Specifying Concurrent Program Modules

LESLIE LAMPORT
SRI International

A method for specifying program modules in a concurrent program is described. It is based upon temporal logic, but uses new kinds of temporal assertions to make the specifications simpler and easier to understand. The semantics of the specifications is described informally, and a sequence of examples are given culminating in a specification of three modules comprising the alternating-bit communication protocol. A formal semantics is given in the appendix.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies*; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*

General Terms: Verification

Additional Key Words and Phrases: Multiprocessing, temporal logic, communication protocols

1. INTRODUCTION

In this paper, we describe a method for specifying a module in a concurrent program, where a module is a collection of related subroutines. It involves specifying two kinds of properties:

—*Safety properties*, describing what the program is allowed to do—or, dually, what it may not do.
—*Liveness properties*, describing what the program must do.

We do not specify other kinds of properties, such as performance characteristics. We seek specifications that are as weak as possible, stating only those properties of the program necessary to meet the needs of the user. Such specifications leave the implementer free to choose the best possible implementation.

Many specification methods have been proposed for sequential programs. They generally specify a module in terms of the values returned by its subroutines. However, this is inadequate for a concurrent program, where other parts of the program may be executed at the same time as the module. For example, consider a module to implement a FIFO queue, with PUT and GET subroutines for inserting and removing elements. In a sequential program, this module can be

This work was supported in part by the National Science Foundation under grants MCS-78-16783 and MCS-81-04459.
Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

specified in terms of the values returned by GET as a function of previous calls to PUT and GET, where GET returns an error if the queue is empty. However, for a concurrent program in which PUT and GET are called by different processes, one might want to specify that the GET operation waits if the queue is empty. The concept of waiting cannot be expressed in terms of the values returned.

As this example indicates, a module in a concurrent program must be specified in terms of its behavior, rather than the values it returns. Temporal logic has proved to be a successful tool in reasoning about the behavior of concurrent programs [8, 9], so it is a natural choice as a formalism for specifying concurrent program modules. However, the temporal logic that has been used thus far is not convenient for expressing many properties of concurrent programs. We have therefore introduced new kinds of temporal assertions to make the specifications simpler and easier to understand.

In this paper, we ignore issues that have already been studied for sequential programs. For example, we assume that the data structures we need, such as sequences, have already been defined. A complete specification system would include some method for defining data types, probably using an axiomatic approach.

The standard temporal logics that have been employed can only be used to specify an entire concurrent program, not part of one, because they consider only the changing program states and not the actions that cause the change. Formalizing our specification method requires a generalization of temporal logic to include predicates for describing the actions that are executed. However, the specifications can be understood with no knowledge of the formal temporal logic upon which they are based. We therefore give only a brief, informal semantics for our specifications in the main body of the paper, leaving a precise formalization to the appendix. The rest of the paper is devoted to a sequence of examples, including a FIFO queue and culminating in a specification of three modules comprising a communication protocol—the familiar alternating-bit protocol [1, 2, 11].

The specification of a program module may be viewed as a contract between the user of the module and its implementer. It must contain all the information needed to

(1) Enable the user to design a program that uses the module, and verify its correctness, without knowing anything about how the module is implemented.
(2) Enable the implementer to design the module, and verify its correctness, without knowing anything about the program that uses the module.

It is therefore important that the specifications not only be easy to write and understand, but that they be easy to use for these two purposes. We show by examples that this is true of the specifications produced with our method. In our final example, using the specifications of the three modules comprising the alternating-bit protocol, we show that they satisfy the specification of a FIFO queue, and thus correctly implement a lossless transmission line. All of our proofs are informal. Since the specifications can be translated into temporal logic formulas, formal proof methods are possible. However, they are beyond the scope of this paper.

Unlike many other protocol specification methods, our method specifies not an abstract protocol but an actual program module containing subroutine calls for sending and receiving messages. In practice, one is concerned not with abstract protocols but with the program modules that implement them. An extra layer of formalism relating programs and abstract protocols is needed to verify that a program module correctly implements such an abstract protocol specification. We have avoided this extra layer by specifying the program module itself.

The word "specification" is often used to denote what we would call a high-level design. A specification of a program describes what the program should do; a high-level design describes how it should do it. This distinction is not a formal one, since even a machine language program can be made into a specification by prefacing it with "Any program that computes the same result as ...." This "specification" does not say how the results are to be computed, but it would certainly bias an implementer toward a particular method. Any specification will probably have some bias toward a certain type of implementation, so there is no sharp dividing line between specifications and high-level designs. However, we propose that with any true specification method, it should be easy to specify programs that cannot be implemented, since describing what should be done need not imply that it can be done. For example, specifying a program to decide if a finite-state machine will halt and specifying a program to decide if a Turing machine will halt should be equally easy—even though one of the programs can be written and the other cannot.

Our specifications do not distinguish between distributed and nondistributed programs. The specification of a FIFO queue is the same whether the PUT and GET operations are issued in the same computer, making the queue a data structure in a nondistributed program, or are issued on separate computers, making the queue a transmission medium in a distributed program. We feel that the essential difference between a distributed and a nondistributed program is in their performance, which is outside the scope of our specifications.

2. THE METHOD

2.1 The Underlying Model

We assume that the execution of a concurrent program can be represented as a sequence of state transitions of the form

$$s \xrightarrow{\alpha} s',$$

which denotes that the action α took the program from state s to state s'. Typically, this transition would represent the execution of a single atomic statement in some process, in which case s is the state before the execution, s' is the state immediately after the execution, and α denotes the program statement being executed. However, the exact nature of the states and actions does not concern us. Concurrency is represented by the interleaving of concurrent atomic operations.

We therefore assume a set S of states, a set A of actions, and a set Σ of program execution sequences of the form

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

where the s_i are in S and the α_i are in A. All execution sequences are infinite; terminating programs are represented by having a null action that takes a halted state into itself.

We consider the meaning of the program to be the set Σ of all possible execution sequences. This set contains executions starting from every state in S. For a nondeterministic program, there can be many sequences in Σ with the same starting state s_0.

A state represents a complete "snapshot" of the program at some instant of time. At any point during the execution, the possible future behavior of the program must depend only upon its current state, and not upon how it reached that state. Thus, the state must include not only the value of program variables, but also the values of processes' "program counters", the values of parameter-passing stacks, the contents of message queues, the states of transmission lines, etc.

A *state function* is a mapping from the set S of states into some set of values. A *predicate* is a boolean-valued state function. There are two kinds of primitive state functions that we will use:

Program variables: A program variable x is a state function which assigns to any state s the value of x in that state.

Control predicates: If π is some control point in the program—that is, some possible "program counter" value—then we let $at(\pi)$ denote the predicate which is true for a state if and only if control is at the point π in that state.

We can construct more complex state functions from these primitive ones. For example, $at(\pi) \wedge x > 0$ is a predicate that is true for a state if and only if control is at the point π and the variable x has a positive value in that state.

2.2 Specifications

To specify a program, we must specify the set Σ of all its possible execution sequences. A specification consists of a collection of conditions on execution sequences. A program satisfies the specification if all of its possible execution sequences satisfy each of these conditions. We describe the semantics of these specifications informally; a formal definition in terms of temporal logic is given in the appendix. We advise the reader to skim quickly through the rest of this section on first reading, and to return to it when studying the examples in the following section.

A specification has the following form:

state functions: $f_1 : R_1, \ldots, f_n : R_n$
initial conditions: $I_1, \ldots, I_m$ (*)
properties: $P_1, \ldots, P_q$

This has the following interpretation:

THERE EXIST state functions $f_1, \ldots, f_n$ SUCH THAT:
The range of f_i is R_i, for each i, AND
IF the initial state satisfies $I_1, \ldots, I_m$
THEN properties $P_1, \ldots, P_q$ are satisfied throughout the execution.

In order to verify that a program satisfies such a specification, one must demonstrate the existence of the state functions f_i by defining them as functions of the program state. The values assumed by each f_i must lie in the set R_i.

The initial conditions I_j are predicates. The specification places constraints only on those executions for which each I_j is true for the initial state s_0.

Each property P_j expresses a constraint on the entire execution sequence

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots.$$

There are two basic types of properties: safety properties and liveness properties.

2.2.1 *Safety Properties.* A safety property asserts that something must never happen. The simplest form of safety property is a predicate. If P_j of (*) is a predicate, then it asserts the property that this predicate is true throughout the execution—that is, that P_j is true for every state s_i in the execution sequence. Such a property will be called an *invariance* property.

Another form of safety property asserts that a state function cannot change when it is not supposed to. It is expressed in the form

a leaves unchanged f when Q,

where a is a set of actions, f is a state function, and Q is a predicate. This has the following meaning.

For every transition $s \xrightarrow{\alpha} s'$ in the execution, *if*

(i) α is in a and
(ii) Q is true on state s,

then the value of f on state s' equals its value on state s.

In other words, this property asserts that any action in a leaves f unchanged when it acts in a starting state satisfying Q. If a is the set of all actions, then the "**a leaves**" is omitted. If Q is the trivial predicate *true*, then the "**when Q**" is omitted.

Rather than asserting when state functions may not change, we can assert when they may change by using the following construction.

allowed changes to g_1 when Q_1,
$$\vdots$$
g_p when Q_p:
$a_1 : R_1 \rightarrow S_1$,
$$\vdots$$
$a_u : R_u \rightarrow S_u$

where the g_i are state functions, the Q_i and R_j are predicates, the a_j are sets of actions, and the S_j are boolean functions on ordered pairs of states. This asserts the following for each i:

For every transition $s \xrightarrow{\alpha} s'$ in the execution, *if*

(i) Q_i is true for state s and
(ii) the value of g_i on state s' is different from its value on s,

then there is some j such that

(i) α is in a_j
(ii) R_j is true for state s
(iii) S_j is true for the pair of states (s, s').

This property constrains the changes to the state functions g_i by describing exactly what state transitions may change them. Any transition that changes g_i starting in a state with Q_i true must satisfy one of the transition specifications

$$a_j: R_j \rightarrow S_j.$$

In this transition specification, R_j is an *enabling predicate*, stating what must be true of the starting state, and S_j specifies a relation that must hold between the starting state s and the final state s'. We require that if S_j is true for the pair of states (s, s'), then R_j must be false for s'. In other words, the enabling condition must become false after the transition. This requirement is needed for the formal statement of the property in terms of temporal logic that is given in the appendix.

The boolean functions S_j are described by expressions involving primed and unprimed state functions. When evaluating such an expression on the pair of states (s, s'), the unprimed state functions are evaluated on the starting state s, and the primed ones on the final state s'. Thus, $g_1 > f'$ is true for the pair of states (s, s') if and only if the value of the state function g_1 evaluated on state s is greater than the value of the state function f evaluated on state s'.

The expression S_j is often of the form

$$g_i' = g_i \wedge \cdots,$$

indicating that the transition is not allowed to change the value of g_i. We make the convention that if any of the state functions g_i does not appear primed in the expression S_j, then S_j is assumed to contain an unwritten conjunctive clause of the form $g_i' = g_i$. Thus, any g_i that does not appear in the transition specification cannot be changed by the transition.

As another notational convenience, the "**when** Q_i" is omitted if Q_i is the trivial predicate *true*.

We sometimes write an "**allowed changes**" property in which a transition specification has the form

for all v: $a: R \rightarrow S$,

where v may appear as a free variable in the expressions R and S. This means that for any value of v, a transition satisfying the transition specification $a: R \rightarrow S$ for that value of v may change the values of the g_i.

2.2.2 *Liveness Properties.* Safety properties state what may or may not occur, but do not require that anything ever does happen. For example, an "**allowed changes**" property specifies state transitions that *may* occur in the execution sequence; it does not specify that any such transitions actually do occur. Liveness properties state what *must* occur. They are specified using temporal logic assertions. We shall give here only a brief, informal description of the temporal logic assertions we shall need. A more complete discussion of temporal logic is given in [6] and [8], and a formal semantics is given in the appendix. (We are using the "linear time" logic of [6].)

The properties P_j in the specification (*) are to hold throughout the program execution. We describe what it means for a temporal logic assertion to hold at "time i" in the execution sequence

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

—the time right before the $(i + 1)$st transition, when the program state is s_i. This is done below for four types of assertions, where P denotes any predicate.

P: true at time i if and only if it is true on state s_i.
$\Box P$: true at time i if and only if P is true at all times $j \geq i$.
$\Diamond P$: true at time i if and only if P is true at some time $j \geq i$.
$\Box\Diamond P$: true at time i if and only if P is true at infinitely many times $j \geq i$.

We pronounce $\Box$ as "henceforth", $\Diamond$ as "eventually", and $\Box\Diamond$ as "infinitely often".

All the liveness properties we use will be logical combinations of these four kinds of assertions. For example, the property $\Box P \supset \Diamond Q$ asserts that at any time i, if P is true for all states s_j with $j \geq i$, then Q is true for some state s_j with $j \geq i$. (Remember that the properties in specification (*) are required to hold at all times.) The tautology

$$\sim\Box P \equiv \Diamond\sim P$$

is very important. It states formally the simple observation that P is not always true if and only if it is eventually false.

We write $P \rightsquigarrow Q$ (read "P leads to Q") as an abbreviation for $\Box(P \supset \Diamond Q)$. Thus, the property $P \rightsquigarrow Q$ means that for any time at which P is true, Q must be true then or at some later time.

3. EXAMPLES

3.1 Subroutines

In our examples, we specify various program modules, which requires specifying how the rest of the program communicates with these modules. For convenience, we use a very simple subroutine-calling mechanism. Each module has one or more subroutines that can be called by the rest of the program. There can be only one invocation of a given subroutine active at any instant, although different subroutines may be called concurrently. Thus, a subroutine may be thought of as being "in-line", appearing only once in the program that calls it. (To allow concurrent invocations of the same subroutine, we would have to add some method for naming the different invocations.)

Arguments and results are passed using global variables. A subroutine SUB is called with its arguments placed in the variable SUB.PAR, and SUB leaves its results in that same variable before exiting. Every subroutine SUB therefore includes in its specification the state function SUB.PAR, whose range depends upon the types of arguments and results. For example, if SUB takes an integer as an argument and returns a boolean as a result, then the range of SUB.PAR is **integer or boolean**, denoting the union of the set of integers and the set {*true*, *false*}.

state functions:
SUB.PAR :...
at(SUB) :**boolean**
in(SUB) :**boolean**
after(SUB):**boolean**

initial conditions: ~[*at*(SUB) ∨ *in*(SUB) ∨ *after*(SUB)]

properties:
1. (*a*) α[MOD] **leaves unchanged** SUB.PAR **when** ~ *in*(SUB)
   (*b*) ~α[MOD] **leaves unchanged** SUB.PAR **when** *in*(SUB)
2. **allowed changes to** *at*(SUB)
   in(SUB)
   after(SUB)
   (a) α[*MOD*]: *in*(SUB)→ ~*in*(SUB)′ ∧ *after*(SUB)′ ∧ ~*at*(SUB)′
   (b) α[*MOD*]: *at*(SUB)→ ~*at*(SUB)′ ∧ *in*(SUB)′
   (c) ~α[*MOD*]: ~*in*(SUB)→ *in*(SUB)′ ∧ *at*(SUB)′ ∧ ~*after*(SUB)′
   (d) ~α[*MOD*]:*after*(SUB)→ ~*after*(SUB)′

Fig. 1. Specification of subroutine SUB of module MOD.

There are three other state functions that will be needed for the subroutine SUB: the predicates *at*(SUB), *in*(SUB), and *after*(SUB). These predicates are functions of the control state of SUB—that is, functions of SUB's "program counter". The predicate *at*(SUB) is true if and only if control is at the entry point to SUB, and *after*(SUB) is true if and only if control is at the exit point of SUB—the point reached upon completion of SUB. The predicate *in*(SUB) is true if control is anywhere inside subroutine SUB—including its entry point, but excluding the exit point. (These predicates are the same ones used in [5] and [8].)

We now use our method to specify the relevant properties of the argument/result-passing variable SUB.PAR and the three predicates *at*(SUB), *in*(SUB), and *after*(SUB). We assume that the subroutine SUB is part of a module MOD. The complete specification is given in Figure 1 and is explained below.

The **state functions** part of the specification is self-evident. The range of SUB.PAR will depend upon the particular subroutine SUB. (To improve readability, we have eliminated the commas of our general form (*).) The **initial conditions** specify that in the initial state, control is not in SUB or at its exit.

In the properties, we have introduced the notation that α[MOD] denotes the set of actions in the module MOD, and ~α[MOD] denotes the complement of this set—the set of all program actions not in MOD. Recall that we can think of the actions of a program as the atomic statements of that program, so α[MOD] represents the set of atomic statements in the module MOD.

Property 1(a) states that no action in MOD can change the value of SUB.PAR when control is not in SUB. (MOD can contain other actions besides those of SUB.) Property 1(b) states that no action outside of MOD can change SUB.PAR when control is in SUB. Note that although it is written as part of the specification of module MOD, property 1(b) actually constrains the actions of the environment containing MOD. Such a constraint is necessary, since no argument or value passing could take place if SUB.PAR could be changed by outside activity during the execution of SUB. Property 1(b) really represents a constraint on the parameter passing mechanism.

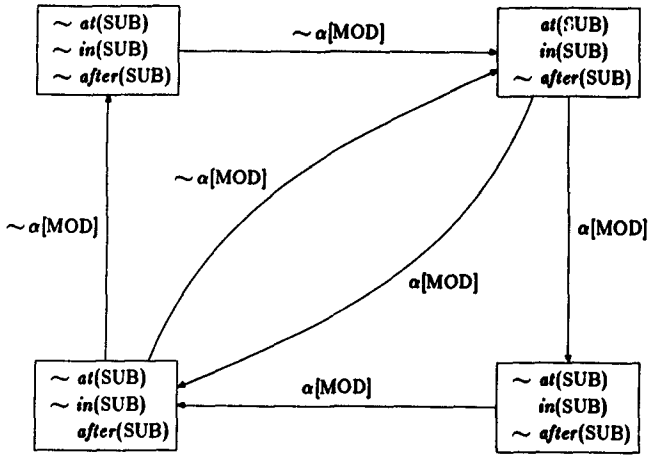

Fig. 2. Changes to *at*(SUB), *in*(SUB), and *after*(SUB).

Property 2 describes how the values of the *at*, *in*, and *after* predicates can change. To understand it, let us consider how they should change. The behavior we want is indicated by the state-transition diagram of Figure 2. The upper left-hand state is the initial one. The only way to leave that state is to call SUB, which causes control to reach the entry point of SUB—indicated by the upper right state. This change of state is caused by an action that is not in MOD. From the entry point of SUB, there are two possible places to go: "further inside" SUB, or directly to the exit. The lower right state is "further inside" SUB, and the lower left one is when control is at the exit point of SUB. The rest of the diagram should now be clear.

There are six transitions in Figure 2, representing all the allowed changes to the predicates *at*(SUB), *in*(SUB), and *after*(SUB). We could describe this with an **allowed changes** property containing six transition specifications. However, we can simplify this by observing that the two transitions to the lower left state can be combined into a single transition specification, stating that starting in a state with control in SUB—whether or not control is at the entry point—an action of MOD can make control reach the exit point. This is specified in the transition specification (a) of Property 2 in Figure 1. Similarly, the two transitions leading to the upper right state have been combined in the transition specification (c). The two remaining transitions are described by (b) and (d). Note that in (b), the fact that *after*(SUB) does not appear means that the transition must leave it unchanged. Similarly, (d) specifies a transition that leaves *at*(SUB) and *in*(SUB) unchanged.

It is interesting to note that from this specification, we can derive the following two properties:

$$at(\text{SUB}) \supset in(\text{SUB}).$$

$$after(\text{SUB}) \supset \sim in(\text{SUB}).$$

In other words, these two predicates will be true for every state reached during

the execution. To prove this, we first observe that their conjunction is implied by the initial conditions, so it is true for the initial state. Next, we note that this conjunction is not changed by any transition that does not change *at*(SUB), *in*(SUB), or *after*(SUB). Hence, we need only show that the truth of the conjunction is preserved by each of the four kinds of transitions allowed by Property 2, which is easily done. This illustrates how safety properties are derived from a specification.

Whenever we specify a module with one or more subroutines, we shall assume that the state functions, initial conditions, and properties of Figure 1—with "MOD" and "SUB" replaced by the appropriate names—are an implicit part of the module's specification. Notice that the two properties of this specification are both safety properties.

The reader may find it strange that the specification of a subroutine specifies properties of actions outside that subroutine—namely, Properties 1(a), 2(c), and 2(d). This is not necessary for sequential programs because there are no other actions taking place while a subroutine is being executed. In sequential programs, the result of executing a subroutine depends only on the state in which it is begun. This is not true for concurrent programs, since the behavior of a subroutine can be affected by the actions of concurrently executing processes. The subroutine cannot be expected to behave properly under completely arbitrary actions of other processes—for example, if they change the values of its local variables. This type of interference is ruled out by most programming languages. The specification of Figure 1 describes the properties of a programming language's subroutine mechanism required for our examples.

3.2 A Simple Subroutine

As a very simple example, we consider a module containing only the single subroutine SQUARE, which takes an integer argument and returns its square. To specify this subroutine, we must first decide what state functions are needed (besides the functions SQUARE.PAR, *at*(SQUARE), *in*(SQUARE), and *after*(SQUARE), which we do not explicitly mention). We specify a single state function *val* which, at any point during the execution of SQUARE, equals the value that SQUARE will return. Since the state at any instant determines the future behavior of SQUARE, including the value that it will return, it must always be possible to define such a state function.

The complete specification is given in Figure 3. Since the implicit initial condition states that control is outside the SQUARE subroutine, we need make no initial hypothesis about the value of *val*. Hence, there are no explicit initial conditions. For convenience, we separate the safety and liveness properties as indicated. Properties 1 and 2 state the relations between the value of the argument/result-passing variable SQUARE.PAR and the final output value *val*. Property 3 states that the value of *val* does not change during the execution of SQUARE. Combining these three properties, we see that if control is at the entry point of SQUARE (*at*(SQUARE) true) with SQUARE.PAR = v, then if control reaches the exit point (*after*(SQUARE) true), at the instant it reaches the exit point SQUARE.PAR will equal v^2. Thus, Properties 1–3 express the usual partial correctness conditions for SQUARE.

module SQUARE **with subroutine** SQUARE:

state functions:

val: **integer**

initial conditions:

safety properties:

1. *at*(SQUARE) ⊃ *val* = SQUARE.PAR^2
2. *after*(SQUARE) ⊃ SQUARE.PAR = *val*
3. **unchanged** *val* **when** *in*(SQUARE)

liveness properties:

4. *in*(SQUARE) ⇝ *after*(SQUARE)

Fig. 3. Specification of the SQUARE subroutine.

```
subroutine SQUARE
  declare x, y integer;
  begin
    a: ⟨x := |SQUARE.PAR|⟩;
    b: ⟨y := x⟩;
    c: ⟨SQUARE.PAR := 0⟩;
       while d: ⟨y ≥ 1⟩
          do e: ⟨SQUARE.PAR := SQUARE.PAR + x⟩;
             f: ⟨y := y − 1⟩
          od
    g:
  end
```

Fig. 4. Implementation of the SQUARE subroutine.

These safety properties imply that SQUARE cannot reach its exit point without producing the correct result. However, they do not imply that SQUARE ever will reach its exit point. Liveness Property 4 expresses the requirement that SQUARE must always terminate. It states that if control is ever in SQUARE, then it must eventually reach the exit point.

We now indicate how one verifies that a particular program meets this specification. We consider the simple program of Figure 4. The angle brackets enclose the atomic program actions. It is assumed that SQUARE.PAR is declared externally to be an integer variable. There are six control points (possible program counter values) inside the subroutine—the points labeled *a*–*f*. The control point *g* is the exit point of the subroutine.

To prove that this program meets the specification, we must define the state functions of the specification in terms of the program state. The control predicates are defined in the obvious manner as follows:

$$at(\text{SQUARE}) \equiv at(a).$$

$$in(\text{SQUARE}) \equiv at(a) \vee at(b) \vee \cdots \vee at(f).$$

$$after(\text{SQUARE}) \equiv at(g).$$

(Recall that $at(a)$ is the predicate that is true if and only if program control is at control point a.)

The definition of the state function *val* is more difficult, and is given below:

$$
\begin{aligned}
val \equiv\ & \textbf{if } at(a) \vee at(b) \\
& \quad \textbf{then } \text{SQUARE.PAR}^2 \\
& \quad \textbf{else } \text{SQUARE.PAR} + (\bar{y} * x) \\
& \textbf{where } \bar{y} \equiv \textbf{if } at(f) \textbf{ then } y - 1 \\
& \qquad\qquad\ \ \textbf{else } y.
\end{aligned}
$$

We now have to show that the properties of the specification are satisfied with these definitions. We do not attempt to describe any formal method for doing this, but simply sketch an informal proof. We must first verify the implicit part of the specification, given by Figure 1.

For the initial conditions, which state that control is not initially in or after SQUARE, to be true, we need an assumption about the programming language stating that program control is not initially at any of the points *a*–*g*. Property 1(a) of Figure 1 for subroutine SQUARE is obvious, since the subroutine cannot take any action unless program control is inside it. Property 1(b) must be achieved by some kind of programming language rule or convention to prohibit any concurrently executed program from modifying SQUARE.PAR while control is in subroutine SQUARE. Property 2 follows from the ordinary rules for program control. The verification of the properties specified in Figure 1 is essentially the same for any subroutine, and will not be repeated for subsequent examples.

We must next verify the explicit part of the specification given in Figure 3. Property 1 follows immediately, since *val* is defined to equal SQUARE.PAR^2 when program control is at *a*. Property 2 follows from the easily verified fact that $\bar{y} = y = 0$ when program control is at *g*. Finally, to verify Property 3, one must show that the value of *val* cannot change when control is in the subroutine. This requires first of all that no action by any other subprogram can change *val* while control is in SQUARE, for which we must assume that the programming language prevents any other subprogram from modifying the local variables *x* and *y*. (We have already assumed that no other subprogram can change SQUARE.PAR while control is in SQUARE.) Next, we must show that no action in the subroutine SQUARE can change the value of *val*. This is done by examining the effect of executing each of the six atomic actions of the program (five statements plus the evaluation of the **while** condition), and showing that none of them changes the value of *val*. Verifying Liveness Property 4 requires an ordinary termination proof.

The reader will note that the proof of the safety properties resembles a standard assertional proof of partial correctness. Property 3, stating that *val* is unchanged, plays the part of the attached assertions in the Floyd method. It is characteristic of our specification method that proving safety properties of an implementation involves an assertional correctness proof.

3.3 An Unimplementable Specification

We can easily modify the specification of the SQUARE subroutine in Figure 3 to specify a square-root subroutine. In addition to the obvious name changes, we simply change Property 1 to

$$at(\text{SQUARE\_ROOT}) \supset val^2 = \text{SQUARE\_ROOT.PAR}.$$

This means that if SQUARE_ROOT is called with any integer argument v, then it must eventually terminate (by Property 4), producing an integer result equal to the square root of v.

Since not every integer has an integer square root, such a subroutine cannot be implemented. However, it is important to observe exactly why it cannot be implemented. At first glance, one might suppose that the specification is self-contradictory. One can easily write self-contradictory specifications—for example, by including contradictory properties such as $f > 0$ and $f < 0$. However, this is not the case with the SQUARE_ROOT specification. This specification is not contradictory, but rather implies the property

$$at(\text{SQUARE\_ROOT}) \supset \text{SQUARE\_ROOT.PAR is a perfect square,}$$

meaning that the SQUARE_ROOT subroutine is called only with an argument that is a perfect square.

When we say that the SQUARE_ROOT specification is not implementable, we mean that we cannot write a subroutine that meets the specification regardless of the environment in which it is placed. The specifications we write for a module do not specify the behavior of just that module; they specify the behavior of the entire program—for example, Property 3 of Figure 3 states that *val* is left unchanged not just by the SQUARE subroutine, but by the rest of the program as well. Any method for specifying a module in a concurrent program must permit the specification of what the rest of the program can do, since other subprograms can be executed concurrently with the module. (No module can behave correctly if concurrently executed subprograms can arbitrarily change the value of its variables.) We consider a module to meet a specification only if the specification is satisfied by any program that includes the module. This means that the programming language in which the module is implemented must permit the appropriate encapsulation of the module's variables.

3.4 A Queue

3.4.1 *The Specification.* We now specify a module that involves concurrent processing, having two subroutines that can be called concurrently. The QUEUE module provides a FIFO queue, having two subroutines: PUT that inserts an element at the tail of the queue, and GET that removes the element from the head of the queue. We require that if the queue is empty, then the GET subroutine must wait until an element is inserted by a call to the PUT subroutine. We also require that the queue have a capacity of at least m elements, but do not place any upper bound on its capacity.

The crucial part of writing the specification is deciding what the state functions should be. The state functions will have to be definable in terms of the program state of the implementation. To obtain the most general specification—one permitting the widest variety of implementations—one must choose state functions that can be defined for any possible implementation. One natural choice is a state function *queue* indicating the current contents of the queue. When a call of the PUT subroutine is being executed with an argument v—the element to be put on the queue—there will be a period of time between entering the subroutine and placing v on the queue. We need a state function *parg* to indicate the value

module QUEUE **with subroutines** PUT, GET

state functions:

queue: **sequence of** *elements*
parg : *element* **or** *NULL*
gval : *element* **or** *NULL*

initial conditions:

$|queue| = 0$

safety properties:

1. (a) $at(\text{PUT}) \supset parg = \text{PUT.PAR}$
   (b) $after(\text{PUT}) \supset parg = NULL$
2. (a) $at(\text{GET}) \supset gval = NULL$
   (b) $after(\text{GET}) \supset \text{GET.PAR} = gval$
3. **allowed changes to** *queue*
   parg **when** $in(\text{PUT})$
   gval **when** $in(\text{GET})$
   (a) $\alpha[\text{QUEUE}]: in(\text{PUT}) \wedge parg \neq NULL \rightarrow parg' = NULL \wedge queue' = queue * parg$
   (b) $\alpha[\text{QUEUE}]: in(\text{GET}) \wedge gval = NULL \wedge |queue| > 0 \rightarrow gval' \neq NULL \wedge queue = gval' * queue'$

liveness properties:

4. $in(\text{PUT}) \wedge |queue| < m \rightsquigarrow after(\text{PUT})$
5. $in(\text{GET}) \wedge |queue| > 0 \rightsquigarrow after(\text{GET})$

Fig. 5. Specification of a FIFO queue.

of the element that is about to be put on the queue. When control is at the entry point, this value will equal the value of the program variable PUT.PAR. However, the argument may be moved out of PUT.PAR before being placed on the queue, so *parg* will not necessarily equal PUT.PAR after leaving the entry point. We let *parg* assume the value *NULL* when the element is placed on the queue.

We use a similar state function *gval* to indicate the value that the GET subroutine has just taken off the queue and is about to return as its result. We let *gval* have the *NULL* value before GET has removed an element from the queue.

The specification of the QUEUE module is given in Figure 5. We let $|queue|$ denote the length of the queue and let $*$ denote concatenation, where the head of the queue is on the left. The initial condition simply states that the queue is initially empty. Properties 1(a) and (b) state that *parg* equals the argument to PUT when at the entry point, and is *NULL* when at the exit point. Since a *NULL* value of *parg* denotes that the argument has already been put on the queue, Property 1(b) states that the PUT subroutine cannot exit before it has put its element on the queue. Properties 2(a) and (b) are the analogous ones for GET.

Property 3 specifies when these values may change. The value of *parg* is meaningful only when control is in PUT or at its exit point, and Property 1(b) defines its value at the exit point. Hence, we need only specify how *parg* changes when control is in PUT, which explains the "**when** $in(\text{PUT})$" of Property 3. Similar reasoning applies to *gval*. There are two changes that can occur: the first when an element is put on the queue by PUT, and the second when an element is removed from the queue by GET. The latter change can occur only when the queue is nonempty.

Properties 1–3 are safety properties, and they do not imply that PUT and GET actually do anything. For that we need liveness properties. The liveness properties

for PUT specify under what circumstances a call to PUT must succeed in putting its argument into the queue and returning. We do not want to require that PUT always do this, because a program could call PUT arbitrarily many times without ever calling GET. Requiring PUT always to return would require that the queue be able to hold arbitrarily many elements, which is not possible for any real implementation. We therefore require in Liveness Property 4 only that PUT must return (which by Properties 1 and 3(a) it can only do after putting its argument into the queue) if there are fewer than m elements in the queue, for some constant m. This means that in any implementation, the queue must have room for at least m elements. Property 5 states the analogous requirements for the GET subroutine.

Note that had we used *at*(PUT) rather than *in*(PUT) in Property 4, we would have obtained too weak a property, for the following reason. If PUT were called with m elements in the queue, and left its entry point before any elements were removed from the queue, then the weaker property would imply nothing about the future behavior of the PUT subroutine. Even if GET were called repeatedly to remove all the elements of the queue, PUT would not be required to do anything.

We specified a minimum capacity of m for the queue, but did not specify any maximum capacity. Had we wanted to specify that the queue should hold no more than n elements, we could either have added the property $|queue| \leq n$ or else added the clause $|queue| < n$ to the enabling condition of transition specification (a) in Property 3. We would then get a specification in which the PUT subroutine must wait until there are fewer than n elements in the queue before it can add its argument and return.

We specified that the GET subroutine must wait until there is an element in the queue. It is just as easy to write a different specification in which GET must return a special value if it finds the queue empty. We suggest this as an exercise for the reader.

3.4.2 *An Implementation.* In this specification, we have required that the queue behave as if adding or removing an element from it were atomic operations. This does not mean that the entire PUT and GET subroutines have to be implemented as single atomic actions. It does mean that when the PUT operation is adding an element to the queue, there must be some instant at which that element becomes visible to the GET subroutine, and similarly some instant at which the GET operation finishes removing the element from the queue. If there were not such an instant, then the GET subroutine might try to remove an element that the PUT subroutine had not finished putting in the queue, obtaining only part of the element.[1]

We illustrate this with the implementation in Figure 6, where we assume that the queue elements are N-bit integers. The array Q is used as a ring buffer, with HEAD pointing to the element holding the head of the queue, and TAIL pointing

[1] We could actually write a somewhat more general specification in which an element "flickered" for a while after it was put in the queue, and while it was "flickering" the GET subroutine nondeterministically might or might not see it. However, this generalization is of little practical interest.

```
module QUEUE
  global variables
  Q      :array indexed by 0:m − 1 of n-bit numbers
  HEAD:integer initially 0
  TAIL  :integer initially 0

begin subroutine PUT
    declare I integer;
    while a: ⟨TAIL − HEAD = m⟩ do b: ⟨skip⟩ od;
  c: ⟨I := 0⟩;
    while d: ⟨I < N⟩
          do e: ⟨shift.left 1 (Q[TAIL mod m], PUT.PAR)⟩;
             f: ⟨I := I + 1⟩                                    od;
  g: ⟨TAIL := TAIL + 1⟩
  h:
end subroutine

begin subroutine GET
    declare J integer;
    while r: ⟨TAIL − HEAD = 0⟩ do s: ⟨skip⟩ od;
  t: ⟨J := 0⟩;
    while u: ⟨J < N⟩
          do v: ⟨shift.right 1 (Q[HEAD mod m], GET.PAR)⟩
             w: ⟨J := J + 1⟩                                    od;
  x: ⟨HEAD := HEAD + 1⟩
  y:
end subroutine

end module
```

Fig. 6. An implementation of the QUEUE module.

to the element holding its tail. For simplicity, we let HEAD and TAIL be integers, and use their values modulo m as pointers. In a more realistic implementation, they would be integers modulo $2m$, but this makes the reasoning slightly more complicated. To emphasize that the adding and removing of elements from the queue need not be atomic, these operations are performed by shifting the elements one bit at a time out of or into the ".PAR" variable. This is done with *shift.left* and *shift.right* operations, whose meaning should be obvious. Atomic operations are enclosed by angle brackets. Note that the queue has space for only m elements, and a PUT operation must wait until there is room to add the element.

To prove that this implementation meets the specification of Figure 5, we must first define the state functions *queue*, *parg*, and *gval* in terms of the program state. This requires deciding at what point during the execution of each subroutine the change to the queue is considered to have taken place. It is most convenient to consider the queue to change when HEAD or TAIL is incremented. This leads to the following definitions, where

right.half(shift.right i (p, q))

denotes the right half of the double-length word obtained by applying the *shift.right* i operation to the double-length word (p, q), and

left.half(shift.left i (p, q))

has the analogous meaning.

$parg \equiv$ **if** $at(a) \vee at(b) \vee at(c)$
then PUT.PAR
else if $at(d) \vee at(e) \vee at(f) \vee at(g)$
then right.half(shift.right $\bar{\mathrm{I}}$ (Q[TAIL *mod* m], PUT.PAR))
else *NULL*
where $\bar{\mathrm{I}} \equiv$ **if** $at(f)$ **then** I + 1
else I.

$gval \equiv$ **if** $at(y)$ **then** GET.PAR
else *NULL*.

$queue \equiv$ **if** $at(u) \vee at(v) \vee at(w) \vee at(x)$
then left.half(shift.left $\bar{\mathrm{J}}$ (Q[HEAD *mod* m], GET.PAR))
$*$ Q[HEAD $-$ 1 *mod* m] $* \cdots *$ Q[TAIL *mod* m]
else Q[HEAD *mod* m] $* \cdots *$ Q[TAIL *mod* m]
where $\bar{\mathrm{J}} \equiv$ **if** $at(w)$ **then** J + 1
else J.

The initial condition and Properties 1 and 2 follow immediately from these definitions. To prove Property 3, we must show that every atomic action of each process either does not change the value of any of the state functions, or else changes them as prescribed by one of the two transition specifications. We leave the details of this to the reader. Note that such a proof is essentially an invariant assertion proof of the program's safety properties.

A method for proving liveness properties such as Properties 4 and 5 is described in [8] and is not discussed here.

In the above proof, we relied upon the fact that the operations of incrementing HEAD and TAIL were taken to be atomic. This same basic algorithm can be used even when the only atomic operations are reads and writes of single bits, using the techniques of [7] to implement the tests for whether the queue is empty or full (actions a and r). The state function definitions for this finer-grained implementation are more complicated, but the basic idea remains the same—we first choose when the change to the queue is considered to have taken place, and define the state functions accordingly. This choice is somewhat arbitrary, and there will be many possible ways to define the state functions which satisfy the specification.

Defining the state functions for an implementation essentially requires finding the invariants for its correctness proof. This is more difficult for a finer-grained implementation than for a coarser-grained one, since the finer-grained program has more possible execution sequences and is therefore harder to prove correct.

3.5 A Lossy Transmission Protocol

We now specify a message transmission protocol, in which messages are sent by calling a subroutine TMT and are received by calling a subroutine RCV. Messages are to be queued and delivered in the order in which they are sent. However, messages are allowed to be lost.

Were we to require that no messages be lost, then the QUEUE module would provide such a protocol, with PUT serving as TMT and GET serving as RCV. (Although we gave an implementation as a two-process shared memory program, nothing in the specification prevents an implementation with the PUT and GET subroutines in two separate computers.) An examination of Figure 5 reveals that to allow messages to be lost, we need merely add to Property 3 a transition

module XMIT with subroutines TMT, RCV

state functions:
queue: **sequence of** *elements*
targ : *element* **or** *NULL*
rval : *element* **or** *NULL*

initial conditions:
$|queue| = 0$

safety properties:
1. (a) $at(\text{TMT}) \supset targ = \text{TMT.PAR}$
   (b) $after(\text{TMT}) \supset targ = NULL$
2. (a) $at(\text{RCV}) \supset rval = NULL$
   (b) $after(\text{RCV}) \supset RCV.PAR = rval$
3. **allowed changes to** *queue*
   targ **when** $in(\text{TMT})$
   rval **when** $in(\text{RCV})$
   (a) $\alpha[\text{XMIT}]: in(\text{TMT}) \wedge targ \neq NULL \rightarrow$
   $targ' = NULL \wedge queue' = queue * targ$
   (b) $\alpha[\text{XMIT}]: in(\text{RCV}) \wedge rval = NULL \wedge |queue| > 0 \rightarrow$
   $rval' \neq NULL \wedge queue = rval' * queue'$
   (c) **for all** $Q: \alpha[\text{XMIT}]: queue = Q \rightarrow queue' \prec Q$

liveness properties:
4. $in(\text{TMT}) \wedge |queue| < m \rightsquigarrow after(\text{TMT})$
5. $in(\text{RCV}) \wedge \Box\Diamond|queue| > 0 \rightsquigarrow after(\text{RCV})$

Fig. 7. Specification of a transmission protocol.

specification in which an element is removed from the queue. (Note that adding a property strengthens a specification, while adding a transition specification to an **allowed changes** property weakens the specification.) This is done in Figure 7, where we have named the module XMIT and changed PUT and GET to TMT and RCV. In the new Property 3(c), $\prec$ denotes the relation "is a proper subsequence of".

We have made one additional change to the QUEUE specification in Figure 7: we have weakened Liveness Property 5. Property 5 for the QUEUE module states that if control is in the GET subroutine and the queue is nonempty, then GET must eventually remove an element from the queue. This requirement cannot be met if the queue can lose elements, since control can be in RCV when the queue is nonempty, but all the elements could disappear from the queue before the RCV subroutine had a chance to remove one. In order for the protocol to be useful, we need a liveness condition which guarantees that some messages are eventually received. (Otherwise, an implementation that simply threw away all messages would be correct.) There are a number of possible conditions we could require. The one we choose states that if the queue is infinitely often nonempty, then a message will be received. Thus, if the sender keeps issuing TMT calls, then eventually a message will be received.

Any single message sent with a TMT call could be lost. However, if the sender keeps calling TMT to send the same message, and the receiver keeps calling RCV, then the message will eventually be received. This is expressed formally by the following property:

$$\text{ML. } \Box\Diamond in(\text{TMT}) \wedge \Box(at(\text{TMT}) \supset \text{TMT.PAR} = msg) \wedge \Box\Diamond in(\text{RCV}) \rightsquigarrow after(\text{RCV}) \wedge \text{RCV.PAR} = msg.$$

To see why this always holds, observe that $\Box\Diamond in(\text{RCV})$ means that either (i) RCV is entered and exits infinitely many times, or (ii) eventually RCV is entered and never exits. Suppose that (ii) holds. Then by Property 5, this means that $\Box\Diamond|queue| > 0$ is false, which implies that eventually a time is reached after which the queue remains empty forever. However, Property 4 and the hypothesis $\Box\Diamond in(\text{TMT})$ then imply that TMT must eventually add an element to the queue, which is a contradiction. Hence, (ii) is impossible, so control must enter and exit RCV infinitely many times. Each time RCV exits, it removes an element from the queue. Since the queue can have only a finite number of elements different from *msg*, eventually RCV must remove *msg* from the queue and exit with RCV.PAR = *msg*, proving the property. The proof lattice method of [8] can be used to convert this informal reasoning into a formal temporal logic proof.

Property ML also holds after RCV has exited with the value *msg*, so RCV must eventually exit again with the same value. In fact, the same message must be received an infinite number of times. Hence, the following stronger version of ML must hold.

$$\text{ML}'.\ \Box\Diamond in(\text{TMT}) \wedge \Box(at(\text{TMT}) \supset \text{TMT.PAR} = msg) \wedge \Box\Diamond in(\text{RCV}) \rightsquigarrow \Box\Diamond(after(\text{RCV}) \wedge \text{RCV.PAR} = msg).$$

Liveness Properties ML and ML′ may not seem very interesting, since in real programs the sender does not keep transmitting the same message forever. However, we shall see below how they can be quite useful for reasoning about a more realistic protocol.

3.6 The Alternating-Bit Protocol

As our final example, we consider a standard problem from the domain of protocols: the specification of an "alternating-bit" communication protocol. We must specify three separate modules.

—A SENDER module, with a SEND subroutine that is used to send messages.
—A RECEIVER module, with a RECEIVE subroutine that is used to receive messages.
—A TRANSMISSION_MEDIUM module, used by the SENDER and RECEIVER modules to communicate messages and acknowledgments.

The SEND and RECEIVE subroutines must implement a lossless, queued transmission line—that is, they must satisfy the specifications for the PUT and GET subroutines of the QUEUE module. Hence, these three modules must together implement the QUEUE module.

In the terminology generally used to describe protocols [1], the specification of the QUEUE module is the "service specification", and the specifications of the SENDER, RECEIVER, and TRANSMISSION_MEDIUM modules comprise the "protocol specification". We specify these three modules, and sketch a proof that they implement the QUEUE module.

We let the TRANSMISSION_MEDIUM module consist of two "copies" of the XMIT module—a module MXMIT by which the sender sends messages to the receiver, and a module AXMIT by which the receiver sends acknowledgments

module MXMIT **with subroutines** MTMT, MRCV

state functions:
mqueue: **sequence of** ⟨**integer mod** 2, *message*⟩
mtarg : ⟨**integer mod** 2, *message*⟩ **or** *NULL*
mrval : ⟨**integer mod** 2, *message*⟩ **or** *NULL*
⋮

module AXMIT **with subroutines** ATMT, ARCV

state functions:
aqueue: **sequence of integer mod** 2
atarg : **integer mod** 2 **or** *NULL*
arval : **integer mod** 2 **or** *NULL*
⋮

Fig. 8. The TRANSMISSION_MEDIUM module.

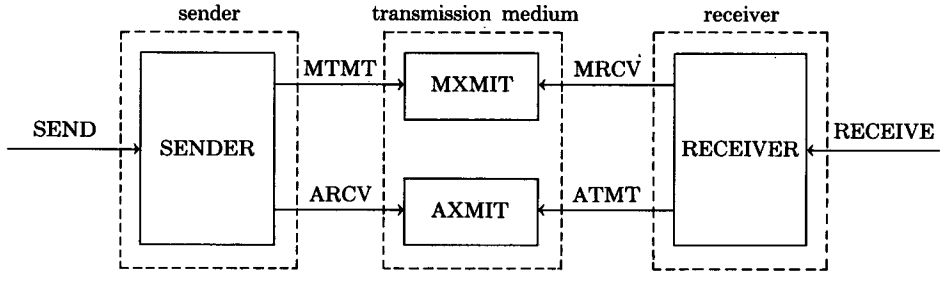

Fig. 9. The modules of the alternating-bit protocol.

to the sender.[2] The TRANSMISSION_MEDIUM module thus provides lossy two-way transmission between the sender and receiver.

Figure 8 indicates how the subroutines and state functions of these modules are named. The initial conditions and properties are obtained from Figure 7 by the obvious renamings. Figure 9 shows the different modules and the subroutine calls by which they interact.

The sender maintains a queue of messages waiting to be sent. It transmits a message M from this queue by sending the message $\langle b, M \rangle$, where b is a sequence number equal to either zero or one. The sender repeatedly sends this same message $\langle b, M \rangle$ until it receives the acknowledgment message b. It then removes the message M from its queue and transmits the next message M' from the queue by sending $\langle b + 1, M' \rangle$—where addition is modulo two. The sender transmits messages by calling MTMT, and receives acknowledgments by calling ARCV. Messages to be sent are added to its queue by calls to the SEND subroutine.

When the receiver gets a message $\langle b, M \rangle$, it places it on a queue of received messages and sends the acknowledgment message b. It ignores any messages with the same message number b as it waits for the next message $\langle b + 1, M' \rangle$. The receiver obtains the messages by calling MRCV, and sends acknowledgments by

[2] Although we have not formally defined the concept of a module composed of submodules, it should be self-evident.

module SENDER **with subroutine** SEND

state functions:

squeue : **sequence of** *messages*
snum : **integer mod** 2
sarg : *message* **or** *NULL*
stmtarg: ⟨**integer mod** 2, *message*⟩ **or** *NULL*
srcvval : **integer mod** 2 **or** *NULL*

initial conditions:

1. $|squeue| = 0$
2. $snum = 0$
3. $stmtarg = NULL \wedge srcvval = NULL$

safety properties:

1. (a) at(SEND) $\supset sarg =$ SEND.PAR
   (b) $after$(SEND) $\supset sarg = NULL$
2. (a) at(MTMT) $\supset$ MTMT.PAR $= stmtarg \neq NULL$
   (b) $after$(MTMT) $\supset stmtarg = NULL$
3. (a) at(ARCV) $\supset srcvval = NULL$
   (b) $after$(ARCV) $\supset srcvval =$ ARCV.PAR
4. **allowed changes to** *squeue*
   snum
   sarg **when** in(SEND)
   srcvval **when** $\sim in$(ARCV)
   (a) α[SENDER]: in(SEND) $\wedge$ $sarg \neq NULL \rightarrow$
   $sarg' = NULL \wedge squeue' = squeue * sarg$
   (b) α[SENDER]: $\sim in$(ARCV) $\wedge$ $srcvval = snum \rightarrow$
   $srcvval' = NULL \wedge squeue' = tail(squeue) \wedge snum' = snum + 1$
   (c) α[SENDER]: $\sim in$(ARCV) $\wedge$ $srcvval \neq NULL \wedge srcvval \neq snum \rightarrow$
   $srcvval' = NULL$
5. **allowed changes to** *stmtarg* **when** $\sim in$(MTMT)
   α[SENDER]: $stmtarg = NULL \wedge |squeue| > 0 \rightarrow stmtarg' = \langle snum, head(squeue)\rangle$

liveness properties:

6. in(SEND) $\wedge$ $|squeue| < sm$ $\rightsquigarrow$ $after$(SEND)
7. $\sim in$(MTMT) $\wedge \Box |squeue| > 0$ $\rightsquigarrow$ at(MTMT)
8. $\sim in$(ARCV) $\wedge \Box |squeue| > 0$ $\rightsquigarrow$ at(ARCV)
9. $after$(ARCV) $\rightsquigarrow srcvval = NULL$

Fig. 10. Specification of the SENDER module.

calling ASND. Messages are removed from its queue by calls to the RECEIVE subroutine.

3.6.1 *Specification of the Sender.* The specification of the sender is given in Figure 10. The state functions have the following meanings.

squeue: The sender's queue of messages still to be sent. Its head will contain the message currently being sent.
snum: The sequence number of the message currently being sent.
sarg: Plays the same role for the SEND subroutine that *parg* does for the PUT subroutine.
stmtarg: The argument of the next call to MTMT, or *NULL* if the argument for the next call has not yet been determined.

srcvval: The value returned by the last call to ARCV that is waiting to be examined, or *NULL* if that value was already processed.

For a truly general specification, the state functions should describe information that must be contained in the program state of any real implementation. It is reasonably clear that for any implementation, it must be possible to define the queue of messages waiting to be sent and the current sequence number. The need for a state function describing the argument of the current call to SEND was explained in the discussion of the QUEUE module.

There must be some point in the execution at which the argument for the next call of MTMT is chosen, and the value of that argument must then be derivable from the program state until MTMT is called. When it first becomes different from *NULL*, the value of *stmtarg* should equal $\langle snum, head(squeue) \rangle$, but *snum* and *squeue* could change before MTMT is actually called. Similarly, the value returned by the most recent call to ARCV must be derivable from the state until that value is acted upon.

It is important to realize that the implementation need not contain any explicit data structures corresponding to these state functions. Instead of keeping the sequence number of the next message to be sent, the implementation might have a program variable that holds the sequence number of the last message sent. Instead of a queue of messages waiting to be sent, the implementation might have a queue of unsent messages plus a buffer to hold the last message sent. However, we believe that in any implementation satisfying the above informal description of what the sender does, these five state functions will be definable in terms of the program state. Actually defining these state functions is tantamount to proving the correctness of the implementation, and can be fairly difficult.

The initial conditions are reasonably obvious. The choice of zero as the initial value of *snum* is arbitrary, but we wish to specify the first sequence number that will be sent so that we can specify the first sequence number that the receiver should expect.

Property 1 is the obvious analogue of Property 1 of the QUEUE module's specification. Property 2 is a similar property for the *stmtarg* function. However, note that whereas *sarg* is "getting its value from" SEND.PAR, *stmtarg* is "giving its value to" MTMT.PAR. Property 2(a) states that *stmtarg* is the value with which MTMT is called, and Property 2(b) states that *stmtarg* should be "reset to *NULL*" by the execution of MTMT. Property 3 is a similar property for *srcvval*, which "obtains its value from" ARCV.PAR upon exit from the ARCV subroutine. Property 3(a) means that ARCV is not called until the last value returned by ARCV has been processed.

Property 4 is obtained by considering when *squeue* is allowed to change. Elements are added to *squeue* only by calls to SEND, and this possibility is indicated by transition specification 4(a). It is the analogue of 3(a) of the QUEUE specification, and is the only way in which *sarg* can change while control is in the SEND subroutine. Elements are deleted from *squeue* only when an acknowledgment is received containing the current sequence number. This is indicated by transition specification 4(b). This transition also increments *snum* and sets *srcvval* to *NULL*, indicating that the last acknowledgment received has been processed. This is the only occasion on which *snum* can change. Finally, the only other time that *srcvval* can change when control is not in ARCV is when an

acknowledgment for a previous message is processed, and this is indicated by transition specification 4(c). We do not specify what value *srcvval* should have when control is inside ARCV.

Note that we have specified that the sender never ignore an acknowledgment for the current message. The protocol would still work—that is, it would still implement the QUEUE module—even if the sender threw away some acknowledgments. We could have allowed that possibility in our specification, but chose not to for simplicity.

The only state function not constrained by Property 4 is *stmtarg*. When control is not in the MTMT subroutine, it can change only when the SENDER module determines the value of the next call to MTMT, as indicated by Property 5. We do not constrain the value of *stmtarg* when control is in MTMT.

Liveness Property 6 is the analogue of Property 4 of the QUEUE module, where *sm* denotes the minimum capacity of the sender's queue. Property 7 states that the sender must keep sending messages as long as its queue is not empty. To see why the "□" is necessary, consider the situation in which the last call to MTMT has exited and the queue has a single element, but before the sender decides to call MTMT again it receives an acknowledgment and deletes the one remaining message from the queue. Without the "□", Property 7 would force the sender to call MTMT again, even though it has nothing more to send. Remembering the definition of $\rightsquigarrow$, we can rewrite Property 7 as

$$\Box[\sim in(\text{MTMT}) \supset (\Diamond at(\text{MTMT}) \vee \Diamond|squeue| = 0)],$$

which the reader may find more agreeable.

Property 8 similarly states that the sender must keep calling ARCV to receive acknowledgments when its queue is nonempty. Finally, Property 9 states that the sender must eventually process the acknowledgment that it receives.

3.6.2 *Specification of the Receiver.* The receiver's specification is given in Figure 11. The state functions have the following interpretations.

rqueue: The queue of messages received from the sender.
rnum: The sequence number of the most recent message received.
rval: Similar to *gval* for the GET subroutine.
rtmtarg: The argument for the next call of ATMT, or *NULL* if that argument has not yet been determined.
rrcvval: The last message received that is waiting to be processed, or *NULL* if there is no such unprocessed message.

The specifications of the sender and the receiver are symmetric, so we do not explain this specification. The only asymmetry is in Liveness Property 7, which states that the receiver must eventually send an acknowledgment after receiving a message.

3.6.3 *Correctness of the Protocol.* We now sketch the proof that the three modules TRANSMISSION_MEDIUM, SENDER, and RECEIVER correctly implement the QUEUE module, with SEND as the PUT subroutine and RECEIVE as the GET subroutine. To do this, we must first define the state functions of the QUEUE module in terms of the state functions of the three implementing modules. This is done as follows.

module RECEIVER **with subroutine** RECEIVE

state functions:

rqueue : **sequence of** *messages*
rnum : **integer mod** 2
rval : *message* **or** *NULL*
rtmtarg: **integer mod** 2 **or** *NULL*
rrcvval : ⟨**integer mod** 2, *message*⟩ **or** *NULL*

initial conditions:

1. $|rqueue| = 0$
2. $rnum = 1$
2. $rtmtarg = NULL \wedge rrcvval = NULL$

safety properties:

1. (a) at(RECEIVE) $\supset rval = NULL$
   (b) $after$(RECEIVE) $\supset rval =$ RECEIVE.PAR $\neq NULL$
2. (a) at(ATMT) $\supset$ ATMT.PAR $= rtmtarg \neq NULL$
   (b) $after$(ATMT) $\supset rtmtarg = NULL$
3. (a) at(MRCV) $\supset rrcvval = NULL$
   (b) $after$(MRCV) $\supset rrcvval =$ MRCV.PAR
4. **allowed changes to** *rqueue*
   rnum
   rval **when** in(RECEIVE)
   rrcvval **when** $\sim in$(MRCV)
   (a) α[RECEIVER]: in(RECEIVE) $\wedge$ $rval = NULL \wedge |rqueue| > 0 \rightarrow$
   $rval' \neq NULL \wedge rqueue = rval' * rqueue'$
   (b) α[RECEIVER]: $\sim in$(MRCV) $\wedge$ $rrcvval \neq NULL \wedge first(rrcvval) \neq rnum \rightarrow$
   $rrcvval' = NULL \wedge rqueue' = rqueue * second(rrcvval)$
   $\wedge$ $rnum' = first(rrcvval)$
   (c) α[RECEIVER]: $\sim in$(MRCV) $\wedge$ $rrcvval \neq NULL \wedge first(rrcvval) = rnum$
   $\rightarrow rrcvval' = NULL$
5. **allowed changes to** *rtmtarg* **when** $\sim in$(ATMT)
   α[RECEIVER]: $rtmtarg = NULL \rightarrow rtmtarg' = rnum$

liveness properties:

6. in(RECEIVE) $\wedge$ $|rqueue| > 0 \rightsquigarrow after$(RECEIVE)
7. $after$(MRCV) $\rightsquigarrow at$(ATMT)
8. $\sim in$(MRCV) $\wedge \Box |rqueue| < rm \rightsquigarrow at$(MRCV)
9. $after$(MRCV) $\rightsquigarrow rrcvval = NULL$

Fig. 11. Specification of the RECEIVER module.

queue ≡ **if** *snum* ≠ *rnum*
then *rqueue* * *squeue*
else *rqueue* * *tail*(*squeue*).
parg ≡ *sarg*.
gval ≡ *sval*.

It is easy to see that the initial conditions for the SENDER and RECEIVER modules imply that *queue* is initially empty, which is the initial condition for the QUEUE module.

Properties 1 and 2 of the QUEUE module follow immediately from Property 1 of the SEND module and Property 1 of the RECEIVE module, respectively.

To prove Property 3 of the QUEUE module, we first observe that changes to *queue*, *parg*, and *gval* can be caused only by changes to *squeue*, *snum*, *rqueue*,

and *rnum*. All the allowed changes to these latter state functions are specified in Property 4 of the SENDER and RECEIVER modules. We therefore have to show that the only changes to *queue*, *parg*, and *gval* allowed by Property 4 of the SENDER and RECEIVER are changes that are allowed by Property 3 of the QUEUE module.

It is easy to verify that transitions allowed by the SENDER's transition specification 4(a) cause changes to *queue* and *parg* that are allowed by the QUEUE's transition specification 3(a). Similarly, 4(a) of the RECEIVER allows only transitions that are allowed by 3(b) of the QUEUE. Transition specifications 4(c) of both the SENDER and the RECEIVER obviously do not allow any changes to the state functions of the QUEUE.

To complete the verification of the QUEUE's safety properties, we must prove that transition specifications 4(b) of both the SENDER and the RECEIVER do not change the value of *queue*. Proving this for 4(b) of the sender essentially shows that the sender does not delete a message from its queue until it has been received by the receiver, and proving it for 4(b) of the receiver shows that the receiver adds to its queue only messages that it has not already received. Thus, these proofs demonstrate that the alternating-bit protocol does not lose messages or create duplicate messages. These are the basic safety properties of the protocol, and we should not expect the proofs to be trivial.

We begin by defining a state function *xqueue* for the XMIT module (Figure 7) "extending" the module's queue with the element that has just been removed from it and the element about to be added to it—if there are such elements. It is defined formally as follows.

$$
\begin{aligned}
&\mathit{xqueue} \equiv \mathit{xhead} * \mathit{queue} * \mathit{xtail} \\
&\qquad \text{where} \quad \mathit{xhead} \equiv \textbf{if } \mathit{in}(\text{RCV}) \textbf{ then } \mathit{rval} \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{NULL} \\
&\qquad\qquad\quad \mathit{xtail} \equiv \textbf{if } \mathit{in}(\text{TMT}) \textbf{ then } \mathit{targ} \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{NULL}.
\end{aligned}
$$

The state function *xqueue* represents the value that the queue would have if elements were added to the queue immediately upon entry to the TMT subroutine, and not removed from the queue until exit from the RCV subroutine. It is useful for proving safety properties of programs that use the XMIT module because it describes what the queue looks like to these programs.

In this way, we define the functions *mxqueue* and *axqueue* for the modules MXMIT and AXMIT, respectively. We further extend *mxqueue* to the state function *xmxqueue* by including the next message that the SENDER has decided to transmit and the last message that the RECEIVER has received but not yet processed. This is done as follows.

$$
\begin{aligned}
&\mathit{xmxqueue} \equiv \mathit{xhead} * \mathit{mxqueue} * \mathit{xmtail} \\
&\qquad \text{where} \quad \mathit{xmhead} \equiv \textbf{if } \sim\mathit{in}(\text{MRCV}) \textbf{ then } \mathit{rrcvval} \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{NULL} \\
&\qquad\qquad\quad \mathit{xmtail} \equiv \textbf{if } \sim\mathit{in}(\text{MTMT}) \textbf{ then } \mathit{stmtval} \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{NULL}.
\end{aligned}
$$

We similarly extend *axqueue* as follows.

$$
\begin{aligned}
xaxqueue \equiv\ & xahead * axqueue * xatail \\
\text{where}\quad & xahead \equiv \textbf{if } \sim in(\text{ARCV}) \textbf{ then } srcvval \\
& \qquad\qquad\qquad\quad \textbf{else } NULL \\
& xatail \equiv \textbf{if } \sim in(\text{ATMT}) \textbf{ then } rtmtval \\
& \qquad\qquad\qquad\quad \textbf{else } NULL.
\end{aligned}
$$

The heart of the proof consists of proving the following invariance property:

XQ. *There exist natural numbers a, b, c, d and a message x such that*

$$xmxqueue = \langle rnum, x\rangle^a * \langle snum, head(squeue)\rangle^b$$

and

$$xaxqueue = (snum + 1)^c * rnum^d,$$

where y^z denotes a string consisting of z copies of y. To prove that XQ is true throughout the execution, we show that it is true in the initial state, and that it is left unchanged by every allowed transition.

It is easy to see that XQ is true initially with $a = b = c = d = 0$. To show that XQ cannot be made false by any allowed transition, we observe that the only transitions that can change *xmxqueue* are the ones allowed by the following transition specifications:

- 3(c) of the MXMIT module—which deletes an element.
- 5 of the SENDER—which inserts an element.
- 4(b) and (c) of the RECEIVER—which remove an element.

Similarly, the only transitions that can change *xaxqueue* are the ones allowed by the following transition specifications:

- 3(c) of the AXMIT module—which deletes an element.
- 5 of the RECEIVER—which inserts an element.
- 4(b) and (c) of the SENDER—which remove an element.

The proof that these transitions leave XQ true is easy, except for 4(b) of the SENDER and RECEIVER. For a transition allowed by 4(b) of the SENDER, note that if XQ holds in the initial state, then the enabling condition $srcvval = snum$ implies that $c = 0$ and $rnum = snum$ in the initial state. We can therefore take $b = 0$ in the initial state. Condition XQ then reads

$$xmxqueue = \langle rnum, x\rangle^a.$$

$$xaxqueue = rnum^d.$$

Since 4(b) does not permit any change to *xmxqueue*, *xaxqueue*, or *rnum*, XQ must hold for the final state of the transition. For a transition allowed by 4(b) of the RECEIVER, we similarly observe that the enabling condition $first(rrcvval) \neq rnum$ implies that $a = 0$ and $snum \neq rnum$ in the initial state. We can therefore take $d = 0$, and XQ holds after the transition. This completes the proof of the invariance property XQ.

To complete our proof of Safety Property 3 for the QUEUE module, we must use property XQ to show that any transition allowed by 4(b) of the SENDER or

RECEIVER leaves *queue* unchanged. For the SENDER, observe that the truth of the enabling condition and the definition of *xaxqueue* imply that *srcvval* is equal to the head of *xaxqueue*. Since the enabling condition states that *srcvval* = *snum*, Property XQ implies that $c = 0$ and *snum* = *rnum*. It then follows easily from the definition of *queue* that it is not changed by the transition.

Similarly, for a transition allowed by 4(b) of the receiver, the enabling condition implies that the head of *xmxqueue* equals *rrcvval*, which is not equal to *rnum*. Property XQ then implies that $a = 0$ and $rnum \neq snum$, and it follows that the transition does not change the value of *queue*. This completes the proof of the QUEUE module's safety properties.

We now sketch the proof of the QUEUE module's Liveness Property 4. Observe that by Property 6 of the SENDER, it suffices to prove

$$in(\text{SEND}) \wedge |queue| < m \rightsquigarrow |squeue| < sm.$$

To do this, we show that if

$$in(\text{SEND}) \wedge |queue| < m \wedge |squeue| \geq sm \tag{1}$$

holds, then eventually an element is removed from *squeue*. The proof is by contradiction. We assume that (1) holds but no element is ever removed from *squeue*, and arrive at a contradiction.

Since $m = sm + rm$, it follows from the definition of *queue* that $|squeue| \geq sm$ and $|squeue| < m$ imply

1. $|rqueue| < rm \wedge snum \neq rnum$, or
2. $|rqueue| \leq rm \wedge snum = rnum$.

We now show that if 1 holds, then eventually 2 will hold. Assume the contrary, that 1 holds but 2 never does. Property 7 of the SENDER then implies that whenever control is not in MTMT it will eventually be in MTMT, which means that control must infinitely often be in MTMT, so we have

$$\Box\Diamond in(\text{MTMT}).$$

Property 4 of the SENDER implies that *snum* can change only when an element is removed from *squeue*, so our assumption that no element is removed from *squeue* means that *snum* does not change. Hence, the SENDER keeps calling MTMT with the same value, and we must eventually have

$$\Box(at(\text{MTMT}) \supset \text{MTMT.PAR} = \langle snum, head(queue)\rangle).$$

Since 2 never holds and *snum* never changes, this means that *rnum* never changes and no new elements are ever added to *rqueue*. This in turn means that $\Box|rqueue| < rm$, so we can can conclude from Property 8 of the receiver that

$$\Box\Diamond in(\text{MRCV}).$$

Combining these three properties with liveness property ML of the MXMIT module shows that eventually MRCV returns the value $\langle snum, head(queue)\rangle$ which leads to condition 2—contradicting our assumption that 2 never holds.

Finally, assume that 2 holds. Again, if no element is removed from *squeue*, then *snum* never changes. By the same reasoning as above, this implies that the SENDER calls MTMT infinitely often with the same message. Applying Property

ML′ of the MXMIT module, we conclude that

$$\Box\Diamond after(\text{MRCV}).$$

From Property 8 of the RECEIVER, we then obtain

$$\Box\Diamond at(\text{ATMT}).$$

In our proof of the safety properties of the QUEUE module, we showed that if *snum* = *rnum*, then *squeue* is nonempty. (Otherwise, our definition of *queue* would not make sense.) It then follows from the first part of Property XQ that when *snum* = *rnum*, *rnum* cannot change until *snum* does. Hence, *rnum* never changes, so we eventually must have

$$\Box(at(\text{ATMT}) \supset \text{ATMT.PAR} = rnum)$$

for the constant value of *rnum*. Combining these properties with Property 7 of the RECEIVER, we can apply Property ML of the AXMIT module to conclude that the ARCV subroutine eventually returns with the value *rnum* = *snum*. It then follows from the properties of the SENDER that an element must be deleted from *squeue*, which is the required contradiction.

The proof of Property 5 of the QUEUE module is similar. Note the strong reliance upon proof by contradiction, which is typical of temporal logic liveness proofs. It is because of this reductio ad absurdum style of reasoning that properties ML and ML′ of the XMIT module can be used.

4. CONCLUSION

We have chosen the alternating-bit protocol as our major example because it has also been specified using a number of other methods. We briefly discuss a few of these methods here, and refer the reader to [10] for a more complete survey.

There are two basic ways to specify a program:

—By an abstract program that describes its behavior.
—By a collection of properties that it must satisfy.

The first kind of specification, which includes state machine [4] and Petri net [3] specifications, has usually proved to be easier to understand, since writing programs provides a natural method for describing programs. However, this tends to produce overly restrictive specifications that describe how the program should be implemented rather than what it should do. If one chooses an implementation different from the one envisioned when writing the specification, then verifying its correctness becomes a difficult problem of proving the equivalence of two concurrent programs, and requires complex reasoning about behaviors.

In principle, we find the idea of specifying a program in terms of the properties it must satisfy to be very attractive. Moreover, temporal logic seems to be a very convenient tool for expressing these properties. However, we have found previous specification methods using temporal logic to be unsatisfactory. The method of Schwartz and Melliar-Smith [11] requires complicated temporal logic expressions—in particular, expressions involving nested "until" operators. Not only are these expressions hard to understand, but we have found that it is difficult to use them for reasoning about the program. Such expressions are obtained because specification of the program state is replaced by temporal specifications. For

example, to specify the safety properties of the SQUARE subroutine, instead of using the state function *val*, Schwartz and Melliar-Smith would simply assert that control cannot reach the exit with a result v unless it had entered the subroutine with an argument a such that $v = a^2$. They feel that theirs is a more abstract specification because it does not mention the program state. This is true only if abstractness is taken to be an undefinable aesthetic property, since their specification is no more general than ours. Any implementation satisfying their purely temporal specification will also satisfy our specification; it will be possible to define the state function *val* because the value that the subroutine eventually returns must be determined by its current state.

The method of Hailpern and Owicki [2], which uses histories, produces specifications that are quite simple and easy to understand. They would specify the SQUARE subroutine by stating that each element in the sequence of returned values is the square of the corresponding element in the sequence of arguments with which the subroutine is called. Their specifications look even simpler than the ones produced by our method. However, a specification should not be judged on how simple it looks, but on how easy it is to use. There are two ways in which the formal specification of a module is used:

—To determine the correctness of an implementation of the module.
—To prove properties of programs using the module.

To prove that a program implements a specification based upon histories, one must change the program by adding dummy history variables and assignments to them—for example, a variable that records the sequence of values returned by a subroutine. These dummy variables and statements have nothing to do with the behavior of the program, and they are not needed for reasoning about it. Similarly, the histories are irrelevant for reasoning about a program that uses the specified module. Moreover, we feel that the presence of histories in the module specification will encourage behavioral reasoning about the program, and experience has shown that such reasoning leads to more complicated proofs—proofs more likely to contain errors—than does assertional reasoning.

In developing our method, we have been guided by our experience in verifying concurrent programs. While specifications written with other methods may appear simpler, we do not think that they will be easier to use. In our examples, we have not just written specifications, but also indicated how those specifications can be used. For the SQUARE and QUEUE modules, we showed how nontrivial implementations can be proved to correctly implement the specifications. In the alternating-bit protocol example, we showed how the specifications of the four modules could be used to prove properties of a program containing them—namely, that the SEND and RECEIVE subroutines implement a lossless transmission line. The proof may have seemed rather difficult for such a simple protocol. However, we believe that the alternating-bit protocol, like many innocent-looking concurrent programs, is more complicated than it appears. We do not expect any other specification method will permit a simpler proof for specifications as general as our SENDER and RECEIVER modules.

We feel that our proofs demonstrate the feasibility of reasoning with these specifications. However, the proofs were informal. Proofs of liveness properties can be formalized using the approach of [8]. Formalizing the proofs of safety

properties requires formal methods for reasoning about **leaves unchanged** and **allowed changes** properties. Such formal methods do exist, and we believe that they can be used to provide reasonably simple formal proofs. However, further work is needed to demonstrate this.

In our method, we have tried to combine the best features of state machine and temporal logic methods. We have restricted the temporal logic formulas in our specifications to ones that do not use nested "untils". This requires that the specification include a complete set of state functions—complete in the sense that any property can be expressed in terms of the current program state. There are two new ideas that we have introduced in doing this:

—Action sets.
—The **allowed changes** construction.

We had to introduce action sets in order to specify a module without specifying the entire program. This was because the specification had to state properties of the whole program, so some method was needed to indicate which actions were caused by the module and which by the rest of the program.

We have found the **allowed changes** properties to be easy to understand because they look like descriptions of programs. However, they have a corresponding drawback: they make it easy to write a specification that is essentially an abstract program, rather than one stating the properties the program must satisfy. This is avoided by using a minimal set of state functions, so no state function appears in the specification unless it can be defined for every implementation.

The example of the alternating-bit protocol illustrates the problem. We expect that the reader will find our specifications of the SENDER and RECEIVER to be reasonably easy to understand, and will be convinced that they specify behavior consistent with the informal description of the protocol. However, because the specifications involve a fairly arbitrary choice of state functions, we do not expect him to find it obvious that they are very general. Given the informal description of the protocol, he would probably write a specification that uses different state functions and looks quite different from ours. However, we believe that the two specifications would be equivalent. Such an equivalence is demonstrated by showing that each specification is a correct implementation of the other—that is, that each one's state functions can be defined in terms of the other's state functions in such a way that its properties can be proved from the other's properties.

Any powerful specification method will permit equivalent specifications that look quite different. Moreover, as with programs written in a sufficiently expressive programming language, the general problem of determining the equivalence of two specifications will be recursively unsolvable. Hence, we cannot be surprised if two people do not write identical specifications for the same program.

APPENDIX. FORMAL SEMANTICS

A program is a triple (S, A, Σ), where S is a set of states, A is a set of actions, and Σ is a set of infinite sequences of the form

$$\sigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots$$

with the s_i in S and the α_i in A. For any such σ in Σ, the sequence

$$\sigma^{+n} \equiv s_n \xrightarrow{\alpha_{n+1}} s_{n+1} \xrightarrow{\alpha_{n+2}} s_{n+2} \cdots$$

must also be in Σ, for any $n \geq 0$.

A state function is a function whose domain is S and whose range is a subset of some set V of values containing the elements *true* and *false*. A state predicate is a state function whose range is included in {*true*, *false*}. We assume a first-order predicate calculus with equality for state functions, allowing quantification over V. We write $s \models P$ to denote the value of the state predicate P on the element s of S.

An action predicate is a boolean function on A. We let $\alpha \models Q$ denote the value of the action predicate Q on the element α of A.

The formulas of temporal logic are constructed from state predicates and action predicates using ordinary logical operators $\sim$ and $\vee$, quantification over V, and the binary connective $\trianglelefteq$. We inductively define the relation $\sigma \models P$, read "P is valid for the sequence σ", for any sequence

$$\sigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots$$

in Σ and temporal logic formula P as follows.

$\sigma \models P \quad \equiv s_0 \models P$ if P is a state predicate

$\sigma \models P \quad \equiv \alpha_1 \models P$ if P is an action predicate

$\sigma \models \sim P \quad \equiv \sim(\sigma \models P)$

$\sigma \models (P \vee Q) \equiv (\sigma \models P) \vee (\sigma \models Q)$

$\sigma \models \forall v : P \quad \equiv \forall v \in V : \sigma \models P$

$\sigma \models P \trianglelefteq Q \quad \equiv \forall n \geq 0 : (\forall m : 0 \leq m \leq n \supset \sigma^{+m} \models P) \supset \sigma^{+n} \models Q$

The formula P is said to be valid for the program (S, A, Σ), written simply $\models P$, if $\sigma \models P$ is true for all σ in Σ. In [6], the operator $\trianglelefteq$ was denoted by $\Box$. The reason for this new notation will become apparent below.

The unary operators $\Box$ and $\Diamond$ are defined by

$$\Box P \equiv \mathit{true} \trianglelefteq P.$$

$$\Diamond P \equiv \sim \Box \sim P.$$

We also define the binary operator $\triangleleft$ by

$$P \triangleleft Q \equiv (P \vee \sim Q) \trianglelefteq Q.$$

It follows from the definition of $\sigma \models P \trianglelefteq Q$ that

$$\sigma \models P \triangleleft Q \equiv \forall n \geq 0 : (\forall m : 0 \leq m < n \supset \sigma^{+m} \models P) \supset \sigma^{+n} \models Q.$$

Thus, $P \trianglelefteq Q$ asserts that Q is true at least as long as P, and $P \triangleleft Q$ asserts that Q is true at least one step longer than P—where "one step longer than" forever is still forever. It is easy to remember their meaning by thinking of $\trianglelefteq$ as "true for a $\leq$ duration", and $\triangleleft$ as "true for a $<$ duration". The two operators satisfy the expected transitivity relations—for example, $P \trianglelefteq Q$ and $Q \triangleleft R$ imply $P \triangleleft R$.

The specification (*) is interpreted as the following (second-order) temporal logic formula:

$$\exists f_1, \ldots, f_n : \Box(f_1 \in R_1 \wedge \cdots \wedge f_n \in R_n) \wedge [I_1 \wedge \cdots \wedge I_m \supset \Box(P_1 \wedge \cdots \wedge P_q)],$$

where $f_i \in R_i$ is a state predicate which, when applied to a state s, has the value *true* if and only if $f_i(s)$ is an element of R_i.

We now interpret the properties P_j as temporal logic formulas. Liveness properties are already written as temporal logic formulas, so we need only interpret our three kinds of safety properties. An invariance property is just a predicate, which is already a temporal logic formula. The formula

a **leaves unchanged** f **when** Q

is interpreted to be the temporal logic formula

$$\forall v : (f = v) \supset [(a \wedge Q) \lhd (f = v)],$$

where we consider the "set of actions" a to be the action predicate which is true for any action α if and only if $\alpha \in a$.

Finally, we define the property

allowed changes to g_1 **when** Q_1,
$\vdots$
g_p **when** Q_p:
$a_1 : R_1 \rightarrow S_1$,
$\vdots$
$a_u : R_u \rightarrow S_u$

to be the conjunction of p **allowed changes to** properties, one for each g_i, all having the same transition specifications. Hence, we need only consider the property

allowed changes to g **when** Q:
$a_1 : R_1 \rightarrow S_1$,
$\vdots$
$a_u : R_u \rightarrow S_u$ (**)

We can replace any single transition specification

$a \vee b : R \rightarrow S$

in (**) by the two transition specifications

$a : R \rightarrow S.$

$b : R \rightarrow S.$

This allows us to rewrite (**) in such a way that all the a_i in (**) are disjoint. Similarly, we can replace the single transition specification

$a : (R \vee R') \rightarrow S$

by the two transition specifications

$a : R \rightarrow S.$

$a : R' \rightarrow S.$

We can therefore assume that all the formulas $a_i \wedge R_i$ in (**) are disjoint. We then define (**) to be the conjunction of the clause

$$\forall v : [g = v] \supset [\sim((a_1 \wedge R_1) \vee \cdots \vee (a_u \wedge R_u))] \lhd [g = v]$$

and u clauses defined as follows, for $i = 1, \ldots, u$:

$$\forall v_1 \cdots \forall v_r : [(h_1 = v_1) \wedge \cdots \wedge (h_r = v_r) \wedge a_i \wedge R_i]$$
$$\supset [a_i \wedge R_i] \lhd [(R_i \wedge g = v) \vee (\sim R_i \wedge S_i')],$$

where the h_r are all the state functions mentioned (with or without primes) in the S_i—including g—and S_i' is the predicate obtained by substituting v_j for h_j and h_j for h_j' in S_i, for each j.

ACKNOWLEDGMENTS

The generalized version of temporal logic described in this paper was developed jointly with Susan Owicki. The work has also been influenced by discussions with Nissim Francez, Shmuel Katz, K. J. Koomen, P. M. Melliar-Smith, Richard Schwartz, Michel Sintzoff, and J. Sifakis.

REFERENCES

1. BOCHMANN, G., AND SUNSHINE, C. Formal methods in communication protocol design. *IEEE Trans. Commun. Com-28*, 4 (Apr. 1980), 624–631.
2. HAILPERN, B.T., AND OWICKI, S.S. Verifying network protocols using temporal logic. In *Proceedings Trends and Applications 1980: Computer Network Protocols.* IEEE Computer Society, 1980, pp. 18–28.
3. HERZOG, O. Static analysis of concurrent processes for dynamic properties using Petri nets. In *Lecture Notes in Computer Science*, vol. 70: *Semantics of Concurrent Computation*, G. Kahn (Ed.). Springer-Verlag, Berlin, 1979, pp. 66–90.
4. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION TC97/SC16/WG1 SUBGROUP B ON STATE MACHINES. A FDT based on an extended state transition model. Working draft of tech. rep., Dec. 1981.
5. LAMPORT, L. The "Hoare logic" of concurrent programs. *Acta Inf. 14*, 1 (June 1980), 21–37.
6. LAMPORT, L. "Sometime" is sometimes "not never": On the temporal logic of programs. In Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nev., Jan. 28–30, 1980, pp. 174–185.
7. LAMPORT, L. Concurrent reading and writing. *Commun. ACM 20*, 11 (Nov. 1977), 806–811.
8. OWICKI, S., AND LAMPORT, L. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 455–495.
9. PNUELI, A. The temporal logic of programs. In 18th Annual Symposium on Foundations of Computer Science, Providence, R.I., Oct. 31–Nov. 2, 1977, pp. 46–57.
10. SCHWARTZ, R., AND MELLIAR-SMITH, P.M. From state machines to temporal logic: Specification methods for protocol standards. *IEEE Trans. Commun. Com-30*, 11 (Nov. 1982).
11. SCHWARTZ, R.L., AND MELLIAR-SMITH, P.M. Temporal logic specification of distributed systems. In *Proceedings of the 2nd International Conference on Distributed Computing Systems.* IEEE Computer Society Press, 1981, pp. 446–454.

Received January 1982; revised June 1982; accepted July 1982