Verification and Specification of Concurrent Programs

Leslie Lamport

16 November 1993

To appear in the proceedings of a REX Workshop held in The Netherlands in June, 1993.

Verification and Specification of Concurrent Programs

Leslie Lamport

Digital Equipment Corporation
Systems Research Center

Abstract. I explore the history of, and lessons learned from, eighteen years of assertional methods for specifying and verifying concurrent programs. I then propose a Utopian future in which mathematics prevails.

Keywords. Assertional methods, fairness, formal methods, mathematics, Owicki-Gries method, temporal logic, TLA.

Table of Contents

1 A Brief and Rather Biased History of State-Based Methods for Verifying Concurrent Systems

A large body of research on formal verification can be characterized as state-based or assertional. A noncomprehensive overview of this line of research is here. I mention only work that I feel had—or should have had—a significant impact. The desire for brevity combined with a poor memory has led me to omit a great deal of significant work.

1.1 From Floyd to Owicki and Gries, and Beyond

In the beginning, there was Floyd [18]. He introduced the modern concept of correctness for sequential programs: partial correctness, specified by a pre- and postcondition, plus termination. In Floyd's method, partial correctness is proved by annotating each of the program's control points with an assertion that should hold whenever control is at that point. (In a straight-line segment of code, an assertion need be attached to only one control point; the assertions at the other control points can be derived.) The proof is decomposed into separate verification conditions for each program statement. Termination is proved by counting-down arguments, choosing for each loop a variant function—an expression whose value is a natural number that is decreased by every iteration the loop.

Hoare [22] recast Floyd's method for proving partial correctness into a logical framework. A formula in Hoare's logic has the form $P\{S\}Q$, denoting that if the assertion P is true before initiation of program S, then the assertion Q will be true upon S's termination. Rules of inference reduce the proof of such a formula for a complete program to the proof of similar formulas for individual program statements. Hoare did not consider termination. Whereas Floyd considered programs with an arbitrary control structure ("flowchart" programs), Hoare's approach was based upon the structural decomposition of structured programs.

Floyd and Hoare changed the way we think about programs. They taught us to view a program not as a generator of events, but as a state transformer. The concept of state became paramount. States are the province of everyday mathematics—they are described in terms of numbers, sequences, sets, functions, and so on. States can be decomposed as Cartesian products; data refinement is just a function from one set of states to another. The Floyd-Hoare method works because it reduces reasoning about programs to everyday mathematical reasoning. It is the basis of most practical methods of sequential program verification.

Ashcroft [8] extended state-based reasoning to concurrent programs. He generalized Floyd's method for proving partial correctness to concurrent programs, where concurrency is expressed by fork and join operations. As in Floyd's method, one assigns to each control point an assertion that should hold whenever control is at that point. However, since control can be simultaneously at multiple control points, the simple locality of Floyd's method is lost. Instead, the annotation is viewed as a single large *invariant*, and one must prove that executing each statement leaves this invariant true. Moreover, to capture the intricacies of interprocess synchronization, the individual assertions attached to the control points may have to mention the control state explicitly.

Owicki and Gries [36] attempted to reason about concurrent programs by generalizing Hoare's method. They considered structured programs, with concurrency expressed by **cobegin** statements, and added to Hoare's logic the following proof rule:

$$\frac{\{P_1\}S_1\{Q_1\},\ \ldots,\ \{P_n\}S_n\{Q_n\}}{\{P_1 \wedge \ldots \wedge P_n\}\ \textbf{cobegin}\ S_1 \parallel \ldots \parallel S_n\ \textbf{coend}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

provided $\{P_1\}S_1\{Q_1\}, \ldots, \{P_n\}S_n\{Q_n\}$ *are interference-free*

This looks very much like one of Hoare's rules of inference that decomposes the proof of properties of the complete program into proofs of similar properties of the individual program statements. Unlike Ashcroft's method, the assertions in the Owicki-Gries method do not mention the control state.

Despite its appearance, the Owicki-Gries method is really a generalization of Floyd's method in Hoare's clothing. Interference freedom is a condition on the complete annotations used to prove the Hoare triples $\{P_k\}S_k\{Q_k\}$. It asserts that for each i and j with $i \neq j$, executing any statement in S_i with its precondition true leaves invariant each assertion in the annotation of S_j. Moreover, Owicki and Gries avoid explicitly mentioning the control state in the annotations only by introducing auxiliary variables to capture the control information. This casting of a Floyd-like method in Hoare-like syntax has led to a great deal of confusion [14].

Like Floyd's method, the Owicki-Gries method decomposes the proof into verification conditions for each program statement. However, for an n-statement program, there are $O(n^2)$ verification conditions instead of the $O(n)$ conditions of Floyd's method. Still, unlike Ashcroft's method, the verification conditions involve only local assertions, not a global invariant. I used to think that this made it a significant improvement over Ashcroft's method. I now think otherwise.

The fundamental idea in generalizing from sequential to concurrent programs is switching from partial correctness to invariance. Instead of thinking only about what is true before and after executing the program, one must think about what remains true throughout the execution. The concept of invariance was introduced by Ashcroft; it is hidden inside the machinery of the Owicki-Gries method. A proof is essentially the same when carried out in either method. The locality of reasoning that the Owicki-Gries method obtains from the structure of the program is obtained in the Ashcroft method from the structure of the invariant. By hiding invariance and casting everything in terms of partial correctness, the Owicki-Gries method tends to obscure the fundamental principle behind the proof.

Owicki and Gries considered a toy programming language containing only a few basic sequential constructs and **cobegin**. Over the years, their method has been extended to a variety of more sophisticated toy languages. CSP was handled independently by Apt, Francez, and de Roever [6] and by Levin and Gries [31]. de Roever and his students have developed a number of Owicki-Gries-style proof systems, including one for a toy version of ADA [20]. The Hoare-style syntax, with the concomitant notion of interference freedom, has been the dominant fashion.

Although these methods have been reasonably successful at verifying simple algorithms, they have been unsuccessful at verifying real programs. I do not know of a single case in which the Owicki-Gries approach has been used for the formal verification of code that was actually compiled, executed, and used. I don't expect the situation to improve any time soon. Real programming languages are too complicated for this type of language-based reasoning to work.

1.2 Temporal Logic

The first major advance beyond Ashcroft's method was the introduction by Pnueli of temporal logic for reasoning about concurrent programs [38]. I will now describe Pnueli's original logic.

Formulas in the logic are built up from state predicates using Boolean connectives and the temporal operator $\Box$. A state predicate (called a predicate for short) is a Boolean-valued expression, such as $x + 1 > y$, built from constants and program variables.

The semantics of the logic is defined in terms of states, where a state is an assignment of values to program variables. An infinite sequence of states is called a behavior; it represents an execution of the program. (For convenience, a terminating execution is represented by an infinite behavior in which the final state is repeated.) The meaning $[\![P]\!]$ of a predicate P is a Boolean-valued function on program states. For example, $[\![x+1 > y]\!](s)$ equals *true* iff one plus the value of x in state s is greater than the value of y in state s. The meaning $[\![F]\!]$ of a formula F is a Boolean-valued function on behaviors, defined as follows.

$[\![P]\!](s_0, s_1, \ldots) \ \triangleq\ [\![P]\!](s_0)$, for any state predicate P.

$[\![F \diamond G]\!](s_0, s_1, \ldots) \ \triangleq\ [\![F]\!](s_0, s_1, \ldots) \diamond [\![G]\!](s_0, s_1, \ldots)$, for any Boolean operator $\diamond$.

$[\![\Box F]\!](s_0, s_1, \ldots) \ \triangleq\ \forall\, n : [\![F]\!](s_n, s_{n+1}, \ldots)$

Intuitively, a formula is an assertion about the program's behavior from some fixed time onwards. The formula $\Box F$ asserts that F is always true—that is, true now and at all future times. The formula $\Diamond F$, defined to equal $\neg\Box\neg F$, asserts that F is eventually true—that is, true now or at some future time. The formula $F \leadsto G$ (read F *leads to* G), defined to equal $\Box(F \Rightarrow \Diamond G)$, asserts that if F ever becomes true, then G will be true then or at some later time.

To apply temporal logic to programs, one defines a programming-language semantics in which the meaning $[\![\Pi]\!]$ of a program Π is a set of behaviors. The assertion that program Π satisfies temporal formula F, written $\Pi \models F$, means that $[\![F]\!](\sigma)$ equals *true* for all behaviors in $[\![\Pi]\!]$. Invariance reasoning is expressed by the following Invariance Rule.

$$\frac{\{I\}\, S\, \{I\},\ \text{for every atomic operation } S \text{ of } \Pi}{\Pi \models I \Rightarrow \Box I}$$

The Ashcroft and Owicki-Gries methods, and all similar methods, can be viewed as applications of this rule.

Although the Invariance Rule provides a nice formulation of invariance, temporal logic provides little help in proving invariance properties. The utility of temporal logic comes in proving liveness properties—properties asserting that something eventually happens. Such properties are usually expressed with the leads-to operator $\leadsto$.

To prove liveness, one needs some sort of fairness requirement on the execution of program statements. Two important classes of fairness requirements have emerged: weak and strong fairness. Weak fairness for an atomic operation S means that if S is continuously enabled (capable of being executed), then S must eventually be executed. Strong fairness means that if S is repeatedly enabled (perhaps repeatedly disabled as well), then S must eventually be executed. These requirements can be expressed by the following proof rules.

$$\text{Weak Fairness:}\ \frac{P \Rightarrow S \text{ enabled},\ \{P\}\,S\,\{Q\}}{\mathit{\Pi} \models (\Box P) \leadsto Q}$$

$$\text{Strong Fairness:}\ \frac{P \Rightarrow S \text{ enabled},\ \{P\}\,S\,\{Q\}}{\mathit{\Pi} \models (\Box\Diamond P) \leadsto Q}$$

A method for proving liveness properties without temporal logic had been proposed earlier [25]. However, it was too cumbersome to be practical. Temporal logic permitted the integration of invariance properties into liveness proofs, using the following basic rule:

$$\frac{\mathit{\Pi} \models P \leadsto Q,\ \mathit{\Pi} \models Q \Rightarrow \Box Q}{\mathit{\Pi} \models P \leadsto \Box Q}$$

The use of temporal logic made proofs of liveness properties practical. The basic method of proving $F \leadsto G$ is to find a well-founded collection $\mathcal{H}$ of formulas containing F and prove, for each H in $\mathcal{H}$, that $\mathit{\Pi} \models H \leadsto (J \vee G)$ holds for some $J \prec H$ [37]. This generalizes the counting-down arguments of Floyd, where the variant function v corresponds to taking the set $\{v = n : n \text{ a natural number}\}$ for $\mathcal{H}$.

1.3 Unity

Chandy and Misra recognized the importance of invariance in reasoning about concurrent programs. They observed that the main source of confusion in the Owicki-Gries method is the program's control structure. They cut this Gordian knot by introducing a toy programming language, called Unity, with no control [11]. Expressed in terms of Dijkstra's **do** construct, all Unity programs have the following simple form.

$$\mathbf{do}\ \ P_1 \rightarrow S_1 \ \Box\ \ldots\ \Box\ P_n \rightarrow S_n\ \ \mathbf{od}$$

Fairness is assumed for the individual clauses of the **do** statement.

The idea of reasoning about programs by translating them into this form had been proposed much earlier by Flon and Suzuki [17]. However, no-one had considered dispensing with the original form of the program.

For reasoning about Unity programs, Chandy and Misra developed Unity logic. Unity logic is a restricted form of temporal logic that includes the formulas $\Box P$ and $P \leadsto Q$ for predicates P and Q, but does not allow nested temporal operators. Chandy and Misra developed proof rules to formalize the style of reasoning that had been developed for proving invariance and leads-to properties. Unity provided the most elegant formulation yet for these proofs.

2 An Even Briefer and More Biased History of State-Based Specification Methods for Concurrent Systems

In the early '80s, it was realized that proving invariance and leads-to properties of a program is not enough. One needs to express and prove more sophisticated requirements, like first-come-first-served scheduling. Moreover, the standard program verification methods required that properties such as mutual exclusion be stated in terms of the implementation. To convince oneself that the properties proved actually ensured correctness, it is necessary to express correctness more abstractly.

A specification is an abstract statement of what it means for a system to be correct. A specification is not very satisfactory if one has no idea how to prove that it is satisfied by an implementation. Hence, a specification method should include a method for proving correctness of an implementation. Often, one views an implementation as a lower-level specification. Verification then means proving that one specification implements another.

2.1 Axiomatic Specifications

In an axiomatic method, a specification is written as a list of properties. Formally, each property is a formula in some logic, and the specification is the conjunction of those formulas. Specification L implements specification H iff (if and only if) the properties of L imply that the properties of H are satisfied. In other words, implementation is just logical implication.

The idea of writing a specification as a list of properties sounds marvelous. One can specify *what* the system is supposed to do, without specifying *how* it should do it. However, there is one problem: in what language does one write the properties?

After Pnueli, the obvious language for writing properties was temporal logic. However, Pnueli's original logic was not expressive enough. Hence, researchers introduced a large array of new temporal operators such as $\bigcirc F$, which asserts that F holds in the next state, and $F \,\mathcal{U}\, G$, which asserts that F holds until G does [33]. Other methods of expressing temporal formulas were also devised, such as the interval logic of Schwartz, Melliar-Smith, and Vogt [39].

Misra has used Unity logic to write specifications [35]. However, Unity logic by itself is not expressive enough; one must add auxiliary variables. The auxiliary variables are not meant to be implemented, but they are not formally distinguished from the "real" variables. Hence, Unity specifications must be viewed as semi-formal.

2.2 Operational Specifications

In an operational approach, a specification consists of an abstract program written in some form of abstract programming language. This approach was advocated in the early '80s by Lam and Shankar [24] and others [26]. More recent instances include the I/O automaton approach of Lynch and Tuttle [32] and Kurki-Suonio's DisCo language [23].

An obvious advantage of specifying a system as an abstract program is that while few programmers are familiar with temporal logic, they are all familiar with programs. A disadvantage of writing an abstract program as a specification is that a programmer is apt to take it too literally, allowing the specification of what the system is supposed to do bias the implementor towards some particular way of getting the system to do it.

An abstract program consists of three things: a set of possible initial states, a next-state relation describing the states that may be reached in a single step from any given state, and some fairness requirement. The next-state relation is usually partitioned into separate *actions*, and the fairness requirement is usually expressed in terms of these actions. To prove that one abstract program Π_1 implements another abstract program Π_2, one must prove:

1. Every possible initial state of Π_1 is a possible initial state of Π_2.
2. Every step allowed by Π_1's next-state relation is allowed by Π_2's next-state relation—a condition called *step simulation*. To prove step simulation, one first proves an invariant that limits the set of states to be considered.
3. The fairness requirement of Π_1 implies the fairness requirement of Π_2. How this is done depends on how fairness is specified.

Thus far, most of these operational approaches have been rather *ad hoc*. To my knowledge, none has a precisely defined language, with formal semantics, and proof rules. This is probably due to the fact that an abstract program is still a program, and even simple languages are difficult to describe formally.

The Unity language has also been proposed for writing specifications. However, it has several drawbacks as a specification language:

- It provides no way of defining nontrivial data structures.
- It has no abstraction mechanism for structuring large specifications. (The procedure is the main abstraction mechanism of conventional languages.)
- It lacks a hiding mechanism. (Local variable declarations serve this purpose in conventional languages.)
- It has a fixed fairness assumption; specifying other fairness requirements is at best awkward.

2.3 Finite-State Methods

An important use of state-based methods has been in the automatic verification of finite-state systems. Specifications are written either as abstract programs or temporal-logic formulas, and algorithms are applied to check that one specification implements another. This work is discussed by Clarke [12], and I will say no more about it.

3 What We Have Learned

History serves as a source of lessons for the future. It is useful to reflect on what we have (or should have) learned from all this work on specification and verification of concurrent programs.

3.1 Not Sequential vs. Concurrent, but Functional vs. Reactive

Computer scientists originally believed that the big leap was from sequentiality to concurrency. We thought that concurrent systems needed new approaches because many things were happening at once. We have learned instead that, as far as formal methods are concerned, the real leap is from functional to reactive systems.

A functional system is one that can be thought of as mapping an input to an output. (In the Floyd/Hoare approach, the inputs and outputs are states.) A behavioral system is one that interacts in more complex ways with its environment. Such a system cannot be described as a mapping from inputs to outputs; it must be described by a set of behaviors. (A temporal-logic formula F specifies the set of all behaviors σ such that $[\![F]\!](\sigma)$ equals *true*.)

Even if the purpose of a concurrent program is just to compute a single output as a function of a single input, the program must be viewed as a reactive system if there is interaction among its processes. If there is no such interaction—for example, if each process computes a separate part of the output, without communicating with other processes—then the program can be verified by essentially the same techniques used for sequential programs.

I believe that our realization of the significance of the reactive nature of concurrent systems was due to Harel and Pnueli [21].

3.2 Invariance Under Stuttering

A major puzzle that arose in the late '70s was how to prove that a fine-grained program implements a coarser-grained one. For example, how can one prove that a program in which the statement $x := x + 1$ is executed atomically is correctly implemented by a lower-level program in which the statement is executed by separate *load*, *add*, and *store* instructions? The answer appeared in the early '80s: invariance under stuttering [27].

A temporal formula F is said to be invariant under stuttering if, for any behavior σ, adding finite state-repetitions to σ (or removing them from σ) does not change the value of $[\![F]\!](\sigma)$. (The definition of what it means for a set of behaviors to be invariant under stuttering is similar.)

To understand the relevance of stuttering invariance, consider a simple specification of a clock that displays hours and minutes. A typical behavior allowed by such a specification is the sequence of clock states

11:23, 11:24, 11:25, ...

We would expect this specification to be satisfied by a clock that displays hours, minutes, and seconds. More precisely, if we ignore the seconds display, then the hours/minutes/seconds clock becomes an hours/minutes clock. A typical behavior of the hours/minutes/seconds clock is

11:23:57, 11:23:58, 11:23:59, 11:24:00, ...

and ignoring the seconds display converts this behavior to

11:23, 11:23, 11:23, 11:24, ...

This behavior will satisfy the specification of the hours/minutes clock if that specification is invariant under stuttering.

All formulas of Pnueli's original temporal logic are invariant under stuttering. However, this is not true of most of the more expressive temporal logics that came later.

3.3 The Definitions of Safety and Liveness

The informal definitions of safety and liveness appeared in the '70s [25]: a safety property asserts that something bad does not happen; a liveness property asserts that something good does happen. Partial correctness is a special class of safety property, and termination is a special class of liveness property. Intuitively, safety and liveness seemed to be the fundamental way of categorizing correctness properties. But, this intuition was not backed up by any theory.

A more precise definition of safety is: a safety property is one that is true for an infinite behavior iff it is true for every finite prefix [4]. Alpern and Schneider made this more precise and defined liveness [5]. They first defined what it means for a finite behavior to satisfy a temporal formula: a finite behavior ρ satisfies formula F iff ρ can be extended to an infinite behavior that satisfies F. Formula F is a safety property iff the following condition holds: F is satisfied by a behavior σ iff it is satisfied by every finite prefix of σ. Formula F is a liveness property iff it is satisfied by every finite behavior. Alpern and Schneider then proved that every temporal formula can be written as the conjunction of a safety and a liveness property. This result justified the intuition that safety and liveness are the two fundamental classes of properties.

In a letter to Schneider, Gordon Plotkin observed that, if we view a temporal formula as a set of behaviors (the set of behaviors satisfying the formula), then safety properties are the closed sets and liveness properties are the dense sets in a standard topology on sequences. The topology is defined by a distance function in which the distance between sequences $s_0, s_1, \ldots$ and $t_0, t_1, \ldots$ is $1/(n+1)$, where n is the smallest integer such that $s_n \neq t_n$. Alpern and Schneider's result is a special case of the general result in topology that every set can be written as the intersection of a closed set and a dense set.

3.4 Fairness is Machine Closure

Fairness and concurrency are closely related. In an interleaving semantics, concurrent systems differ from nondeterministic sequential ones only through their fairness requirements. However, for many years, we lacked a precise statement of what constitutes a fairness requirement. Indeed, this question is not even addressed in a 1986 book titled *Fairness* [19]. The topological characterization of safety and liveness provided the tools for formally characterizing fairness.

Let $\mathcal{C}(F)$ denote the closure of a property F—that is, the smallest safety property containing F. (In logical terms, $\mathcal{C}(F)$ is the strongest safety property implied by F.) The following two conditions on a pair of properties (S, L) are equivalent.

$S = \mathcal{C}(S \wedge L)$

S is a safety property, and every finite behavior satisfying S is a prefix of an infinite behavior satisfying $S \wedge L$.

A pair of properties satisfying these conditions is said to be *machine closed.* (Essentially the same concept was called "feasibility" by Apt, Francez, and Katz [7].)

Fairness means machine closure. Recall that a program can be described by an initial condition, a next-state relation, and a fairness requirement. Let S be the property asserting that a behavior satisfies the initial condition and next-state relation, and let L be the property asserted by the fairness requirement. Machine closure of (S, L) means that a scheduler can execute the program without having to worry about "painting itself into a corner" [7]. As long as the program is started in a correct initial state and the program's next-state relation is obeyed, it always remains possible to satisfy L. This condition characterizes what it means for L to be a fairness requirement. So-called fair scheduling of processes is actually a fairness requirement iff the pair (S, L) is machine closed, for every program in the language.

In most specification methods, one specifies separately a safety property S and a liveness property L. Machine closure of (S, L) appears to be an important requirement for a specification. The lack of machine closure means that the liveness property L is actually asserting an additional safety property. This usually indicates a mistake, since one normally intends S to include all safety properties. Moreover, the absence of machine closure is a potential source of incompleteness for proof methods [1].

3.5 Hiding is Existential Quantification

An *internal* variable is one that appears in a specification, but is not meant to be implemented. Consider the specification of a memory module. What one must specify is the sequence of read and write operations. An obvious way to write such a specification is in terms of a variable M whose value denotes the current contents of the memory. The variable M is an internal variable; there is no requirement that M be implemented.

Advocates of axiomatic specifications denigrated the use of internal variables, claiming it leads to insufficiently abstract specifications that bias the implementation. Advocates of operational specifications, which rely heavily on internal variables, claimed that the use of internal variables simplifies specifications.

An operational specification consists of an abstract program, and one can describe such a program by a temporal logic formula. The temporal logic representation of an operational specification differs formally from an axiomatic specification only because some of its variables are internal. If those internal variables can be formally "hidden", then the distinction between operational and axiomatic specification vanishes.

In temporal logic, variable hiding is expressed by existential quantification. If F is a temporal formula, then the formula $\boldsymbol{\exists} x : F$ asserts that there is some way of choosing values for the variable x that makes F true. This is precisely what it means for x to be an internal variable of a specification F—the system behaves as if there were such a variable, but that variable needn't be implemented, so its actual value is irrelevant.

The temporal quantifier $\boldsymbol{\exists}$ differs from the ordinary existential quantification operator $\exists$ because $\boldsymbol{\exists}$ asserts the existence of a sequence of values—one for each state in the behavior—rather than a single value. However, $\boldsymbol{\exists}$ obeys the usual predicate calculus rules for existential quantification. The precise definition of $\boldsymbol{\exists}$ that makes it preserve invariance under stuttering is a bit tricky [28]; the definition appears in Figure 5 of Section 5.3.

The quantifier $\boldsymbol{\exists}$ allows a simple statement of what it means to add an auxiliary variable to a program. Recall that Owicki and Gries introduced auxiliary variables—program variables that are added for the proof, but that don't change the behavior of the program. If F is the temporal formula that describes the program, then adding an auxiliary variable v means finding a new formula F^v such that $\boldsymbol{\exists} v : F^v$ is equivalent to F.

3.6 Specification Methods that Don't Work

Knowing what doesn't work is as important as knowing what does. Progress is the result of learning from our mistakes.

The lesson I learned from the specification work of the early '80s is that axiomatic specifications don't work. The idea of specifying a system by writing down all the properties it satisfies seems perfect. We just list what the system must and must not do, and we have a completely abstract specification. It sounds wonderful; it just doesn't work in practice.

After writing down a list of properties, it is very hard to figure out what one has written. It is hard to decide what additional properties are and are not implied by the list. As a result, it becomes impossible to tell whether a specification says all that it should about a system. With perseverance, one can write an axiomatic specification of a simple queue and convince oneself that it is correct. The task is hopeless for systems with any significant degree of complexity.

To illustrate the complexity of real specifications, I will show you a small piece of one. The specification describes one particular view of a switch for a local area network. Recall that in an operational specification, one specifies a next-state relation by decomposing it into separate actions. Figure 1 shows the specification of a typical action, one of about a dozen. It has been translated into an imaginary dialect of Pascal. The complete specification is about 700 lines. It would be futile to try to write such a specification as a list of properties.

3.7 Specification Methods that Work for the Wrong Reason

The specification methods that do work describe a system as an abstract program. Unfortunately, when computer scientists start describing programs, their particular taste in programming languages comes to the fore.

Suppose a computer scientist wants to specify a file system. Typically, he starts with his favorite programming language—say TEX. He may modify the language a bit, perhaps by adding constructs to describe concurrency and/or nondeterminism, to get Concurrent TEX. We might not think that TEX is a very good specification language, but it is Turing complete, so it can describe anything, including file systems. In the course of describing a system precisely, one is forced to examine it closely—a process that usually reveals problems. The exercise of writing the specification is therefore a great success, since it revealed problems with the system. The specifier, who used TEX because it was his favorite language, is led to an inescapable conclusion: the specification was a success because Concurrent TEX is a wonderful language for writing specifications.

Although Concurrent TEX, being Turing complete, can specify anything, there are two main problems with using it as a specification language.

First, one has to introduce a lot of irrelevant mechanism to specify something in Concurrent TEX. It is hard for someone reading the specification to tell how much of the mechanism is significant, and how much is there because TEX lacks the power to write the specification more abstractly. I know of one case of a bus protocol that was specified in a programming language—Pascal rather than TEX. Years later, engineers wanted to redesign the bus, but they had no idea which aspects of the specification were crucial and which were artifacts of Pascal. For example, the specification stated that two operations were performed in a certain order, and the engineers didn't know if that order was important or was an arbitrary choice made because Pascal requires all statements to be ordered.

The second problem with Concurrent TEX is more subtle. Any specification is an abstraction, and the specifier has a great deal of choice in what aspects of the system appear in the abstraction. The choice should be made on the basis of what is important. However, programming languages inevitably make it harder to describe some aspects of a system than others. The specifier who thinks in terms of a programming language will unconsciously choose his abstractions on the basis of how easily they can be described, not on the basis of their

```
ACTION LinkCellArrives(l : Link, p : P) :
  LOCAL lState := inport[p].lines[l]       ;
        lc     := Head(lState.in)          ;
        vv     := lState.vcMap[lc.v]       ;
        incirc := inport[p].circuits[vv]   ;
        ww     := lState.vcMap[lc.ack]     ;
        noRoom := lState.space = 0         ;
        toQ    := incirc.enabled           ;
                    AND NOT incirc.stopped
                    AND NOT incirc.discard
                    AND NOT incirc.cbrOnly
                    AND NOT noRoom         ;
        ocirc  := outport[p].circuits[ww]  ;
        outq   := CHOOSE qq : qq IN incirc.outPort ;
        iStart := NOT ocirc.discardC
                    AND ocirc.balance = 0
                    AND NOT ocirc.startDis ;
  BEGIN
    IF lState.in /= < > AND NOT inport[p].cellArr
     THEN IF incirc.discard  OR noRoom
           THEN inport[p].circuits[vv].cells.body :=
                  CONCAT(inport[p].circuits[vv].cells.body,
                          RECORD body := lc.body ;
                                 from := TRUE)     END    ;
          IF toQ
           THEN inport[p].circuits[vv].cells.queued := TRUE;
                IF NOT incirc.queued
                  THEN inport[p].outportQ[outq] :=
                         CONCAT(inport[p].outportQ[outq], vv);
          IF noRoom
           THEN inport[p].lines[l].space :=
                  inport[p].lines[l].space - 1               ;
          inport[p].lines[l].in :=
                Tail(inport[p].lines[l].in)                  ;
          inport[p].cellArr := TRUE                          ;
          IF NOT ocirc.discardC
            THEN outport[p].circuits[ww].balance :=
                         outport[p].circuits[ww].balance + 1 ;
                 outport[p].circuits[ww].sawCred := TRUE
            ELSE outport[p].circuits[ww].sawCred := FALSE
          IF iStart THEN outport[p].delayL.StartDL :=
                           RECORD w := ww ;
                                  p := ocirc.inPort END
  END
```

Fig. 1. One small piece of a real specification, written in pseudoPascal.

importance. For example, no matter how important the fairness requirements of a system might be, they will not appear in the specification if the language does not provide a convenient and flexible method of specifying fairness.

Computer scientists tend to be so conscious of the particular details of the language they are using, that they are often unaware that they are just writing a program. The author of a CCS specification [34] will insist that he is using process algebra, not writing a program. But, it would be hard to state formally the difference between his CCS specification and a recursive program.

4 Other Methods

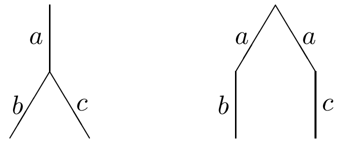
So far, I have discussed only state-based methods. There are two general ways of viewing an execution of a system—as a sequence of states or as a collection of events (also called actions). Although some of the formalisms I have mentioned, such as I/O automata, do consider a behavior to be a sequence of events rather than states, their way of specifying sequences of events are very much state-based. State-based methods all share the same assertional approach to verification, in which the concept of invariance plays a central role.

Event-based formalisms include so-called algebraic approaches like CCS [34] and functional approaches like the method of Broy [9, 10]. They attempt to replace state-based assertional reasoning with other proof techniques. In the algebraic approach, verification is based on applying algebraic transformations. In the functional approach, the rules of function application are used.

I have also ignored state-based approaches based on branching-time temporal logic instead of the linear-time logic described above [15]. In branching-time logic, the meaning of a program is a tree of possibilities rather than a set of sequences. The formula $\Box F$ asserts that F is true on all branches, and $\Diamond F$ (defined to be $\neg\Box\neg F$) asserts that F is true on some branch. While $\Box$ still means *always*, in branching-time logic $\Diamond$ means *possibly* rather than *eventually*. A branching-time logic underlies most algebraic methods.

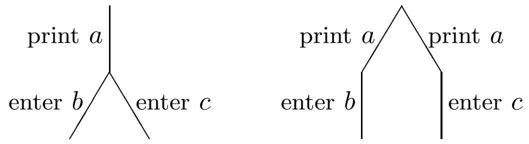
Comparisons between radically different formalisms tend to cause a great deal of confusion. Proponents of formalism A often claim that formalism B is inadequate because concepts that are fundamental to specifications written with A cannot be expressed with B. Such arguments are misleading. The purpose of a formalism is not to express specifications written in other formalism, but to specify some aspects of some class of computer systems. Specifications of the same system written with two different formalisms are likely to be formally incomparable.

To see how comparisons of different formalisms are misleading, let us consider the common argument that sequences are inadequate for specifying systems, and one needs to use trees. The argument goes as follows. The set of sequences that forms the specification in a sequence-based method is the set of paths through the tree of possible system events. Since a tree is not determined by its set of paths, sequences are inadequate for specifying systems. For example, consider the following two trees.



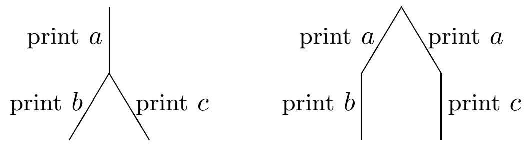
The first tree represents a system in which an a event occurs, then a choice is made between doing a b event or a c event. The second tree represents a system that chooses immediately whether to perform the sequence of events $\langle a, b\rangle$ or the sequence $\langle a, c\rangle$. Although these trees are different, they both have the same set of paths—namely, the two-element set $\{\langle a, b\rangle, \langle a, c\rangle\}$. Sequence-based formalisms are supposedly inadequate because they cannot distinguish between these two trees.

This argument is misguided because a sequence-based formalism doesn't have to distinguish between different trees, but between different systems. Let us replace the abstract events a, b, and c by actual system events. First, suppose a represents the system printing the letter a, while b and c represent the user entering the letter b or c. The two trees then become



In the first tree, after the system has printed a, the user can enter either b or c. In the second tree, the user has no choice. But, why doesn't he have a choice? Why can't he enter any letter he wants? Suppose he can't enter other letters because the system has locked the other keys. In a state-based approach, the first system is described by a set of behaviors in which first a is printed and the b and c keys are unlocked, and then either b or c is entered. The second system is described by a set of behaviors in which first a is printed and then either the b key is unlocked and b is entered or else the c key is unlocked and c is entered. These two different systems are specified by two different sets of behaviors.

Now, suppose events a, b, and c are the printing of letters by the system. The two trees then become



The first tree represents a system that first prints a and then chooses whether to print b or c next. The second represents a system that decides which of the sequences of letters to print before printing the first a. Let us suppose the system makes its decision by tossing a coin. In a state-based approach, the first system is specified by behaviors in which the coin is first tossed, then two letters are printed. The second system is specified by behaviors in which first a is printed, then the coin is tossed, and then the second letter is printed. These are two different systems, with two different specifications.

Finally, suppose that the system's coin is internal and not observable. In a state-based method, one hides the state of the coin. With the coin hidden, the resulting two sets of behaviors are the same. In a state-based method, the two specifications are equivalent. But, the two systems *are* equivalent. If the coin is not observable, then there is no way to observe any difference between the systems.

Arguments that compare formalisms directly, without considering how those formalisms are used to specify actual systems, are useless.

5 A Brief Advertisement for My Approach to State-Based Verification and Specification of Concurrent Systems

I claim that axiomatic methods of writing specification don't work, and that operational methods are unsatisfactory because they require a complicated programming language. The resolution of this dilemma is to combine the best of both worlds—the elegance and conceptual simplicity of the axiomatic approach and the expressive power of abstract programs. This is done by writing abstract programs as mathematical formulas.

5.1 The Power of Formal Mathematics

Mathematicians tend to be fairly informal, inventing notation as they need it. Many computer scientists believe that formalizing mathematics would be an enormously difficult undertaking. They are wrong. Everyday mathematics can be formalized using only a handful of operators. The operators $\wedge$, $\neg$, $\in$, and **choose** (Hilbert's ε [30]) are a complete set. In practice, one uses a larger set of operators, such as the ones in Figure 2. These operators should all be familiar, except perhaps for **choose**, which is defined by letting **choose** $x : p$ equal an arbitrary x such that p is true, or an unspecified value if no such x exists. Among the uses of this operator is the formalization of recursive definitions, the definition

$$\mathit{fact}[n : \mathbf{N}] \;\stackrel{\Delta}{=}\; \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * \mathit{fact}[n-1]$$

being syntactic sugar for

$$\mathit{fact} \;\stackrel{\Delta}{=}\; \textbf{choose } f \;:\; f \;=\; [n \in \mathbf{N} \mapsto \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n \cdot f[n-1]\,]$$

As an example of how easy it is to formalize mathematical concepts with these operators, Figure 3 shows the definition of the Riemann integral, assuming only the sets $\mathbf{N}$ of natural numbers and $\mathbf{R}$ of real numbers, and the usual operators $+$, $*$, $<$, and $\leq$. This is a completely formal definition, written in a language with a

$\wedge \quad \vee \quad \neg \quad \Rightarrow$ [implication] $\quad \forall \quad \exists$

$= \quad \neq \quad \in \quad \notin \quad \emptyset \quad \cup \quad \cap \quad \subseteq \quad \setminus$ [set difference]

$\{e_1, \ldots, e_n\}$	[Set consisting of elements e_i]
$\{x \in S : p\}$	[Set of elements x in S satisfying p]
$\{e : x \in S\}$	[Set of elements e, for all x in S]
2^S	[Set of subsets of S]
$\bigcup S$	[Union of all elements of S]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} component is e_i]
$S_1 \times \ldots \times S_n$	[Set of n-tuples with i^{th} component in S_i]
choose $x : p$	[Hilbert's ε operator]
$f[e]$	[Function application]
dom f	[Domain of the function f]
$[S \rightarrow T]$	[Set of functions with domain S and range subset of T]
$[x \in S \mapsto e]$	[Function f such that **dom** $f = S$ and $f[x] = e$ for $x \in S$]
if p **then** e_1 **else** e_2	

Fig. 2. Operators for formalizing mathematics.

precise syntax and semantics. For greater readability, some of the operators have been printed in conventional mathematical notation. In the actual language, one would have to write something like `Int(a, b, f)` rather than $\int_a^b f$. The language uses the convention that a list of formulas bulleted with $\wedge$ or $\vee$ denotes the conjunction or disjunction of the formulas, and indentation is used to eliminate parentheses.

5.2 Specifying Programs with Mathematical Formulas

Consider the following simple program.

program *Increment*
var x, y **initially** 0
cobegin $x := x + 1 \;\square\; y := y + 1$ **coend**

To specify this program, we must specify its initial condition, next-state relation, and fairness requirements. The initial condition *Init* is obvious:

$$Init \;\triangleq\; (x = 0) \wedge (y = 0)$$

$$
\begin{array}{lcl}
\{m \ldots n\} & \triangleq & \{i \in \mathbf{N} : (m \leq i) \wedge (i \leq n)\} \\
\mathit{seq}(S) & \triangleq & \bigcup \{\, [\{1 \ldots n\} \rightarrow S] : n \in \mathbf{N}\} \\
\mathit{len}(s) & \triangleq & \textbf{choose}\ n : (n \in \mathbf{N}) \wedge (\textbf{dom}\ s = \{1 \ldots n\}) \\
\mathbf{R}^{+} & \triangleq & \{r \in \mathbf{R} : 0 < r\} \\
|r| & \triangleq & \textbf{if}\ r < 0\ \textbf{then}\ -r\ \textbf{else}\ r \\
\sum_{m}^{n : \mathbf{N}} f & \triangleq & \textbf{if}\ n < m\ \textbf{then}\ 0\ \textbf{else}\ f[n] + \sum_{m}^{n-1} f \\
\mathcal{P}_a^b(\delta) & \triangleq & \{\, p \in \mathit{seq}(\mathbf{R}) : \\
 & & \qquad \wedge\ (p[1] = a)\ \wedge\ (p[\mathit{len}(p)] = b) \\
 & & \qquad \wedge\ \forall\, i \in \{1 \ldots \mathit{len}(p){-}1\} :\ \wedge\ \textbf{if}\ a \leq b\ \textbf{then}\ p[i] \leq p[i{+}1] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ p[i{+}1] \leq p[i] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\ \ \wedge\ |p[i{+}1] - p[i]| < \delta \qquad\qquad \} \\
\mathcal{S}_p f & \triangleq & \sum_{1}^{\mathit{len}(p)-1} [\, i \in \mathbf{N} \mapsto\ (p[i{+}1] - p[i]) * (\, f[\, p[i]\,] + f[\, p[i{+}1]\,]\,)\,/\,2\,] \\
\int_a^b f & \triangleq & \textbf{choose}\ r :\ \wedge\ r \in \mathbf{R} \\
 & & \qquad\qquad\qquad \wedge\ \forall\, \epsilon \in \mathbf{R}^{+} :\ \exists\, \delta \in \mathbf{R}^{+} : \forall\, p \in \mathcal{P}_a^b(\delta) : |\, r - \mathcal{S}_p f\,| < \epsilon
\end{array}
$$

Fig. 3. The definition of the Riemann integral.

The next-state relation is described by a formula relating the old and new values of the variables. For program *Increment*, there are two possibilities: either x is incremented by 1 and y is unchanged, or y is incremented by 1 and x is unchanged. Letting unprimed variables denote old values and primed variables denote new values, the next-state relation $\mathcal{N}$ is defined by

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & \vee\ (x' = x + 1)\ \wedge\ (y' = y) \\
 & & \vee\ (y' = y + 1)\ \wedge\ (x' = x)
\end{array}
$$

Formula $\mathcal{N}$ defines a relation between a pair of states. Such a formula is called an *action*. Formally, the meaning $[\![\mathcal{N}]\!]$ of action $\mathcal{N}$ is a Boolean-valued function on pairs of states, where $[\![\mathcal{N}]\!](s,\, t)$ equals *true* iff formula $\mathcal{N}$ holds when x and y are replaced by the values of x and y in state s, and x' and y' are replaced by the values of x and y in state t. We can generalize ordinary temporal logic to include actions by defining $[\![\mathcal{A}]\!]$, for any action $\mathcal{A}$, to be true of a sequence iff it is true of the first pair of states:

$$
[\![\mathcal{A}]\!](s_0,\, s_1,\, \ldots) \ \triangleq\ \mathcal{A}(s_0,\, s_1)
$$

Ignoring fairness for now, the obvious way to specify program *Increment* is by the formula $\mathit{Init} \wedge \Box\mathcal{N}$, which asserts of a behavior that the initial state satisfies *Init* and every step (successive pair of states) satisfies $\mathcal{N}$. However, this formula is not invariant under stuttering. To obtain invariance under stuttering, we allow steps that leave x and y unchanged. The assertion that x and y are unchanged can be expressed as $\langle x, y\rangle' = \langle x, y\rangle$, where v' denotes the expression v with all variables primed. Defining

$$[\mathcal{N}]_v \;\triangleq\; \mathcal{N} \lor (v' = v)$$

we can specify program *Increment* (with no fairness requirement) by the formula $Init \land \Box[\mathcal{N}]_{\langle x,\, y\rangle}$. This formula allows not only finite numbers of stuttering steps, but also infinite stuttering. It is satisfied by a behavior in which, after some finite number of steps, x and y stop changing. These behaviors must be ruled out by the fairness requirement.

Without knowing the semantics of **cobegin**, we cannot infer from the program text what the fairness requirement for program *Increment* should be. We make the requirement that both x and y should be incremented infinitely often. Since *infinitely often* is expressed in temporal logic by $\Box\Diamond$, the obvious way to express this requirement is

$$\Box\Diamond(x' = x + 1) \;\land\; \Box\Diamond(y' = y + 1)$$

However, it's a bad idea to write arbitrary temporal formulas as the fairness requirement, since that can easily lead to specifications that are not machine closed. We now show how to write machine-closed fairness requirements.

For any action $\mathcal{A}$, we define *Enabled* $\mathcal{A}$ to be the predicate that is true of a state iff an $\mathcal{A}$ step is possible starting in that state. The semantic definition is

$$[\![\mathit{Enabled}\ \mathcal{A}]\!](s) \;\triangleq\; \exists\, t : [\![\mathcal{A}]\!](s, t)$$

We then define weak and strong fairness for an action $\mathcal{A}$ to assert that if $\mathcal{A}$ remains continuously enabled (weak fairness) or repeatedly enabled (strong fairness), then an $\mathcal{A}$ step must eventually occur. The precise definitions are

$$\mathrm{WF}(\mathcal{A}) \;\triangleq\; \Box\Diamond\neg(\mathit{Enabled}\ \mathcal{A}) \;\lor\; \Box\Diamond\,\mathcal{A}$$

$$\mathrm{SF}(\mathcal{A}) \;\triangleq\; \Diamond\Box\neg(\mathit{Enabled}\ \mathcal{A}) \;\lor\; \Box\Diamond\,\mathcal{A}$$

However, there is one problem with these definitions: the formulas $\mathrm{WF}(\mathcal{A})$ and $\mathrm{SF}(\mathcal{A})$ are not invariant under stuttering. The proper definitions are the following, where $\langle\mathcal{A}\rangle_v$ is defined to equal $\mathcal{A} \land (v' \neq v)$, so an $\langle\mathcal{A}\rangle_v$ step is an $\mathcal{A}$ step that changes v.

$$\mathrm{WF}_v(\mathcal{A}) \;\triangleq\; \Box\Diamond\neg(\mathit{Enabled}\ \langle\mathcal{A}\rangle_v) \;\lor\; \Box\Diamond\,\langle\mathcal{A}\rangle_v$$

$$\mathrm{SF}_v(\mathcal{A}) \;\triangleq\; \Diamond\Box\neg(\mathit{Enabled}\ \langle\mathcal{A}\rangle_v) \;\lor\; \Box\Diamond\,\langle\mathcal{A}\rangle_v$$

Usually, v is the tuple of all variables and $\mathcal{A}$ is defined so any $\mathcal{A}$ step changes v, making $\langle\mathcal{A}\rangle_v$ equal to $\mathcal{A}$.

We can finish the specification of program *Increment* by adding weak fairness requirements for the actions of incrementing x and incrementing y. The complete specification Π appears in Figure 4. Formula Π has the *canonical form* of a specification: $Init \land \Box[\mathcal{N}]_v \land L$, where $Init$ is the initial predicate, $\mathcal{N}$ is the next-state action, v is the tuple of all relevant variables, and L is the conjunctions of formulas of the form $\mathrm{WF}_v(\mathcal{A})$ and/or $\mathrm{SF}_v(\mathcal{A})$.[1] It can be shown that if each

[1] More generally, the formula may be preceded by quantifiers $\exists\!\!\exists$ to hide internal variables.

action $\mathcal{A}$ implies the next-state action $\mathcal{N}$, then the pair $(\mathit{Init} \wedge \Box[\mathcal{N}]_v,\, L)$ is machine closed [2].

$$
\begin{array}{lcl}
\mathit{Init} & \triangleq & (x = 0) \wedge (y = 0) \\
\mathcal{X} & \triangleq & (x' = x + 1) \wedge (y' = y) \\
\mathcal{Y} & \triangleq & (y' = y + 1) \wedge (x' = x) \\
\mathcal{N} & \triangleq & \mathcal{X} \vee \mathcal{Y} \\
\mathit{\Pi} & \triangleq & \mathit{Init} \wedge \Box[\mathcal{N}]_{\langle x,\, y\rangle} \wedge \mathrm{WF}_{\langle x,\, y\rangle}(\mathcal{X}) \wedge \mathrm{WF}_{\langle x,\, y\rangle}(\mathcal{Y})
\end{array}
$$

Fig. 4. The complete specification $\mathit{\Pi}$ of program *Increment*.

5.3 TLA

Syntax and Semantics

I have augmented everyday mathematics with some new operators to get a kind of mathematics called the Temporal Logic of Actions (TLA for short). These new TLA operators can all be expressed in terms of $'$ (prime), $\Box$, and $\boldsymbol{\exists}$. The syntax and formal semantics of TLA are given in Figure 5. Missing from that figure are the syntax and semantics of state functions and actions (the $\langle \mathit{st\ fcn}\rangle$s and $\langle \mathit{action}\rangle$s of the figure). State functions and actions are written using the operators of Figure 2; their semantics are the semantics of everyday mathematics, which can be found in any standard treatment of set theory [40]. Since reasoning about programs requires reasoning about data structures, any verification method must have everyday mathematics embedded within it in some form.

The analog of Figure 5 for a programming language would be a complete syntax and semantics of every part of the language except expressions. I know of no programming language, except perhaps Unity, with as simple a semantics as TLA. Moreover, because TLA is mathematics, it has an elegance and power unmatched by any programming language.

Proofs

In a mathematical approach, there is no distinction between programs, specifications, and properties. They are all just mathematical formulas. Implementation is implication. Program $\mathit{\Pi}$ satisfies specification or property S iff every behavior that satisfies $\mathit{\Pi}$ also satisfies S. In other words, $\mathit{\Pi}$ satisfies S if and only if $\models \mathit{\Pi} \Rightarrow S$, where $\models F$ means that formula F is satisfied by all behaviors.

Writing programs and specifications as mathematical formulas conceptually simplifies verification. One does not have to extract verification conditions from a programming language; what has to be proved is already expressed as a mathematical formula. Consider the proof that a low-level specification $\mathit{\Pi}_1$ implements a high-level specification $\mathit{\Pi}_2$. The theorem to be proved is $\mathit{\Pi}_1 \Rightarrow \mathit{\Pi}_2$. When each

Syntax

$\langle \mathit{formula} \rangle$	$::=$	$\langle \mathit{pred} \rangle \mid \Box[\langle \mathit{action} \rangle]_{\langle \mathit{st\ fcn} \rangle} \mid \langle \mathit{formula} \rangle \wedge \langle \mathit{formula} \rangle \mid \neg \langle \mathit{formula} \rangle$ $\mid \Box \langle \mathit{formula} \rangle \mid \boldsymbol{\exists}\, \langle \mathit{var} \rangle : \langle \mathit{formula} \rangle$
$\langle \mathit{st\ fcn} \rangle$	$::=$	nonBoolean expression with unprimed variables
$\langle \mathit{action} \rangle$	$::=$	Boolean expression with primed and unprimed variables
$\langle \mathit{pred} \rangle$	$::=$	$\langle \mathit{action} \rangle$ with no primed variables

Semantics

$\llbracket f \rrbracket(s)$	$\triangleq f(\forall\, `v\text{'} : \llbracket v \rrbracket(s)/v)$	$\llbracket \neg F \rrbracket(\sigma)$	$\triangleq \neg \llbracket F \rrbracket(\sigma)$
$\llbracket f' \rrbracket(s, t)$	$\triangleq \llbracket f \rrbracket(t)$	$\models F$	$\triangleq \forall \sigma : \models \llbracket F \rrbracket(\sigma)$
$\llbracket \mathcal{A} \rrbracket(s, t)$	$\triangleq \mathcal{A}(\forall\, `v\text{'} : \llbracket v \rrbracket(s)/v,\ \llbracket v \rrbracket(t)/v')$	$\Diamond F$	$\triangleq \neg \Box \neg F$
$\models \mathcal{A}$	$\triangleq \forall s, t : \models \llbracket \mathcal{A} \rrbracket(s, t)$	$[\mathcal{A}]_f$	$\triangleq \mathcal{A} \vee (f' = f)$
$\llbracket \mathit{Enabled}\ \mathcal{A} \rrbracket(s)$	$\triangleq \exists\, t : \llbracket \mathcal{A} \rrbracket(s, t)$	$\Diamond \langle \mathcal{A} \rangle_v$	$\triangleq \neg \Box [\neg \mathcal{A}]_v$
$\llbracket \mathcal{A} \rrbracket(s_0, s_1, \ldots)$	$\triangleq \llbracket \mathcal{A} \rrbracket(s_0, s_1)$	$\mathrm{WF}_v(\mathcal{A})$	$\triangleq \Box\Diamond\neg(\mathit{Enabled}\ \mathcal{A}) \vee \Box\Diamond\langle \mathcal{A} \rangle_v$
$\llbracket F \wedge G \rrbracket(\sigma)$	$\triangleq \llbracket F \rrbracket(\sigma) \wedge \llbracket G \rrbracket(\sigma)$	$\mathrm{SF}_v(\mathcal{A})$	$\triangleq \Diamond\Box\neg(\mathit{Enabled}\ \mathcal{A}) \vee \Box\Diamond\langle \mathcal{A} \rangle_v$

$$\llbracket \Box F \rrbracket(s_0, s_1, \ldots) \triangleq \forall n : \llbracket F \rrbracket(s_n, s_{n+1}, \ldots)$$

$$\llbracket \boldsymbol{\exists}\, x : F \rrbracket(\sigma) \triangleq \exists \rho, \tau : (\natural\sigma = \natural\rho) \wedge (\rho =_x \tau) \wedge \llbracket F \rrbracket(\tau)$$

$$\natural(s_0, s_1, \ldots) \triangleq \textbf{if } s_0 = s_1 \textbf{ then if } \forall n : s_n = s_0 \textbf{ then } s_0 \textbf{ else } s_1, s_2, \ldots \quad \textbf{else } s_0, \natural(s_1, s_2, \ldots)$$

$$(s_0, s_1, \ldots) =_x (t_0, t_1 \ldots) \triangleq \forall `v\text{'} \neq `x\text{'} : \forall n : \llbracket v \rrbracket(s_n) = \llbracket v \rrbracket(t_n)$$

Notation

f is a $\langle \mathit{st\ fcn} \rangle$	$s, t, s_0, t_0, \ldots$ are states
$\mathcal{A}$ is an $\langle \mathit{action} \rangle$	σ, ρ, τ are infinite sequences of states
F, G are $\langle \mathit{formula} \rangle$s	$(\forall\, `v\text{'} : \ldots/v, \ldots/v')$ denotes substitution for all variables v
x is a variable	

Fig. 5. The syntax and formal semantics of the Temporal Logic of Actions (TLA).

Π_i is written in canonical form as $\mathit{Init}_i \wedge \Box[\mathcal{N}_i]_v \wedge L_i$, the proof has the structure shown in Figure 6. The three high-level steps correspond to the three steps described in Section 2.2:

1. Every possible initial state of Π_1 is a possible initial state of Π_2.
2. Step simulation.
3. The fairness requirement of Π_1 implies the fairness requirement of Π_2.

Step 1 and steps 2.1.1, 2.1.2, and 2.2.1 (the "leaves" in the proof of step 2) involve no temporal operators. They are proved by everyday math, where v and v' are treated as separate variables, for each variable v. Steps 1 and 2.1.1 are usually easy. The hard part of the proof is finding the invariant Inv and proving 2.1.2 and 2.2.1. The structure of the formulas to be proved leads to a further decomposition of the proof. For example, $\mathcal{N}_1$ is usually written as a disjunction $\mathcal{N}_1^1 \vee \ldots \vee \mathcal{N}_1^n$, allowing the proof of step 2.1.2 to be broken into the following steps.

Theorem: $Init_1 \land \Box[\mathcal{N}_1]_v \land L_1 \Rightarrow Init_2 \land \Box[\mathcal{N}_2]_v \land L_2$

Proof: 1. $Init_1 \Rightarrow Init_2$

2. $Init_1 \land \Box[\mathcal{N}_1]_v \Rightarrow \Box[\mathcal{N}_2]_w$

2.1. $Init_1 \land \Box[\mathcal{N}_1]_v \Rightarrow \Box Inv$

2.1.1. $Init_1 \Rightarrow Inv$

2.1.2. $Inv \land [\mathcal{N}_1]_v \Rightarrow Inv'$

2.2. $\Box[\mathcal{N}_1]_v \land \Box Inv \Rightarrow \Box[\mathcal{N}_2]_w$

2.2.1. $Inv \land [\mathcal{N}_1]_v \Rightarrow [\mathcal{N}_2]_w$

3. $Init_1 \land \Box[\mathcal{N}_1]_v \land L_1 \Rightarrow L_2$

...

Fig. 6. The structure of the proof that specification Π_1 implements specification Π_2.

2.1.2. $Inv \land [\mathcal{N}_1]_v \Rightarrow Inv'$

2.1.2.1. $Inv \land \mathcal{N}_1^1 \Rightarrow Inv'$

...

2.1.2.n. $Inv \land \mathcal{N}_1^n \Rightarrow Inv'$

2.1.2.n+1. $Inv \land (v' = v) \Rightarrow Inv'$

The invariant Inv is usually written as a conjunction $Inv_1 \land \ldots \land Inv_m$, allowing a further decomposition of the proof as follows.

2.1.2.i. $Inv \land \mathcal{N}_1^i \Rightarrow Inv'$

2.1.2.i.1. $Inv \land \mathcal{N}_1^i \Rightarrow Inv_j'$

...

2.1.2.i.m. $Inv \land \mathcal{N}_1^i \Rightarrow Inv_j'$

This is precisely the decomposition performed by the Owicki-Gries method. In that method, $\mathcal{N}_1^i$ corresponds to an individual program statement and Inv_j equals $at(\pi_j) \Rightarrow P_j$, where $at(\pi_j)$ asserts that control is at point π_j and P_j is the assertion attached to that control point.

In general, specifications may have internal variables. For simplicity, assume that each specification has a single internal variable; the generalization to arbitrary numbers of internal variables is easy. Proving that one specification implements another then requires proving a formula of the form $\models (\exists\, x : \Pi_1) \Rightarrow (\exists\, y : \Pi_2)$, where the Π_i are as above. By simple logic, this is equivalent to proving $\models \Pi_1 \Rightarrow (\exists\, y : \Pi_2)$, assuming the variable x does not occur in Π_2. To prove this formula, it suffices to prove $\models \Pi_1 \Rightarrow \Pi_2[f/y]$ for some expression f, where $\Pi_2[f/y]$ denotes the formula obtained by substituting f for the variable y. This is the same kind of formula whose proof is outlined in Figure 6. The expression f is called a refinement mapping [1].

Composition

One advantage of writing specifications as mathematical formulas is that they can be manipulated with simple mathematical laws. For example, let $\mathcal{X}$ and $\mathcal{Y}$

be defined as in Figure 4, and let

$$\begin{aligned}
\Pi_x &\triangleq (x = 0) \wedge \Box[\mathcal{X}]_x \wedge \mathrm{WF}_x(\mathcal{X}) \\
\Pi_y &\triangleq (y = 0) \wedge \Box[\mathcal{Y}]_y \wedge \mathrm{WF}_y(\mathcal{Y})
\end{aligned}$$

Applying the temporal logic identity $(\Box F) \wedge (\Box G) \equiv \Box(F \wedge G)$ and observing that $[\mathcal{X}]_x \wedge [\mathcal{Y}]_y$ equals $[\mathcal{X} \vee \mathcal{Y}]_{\langle x,\, y\rangle}$, we can show that $\Pi_x \wedge \Pi_y$ is equivalent to Π.

Program *Increment*, which is specified by Π, can be viewed as the composition of two processes, each incrementing one of the variables. Formulas Π_x and Π_y are the specifications of these processes. This example illustrates the general principle that, in the mathematical approach, composition is conjunction. There is no need to define a new parallel composition operator. A more detailed explanation of why composition is conjunction can be found in [3].

Real Time

To demonstrate the power of mathematics as a specification language, I will show how to write real-time specifications. As an example, I will add to program *Increment* the requirement that x must be incremented at least once every $\sqrt{2}$ seconds.

The usual approach to specifying real-time systems is to devise a real-time pseudo-programming language or a real-time temporal logic. A new language or logic means a new semantics, new proof rules, and new tools. It isn't a very comforting thought that, when faced with a new problem domain, one must redo everything. Using mathematics, we don't have to change or add anything; we just define what we need.

In mathematics, time is simply represented by a variable. So, we introduce a variable *now* to represent the current time. For simplicity, we pretend that incrementing x and y takes no time, which means that x and y don't change when *now* does. We begin by writing a formula *RT* asserting that *now* is a nondecreasing real number that gets arbitrarily large, and that x and y don't change when *now* does. The formula *RT* has the canonical form $\mathit{Init} \wedge \Box[\mathcal{N}]_v \wedge L$, where

- The initial predicate *Init* asserts that *now* is a real number.
- The next-state relation $\mathcal{N}$ asserts that the new value of *now* is a real number greater than its old value, and x and y are left unchanged.
- v equals *now*, so $[\mathcal{N}]_v$ allows steps that leave *now* unchanged.
- L asserts that *now* gets arbitrarily large.

Letting $\mathbf{R}$ be the set of real numbers and (r, ∞) be the set of all real numbers greater than r, and observing that $\mathrm{WF}_{now}(now' > r)$ implies that *now* is eventually greater than r, we can define *RT* by

$$\begin{aligned}
RT \;\triangleq\; & \wedge\ now \in \mathbf{R} \\
& \wedge\ \Box \begin{bmatrix} \wedge\ now' \in (now, \infty) \\ \wedge\ \langle x,\, y\rangle' = \langle x,\, y\rangle \end{bmatrix}_{now} \\
& \wedge\ \forall r \in \mathbf{R} : \mathrm{WF}_{now}(now' > r)
\end{aligned}$$

We next introduce a variable t to act as a timer, and define $MaxT$ to be a formula asserting that

1. t initially equals $\sqrt{2}$ plus the time when x was last incremented (where we consider x to be first incremented when the program is started).
2. now is always less than or equal to t.

These two conditions imply that x must be incremented at least once every $\sqrt{2}$ seconds. The second condition is easily expressed as $\Box(now \leq t)$. The first condition is expressed by a formula of the form $Init \wedge \Box[\mathcal{N}]_v$, where

- $Init$ asserts that t equals $\sqrt{2} + now$.
- $\mathcal{N}$ asserts that if the current step is an $\mathcal{X}$ step (one that increments x), then the step must make the new value of t equal to $\sqrt{2} + now$; otherwise the step must leave t unchanged.
- v is the tuple $\langle x, t\rangle$, so $[\mathcal{N}]_v$ allows steps that leave both x and t unchanged.

We can therefore define $MaxT$ by

$$
MaxT \triangleq \begin{array}{l} \wedge\ t = \sqrt{2} + now \\ \wedge\ \Box\left[\begin{array}{l} t' \ = \ \mathbf{if}\ \mathcal{X}\ \mathbf{then}\ \sqrt{2} + now \\ \qquad\qquad\quad \mathbf{else}\ \ t \end{array}\right]_{\langle x,\, t\rangle} \\ \wedge\ \Box(now \leq t) \end{array}
$$

Recall that Π, defined in Figure 4, is the formula representing the untimed version of program *Increment*. The formula representing the timed version is then

$$
\Pi \wedge RT \wedge \exists\!\!\exists\, t : MaxT
$$

which asserts of a behavior that it satisfies

- Π, so x and y are incremented as specified by the *Increment* program.
- RT, so now behaves the way real time should.
- $\exists\!\!\exists\, t : MaxT$, so x is incremented at least once every $\sqrt{2}$ seconds. (Note how t is hidden, so the formula describes how the values of x and now can change, but asserts nothing about the value of t.)

The approach used in this simple example is quite general. To write real-time specifications, one specifies the non-real-time aspects and then conjoins the real-time constraints. These constraints are expressed in terms of a few simple parametrized formulas [2].

Remarks

TLA is simple. All TLA formulas, such as the specification Π in Figure 4, can be expressed using only the operators $\wedge$, $\neg$, $\in$, **choose**, $'$ (prime), $\Box$, and $\exists\!\!\exists$. TLA is the basis for a complete specification language, called TLA$^+$ [29] that includes a module structure for writing large specifications.

When seeing a TLA specification like the one in Figure 4 for the first time, computer scientists are struck by what is unfamiliar about it. Typically, they are impelled to add some syntactic sugar to make the specification look more familiar. First, they observe that because one writes $x' = x + 1$ instead of the traditional assignment statement $x := x+1$, TLA forces one to write the explicit conjunct $y' = y$ to state that y is unchanged. So, they suggest writing something like an assignment statement to avoid having to say that other variables are unchanged. Next, they want to eliminate the temporal operators by just writing the initial condition and the actions. The next-state relation $\mathcal{N}$ and the $\Box[\mathcal{N}]_{\langle x, y \rangle}$ would be implicit. Instead of writing $\mathrm{WF}_{\langle x, y \rangle}(\mathcal{X}) \wedge \mathrm{WF}_{\langle x, y \rangle}(\mathcal{Y})$, the actions $\mathcal{X}$ and $\mathcal{Y}$ would be marked in some way.

To see how useful this syntactic sugaring would really be, consider the TLA$^+$ specification of a switch for a local area network. This specification was mentioned in Section 3.6, and a portion of it translated into pseudoPascal appeared in Figure 1. The specification is about 700 lines long. Ten of the shorter of those lines are devoted to assertions that variables remain unchanged. Replacing the mathematician's "=" with the computer scientist's ":=" would reduce the length of the specification by about 1%. Making the $\Box[\mathcal{N}]_v$ implicit would make the specification .1% shorter. The fairness requirements represent about 4% of the specification. However, they are not expressible as simple fairness conditions on disjuncts of the next-state relation. Expressing them requires the full power of the WF and SF operators. While it is dangerous to generalize from a single example, it is clear that replacing the mathematical notation with programming-language notation would shorten a specification by only a few percent, and would seriously restrict the ability to express fairness conditions.

The nontemporal part—the initial condition and next-state relation together with their subsidiary definitions—forms about 95% of the switch specification. It consists entirely of simple, familiar mathematics: numbers, sequences, sets, and so on. TLA works in practice because 95% of a specification consists of everyday mathematics, and 95% of a proof consists of ordinary mathematical reasoning. Temporal operators and temporal reasoning are used only for what they do best—expressing and proving fairness properties.

6 Conclusion

It has been 18 years since Ashcroft introduced the use of an invariant for reasoning about concurrent programs. Thinking about concurrent programs in terms of invariants is now standard practice among good programmers. (Unfortunately, good programmers are probably still in the minority.)

Ashcroft's work spawned a plethora of formalisms for specifying and reasoning about concurrent programs. Regardless of the formalism used, there is usually only one way to prove the correctness of any particular algorithm. The precise formulation may differ, but the proof will be essentially the same no matter what formalism is used. In any sensible method, one reasons about the algorithm, not about its particular representation. There are nonsensical methods in which the representation is so cumbersome that it interferes with the proof, but these methods can (and should) be ignored.

Formalisms do differ in what they can express. Ones based directly on program texts can usually express only invariance and simple liveness properties. Other methods allow the specification and proof of a much more general class of properties. The methods that work in practice specify properties as abstract programs, with hidden internal variables. They allow one to prove that one abstract program implements another.

I believe that the best language for writing specifications is mathematics. Mathematics is extremely powerful because it has the most powerful abstraction mechanism ever invented—the definition. With programming languages, one needs different language constructs for different classes of system—message-passing primitives for communication systems, clock primitives for real-time systems, Riemann integrals for hybrid systems. With mathematics, no special-purpose constructs are necessary; we can define what we need.

Writing specifications as mathematical formulas conceptually simplifies verification, making the rigorous formalization of proofs needed for mechanical verification easier. A system for mechanically verifying TLA specifications is currently being developed [16].

Perhaps the greatest advantage of specifying with mathematics is that it allows us to describe systems the way we want to, without being constrained by ad hoc language constructs. Mathematical manipulation of specifications can yield new insight. A producer/consumer system can be written as the conjunction of two formulas, representing the producer and consumer processes. Simple mathematics allows us to rewrite the same specification as the conjunction of n formulas, each representing a single buffer element. We can view the system not only as the composition of a producer and a consumer process, but also as the composition of n buffer-element processes. Processes are not fundamental components of a system, but abstractions that we impose on it. This insight could not have come from writing specifications in a language whose basic component is the process.

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. Research Report 91, Digital Equipment Corporation Systems Research Center, 1992. An earlier version, without proofs, appeared in [13, pages 1–27].
3. Martín Abadi and Leslie Lamport. Conjoining specifications. To appear as an SRC Research Report, 1993.
4. M. W. Alford et al. *Distributed Systems: Methods and Tools for Specification*, chapter 5. Lecture Notes in Computer Science, 190. Springer-Verlag, 1985.
5. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

6. Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.
7. Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.
8. E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.
9. Manfred Broy. Algebraic and functional specification of an interactive serializable database interface. *Distributed Computing*, 6(1):5–18, 1992.
10. Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Enginnering and Methodology*, 2(1):1–46, 1993.
11. K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, Reading, Massachusetts, 1988.
12. Edmund M. Clarke, Jr., Orna Grumberg, and D. Long. Verification tools for finite-state concurrent systems. This volume.
13. J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992. Proceedings of a REX Real-Time Workshop, held in The Netherlands in June, 1991.
14. Edsger W. Dijkstra. A personal summary of the Gries-Owicki theory. In Edsger W. Dijkstra, editor, *Selected Writings on Computing: A Personal Perspective*, chapter EWD554, pages 188–199. Springer-Verlag, New York, Heidelberg, Berlin, 1982.
15. E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier, Amsterdam, 1990.
16. Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In *Computer-Aided Verification*, Lecture Notes in Computer Science, Berlin, Heidelberg, New York, June 1992. Springer-Verlag. Proceedings of the Fourth International Conference, CAV'92.
17. 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*, pages 184–192. IEEE, October 1978.
18. R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32. American Mathematical Society, 1967.
19. Nissim Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.
20. Rob Gerth and Willem P. de Roever. A proof system for concurrent ADA programs. *Science of Computer Programming*, 4(2):159–204, 1984.
21. David Harel and Amir Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and models of concurrent systems*, volume F13 of *NATO ASI Series*, pages 477–498. Springer-Verlag, 1985.
22. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.
23. Reino Kurki-Suonio. Operational specification with joint actions: Serializable databases. *Distributed Computing*, 6(1):19–37, 1992.
24. Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4):325–342, July 1984.
25. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

26. Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.
27. 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.
28. Leslie Lamport. The temporal logic of actions. Research Report 79, Digital Equipment Corporation, Systems Research Center, December 1991. To appear in *Transactions on Programming Languages and Systems*.
29. Leslie Lamport. Hybrid systems in TLA$^+$. In Robert L. Grossman, Anil Nerode, Hans Rischel, and Anders P. Ravn, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 77–102, Berlin, Heidelberg, 1993. Springer-Verlag.
30. A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach, New York, 1969.
31. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
32. Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Symposium on the Principles of Distributed Computing*, pages 137–151. ACM, August 1987.
33. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1991.
34. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.
35. Jayadev Misra. Specifying concurrent objects as communicating processes. *Science of Computer Programming*, 14(2–3):159–184, 1990.
36. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.
37. Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.
38. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.
39. Richard Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level reasoning. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 173–186. Association for Computing Machinery, August 1983.
40. J. R. Shoenfield. The axioms of set theory. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter B1, pages 317–344. North-Holland, Amsterdam, 1977.