INFORMATION PROCESSING 83, R.E.A. Mason (ed.)
Elsevier Science Publishers B.V. (North-Holland)
© IFIP, 1983

WHAT GOOD IS TEMPORAL LOGIC?

Leslie LAMPORT*
Computer Science Laboratory, SRI International
Menlo Park, California 94025, USA

Invited Paper

Temporal logic is a formal system for specifying and reasoning about concurrent programs. It provides a uniform framework for describing a system at any level of abstraction, thereby supporting hierarchical specification and verification.

1. INTRODUCTION

1.1. The Answer

The question posed in the title has a simple answer: temporal logic is a good method for specifying and reasoning about a concurrent program. The purpose of this paper is to explain why. It is primarily about my method for using temporal logic, but also discusses some other approaches.

Temporal logic is a branch of formal logic that is new to many of you. When advocating a new formalism, one usually argues that it is simple, natural and easy to use. Temporal logic is a rather simple extension of classical logic, and it does provide a natural way of describing the temporal behavior of a program. However, learning to use a new formalism is never easy because it requires learning to think in a new way. You probably found it hard at first to use the $\forall$ and $\exists$ of predicate calculus; only after changing your way of thinking did the formalism become simple and natural. Learning to reason about programs in terms of temporal logic is just as hard as learning to use $\forall$ and $\exists$. Moreover, there are other formalisms that at first seem more natural than temporal logic; you would probably find them easier to learn.

If temporal logic is not simpler and more natural than other methods, why should we use it? Although there are many reasons why I like temporal logic, there is one that is paramount: temporal logic supports hierarchical specification and reasoning in a simple, natural way. Experience has shown that the best way to describe a complex system is through a hierarchy of levels of abstraction, starting from a high-level specification and ending with the implementation in some programming language. Each level is a specification of the next lower level and an implementation of the next higher one. Temporal logic provides a single logical system for describing the program at any level of abstraction—from the highest-level specification through the programming-language implementation. A statement about the program at one level is a meaningful statement about any lower level. Thus, hierarchical design methods are supported directly, with no extra mechanism needed to link the different levels of description.

*This work was supported in part by the National Science Foundation under grants MCS-7816783 and MCS-8104459.

The remainder of this section explains in vague generalities how temporal logic differs from other approaches. Subsequent sections describe what temporal logic is, how it is used to specify programs, and how it supports hierarchical specification.

1.2. How Do You Say "Maybe" in Temporal Logic?

Natural languages are very expressive and very imprecise. You can express any property of a program in English—for example, you can say that it is cute—but it is hard to be sure that someone else will understand exactly what you mean. Formal languages are not very expressive, but they are precise. You can't say very much in a formal language, but what you can say is completely unambiguous.

A temporal logic is a formal language for expressing temporal properties. A number of different temporal logics have been studied by logicians and computer scientists [20], but there is one that I find useful for specifying and reasoning about concurrent programs. This temporal logic allows you to express two kinds of properties of a program.

- *Safety properties*, which assert that the program does not do something bad.
- *Liveness properties*, which assert that the program does eventually do something good.

Partial correctness (the program does not produce the wrong answer), mutual exclusion (two processes are not in their critical sections at the same time), and deadlock-freedom (the program does not reach a deadlocked state) are safety properties. Termination (the program eventually does terminate) and starvation-freedom (a process eventually receives service) are liveness properties.

Safety and liveness are not the only properties of a program that are of interest. For example, one might like to know the program's probability of terminating, or whether it is cute. However, I do not know of any practical method for reasoning about such properties. I do know how to reason about safety and liveness properties, and those are the only ones that I consider.

One type of property that is not of interest is the possibility that the program might do something. We want to specify not that the program *might* produce the right answer, but that it *must* do so. Because many formalisms cannot express liveness properties, they have led people to consider such "possibility" properties instead. If you can't prove that the program must do something, proving that it might do it is the next best thing. Since temporal logic can express liveness, we do not need to talk about possibilities. In fact, the temporal logic I use cannot even express such properties; you can't say "maybe" in this temporal logic.

Temporal logic was developed to describe the order in which things must happen rather than the actual times at which they happen. To talk about real time, one can introduce the notion of a clock. Real-time properties of programs can then be expressed as safety properties; for example, the assertion "the program must respond to an input within ten milliseconds" can be expressed as "the clock cannot reach ten milliseconds after the time of the last input without an output being produced", which is a safety property. Liveness properties, which state that something must eventually happen, follow from the physical nature of time—the ten milliseconds must eventually elapse. A somewhat different method of applying temporal logic to real-time programs was used in [2], where the number of program steps executed replaces real time.

1.3. Axioms or Models?

There are two ways to specify a program:

- Describe an abstract model that tells how the program should behave.
- State what properties the program should have.

I will call the first approach *constructive* and the second *axiomatic*. Since one can describe an abstract model by stating its properties, there is no way to formalize this distinction. Nevertheless, it is useful to classify methods in this way. Perhaps the most elegant example of a constructive method is Milner's Calculus of Communicating Systems [15]. Temporal logic is an axiomatic method.

Hierarchical specification poses a problem for constructive methods. As an example, consider the specification of a queue. At a high level, the queue operations consist of adding an element to or removing one from the queue. A constructive specification uses a model in which these are the elementary operations. At the implementation level, each queue operation is performed by executing many program steps. The individual program steps are this level's elementary operations; they are the elementary operations in a constructive specification of the implementation. What does it mean for the program to correctly implement the specification? In the constructive approach, the answer requires a definition of what it means for a sequence of lower-level operations to correctly implement a single high-level operation. This may not be easy, since actions from other parts of the program may be executed concurrently with the actions that implement the queue operation. Section 4 explains how temporal logic specifications avoid this problem.

There is another problem facing constructive methods. They specify a program essentially by writing another, presumably simpler, program. However, concurrent programs that seem simple can exhibit quite unexpected behavior. How can we be sure that we understand the program if its specification says only that it behaves like some other program which we may not understand?

Axiomatic methods attempt to overcome this difficulty by stating directly what properties the program must have. They will succeed only if the properties are easy to understand. The challenge facing the axiomatic approach is therefore to express the required properties so they are both precise and understandable.

1.4. Operations or States?

In an axiomatic specification, one expresses the properties that the program must have as assertions in some formal logical system. What should these assertions talk about? Two possibilities arise:

- What operations are performed by the program.
- How the state of the program changes.

At first, talking about operations seems the more natural approach. However, the work of Floyd and Hoare on sequential program verification demonstrated the benefits of thinking in terms of states rather than operations. Hence, methods were developed for state-based, assertional reasoning about concurrent programs [8], [17].

In the seven or eight years I have used assertional methods to reason about concurrent programs, the following has occurred over and over again. First, I find a proof in terms of the sequence of operations executed by the program. The proof seems natural, and there appears to be no nice way of expressing it in terms of assertions about states. However, I persevere, and one of two things happens: I either find an assertional proof that is simpler and more elegant than the operational one—a proof that sheds new light on why the program works—or else I discover an error in the proof and a corresponding bug in the algorithm. The latter happens quite often; it is easy to make a mistake when reasoning about the sequence of operations executed by a concurrent program.

I have had a similar experience specifying properties of programs. When describing a program informally, I tend to speak in terms of operations rather than states—*i.e.*, in terms of verbs rather than nouns. Saying that the program displays a number on the terminal seems more natural than saying that it changes the terminal's state. However, when I try to specify something more precisely, I find myself talking about states. Trying to say exactly what it means to display a number on a terminal, I am led to talk about the state of the screen.

This experience made me decide to base an axiomatic system for describing concurrent programs upon states rather than operations.

1.5. Why Temporal Logic?

Having decided upon a state-based axiomatic method, why should I use temporal logic? People have been specifying and reasoning about sequential programs for years using ordinary, nontemporal forms of logic. Why does the introduction of concurrency require the use of temporal reasoning?

To describe a sequential program, we need consider the state at only two times: before the program is executed and afterwards. The program can be considered as a function from an initial to a final state, so it can be specified in terms of input and output conditions. The simple before/after temporal reasoning that is used need not be made explicit.

This is not true for concurrent programs. For example, consider the following two program statements, where the angle brackets denote atomic operations. (An atomic operation of a concurrent program is one that is indivisible with respect to concurrently executed operations.)

```
S₁: ⟨ x := x + 1 ⟩

S₂: begin
        ⟨ x := x + y + 1 ⟩;
        ⟨ x := x − y ⟩
    end
```

Statements S_1 and S_2 produce the same mapping from initial to final states. They satisfy exactly the same input and output conditions, so they are completely equivalent when used in a sequential program. However, consider the following concurrent program, where the **cobegin** indicates that its two clauses are to be executed concurrently.

cobegin $\langle\, y := y - 7 \,\rangle \;\square\; S$ **coend**

Substituting S_1 for S yields a program that increments x by one, while substituting S_2 for S yields a program that increments x by either one or eight—the latter possibility occurring if the statement $\langle\, y := y - 7 \,\rangle$ is executed between the execution of the two statements of S_2. Hence, S_1 and S_2, which satisfy identical input/output conditions, are not equivalent when they appear as part of a concurrent program.

This example shows that, when describing a concurrent program, we cannot restrict our attention to what is true before and after its execution; we must also consider what happens *during* its execution. To prove simple safety properties, one need consider only properties that are true at all times during the program's execution. Since "at all times" is also a simple temporal concept, one can reason about these safety properties with no explicit temporal reasoning, as in [9] and [17]. However, the more complex temporal reasoning needed for liveness properties cannot be left implicit; it must be explicitly described.

Pnueli was the first to recognize the usefulness of temporal logic for reasoning about liveness properties of concurrent programs [19], although Burstall had earlier suggested applying temporal logic to program verification [1].

1.6. What Isn't Temporal Logic Good For?

Like any formal system, temporal logic has its limitations. One limitation is obvious, but I will state it anyway: temporal logic will not make the specification and verification of concurrent programs easy. Designing a concurrent program is a difficult task; no formalism can make it easy. What temporal logic does provide is a method for specifying precisely what the program should do and analyzing rigorously what it will do. This method can be applied with varying degrees of rigor, ranging from an informal specification that serves only to guide the designer, to a completely formal specification with a machine-verified correctness proof of the implementation. Rigorous reasoning in any formal system is difficult and time consuming, but it is the only way I know to eliminate the subtle timing-dependent errors that are endemic to concurrent programs.

Our temporal logic also has less obvious limitations. Its expressiveness has been carefully limited so it is just adequate for its intended purpose: specifying and reasoning about a concurrent program. Its limited expressiveness is what makes temporal logic so good for this purpose because it prevents one from saying things that could get him into trouble. However, it means that our temporal logic is worthless for many other purposes. For example, it cannot express the statement that two programs are equivalent. This is because two programs are equivalent if they have the same set of possible behaviors, and our temporal logic cannot express the notion of possibility.

In general, temporal logic is not good for comparing programs or describing sets of programs. When specifying and reasoning about a single program, the only relevant comparison between programs is that a lower-level version correctly implements a higher-level one. There is only one program that concerns us: the implementation running on the computer. It is this single program that one specifies and reasons about with temporal logic.

2. WHAT IS TEMPORAL LOGIC?

2.1. What Is Temporal Logic Talking About?

Viewed formally, temporal logic consists of a language of formulas and a set of axioms and inference rules for determining which formulas are theorems. However, a formal description does not tell us what temporal logic is really about. Just as Peano's axioms would be meaningless if we didn't know they were about the integers, temporal logic is meaningless without an understanding of the models that underlie it.

We model a program as a set of (finite or infinite) behaviors of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots ,$$

where the s_i are *states* and the α_i are *actions*. For now, think of each behavior as a possible execution of some single program written in a Pascal-like language with constructs for expressing concurrency. For the sake of euphony, I will refer to this Pascal-like language simply as "Pascal".

A state consists of a possible "snapshot" taken in the middle of an execution, containing all the information needed to resume execution right from that point. Thus, the state must describe the value of each program variable, the contents of the subroutine-calling stack, the value of each "program counter", *etc.* An action is a specific atomic action of the program—for example, the execution of some particular atomic assignment statement. The behaviors in the model represent all possible executions of the program.

We think of the behavior

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots ,$$

as describing an execution that starts at time zero in state s_0, is in state s_1 at time one, *etc.* Time zero is regarded as the present, all later times as the future.

Observe that we are modeling concurrency as the interleaving of atomic actions. Computer scientists often feel that something is lost by sequentializing a concurrent program in this way, and that one should instead use a partial ordering among the actions. However, so long as we are talking only about safety and liveness properties, there is no loss of generality in considering totally ordered sequences of actions. Our model includes all possible sequences, and a partial ordering is completely equivalent to the set of all total orderings consistent with it. The real assumption implicit in the model is the existence of atomic actions. Since this assumption is made in virtually all formal models of concurrency, I will not discuss it here. As you will see, temporal logic requires only that at some sufficiently low level—perhaps at the level of microcode —the program's execution be accurately described by a collection of atomic actions.

2.2. How Do You Speak Temporal Logic?

It would be appropriate here to define exactly what the formulas of temporal logic are, and how each formula is interpreted as a statement about the underlying model. However, rather than bore you with formalism, I will describe the language of temporal logic informally. A more formal definition can be found in the appendix of [11].

In the temporal logic that I use, assertions are constructed from state predicates, action predicates, the usual logical operators, and the single temporal operator $\trianglelefteq$. A state predicate is an assertion about a state, a typical example being

$$(x > 0) \wedge (\text{control at program location } l) ,$$

which asserts that the state is one in which: (i) the value of the variable x is positive and (ii) the program counter indicates that control is at a certain location l. This assertion is true for some states and false for others.

An action predicate is an assertion about an action. The action predicates we need have the form

$$\text{this is an action of } \pi ,$$

where π is some part of the program, such as a particular subroutine or **while** loop. Although needed to formalize the concepts, action predicates play a minor role in specifying and reasoning about a program. For simplicity, they will be largely ignored from now on.

Temporal relations are expressed with the operator $\trianglelefteq$. The formula $A \trianglelefteq B$ is interpreted to mean that B remains true at least as long as A does. You can remember this by thinking of $\trianglelefteq$ as a temporal $\leq$ operator, where $A \trianglelefteq B$ means that the length of time for which A holds continuously is $\leq$ the length of time that B holds. Thus, $(x = 6) \trianglelefteq (y > 7)$ asserts that if x has the value 6 from time 0 through time 27, then the value of y is greater than 7 from time 0 through time 27. If the value of x becomes different from 6 at time 28, then this formula asserts nothing about the value of y from time 28 on. The $\trianglelefteq$ operator is related to the more usual **until** operator by $A \trianglelefteq B \equiv B$ **until** $\sim A$.

An important type of temporal formula is one of the form $true \trianglelefteq A$. It asserts that A remains true at least as long as $true$ does—in other words, A remains true forever. This assertion is written $\Box A$, and is read "henceforth A". Another important type of formula has the form $\sim \Box \sim A$ and asserts that it is not the case that A is always false—in other words, A must eventually become true. This assertion is written $\Diamond A$ and read "eventually A". (While most speakers of English would say that "A does not always remain false" is equivalent to "A eventually becomes true", some computer scientists disagree. If you are one of them, see [10] for an explanation.)

A temporal logic formula is interpreted as an assertion about a single behavior of the program. It may be true of some behaviors and false of others. We say that the formula is satisfied by a program, or that it is true for the program, if it is true of every behavior in our model of the program. A temporal logic specification consists of a collection of temporal formulas that the program must satisfy—*i.e.*, formulas that must be true for every possible program behavior.

2.3. What Did You S...S...S...Say?

While the exact form of our temporal logic formulas need not concern us here, they have one crucial property that must be discussed: they cannot detect "stuttering". To see what this means, let σ be the behavior

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots ,$$

and let σ' be another behavior which is the same as σ except that some particular transition $s_{i-1} \xrightarrow{\alpha_i} s_i$ is replaced by the stuttering sequence

$$s_{i-1} \xrightarrow{\alpha_i} s_{i-1} \xrightarrow{\alpha_i} s_{i-1} \xrightarrow{\alpha_i} \cdots \xrightarrow{\alpha_i} s_{i-1} \xrightarrow{\alpha_i} s_i .$$

No temporal logic formula can distinguish between σ and σ'. In other words, any formula in our temporal logic is true for the behavior σ if and only if it is true for σ'. This fact is easily proved from the formal semantics given in [11]. A little experimentation should also convince you that it is impossible to construct an assertion from state predicates and the operator $\trianglelefteq$ that can distinguish between these two behaviors.

Not only can a temporal logic assertion not detect the addition of identical copies of a transition, but it cannot detect the replacement of a transition by a sequence of transitions if the state and action predicates from which it is constructed cannot distinguish between the original state and action and the interposed states and actions. More precisely, let σ be as above, but this time let σ' be the same as σ except with the transition $s_{i-1} \xrightarrow{\alpha_i} s_i$ replaced by the sequence

$$s^1_{i-1} \xrightarrow{\alpha^1_i} s^2_{i-1} \xrightarrow{\alpha^2_i} s^3_{i-1} \xrightarrow{\alpha^3_i} \cdots \xrightarrow{\alpha^{41}_i} s^{42}_{i-1} \xrightarrow{\alpha^{42}_i} s_i \, .$$

Let A be any temporal logic assertion such that none of the state predicates appearing in A can distinguish between s_{i-1} and the various s^j_{i-1}, and none of the action predicates appearing in A can distinguish between α_i and the various α^j_i. In other words, for every j: each state predicate appearing in A is true for the state s^j_{i-1} if and only if it is true for the state s_{i-1}, and similarly for each action predicate appearing in A. In this case, A is true for σ if and only if it is true for σ'.

When most computer scientists see that our temporal logic cannot detect stuttering, they want to overcome this deficiency by making the logic expressive. Many of them have done this—usually by adding a *next* operator, where *next* A means that A is true after the next action. I have very definite reasons for not adding a *next* operator or any other operator to increase the expressiveness of this temporal logic. We use an abstract logic instead of reasoning directly about the underlying model because the logic does not let us talk about irrelevant properties of the model. Peano's axioms prevent us from saying such things as "8 is rounder than 7" by giving us no means of expressing them. Using the *next* operator, one could write a specification for a queue that includes the requirement "putting an element in the queue should take exactly 17 steps". The number of steps in a Pascal implementation is not a meaningful concept when one gives an abstract, high level specification of a queue. It is just as irrelevant as the roundness of an integer, since it is a property of a specific representation of the operation, not of the operation itself.

When one talks about the next state, one really means the next state in which a significant change occurs—where significant means visible at the level of detail of the specification. For example, in saying that an element is added to the queue in the next state, one means that the element will be added to the queue before certain other parts of the state change. This can be expressed in temporal logic without using the *next* operator. It takes a little time to get used to expressing properties in this way, without the *next* operator, but it is worth it. You will see that increasing the expressiveness of our temporal logic with a *next* operator would destroy the entire logical foundation for its use in hierarchical methods.

3. SPECIFICATIONS

3.1. What Good is a Specification?

A specification serves as a contract between the implementer of a program module and the user of that module. This contract has two functions:

1. It allows the implementer to design the module, knowing that his only responsibility is to meet the requirements imposed by the specification.
2. It allows the user to write programs that call upon the module, knowing that he need worry only about the correctness of his programs, and can assume that the module will behave as specified.

How can one be sure that a specification method really serves these functions? How do we know that the specification tells the implementer and the user everything they need to know? In principle, we know that these functions are satisfied if

1. The implementer can formally prove that his implementation satisfies the specification.
2. The user, given only the specification of the module, with no knowledge of how it is implemented, can formally prove the correctness of his programs that use the module.

Even if complete formal verification is not attempted, the fact that a method permits this kind of logical reasoning ensures that it will produce precise, unambiguous specifications.

In an ideal world, a specification is written once, and the implementer and user need never talk to each other again. In practice, like any real contract, a specification must be renegotiated if the implementer realizes he has promised more than he can deliver, or if the user realizes that the specified module does not meet his needs. Contracts are no less useful if they are renegotiated. I believe that whenever a system is "too complicated to be formally specified"—its sole specification being several hundred thousand lines of Pascal code— it is because the system evolved without the implementer/user paradigm, so there was no one to negotiate on behalf of the users.

3.2. An Example

The method I advocate for writing temporal logic specifications is described in detail in [11]. In this section, I will present a rough sketch of the way it works by considering a simple example. Other specification methods based upon temporal logic are discussed at the end of the section.

The example is a FIFO (First-In-First-Out) queue. There are two operations performed on the queue: PUT, which inserts an element at the tail of the queue, and GET, which removes an element from the head of the queue. More precisely, PUT and GET are two subroutines that are called to perform these operations. They can be called concurrently by two different processes, but, for simplicity, I assume that each can be called by only one process at a time.

We must decide what GET should do if it finds the queue empty. In a sequential program, this is treated as an anomalous condition—for example, by returning a special error value. However, for a concurrent program, there is the additional possibility of waiting for another process to put something in the queue. This possibility is more interesting, since it exists only for concurrent programs, so we choose it.

You should be very suspicious when someone uses a queue to illustrate a specification method. Many methods have queues built in as primitive concepts, sometimes implicitly in the form of sequences, so it is not surprising that they can very easily specify a queue. A good way to judge whether the example is biased is to compare the specifications of different kinds of queues: FIFO versus LIFO (Last-In-First-Out), unbounded versus bounded, waiting on a GET to an empty queue versus returning an error value, *etc.* If two programs that have very similar informal descriptions require very different specifications, then the method may be difficult to use in practice.

3.3. State Functions

We want to think about programs in terms of states, so a specification should be written in terms of states. The basic object mentioned in a specification is the *state function*, which is a function that assigns a value to any program state. In specifying a queue, there are many ways of choosing the state functions, but the different choices I might make all produce specifications that are easily shown to be equivalent. Here, I will use the following three main state functions:

queue:	The current contents of the queue.
putarg:	When PUT is called, *putarg* equals the element to be inserted into the queue. After the element is put on the queue, *putarg* assumes a special value *NULL*. Thus, *putarg* serves to "remember" both the argument with which PUT is called and whether or not PUT has finished inserting an element into the queue.
getval:	The analogue of *putarg* for the GET operation. It equals *NULL* when GET is called, and equals the element removed by GET when *queue* is changed.

These descriptions of the state functions are not part of the specification; they are just comments to help you understand it. The specification simply declares *queue*, *putarg* and *getval* to be state functions, and indicates what type of values they may assume. Formally, the specification has the form

> there exist state functions *queue*, *putarg*, ... such that ....

Two other, less interesting, state functions are needed: *in*(PUT) and *in*(GET). These are boolean-valued state functions that are true when control is inside a call to the PUT and GET subroutines, respectively. They are thus functions of "program counter" values. A few other state functions are needed to specify how arguments and values are passed in calls to PUT and GET, but I will ignore them.

3.4. Specifying Safety Properties

Recall that an axiomatic specification consists of a list of properties, also called axioms, each of which must be satisfied by every possible behavior of the implementation. I will describe only the interesting axioms for the queue, omitting the initial conditions and some simple safety properties that describe the passing of arguments and values in subroutine calls. The complete specification, which is not very long, can be found in [11].

The only interesting safety property is one that describes how the three major state functions are allowed to change. There are two occasions on which *queue* can change:

(a) When PUT adds the current value of *putarg* to the end of the queue, setting *putarg* to *NULL*.

(b) When GET removes an element from the front of the queue, changing the value of *getval* from *NULL* to that element.

These are also the only interesting changes that can occur to *putarg* and *getval*. More precisely, (a) describes the only time *putarg* can change when control is in the PUT subroutine and (b) describes the only time *getval* can change when control is in the GET subroutine.

This restriction on when these three state functions are allowed to change is expressed more or less as follows, where $*$ denotes concatenation. (The notation of [11] has been changed a bit to make the property less compact and easier to read.)

allowed changes to *queue*
putarg **when** *in*(PUT)
getval **when** *in*(GET)

(a) $in(\mathrm{PUT})_{old} \wedge (putarg_{old} \neq NULL) \rightarrow$
$(putarg_{new} = NULL) \wedge (queue_{new} = queue_{old} * putarg_{old})$

(b) $in(\mathrm{GET})_{old} \wedge (getval_{old} = NULL) \wedge (queue_{old}\text{ not empty})$
$\rightarrow$
$(getval_{new} \neq NULL) \wedge (queue_{old} = getval_{new} * queue_{new})$

This property states that for any transition $s_{i-1} \xrightarrow{\alpha_i} s_i$ in the behavior

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots ,$$

if one of the following holds:

- the value of the state function *queue* is not the same in states s_{i-1} and s_i, or
- the value of *putarg* is not the same in s_{i-1} and s_i, and *in*(PUT) is true for state s_{i-1}, or
- the value of *getval* is not the same in s_{i-1} and s_i, and *in*(GET) is true for state s_{i-1},

then the transition must be either of type (a) or type (b). It is a type (a) transition if the predicate $in(PUT) \wedge (putarg \neq NULL)$ is true for state s_{i-1}, *putarg* equals *NULL* in state s_i, and the value of *queue* in state s_i equals the concatenation of its value in state s_{i-1} with the value of *putarg* in state s_{i-1}. It is a type (b) transition if the predicate $in(GET) \land (getval = NULL) \land queue$ not empty is true for state s_{i-1}, *getval* is not *NULL* in state s_i, and the value of *queue* in state s_{i-1} equals the value of $getval * queue$ in state s_i.

To specify a LIFO queue, one requires that the new element be added to the head rather than the tail of the queue, which is done by simply replacing $queue_{old} * putarg_{old}$ in (a) with $putarg_{old} * queue_{old}$. A queue of maximum length 23 is specified by adding the condition

$$\text{length of } queue_{old} < 23$$

to the "precondition" of (a). To allow the GET subroutine to return a special value *EMPTY* when the queue is empty, one just adds the following allowed transition.

(c) $in(\text{GET})_{old} \land (getval_{old} = NULL) \land (queue_{old}\ \text{empty})$
$\rightarrow getval_{new} = EMPTY$

Our **allowed changes** property states that *queue* changes atomically when an element is added to or removed from the queue by the PUT and GET operations. It looks just like a constructive specification which says that adding or removing a queue is a single atomic action. If this is so, how can the property be valid for a Pascal implementation in which the PUT subroutine requires 42 program steps to add an element to the queue?

Although the syntax of the **allowed changes** statement makes it look like a constructive specification, it is actually a temporal logic formula that can be expressed using only the temporal operator $\trianglelefteq$. Writing it as such a formula is not easy (the general method is given in [11]), but we never have to; we just reason directly about the **allowed changes** statement. Being a temporal logic formula, it cannot distinguish between the transition $s_{i-1} \xrightarrow{\alpha_i} s_i$ and the sequence of transitions

$$s_{i-1}^1 \xrightarrow{\alpha_i^1} s_{i-1}^2 \xrightarrow{\alpha_i^2} s_{i-1}^3 \xrightarrow{\alpha_i^3} \cdots \xrightarrow{\alpha_i^{41}} s_{i-1}^{42} \xrightarrow{\alpha_i^{42}} s_i$$

when the state functions that appear in it—namely, *queue*, *putarg*, *getval*, *in*(PUT) and *in*(GET)— have the same value for each state s_{i-1}^j as they do for s_{i-1}.

Although the **allowed changes** property asserts that the state function *queue* must change atomically, it does not say anything about how many steps it takes the PUT subroutine to perform the operation. Remember that *queue* is not a program variable; it is a state function whose value is a function of the values of program variables, program counters, *etc*. Each of the PUT subroutine's 42 program steps may change variable and program counter values. However, the state functions *queue* and *putarg* must be defined in such a way that they are changed by only one of those steps.

To prove the correctness of an implementation, one must define the state functions *queue*, *putarg* and *getval* so they change atomically even if the program moves data around in small pieces. This may seem like magic, but it really isn't hard. You can see how it is done in [11], where these state functions are defined for an implementation in which the data are moved one bit at a time. The idea does not seem so strange if we remember that all assertional methods for proving safety properties involve finding an *invariant*—a boolean state function whose value does not change during program execution. Instead of finding state functions that do not change at all, verifying the specification requires finding state functions that change only the way the **allowed changes** statement permits them to. In principle, these state functions can always be defined—in fact, there are usually many different ways of doing so. In practice, the appropriate definitions are just as hard to find as the invariants for proving safety properties. If the program is complex, or if its correctness is based upon subtle interactions among the processes, proving its correctness will not be easy. No method will make it easy to prove the correctness of a complex concurrent program.

I have indicated how this method of specifying safety properties fulfills the first function of a specification: allowing one to prove the correctness of an implementation. How does it achieve the second function: allowing one to prove safety properties of programs that call the PUT and GET subroutines? Proving safety properties requires finding invariants. The invariants for programs that use PUT and GET will be functions not only of the program's variables and program counter values, but of the state functions *queue*, *putarg* and *getval* as well. The **allowed changes** statement in the queue module's specification, which constrains how these state functions can change, will be used in the proof of invariance.

3.5. Liveness Properties

Safety properties state what the program may or may not do, not what it must do. The **allowed changes** statement specifies how state functions are allowed to change; it does not say that they must change. Liveness properties assert that the program must do something. Temporal logic was applied to programs especially for reasoning about liveness properties, so it is not surprising that it excels at specifying these properties.

The liveness property we require of the GET subroutine is that if the queue is nonempty, then GET will eventually exit. Letting *after*(GET) be the state function that is true when the GET subroutine is at its exit point, this requirement is stated simply as

$$in(\text{GET}) \land (queue \text{ not empty}) \supset \Diamond\, after\ \text{GET}\ .$$

(Remember that $\Diamond A$ means that A eventually becomes true.) The obvious liveness property for the PUT subroutine is that it must terminate, stated simply as

$$in(\text{PUT}) \supset \Diamond\, after\ \text{PUT}\ .$$

However, this specifies that there is always room in the queue for another element, so the queue must have an unbounded capacity. A more realistic specification would state that the queue must have room for at least 17 elements, so PUT will exit if it finds fewer than 17 elements on the queue. This is stated formally as

$$in(\text{PUT}) \land (\text{length of } queue < 17) \supset \Diamond\, after\ \text{PUT}\ .$$

How one proves that an implementation satisfies these liveness properties is described in [18] and will not be discussed here. The same techniques allow one to use the liveness properties of the PUT and GET subroutines to deduce liveness properties of programs that call them.

3.6. Other Temporal Logic Specification Methods

There has been considerable interest in using temporal logic to specify concurrent programs, and several methods have been proposed. Since these methods are all based upon temporal logic, they are in some sense formally equivalent. However, they are quite different in practice.

One method, described in [6] and [7], involves making temporal logic assertions about histories. Instead of using a state function *queue*, it uses as state functions the sequence of values with which PUT has been called and the sequence of values returned by GET. This approach is appealing for two reasons. First, it produces some very simple specifications. For example, when this method is used to specify a queue, the property that corresponds to our **allowed changes** statement is that the sequence of values returned by GET is a subsequence of the sequence of values with which PUT has been called. Second, talking about the sequence of inputs and outputs seems like a natural way to describe a subroutine.

However, these sequences of values are really behaviors disguised as state functions. I have already indicated that it is easy to make errors when reasoning informally in terms of behaviors, so I am suspicious of formal methods based upon them. Moreover, the simplicity of the specifications is deceptive. While specifying a FIFO queue is simple, a specification of a LIFO queue in terms of history sequences is no simpler than the one described above. Methods based upon history sequences will work well for specifying FIFO queueing disciplines, but will not work so well with more complicated interactions. These methods may be appropriate in certain areas, such as communication protocols.

Another approach to temporal logic specification has been advocated by Schwartz and Melliar-Smith [21]. Their approach, which might be called a "more temporal" one, differs from the method I have advocated by replacing many of the state functions with temporal assertions. A more temporal specification of a queue might use the state function *queue*, but not *putarg* and *getval*. Instead of asserting that *queue* can change by adding to it the value of *putarg*, such a specification asserts that the value with which PUT was most recently called can be added to *queue*. The state function *putarg* is replaced by the temporal construction "with which PUT was most recently called".

While the state function *queue* seems natural, the state functions *putarg* and *getval* seem rather contrived. Replacing them with temporal assertions is quite appealing. Unfortunately, the resulting temporal logic specification seems difficult to understand. I introduced the state functions *putarg* and *getval* to eliminate the complex temporal logic formulas that arise when they are not used. More recently, Schwartz, Melliar-Smith and Vogt have developed a new way of writing temporal logic specifications, called interval logic, that seems to yield more understandable specifications [22]. I have also been working on a method of eliminating explicit mention of state functions like *putarg* and *getval* by the use of temporal expressions, but, in my approach, these temporal expressions turn out to be state functions in disguise.

4. HIERARCHY

4.1. The Hierarchy of Specifications

People have finite capacities; they can keep only so much in their minds at once. For a system to be understandable, it should have a high level specification simple enough to understand in its entirety. However, the thousands of lines of code in its implementation cannot be comprehended all at once. Therefore, the system's description must be structured so it can be read in small pieces. This is done with a hierarchy of specifications. At each level of the hierarchy, the system is described as a number of interacting modules, where a module is a collection of related procedures.

As an example, consider a system for sending messages from computer S to computer R. A message is sent by calling a PUT subroutine in computer S, and it is received by calling a GET subroutine in computer R. At the highest level, the system is described as a queue—sending is viewed as putting a message on the queue, and receiving is viewed as taking one off the queue. The entire system is regarded as a single module at this level.

At the next lower level of the hierarchy, which I will call the *protocol level*, the message-passing system is described in terms of three modules:

- A *transmission* module that performs the actual transfer of messages from computer S to computer R. Its implementation consists of code in both computers plus the actual transmission line.
- A *sender* module in computer S that accepts calls to PUT and calls the transmission module to send the messages to computer R.
- A *receiver* module in computer R that accepts calls to GET and in turn calls the transmission module to receive the messages coming from S.

At the level below the protocol level, each of these three modules is described as a set of interacting modules. For example, the sender module might consist of two concurrently active modules: one that handles calls to GET and puts messages on an internal queue, and another that takes messages from this queue and passes them to the transmission module. This process of hierarchical refinement continues down to a level whose modules are implemented directly in Pascal.

The high-level specification describes the message-passing system in terms of state functions like *queue* and *putarg*. The three modules of the protocol level are specified in terms of other state functions, perhaps including the following:

squeue: a queue of messages that have been given to the sender module by calls to PUT, but have not yet been sent to the receiver.

rqueue: a queue of messages received by the receiver module, but not yet fetched by calls to GET.

msg: the message currently in transit, or *NULL* if there is none.

What does it mean for the protocol-level specification to be a refinement of the high-level specification? Recall that the high-level specification has the form

> there exist state functions *queue*, *putarg*, ... such that $A_1, \ldots, A_m$,

where the A_i are temporal logic formulas. The protocol-level specification likewise has the form

> there exist state functions *squeue*, *rqueue*, *msg*, ... such that $B_1, \ldots, B_n$,

where the B_j are also temporal logic formulas. For the protocol-level specification to be a correct refinement of the high-level specification, the existence of state functions *squeue*, *rqueue*, *msg*, ... satisfying properties $B_1, \ldots, B_n$ must imply the existence of state functions *queue*, *putarg*, ... satisfying properties $A_1, \ldots, A_m$. How do we prove that this is the case?

To prove the existence of *queue*, ... , we must define them in terms of *squeue*, *rqueue*, *msg*, ... For example, we might define *queue* to be the concatenation of

- *rqueue*
- *msg* if it is not *NULL*
- *squeue*.

When we substitute for *queue*, *putarg*, ... their definitions in terms of *squeue*, *rqueue*, *msg*, ... , the assertions A_k of the high-level specification become assertions A'_k about the state functions of the protocol-level specification. The protocol-level specification is a correct refinement of the high-level specification if these assertions A'_k are true—more precisely, if they follow from the truth of properties $B_1, \ldots, B_n$. An example of how this is done can be found in [11], where the protocol-level specification describes the alternating-bit protocol.

In summary, to prove that a lower level specification

> B: there exist state functions $g_1, \ldots, g_s$ such that $B_1, \ldots, B_n$

is a correct refinement of a higher level specification

> A: there exist state functions $f_1, \ldots, f_r$ such that $A_1, \ldots, A_m$,

we must find expressions $F_i(g_1, \ldots, g_s)$ such that each formula A'_k obtained from A_k by performing the substitutions

$$f_i \leftarrow F_i(g_1, \ldots, g_s)$$

follows logically from the axioms $B_1, \ldots, B_n$.

Now consider a still lower-level specification

> C: there exist state functions $h_1, \ldots, h_t$ such that $C_1, \ldots, C_p$

We prove it to be a correct refinement of specification B by finding substitutions

$$g_j \leftarrow G_j(h_1, \ldots, h_t)$$

which yield formulas B'_l that can be proved from $C_1, \ldots, C_p$. It follows that the formulas A''_k obtained from the A_k by the substitutions

$$f_i \leftarrow F_i(G_1(h_1, \ldots, h_t), \ldots, G_s(h_1, \ldots, h_t))$$

can be proved from $B'_1, \ldots, B'_n$, which in turn can be proved from $C_1, \ldots, C_p$. Hence, if specification B is a correct refinement of specification A and specification C is a correct refinement of specification B, then specification C is, as expected, also a correct refinement of specification A. This means that if each refinement is correct, then proving that the program correctly implements the lowest level specification shows that it correctly implements every specification in the hierarchy.

The definition of what it means for a hierarchical refinement to be correct makes sense because both the higher- and lower-level specifications are assertions about the same Pascal model. The mapping between levels of the hierarchy is a mapping between state functions; higher-level state functions are defined in terms of lower-level ones. Refining a specification means giving a more detailed description of the program state—a simple, well-understood concept. This is in contrast to the behavioral approach, in which refinement requires that actions be decomposed into sequences of simpler actions—a much more complicated concept.

4.2. The Hierarchy of Programming Languages

I have interpreted the temporal logic formulas in a specification as assertions about a Pascal implementation, and have indicated how one can prove that a Pascal program correctly implements the specification. However, computers don't execute Pascal programs; they execute machine language programs. It would be nice to prove not only that the Pascal program correctly implements the specification, but that a machine language program correctly implements the Pascal program.

The Pascal program was described by a model consisting of sequences of atomic actions—*i.e.*, a behavioral model. In it, the execution of an atomic statement such as

$$\langle x := x + 1 \rangle$$

is a single atomic action. However, in a machine language version of the program, executing this statement could take 42 program steps (most of them performing the synchronization needed to guarantee atomicity). What does it mean for these 42 machine language steps to implement a single Pascal action? Do we need behavioral methods to couple the Pascal view to the machine language view?

Of course, the answer is no. Just as putting an element on a queue can be an atomic action in the high-level specification and be implemented by 42 Pascal operations, so can the execution of an assignment statement be atomic at the

Pascal level and be implemented by 42 machine language instructions. Although we have been thinking of Pascal executions as our underlying model, we can just as well let the underlying model consist of execution sequences of a machine language program. In this model, a state consists of an assignment of values to all machine registers and memory locations, while an action is the execution of a single machine language instruction.

With this underlying model, the Pascal program can be viewed as a specification of a machine language program. Just like *queue* in our high-level specification, the variable x is a state function in the Pascal program/specification—a function of the values of machine registers and memory locations. Even though incrementing x may require several machine language instructions (for example, x might be a multiple precision integer that occupies more than one memory location), the state function x can be defined in such a way that its value changes atomically.

This sounds good in principle, but how does it really work? When we write a temporal logic specification, we are writing temporal logic formulas, or constructs like the **allowed changes** statement that can be translated directly into temporal logic formulas. It is not clear how one translates a Pascal program into a temporal logic specification of a machine language program.

A semantics for a programming language is a way of assigning a meaning to every legal program in the language. In a temporal logic semantics, the meaning of the program is a temporal logic specification, which is a collection of temporal logic axioms. To give a temporal logic semantics for Pascal, one must define a method for translating from Pascal statements to temporal logic axioms. For example, an atomic assignment statement

$$l: \langle\, x := x + 1 \,\rangle$$

might yield the safety axiom

allowed changes to x **while** $in(l)$
$in(l)_{old} \;\rightarrow\; (x_{new} = x_{old} + 1) \;\wedge\; after(l)_{new}$

and the liveness axiom

$in(l) \;\supset\; \diamond\, after(l)\,.$

Defining how one translates from any arbitrary statement in a complex programming language into temporal logic formulas is a difficult task that has not been attempted. It has been done for some simple constructs in an idealized language, but this work has not yet appeared.

Machine language versions of individual Pascal programs are usually produced by a compiler. Instead of verifying each machine language program individually, one wants to show that the compiler is correct. A temporal logic semantics for Pascal defines what it means for the compiler to be correct—namely, that the machine language program it produces is always a correct implementation of the Pascal program it is given. However, there is still a big step from this to proving the correctness of a real compiler.

Of course, the hierarchy need not stop with the machine language program. We can just as well choose a *register transfer model* in which the atomic actions are the setting of individual registers inside the computer. The machine language program can then be viewed as a higher-level specification of a register transfer "program". Proving the correctness of the register transfer implementation of a machine language program involves proving that the computer correctly implements its machine-language description. This is the first step in hardware verification: proving the correctness of the register-transfer-level design.

One can proceed further into the hierarchy of descriptions of the computer. Since our model of temporal logic involves discrete actions, it would appear that we must stop at the level of physical electronics, where a flip-flop assumes a continuous range of voltage levels instead of a binary value. However, the temporal logic operator $\trianglelefteq$ is meaningful even when time is continuous; our temporal logic does not rule out such an underlying model. (This is a further advantage of eschewing the *next* operator.) The refinement process can continue until we reach a level where quantum effects appear. Because a quantum-mechanical state is a superposition of the states that temporal logic formulas talk about [4], I have no idea how one could use temporal logic at the quantum level.

Several researchers have investigated the use of temporal logic for specifying and verifying hardware [3],[13],[16]. An assertional method is proposed in [23] that, while not expressed in temporal logic, is compatible with our method. It seems clear that the design of VLSI circuits could benefit greatly from the hierarchical approach made possible by temporal logic.

5. SOME CONCLUDING REMARKS

I have not yet used the word "distributed". How one specifies and reasons about a system is independent of whether or not it is distributed. Although they may require different implementation techniques and different programming languages, distributed and nondistributed systems can be described in the same logical framework. It is shown in [12] that no special methods are needed to reason about distributed systems. You have just seen that one high-level specification describes both a distributed message-passing system and a shared memory queue; only their implementations differ.

One important aspect of specification was ignored in the queue example. The actions of PUT and GET were specified in terms of certain operations on the data structures—for example, the concatenation operation $*$. The question of how these operations are specified was not considered. A complete specification method must permit the precise definition of arbitrary data structures and the operations on them. Fortunately, a large body of research has been devoted to this topic—*i.e.*, research on "abstract data types" and "algebraic specification" [5]. The goal of this work has been to specify programs, which is possible in the sequential case where programs can be described by the relation between their input and output, so they can be specified purely in terms of operations on data structures. Concurrent programs cannot be specified in this way, which is why temporal logic was introduced. However, these algebraic methods can specify operations on the values assumed by state functions, while temporal logic specifies how the state functions change.

Concurrent systems are inherently complex and are often plagued by timing-dependent synchronization errors that are impossible to find by conventional testing methods. There is reason to suspect that over 90 percent of the "crashes" in some systems are due to such synchronization errors. Rigorous, formal reasoning is the best way to eliminate them. Computers are much better than people at formal reasoning, so a formal correctness proof should be checked by a computer. The temporal logic reasoning I have described can be formalized and used as the basis for a mechanical verification system.

Mechanical verification is now feasible for small systems [14], but it is expensive. Further development of verification systems is needed before they can be applied to large concurrent programs. Fortunately, temporal logic is useful without mechanical verification. Writing a formal specification can be helpful even with no attempt at verification because it forces one to understand precisely what the system is supposed to do. Informal reasoning—the kind that mathematicians have been using for centuries—can catch many errors.

ACKNOWLEDGEMENTS

I wish to thank the following people for their helpful comments on an earlier draft of this paper: Edsgar Dijkstra, Brent Hailpern, C.A.R. Hoare, Cees Jan Koomen, Gerard LeLann, J. Misra, Ben Moszkowski, Fred Schneider, Richard Schwartz, and Michel Sintzoff.

REFERENCES

[1] R.M. Burstall. Program Proving as Hand Simulation with a Little Induction, 308-312 in *Information Processing 74*, North-Holland Pub. Co., 1974.

[2] Arthur Bernstein and Paul K. Harter. Proving Real-Time Properties of Programs with Temporal Logic, 1-11 in *Proceedings of the Eighth Symposium on Operating System Principles*, ACM SIGOPS, Pacific Grove, California, 1981.

[3] G. V. Bochmann. Hardware Specification with Temporal Logic. *IEEE Trans. on Computers C-31*, 3 (Mar. 1982), 223-231.

[4] B. Dewitt and N. Graham, Eds. The Many-Worlds Interpretation of Quantum Mechanics, Princeton University Press, Princeton, New Jersey, 1973.

[5] J. A. Goguen, J. W. Thatcher and E. G. Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, 4, 80-149 in *Current Trends in Programming Methodology*, Prentice-Hall, Inc., 1978.

[6] Brent T. Hailpern and Susan S. Owicki. Verifying Network Protocols Using Temporal Logic, 18-28 in *Proceedings Trends and Applications 1980: Computer Network Protocols*, IEEE Computer Society, 1980.

[7] Brent T. Hailpern and Susan S. Owicki. Modular Verification of Computer Communication Protocols. *IEEE Trans. on Commun. COM-31*, 1 (Jan. 1983), 56-68.

[8] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. on Soft. Eng. SE-3*, 2 (Mar. 1977), 21-37.

[9] L. Lamport. The 'Hoare Logic' of Concurrent Programs. *Acta Informatica 14*, 1 (1980), 21-37.

[10] L. Lamport. 'Sometime' is Sometimes 'Not Never': A Tutorial on the Temporal Logic of Programs, in *Proceedings of the Seventh Annual Symposium on Principles of Programming Languages*, ACM SIGACT-SIGPLAN, Jan. 1980.

[11] L. Lamport. Specifying Concurrent Program Modules. *ACM Trans. on Prog. Lang. and Systems 5*, 2 (April 1983).

[12] L. Lamport. An Assertional Correctness Proof of a Distributed Algorithm. To appear in *Science of Computer Programming.*

[13] Y. Malachi and S.S. Owicki. Temporal Specifications of Self-Timed Systems, 203-212 in *VLSI Systems and Computations*, Rockville, Maryland, 1981.

[14] P. M. Melliar-Smith and R. Schwartz. Specification and Mechanical Verification of SIFT: A Fault Tolerant Flight Control System. *IEEE Transactions on Computers C-31*,7 (Jul. 1982).

[15] R. Milner. A Calculus of Communicating Systems, Springer-Verlag, Berlin, 1980.

[16] B. Moszkowski. A Temporal Logic for Multi-level Reasoning About Hardware, in *Proceedings of the IFIP Sixth International Conference on Computer Hardware Description Languages and Their Applications*, Carnegie-Mellon University, Pittsburgh, 1983.

[17] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica 6*, 4 (1976).

[18] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Trans. on Prog. Lang. and Sys. 4*, 3 (July 1982), 455-495.

[19] A. Pnueli. The Temporal Logic of Programs, in *Proc. of the 18th Symposium on the Foundations of Computer Science*, ACM, Nov. 1977.

[20] N. Rescher and A. Urquhart. Temporal Logic, Springer-Verlag, New York, 1971.

[21] Richard L. Schwartz and P. M. Melliar-Smith. Temporal Logic Specification of Distributed Systems, 446-454 in *Proceedings of the 2nd International Conference on Distributed Computing Systems*, IEEE Computer Society Press, 1981.

[22] Richard L. Schwartz, P.M Melliar-Smith and Friedrich H. Vogt. An Interval Logic For Higher-Level Temporal Reasoning, SRI International Tech. Report CSL-138 (Feb. 1983).

[23] R.E. Shostak. Formal Verification of Circuit Designs, in *Proceedings of the Third Cal Tech Conference on VLSI*, California Institute of Technology, Pasadena, California, 1983.