_2 = \text{“}\mathsf{g}\text{”} \\
 & & \land\ sem' = sem + 1 \\
 & & \land\ \mathit{Unchanged}\ (x, y, pc_1) \\
\mathcal{N}_2 & \triangleq & \alpha_2 \lor \beta_2 \lor \gamma_2
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & \mathcal{N}_1 \lor \mathcal{N}_2 \\
w & \triangleq & (x, y, sem, pc_1, pc_2) \\
\Psi & \triangleq & Init_{\Psi} \land \Box[\mathcal{N}]_w \land \mathrm{SF}_w(\mathcal{N}1) \land \mathrm{SF}_w(\mathcal{N}2)
\end{array}
$$

Figure 7: The formula Ψ describing Program 2.

3. Decrements sem.

4. Does not change the values of x, y, and pc_2

Thus, an α_1 step represents an execution of statement α_1 of Figure 6. Similarly, the other actions represent the other operations of the program in Figure 6.

An $\mathcal{N}_1$ step is either an α_1 step, a β_1 step, or a γ_1 step, so it represents an execution of an atomic operation by the first process. Similarly, an $\mathcal{N}_2$ step represents an execution of an atomic operation by the second process. An $\mathcal{N}$ step represents a step of either process, so every program step is an $\mathcal{N}$ step—in other words, $\mathcal{N}$ is the program's next-state relation. Thus, $\Box[\mathcal{N}]_w$ is true for a behavior iff every step of the behavior is either a program step or else leaves the variables x, y, sem, pc_1, and pc_2 unchanged.

8.1.2 The Fairness Condition

We want program Ψ to implement program Φ. Hence, Ψ must guarantee that both x and y are incremented infinitely often. To guarantee that x is incremented infinitely often, we need some fairness condition to ensure that infinitely many $\mathcal{N}_1$ steps occur. This condition must rule out the following behavior, in which process 1 is never executed.

$$
\begin{array}{l}
\langle\!\langle x \triangleq 0,\ y \triangleq 0,\ sem \triangleq 1,\ pc_1 \triangleq \text{“a”},\ pc_2 \triangleq \text{“a”},\ \ldots \rangle\!\rangle \\
\langle\!\langle x \triangleq 0,\ y \triangleq 0,\ sem \triangleq 0,\ pc_1 \triangleq \text{“a”},\ pc_2 \triangleq \text{“b”},\ \ldots \rangle\!\rangle \\
\langle\!\langle x \triangleq 0,\ y \triangleq 1,\ sem \triangleq 0,\ pc_1 \triangleq \text{“a”},\ pc_2 \triangleq \text{“g”},\ \ldots \rangle\!\rangle \\
\langle\!\langle x \triangleq 0,\ y \triangleq 1,\ sem \triangleq 1,\ pc_1 \triangleq \text{“a”},\ pc_2 \triangleq \text{“a”},\ \ldots \rangle\!\rangle \\
\langle\!\langle x \triangleq 0,\ y \triangleq 1,\ sem \triangleq 0,\ pc_1 \triangleq \text{“a”},\ pc_2 \triangleq \text{“b”},\ \ldots \rangle\!\rangle \\
\langle\!\langle x \triangleq 0,\ y \triangleq 2,\ sem \triangleq 0,\ pc_1 \triangleq \text{“a”},\ pc_2 \triangleq \text{“g”},\ \ldots \rangle\!\rangle \\
\langle\!\langle x \triangleq 0,\ y \triangleq 2,\ sem \triangleq 1,\ pc_1 \triangleq \text{“a”},\ pc_2 \triangleq \text{“a”},\ \ldots \rangle\!\rangle \\
\quad\vdots
\end{array}
$$

Observe that an α_1 step is possible iff pc_1 equals “a” and sem is positive, so *Enabled* α_1 equals $(pc_1 = \text{“a”}) \wedge (0 < sem)$. In this behavior, *Enabled* α_1 is true whenever pc_2 equals “a”, and false otherwise—both situations occurring infinitely often. An α_1 step is also an $\mathcal{N}_1$ step. Moreover, every α_1 step changes pc_1 and sem, so it changes w. Hence, any α_1 step is an $\langle\mathcal{N}_1\rangle_w$ step, so $\langle\mathcal{N}_1\rangle_w$ is enabled and disabled infinitely often in this behavior.

The weak fairness condition $\mathrm{WF}_w(\mathcal{N}_1)$ asserts that $\langle\mathcal{N}_1\rangle_w$ is disabled infinitely often or infinitely many $\langle\mathcal{N}_1\rangle_w$ steps occur. Since $\langle\mathcal{N}_1\rangle_w$ is disabled infinitely often, $\mathrm{WF}_w(\mathcal{N}_1)$ does not rule out this behavior.

The strong fairness condition $\mathrm{SF}_w(\mathcal{N}_1)$ asserts that either $\langle\mathcal{N}_1\rangle_w$ is eventually forever disabled or else infinitely many $\langle\mathcal{N}_1\rangle_w$ steps occur. Neither assertion is true for this behavior, so the behavior does not satisfy $\mathrm{SF}_w(\mathcal{N}_1)$. This example indicates why we need the fairness condition $\mathrm{SF}_w(\mathcal{N}_1)$ to guarantee that x is incremented infinitely often.

There are other ways of writing this fairness condition. An equivalent definition of Ψ is obtained by replacing $\mathrm{SF}_w(\mathcal{N}_1)$ with $\mathrm{SF}_w(\alpha_1) \wedge \mathrm{SF}_w(\beta_1) \wedge \mathrm{SF}_w(\gamma_1)$ or with $\mathrm{SF}_w(\alpha_1) \wedge \mathrm{WF}_w(\beta_1) \wedge \mathrm{WF}_w(\gamma_1)$. Equivalence of these definitions follows from the formulas

$$\begin{aligned}
&\mathit{Init}_\Psi \wedge \Box[\mathcal{N}]_w \;\Rightarrow\; \mathrm{SF}_w(\mathcal{N}_1) \equiv \begin{array}[t]{l} \wedge\, \mathrm{SF}_w(\alpha_1) \\ \wedge\, \mathrm{SF}_w(\beta_1) \\ \wedge\, \mathrm{SF}_w(\gamma_1) \end{array} && (33)\\
&\mathit{Init}_\Psi \wedge \Box[\mathcal{N}]_w \;\Rightarrow\; \mathrm{SF}_w(\beta_1) \equiv \mathrm{WF}_w(\beta_1) && (34)\\
&\mathit{Init}_\Psi \wedge \Box[\mathcal{N}]_w \;\Rightarrow\; \mathrm{SF}_w(\gamma_1) \equiv \mathrm{WF}_w(\gamma_1) && (35)
\end{aligned}$$

Intuitively, (33) holds because once control reaches α_1, β_1, or γ_1, it remains there until the corresponding action is executed; (34) holds because once control reaches β_1, action β_1 is enabled until it is executed; and (35) is similar to (34).

Corresponding reasoning about y and $\mathcal{N}_2$ leads to the fairness condition $\mathrm{SF}_w(\mathcal{N}_2)$ for the second process.

8.2 Proving Program 2 Implements Program 1

To show that Program 2 implements Program 1, we must prove the TLA formula $\Psi \Rightarrow \Phi$, where Ψ is defined in Figure 7 on page 32 and Φ is defined in Figure 3 on page 18. By these definitions, $\Psi \Rightarrow \Phi$ follows from the following three formulas.

$$\begin{aligned}
\mathit{Init}_\Psi &\;\Rightarrow\; \mathit{Init}_\Phi && (36)\\
\Box[\mathcal{N}]_w &\;\Rightarrow\; \Box[\mathcal{M}]_{(x,y)} && (37)\\
\Psi &\;\Rightarrow\; \mathrm{WF}_{(x,y)}(\mathcal{M}_1) \wedge \mathrm{WF}_{(x,y)}(\mathcal{M}_2) && (38)
\end{aligned}$$

Formula (36) asserts that the initial condition of Ψ implies the initial condition of Φ. It follows easily from the definitions of Init_Ψ and Init_Φ.

Roughly speaking, formula (37) asserts that every $\mathcal{N}$ step simulates an $\mathcal{M}$ step, and (38) asserts that Program 2 implements Program 1's fairness conditions. We now sketch the proofs of these two formulas.

8.2.1 Proof of Step-Simulation

Applying rule TLA2 of Figure 5 with true substituted for P and Q shows that (37) follows from

$$[\mathcal{N}]_w \;\Rightarrow\; [\mathcal{M}]_{(x,y)} \tag{39}$$

By definition, $[\mathcal{N}]_w$ equals $\alpha_1 \vee \ldots \vee \gamma_2 \vee (w' = w)$ and $[\mathcal{M}]_{(x,y)}$ equals $\mathcal{M}_1 \vee \mathcal{M}_2 \vee ((x,y)' = (x,y))$. Formula (37) therefore follows from

$$\begin{array}{ll} \alpha_1 \;\Rightarrow\; (x,y)' = (x,y) & \alpha_2 \;\Rightarrow\; (x,y)' = (x,y) \\ \beta_1 \;\Rightarrow\; \mathcal{M}_1 & \beta_2 \;\Rightarrow\; \mathcal{M}_2 \\ \gamma_1 \;\Rightarrow\; (x,y)' = (x,y) & \gamma_2 \;\Rightarrow\; (x,y)' = (x,y) \\ \multicolumn{2}{c}{(w' = w) \;\Rightarrow\; (x,y)' = (x,y)} \end{array} \tag{40}$$

The reader can check that all these implications are trivial consequences of the definitions.

8.2.2 Proof of Fairness

For the fairness condition (38), we sketch the proof that Ψ implies $\mathrm{WF}_{(x,y)}(\mathcal{M}_1)$. The proof that it implies $\mathrm{WF}_{(x,y)}(\mathcal{M}_2)$ is similar.

Strong fairness of Program 2 is necessary to insure that x is incremented infinitely often, so Figure 5 suggests applying SF2 (without the overbars). At first glance, SF2 doesn't seem to work because its conclusion implies a strong fairness condition, and we want to prove $\Psi \Rightarrow \mathrm{WF}_{(x,y)}(\mathcal{M}_1)$. However, if x is a natural number, so $x' \neq x+1$, then *Enabled* $\mathcal{M}_1$ equals true. A simple invariance argument proves $\Phi \Rightarrow \Box(x \in \mathsf{Nat})$, so $\Phi \Rightarrow \Box(\mathit{Enabled}\ \mathcal{M}_1)$. Hence, Φ implies that $\mathrm{SF}_{(x,y)}(\mathcal{M}_1)$ and $\mathrm{WF}_{(x,y)}(\mathcal{M}_1)$ are equivalent—both being equal to $\Box\Diamond\langle\mathcal{M}_1\rangle_{(x,y)}$. It therefore suffices to prove $\Psi \Rightarrow \mathrm{SF}_{(x,y)}(\mathcal{M}_1)$.

Comparing the conclusion of rule SF2 with the formula we are trying to prove apparently leads to the following substitutions in the rule.

$$\mathcal{N} \leftarrow \mathcal{N} \qquad \mathcal{M} \leftarrow \mathcal{M}_1 \qquad f \leftarrow w \qquad g \leftarrow (x,y)$$

However, it turns out that we need to strengthen $\mathcal{N}$ by the use of an invariant. We must find a predicate I (an invariant) that rules out these bad states and satisfies

$$\mathit{Init}_\Psi \wedge \Box[\mathcal{N}]_w \;\Rightarrow\; \Box I \tag{41}$$

By rule INV2, we can then rewrite Ψ as

$$Init_{\Psi} \;\wedge\; \Box[\mathcal{N} \wedge I \wedge I']_w \;\wedge\; \mathrm{SF}_w(\mathcal{N}_1) \;\wedge\; \mathrm{SF}_w(\mathcal{N}_2)$$

and substitute $\mathcal{N} \wedge I \wedge I'$ for $\mathcal{N}$. We will discover the invariant I in the course of the proof.

The first hypothesis of the rule and (40) suggest substituting β_1 for $\mathcal{B}$. The conclusion and the second hypothesis leads to the substitution of $\mathcal{N}_1$ for $\mathcal{A}$ and $\mathrm{SF}_w(\mathcal{N}_2)$ for $\Box F$, using the temporal tautology $\mathrm{SF}_w(\mathcal{N}_2) \equiv \Box\mathrm{SF}_w(\mathcal{N}_2)$. The second and fourth hypotheses lead to the substitution of $pc_1 = \text{“b”}$ for P. With these substitutions, the proof rule becomes

$$\frac{\begin{array}{l}\langle \mathcal{N} \wedge I \wedge I' \wedge \beta_1\rangle_w \;\Rightarrow\; \langle \mathcal{M}_1\rangle_{(x,y)} \\ (pc_1 = \text{“b”}) \wedge (pc_1' = \text{“b”}) \wedge \langle \mathcal{N} \wedge I \wedge I' \wedge \mathcal{N}_1\rangle_w \;\Rightarrow\; \beta_1 \\ (pc_1 = \text{“b”}) \wedge \mathit{Enabled}\ \langle \mathcal{M}_1\rangle_{(x,y)} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{N}_1\rangle_w \\ \Box[\mathcal{N} \wedge I \wedge I' \wedge \neg\beta_1]_w \wedge \mathrm{SF}_w(\mathcal{N}_1) \wedge \mathrm{SF}_w(\mathcal{N}_2) \;\Rightarrow\; \Diamond\Box(pc_1 = \text{“b”})\end{array}}{\Box[\mathcal{N} \wedge I \wedge I']_w \wedge \mathrm{SF}_w(N_1) \wedge \mathrm{SF}_w(\mathcal{N}_2) \;\Rightarrow\; \mathrm{SF}_{(x,y)}(\mathcal{M}_1)}$$

The first three hypotheses are simple action formulas. The second and third follow easily from the definitions of $\mathcal{N}_1$, β_1 and $\mathcal{M}_1$. To prove the first hypothesis, we must show that $\mathcal{N} \wedge I \wedge I' \wedge \beta_1 \wedge (w' \neq w)$ implies $\mathcal{M}_1 \wedge ((x,y)' \neq (x,y))$. As we observed in (40), β_1 implies $\mathcal{M}_1$. Since β_1 also implies $x' = x + 1$, which implies $x' \neq x$ if x is a natural number, the first hypothesis holds if the invariant I implies $x \in \mathsf{Nat}$.

The fourth hypothesis is a temporal formula, which we now examine. To simplify the intuitive reasoning, let us ignore steps that don't change w. The fourth hypothesis then asserts that if every step is an $\mathcal{N} \wedge I \wedge I'$ step that is not a β_1 step, and the fairness conditions hold, then eventually control reaches β_1 and remains there forever. From the informal description of the program in Figure 6, this seems valid. No matter where control starts in process 1, fairness implies that eventually it must reach β_1, where it must remain forever if no β_1 step is performed.

Unfortunately, this intuitive reasoning is wrong. The fourth hypothesis is not a valid TLA formula. For example, consider a behavior that starts in a state with $pc_1 = pc_2 = \text{“a”}$ and $sem = 0$, and that remains in this state forever. In such a behavior, the left-side of the implication in the fourth hypothesis is true, but pc_1 never becomes equal to “b”. Thus, the hypothesis is not satisfied by these behaviors.

The fourth hypothesis is invalid for behaviors starting in “bad” states—ones that are not reachable by executing the program from an initial state satisfying $Init_\Psi$. Such states have to be ruled out by the invariant I. We must substitute $\mathrm{SF}_w(\mathcal{N}_2) \wedge \Box I$ for F and $(pc_1 = \text{“}\mathsf{b}\text{”}) \wedge I$ for P in Rule SF2, obtaining the following as the fourth hypothesis.

$$G \;\Rightarrow\; \Diamond\Box((pc_1 = \text{“}\mathsf{b}\text{”}) \wedge I) \tag{42}$$
$$\text{where } G \;\triangleq\; \Box[\mathcal{N} \wedge I \wedge I' \wedge \neg\beta_1]_w \;\wedge\; \mathrm{SF}_w(\mathcal{N}_1) \;\wedge\; \mathrm{SF}_w(\mathcal{N}_2) \;\wedge\; \Box I$$

Remembering that I must imply $x \in \mathsf{Nat}$, the reader with experience reasoning about concurrent programs will discover that the appropriate invariant is

$$\begin{array}{lll} I & \triangleq & \wedge\; x \in \mathsf{Nat} \\ & & \wedge\; \vee\; (sem = 1) \;\wedge\; (pc_1 = pc_2 = \text{“}\mathsf{a}\text{”}) \\ & & \quad\;\vee\; (sem = 0) \;\wedge\; \vee\; (pc_1 = \text{“}\mathsf{a}\text{”}) \;\wedge\; (pc_2 \in \{\text{“}\mathsf{b}\text{”}, \text{“}\mathsf{g}\text{”}\}) \\ & & \qquad\qquad\qquad\quad\;\vee\; (pc_2 = \text{“}\mathsf{a}\text{”}) \;\wedge\; (pc_1 \in \{\text{“}\mathsf{b}\text{”}, \text{“}\mathsf{g}\text{”}\}) \end{array}$$

(Note the use of “$\vee$” lists to denote disjunctions, and of indentation to eliminate parentheses.) With this definition of I, the invariance property (41) follows easily from Rule INV1.

Having deduced that we need to prove (42), we must understand why it is true. A little thought reveals that (42) holds because control in process 1 must eventually reach β_1, and $\Box[\mathcal{N} \ldots \wedge \neg\beta_1]_w$, which asserts that a β_1 action is never executed, implies that control must then remain there forever. This reasoning is formalized by applying simple temporal reasoning based on the Lattice Rule to derive (42) from:

$$G \;\Rightarrow\; (\,(pc_1 = \text{“}\mathsf{g}\text{”}) \wedge I \leadsto (pc_1 = \text{“}\mathsf{a}\text{”}) \wedge I\,) \tag{43}$$
$$G \;\Rightarrow\; (\,(pc_1 = \text{“}\mathsf{a}\text{”}) \wedge I \leadsto (pc_1 = \text{“}\mathsf{b}\text{”}) \wedge I\,) \tag{44}$$
$$G \;\Rightarrow\; (\,(pc_1 = \text{“}\mathsf{b}\text{”}) \wedge I \Rightarrow \Box((pc_1 = \text{“}\mathsf{b}\text{”}) \wedge I)\,) \tag{45}$$

To see how to prove these formulas, we once again use simple pattern-matching against the proof rules of Figure 5. We find that (43) and (44) should be proved by Rule SF1 with $\mathcal{N}_1$ substituted for $\mathcal{A}$ and $\mathrm{SF}_w(\mathcal{N}_2)$ substituted for $\Box F$, and that (45) should be proved by rule INV1 with $(pc_1 = \text{“}\mathsf{b}\text{”}) \wedge I$ substituted for I. The proofs of (43) and (45) are simple. The proof of (44) is not so easy, the hard part being the proof of the third hypothesis:

$$H \;\Rightarrow\; \Diamond \mathit{Enabled}\,\langle \mathcal{N}_1 \rangle_w \tag{46}$$
$$\begin{array}{lll} \text{where } H & \triangleq & \wedge\; \Box((pc_1 = \text{“}\mathsf{a}\text{”}) \wedge I) \\ & & \wedge\; \Box[\mathcal{N} \wedge I \wedge I' \wedge \neg\beta_1]_w \\ & & \wedge\; \mathrm{SF}_w(\mathcal{N}_2) \end{array}$$

Once again, we have reached a point where blind application of rules fails; we must understand why (46) is true. If pc_1 equals "a", then action $\mathcal{N}_1$ is enabled when control in process 2 is at α_2, and strong fairness for $\mathcal{N}_2$ implies that control must eventually reach α_2. This intuitive reasoning leads us to deduce (46) by temporal reasoning from

$$
\begin{array}{l}
(pc_1 = \text{“a”}) \wedge I \;\; \Rightarrow \;\; (\mathit{Enabled}\ \langle \mathcal{N}_1 \rangle_w \equiv (pc_2 = \text{“a”})) \\
H \;\; \Rightarrow \;\; ((pc_2 = \text{“b”}) \leadsto (pc_2 = \text{“g”})) \\
H \;\; \Rightarrow \;\; ((pc_2 = \text{“g”}) \leadsto (pc_2 = \text{“a”}))
\end{array}
$$

The first formula follows from the observation that

$$
\mathit{Enabled}\ \langle \mathcal{N}_1 \rangle_w \;\; \equiv \;\; \begin{array}{l} \vee \;\; pc_1 = \text{“a”} \;\; \wedge \;\; 0 < \mathit{sem} \\ \vee \;\; pc_1 = \text{“b”} \\ \vee \;\; pc_1 = \text{“g”} \end{array}
$$

Pattern-matching against the proof rules leads to simple proofs of the remaining two formulas by substituting $\mathcal{N}_2$ for $\mathcal{A}$ and true for F in SF1.

8.3 Comments on the Proof

This example illustrates the general method of proving that a lower-level program Ψ implements a higher-level program Φ. There are three things to prove: (i) the initial predicate of Ψ implies the initial predicate of Φ, (ii) a step of Ψ simulates a step of Φ, and (iii) Ψ implies the fairness condition of Φ.

As in the example, proving the initial condition is generally straightforward. Of course, in more realistic examples there will be more details to check.

Because our example was so simple, the proof of step-simulation was atypical. Usually, a step of the lower-level program starting in a completely arbitrary state does not simulate a step of the higher-level program. We must first find the proper invariant, and then apply Rule INV2 to prove step-simulation. Once the invariant is found, the proof is a straightforward exercise in showing that one action implies another. The structure of the formulas tells us how to decompose a large proof into a number of smaller ones.

Our proof of fairness was quite typical in its alternation of blind application of proof rules with the need to understand why a property holds. As in this proof, an invariant is almost always required. Usually, it is the same invariant as in the proof of step-simulation. Of course, the proofs of real algorithms will be more complicated.

Our proof may already have seemed rather complicated for such a simple example, but the example is a bit more subtle than it appears. The reader who attempts a rigorous informal proof will discover that each step in the TLA proof mirrors a step in the informal proof. The more rigorous the informal proof, the more it will resemble the TLA proof. Rules SF1 and SF2 conveniently encapsulate reasoning that occurs over and over again in informal proofs. We believe that temporal logic provides an ideal formalism for translating intuitive understanding of why a liveness property holds into a formal proof.

Were we to choose the weaker fairness requirement $\mathrm{WF}_{(x,y)}(\mathcal{M})$ for Program 1 , then Program 2's fairness condition could be weakened to $\mathrm{WF}_w(\mathcal{N})$. The proof of $\Psi \Rightarrow \Phi$, using WF2 instead of SF2, would then be simpler. Writing out this proof is a good exercise in applying TLA.

8.4 Stuttering and Refinement

Observe that Program 2 is *finer-grained* than Program 1, in the sense that the three atomic operations of each process's loop in Program 2 correspond to a single atomic operation of Program 1. Besides the steps that increment x or y, Program 2 takes steps that modify sem and pc_1 or pc_2, but leave x and y unchanged. Program 2 implements Program 1—that is, the formula $\Psi \Rightarrow \Phi$ is valid—only because Φ allows "stuttering" steps that do not change x and y. Program 2 can in turn be implemented by a still finer-grained program because Ψ allows steps that do not change any of its variables.

Allowing stuttering steps is the key to refining the grain of atomicity. We abandoned the simple RTLA formula Φ of Figure 2 on page 14 because it did not allow stuttering steps.

A temporal formula F is said to be *invariant under stuttering* iff $\sigma[\![F]\!]$ and $\tau[\![F]\!]$ are equivalent whenever σ and τ differ only by stuttering steps. To formalize this definition, we first define $\natural\sigma$ to be the behavior obtained from the behavior σ by removing all stuttering steps—except that if σ ends with infinite stuttering, then those final stuttering steps are kept. The precise definition is:

$$\natural\langle\!\langle s_0, s_1, s_2, \ldots\rangle\!\rangle \;\triangleq\; \begin{array}[t]{l}\textbf{if } \forall \mathsf{n} \in \mathsf{Nat} : s_{\mathsf{n}} = s_0 \\ \quad \textbf{then } \langle\!\langle s_0, s_0, s_0, \ldots\rangle\!\rangle \\ \quad \textbf{else } \begin{array}[t]{l}\textbf{if } s_1 = s_0 \textbf{ then } \natural\langle\!\langle s_1, s_2, s_3, \ldots\rangle\!\rangle \\ \phantom{\textbf{if } s_1 = s_0 } \textbf{ else } \langle\!\langle s_0\rangle\!\rangle \circ \natural\langle\!\langle s_1, s_2, \ldots\rangle\!\rangle\end{array}\end{array} \qquad (47)$$

where $\circ$ denotes concatenation of sequences. A temporal formula F is invariant under stuttering iff $\natural\sigma = \natural\tau$ implies $\sigma[\![F]\!] \equiv \tau[\![F]\!]$, for all behaviors σ and τ. Every TLA formula is invariant under stuttering, but not every RTLA formula is.

9 Hiding Variables

9.1 A Memory Specification

We now consider another example: a simple processor/memory interface. The processor issues *read* and *write* operations that are executed by the memory. The interface consists of three registers, represented by the following three variables.

op: Set by the processor to indicate the desired operation, and reset by the memory after executing the operation.

adr: Set by the processor to indicate the address of the memory location to be read or written.

val: Set by the processor to indicate the value to be written by a *write*, and set by the memory to return the result of a *read*.

Here is a typical behavior, where "—" indicates that the value is irrelevant, and memory location 432 happens to have the initial value 777.

$$
\begin{array}{l}
\langle\!\langle\, op \triangleq \text{“}\mathsf{ready}\text{”},\ adr \triangleq \text{—},\ \ val \triangleq \text{—},\ \ \ldots\ \rangle\!\rangle \\
\langle\!\langle\, op \triangleq \text{“}\mathsf{read}\text{”},\ \ adr \triangleq 432,\ val \triangleq \text{—},\ \ \ldots\ \rangle\!\rangle \\
\langle\!\langle\, op \triangleq \text{“}\mathsf{ready}\text{”},\ adr \triangleq \text{—},\ \ val \triangleq 777,\ \ldots\ \rangle\!\rangle \\
\langle\!\langle\, op \triangleq \text{“}\mathsf{write}\text{”},\ adr \triangleq 196,\ val \triangleq 0,\ \ \ \ \ldots\ \rangle\!\rangle \\
\langle\!\langle\, op \triangleq \text{“}\mathsf{ready}\text{”},\ adr \triangleq \text{—},\ \ val \triangleq \text{—},\ \ \ldots\ \rangle\!\rangle \\
\quad \vdots
\end{array}
$$

It is easy to specify this interface if we introduce an additional variable *memory* to denote the contents of memory, so $memory(\mathsf{n})$ is the current value of memory location n. The property Φ describing the desired behaviors is shown in Figure 8, where $\mathsf{Address}$ is the set of legal addresses, and MemVal is the set of possible memory values. Action $\mathcal{S}(m, v)$ represents the assign-
Note 13 ment $memory(m) := v$. Actions $\mathcal{R}_{proc}$ and $\mathcal{W}_{proc}$ represent the processor's *read*- and *write*-request operations; actions $\mathcal{R}_{mem}$ and $\mathcal{W}_{mem}$ represent the

$$
\begin{array}{lcl}
Init_{\Phi} & \triangleq & \land\ op = \text{“}\mathsf{ready}\text{”} \\
 & & \land\ \forall \mathsf{n} \in \mathsf{Address} : memory(\mathsf{n}) \in \mathsf{MemVal}
\end{array}
$$

$$
\mathcal{S}(m, v) \ \triangleq\ \forall \mathsf{n} \in \mathsf{Address} : \begin{array}{l} \land\ (\mathsf{n} = m) \Rightarrow (memory(\mathsf{n})' = v) \\ \land\ (\mathsf{n} \neq m) \Rightarrow (memory(\mathsf{n})' = memory(\mathsf{n})) \end{array}
$$

$$
\begin{array}{lcl}
\mathcal{R}_{proc} & \triangleq & \land\ op = \text{“}\mathsf{ready}\text{”} \\
 & & \land\ op' = \text{“}\mathsf{read}\text{”} \\
 & & \land\ adr' \in \mathsf{Address} \\
 & & \land\ memory' = memory
\end{array}
\qquad
\begin{array}{lcl}
\mathcal{R}_{mem} & \triangleq & \land\ op = \text{“}\mathsf{read}\text{”} \\
 & & \land\ op' = \text{“}\mathsf{ready}\text{”} \\
 & & \land\ val' = memory(adr) \\
 & & \land\ memory' = memory
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{W}_{proc} & \triangleq & \land\ op = \text{“}\mathsf{ready}\text{”} \\
 & & \land\ op' = \text{“}\mathsf{write}\text{”} \\
 & & \land\ adr' \in \mathsf{Address} \\
 & & \land\ val' \in \mathsf{MemVal} \\
 & & \land\ memory' = memory
\end{array}
\qquad
\begin{array}{lcl}
\mathcal{W}_{mem} & \triangleq & \land\ op = \text{“}\mathsf{write}\text{”} \\
 & & \land\ op' = \text{“}\mathsf{ready}\text{”} \\
 & & \land\ \mathcal{S}(adr, val)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{N}_{mem} & \triangleq & \mathcal{R}_{mem} \lor \mathcal{W}_{mem} \\
\mathcal{N} & \triangleq & \mathcal{N}_{mem} \lor \mathcal{R}_{proc} \lor \mathcal{W}_{proc} \\
w & \triangleq & (op, adr, val, memory) \\
\Phi & \triangleq & Init_{\Phi} \ \land\ \Box[\mathcal{N}]_w \ \land\ \mathrm{WF}_w(\mathcal{N}_{mem})
\end{array}
$$

Figure 8: “Internal” specification of a processor/memory interface.

memory's responses to those requests. Action $\mathcal{N}_{mem}$ denotes the memory's next-state relation. The fairness condition $\mathrm{WF}_w(\mathcal{N}_{mem})$ asserts that the memory eventually responds to each request; there is no requirement that the processor ever issues requests.

Observe that the action $\mathcal{S}(m, v)$ is used only to define $\mathcal{W}_{mem}$; it was introduced just to keep the definition of $\mathcal{W}_{mem}$ from running off the page. There is no formal significance to our choice of names such as $\mathcal{R}_{proc}$. Our decision to define $\mathcal{N}_{mem}$ as the disjunction of two simpler actions was completely arbitrary; we could just as well have defined it all at once, or as the disjunction of more than two actions. There are countless ways of writing logically equivalent formulas Φ.

The formula Φ specifies the right behavior for the interface variables *op*, *adr*, and *val*. However, it also specifies the value of the variable *memory*, which we did not want to specify. We want to specify only how the three interface variables change; we do not care how any other variables such as *x*, *sem*, or *memory* change. We therefore want a formula asserting that *op*, *adr*, and *val* behave as described by Φ, but that it doesn't matter what values *memory* assumes. Such a formula is sometimes described as Φ with the variable *memory* "hidden". This formula is written $\exists\ memory : \Phi$.

The precise meaning of the formula $\exists\ memory : \Phi$ is defined below. Here, we simply want to observe that the free (program) variables of this formula are *op*, *adr*, and *val*. Since x, *sem*, and *memory* do not occur free, the formula does not constrain them in any way.

9.2 Quantification over Program Variables

We now define $\exists\ x : F$, where x is a (program) variable and F a temporal formula. Intuitively, $\exists\ x : F$ asserts that it doesn't matter what the actual values of x are, but that there are some values x can assume for which F holds. For example, $\exists\ x : \Box[y = x']_{(x,y)}$ is satisfied by the behavior

$$
\begin{array}{llll}
(\!(x \triangleq \text{“}\mathsf{a}\text{”}, & y \triangleq 0, & z \triangleq \mathsf{true}, & \ldots)\!) \\
(\!(x \triangleq \text{“}\mathsf{b}\text{”}, & y \triangleq 1, & z \triangleq -13, & \ldots)\!) \\
(\!(x \triangleq \text{“}\mathsf{c}\text{”}, & y \triangleq 1, & z \triangleq -13, & \ldots)\!) \\
(\!(x \triangleq 77, & y \triangleq 2, & z \triangleq \mathsf{true}, & \ldots)\!) \\
\quad\vdots
\end{array}
$$

because by changing only the values of x, we get the following behavior that satisfies $\Box[y = x']_{(x,y)}$.

$$
\begin{array}{llll}
(\!(x \triangleq \text{“}\mathsf{a}\text{”}, & y \triangleq 0, & z \triangleq \mathsf{true}, & \ldots)\!) \\
(\!(x \triangleq 0, & y \triangleq 1, & z \triangleq -13, & \ldots)\!) \\
(\!(x \triangleq 1, & y \triangleq 1, & z \triangleq -13, & \ldots)\!) \\
(\!(x \triangleq 1, & y \triangleq 2, & z \triangleq \mathsf{true}, & \ldots)\!) \\
\quad\vdots
\end{array}
$$

In fact, every behavior satisfies $\exists\ x : \Box[y = x']_{(x,y)}$.

To define $\exists\ x : F$ formally, we need some auxiliary definitions. For any variable x and states s and t, let $s =_x t$ mean that s and t assign the same values to all variables other than x. More precisely,

$$s =_x t \quad \triangleq \quad \forall\ `v' \neq `x' : s[\![v]\!] = t[\![v]\!]$$

We extend the relation $=_x$ to behaviors in the obvious way:

$$\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle =_x \langle\!\langle t_0, t_1, \ldots \rangle\!\rangle \quad \triangleq \quad \forall\ \mathsf{n} \in \mathsf{Nat} : s_{\mathsf{n}} =_x t_{\mathsf{n}}$$

The obvious next step is to define

$$\sigma[\![\exists\, x : F]\!] \;\triangleq\; \exists\, \tau \in \mathbf{St}^{\infty} : (\sigma =_x \tau) \wedge \tau[\![F]\!] \tag{48}$$

for any behavior σ. (Recall that $\mathbf{St}^{\infty}$ is the set of all behaviors.) However, this definition is not quite right, because the formula it defines is not necessarily invariant under stuttering. For example, suppose F is satisfied only by behaviors in which x changes before y does, including the behavior

$$\begin{array}{l}
(\!(x \triangleq 1,\; y \triangleq \text{“a”},\; z \triangleq \mathsf{true},\; \ldots)\!) \\
(\!(x \triangleq 2,\; y \triangleq \text{“a”},\; z \triangleq \mathsf{true},\; \ldots)\!) \\
(\!(x \triangleq 2,\; y \triangleq \text{“b”},\; z \triangleq \mathsf{true},\; \ldots)\!) \\
\quad\vdots
\end{array}$$

Then definition (48) implies that the behavior

$$\begin{array}{l}
(\!(x \triangleq 999,\; y \triangleq \text{“a”},\; z \triangleq \mathsf{true},\; \ldots)\!) \\
(\!(x \triangleq 999,\; y \triangleq \text{“a”},\; z \triangleq \mathsf{true},\; \ldots)\!) \\
(\!(x \triangleq 999,\; y \triangleq \text{“b”},\; z \triangleq \mathsf{true},\; \ldots)\!) \\
\quad\vdots
\end{array}$$

satisfies $\exists\, x : F$ (because we can produce a behavior satisfying F by changing only the values of x). However, the behavior

$$\begin{array}{l}
(\!(x \triangleq 999,\; y \triangleq \text{“a”},\; z \triangleq \mathsf{true},\; \ldots)\!) \\
(\!(x \triangleq 999,\; y \triangleq \text{“b”},\; z \triangleq \mathsf{true},\; \ldots)\!) \\
\quad\vdots
\end{array}$$

does not satisfy $\exists\, x : F$ (because of the assumption that F requires x to change before y does). With appropriate values for all other variables, these two behaviors differ only by stuttering steps. Hence, with definition (48), $\exists\, x : F$ is not necessarily invariant under stuttering even though F is.

To obtain invariance under stuttering, we must define $\exists\, x : F$ to be satisfied by a behavior σ iff we can obtain a behavior that satisfies F by first adding stuttering and then changing the values of x. The appropriate definition is

$$\sigma[\![\exists\, x : F]\!] \;\triangleq\; \exists\, \rho, \tau \in \mathbf{St}^{\infty} : (\natural\sigma = \natural\rho) \wedge (\rho =_x \tau) \wedge \tau[\![F]\!] \tag{49}$$

The operator $\exists\, x$ differs from ordinary existential quantification because it asserts the existence not of a single value to be substituted for x, but of an infinite sequence of values. However, it really is existential quantification because it obeys the ordinary laws of existential quantification. In particular, the usual rules E1 and E2 of Figure 9 on the next page are sound. From these rules, one can deduce the expected properties of existential quantification, such as

$$(\exists\, x : F \vee G) \;\equiv\; (\exists\, x : F) \vee (\exists\, x : G)$$

We can extend TLA to allow quantification over rigid variables as well as program variables. Since the value of a rigid variable is constant throughout a behavior, quantification over rigid variables is much simpler than quantifi-
Note 14 cation over program variables. However, it is of less use. The semantics of quantification over rigid variables is defined in Figure 9.

General TLA formulas consist of all formulas obtained from simple TLA formulas by logical operators and quantification over program and rigid variables. The syntax and semantics of quantification are summarized in Figure 9 on the next page, which together with Figure 4 on page 21 gives the complete definition of TLA.

9.3 Refinement Mappings

9.3.1 Implementing The Memory Specification

We now give a simple implementation of the processor/memory interface specified by the formula $\exists\, memory : \Phi$, where Φ is defined in Figure 8. The implementation uses a main memory and a cache, represented by variables *main* and *cache*. The value of $cache(\mathsf{m})$ represents the cache's value for memory location m, the special value $\perp$ (assumed not to be in MemVal) denoting that this memory location is not in the cache. The processor's read and write requests are serviced from the cache, and separate internal actions (not visible from the interface) move values between the cache and main memory. When the processor reads a value not in the cache, the value is first moved into the cache and then put in *val*.

The "internal" description, in which *main* and *cache* are free variables, is the formula Ψ of Figure 10 on page 46. The actions defined in the figure have the following interpretations.

$\mathcal{T}(a, m, v)$ Represents the assignment $a(m) := v$. This action is introduced only to simplify the definitions of other actions.

$\mathcal{R}_{pro}$, $\mathcal{W}_{pro}$ The processor's *read*- and *write*-request operations.

Syntax

$$
\begin{array}{lcl}
\langle \mathit{general\ formula} \rangle & \triangleq & \langle \mathit{formula} \rangle \mid \exists\, \langle \mathit{variable} \rangle : \langle \mathit{general\ formula} \rangle \\
& & \mid \exists\, \langle \mathit{rigid\ variable} \rangle : \langle \mathit{general\ formula} \rangle \\
& & \mid \langle \mathit{general\ formula} \rangle \wedge \langle \mathit{general\ formula} \rangle \\
& & \mid \neg \langle \mathit{general\ formula} \rangle \\
\langle \mathit{formula} \rangle & \triangleq & \text{a simple TLA formula (see Figure 4)}
\end{array}
$$

Semantics

$$\langle\!\langle s_0, s_1, \ldots \rangle\!\rangle =_x \langle\!\langle t_0, t_1, \ldots \rangle\!\rangle \;\triangleq\; \forall\, \mathsf{n} \in \mathsf{Nat} : \forall\, `v' \neq `x' : s_{\mathsf{n}}[\![v]\!] = t_{\mathsf{n}}[\![v]\!]$$

$$
\begin{array}{lll}
\natural\langle\!\langle s_0, s_1, s_2, \ldots \rangle\!\rangle & \triangleq & \textbf{if } \forall \mathsf{n} \in \mathsf{Nat} : s_{\mathsf{n}} = s_0 \\
& & \quad \textbf{then } \langle\!\langle s_0, s_0, s_0, \ldots \rangle\!\rangle \\
& & \quad \textbf{else } \; \textbf{if } s_1 = s_0 \textbf{ then } \natural\langle\!\langle s_1, s_2, s_3, \ldots \rangle\!\rangle \\
& & \qquad\qquad\qquad\qquad \textbf{else } \; \langle\!\langle s_0 \rangle\!\rangle \circ \natural\langle\!\langle s_1, s_2, \ldots \rangle\!\rangle
\end{array}
$$

$$\sigma[\![\exists\, x : F]\!] \;\triangleq\; \exists\, \rho, \tau \in \mathbf{St}^{\infty} : (\natural\sigma = \natural\rho) \wedge (\rho =_x \tau) \wedge \tau[\![F]\!]$$

$$\sigma[\![\exists\, \mathsf{c} : F]\!] \;\triangleq\; \exists \mathsf{c} \in \mathsf{Val} : \sigma[\![F]\!]$$

Proof Rules

$E1.\ \ \vdash F(f/x) \Rightarrow \exists\, x : F$

$E2.$ $$\frac{F \Rightarrow G \qquad x \text{ does not occur free in } G}{(\exists\, x : F) \Rightarrow G}$$

$F1.\ \ \vdash F(\mathsf{e}/\mathsf{c}) \Rightarrow \exists\, \mathsf{c} : F$

$F2.$ $$\frac{F \Rightarrow G \qquad \mathsf{c} \text{ does not occur free in } G}{(\exists\, \mathsf{c} : F) \Rightarrow G}$$

where x is a $\langle \mathit{variable} \rangle$ — F, G are $\langle \mathit{general\ formula} \rangle$s
f is a state function — s, s_0, t_0, s_1, t_1, $\ldots$ are states
c is a $\langle \mathit{rigid\ variable} \rangle$ — σ is a behavior
e is a constant expression

Figure 9: Quantification in TLA.

$$
\begin{array}{lcl}
Init_{\Psi} & \triangleq & \land\ op = \text{"ready"} \\
& & \land\ \forall \mathsf{n} \in \mathsf{Address} : (main(\mathsf{n}) \in \mathsf{MemVal}) \land (cache(\mathsf{n}) = \bot)
\end{array}
$$

$$
\mathcal{T}(a, m, v) \triangleq \forall \mathsf{n} \in \mathsf{Address} : \begin{array}{l} \land\ (\mathsf{n} = m) \Rightarrow (a'(\mathsf{n}) = v) \\ \land\ (\mathsf{n} \neq m) \Rightarrow (a'(\mathsf{n}) = a(\mathsf{n})) \end{array}
$$

$$
\mathcal{R}_{pro} \triangleq \begin{array}{l} \land\ op = \text{"ready"} \\ \land\ op' = \text{"read"} \\ \land\ adr' \in \mathsf{Address} \\ \land\ Unchanged\ (main, cache) \end{array}
\qquad
\mathcal{R}_{cch} \triangleq \begin{array}{l} \land\ op = \text{"read"} \\ \land\ cache(adr) \neq \bot \\ \land\ op' = \text{"ready"} \\ \land\ val' = cache(adr) \\ \land\ Unchanged\ (main, cache) \end{array}
$$

$$
\mathcal{W}_{pro} \triangleq \begin{array}{l} \land\ op = \text{"ready"} \\ \land\ op' = \text{"write"} \\ \land\ adr' \in \mathsf{Address} \\ \land\ val' \in \mathsf{MemVal} \\ \land\ Unchanged\ (main, cache) \end{array}
\qquad
\mathcal{W}_{cch} \triangleq \begin{array}{l} \land\ op = \text{"write"} \\ \land\ op' = \text{"ready"} \\ \land\ \mathcal{T}(cache, adr, val) \\ \land\ Unchanged\ main \end{array}
$$

$$
\mathcal{C}_{get}(m) \triangleq \begin{array}{l} \land\ cache(m) = \bot \\ \land\ \mathcal{T}(cache, m, main(m)) \\ \land\ Unchanged\ (op, adr, \\ \qquad\qquad\qquad val, main) \end{array}
\qquad
\mathcal{C}_{fl}(m) \triangleq \begin{array}{l} \land\ cache(m) \neq \bot \\ \land\ \begin{array}{l} \lor\ op \neq \text{"read"} \\ \lor\ m \neq adr \end{array} \\ \land\ \mathcal{T}(main, m, cache(m)) \\ \land\ \mathcal{T}(cache, m, \bot) \\ \land\ Unchanged\ (op, adr, val) \end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P} & \triangleq & \mathcal{R}_{pro} \lor \mathcal{W}_{pro} \lor \mathcal{R}_{cch} \lor \mathcal{W}_{cch} \lor (\exists\, \mathsf{m} \in \mathsf{Address} : \mathcal{C}_{get}(\mathsf{m}) \lor \mathcal{C}_{fl}(\mathsf{m})) \\
\mathcal{F} & \triangleq & \mathcal{R}_{cch} \lor \mathcal{W}_{cch} \lor (\mathcal{C}_{get}(adr) \land (op = \text{"read"})) \\
u & \triangleq & (op, adr, val, main, cache) \\
\Psi & \triangleq & Init_{\Psi} \land \Box[\mathcal{P}]_u \land \mathrm{WF}_u(\mathcal{F})
\end{array}
$$

Figure 10: A simple cached memory.

$\mathcal{R}_{cch}$, $\mathcal{W}_{cch}$ The memory's responses to processor requests, the value being read from or written to the cache. An $\mathcal{R}_{cch}$ action can be executed only if the value to be read is in the cache.

$\mathcal{C}_{get}(m)$, $\mathcal{C}_{fl}(m)$ The internal actions of moving a value from main memory to the cache, and of flushing a value from the cache to main memory. The second conjunct of $\mathcal{C}_{fl}(m)$ prevents a value from being flushed while it is being read. This is the only constraint on when values can be moved into or out of the cache; no particular cache maintenance policy is specified.

$\mathcal{P}$ The next-state relation, which is the disjunction of all possible actions of the processor and the memory.

$\mathcal{F}$ The disjunction of all the memory actions that must be performed to respond to a processor request. The third disjunct represents the action of moving the value for a *read* request from main memory into the cache. (It is enabled only if the value is not already in the cache.)

If we consider *main* and *cache* to be internal variables, then the cached memory is described by the TLA formula[5] $\exists\, main, cache : \Psi$. The assertion that the cached memory correctly implements the processor/memory interface is expressed by the formula

$$(\exists\, main, cache : \Psi) \;\Rightarrow\; (\exists\, memory : \Phi) \tag{50}$$

To prove (50), we define the state function $\overline{memory}$ by

$$\overline{memory}(m) \;\stackrel{\Delta}{=}\; \textbf{if } cache(m) = \bot \textbf{ then } main(m) \textbf{ else } cache(m)$$

and then prove $\Psi \Rightarrow \overline{\Phi}$, where $\overline{\Phi}$ denotes the formula $\Phi(\overline{memory}/memory)$ obtained by substituting $\overline{memory}$ for all free occurrences of $memory$ in Φ. Applying rule E1 of Figure 9, substituting $\overline{memory}$ for f and $memory$ for x, we obtain $\Psi \;\Rightarrow\; \exists\, memory : \Phi$. Rule E2 then yields (50).

The formula $\Psi \Rightarrow \overline{\Phi}$ asserts that any sequence of values for the variables *op*, *adr*, and *val*, and for the state function $\overline{memory}$, that is allowed by Ψ is a sequence of values that Φ allows for the variables *op*, *adr*, *val*, and *memory*. We can regard $\overline{memory}$ as the "concrete" state function with which Ψ implements the "abstract" variable *memory*.

[5] As usual in logic, we write $\exists\, x, y : F$ as an abbreviation for $\exists\, x : \exists\, y : F$, which by E1 and E2 of Figure 9 is equivalent to $\exists\, y : \exists\, x : F$.

How do we prove that Ψ implies $\overline{\Phi}$? To find the answer, we examine the structure of $\overline{\Phi}$. For any formula F, let $\overline{F}$ denote the formula $F(\overline{memory}/memory)$ obtained by substituting $\overline{memory}$ for all free occurrences of *memory* in F. For example, $\overline{w}$ is the state function $(op, adr, val, \overline{memory})$. Then $\overline{\Phi}$ equals $\overline{Init_\Phi} \wedge \Box[\overline{\mathcal{N}}]_{\overline{w}} \wedge \overline{\mathrm{WF}_w(\mathcal{N}_{mem})}$. The formula $\overline{\Phi}$ therefore looks much like an ordinary TLA formula representing a program, with initial condition $\overline{Init_\Phi}$ and next-state relation $\overline{\mathcal{N}}$. The only difference is that instead of an ordinary weak fairness condition, $\overline{\Phi}$ has as a conjunct the "barred" fairness condition $\overline{\mathrm{WF}_w(\mathcal{N}_{mem})}$.

The proof of $\Psi \Rightarrow \overline{\Phi}$ is similar to the proof in Section 8.2 that Program 2 implements Program 1. We first prove that $Init_\Psi$ implies $\overline{Init_\Phi}$. We next prove that Ψ implies $\Box[\overline{\mathcal{N}}]_{\overline{w}}$ (step-simulation) by applying rule TLA2 of Figure 5 (page 22) with the substitutions

$$\mathcal{A} \leftarrow \mathcal{P} \qquad \mathcal{B} \leftarrow \overline{\mathcal{N}} \qquad f \leftarrow u \qquad g \leftarrow \overline{w} \qquad P \leftarrow \mathsf{true} \qquad Q \leftarrow \mathsf{true}$$

Finally, we prove that Ψ implies $\overline{\mathrm{WF}_w(\mathcal{N}_{mem})}$ (fairness) by applying WF2 with the substitutions

$$\begin{array}{lll} \mathcal{M} \leftarrow \mathcal{N} & \mathcal{A} \leftarrow \mathcal{F} & f \leftarrow u \\ \mathcal{N} \leftarrow \mathcal{P} & \mathcal{B} \leftarrow \mathcal{R}_{cch} \vee \mathcal{W}_{cch} & g \leftarrow \overline{w} \\ \multicolumn{3}{l}{P \leftarrow (op = \text{"}\mathsf{write}\text{"}) \vee (op = \text{"}\mathsf{read}\text{"} \wedge cache(adr) \neq \perp)} \end{array}$$

(Observe that Rule WF2 has the appropriate "bars" to prove the desired conclusion.) As in our previous example, the proofs consist of straightforward calculations punctuated by the occasional need for insight into why what we are trying to prove is true.

This cached memory is quite abstract; it allows any policy for deciding when to move values between the cache and main memory. Given a particular caching algorithm, we would prove that it implements the simple cached memory—meaning that the TLA formula representing the algorithm implies $\exists\, main, cache : \Psi$. By the transitivity of implication, this proves that the algorithm implements the memory/processor interface.

9.3.2 Refinement Mappings

It is clear how to generalize the example above to the problem of proving

$$(\exists\, x_1, \ldots, x_m : \Psi) \;\; \Rightarrow \;\; (\exists\, y_1, \ldots, y_n : \Phi) \qquad (51)$$

for arbitrary Ψ and Φ. We must define state functions $\overline{y_1}, \ldots, \overline{y_n}$ in terms of the variables that occur in Ψ and prove $\Psi \Rightarrow \overline{\Phi}$, where $\overline{\Phi}$ denotes the formula

$\Phi(\overline{y_1}/y_1, \ldots, \overline{y_n}/y_n)$ obtained by substituting $\overline{y_i}$ for the free occurrences of y_i in Φ, for all i. We then infer (51) from rules E1 and E2.

The collection of state functions $\overline{y_1}$, ..., $\overline{y_n}$ is called a *refinement mapping.* The "barred variable" $\overline{y_i}$ is the state function with which Ψ implements the variable y_i of Φ.

To prove (51), one must find a refinement mapping such that $\Psi \Rightarrow \overline{\Phi}$ is valid. The completeness theorem then implies that the proof of $\Psi \Rightarrow \overline{\Phi}$ can be reduced to the proof of valid action formulas by using the rules of Figure 5. But can the requisite refinement mapping always be found? Does the validity of (51) imply the existence of a refinement mapping such that $\Psi \Rightarrow \overline{\Phi}$ is valid?

The answer is no; a refinement mapping need not exist. As an example, we return to Programs 1 and 2, represented by formulas Φ of Figure 3 on page 18 and Ψ of Figure 7 on page 32. Program 2 permits precisely the same sequences of values for x and y as does Program 1. Therefore, the formula $\exists\, sem, pc_1, pc_2 : \Psi$, which describes only the sequences of values for x and y allowed by Program 2, is equivalent to Φ. Can we prove this equivalence?

We already sketched the proof of $\Psi \Rightarrow \Phi$, which by Rule E2 implies $(\exists\, sem, pc_1, pc_2 : \Psi) \Rightarrow \Phi$. In this case, Φ has no internal variables, so the refinement mapping is the trivial one consisting of the empty set of barred variables. Now consider the converse,

$$\Phi \quad \Rightarrow \quad (\exists\, sem, pc_1, pc_2 : \Psi) \tag{52}$$

Can we define the requisite state functions $\overline{sem}$, $\overline{pc_1}$, and $\overline{pc_2}$ in terms of x and y (the only variables that occur in Φ) so that Program 1 allows them to assume only those sequences of values that Program 2 allows the corresponding variables to assume? Clearly not. There is no way to infer from the values of x and y what the values of sem, pc_1, and pc_2 should be. Thus, there does not exist a refinement mapping for which Φ implies $\overline{\Psi}$.

To prove (52), one must modify Φ by adding *dummy variables.* Intuitively, a dummy variable is one that is added to a program without affecting the program's behavior. Formally, adding a dummy variable d to a formula Π means finding a formula Π^d such that $\exists\, d : \Pi^d$ is equivalent to Π. (The variable d is assumed not to occur free in Π.) Formula (52) can be proved by adding two dummy variables h and p to Φ. That is, we can construct a formula Φ^{hp} such that $\exists\, h, p : \Phi^{hp}$ is equivalent to Φ, and can then prove

$$(\exists\, h, p : \Phi^{hp}) \quad \Rightarrow \quad (\exists\, sem, pc_1, pc_2 : \Psi)$$

by constructing a refinement mapping such that Φ^{hp} implies $\overline{\Psi}$. The refinement mapping can be found because the state functions $\overline{sem}$, $\overline{pc_1}$, and $\overline{pc_2}$ are allowed to depend upon h and p as well as x and y.

In general, refinement mappings can always be found if we add the right dummy variables. The completeness theorem of [1] shows that, under certain reasonable assumptions about Ψ and Φ, if (51) is valid, then it can in principle be proved by adding dummy variables to Ψ and finding the requisite refinement mapping. This theorem and the completeness theorem provide a relative completeness result for TLA formulas of the form (51) when Φ and Ψ are formulas of the form $Init \wedge \Box[\mathcal{N}]_f \wedge F$, with F the conjunction of fairness conditions.

9.3.3 "Barring" Fairness

When Φ has the canonical form $Init \wedge \Box[\mathcal{N}]_f \wedge F$, the formula $\overline{\Phi}$ equals $\overline{Init} \wedge \Box[\overline{\mathcal{N}}]_{\overline{f}} \wedge \overline{F}$. If F is the conjunction of fairness conditions of the form $\mathrm{WF}_g(\mathcal{M})$ and $\mathrm{SF}_g(\mathcal{M})$, then $\overline{F}$ is the conjunction of "barred" fairness conditions $\overline{\mathrm{WF}_g(\mathcal{M})}$ and $\overline{\mathrm{SF}_g(\mathcal{M})}$.

We might expect that $\overline{\mathrm{WF}_g(\mathcal{M})}$ would be equivalent to $\mathrm{WF}_{\overline{g}}(\overline{\mathcal{M}})$ and $\overline{\mathrm{SF}_g(\mathcal{M})}$ equivalent to $\mathrm{SF}_{\overline{g}}(\overline{\mathcal{M}})$, but that need not be the case. It is true that

$$\begin{aligned} \overline{\mathrm{WF}_g(\mathcal{M})} &\equiv \Box\Diamond\neg\overline{\mathit{Enabled}\ \langle\mathcal{M}\rangle_g} \vee \Box\Diamond\langle\overline{\mathcal{M}}\rangle_{\overline{g}} \\ \overline{\mathrm{SF}_g(\mathcal{M})} &\equiv \Box\Diamond\neg\overline{\mathit{Enabled}\ \langle\mathcal{M}\rangle_g} \vee \Diamond\Box\langle\overline{\mathcal{M}}\rangle_{\overline{g}} \end{aligned} \tag{53}$$

However, $\overline{\mathit{Enabled}\ \langle\mathcal{M}\rangle_g}$ is not necessarily equivalent to $\mathit{Enabled}\ \langle\overline{\mathcal{M}}\rangle_{\overline{g}}$. For example, let $\mathcal{M}$ be the action $(x' = x) \wedge (y' \neq y)$, let g equal (x, y), and let the refinement mapping be defined by $\overline{x} = z$ and $\overline{y} = z$. Then $\mathit{Enabled}\ \langle\mathcal{M}\rangle_g$ equals true, so $\overline{\mathit{Enabled}\ \langle\mathcal{M}\rangle_g}$ equals true. But

$$\begin{aligned} &\mathit{Enabled}\ \langle\overline{\mathcal{M}}\rangle_{\overline{g}} \\ &\quad\equiv \mathit{Enabled}\ \langle\overline{(x' = x) \wedge (y' \neq y)}\rangle_{\overline{(x,y)}} && \{\text{by definition of } \mathcal{M} \text{ and } g\} \\ &\quad\equiv \mathit{Enabled}\ \langle(\overline{x}' = \overline{x}) \wedge (\overline{y}' \neq \overline{y})\rangle_{(\overline{x},\overline{y})} && \{\text{by definition of } \overline{\ldots}\} \\ &\quad\equiv \mathit{Enabled}\ \langle(z' = z) \wedge (z' \neq z)\rangle_{(z,z)} && \{\text{by definition of } \overline{x} \text{ and } \overline{y}\} \\ &\quad\equiv \mathit{Enabled}\ \mathsf{false} && \{\text{by definition of } \langle\ldots\rangle_{\ldots}\} \\ &\quad\equiv \mathsf{false} && \{\text{by definition of } \mathit{Enabled}\,\} \end{aligned}$$

Thus, $\overline{\mathit{Enabled}\ \langle\mathcal{M}\rangle_g}$ is not equivalent to $\mathit{Enabled}\ \langle\overline{\mathcal{M}}\rangle_{\overline{g}}$. In general, the primed variables in the action $\langle\mathcal{M}\rangle_g$ are not free variables of the expression

Enabled $\langle \mathcal{M} \rangle_g$, so we can't obtain $\overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g}$ from *Enabled* $\langle \mathcal{M} \rangle_g$ by blindly barring all variables.

In rules WF2 and SF2, the formulas $\overline{\mathrm{WF}_g(\mathcal{M})}$ and $\overline{\mathrm{SF}_g(\mathcal{M})}$ are defined by (53). The rules are sound when $\overline{\mathcal{M}}$ is any action, $\overline{g}$ any state function, and $\overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g}$ any predicate—assuming that $\overline{\mathrm{WF}_g(\mathcal{M})}$ and $\overline{\mathrm{SF}_g(\mathcal{M})}$ are defined by (53). In practice, the barred formulas will be obtained from unbarred ones by substituting "barred variables" (state functions) for variables, as in our example.

10 Further Comments

10.1 Mechanical Verification

Because it is a simple logic, TLA is ideally suited for mechanization. Urban Engberg and Peter Grønning have been working on the mechanical verification of TLA, using LP—an "off-the-shelf" verification system based on rewriting [11]. Although initial experiments showed that LP can be used directly, it was decided to write a preprocessor. In addition to allowing more readable specifications, the preprocessor allows separate LP proofs for action formulas and temporal formulas, using simpler encodings of the formulas than would be possible with a single proof. Since most reasoning in a TLA proof is about actions, a simple encoding of action formulas is important.

The proof in Section 8.2, that the formula Ψ describing Program 2 implies the formula Φ describing Program 1, has been checked with LP. Figure 11 shows the definitions of Φ and Ψ in the actual preprocessor input. (Declarations and preprocessor directives have been omitted.) Observe that these definitions are almost perfect transliterations of the ones in Figures 3 and 7. The major differences are the use of "`*`" to represent tuples and "`<<`" instead of "$<$"—differences introduced because comma and "`<`" have other meanings.

Following these definitions is the preprocessor directive

```
Prove Temp  Psi  =>  Phi
```

that creates an LP theorem whose proof implies the validity of the temporal formula $\Psi \Rightarrow \Phi$. (The directive `Prove Act` is used for action formulas.) The rest of the preprocessor input is a proof of this theorem, consisting of a sequence of `Prove` directives and LP commands to prove the corresponding assertions. The only part of the proof that was not checked by LP is

```
InitPhi ==  (x = 0) /\ (y = 0)
M1      ==  (x' = x + 1) /\ (y' = y)
M2      ==  (y' = y + 1) /\ (x' = x)
M       ==  M1 \/ M2
v       ==  (x * y)
Phi     ==  InitPhi /\ [][M]_v /\ WF(v,M1) /\ WF(v,M2)

InitPsi  ==  /\ (pc1 = a) /\ (pc2 = a)
             /\ (x = 0) /\ (y = 0)
             /\ sem = 1
alpha1   ==  /\ (pc1 = a) /\ (0 << sem)
             /\ pc1' = b
             /\ sem' = sem - 1
             /\ Unchanged(x * y * pc2)

           ⋮

gamma2   ==  /\ pc2 = g
             /\ pc2' = a
             /\ sem' = sem + 1
             /\ Unchanged(x * y * pc1)
N1       ==  alpha1 \/ beta1 \/ gamma1
N2       ==  alpha2 \/ beta2 \/ gamma2
N        ==  N1 \/ N2
w        ==  (x * y * pc1 * pc2 * sem)
Psi      ==  InitPsi /\ [][N]_w /\ SF(w,N1) /\ SF(w,N2)
```

Figure 11: The representation of the formulas Φ and Ψ of Figures 3 and 7 for the mechanical verification of the theorem $\Psi \Rightarrow \Phi$.

the computation of the *Enabled* predicates. Although algorithmically simple, these computations are awkward to do in LP. A future version of the preprocessor will compute *Enabled* predicates.

The work on mechanically verifying TLA formulas with LP is preliminary. So far, only very simple examples have been attempted. The preprocessor is a prototype, representing about two man-months of effort. The goal of the project is to assess the feasibility of implementing a verification system that will be useful for real problems.

10.2 TLA versus Programming Languages

Let us compare Figure 6, the description of Program 2 in a conventional programming language, with Figure 7, its representation as a TLA formula. At first glance, the program looks simpler than the TLA formula. However, the program seems simple only because you are already familiar with its notation. To understand what the program means, you need to understand the meaning of the **var** declarations, the **cobegin**, **loop**, ";", and ":=" constructs, and the P and V operations. In contrast, everything needed to understand the TLA formula appears in Figure 4. It is easy to make something seem simple by omitting the complicated definitions needed to understand it.

One reason for the conventional program's apparent simplicity is that it does not specify liveness properties. Nothing in Figure 6 told us that the fairness condition for Ψ should be strong fairness ($\mathrm{SF}_w(\mathcal{M}_1) \wedge \mathrm{SF}_w(\mathcal{M}_2)$) rather than weak fairness ($\mathrm{WF}_w(\mathcal{M})$). To allow either fairness condition, a programming language should provide different flavors of **cobegin** and semaphore operations. If the language provides only one kind of fairness, specifying a different fairness condition requires a complicated encoding with additional variables—if it is even possible.

The TLA formula is longer than the conventional program. The formula can be made shorter by defining some notation. For example, letting $P(sem)$ equal $(0 < sem) \wedge (sem' = sem - 1)$ and $v : d \rightarrow e$ denote $(v = d) \wedge (v' = e)$, action α_1 can be written as

$$\alpha_1 \;\triangleq\; P(sem) \wedge (pc_1 : \text{“}\mathsf{a}\text{”} \rightarrow \text{“}\mathsf{b}\text{”}) \wedge \mathit{Unchanged}\ (x, y, pc_2)$$

There are just two basic reasons why a TLA formula is longer than the corresponding conventional program: (i) what remains unchanged is implicit in a program statement, but must be stated explicitly in an action definition; and (ii) how the control state changes is implicit in the program, but is described explicitly in the formula. We now discuss these two sources of length.

The explicit *Unchanged* clauses add only about 10% to the length of the definition of Ψ in Figure 7; in more realistic examples, they might add somewhat less. Still, why pay that price? An obvious way of simplifying the formulas is to let the omission of a variable from an action mean that the variable is left unchanged. Thus, $x' = x + 1$ would be equivalent to $(x' = x + 1) \wedge (y' = y)$. However, this "simplification" would in fact make TLA much more complicated. For example, it would mean that the obviously true formula $y' = y'$ is not equivalent to true, since the formulas $(x' = x+1) \wedge (y' = y')$ and $x' = x + 1$ would not be equivalent—the first would allow y to change and the second would not. Like ordinary mathematics, TLA is simple because a formula constrains only the variables that it explicitly mentions. This is what makes $x' = x + 1$ so much simpler than $x := x + 1$. Writing *Unchanged* clauses is a small price to pay for the simplicity of ordinary mathematics.

The control structures of ordinary programming languages provide a convenient method of specifying that operations are to be performed in a particular order. Specifying this in TLA requires the use of explicit control variables, which are a source of complexity. However, remember that we are interested in reasoning about abstract descriptions of algorithms, not C code. An abstract algorithm usually has few separate actions, so its control structure is simple; if control variables are needed, they usually add little complexity.

In abstract algorithms, it is just as common to specify that actions can occur in any order as it is to specify that they occur in some particular order. Conventional languages make it is awkward to allow operations to occur in any order. Dijkstra's guarded commands provide a simple mechanism for allowing nondeterminism, but they are ill-suited to describing concurrent programs. For example, can the statement

$$\mathbf{do}\ \langle\, b \rightarrow \mathit{skip} \rangle\ \ \Box\ \langle\, \neg b \rightarrow \mathit{skip} \rangle\ \ \mathbf{do}$$

terminate if some other process is concurrently changing the boolean variable b? We do not know if Dijkstra has ever answered this question, but we believe that there is no single answer that is right in all circumstances. We urge the reader to code the cache example of Figure 10 in his favorite programming language. The precise liveness condition will probably be very difficult or even impossible to express within the language. Even ignoring the liveness condition, we expect that the TLA formula will be simpler than the program.

Any language will be better than TLA at representing a program written especially for that language. Furthermore, a familiar notation, no matter how cumbersome, invariably seems simpler than an unfamiliar one. Our experience suggests that after one gets used to its notation, the TLA description of a "randomly chosen" algorithm is likely to seem simpler than its representation in a conventional programming language, though it may be longer. (If brevity were synonymous with simplicity, APL would be easier to read than Pascal.)

10.3 Reduction

An algorithm must ultimately be translated into a computer program. One develops a program through a series of refinements, starting from a high-level algorithm and eventually reaching a low-level program. Just as we went from Program 1 to the finer-grained Program 2, and from the simple processor/memory interface to the more complicated cached memory, the entire process from specification to C code could in principle be carried out in TLA. "All" we would need is a precise semantics of C, which would allow the translation of any C program into a TLA formula.

In practice, the refinement will be carried out in TLA until it becomes obvious how to hand-translate the TLA formula into a program in a real programming language—one with a compiler that produces satisfactory code. But what does it mean for the translation to be obvious? From the point of view of concurrency, the translation from the TLA formula to the program is obvious when any step of the next-state relation corresponds to an atomic operation of the program. In this sense, the translation from an action $(sem' = sem + 1) \wedge \mathit{Unchanged}\ (\ldots)$ to an atomic $V(sem)$ program statement is obvious.

Real programming languages usually guarantee only an extremely fine grain of atomicity. When executing the statement $x := x + 1$, the read and write of x might each consist of several atomic operations. It would be impractical to describe such a fine-grained program with a TLA formula. Instead, one refines the TLA formula to the point where each step of the next-state relation either corresponds to an atomic program operation like $V(sem)$, or else can be implemented with any grain of atomicity—for example, because it occurs inside an appropriate critical section.

When can an atomic operation be implemented with any grain of atomicity? To answer this, we must first ask: when does a fine-grained program implement a coarser-grained one? There have been a number of partial answers to this question. Some lie in folk theorems—for example, that if shared variables are accessed only in critical sections, then an entire critical section is equivalent to a single atomic operation. Other answers lie in precise results stating that certain classes of properties are satisfied by a fine-grained program if they are satisfied by a coarser-grained version [18].

The question of when a fine-grained program implements a coarser-grained one is answered in TLA by a "reduction" theorem. This theorem seems to include all prior answers as special cases—both the folk theorems and the precise results. The precise statement of the theorem is somewhat complicated, and will be given elsewhere. Here, we give only a rough description of what it says. The theorem's conclusion is approximately

$$\Phi \;\Rightarrow\; \exists\, w_1, \ldots, w_n : \Phi_{red}(w_1/v_1, \ldots, w_n/v_n) \;\wedge\; \Box R \tag{54}$$

where

Φ is the simple TLA formula (with no hidden variables) describing the original program.

Φ_{red} is the coarser-grained "reduced" version of the program.

$v_1, \ldots, v_n$ are all the variables that occur in Φ and Φ_{red}.

R is a predicate containing the variables w_i and v_i.

Think of the v_i as "real" variables and the w_i as "pretend" variables. Formula (54) asserts that there exist pretend variables such that the original program operating on the real variables implements the reduced program operating on the pretend variables, and the relation R always holds between the real and the pretend variables.

In applying the reduction theorem to critical sections, the reduced formula Φ_{red} is obtained from the original formula Φ by changing the next-state relation to turn an entire execution of a critical section into a single step. The relation R asserts that the real and the pretend variables are equal when no process is in its critical section.

In practice, one reasons about the reduced formula Φ_{red} and checks that (54) implies the correctness of the fine-grained formula Φ. For example, in the critical-section application, if a property does not depend on the values assumed by variables while processes are in their critical sections, then Φ

satisfies the property if Φ_{red} does. One must then verify that the formula Φ representing the actual program satisfies the hypotheses of the Reduction Theorem, without actually writing Φ. For complicated languages like C, which lack a reasonable formal semantics, this verification must be informal. Whether formal verification is practical, with either a new language or a useful subset of an existing one, is a topic for research.

10.4 What is TLA Good For?

TLA, like any useful formal system, has a limited domain of applicability. A formalism that encompasses everything is good for nothing. We believe that TLA is useful for specifying and verifying safety and liveness properties of discrete systems. Intuitively, a safety property asserts that something bad does not happen, and a liveness property asserts that something good does eventually happen.

We feel that the most significant limitation of TLA is that TLA properties are true or false for an individual behavior. Thus, one cannot express statistical properties of sets of behaviors—for example, that the program has probability greater than .99 of terminating. The only way we know of verifying such properties is to construct a formal model of the system, use TLA to verify that the system correctly implements the model, and then apply other techniques such as Markov analysis to verify that the model has the desired property.

The limited expressiveness of TLA is not always a disadvantage. As we have seen, TLA allows fine-grained implementations of coarser-grained specifications because it can express only properties that are invariant under stuttering. A formalism that distinguished between doing nothing and taking a step that produces no change would seem to have a tenuous relation to reality. Another class of properties whose inexpressibility in TLA causes us no concern are possibility properties. We have never found it useful to be able to assert that it is possible for a system to produce the right answer. Some formalisms use possibility properties as a substitute for liveness; they cannot prove that the system eventually does produce the right answer, so they prove instead that it might. Since TLA can express liveness properties, it needs no such substitute.

We are advocating TLA as a logic for reasoning about systems that exhibit concurrent activity. Yet, the semantics of TLA is based on sequences of states, with no concept of concurrent activity. The execution of a concurrent algorithm is modeled as a nondeterministic interleaving of "events", where an event is the state-change produced by executing a single atomic operation. Various formalisms for describing concurrent systems have been proposed in which an execution is modeled as a partially ordered set of events, concurrent activity being represented by unordered events. For reasoning about safety and liveness properties, partial orderings are completely equivalent to interleavings. In an interleaving model, a partial ordering of events corresponds to the set of total orderings consistent with it. A system satisfies a safety or liveness property iff every possible execution does. Asserting in a partial-ordering model that the property holds for all possible partial orderings of events is equivalent to asserting in an interleaving model that it holds for all possible sequences of events. Thus, a partial ordering semantics is not needed for reasoning about safety and liveness.

The basic assumption underlying TLA (and most formalisms in computer science) is that an execution can be represented by a discrete collection of atomic events. This is what distinguishes discrete systems from continuous ones. In a register-transfer model of a computer, moving a value into a register is represented as a discrete event, even though it is achieved by continuously changing voltages.

TLA can be used to reason about a discrete system even if its events depend upon continuous physical values. A particularly important physical value is time. Best- and worst-case time bounds on algorithms can be expressed as safety properties and proved with TLA. For example, the assertion that an algorithm always terminates within 15 seconds is a safety property, where time having advanced 15 seconds without the algorithm having terminated is the "something bad" that does not happen. A description of how TLA is used to reason about real time appears in [2].

11 Historical Note

TLA is in the tradition of assertional methods for reasoning about programs. These methods go back to Floyd [10], who first proved partial correctness and termination of sequential programs. Hoare [13] recast partial correctness reasoning into a logical framework. The first practical assertional method for reasoning about concurrent programs was proposed by Ashcroft [5]. Ashcroft's work was followed by a number of variations on the same theme [9, 14, 15]; but the one that became popular is the Owicki/Gries method, developed by Susan Owicki in her thesis [19], which was supervised by David Gries. All these methods, though clothed in different notations, proved safety properties by the use of an invariant; they would be described in TLA as applications of Rule INV1.

Temporal logic was first used to reason about concurrency by Pnueli [20]. It provided the first practical approach to proving more general liveness properties than simple termination. Pnueli introduced the simple temporal logic described in Section 4, with predicates as the only elementary formulas. Pnueli's logic was not expressive enough to describe all desired properties. It was followed by a plethora of proposals for more expressive logics, all obtained by introducing more powerful temporal operators. Pnueli was the first to describe a program by a temporal logic formula [21]. He, and almost everyone else who followed him, represented programs by formulas that are not invariant under stuttering, so a finer-grained program could not implement a coarser-grained one. The observation that invariance under stuttering permits refinement first appeared in [17]. While many of the earlier logics were expressive enough in theory, we believe that TLA is the first logic to provide a practical method for expressing a program as a formula that is invariant under stuttering.

The use of primed and unprimed variables (or their equivalent) for describing "before" and "after" states of a program probably goes back to the early 1970s; we do not know where it first appeared. The idea of actually specifying a program operation by a relation between primed and unprimed variables appears to have been introduced independently by us [16], Hehner [12], and Shankar and Lam [23]. These approaches all used the convention that variables not mentioned are not changed, so they had the inherent complexity epitomized by the observation that $y' = y'$ is not equivalent to true.

The key contribution of TLA is the generalization of Pnueli's simple logic by allowing actions as elementary formulas. This provides the needed expressive power with minimal additional complexity. Another important feature of TLA is the mathematical simplicity it achieves by eschewing variable declarations and types. While many systems for reasoning about programs have been called "logics", not all of them share with TLA the property that if F and G are formulas in the logic, then $\neg F$ and $F \wedge G$ are also formulas in the logic.

Notes

Note 1 (page 4)

Although Val is infinite, from a mathematical point of view it can be "small", since it can be restricted to values constructible by simple rules from primitive values such as booleans and integers. In particular, Russell's paradox can be avoided.

Note 2 (page 5)

Formally, an operator like $+$ is a value (an element of Val), and we regard $\mathsf{m} + \mathsf{n}$ as syntactic sugar for $+(\mathsf{m}, \mathsf{n})$. We assume an evaluation function *eval* that maps tuples of values to values—for example, $eval(+, 2, 3)$ equals 5, since $2 + 3 = +(2, 3) = 5$. The mapping *eval* is assumed to be total, so $eval(+, \text{“}\mathsf{abc}\text{”}, \mathsf{true})$ and $eval(2, +, \mathsf{false})$ are values, though we have no idea what values. (See the discussion of types on pages 27–28.) A state function is either a value, a variable, or an expression of the form $f(f_1, \ldots, f_n)$ where f, f_1, …, f_n are state functions. We define $s[\![f(f_1, \ldots, f_n)]\!]$ to equal $eval(s[\![f]\!], s[\![f_1]\!], \ldots, [\![f_n]\!])$, for any state s.

Note 3 (page 5)

We assume a syntactic class of boolean expressions, so a predicate is a boolean expression built from variables and values. For any values c and d, we assume that $\mathsf{c} = \mathsf{d}$ and $\mathsf{c} \in \mathsf{d}$ are booleans. This implies that $e = f$ and $e \in f$ are boolean expressions, for any expressions e and f. One can also define a richer class of boolean expressions. For example, one can define $\mathsf{c} > \mathsf{d}$ to have the expected meaning if c and d are both numbers, and to equal false otherwise. With this definition, $x > y$ is a predicate if x and y are variables.

Note 4 (page 7)

When proving the validity of an action by ordinary reasoning, x and x' must be considered distinct variables. For example, $(x = y) \Rightarrow (x' = y')$, which one might deduce by naive substitution of equals for equals, is not valid.

Note 5 (page 12)

Formally, we should distinguish actions from temporal formulas. Letting $\theta(\mathcal{A})$ denote the temporal formula that we now write $\mathcal{A}$, we should rewrite (10) as

$$\langle\!\langle s_0, s_1, s_2, \ldots \rangle\!\rangle[\![\theta(\mathcal{A})]\!] \;\stackrel{\Delta}{=}\; s_0[\![\mathcal{A}]\!]s_1$$

We would then notice that the temporal formula we now write $\mathcal{A} \vee \mathcal{B}$ can denote either $\theta(\mathcal{A} \vee \mathcal{B})$ or $\theta(\mathcal{A}) \vee \theta(\mathcal{B})$. However, these two formulas are equivalent, which is why we can get away with writing $\mathcal{A}$ instead of $\theta(\mathcal{A})$.

Note 6 (page 13)

Formula Φ of Figure 2 asserts that every step of Program 1 increments either x or y, but not both. We could allow simultaneous incrementing of x and y by simply redefining $\mathcal{M}$ to equal $\mathcal{M}_1 \vee \mathcal{M}_2 \vee \mathcal{M}_{12}$, where

$$\mathcal{M}_{12} \;\stackrel{\Delta}{=}\; (x' = x + 1) \wedge (y' = y + 1)$$

However, there is no reason to complicate Φ in this way. In representing the execution of $x := x + 1$ by a single step, we are already modeling a complex operation as one event. Nothing would be gained by allowing the additional possibility of representing the executions of two separate statements as a single step.

Note 7 (page 14)

To write the state function (x, y), we must assume that any pair of values is a value. More generally, we assume that $(\mathsf{c}_1, \ldots, \mathsf{c}_n)$ is a value, for any values $\mathsf{c}_1, \ldots, \mathsf{c}_n$.

Note 8 (page 16)

We have told a white lie; $\mathcal{M}_1$ is not equivalent to $\langle \mathcal{M}_1 \rangle_{(x,y)}$. For example, suppose there is a value ∞ such that $\infty + 1$ equals ∞, and let s be a state in which x has the value ∞. Then the pair s, s is an $\mathcal{M}_1$ step, but not an $\langle \mathcal{M}_1 \rangle_{(x,y)}$ step. However, it is true that definitions (15) and (17) are equivalent, because $\mathit{Init}_\Phi \wedge \Box[\mathcal{M}]_{(x,y)}$ implies that the values of x and y are always natural numbers, and $\mathsf{n} + 1 \neq \mathsf{n}$ is true for any natural number n.

Note 9 (page 18)

Observe that *Enabled* $\langle \mathcal{M}_1 \rangle_{(x,y)}$ is not equivalent to true. For example, $\langle \mathcal{M}_1 \rangle_{(x,y)}$ is not enabled in a state in which x equals ∞ (see Note 8). However, $\langle \mathcal{M}_1 \rangle_{(x,y)}$ is enabled in any state in which x is a natural number, so $Init_\Phi \land \Box[\mathcal{M}]_{(x,y)}$ implies $\Box Enabled\ \langle \mathcal{M}_1 \rangle_{(x,y)}$. Hence, (17) and the definition of Φ in Figure 3 are equivalent.

Note 10 (page 19)

Program 1 itself provides another example of parallel composition as conjunction. Rule STL5 of Figure 5 (page 22) and some simple logic shows that Φ is equivalent to $\Phi_1 \land \Phi_2$, where

$$\begin{aligned} \Phi_1 &\triangleq (x = 0) \ \land \ \Box[\mathcal{M}_1]_x \ \land \ \mathrm{WF}_x(\mathcal{M}_1) \\ \Phi_2 &\triangleq (y = 0) \ \land \ \Box[\mathcal{M}_2]_y \ \land \ \mathrm{WF}_y(\mathcal{M}_2) \end{aligned}$$

Formulas Φ_1 and Φ_2 can be viewed as specifications of the two processes forming Program 1, so $\Phi = \Phi_1 \land \Phi_2$ asserts that Φ is the parallel composition of these two components. The TLA formula for a multiprocess program can be written in this way as the conjunction of formulas representing its individual processes whenever each variable is modified by only one process.

Note 11 (page 20)

The hypothesis of STL1 means that F is a propositional tautology or is derivable by the laws of propositional logic from provable formulas.

Note 12 (page 28)

We are assuming that $\mathsf{c} > \mathsf{d}$ is a boolean, for any values c and d (see Note 3), since $\mathsf{p} \Rightarrow \mathsf{q}$ is guaranteed to be a boolean only if p and q are booleans. It is actually not necessary for "abc" $\in \mathsf{Nat}$ to equal false. The formula equals true even if "abc" should happen to equal 135. By not assuming that strings and numbers are disjoint sets, we allow implementations in which strings and numbers share a common representation—for example, as strings of bits. We do, however, assume that "abc" does not equal "xyz".

Note 13 (page 40)

In the definition of $\mathcal{S}(m, v)$, the symbols m and v are parameters. The expression $\mathcal{S}(adr, val)$ denotes the formula obtained by substituting adr for

m and *val* for v in this definition.

Note 14 (page 44)

Quantification over rigid variables can be defined in terms of quantification over program variables by

$$\exists \mathsf{c} : F \quad \triangleq \quad \exists x : F(x/\mathsf{c}) \wedge \Box[\mathsf{false}]_x$$

where c is a rigid variable and x is any program variable that does not occur in the temporal formula F.

Acknowledgments

Frits Vaandrager pointed out the need for the *eval* function, helped me clarify many parts of the exposition, and made a valiant but unsuccessful attempt to persuade me that types are useful in logic. Fred Schneider, Reino Kurki-Suonio, Urban Engberg and Peter Grønning caught a number of errors in earlier versions. A study group at Aarhus University that included Douglas Gurr, Carolyn Brown, Henrik Andersen, and Urban Engberg provided helpful comments. Jeff Line and Jack Wiledon discovered some typographical errors. Peter Ladkin requested Note 11. Luca Cardelli and Cynthia Hibbard suggested some improvements.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. Proceedings of a REX Real-Time Workshop, held in The Netherlands in June, 1991, to be published by Springer-Verlag, 1991.

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

[4] Krzysztof R. Apt. Ten years of Hoare's logic: A survey—part one. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

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

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

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[8] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[9] Lawrence Flon and Norihisa Suzuki. Consistent and complete proof rules for the total correctness of parallel programs. In *Proceedings of 19th Annual Symposium on Foundations of Computer Science*. IEEE, October 1978.

[10] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.

[11] Stephen J. Garland and John V. Guttag. An overview of LP, the Larch prover. In N. Dershowitz, editor, *Proceedings Proc. 3rd Intl. Conf. Rewriting Techniques and Applications*, volume 355 of *Lecture Notes on Computer Science*, pages 137–151. Springer-Verlag, April 1989.

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

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

[14] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, July 1976.

[15] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[16] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[17] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 657–668, Paris, September 1983. IFIP, North Holland.

[18] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[19] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, July 1975.

[20] 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.

[21] Amir Pnueli. The temporal semantics of concurrent programs. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, July 1979.

[22] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Research Report 59, Digital Equipment Corporation Systems Research Center, April 1990.

[23] A. Udaya Shankar and Simon S. Lam. Time-dependent communication protocols. In Simon S. Lam, editor, *Principles of Communication and Networking Protocols*, pages 504–519. IEEE Computer Society Press, Silver Spring, Maryland, 1984.

Index