An Old-Fashioned Recipe for Real Time

MARTÍN ABADI and LESLIE LAMPORT
Digital Equipment Corporation

Traditional methods for specifying and reasoning about concurrent systems work for real-time systems. Using TLA (the temporal logic of actions), we illustrate how they work with the examples of a queue and of a mutual-exclusion protocol. In general, two problems must be addressed: avoiding the real-time programming version of Zeno's paradox, and coping with circularities when composing real-time assumption/guarantee specifications. Their solutions rest on properties of machine closure and realizability.

Categories and Subject Descriptors: 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: Theory, Verification

Additional Key Words and Phrases: Composition, concurrent programming, liveness properties, real time, safety properties, temporal logic, Zeno

1. INTRODUCTION

A new class of systems is often viewed as an opportunity to invent a new semantics. A number of years ago, the new class was distributed systems. More recently, it has been real-time systems. The proliferation of new semantics may be fun for semanticists, but developing a practical method for reasoning formally about systems is a lot of work. It would be unfortunate if every new class of systems required inventing new semantics, along with proof rules, languages, and tools.

Fortunately, no fundamental change to the old methods for specifying and reasoning about systems is needed for these new classes. It has long been known that the methods originally developed for shared-memory multiprocessing apply equally well to distributed systems [Chandy and Misra 1988; Lamport 1982]. The first application we have seen of a clearly "off-the-shelf" method to a real-time algorithm was in 1983 [Neumann and Lamport 1983], but there were probably earlier ones. Indeed, the "extension" of an existing temporal logic to real-time programs by Bernstein and Harter [1981] can be viewed as an application of that logic.

The old-fashioned methods handle real time by introducing a variable, which we call *now*, to represent time. This idea is so simple and obvious that it seems hardly

Author's address: Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.

worth writing about, except that few people appear to be aware that it works in practice. We therefore describe how to apply a conventional method to real-time systems.

Any formalism for reasoning about concurrent programs can be used to prove properties of real-time systems. However, in a conventional formalism based on a programming language, real-time assumptions are expressed by adding program operations that read and modify the variable *now*. The result can be a complicated program that is hard to understand and easy to get wrong. We take as our formalism TLA, the temporal logic of actions [Lamport 1994]. In TLA, programs and properties are represented as logical formulas. A real-time program can be written as the conjunction of its untimed version, expressed in a standard way as a TLA formula, and its timing assumptions, expressed in terms of a few standard parameterized formulas. This separate specification of timing properties makes real-time specifications easier to write and understand.

The method is illustrated with two examples. The first is a queue in which the sender and receiver synchronize by the use of timing assumptions instead of acknowledgements. We indicate how safety and liveness properties of the queue can be proved. The second example is an n-process mutual exclusion protocol, in which mutual exclusion depends on assumptions about the length of time taken by the operations. Its correctness is proved by a conventional invariance argument.

We also discuss two problems that arise when time is represented as a program variable—problems that seem to have received little attention—and present new solutions.

The first problem is how to avoid the real-time programming version of Zeno's paradox. If time becomes an ordinary program variable, then one can inadvertently write programs in which time behaves improperly. An obvious danger is deadlock, where time stops. A more insidious possibility is that time keeps advancing but is bounded, approaching closer and closer to some limit. One way to avoid such "Zeno" behaviors is to place an a priori lower bound on the duration of any action, but this can complicate the representation of some systems. We provide a more general and, we feel, a more natural solution.

The second problem is coping with the circularity that arises in open system specifications. The specification of an open system asserts that it operates correctly under some assumptions on the system's environment. A modular specification method requires a rule asserting that, if each component satisfies its specification, then it behaves correctly in concert with other components. This rule is circular, because a component's specification requires only that it behave correctly if its environment does, and its environment consists of all the other components. Despite its circularity, the rule is sound for specifications written in a particular style [Abadi and Lamport 1993; Misra and Chandy 1981; Pnueli 1984]. By examining an apparently paradoxical example, we discover how real-time specifications of open systems can be written in this style.

We express these problems and their solutions in terms of TLA. However, we believe that the problems will arise in any formalism that permits sufficiently general specifications. Our solutions should be applicable to any formalism whose semantics is based on sequences of states or actions.

2. CLOSED SYSTEMS

We briefly review how to represent closed systems in TLA. A closed system is one that is self-contained and does not communicate with an environment. No one intentionally designs autistic systems; in a closed system, the environment is represented as part of the system. Open systems, in which the environment and system are separated, are discussed in Section 4.

We begin our review of TLA in Section 2.1 with an informal presentation of an example. The formal definitions are summarized in Section 2.2. A more leisurely exposition appears in [Lamport 1994], and most definitions in the current paper are repeated in a list in the appendix. Section 2.3 reviews the concepts of safety [Alpern and Schneider 1985] and machine closure [Abadi and Lamport 1991] (also known as feasibility [Apt et al. 1988]) and relates them to TLA, and Section 2.4 defines a useful class of history variables [Abadi and Lamport 1991]. Propositions and theorems are proved in the appendix.

2.1 The Lossy-Queue Example

We introduce TLA with the example of the lossy queue shown in Figure 1. The interface consists of two pairs of "wires", each pair consisting of a *val* wire that holds a message and a boolean-valued *bit* wire. A message m is sent over a pair of wires by setting the *val* wire to m and complementing the *bit* wire. The receiver detects the presence of a new message by observing that the *bit* wire has changed value. Input to the queue arrives on the wire pair $(ival, ibit)$, and output is sent on the wire pair $(oval, obit)$. There is no acknowledgment protocol, so inputs are lost if they arrive faster than the queue processes them. (Because of the way *ibit* is used, inputs are lost in pairs.) The property guaranteed by this lossy queue is that the sequence of output messages is a subsequence of the sequence of input messages. In Section 3.1, we add timing constraints to rule out the possibility of lost messages.

A specification is a TLA formula Π describing a set of allowed behaviors. A property P is also a TLA formula. The specification Π satisfies property P iff (if and only if) every behavior allowed by Π is also allowed by P—that is, if Π implies P. Similarly, a specification Ψ implements Π iff every behavior allowed by Ψ is also allowed by Π, so implementation means implication.

The specification of the lossy queue is a TLA formula that mentions the four variables *ibit*, *obit*, *ival*, and *oval*, as well as two internal variables: q, which equals

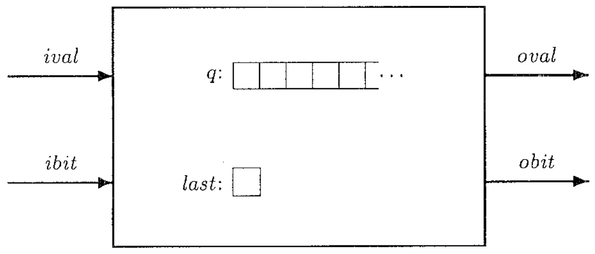

Fig. 1. A simple queue.

$$
\begin{array}{lll}
Init_Q & \triangleq & \land\ ibit, obit \in \{\mathsf{true}, \mathsf{false}\} \\
 & & \land\ ival, oval \in \mathsf{Msg} \\
 & & \land\ last = ibit \\
 & & \land\ q = \langle\langle\rangle\rangle \\
Inp & \triangleq & \land\ ibit' = \neg ibit \\
 & & \land\ ival' \in \mathsf{Msg} \\
 & & \land\ (obit, oval, q, last)' = (obit, oval, q, last) \\
EnQ & \triangleq & \land\ last \neq ibit \\
 & & \land\ q' = q \circ \langle\langle ival \rangle\rangle \\
 & & \land\ last' = ibit \\
 & & \land\ (ibit, obit, ival, oval)' = (ibit, obit, ival, oval) \\
DeQ & \triangleq & \land\ q \neq \langle\langle\rangle\rangle \\
 & & \land\ oval' = Head(q) \\
 & & \land\ q' = Tail(q) \\
 & & \land\ obit' = \neg obit \\
 & & \land\ (ibit, ival, last)' = (ibit, ival, last) \\
\mathcal{N}_Q & \triangleq & Inp \lor EnQ \lor DeQ \\
v & \triangleq & (ibit, obit, ival, oval, q, last) \\
\Pi_Q & \triangleq & Init_Q \land \Box[\mathcal{N}_Q]_v \\
\Phi_Q & \triangleq & \exists q, last : \Pi_Q
\end{array}
$$

Fig. 2. The TLA specification of a lossy queue.

the sequence of messages received but not yet output; and *last*, which equals the value of *ibit* for the last received message. (The variable *last* is used to prevent the same message from being received twice.) These six variables are *flexible variables*; their values can change during a behavior. We also introduce a *rigid variable* Msg denoting the set of possible messages; it has the same value throughout a behavior. We usually refer to flexible variables simply as variables, and to rigid variables as *constants*.

The TLA specification is shown in Figure 2, using the following notation. A list of formulas, each prefaced by $\land$, denotes the conjunction of the formulas, and indentation is used to eliminate parentheses. The expression $\langle\langle\,\rangle\rangle$ denotes the empty sequence, $\langle\langle m \rangle\rangle$ denotes the singleton sequence having m as its one element, $\circ$ denotes concatenation, $Head(\sigma)$ denotes the first element of σ, and $Tail(\sigma)$ denotes the sequence obtained by removing the first element of σ. The symbol $\triangleq$ means *is defined to equal.*

The first definition is of the *predicate* $Init_Q$, which describes the initial state. This predicate asserts that the values of variables *ibit* and *obit* are arbitrary booleans, the values of *ival* and *oval* are elements of Msg, the values of *last* and *ibit* are equal, and the value of q is the empty sequence.

Next is defined the *action Inp*, which describes all state changes that represent the sending of an input message. (Since this is the specification of a closed system, it includes the environment's *Inp* action.) The first conjunct, $ibit' = \neg ibit$, asserts that the new value of *ibit* equals the complement of its old value. The second conjunct asserts that the new value of *ival* is an element of Msg. The third conjunct asserts that the value of the four-tuple $(obit, oval, q, last)$ is unchanged; it is equivalent to the assertion that the value of each of the four variables *obit*, *oval*, q, and *last* is unchanged. The action *Inp* is always *enabled*, meaning that, in any state, a new input message can be sent.

Action *EnQ* represents the receipt of a message by the system. The first conjunct asserts that *last* is not equal to *ibit*, so the message on the input wire has not yet been received. The second conjunct asserts that the new value of q equals the sequence obtained by concatenating the old value of *ival* to the end of q's old value. The third conjunct asserts that the new value of *last* equals the old value of *ibit*. The final conjunct asserts that the values of *ibit*, *obit*, *ival*, and *oval* are unchanged. Action *EnQ* is enabled in a state iff the values of *last* and *ibit* in that state are unequal.

The action *DeQ* represents the operation of removing a message from the head of q and sending it on the output wire. It is enabled iff the value of q is not the empty sequence.

The action $\mathcal{N}_Q$ is the specification's *next-state relation*. It describes all allowed changes to the queue system's variables. Since the only allowed changes are the ones described by the actions *Inp*, *EnQ*, and *DeQ*, action $\mathcal{N}_Q$ is the disjunction of those three actions.

In TLA specifications, it is convenient to give a name to the tuple of all relevant variables. Here, we call it v.

Formula Π_Q is the internal specification of the lossy queue—the formula specifying all sequences of values that may be assumed by the queue's six variables, including the internal variables q and *last*. Its first conjunct asserts that $Init_Q$ is true in the initial state. Its second conjunct, $\Box[\mathcal{N}_Q]_v$, asserts that every step is either an $\mathcal{N}_Q$ step (a state change allowed by $\mathcal{N}_Q$) or else leaves v unchanged, meaning that it leaves all six variables unchanged.

Formula Φ_Q is the actual specification, in which the internal variables q and *last* have been hidden. A behavior satisfies Φ_Q iff there is some way to assign sequences of values to q and *last* such that Π_Q is satisfied. The free variables of Φ_Q are *ibit*, *obit*, *ival*, and *oval*, so Φ_Q specifies what sequences of values these four variables can assume. All the preceding definitions just represent one possible way of structuring the definition of Φ_Q; there are infinitely many ways to write formulas that are equivalent to Φ_Q and are therefore equivalent specifications.

TLA is an untyped logic; variables may assume any values in a fixed universal domain. Type correctness can be expressed by the formula $\Box T$, where T is the predicate asserting that all relevant variables have values of the expected "types". For the internal queue specification, the type-correctness predicate is

$$T_Q \triangleq \begin{array}{l} \wedge\ ibit, obit, last \in \{\mathsf{true}, \mathsf{false}\} \\ \wedge\ ival, oval \in \mathsf{Msg} \\ \wedge\ q \in \mathsf{Msg}^* \end{array} \qquad (1)$$

where Msg^* is the set of finite sequences of messages. Type correctness of Π_Q is asserted by the formula $\Pi_Q \Rightarrow \Box T_Q$, which is easily proved [Lamport 1994]. Type correctness of Φ_Q follows from $\Pi_Q \Rightarrow \Box T_Q$ by the usual rules for reasoning about quantifiers.

Formulas Π_Q and Φ_Q are *safety properties*, meaning that they are satisfied by an infinite behavior iff they are satisfied by every finite initial portion of the behavior. Safety properties allow behaviors in which a system performs properly for a while and then the values of all variables are frozen, never to change again. In asynchronous systems, such undesirable behaviors are ruled out by adding *fairness* properties. We could strengthen our lossy-queue specification by conjoining the *weak fairness* property $\mathrm{WF}_v(\mathit{DeQ})$ and the *strong fairness* property $\mathrm{SF}_v(\mathit{EnQ})$ to Π_Q, obtaining

$$\exists\, q, \mathit{last} : (\mathit{Init}_Q \wedge \Box[\mathcal{N}_Q]_v \wedge \mathrm{WF}_v(\mathit{DeQ}) \wedge \mathrm{SF}_v(\mathit{EnQ})) \tag{2}$$

Property $\mathrm{WF}_v(\mathit{DeQ})$ asserts that if action DeQ is enabled forever, then infinitely many DeQ steps must occur. This property implies that every message reaching the queue is eventually output. Property $\mathrm{SF}_v(\mathit{EnQ})$ asserts that if action EnQ is enabled infinitely often, then infinitely many EnQ steps must occur. It implies that if infinitely many inputs are sent, then the queue must receive infinitely many of them. The formula (2) implies the *liveness property* [Alpern and Schneider 1985] that an infinite number of inputs produces an infinite number of outputs. A formula such as (2), which is the conjunction of an initial predicate, a term of the form $\Box[\mathcal{A}]_f$, and a fairness property, is said to be in *canonical form.*

2.2 The Semantics of TLA

We begin with some definitions. We assume a universal domain of *values*, and we let $[\![F]\!]$ denote the semantic meaning of a formula F.

state. A mapping from variables to values. We let $s.x$ denote the value that state s assigns to variable x.

state function. An expression formed from variables, constants, and operators. The meaning of a state function is a mapping from states to values. For example, $x+1$ is a state function such that $[\![x+1]\!](s)$ equals $s.x+1$, for any state s.

predicate. A boolean-valued state function, such as $x > y+1$.

transition function. An expression formed from variables, primed variables, constants, and operators. The meaning of a transition function is a mapping from pairs of states to values, with unprimed variables referring to the first state of a pair and primed variables to the second. For example, $x + y' + 1$ is a transition function, and $[\![x+y'+1]\!](s,t)$ equals the value $s.x + t.y + 1$, for any pair of states s, t.

action. A boolean-valued transition function, such as $x > (y'+1)$.

step. A pair of states s, t. For an action $\mathcal{A}$, the pair is called an $\mathcal{A}$ *step* iff $[\![\mathcal{A}]\!](s,t)$ equals true. It is called a *stuttering* step iff $s = t$.

f'. The transition function obtained from the state function f by priming all the free variables of f, so $[\![f']\!](s,t) = [\![f]\!](t)$ for any states s and t.

$[\mathcal{A}]_f$. The action $\mathcal{A} \vee (f' = f)$, for any action $\mathcal{A}$ and state function f.

$\langle\mathcal{A}\rangle_f$. The action $\mathcal{A} \wedge (f' \neq f)$, for any action $\mathcal{A}$ and state function f.

Enabled $\mathcal{A}$. For any action $\mathcal{A}$, the predicate such that $[\![\mathit{Enabled}\ \mathcal{A}]\!](s)$ equals $\exists t : [\![\mathcal{A}]\!](s,t)$, for any state s.

Informally, we often identify a formula and its meaning. For example we say that a predicate P is true in state s instead of $[\![P]\!](s) = \mathsf{true}$.

An RTLA (raw TLA) formula is a boolean expression built from actions, classical operators (boolean operators and quantification over rigid variables), and the unary temporal operator $\Box$. The meaning of an RTLA formula is a boolean-valued function on *behaviors*, where a behavior is an infinite sequence of states. The meaning of the operator $\Box$ is defined by

$$[\![\Box F]\!](s_1, s_2, s_3, \ldots) \;\triangleq\; \forall n > 0 : [\![F]\!](s_n, s_{n+1}, s_{n+2}, \ldots)$$

Intuitively, $\Box F$ asserts that F is always true. The meaning of an action as an RTLA formula is defined in terms of its meaning as an action by letting $[\![\mathcal{A}]\!](s_1, s_2, s_3, \ldots)$ equal $[\![\mathcal{A}]\!](s_1, s_2)$. A predicate P is an action; P is true for a behavior iff it is true for the first state of the behavior, and $\Box P$ is true iff P is true in all states. For any action $\mathcal{A}$ and state function f, the formula $\Box[\mathcal{A}]_f$ is true for a behavior iff each step is an $\mathcal{A}$ step or else leaves f unchanged. The classical operators have their usual meanings.

A TLA formula is one that can be constructed from predicates and formulas $\Box[\mathcal{A}]_f$ using classical operators, $\Box$, and existential quantification over flexible variables. Thus, an action that is not a predicate can appear in a TLA formula F only as a subformula of an action $\mathcal{A}$ in a subformula $\Box[\mathcal{A}]_f$ or *Enabled* $\mathcal{A}$ of F. The semantics of actions, classical operators, and $\Box$ are defined as before. The approximate meaning of quantification over a flexible variable is that $\exists\!\!\exists x : F$ is true for a behavior iff there is some sequence of values that can be assigned to x that makes F true. The precise definition appears in [Lamport 1994] and is recalled in the appendix. As usual, we write $\exists\!\!\exists\, x_1, \ldots, x_n : F$ instead of $\exists\!\!\exists\, x_1 : \ldots \exists\!\!\exists\, x_n : F$.

A *property* is a set of behaviors that is *invariant under stuttering*, meaning that it contains a behavior σ iff it contains every behavior obtained from σ by adding and/or removing stuttering steps. The set of all behaviors satisfying a TLA formula is a property, which we often identify with the formula.

For any TLA formula F, action $\mathcal{A}$, and state function f:

$$\begin{array}{lcl} \Diamond F & \triangleq & \neg\Box\neg F \\ \mathrm{WF}_f(\mathcal{A}) & \triangleq & \Box\Diamond\neg(\mathit{Enabled}\ \langle\mathcal{A}\rangle_f) \;\vee\; \Box\Diamond\langle\mathcal{A}\rangle_f \\ \mathrm{SF}_f(\mathcal{A}) & \triangleq & \Diamond\Box\neg(\mathit{Enabled}\ \langle\mathcal{A}\rangle_f) \;\vee\; \Box\Diamond\langle\mathcal{A}\rangle_f \end{array}$$

These are TLA formulas, since $\Diamond\langle\mathcal{A}\rangle_f$ equals $\neg\Box[\neg\mathcal{A}]_f$.

2.3 Safety and Fairness

A *finite behavior* is a finite sequence of states. We say that a finite behavior satisfies a property F iff it can be continued to an infinite behavior in F. A property F is a *safety property* [Alpern and Schneider 1985] iff the following condition holds: F contains a behavior iff it is satisfied by every finite prefix of the behavior.[1] Intuitively, a safety property asserts that something "bad" does not happen. Predicates and formulas of the form $\Box[\mathcal{A}]_f$ are safety properties.

Safety properties are closed sets in a topology on the set of all behaviors [Abadi and Lamport 1991]. Hence, if two TLA formulas F and G are safety properties,

[1]One sometimes defines $s_1, \ldots, s_n$ to satisfy F iff the behavior $s_1, \ldots, s_n, s_n, s_n, \ldots$ is in F. Since properties are invariant under stuttering, this alternative definition leads to the same definition of a safety property.

then $F \wedge G$ is also a safety property. The *closure* $\mathcal{C}(F)$ of a property F is the smallest safety property containing F. It can be shown that $\mathcal{C}(F)$ is expressible in TLA, for any TLA formula F.

If Π is a safety property and L an arbitrary property, then the pair (Π, L) is *machine closed* iff every finite behavior satisfying Π can be extended to an infinite behavior in $\Pi \wedge L$. Two equivalent definitions are that (Π, L) is machine closed iff (i) $\mathcal{C}(\Pi \wedge L)$ equals Π, or (ii) for any safety property Ψ, if $\Pi \wedge L$ implies Ψ then Π implies Ψ. The lack of machine closure can be a source of incompleteness for proof methods. Most methods for proving safety properties use only safety properties as hypotheses, so they can prove $\Pi \wedge L \Rightarrow \Psi$ for safety properties Π and Ψ only by proving $\Pi \Rightarrow \Psi$. If Π is not machine closed, then $\Pi \wedge L \Rightarrow \Psi$ could hold even though $\Pi \Rightarrow \Psi$ does not, and these methods will be unable to prove that the system with specification Π satisfies Ψ.

Proposition 1 below shows that machine closure generalizes the concept of fairness. The *canonical form* for a TLA formula is

$$\exists\!\!\exists\, x : (\mathit{Init} \wedge \Box[\mathcal{N}]_v \wedge L) \tag{3}$$

where Init is a predicate, $\mathcal{N}$ is an action, v is a state function, L is a formula, $(\mathit{Init} \wedge \Box[\mathcal{N}]_v, L)$ is machine closed, and x is a tuple of variables called the *internal variables* of the formula. Usually, v will be the tuple of all variables appearing free in Init, $\mathcal{N}$, and L (including the variables of x). It follows from the definitions that a behavior satisfies (3) iff there is some way of choosing values for x such that (a) Init is true in the initial state, (b) every step is either an $\mathcal{N}$ step or leaves all the variables in v unchanged, and (c) the entire behavior satisfies L.

An action $\mathcal{A}$ is said to be a *subaction* of a safety property Π iff, whenever an $\mathcal{A}$ step is possible, an $\mathcal{A}$ step that satisfies Π is possible. More precisely, $\mathcal{A}$ is a subaction of Π iff for every finite behavior $s_1, \ldots, s_n$ satisfying Π with *Enabled* $\mathcal{A}$ true in state s_n, there exists a state s_{n+1} such that (s_n, s_{n+1}) is an $\mathcal{A}$ step and $s_1, \ldots, s_{n+1}$ satisfies Π. By this definition, if $\mathcal{A}$ implies $\mathcal{N}$ then $\mathcal{A}$ is a subaction of $\mathit{Init} \wedge \Box[\mathcal{N}]_v$. The exact condition is that $\mathcal{A}$ is a subaction of $\mathit{Init} \wedge \Box[\mathcal{N}]_v$ iff[2]

$$\mathit{Init} \wedge \Box[\mathcal{N}]_v \;\Rightarrow\; \Box((\mathit{Enabled}\ \mathcal{A}) \Rightarrow \mathit{Enabled}\ (\mathcal{A} \wedge [\mathcal{N}]_v))$$

Two actions are *disjoint* iff their conjunction is identically false. A weaker notion is disjointness of two actions for a property: $\mathcal{A}$ and $\mathcal{B}$ are *disjoint for* Π iff no behavior satisfying Π contains an $\mathcal{A} \wedge \mathcal{B}$ step. By this definition, $\mathcal{A}$ and $\mathcal{B}$ are disjoint for $\mathit{Init} \wedge \Box[\mathcal{N}]_v$ iff

$$\mathit{Init} \wedge \Box[\mathcal{N}]_v \;\Rightarrow\; \Box\neg\mathit{Enabled}\ (\mathcal{A} \wedge \mathcal{B} \wedge [\mathcal{N}]_v)$$

The following result shows that the conjunction of WF and SF formulas is a fairness property. It is a special case of Proposition 4 of Section 4.

PROPOSITION 1. *If* Π *is a safety property and* L *is the conjunction of a finite or countably infinite number of formulas of the form* $\mathrm{WF}_w(\mathcal{A})$ *and/or* $\mathrm{SF}_w(\mathcal{A})$ *such that each* $\langle\mathcal{A}\rangle_w$ *is a subaction of* Π, *then* (Π, L) *is machine closed.*

[2]We let $\Rightarrow$ have lower precedence than the other boolean operators.

In practice, each w will usually be a tuple of variables changed by the corresponding action $\mathcal{A}$, so $\langle \mathcal{A} \rangle_w$ will equal $\mathcal{A}$.[3] In the informal exposition, we often omit the subscript and talk about $\mathcal{A}$ when we really mean $\langle \mathcal{A} \rangle_w$.

Machine closure for more general classes of properties can be proved with the following two propositions, which are proved in the appendix. To apply the first, one must prove that $\exists x : \Pi$ is a safety property. By Proposition 2 of [Abadi and Lamport 1991, page 265], it suffices to prove that Π has finite internal nondeterminism (fin), with x as its internal state component. Here, fin means roughly that there are only a finite number of sequences of values for x that can make a finite behavior satisfy Π.

PROPOSITION 2. *If* (Π, L) *is machine closed,* x *is a tuple of variables that do not occur free in* L*, and* $\exists x : \Pi$ *is a safety property, then* $((\exists x : \Pi), L)$ *is machine closed.*

PROPOSITION 3. *If* (Π, L_1) *is machine closed and* $\Pi \wedge L_1$ *implies* L_2*, then* (Π, L_2) *is machine closed.*

2.4 History-Determined Variables

A *history-determined* variable is one whose current value can be inferred from the current and past values of other variables. For the precise definition, let

$$Hist(h, f, g, v) \ \triangleq\ (h = f) \ \wedge\ \Box[(h' = g) \wedge (v' \neq v)]_{(h,v)} \tag{4}$$

where f and v are state functions and g is a transition function. A variable h is a history-determined variable for a formula Π iff Π implies $Hist(h, f, g, v)$, for some f, g, and v such that h occurs free in neither f nor v, and h' does not occur free in g.

If f and v do not depend on h, and g does not depend on h', then $\exists h : Hist(h, f, g, v)$ is true for all behaviors. Therefore, if h does not occur free in formula Φ, then $\exists h : (\Phi \wedge Hist(h, f, g, v))$ is equivalent to Φ. In other words, conjoining $Hist(h, f, g, v)$ to Φ does not change the behavior of its variables, so it makes h a "dummy variable" for Φ—in fact, it is a special kind of history variable [Abadi and Lamport 1991, page 270].

As an example, we add to the lossy queue's specification Φ_Q a history variable *hin* that records the sequence of values transmitted on the input wire. Let

$$\begin{aligned} H_{in} \ \triangleq\ & \wedge\ hin = \langle\!\langle\,\rangle\!\rangle \\ & \wedge\ \Box[\ \wedge\ hin' = hin \circ \langle\!\langle ival' \rangle\!\rangle \\ & \qquad\ \wedge\ (ival, ibit)' \neq (ival, ibit)\ ]_{(hin,ival,ibit)} \end{aligned} \tag{5}$$

Then H_{in} equals $Hist(hin, \langle\!\langle\,\rangle\!\rangle, hin \circ \langle\!\langle ival' \rangle\!\rangle, (ival, ibit))$; therefore, *hin* is a history-determined variable for $\Phi_Q \wedge H_{in}$, and $\exists\, hin : (\Phi_Q \wedge H_{in})$ equals Φ_Q.

If h is a history-determined variable for a property Π, then Π is fin, with h as its internal state component. Hence, if Π is a safety property, then $\exists h : \Pi$ is also a safety property.

[3]More precisely, $T \wedge \mathcal{A}$ will imply $w' \neq w$, where T is the type-correctness invariant.

3. REAL-TIME CLOSED SYSTEMS

We now use TLA to specify and reason about timing properties of closed systems. We focus on worst-case upper and lower bounds on real-time delays. However, our approach should be applicable to other real-time properties as well. We believe that its only inherent limitation is that it cannot handle probabilistic real-time properties, such as average delay requirements. This limitation arises because, like most specification formalisms, TLA cannot express probabilistic properties.

Section 3.1 explains how time and timing properties can be represented with TLA formulas, and Section 3.2 describes how to reason about these formulas. The problem of Zeno specifications is addressed in Section 3.3. Our method of specifying and reasoning about timing properties is illustrated in Section 3.4 with the example of a real-time mutual exclusion protocol.

3.1 Time and Timers

In real-time TLA specifications, real time is represented by the variable *now*. Although it has a special interpretation, *now* is just an ordinary variable of the logic. The value of *now* is always a real number, and it never decreases—conditions expressed by the TLA formula

$$RT \;\triangleq\; (now \in \mathbf{R}) \wedge \Box[now' \in (now, \infty)]_{now}$$

where $\mathbf{R}$ is the set of real numbers and (r, ∞) is $\{t \in \mathbf{R} : t > r\}$.

It is convenient to make time-advancing steps distinct from ordinary program steps. This is done by strengthening the formula RT to

$$RT_v \;\triangleq\; (now \in \mathbf{R}) \wedge \Box[(now' \in (now, \infty)) \wedge (v' = v)]_{now}$$

This property differs from RT only in asserting that v does not change when *now* advances. Simple logical manipulation shows that RT_v is equivalent to $RT \wedge \Box[now' = now]_v$, and

$$Init \wedge \Box[\mathcal{N}]_v \wedge RT_v \;=\; Init \wedge \Box[\mathcal{N} \wedge (now' = now)]_v \wedge RT$$

We express real-time constraints by placing timing bounds on actions. Such bounds are the bread-and-butter of many real-time formalisms, such as real-time process algebras [de Bakker et al. 1992]. Timing bounds on actions are imposed by using *timers* to restrict the increase of *now*. A *timer for* Π is a state function t such that Π implies $\Box(t \in \mathbf{R} \cup \{\pm\infty\})$. Timer t is used as an upper-bound timer by conjoining the formula

$$MaxTime(t) \;\triangleq\; (now \leq t) \wedge \Box[now' \leq t']_{now}$$

to a specification. This formula asserts that *now* is never advanced past t. Timer t is used as a lower-bound timer for an action $\mathcal{A}$ by conjoining the formula

$$MinTime(t, \mathcal{A}, v) \;\triangleq\; \Box[\mathcal{A} \Rightarrow (t \leq now)]_v$$

to a specification. This formula asserts that an $\langle \mathcal{A} \rangle_v$ step cannot occur when *now* is less than t.[4]

[4]Unlike the usual timers in computer systems that represent an increment of time, our timers

A common type of timing constraint asserts that an $\mathcal{A}$ step must occur within δ seconds of when the action $\mathcal{A}$ becomes enabled, for some constant δ. After an $\mathcal{A}$ step, the next $\mathcal{A}$ step must occur within δ seconds of when action $\mathcal{A}$ is re-enabled. There are at least two reasonable interpretations of this requirement.

The first interpretation is that the $\mathcal{A}$ step must occur if $\mathcal{A}$ has been continuously enabled for δ seconds. This is expressed by $MaxTime(t)$ when t is a state function satisfying

$$\begin{array}{lll} VTimer(t, A, \delta, v) & \triangleq & \land\ t = \textbf{if } Enabled\ \langle\mathcal{A}\rangle_v \textbf{ then } now + \delta \\ & & \qquad\qquad\qquad\qquad \textbf{else } \infty \\ & & \land\ \Box[\land\ t' = \textbf{if } (Enabled\ \langle\mathcal{A}\rangle_v)' \\ & & \qquad\qquad \textbf{then if } \langle\mathcal{A}\rangle_v \lor \neg Enabled\ \langle\mathcal{A}\rangle_v \\ & & \qquad\qquad\qquad \textbf{then } now + \delta \\ & & \qquad\qquad\qquad \textbf{else } t \\ & & \qquad\qquad \textbf{else } \infty \\ & & \quad \land\ v' \neq v\ ]_{(t,v)} \end{array}$$

Such a t is called a *volatile δ-timer.*

Another interpretation of the timing requirement is that an $\mathcal{A}$ step must occur if $\mathcal{A}$ has been enabled for a total of δ seconds, though not necessarily continuously enabled. This is expressed by $MaxTime(t)$ when t satisfies

$$\begin{array}{lll} PTimer(t, A, \delta, v) & \triangleq & \land\ t = now + \delta \\ & & \land\ \Box[\land\ t' = \textbf{if } Enabled\ \langle\mathcal{A}\rangle_v \\ & & \qquad\qquad \textbf{then if } \langle\mathcal{A}\rangle_v \textbf{ then } now + \delta \\ & & \qquad\qquad\qquad\qquad\quad \textbf{else } t \\ & & \qquad\qquad \textbf{else } t + (now' - now) \\ & & \quad \land\ (v, now)' \neq (v, now)\ ]_{(t,v,now)} \end{array}$$

Such a t is called a *persistent δ-timer.* We can use δ-timers as lower-bound timers as well as upper-bound timers.

Observe that $VTimer(t, \mathcal{A}, \delta, v)$ has the form $Hist(t, f, g, v)$ and $PTimer(t, \mathcal{A}, \delta, v)$ has the form $Hist(t, f, g, (v, now))$, where $Hist$ is defined by (4). Thus, if formula Π implies that a variable t satisfies either of these formulas, then t is a history-determined variable for Π.

As an example of the use of timers, we make the lossy queue of Section 2.1 nonlossy by adding the following timing constraints.

—Values must be put on a wire at most once every δ_{snd} seconds. There are two conditions—one on the input wire and one on the output wire. They are expressed by using δ_{snd}-timers t_{Inp} and t_{DeQ}, for the actions Inp and DeQ, as lower-bound timers.

—A value must be added to the queue at most Δ_{rcv} seconds after it appears on the input wire. This is expressed by using a Δ_{rcv}-timer T_{EnQ}, for the enqueue action, as an upper-bound timer.

represent an absolute time. To allow the type of strict time bound that would be expressed by replacing $\leq$ with $<$ in the definition of $MaxTime$ or $MinTime$, we could introduce, as additional possible values for timers, the set of all "infinitesimally shifted" real numbers r^-, where $t \leq r^-$ iff $t < r$, for any reals t and r.

—A value must be sent on the output wire within Δ_{snd} seconds of when it reaches the head of the queue. This is expressed by using a Δ_{snd}-timer T_{DeQ}, for the dequeue action, as an upper-bound timer.

The timed queue will be nonlossy if $\Delta_{rcv} < \delta_{snd}$. In this case, we expect the *Inp*, *EnQ*, and *DeQ* actions to remain enabled until they are "executed", so it doesn't matter whether we use volatile or persistent timers. We use volatile timers because they are a little easier to reason about.

The timed version Π_Q^t of the queue's internal specification Π_Q is obtained by conjoining the timing constraints to Π_Q:

$$\begin{aligned}\Pi_Q^t \triangleq\; & \wedge\, \Pi_Q \;\wedge\; RT_v \\ & \wedge\, VTimer(t_{Inp}, Inp, \delta_{snd}, v) \;\wedge\; MinTime(t_{Inp}, Inp, v) \\ & \wedge\, VTimer(t_{DeQ}, DeQ, \delta_{snd}, v) \;\wedge\; MinTime(t_{DeQ}, DeQ, v) \\ & \wedge\, VTimer(T_{EnQ}, EnQ, \Delta_{rcv}, v) \;\wedge\; MaxTime(T_{EnQ}) \\ & \wedge\, VTimer(T_{DeQ}, DeQ, \Delta_{snd}, v) \;\wedge\; MaxTime(T_{DeQ})\end{aligned} \tag{6}$$

The external specification Φ_Q^t of the timed queue is obtained by existentially quantifying first the timers and then the variables q and *last*.

Formula Π_Q^t of (6) is not in the canonical form for a TLA formula. A straightforward calculation, using the type-correctness invariant (1) and the equivalence of $(\Box F) \wedge (\Box G)$ and $\Box(F \wedge G)$, converts the expression (6) for Π_Q^t to the canonical form given in Figure 3.[5] Observe how each subaction $\mathcal{A}$ of the original formula has a corresponding timed version $\mathcal{A}^t$. Action $\mathcal{A}^t$ is obtained by conjoining $\mathcal{A}$ with the appropriate relations between the old and new values of the timers. If $\mathcal{A}$ has a lower-bound timer, then $\mathcal{A}^t$ also has a conjunct asserting that it is not enabled when *now* is less than this timer. (The lower-bound timer t_{Inp} for *Inp* does not affect the enabling of other subactions because *Inp* is disjoint from all other subactions; a similar remark applies to the lower-bound timer t_{DeQ}.) There is also a new action, *QTick*, that advances *now*.

Formula Π_Q^t is the TLA specification of a program that satisfies each maximum-delay constraint by preventing *now* from advancing before the constraint has been satisfied. Thus, the program "implements" timing constraints by stopping time, an apparent absurdity. However, the absurdity results from thinking of a TLA formula, or the abstract program that it represents, as a *prescription of how* something is accomplished. A TLA formula is really a *description of what* is supposed to happen. Formula Π_Q^t says only that an action occurs before *now* reaches a certain value. It is just our familiarity with ordinary programs that makes us jump to the conclusion that *now* is being changed by the system.

3.2 Reasoning About Time

Formula Π_Q^t is a safety property; it is satisfied by a behavior in which no variables change values. In particular, it allows behaviors in which time stops. We can rule

[5]Further simplification of this formula is possible, but it requires an invariant. In particular, the fourth conjunct of DeQ^t can be replaced by $T'_{EnQ} = T_{EnQ}$.

$$
\begin{array}{lll}
Init^t_Q & \triangleq & \land\ Init_Q \\
 & & \land\ now \in \mathbf{R} \\
 & & \land\ t_{Inp} = now + \delta_{snd} \\
 & & \land\ t_{DeQ} = T_{EnQ} = T_{DeQ} = \infty \\
Inp^t & \triangleq & \land\ Inp \\
 & & \land\ t_{Inp} \leq now \\
 & & \land\ t'_{Inp} = now' + \delta_{snd} \\
 & & \land\ T'_{EnQ} = \textbf{if } last' \neq ibit' \textbf{ then } now' + \Delta_{rcv} \textbf{ else } \infty \\
 & & \land\ (t_{DeQ}, T_{DeQ})' = \textbf{if } q = \langle\langle\,\rangle\rangle \textbf{ then } (\infty, \infty) \textbf{ else } (t_{DeQ}, T_{DeQ}) \\
 & & \land\ now' = now \\
EnQ^t & \triangleq & \land\ EnQ \\
 & & \land\ T'_{EnQ} = \infty \\
 & & \land\ (t_{DeQ}, T_{DeQ})' = \textbf{if } q = \langle\langle\,\rangle\rangle \textbf{ then } (now + \delta_{snd},\ now + \Delta_{snd}) \\
 & & \qquad\qquad\qquad\qquad\qquad \textbf{else } (t_{DeQ}, T_{DeQ}) \\
 & & \land\ (t_{Inp}, now)' = (t_{Inp}, now) \\
DeQ^t & \triangleq & \land\ DeQ \\
 & & \land\ t_{DeQ} \leq now \\
 & & \land\ (t_{DeQ}, T_{DeQ})' = \textbf{if } q' = \langle\langle\,\rangle\rangle \textbf{ then } (\infty, \infty) \\
 & & \qquad\qquad\qquad\qquad\qquad \textbf{else } (now + \delta_{snd},\ now + \Delta_{snd}) \\
 & & \land\ T'_{EnQ} = \textbf{if } last' = ibit' \textbf{ then } \infty \textbf{ else } T_{EnQ} \\
 & & \land\ (t_{Inp}, now)' = (t_{Inp}, now) \\
QTick & \triangleq & \land\ now' \in (now, \min(T_{DeQ}, T_{EnQ})] \\
 & & \land\ (v, t_{Inp}, t_{DeQ}, T_{DeQ}, T_{EnQ})' = (v, t_{Inp}, t_{DeQ}, T_{DeQ}, T_{EnQ}) \\
vt & \triangleq & (v, now, t_{Inp}, t_{DeQ}, T_{DeQ}, T_{EnQ}) \\
\Pi^t_Q & \triangleq & \land\ Init^t_Q \\
 & & \land\ \Box[Inp^t \lor EnQ^t \lor DeQ^t \lor QTick]_{vt}
\end{array}
$$

Fig. 3. The canonical form for Π^t_Q, where $(r, s]$ denotes the set of reals u such that $r < u \leq s$.

out such behaviors by conjoining to Π_Q^t the liveness property

$$NZ \stackrel{\Delta}{=} \forall t \in \mathbf{R} : \Diamond(now > t)$$

which asserts that *now* gets arbitrarily large. However, when reasoning only about real-time (safety) properties, this should not be necessary. For example, suppose we want to show that our timed queue satisfies a real-time property expressed by formula Ψ^t, which is also a safety property. If Π_Q^t implies Ψ^t, then $\Pi_Q^t \wedge NZ$ implies $\Psi^t \wedge NZ$. Conversely, we don't expect conjoining a liveness property to add safety properties; if $\Pi_Q^t \wedge NZ$ implies Ψ^t, then Π_Q^t by itself should imply Ψ^t. Hence, there should be no need to introduce the liveness property NZ. Section 3.3 below explains precisely when we can ignore property NZ.

A safety property we might want to prove for the timed queue is that it does not lose any inputs. To express this property, let *hin* be the history variable, determined by H_{in} of (5), that records the sequence of input values; and let *hout* and H_{out} be the analogous history variable and property for the outputs. The assertion that the timed queue loses no inputs is expressed by

$$\Pi_Q^t \wedge H_{in} \wedge H_{out} \Rightarrow \Box(hout \preceq hinp)$$

where $\alpha \preceq \beta$ iff α is an initial prefix of β. This is a standard invariance property. The usual method for proving such properties leads to the following invariant

$$\begin{array}{l}
\wedge\ T_Q\ \wedge\ (t_{Inp}, now \in \mathbf{R})\ \wedge\ (T_{EnQ}, t_{DeQ}, T_{DeQ} \in \mathbf{R} \cup \{\infty\}) \\
\wedge\ now \leq \min(T_{EnQ}, T_{DeQ}) \\
\wedge\ (last = ibit)\ \Rightarrow\ (T_{EnQ} = \infty) \wedge (hinp = hout \circ q) \\
\wedge\ (last \neq ibit)\ \Rightarrow\ (T_{EnQ} < t_{Inp}) \wedge (hinp = hout \circ q \circ \langle\langle ival \rangle\rangle) \\
\wedge\ (q = \langle\langle\,\rangle\rangle)\ \equiv\ (T_{DeQ} = \infty)
\end{array}$$

and to the necessary assumption $\Delta_{rcv} < \delta_{snd}$. (Recall that T_Q is the type-correctness predicate (1) for Π_Q.) Property NZ is not needed to prove this invariant.

Property NZ is needed to prove that real-time properties imply liveness properties. The desired liveness property for the timed queue is that the sequence of input messages up to any point eventually appears as the sequence of output messages. It is expressed by

$$\Pi_Q^t \wedge NZ \Rightarrow \forall \sigma : \Box((hinp = \sigma) \Rightarrow \Diamond(hout = \sigma))$$

This formula is proved by first showing

$$\Pi_Q^t \wedge NZ \Rightarrow \mathrm{WF}_v(EnQ) \wedge \mathrm{WF}_v(DeQ) \tag{7}$$

and then using a standard liveness argument to prove

$$\Pi_Q^t \wedge \mathrm{WF}_v(EnQ) \wedge \mathrm{WF}_v(DeQ) \Rightarrow \forall \sigma : \Box((hinp = \sigma) \Rightarrow \Diamond(hout = \sigma))$$

The proof that $\Pi_Q^t \wedge NZ$ implies $\mathrm{WF}_v(EnQ)$ is by contradiction. Assume EnQ is forever enabled but never occurs. An invariance argument then shows that Π_Q^t implies that T_{EnQ} forever equals its current value, preventing *now* from advancing past that value; and this contradicts NZ. The proof that $\Pi_Q^t \wedge NZ$ implies $\mathrm{WF}_v(DeQ)$ is similar.

3.3 The NonZeno Condition

The timed queue specification Π_Q^t asserts that a *DeQ* action must occur between δ_{snd} and Δ_{snd} seconds of when it becomes enabled. What if $\Delta_{snd} < \delta_{snd}$? If an input occurs, it eventually is put in the queue, enabling *DeQ*. At that point, the value of *now* can never become more than Δ_{snd} greater than its current value, so the program eventually reaches a "time-blocked state". In a time-blocked state, only the *QTick* action can be enabled, and it cannot advance *now* past some fixed time. In other words, eventually a state is reached in which every variable other than *now* remains the same, and *now* either remains the same or keeps advancing closer and closer to some upper bound.

We can attempt to correct such pathological specifications by requiring that *now* increase without bound. This is easily done by conjoining the liveness property *NZ* to the safety property Π_Q^t, but that doesn't accomplish anything. Since $\Pi_Q^t \land NZ$ rules out behaviors in which *now* is bounded, it allows only behaviors in which there is no input, if $\Delta_{snd} < \delta_{snd}$. Such a specification is no better than the original specification Π_Q^t. The fact that the safety property allows the possibility of reaching a time-blocked state indicates an error in the specification. One does not add timing constraints on output actions with the intention of forbidding input.

We call a safety property *Zeno* if it allows the system to reach a state from which *now* must remain bounded. More precisely, a safety property Π is *nonZeno* iff every finite behavior satisfying Π can be completed to an infinite behavior satisfying Π in which *now* increases without bound. In other words, Π is nonZeno iff the pair (Π, NZ) is machine closed.[6]

Zenoness can be a source of incompleteness for proof methods. Only nonZeno behaviors are physically meaningful, so a real-time system with specification Π satisfies a property Ψ if $\Pi \land NZ$ implies Ψ. As observed in Section 2.3, a proof method may be incapable of showing that $\Pi \land NZ$ implies Ψ if (Π, NZ) is not machine closed—that is, if Π is Zeno.

The following result can be used to ensure that a real-time specification written in terms of volatile δ-timers is nonZeno. The main hypotheses are: (1) the maximum-delay timers are on subactions of the untimed specification, and (3) a maximum delay for an action $\mathcal{A}_j$ is not smaller than a minimum delay for an action $\mathcal{A}_i$ if $\mathcal{A}_i$ and $\mathcal{A}_j$ can be simultaneously enabled.

THEOREM 1. *Let v be the tuple of variables free in Init or $\mathcal{N}$. The property*

$$\begin{aligned}&\land\ Init \land \Box[\mathcal{N}]_v \land RT_v\\ &\land\ \forall i \in I : VTimer(t_i, \mathcal{A}_i, \delta_i, v) \ \land\ MinTime(t_i, \mathcal{A}_i, v)\\ &\land\ \forall j \in J : VTimer(T_j, \mathcal{A}_j, \Delta_j, v) \ \land\ MaxTime(T_j)\end{aligned}$$

is nonZeno if now does not appear in v, I and J are finite sets, and for all $i \in I$ and $j \in J$:

(1) $\langle \mathcal{A}_j \rangle_v$ is a subaction of $Init \land \Box[\mathcal{N}]_v$ whose free variables appear in v,

(2) δ_i and Δ_j are positive reals,

[6]An arbitrary property Π is nonZeno iff $(\mathcal{C}(\Pi), \Pi \land NZ)$ is machine closed. We restrict our attention to real-time constraints for safety specifications.

(3) $\delta_i \leq \Delta_j$, or $\langle \mathcal{A}_i \rangle_v$ and $\langle \mathcal{A}_j \rangle_v$ are disjoint for $Init \wedge \Box[\mathcal{N}]_v$ and $i \neq j$,

(4) the t_i and T_j are distinct variables different from now and from the variables in v.

We can apply the theorem to prove that the specification Π^t_Q is nonZeno if $\delta_{snd} \leq \Delta_{snd}$. The hypotheses of the theorem are checked as follows.

(1) Actions $\langle DeQ \rangle_v$ and $\langle EnQ \rangle_v$ imply $\mathcal{N}_Q$, so they are subactions of Π_Q.

(2) Trivial.

(3) The conjunction of any two of the actions $\langle Inp \rangle_v$, $\langle DeQ \rangle_v$, and $\langle EnQ \rangle_v$ equals false, so the actions are pairwise disjoint for Π_Q.[7] The only remaining cases to consider are those in which $i = j$. Only $\langle DeQ \rangle_v$ has both a lower-bound and an upper-bound timer, and $\delta_{snd} \leq \Delta_{snd}$ holds by hypothesis.

(4) Trivial.

The theorem is valid for persistent as well as volatile timers. Any combination of *VTimer* and *PTimer* formulas may occur, except that a single $\mathcal{A}_k$ cannot have a persistent lower-bound timer t_k and a volatile upper-bound timer T_k. In fact, the theorem is valid for any kind of lower-bound timer t_k, not just a persistent or volatile one, provided t_k is never greater than the corresponding upper-bound timer T_k.

All of these results are corollaries of Theorem 2 below, which in turn is a consequence of Theorem 4 of Section 4. To allow arbitrary lower-bound timers, Theorem 2 uses different notation from Theorem 1. The Π of Theorem 2 corresponds to the conjunction of $Init \wedge \Box[\mathcal{N}]_v$ with all the *VTimer* formulas of Theorem 1.

THEOREM 2. *Let*

—Π be a safety property of the form $Init \wedge \Box[\mathcal{N}]_w$,

—t_i and T_j be timers for Π and let $\mathcal{A}_k$ be an action, for all $i \in I$, $j \in J$, and $k \in I \cup J$, where I and J are sets, with J finite,

—$\Pi^t \triangleq \Pi \wedge RT_v \wedge$
$$\forall i \in I : MinTime(t_i, \mathcal{A}_i, v) \wedge \forall j \in J : MaxTime(T_j)$$

If (1) $\Pi^t \Rightarrow \Box(t_i \leq T_j)$, or $\langle \mathcal{A}_i \rangle_v$ and $\langle \mathcal{A}_j \rangle_v$ are disjoint for Π and $i \neq j$, for all $i \in I$ and $j \in J$,

(2) (a) now does not occur free in v,

(b) $(now' = r) \wedge (v' = v)$ is a subaction of Π, for all $r \in \mathbf{R}$,

(3) for all $j \in J$:

(a) $\langle \mathcal{A}_j \rangle_v \wedge (now' = now)$ is a subaction of Π,

(b) $\Pi \Rightarrow VTimer(T_j, \mathcal{A}_j, \Delta_j, v)$ or $\Pi \Rightarrow PTimer(T_j, \mathcal{A}_j, \Delta_j, v)$, for some $\Delta_j \in (0, \infty)$,

(c) $\Pi^t \Rightarrow \Box(Enabled\ \langle \mathcal{A}_j \rangle_v =$
$$Enabled\ (\langle \mathcal{A}_j \rangle_v \wedge (now' = now)))$$

(d) $(v' = v) \Rightarrow (Enabled\ \langle \mathcal{A}_j \rangle_v = (Enabled\ \langle \mathcal{A}_j \rangle_v)')$

then (Π^t, NZ) is machine closed

[7] Actually, the type-correctness predicate T_Q is needed to prove that $\langle Inp \rangle_v \wedge \langle DeQ \rangle_v$ equals false.

Most nonaxiomatic approaches, including both real-time process algebras and more traditional programming languages with timing constraints, essentially use δ-timers for actions. Theorem 2 implies that they automatically yield nonZeno specifications.

Theorem 2 can be further generalized in two ways. First, J can be infinite—if Π^t implies that only a finite number of actions $\mathcal{A}_j$ with $j \in J$ are enabled before time r, for any $r \in \mathbf{R}$. For example, by letting $\mathcal{A}_j$ be the action that sends message number j, we can apply the theorem to a program that sends messages number 1 through n at time n, for every integer n. This program is nonZeno even though the number of actions per second that it performs is unbounded. Second, we can extend the theorem to the more general class of timers obtained by letting the Δ_j be arbitrary real-valued state functions, rather than just constants—if all the Δ_j are bounded from below by a positive constant Δ.

Theorem 2 can be proved using Propositions 1 and 3 and ordinary TLA reasoning. By these propositions, it suffices to display a formula L that is the conjunction of fairness conditions on subactions of Π^t such that $\Pi^t \wedge L$ implies NZ. A suitable L is $\mathrm{WF}_{(now,v)}(\mathcal{C})$, where $\mathcal{C}$ is an action that either (a) advances *now* by $\min_{j\in J} \Delta_j$ if allowed by the upper-bound timers T_j, or else as far as they do allow, or (b) executes an $\langle \mathcal{A}_j \rangle_v$ action for which $now = T_j$. The proof in the appendix of Theorem 4, which implies Theorem 2, generalizes this approach.

Theorem 2 does not cover all situations of interest. For example, one can require of our timed queue that the first value appear on the output line within ϵ seconds of when it is placed on the input line. This effectively places an upper bound on the sum of the times needed for performing the *EnQ* and *DeQ* actions; it cannot be expressed with δ-timers on individual actions. For these general timing constraints, nonZenoness must be proved for the individual specification. The proof uses the method described above for proving Theorem 2: we add to the timed program Π^t a liveness property L that is the conjunction of any fairness properties we like, including fairness of the action that advances *now*, and prove that $\Pi^t \wedge L$ implies NZ. NonZenoness then follows from Propositions 1 and 3.

There is another possible approach to proving nonZenoness. One can make granularity assumptions—lower bounds both on the amount by which *now* is incremented and on the minimum delay for each action. Under these assumptions, nonZenoness is equivalent to the absence of deadlock, which can be proved by existing methods. Granularity assumptions are probably adequate—after all, what harm can come from pretending that nothing happens in less than 10^{-100} nanoseconds? However, they can be unnatural and cumbersome. For example, distributed algorithms often assume that only message delays are significant, so the time required for local actions is ignored. The specification of such an algorithm should place no lower bound on the time required for a local action, but that would violate any granularity assumptions. We believe that any proof of deadlock freedom based on granularity can be translated into a proof of nonZenoness using the method outlined above.

So far, we have been discussing nonZenoness of the internal specification, where both the timers and the system's internal variables are visible. Timers are defined by adding history-determined variables, so existentially quantifying over them preserves nonZenoness by Proposition 2. We expect most specifications to be fin [Abadi and Lamport 1991, page 263], so nonZenoness will also be preserved by existentially quantifying over the system's internal variables. This is the case for the timed queue.

3.4 An Example: Fischer's Protocol

As another example of real-time closed systems, we treat a simplified version of a real-time mutual exclusion protocol proposed by Fischer [1985] and described in [Lamport 1987, page 2]. The example was suggested by Schneider et al. [1992]. The protocol consists of each process i executing the following code, where angle brackets denote instantaneous atomic actions:

```
     a:  await ⟨x = 0⟩;
     b:  ⟨x := i⟩;
     c:  await ⟨x = i⟩;
    cs:  critical section
```

There is a maximum delay Δ_b between the execution of the test in statement a and the assignment in statement b, and a minimum delay δ_c between the assignment in statement b and the test in statement c. The problem is to prove that, with suitable conditions on Δ_b and δ_c, this protocol guarantees mutual exclusion (at most one process can enter its critical section).

As written, Fischer's protocol permits only one process to enter its critical section one time. The protocol can be converted to an actual mutual exclusion algorithm. The correctness proof of the protocol is easily extended to a proof of such an algorithm.

The TLA specification of the protocol is given in Figure 4. The formula Π_F describing the untimed version is standard TLA. We assume a finite set Proc of processes. Variable x represents the program variable x, and variable pc represents the control state. The value of pc will be an array indexed by Proc, where $pc[i]$ equals one of the strings "a", "b", "c", "cs" when control in process i is at the corresponding statement. The initial predicate $Init_F$ asserts that $pc[i]$ equals "a" for each process i, so the processes start with control at statement a. No assumption on the initial value of x is needed to prove mutual exclusion.

Next come the definitions of the three actions corresponding to program statements a, b, and c. They are defined using the formula Go, where $Go(i, u, v)$ asserts that control in process i changes from u to v, while control remains unchanged in the other processes. Action $\mathcal{A}_i$ represents the execution of statement a by process i; actions $\mathcal{B}_i$ and $\mathcal{C}_i$ have the analogous interpretation. In this simple protocol, a process stops when it gets to its critical section, so there are no other actions. The program's next-state action $\mathcal{N}_F$ is the disjunction of all these actions. Formula Π_F asserts that all processes start at statement a, and every step consists of executing the next statement of some process.

Action $\mathcal{B}_i$ is enabled by the execution of action $\mathcal{A}_i$. Therefore, the maximum delay of Δ_b between the execution of statements a and b can be expressed by an upper-bound constraint on a volatile Δ_b-timer for action $\mathcal{B}_i$. The variable T_b is an array of such timers, where $T_b[i]$ is the timer for action $\mathcal{B}_i$.

The constant δ_c is the minimum delay between when control reaches statement c and when that statement is executed. Therefore, we need an array t_c of lower-

$$
\begin{array}{lcl}
Init_F & \triangleq & \forall i \in \mathsf{Proc} : pc[i] = \text{“a”} \\
Go(i,u,v) & \triangleq & \land\ pc[i] = u \\
 & & \land\ pc'[i] = v \\
 & & \land\ \forall j \in \mathsf{Proc} : (j \neq i) \Rightarrow (pc'[j] = pc[j]) \\
\mathcal{A}_i & \triangleq & Go(i, \text{“a”}, \text{“b”}) \land (x = x' = 0) \\
\mathcal{B}_i & \triangleq & Go(i, \text{“b”}, \text{“c”}) \land (x' = i) \\
\mathcal{C}_i & \triangleq & Go(i, \text{“c”}, \text{“cs”}) \land (x = x' = i) \\
\mathcal{N}_F & \triangleq & \exists\, i \in \mathsf{Proc} : (\mathcal{A}_i \lor \mathcal{B}_i \lor \mathcal{C}_i) \\
\Pi_F & \triangleq & Init_F \land \Box[\mathcal{N}_F]_{(x,pc)} \\
\Pi^t_F & \triangleq & \land\ \Pi_F \land RT_{(x,pc)} \\
 & & \land\ \forall i \in \mathsf{Proc} : \land\ VTimer(T_b[i], \mathcal{B}_i, \Delta_b, (x,pc)) \\
 & & \qquad\qquad\qquad\quad \land\ MaxTime(T_b[i]) \\
 & & \land\ \forall i \in \mathsf{Proc} : \land\ VTimer(t_c[i], Go(i, \text{“c”}, \text{“cs”}), \delta_c, (x,pc)) \\
 & & \qquad\qquad\qquad\quad \land\ MinTime(t_c[i], \mathcal{C}_i, (x,pc)) \\
\Phi^t_F & \triangleq & \exists\!\!\exists\, T_b, t_c : \Pi^t_F
\end{array}
$$

Fig. 4. The TLA specification of Fischer's real-time mutual exclusion protocol.

bound timers for the actions $\mathcal{C}_i$. The delay is measured from the time control reaches statement c, so we want $t_c[i]$ to be a δ_c-timer on an action that becomes enabled when process i reaches statement c and is not executed until $\mathcal{C}_i$ is. (Since we are placing only a lower-bound timer on it, the action need not be a subaction of Π_F.) A suitable choice for this action is $Go(i, \text{“c”}, \text{“cs”})$.

Adding these timers and timing constraints to the untimed formula Π_F yields formula Π^t_F of Figure 4, the specification of the real-time protocol with the timers visible. The final specification, Φ^t_F, is obtained by quantifying over the timer variables T_b and t_c. Since $\langle \mathcal{B}_j \rangle_{(x,pc)} \land (now' = now)$ is a subaction of Π_F and $\langle Go(i, \text{“c”}, \text{“cs”}) \rangle_{(x,pc)}$ is disjoint from $\langle \mathcal{B}_j \rangle_{(x,pc)}$, for all i and j in Proc, Theorem 2 implies that Π^t_F is nonZeno if Δ_b is positive. Proposition 2 can then be applied to prove that Φ^t_F is nonZeno.

Mutual exclusion asserts that two processes cannot be in their critical sections at the same time. It is expressed by the predicate

$$Mutex \triangleq \forall i, j \in \mathsf{Proc} : (pc[i] = pc[j] = \text{“cs”}) \Rightarrow (i = j)$$

The property to be proved is

$$Assump \land \Phi^t_F \Rightarrow \Box Mutex \tag{8}$$

where $Assump$ expresses the assumptions about the constants Proc, Δ_b, and δ_c needed for correctness. Since the timer variables do not occur in $Mutex$ or $Assump$, (8) is equivalent to

$$Assump \land \Pi^t_F \Rightarrow \Box Mutex$$

The standard method for proving this kind of invariance property leads to the invariant

$$\begin{array}{l}
\land\ now \in \mathbf{R} \\
\land\ \forall i \in \mathsf{Proc} : \\
\quad \land\ T_b[i], t_c[i] \in \mathbf{R} \cup \{\infty\} \\
\quad \land\ pc[i] \in \{\text{“a”}, \text{“b”}, \text{“c”}, \text{“cs”}\} \\
\quad \land\ (pc[i] = \text{“cs”}) \Rightarrow \land\ x = i \\
\quad \qquad\qquad\qquad\qquad \land\ \forall j \in \mathsf{Proc} : pc[j] \neq \text{“b”} \\
\quad \land\ (pc[i] = \text{“c”}) \Rightarrow \land\ x \neq 0 \\
\quad \qquad\qquad\qquad\qquad \land\ \forall j \in \mathsf{Proc} : (pc[j] = \text{“b”}) \Rightarrow (t_c[i] > T_b[j]) \\
\quad \land\ (pc[i] = \text{“b”}) \Rightarrow (T_b[i] < now + \delta_c) \\
\quad \land\ now \leq T_b[i]
\end{array}$$

and the assumption

$$Assump \triangleq (0 \notin \mathsf{Proc}) \land (\Delta_b, \delta_c \in \mathbf{R}) \land (\Delta_b < \delta_c)$$

4. OPEN SYSTEMS

A closed system is solipsistic. An open system interacts with an environment, where system steps are distinguished from environment steps. Sections 4.1 and 4.2 reformulate a number of concepts introduced in [Abadi and Lamport 1993] that are needed for treating open systems in TLA. Some new results appear in Section 4.3. The following two sections explain how reasoning about open systems is reduced to reasoning about closed systems, and how open systems are composed.

4.1 Receptiveness and Realizability

To describe an open system in TLA, one defines an action μ such that μ steps are attributed to the system and $\neg\mu$ steps are attributed to the environment. A specification should constrain only system steps, not environment steps.

For safety properties, the concept of constraining is formalized as follows: if μ is an action and Π a safety property, then Π *constrains at most* μ iff, for any finite behavior $s_1, \ldots, s_n$ and state s_{n+1}, if $s_1, \ldots, s_n$ satisfies Π and (s_n, s_{n+1}) is a $\neg\mu$ step, then $s_1, \ldots, s_{n+1}$ satisfies Π. The generalization to arbitrary properties of constraining at most μ is μ*-receptiveness.* Intuitively, Π is μ-receptive iff every behavior in Π can be achieved by an implementation that performs only μ steps—the environment being able to perform any $\neg\mu$ step. The concept of receptiveness is due to Dill [1988]. The generalization to μ-receptiveness is developed in [Abadi and Lamport 1993].[8] A safety property is μ-receptive iff it constrains at most μ.

The generalization of machine closure to open systems is *machine realizability.* Intuitively, (Π, L) is μ-machine realizable iff an implementation that performs only

[8]To translate from the semantic model of [Abadi and Lamport 1993] into that of TLA, we let agents be pairs of states and identify an action μ with the set of all agents that are μ steps. A TLA behavior $s_1, s_2, \ldots$ corresponds to the sequence $s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} s_3 \xrightarrow{\alpha_4} \ldots$, where α_i equals (s_{i-1}, s_i). With this translation, the definitions in [Abadi and Lamport 1993] differ from the ones given here and in the appendix mainly by attributing the choice of initial state to the environment rather than to the system, requiring initial conditions to be assumptions about the environment rather than guarantees by the system.

μ steps can ensure that any finite behavior satisfying Π is completed to an infinite behavior satisfying $\Pi \wedge L$. Formally, (Π, L) is defined to be μ-machine realizable iff (Π, L) is machine closed and $\Pi \wedge L$ is μ-receptive. For μ equal to true, machine realizability reduces to machine closure.

4.2 The $\mathrel{-\!\!\triangleright}$ Operator

A common way of specifying an open system is in terms of assumptions and guarantees [Jones 1983], requiring the system to guarantee a property M if its environment satisfies an assumption E. An obvious formalization of such a specification is the property $E \Rightarrow M$. However, this property contains behaviors in which the system violates M and then the environment later violates E. Because the system cannot predict what the environment will do, such behaviors cannot occur in any actual implementation. A behavior σ generated by any implementation satisfies the additional property that if any finite prefix of σ satisfies E, then it satisfies M. We can therefore formalize the assumption/guarantee specification by the property $E \mathrel{-\!\!\triangleright} M$, defined by: $\sigma \in E \mathrel{-\!\!\triangleright} M$ iff $\sigma \in (E \Rightarrow M)$ and, for every finite prefix ρ of σ, if ρ satisfies E then ρ satisfies M. If E and M are safety properties, then $E \mathrel{-\!\!\triangleright} M$ is as well.

For safety properties, the operator $\mathrel{-\!\!\triangleright}$ is the implication operator of an intuitionistic logic [Abadi and Plotkin 1992]. Most valid propositional formulas without negation remain valid when $\Rightarrow$ is replaced by $\mathrel{-\!\!\triangleright}$, if all the formulas that appear on the left of a $\mathrel{-\!\!\triangleright}$ are safety properties. For example, the following formulas are valid if Φ and Π are safety properties.

$$\begin{aligned} \Phi \mathrel{-\!\!\triangleright} (\Pi \mathrel{-\!\!\triangleright} \Psi) &\equiv (\Phi \wedge \Pi) \mathrel{-\!\!\triangleright} \Psi \\ (\Phi \mathrel{-\!\!\triangleright} \Psi) \wedge (\Pi \mathrel{-\!\!\triangleright} \Psi) &\equiv (\Phi \vee \Pi) \mathrel{-\!\!\triangleright} \Psi \end{aligned} \tag{9}$$

For any TLA formulas Φ and Π, the property $\Phi \mathrel{-\!\!\triangleright} \Pi$ is expressible as a TLA formula.

4.3 Proving Machine Realizability

Propositions 1–3, which concern machine closure, have generalizations for machine realizability. Proposition 1 is the special case of Proposition 4 in which Φ and μ are identically true. Proposition 3 is similarly a special case of Proposition 5 if (true, L_2) is machine closed—that is, if L_2 is a liveness property. This is sufficient for our purposes, since NZ is a liveness property. The generalization of Proposition 2 is omitted; it would be analogous to Proposition 10 of [Abadi and Lamport 1993].

Proposition 4 is stated in terms of μ-invariance, which generalizes the ordinary concept of invariance. A predicate P is a μ-invariant of a formula Π iff, in any behavior satisfying Π, no μ-step makes P false. This condition is expressed by the TLA formula $\Pi \Rightarrow \Box[(\mu \wedge P) \Rightarrow P']_P$.

PROPOSITION 4. *If Π and Φ are safety properties, Π constrains at most μ, and L is the conjunction of a finite or countably infinite number of formulas of the form* $\mathrm{WF}_w(\mathcal{A})$ *and/or* $\mathrm{SF}_w(\mathcal{A})$, *where, for each such formula,*

(1) $\langle \mathcal{A} \rangle_w$ *is a subaction of* $\Pi \wedge \Phi$,

(2) $\Pi \wedge \Phi \Rightarrow \Box[\langle \mathcal{A} \rangle_w \Rightarrow \mu]_w$,

(3) if $\mathcal{A}$ appears in a formula $\mathrm{SF}_w(\mathcal{A})$, then Enabled $\langle\mathcal{A}\rangle_w$ is a $\neg\mu$-invariant of $\Pi \wedge \Phi$,

then $(\Phi \mathrel{-\!\!\triangleright} \Pi,\ \Phi \Rightarrow L)$ is μ-machine realizable.

PROPOSITION 5. *If Φ and Π are safety properties, $(\Phi \mathrel{-\!\!\triangleright} \Pi, L_1)$ and (true, L_2) are μ-machine realizable, and $\Phi \wedge \Pi \wedge L_1$ implies L_2, then $(\Phi \mathrel{-\!\!\triangleright} \Pi, L_2)$ is μ-machine realizable.*

4.4 Reduction to Closed Systems

Consider a specification $E \mathrel{-\!\!\triangleright} M$, where E and M are safety properties. We expect the system's requirement to restrict only system steps, meaning that M constrains at most μ. This implies that $E \mathrel{-\!\!\triangleright} M$ also constrains at most μ. We also expect the environment assumption E not to constrain system steps; formally, E *does not constrain* μ iff it constrains at most $\neg\mu$ and it is satisfied by every (finite behavior consisting only of an) initial state.[9]

Suppose E and M have the following form:

$$E \triangleq \Box[\mu \vee \mathcal{N}_E]_v$$
$$M \triangleq \mathit{Init} \wedge \Box[\neg\mu \vee \mathcal{N}_M]_v$$

Then E does not constrain μ and M constrains at most μ. If the system's next-state action $\mathcal{N}_M$ implies μ, and the environment's next-state action $\mathcal{N}_E$ implies $\neg\mu$, then a simple calculation shows that

$$E \wedge M \equiv \mathit{Init} \wedge \Box[\mathcal{N}_E \vee \mathcal{N}_M]_v \tag{10}$$

Conjunction represents parallel composition, so $E \wedge M$ is the formula describing the closed system consisting of the open system together with its environment. Observe that $E \wedge M$ has precisely the form we expect for a closed system comprising two components with next-state actions $\mathcal{N}_E$ and $\mathcal{N}_M$.

We can make the inverse transformation from a closed system specification Π to the corresponding assumption/guarantee specification $E \mathrel{-\!\!\triangleright} M$ such that Π equals $E \wedge M$, where E does not constrain μ and M constrains at most μ. This is possible because any safety property Π can be written as such a conjunction.

Implementation means implication. A system with guarantee M implements a system with guarantee $\widehat{M}$, under environment assumption E, iff $E \mathrel{-\!\!\triangleright} M$ implies $E \mathrel{-\!\!\triangleright} \widehat{M}$. It follows from the definition of $\mathrel{-\!\!\triangleright}$ that, when E and M are safety properties, $E \mathrel{-\!\!\triangleright} M$ implies $E \mathrel{-\!\!\triangleright} \widehat{M}$ iff $E \wedge M$ implies $E \wedge \widehat{M}$. Thus, proving that one open system implements another is equivalent to proving the implementation relation for the corresponding closed systems. Implementation for open systems therefore reduces to implementation for closed systems.

4.5 Composition

The distinguishing feature of open systems is that they can be composed. The proof that the composition of two specifications implements a third specification is

[9]The asymmetry between *constrains at most* and *does not constrain* arises because we assign the system responsibility for the initial state.

based on the following result, which is a reformulation of Theorem 2 of [Abadi and Lamport 1993] for safety properties.

THEOREM 3. *If E, E_1, E_2,, M_1, and M_2 are safety properties and μ_1 and μ_2 are actions such that*

(1) E_1 does not constrain μ_1 and E_2 does not constrain μ_2,

(2) M_1 constrains at most μ_1 and M_2 constrains at most μ_2,

then the following proof rule is valid:

$$\frac{E \wedge M_1 \wedge M_2 \;\Rightarrow\; E_1 \wedge E_2}{(E_1 \mathrel{-\!\!\triangleright} M_1) \wedge (E_2 \mathrel{-\!\!\triangleright} M_2) \;\Rightarrow\; (E \mathrel{-\!\!\triangleright} M_1 \wedge M_2)}$$

This theorem is essentially the same as Theorem 1 of [Abadi and Plotkin 1992]; the proof is omitted.

5. REAL-TIME OPEN SYSTEMS

In Section 3, we saw how we can represent time by the variable *now* and introduce timing constraints with timers. To extend the method to open systems, we need only decide how to separate timing properties into environment assumptions and system guarantees. An examination of a paradoxical example in Section 5.1 leads to the general form described in Section 5.2, where the concept of nonZenoness is generalized.

5.1 A Paradox

Consider the two components $\boldsymbol{\Pi}_1$ and $\boldsymbol{\Pi}_2$ of Figure 5. Let the specification of $\boldsymbol{\Pi}_1$ be $P_y \mathrel{-\!\!\triangleright} P_x$, which asserts that it writes a "good" sequence of outputs on x if its environment writes a good sequence of inputs on y. Let $P_x \mathrel{-\!\!\triangleright} P_y$ be the specification of $\boldsymbol{\Pi}_2$, so $\boldsymbol{\Pi}_2$ writes a good sequence of outputs on y if its environment writes a good sequence of inputs on x. If P_x and P_y are safety properties, then it appears that we should be able to apply Theorem 3, our composition principle, to deduce that the composite system $\boldsymbol{\Pi}_{12}$ satisfies $P_x \wedge P_y$, producing good sequences of values on x and y. (We can define μ_1 and μ_2 so that writing on x is a μ_1 action and writing on y is a μ_2 action.)

Now, suppose P_x and P_y both assert that the value 0 is written by noon. These can be regarded as safety properties, since they assert that an undesirable event never occurs—namely, noon passing without a 0 having been written. Hence, the composition principle apparently asserts that $\boldsymbol{\Pi}_{12}$ sends 0's along both x and y by noon. However, the specifications of $\boldsymbol{\Pi}_1$ and $\boldsymbol{\Pi}_2$ are satisfied by systems that wait for a 0 to be input, whereupon they immediately output a 0. The composition of those two systems does nothing.

This paradox depends on the ability of a system to respond instantaneously to an input. It is tempting to rule out such systems—perhaps even to outlaw specifications like these. We show that this Draconian measure is unnecessary. Indeed, if the specification of $\boldsymbol{\Pi}_2$ is strengthened to assert that a 0 must unconditionally be written on y by noon, then there is no paradox, and the composition does guarantee that a 0 is written on both x and y by noon. All paradoxes disappear when one carefully examines how the specifications must be written.

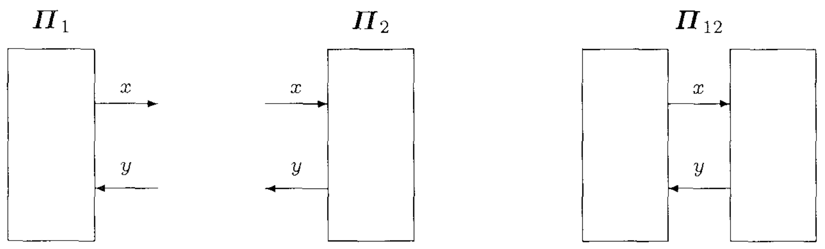

Fig. 5. The composition of two systems.

To resolve the paradox, we examine more closely the specifications S_1 and S_2 of Π_1 and Π_2. For simplicity, let the only possible output actions be the setting of x and y to 0. The untimed version of S_1 then asserts that, if the environment does nothing but set y to 0, then the system does nothing but set x to 0. This is expressed in TLA by letting

$$\begin{aligned} \mathcal{M}_x &\triangleq (x' = 0) \wedge (y' = y) \qquad\qquad \nu_1 \triangleq x' \neq x \\ \mathcal{M}_y &\triangleq (y' = 0) \wedge (x' = x) \end{aligned}$$

and defining the untimed version of specification S_1 to be

$$\Box[\nu_1 \vee \mathcal{M}_y]_{(x,y)} \mathrel{-\!\!\triangleright} \Box[\neg\nu_1 \vee \mathcal{M}_x]_{(x,y)} \tag{11}$$

To add timing constraints, we must first decide whether the system or the environment should change *now*. Since the advancing of *now* is a mythical action that does not have to be performed by any device, either decision is possible. Somewhat surprisingly, it turns out to be more convenient to let the system advance time. With the convention that initial conditions appear in the system guarantee, we define:

$$\begin{aligned} \mathcal{N}_x &\triangleq \mathcal{M}_x \wedge (now' = now) & MT_x &\triangleq MaxTime(T_x) \\ \mathcal{N}_y &\triangleq \mathcal{M}_y \wedge (now' = now) & MT_y &\triangleq MaxTime(T_y) \\ T_x &\triangleq \textbf{if } x \neq 0 \textbf{ then } 12 \textbf{ else } \infty & \mu_1 &\triangleq \nu_1 \vee (now' \neq now) \\ T_y &\triangleq \textbf{if } y \neq 0 \textbf{ then } 12 \textbf{ else } \infty \\ E_1 &\triangleq \Box[\mu_1 \vee \mathcal{N}_y]_{(x,y,now)} \\ M_1 &\triangleq (now = 0) \wedge \Box[\neg\mu_1 \vee \mathcal{N}_x]_{(x,y,now)} \wedge RT_{(x,y)} \wedge MT_x \end{aligned}$$

Adding timing constraints to (11) the same way we did for closed systems then leads to the following timed version of specification S_1.

$$(E_1 \wedge MT_y) \mathrel{-\!\!\triangleright} M_1 \tag{12}$$

However, this does not have the right form for an open system specification because MT_y constrains the advance of *now*, so the environment assumption constrains μ_1. The conjunct MT_y must be moved from the environment assumption to the system guarantee. Using (9), we rewrite (12) as:

$$S_1 \triangleq E_1 \mathrel{-\!\!\triangleright} (MT_y \mathrel{-\!\!\triangleright} M_1)$$

This has the expected form for an open system specification, with an environment assumption E_1 that does not constrain μ_1 and a system guarantee $MT_y \mathrel{-\!\!\triangleright} M_1$ that constrains at most μ_1.

The specification S_2 of the second component in Figure 5 is similar, where μ_2, E_2, M_2, and S_2 are obtained from μ_1, E_1, M_1, and S_1 by substituting 2 for 1, x for y, and y for x.

Both components Π_1 and Π_2 change *now*. This is not a problem because the components do not really control time. We have merely written the specifications S_1 and S_2 as if *now* were an output variable of both Π_1 and Π_2. Formulas S_1 and S_2 express real-time constraints by making assertions about how *now* changes. There is no problem because these constraints do not conflict.

We now compose specifications S_1 and S_2. The definitions of M_1 and M_2 and the observation that $P \mathrel{-\!\!\triangleright} Q$ implies $P \Rightarrow Q$ yield

$$(MT_x \vee MT_y) \wedge (MT_y \mathrel{-\!\!\triangleright} M_1) \wedge (MT_x \mathrel{-\!\!\triangleright} M_2) \Rightarrow M_1 \wedge M_2 \qquad (13)$$

The definitions of M_1 and M_2 and simple temporal reasoning yield

$$E \wedge M_1 \wedge M_2 \Rightarrow E_1 \wedge E_2 \qquad (14)$$

where

$$E \triangleq \Box[\mu_1 \vee \mu_2]_{(x,y,now)}$$

Combining (13) and (14) proves

$$E \wedge (MT_x \vee MT_y) \wedge (MT_y \mathrel{-\!\!\triangleright} M_1) \wedge (MT_x \mathrel{-\!\!\triangleright} M_2) \Rightarrow E_1 \wedge E_2$$

We can therefore apply Theorem 3, substituting $E \wedge (MT_x \vee MT_y)$ for E, $MT_y \mathrel{-\!\!\triangleright} M_1$ for M_1, and $MT_x \mathrel{-\!\!\triangleright} M_2$ for M_2, to deduce

$$S_1 \wedge S_2 \Rightarrow (E \wedge (MT_x \vee MT_y) \mathrel{-\!\!\triangleright} (MT_y \mathrel{-\!\!\triangleright} M_1) \wedge (MT_x \mathrel{-\!\!\triangleright} M_2))$$

Using the implication-like properties of $\mathrel{-\!\!\triangleright}$, this simplifies to

$$S_1 \wedge S_2 \Rightarrow (E \mathrel{-\!\!\triangleright} (MT_y \mathrel{-\!\!\triangleright} M_1) \wedge (MT_x \mathrel{-\!\!\triangleright} M_2)) \qquad (15)$$

All one can conclude about the composition from (15) is: either x and y are both 0 when *now* reaches 12, or neither of them is 0 when *now* reaches 12. There is no paradox.

As another example, we replace S_2 by the specification $E_2 \mathrel{-\!\!\triangleright} M_2$. This specification, which we call S_3, asserts that the system sets y to 0 by noon, regardless of whether the environment sets x to 0. The definitions imply

$$MT_y \wedge E \wedge (MT_y \mathrel{-\!\!\triangleright} M_1) \wedge M_2 \Rightarrow E_1 \wedge E_2$$

and Theorem 3 yields

$$S_1 \wedge S_3 \Rightarrow (E \mathrel{-\!\!\triangleright} (MT_x \mathrel{-\!\!\triangleright} M_1) \wedge M_2)$$

Since M_2 implies MT_x, this simplifies to

$$S_1 \wedge S_3 \Rightarrow (E \mathrel{-\!\!\triangleright} M_1 \wedge M_2)$$

The composition of S_1 and S_3 does guarantee that both x and y equal 0 by noon.

5.2 Timing Constraints in General

Our no-longer-paradoxical example suggests that the form of a real-time open system specification should be

$$E \mathrel{-\!\!\triangleright} (P \mathrel{-\!\!\triangleright} M) \tag{16}$$

where M describes the system's timing constraints and the advancing of *now*, and P describes the upper-bound timing constraints for the environment. Since the environment's lower-bound timing constraints do not constrain the advance of *now*, they can remain in E. As we observed in Section 4.4, proving that one open specification implements another reduces to the proof for the corresponding closed systems. Since $E \mathrel{-\!\!\triangleright} (P \mathrel{-\!\!\triangleright} M)$ is equivalent to $(E \wedge P) \mathrel{-\!\!\triangleright} M$, the closed system corresponding to (16) is the expected one, $E \wedge P \wedge M$.

For the specification (16) to be reasonable, its closed system version, $E \wedge P \wedge M$, should be nonZeno. However, this is not sufficient. Consider a specification guaranteeing that the system produces a sequence of outputs until the environment sends a *stop* message, where the n^{th} output must occur by time $(n-1)/n$. There is no timing assumption on the environment; it need never send a *stop* message. This is an unreasonable specification because *now* cannot reach 1 until the environment sends its *stop* message, so the advance of time is contingent on an optional action of the environment. However, the corresponding closed system specification is nonZeno, since time can always be made to advance without bound by having the environment send a *stop* message.

If advancing *now* is a μ action, then a system that controls μ actions can guarantee time to be unbounded while satisfying a safety specification S iff the pair (S, NZ) is μ-machine realizable. We therefore take this condition to be the definition of nonZenoness for an open system specification S.

For specifications in terms of δ-timers, nonZenoness can be proved with generalizations to open systems of the theorems in Section 3.3. The following is the generalization of the strongest of them, Theorem 2. It is applied to a specification of the form (16) by substituting $E \wedge P$ for E.

THEOREM 4. *With the notation and hypotheses of Theorem 2, if E and M are safety properties such that $\Pi = E \wedge M$, and*

4. M constrains at most μ,

5. (a) $\langle \mathcal{A}_k \rangle_v \Rightarrow \mu$, for all $k \in I \cup J$,
(b) $(now' \neq now) \Rightarrow \mu$

then $(E \mathrel{-\!\!\triangleright} M^t,\ NZ)$ is μ-machine realizable, where

$$\begin{aligned} M^t \triangleq\ & M \ \wedge\ RT_v \wedge \\ & \forall i \in I : MinTime(t_i, \mathcal{A}_i, v)\ \wedge\ \forall j \in J : MaxTime(T_j) \end{aligned}$$

Hypothesis 4 says that μ steps are attributed to the system represented by M. Part (a) of Hypothesis 5 says that the other hypotheses restrict the timing constraints on system actions $\langle \mathcal{A}_k \rangle_v$ only; environment actions may have any timing constraints. Part (b) says that advancing *now* is a system action.

The proof of Theorem 4, which appears in the appendix, is similar to the proof of Theorem 2 sketched in Section 3.3. It uses Propositions 4 and 5 instead of Propositions 1 and 3. Since machine realizability implies machine closure, Theorem 2 follows from Theorem 4 by letting E and μ equal true and M equal Π.

Theorem 4 applies to the internal specifications, where all variables are visible. For closed systems, existential quantification is handled with Proposition 2. For open systems, the generalization of this proposition—the analog of Proposition 10 of [Abadi and Lamport 1993]—is needed.

6. CONCLUSION

6.1 What We Did

We started with a simple idea—specifying and reasoning about real-time systems by representing time as an ordinary variable. This idea led to an exposition that most readers probably found quite difficult. What happened to the simplicity?

About half of the exposition is a review of concepts unrelated to real time. All the fundamental concepts described in Sections 2 and 4, including machine closure, machine realizability, and the $\mathrel{-\!\!\triangleright}$ operator, have appeared before [Abadi and Lamport 1993; Abadi and Lamport 1991]. These concepts are subtle, but they are important for understanding any concurrent system; they were not invented for real-time systems.

We chose to formulate these concepts in TLA. Like any language, TLA seems complicated on first encounter. We believe that a true measure of simplicity of a formal language is the simplicity of its formal description. The complete syntax and formal semantics of TLA are given in about one page in [Lamport 1994].

We never claimed that specifying and reasoning about concurrent systems is easy. Verifying concurrent systems is difficult and error prone. Our assertions that one formula follows from another, made so casually in the exposition, must be backed up by detailed calculations. We have omitted the proofs for our examples, which, done with the same detail as the proofs in the appendix, occupy some twenty pages.

We did claim that existing methods for specifying and reasoning about concurrent systems could be applied to real-time systems. Now, we can examine how hard they were to apply.

We found few obstacles in the realm of closed systems. The second author has more than fifteen years of experience in the formal verification of concurrent algorithms, and we knew that old-fashioned methods could be applied to real-time systems. However, TLA is relatively new, and we were pleased by how well it worked. The formal specification of Fischer's protocol in Figure 4, obtained by conjoining timing constraints to the untimed protocol, is as simple and direct as we could have hoped for. Moreover, the formal correctness proofs of this protocol and of the queue example, using the method of reasoning described in [Lamport 1994], were straightforward. Perhaps the most profound discovery was the relation between nonZenoness and machine closure.

Open systems made up for any lack of difficulty with closed systems. State-based approaches to open systems were a fairly recent development, and we had little experience with them. Studying real-time systems taught us a great deal, and led to a number of changes from the approach in [Abadi and Lamport 1993]. For example, we now write specifications with $\mathrel{-\!\!\triangleright}$ instead of $\Rightarrow$, and we put initial conditions in the system guarantee rather than in the environment assumption. Many alternative ways of writing real-time specifications seemed plausible; choosing one that works was surprisingly hard. Even the simple idea of putting the environment's timing assumptions to the left of a $\rightarrow\!\!\!\triangleright$ in the system's guarantee came only after numerous failed efforts. Although the basic ideas we need to handle real-time open systems seem to be in place, we still have much to learn before reasoning about open systems becomes routine.

6.2 Beyond Real Time

Real-time systems introduce a fundamentally new problem: adding physical continuity to discrete systems. Our solution is based on the observation that, when reasoning about a discrete system, we can represent continuous processes by discrete actions. If we can pretend that the system progresses by discrete atomic actions, we can pretend that those actions occur at a single instant of time, and that the continuous change to time also occurs in discrete steps. If there is no system action between noon and $\sqrt{2}$ seconds past noon, we can pretend that time advances by those $\sqrt{2}$ seconds in a single action.

Physical continuity arises not just in real-time systems, but in "real-pressure" and "real-temperature" process-control systems. Such systems can be described in the same way as real-time systems: pressure and temperature as well as time are represented by ordinary variables. The continuous changes to pressure and temperature that occur between system actions are represented by discrete changes to the variables. The fundamental assumption is that the real, physical system is accurately represented by a model in which the system makes discrete, instantaneous changes to the physical parameters it affects.

The observation that continuous parameters other than time can be modeled by program variables has probably been known for years. However, the first published work we know of that uses this idea, by Marzullo et al. [1991], appeared only recently.

REFERENCES

ABADI, M. AND LAMPORT, L. 1991. The existence of refinement mappings. *Theoretical Computer Science 82,* 2 (May), 253–284.

ABADI, M. AND LAMPORT, L. 1993. Composing specifications. *ACM Trans. on Programm. Lang. Syst. 15,* 1 (Jan.), 73–132.

ABADI, M. AND PLOTKIN, G. 1992. A logical view of composition. Research Report 86 (May), Digital Equipment Corporation, Systems Research Center.

ALPERN, B. AND SCHNEIDER, F. B. 1985. Defining liveness. *Inf. Process. Lett. 21,* 4 (Oct.), 181–185.

APT, K. R., FRANCEZ, N., AND KATZ, S. 1988. Appraising fairness in languages for distributed programming. *Distributed Computing 2*, 226–241.

BERNSTEIN, A. AND HARTER, JR., P. K. 1981. Proving real time properties of programs with temporal logic. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, New York, pp. 1–11. ACM. *Operating Systems Review 15*, 5.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design.* Addison-Wesley, Reading, Massachusetts.

DE BAKKER, J. W., HUIZING, C., DE ROEVER, W. P., AND ROZENBERG, G. (Eds.) 1992. *Real-Time: Theory in Practice*, Volume 600 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin. Proceedings of a REX Real-Time Workshop, held in The Netherlands in June, 1991.

DILL, D. L. 1988. Trace theory for automatic hierarchical verification of speed-independent circuits. Ph. D. thesis, Carnegie Mellon University.

FISCHER, M. 1985. Re: Where are you? E-mail message to Leslie Lamport. Arpanet message sent on June 25, 1985 18:56:29 EDT, number 8506252257.AA07636@YALE-BULLDOG.YALE.ARPA (47 lines).

JONES, C. B. 1983. Specification and design of (parallel) programs. In R. E. A. MASON (Ed.), *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pp. 321–332. IFIP: North-Holland.

LAMPORT, L. 1982. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming 2,* 3 (Dec.), 175–206.

LAMPORT, L. 1987. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems 5,* 1 (Feb.), 1–11.

LAMPORT, L. 1994. The temporal logic of actions. *ACM Trans. on Programm. Lang. Syst. 16,* 3 (May), 872–923.

MARZULLO, K., SCHNEIDER, F. B., AND BUDHIRAJA, N. 1991. Derivation of sequential, real-time process-control programs. In A. M. VAN TILBORG AND G. M. KOOB (Eds.), *Foundations of Real-Time Computing: Formal Specifications and Methods*, Chapter 2, pp. 39–54. Boston, Dordrecht, and London: Kluwer Academic Publishers.

MISRA, J. AND CHANDY, K. M. 1981. Proofs of networks of processes. *IEEE Transactions on Software Engineering SE-7,* 4 (July), 417–426.

NEUMANN, P. G. AND LAMPORT, L. 1983. Highly dependable distributed systems. Technical report (June), SRI International. Contract Number DAEA18-81-G-0062, SRI Project 4180.

PNUELI, A. 1984. In transition from global to modular temporal reasoning about programs. In K. R. APT (Ed.), *Logics and Models of Concurrent Systems*, NATO ASI Series, pp. 123–144. Springer-Verlag.

SCHNEIDER, F. B., BLOOM, B., AND MARZULLO, K. 1992. Putting time into proof outlines. In J. W. DE BAKKER, C. HUIZING, W.-P. DE ROEVER, AND G. ROZENBERG (Eds.), *Real-Time: Theory in Practice*, Volume 600 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, pp. 618–639. Springer-Verlag.

Received November 1992; revised November 1993; accepted April 1994.