An Old-Fashioned Recipe for Real Time

Martín Abadi and Leslie Lamport
Digital Equipment Corporation
130 Lytton Avenue
Palo Alto, California 94301, USA

Abstract. Traditional methods for specifying and reasoning about concurrent systems work for real-time systems. However, two problems arise: the real-time programming version of Zeno's paradox, and circularity in composing real-time assumption/guarantee specifications. Their solutions rest on properties of machine closure and realizability. TLA (the temporal logic of actions) provides a formal framework for the exposition.

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 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 has been 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 [7, 9]. The first application we have seen of a clearly "off-the-shelf" method to a real-time algorithm was in 1983 [13], but there were probably earlier ones. Indeed, the "extension" of an existing temporal logic to real-time programs by Bernstein and Harter in 1981 [6] 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 worth writing about, except that few people appear to be aware that it works in practice. We therefore present a brief description of how to apply conventional methods to real-time systems. We also discuss two problems with this approach that seem to have received little attention, and we present new solutions.

The first problem is 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.

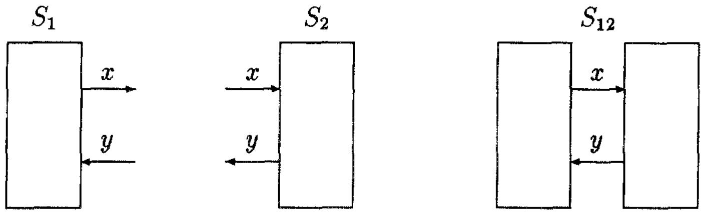

Figure 1: The composition of two systems.

The second problem arises in open system specifications, in which a system is required to operate correctly only under some assumptions on its environment. A modular specification method requires a rule asserting that, if each component behaves correctly in isolation, then it behaves correctly in concert with other components. Consider the two components S_1 and S_2 of Figure 1. Suppose that S_1 guarantees to produce a sequence of outputs on x satisfying P_x assuming it receives a sequence of inputs on y satisfying P_y, and S_2 guarantees to produce a sequence of outputs on y satisfying P_y assuming it receives a sequence of inputs on x satisfying P_x. If P_x and P_y are safety properties, then existing composition principles permit the conclusion that, in the composite system S_{12}, the sequence of values on x and y satisfy P_x and P_y [1]. Now, suppose P_x and P_y both assert that the value 0 is sent by noon. These are safety properties, asserting that the undesirable event of noon passing without a 0 having been sent does not occur. Hence, the composition principle apparently asserts that S_{12} sends 0's along both x and y by noon. However, specifications S_1 and S_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 S_1 and S_2. We show that this Draconian measure is unnecessary. Indeed, if S_2 is replaced by the specification that a 0 must unconditionally be sent over y by noon, then there is no paradox, and the composition does guarantee that a 0 is sent on each wire by noon. All paradoxes disappear when one carefully examines how the specifications must be written.

Our results are relevant for any method whose semantics is based on sequences of states or actions. However, we will describe them only for TLA—the temporal logic of actions [11].

2 Closed Systems

We briefly review here 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.

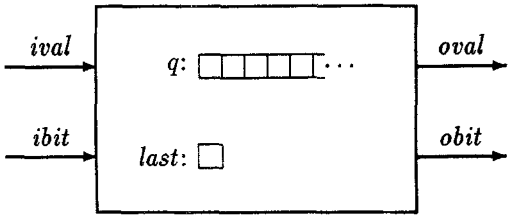

Figure 2: A simple queue.

We begin our review of TLA with an example. Section 2.2 summarizes the formal definitions—a more leisurely exposition can be found in [11]. Section 2.3 reviews the concepts of safety [4] and machine closure [2] (also known as feasibility [5]) and relates them to TLA, and Section 2.4 defines a useful class of history variables [2].

2.1 The Lossy-Queue Example

We introduce TLA with the example of the lossy queue shown in Figure 2. 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. 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. 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.

The specification of the lossy queue is a TLA formula describing the set of allowed *behaviors* of the queue, where a behavior is the sequence of states produced by an execution. The specification mentions the four variables *ibit*, *obit*, *ival*, and *oval*, as well as two internal variables: q, which equals 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 3, using the following notation. A list of formulas prefaced by $\wedge$'s 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

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

Figure 3: The TLA specification of a lossy queue.

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 (if and only if) 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; a variable may assume any value. Type correctness is 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

$$T_Q \;\triangleq\; \begin{array}{l} \land\; ibit, obit, last \in \{\mathsf{true}, \mathsf{false}\} \\ \land\; ival, oval \in \mathsf{Msg} \\ \land\; q \in \mathsf{Msg}^* \end{array} \tag{1}$$

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 [11]. 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 $\mathrm{WF}_v(DeQ)$ and the *strong fairness* property $\mathrm{SF}_v(EnQ)$ to Π_Q, obtaining

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

Property $\mathrm{WF}_v(DeQ)$ 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 $\mathrm{SF}_v(EnQ)$ 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* [4] that an infinite number of inputs produces an infinite number of outputs. This formula also implies the same safety properties as Φ_Q. 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 set of constant *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. 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. It is called an $\mathcal{A}$ *step* iff $[\![\mathcal{A}]\!](s,t)$ equals true, for an action $\mathcal{A}$. 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 $\mathcal{A} \lor (f' = f)$, for any action $\mathcal{A}$ and state function f.

$\langle \mathcal{A} \rangle_f$ The action $\mathcal{A} \land (f' \neq f)$, 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 confuse 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 an 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

$$[\![\Box F]\!](s_0, s_1, s_2, \ldots) \;\triangleq\; \forall n \geq 0 : [\![F]\!](s_n, s_{n+1}, s_{n+2}, \ldots)$$

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 $[\![\mathcal{A}]\!](s_0, s_1, s_2, \ldots)$ equal $[\![\mathcal{A}]\!](s_0, s_1)$. 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. The semantics of actions, classical operators, and $\Box$ are defined as before. The approximate meaning of quantification over a flexible variable is that $\exists x : F$ 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 is in [11]. As usual, we write $\exists\, x_1, \ldots, x_n : F$ instead of $\exists\, x_1 : \ldots, \exists\, x_n : F$.

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 identify the finite behavior $s_0, \ldots, s_n$ with the behavior $s_0, \ldots, s_n, s_n, s_n, \ldots$. A property F is a *safety property* [4] iff the following condition holds: F contains a behavior iff it contains every finite prefix of the behavior. 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 form closed sets for a topology on the set of all behaviors. Hence, if two TLA formulas F and G are safety properties, 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 satisfying $\Pi \wedge L$. If Π is the set of behaviors allowed by the initial condition and next-state relation of a program, then machine closure of (Π, L) corresponds to the intuitive concept that L is a fairness property of the program. The *canonical form* for a TLA formula is

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

where $(\mathit{Init} \wedge \Box[\mathcal{N}]_v, L)$ is machine closed and x is a tuple of variables called the *internal variables* of the formula. The state function v will usually be the tuple of all variables appearing free in *Init*, $\mathcal{N}$, and L (including the variables of x). 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 for every finite behavior $s_0, \ldots, s_n$ in 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 $s_0, \ldots, s_{n+1}$ is in Π. By this definition, $\mathcal{A}$ is a subaction of $\mathit{Init} \wedge \Box[\mathcal{N}]_v$ iff[1]

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

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

Two actions $\mathcal{A}$ and $\mathcal{B}$ are *disjoint for* a safety property 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 $Init \wedge \Box[\mathcal{N}]_v$ iff

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

The following result shows that the conjunction of WF and SF formulas is a fairness property.

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.*

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 $\mathcal{A}$.[2] 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. To apply the first, one must prove that $\exists x : \Pi$ is a safety property. By Proposition 2 of [2, 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

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

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 $\mathit{Hist}(h, f, g, v)$, for some f, g, and v such that h does not occur free in f and 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 $\exists h : \mathit{Hist}(h, f, g, v)$ is identically true. Therefore, if h does not occur free in formula Φ, then $\exists h : (\Phi \wedge \mathit{Hist}(h, f, g, v))$ is equivalent to Φ. In other words, conjoining $\mathit{Hist}(h, f, g, v)$ 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 [2, 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

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

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

H_{in} equals $Hist(hin, \langle\langle\,\rangle\rangle, hin \circ \langle\langle ival' \rangle\rangle, (ival, ibit))$, so 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 Real-Time Closed Systems

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. Thus, RT_v is equivalent to $RT \wedge \Box[now' = now]_v$, and

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

Real-time constraints 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.[3]

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.

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

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

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

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

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

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.

- 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 4.[4] 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 out such behaviors by conjoining to Π_Q^t the liveness property

$$NZ \;\triangleq\; \forall t \in \mathbf{R} : \Diamond(now > t)$$

which asserts that *now* gets arbitrarily large. However, when reasoning only about real-time properties, this is not 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

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

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

Figure 4: The canonical form for Π_Q^t, where $(r, s]$ denotes the set of reals u such that $r < u \leq s$.

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—a point discussed in Section 3.3 below. Hence, there is no need to introduce the liveness 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 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 in TLA 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 TLA 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 $\Pi_Q^t \wedge NZ$ 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. NonZenoness means that the liveness property *NZ* cannot help in proving safety properties.[5] The following result is used to prove that a real-time specification written in terms of δ-timers is nonZeno.

Theorem 1 *Let*

- Π *be a safety property.*
- 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. $\langle \mathcal{A}_i \rangle_v$ *and* $\langle \mathcal{A}_j \rangle_v$ *are disjoint for* Π, *for all* $i \in I$ *and* $j \in J$ *with* $i \neq j$.

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*
   $\Pi \Rightarrow PTimer(T_j, \mathcal{A}_j, \Delta_j, v)$, *where* $\Delta_j \in (0, \infty)$.
   (c) $\Pi^t \Rightarrow \Box(Enabled\ \langle \mathcal{A}_j \rangle_v = Enabled\ (\langle \mathcal{A}_j \rangle_v \wedge (now' = now)))$

4. $\Pi^t \Rightarrow \Box(t_k \leq T_k)$, *for all* $k \in I \cap J$.

then (Π^t, NZ) *is machine closed*

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

We can apply the theorem to prove that the specification Π_Q^t is nonZeno if $\delta_{snd} \leq \Delta_{snd}$ by substituting

$$\begin{aligned} &\Pi_Q \wedge VTimer(t_{Inp}, Inp, \delta_{snd}, v) \wedge VTimer(t_{DeQ}, DeQ, \delta_{snd}, v) \\ &\quad \wedge VTimer(T_{EnQ}, EnQ, \Delta_{rcv}, v) \wedge VTimer(T_{DeQ}, DeQ, \Delta_{snd}, v) \end{aligned}$$

for Π, so Π^t equals Π_Q^t. The hypotheses of the theorem are checked as follows.

1. The actions $\langle Inp \rangle_v$, $\langle DeQ \rangle_v$, and $\langle EnQ \rangle_v$ are pairwise disjoint, so they are pairwise disjoint for Π_Q^t. (Two actions are said to be disjoint if their conjunction equals false.)
2. (a) Trivially satisfied.
   (b) Intuitively, this asserts that Π allows an arbitrary change to *now* when v remains unchanged, which holds because neither Π_Q nor the *VTimer* formulas constrain *now*. Formally, the hypothesis asserts that *Enabled* $((now' = r) \wedge (v' = v))$ implies *Enabled* $(\mathcal{M} \wedge (now' = r) \wedge (v' = v))$, for any $r \in \mathbf{R}$, where $\mathcal{M}$ is the conjunction of $[\mathcal{N}_Q]_v$ and the *VTimer* actions. The definitions of $\mathcal{N}_Q$ and *VTimer* imply that now' does not occur in $\mathcal{M}$, from which it follows that both *Enabled* predicates equal **true**. (The hypothesis would also hold if persistent instead of volatile Δ-timers had been used, but a rigorous proof is a bit more complicated.)
3. (a) Actions $\langle Inp \rangle_v$, $\langle DeQ \rangle_v$, and $\langle EnQ \rangle_v$ imply $\mathcal{N}_Q$, so they are subactions of Π_Q. Since these three actions have no primed variables in common with the *VTimer* formulas, they are subactions of Π. The hypothesis then follows because now' does not occur in the *VTimer* formulas. (Again, the hypothesis is true for persistent timers, but the proof is more involved.)
   (b) Immediate from the definition of Π.
   (c) Holds because now' does not occur in the actions *Inp*, *DeQ*, and *EnQ*.
4. Follows from the general result that $\delta \leq \Delta$ implies

   $$RT_v \wedge VTimer(t, \mathcal{A}, \delta, v) \wedge VTimer(T, \mathcal{A}, \Delta, v) \Rightarrow \Box(t \leq T)$$

   which is proved by a simple invariance argument. (The analogous result holds for persistent timers.)

Theorem 1 can be 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 1 is 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 defined by

$$\begin{array}{rcl}
\mathcal{A}^t_j & \triangleq & (now' = now) \wedge (\textbf{if } j \in I \textbf{ then } \mathcal{A}_j \wedge (now \geq t_j) \textbf{ else } \mathcal{A}_j) \\
J_E & \triangleq & \{j \in J : \mathit{Enabled}\ \langle \mathcal{A} \rangle_j\} \\
T & \triangleq & \min(now + \min_{j \in J} \Delta_j,\ \min_{j \in J_E} T_j) \\
\mathcal{B} & \triangleq & ((now = T) \wedge \mathcal{A}^t_j) \ \vee\ ((now \neq T) \wedge (now' = T) \wedge (v' = v)) \\
L & \triangleq & \mathrm{WF}_{(now, v)}(\mathcal{B})
\end{array}$$

We omit the proof.

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

Theorem 1 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 method of proof is the same as we used to prove Theorem 1: 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 [2, 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 Michael Fischer [10, page 2]. The example was

$$
\begin{aligned}
Init_F \;&\triangleq\; \forall\, i \in \mathsf{Proc} : pc[i] = \text{“a”} \\
Go(i, u, v) \;&\triangleq\; \land\; pc[i] = u \\
&\qquad \land\; pc'[i] = v \\
&\qquad \land\; \forall\, j \in \mathsf{Proc} : (j \neq i) \Rightarrow (pc'[j] = pc[j]) \\
\mathcal{A}_i \;&\triangleq\; Go(i, \text{“a”}, \text{“b”}) \land (x = x' = 0) \\
\mathcal{B}_i \;&\triangleq\; Go(i, \text{“b”}, \text{“c”}) \land (x' = i) \\
\mathcal{C}_i \;&\triangleq\; Go(i, \text{“c”}, \text{“cs”}) \land (x = x' = i) \\
\mathcal{N}_F \;&\triangleq\; \exists\, i \in \mathsf{Proc} : (\mathcal{A}_i \lor \mathcal{B}_i \lor \mathcal{C}_i) \\
\Pi_F \;&\triangleq\; Init_F \land \Box[\mathcal{N}_F]_{(x,pc)} \\
\Pi_F^t \;&\triangleq\; \land\; \Pi_F \land RT_{(x,pc)} \\
&\qquad \land\; \forall\, i \in \mathsf{Proc} : \land\; VTimer(T_b[i],\, \mathcal{B}_i,\, \Delta_b,\, (x, pc)) \\
&\qquad\qquad\qquad\qquad\; \land\; MaxTime(T_b[i]) \\
&\qquad \land\; \forall\, i \in \mathsf{Proc} : \land\; VTimer(t_c[i],\, Go(i, \text{“c”}, \text{“cs”}),\, \delta_c,\, (x, pc)) \\
&\qquad\qquad\qquad\qquad\; \land\; MinTime(t_c[i],\, \mathcal{C}_i,\, (x, pc)) \\
\Phi_F^t \;&\triangleq\; \exists\, T_b, t_c : \Pi_F^t
\end{aligned}
$$

Figure 5: The TLA specification of Fischer's real-time mutual exclusion protocol.

suggested by Fred Schneider [14]. The protocol consists of each process i executing the following code, where angle brackets denote instantaneous atomic actions:

$$
\begin{array}{rl}
a\text{:} & \textbf{await } \langle x = 0 \rangle; \\
b\text{:} & \langle x := i \rangle; \\
c\text{:} & \textbf{await } \langle x = i \rangle; \\
cs\text{:} & \text{critical section}
\end{array}
$$

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 5. 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-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. 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_F^t of Figure 5, the TLA specification of the real-time protocol with the timers visible. The final specification, Φ_F^t, is obtained by quantifying over the timer variables T_b and t_c. Since $\mathcal{B}_j$ is a subaction of Π_F and $pc[i] = \text{“c”}$ is disjoint from $\mathcal{B}_j$, for all i and j in Proc, Theorem 1 implies that Π_F^t is nonZeno if Δ_b is positive. Proposition 2 can then be applied to prove that Φ_F^t 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

$$Assump \wedge \Phi_F^t \;\Rightarrow\; \Box Mutex \tag{8}$$

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

$$Assump \wedge \Pi_F^t \;\Rightarrow\; \Box Mutex$$

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

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

and the assumption

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

4 Open Systems

4.1 Realizability

We begin by recasting the definitions of [1] into TLA. In the semantic model of [1], a behavior is a sequence of alternating states and agents of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots \tag{9}$$

To translate from this semantic model into that of TLA, we identify agents with state transitions. Agents are pairs of states, and a behavior $s_0, s_1, \ldots$ in TLA's model is identified with the behavior (9) in which α_i equals (s_{i-1}, s_i). An action μ is identified with the set of all agents that are μ steps. All the important definitions and results in [1] that do not concern agent-abstractness continue to hold—except that some results require the assumption that μ does not allow stuttering steps. (An action μ does not allow stuttering steps iff μ implies $v' \neq v$, where v is the tuple of all variables occurring in μ.)

If μ is an action and Π a safety property, then Π *does not constrain* μ iff for any finite behavior $s_0, \ldots, s_n$ and state s_{n+1}, if $s_0, \ldots, s_n$ satisfies Π and (s_n, s_{n+1}) is a μ step, then $s_0, \ldots, s_{n+1}$ satisfies Π. Property Π *constrains at most* μ iff Π does not constrain $\neg\mu$ and every behavior consisting of a single state satisfies Π. Any safety property Π can be written as the conjunction of a property Π_1 that does not constrain μ and a property Π_2 that constrains at most μ. If Π equals $Init \wedge \Box[\mathcal{N}]_v$, then we can take Π_1 to be $Init \wedge \Box[\mathcal{N} \vee \mu]_v$ and Π_2 to be $\Box[\mathcal{N} \vee \neg\mu]_v$.

A predicate P is said to be a μ *invariant* of a property Π iff no μ step of a behavior satisfying Π can make P false. More precisely, P is a μ invariant of Π iff

$$\Pi \Rightarrow \Box[\mu \wedge P \Rightarrow P']_P$$

For an action μ and property Π, the *μ-realizable part* $\mathcal{R}_\mu(\Pi)$ is the set of behaviors that can be achieved by an implementation of Π that performs only μ steps—the environment being able to perform any $\neg\mu$ step. The reader is referred to [1] for the precise definition.[6] (The concept of receptiveness is due to Dill [8].) Property Π is said to be *μ-receptive* iff it equals $\mathcal{R}_\mu(\Pi)$. The realizable part $\mathcal{R}_\mu(\Pi)$ of any TLA formula Π can be written as a TLA formula.

The generalization of machine closure to open systems is *machine realizability*. Intuitively, (Π, L) is μ-machine realizable iff an implementation that performs only 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. Corresponding to Propositions 1, 2 and 3 are:

---

[6] $\mathcal{R}_\mu(\Pi)$ was not defined in [1] if μ equals true or false. The appropriate definitions are $\mathcal{R}_{\mathsf{true}}(\Pi) \triangleq \Pi$ and $\mathcal{R}_{\mathsf{false}}(\Pi) \triangleq \mathsf{false}$.

Proposition 4 *If* Π *is a safety property that 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 (a) each* $\langle\mathcal{A}\rangle_w$ *is a subaction of* Π *and (b) Enabled* $\langle\mathcal{A}\rangle_w$ *is a* $\neg\mu$ *invariant of* Π *for each* $\mathcal{A}$ *appearing in a formula* $\mathrm{SF}_w(\mathcal{A})$*, then* (Π, L) *is* μ*-machine realizable.*

Proposition 5 ([1], Proposition 10) *If* μ *does not allow stuttering steps,* x *is a tuple of variables that do not occur free in* μ *or* L*, and*

(a) $\exists x : \mathit{Init}$ *holds.*

(b) $(\Box[\mathcal{N} \vee \neg\mu]_v,\ \mathit{Init} \wedge \Box[\mu \vee (x' = x)]_x \Rightarrow L)$ *is* μ*-machine realizable.*

(c) $\exists x : (\mathit{Init} \wedge \Box[\mu \vee (x' = x)]_x \wedge \Box[\mathcal{N} \vee \neg\mu]_v)$ *is a safety property.*

then $(\exists x : (\mathit{Init} \wedge \Box[\mu \vee (x' = x)]_x \wedge \Box[\mathcal{N} \vee \neg\mu]_v),\ L)$ *is* μ*-machine realizable.*

Proposition 6 *If* (Π, L_1) *is* μ*-machine realizable and* $\Pi \wedge L_1$ *implies* L_2*, then* (Π, L_2) *is* μ*-machine realizable.*

For properties Φ and Π, we define $\Phi \mathrel{-\!\!\triangleright} \Pi$ to be the property satisfied by a behavior σ iff σ satisfies $\Phi \Rightarrow \Pi$ and every finite prefix of σ satisfies $\mathcal{C}(\Phi) \Rightarrow \mathcal{C}(\Pi)$.[7] If Φ and Π are safety properties, then $\Phi \mathrel{-\!\!\triangleright} \Pi$ is the safety property asserting that Π remains true at least as long as Φ does. The property $\Phi \mathrel{-\!\!\triangleright} \Pi$ is sometimes written Π *while* Φ; it is expressible in TLA, for any TLA formulas Φ and Π.

The operator $\mathrel{-\!\!\triangleright}$ is the implication operator for an intuitionistic logic of safety properties [3]. 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.

$$\begin{aligned} \Phi \mathrel{-\!\!\triangleright} (\Pi \mathrel{-\!\!\triangleright} \Psi) &\equiv (\Phi \wedge \Pi) \mathrel{-\!\!\triangleright} \Psi \\ (\Phi \mathrel{-\!\!\triangleright} \Psi) \wedge (\Pi \mathrel{-\!\!\triangleright} \Psi) &\equiv (\Phi \vee \Pi) \mathrel{-\!\!\triangleright} \Psi \end{aligned} \tag{10}$$

Valid formulas can also be obtained by certain partial replacements of $\Rightarrow$ by $\mathrel{-\!\!\triangleright}$ in valid formulas. For example, the following equivalence is valid if P is a safety property.

$$\begin{aligned} &(E \Rightarrow (P \mathrel{-\!\!\triangleright} M_1)) \Rightarrow (E \Rightarrow (P \mathrel{-\!\!\triangleright} M_2)) \\ &\quad \equiv (E \wedge P \wedge M_1) \Rightarrow (E \wedge P \wedge M_2) \end{aligned} \tag{11}$$

A precise relation between $\mathrel{-\!\!\triangleright}$ and $\Rightarrow$ is established by:

Proposition 7 ([1], Proposition 8) *If* μ *is an action that does not permit stuttering steps,* Φ *and* Π *are safety properties,* Φ *does not constrain* μ*, and* Π *constrains at most* μ*, then* $\mathcal{R}_\mu(\Phi \Rightarrow \Pi)$ *equals* $\Phi \mathrel{-\!\!\triangleright} \Pi$*.*

Substituting true for Φ in Proposition 7 proves that a safety property is μ-receptive if it constrains at most μ.

The following variant of Proposition 6 is useful. Note that if (true, L) is μ-machine realizable, then L is a liveness property.

[7] This definition is slightly different from the one in [1]; but the two definitions agree when Φ and Π are safety properties.

Proposition 8 *If μ is an action that does not allow stuttering steps, Φ and Π are safety properties, $(\Phi \dashv\!\triangleright \Pi, L_1)$ and (true, L_2) are μ-machine realizable, and $\Phi \wedge \Pi \wedge L_1$ implies L_2, then $(\Phi \dashv\!\triangleright \Pi, L_2)$ is μ-machine realizable.*

By using Propositions 4 and 8 instead of Propositions 1 and 3, the proof of Theorem 1 generalizes to the proof of the following result. If Π has the form $Init \wedge \Box[\mathcal{N}]_v$, we write Π_0 to denote $Init$ and $\Pi_\Box$ to denote $\Box[\mathcal{N}]_v$.

Theorem 2 *With the notation and hypotheses of Theorem 1, if E and M are safety properties such that $\Pi = E \wedge M$, μ is an action that does not allow stuttering steps, and*

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

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

$$
\begin{aligned}
E^t &\triangleq E \wedge (RT_v)_0 \wedge \forall j \in J : MaxTime(T_j)_0 \\
M^t &\triangleq M \wedge (RT_v)_\Box \wedge \forall i \in I : MinTime(t_i, \mathcal{A}_i, v) \wedge \forall j \in J : MaxTime(T_j)_\Box
\end{aligned}
$$

Observe how the initial predicates of RT_v and $MaxTime(T_j)$ appear in the environment assumption E^t. (Formula $MinTime(t_i, \mathcal{A}_i, v)$ has no initial predicate.) If P is a predicate, then $P \dashv\!\triangleright \Pi$ is equivalent to $P \Rightarrow \Pi$, and $(P \wedge \Pi, L)$ is machine closed if $(P \Rightarrow \Pi, L)$ is. Since machine realizability implies machine closure, Theorem 1 can be obtained from Theorem 2 by letting E and μ equal true and M equal Π.

4.2 Open Systems as Implications

An open system specification is one in which the system guarantees a property M only if the environment satisfies an assumption E. The set of allowed behaviors is described by the formula $E \Rightarrow M$. The specification also includes an action μ that defines which steps are under the control of (or blamed on) the system. For a reasonable specification, $\mathcal{C}(E)$ must not constrain μ, and $\mathcal{C}(M)$ must constrain at most μ.[8] The following result shows that, under reasonable hypotheses, E can be taken to be a safety property.

Proposition 9 ([1], Theorem 1) *If I is a predicate, E_S and M_S are safety properties, and $(E_S, E_S \wedge E_L)$ is $\neg\mu$-machine realizable, then*

$$\mathcal{R}_\mu(I \wedge E_S \wedge E_L \Rightarrow M_S \wedge M_L) = \mathcal{R}_\mu(I \wedge E_S \Rightarrow M_S \wedge (E_L \Rightarrow M_L))$$

An open system specification can then be written as $E \Rightarrow M$, with

$$
\begin{aligned}
E &\triangleq Init \wedge \exists\, e : (Init_e \wedge \Box[(\mu \wedge (e' = e)) \vee \mathcal{N}_E]_{(e,v)}) \\
M &\triangleq \exists\, m : (Init_m \wedge \Box[(\neg\mu \wedge (m' = m)) \vee \mathcal{N}_M]_{(m,v)} \wedge (L_E \Rightarrow L_M))
\end{aligned}
$$

[8] The slight asymmetry in these conditions results from the arbitrary choice that initial conditions appear in E and not in M.

where e and m denote the internal variables of the environment and module, which are each disjoint from all variables appearing in the scope of the other's "$\exists$"; L_E and L_M are conjunctions of suitable fairness properties; $\exists e : Init_e$ and $\exists m : Init_m$ are identically true; the system's next-state action $\mathcal{N}_M$ implies μ, and the environment's next-state action $\mathcal{N}_E$ implies $\neg\mu$. Under these assumptions, it can be shown that $\mathcal{C}(E)$ does not constrain μ, and $\mathcal{C}(M)$ constrains at most μ. It is easy to show that $E \wedge M$, the TLA formula describing the closed system formed by the open system and its environment, equals

$$\exists\, e, m : (Init \wedge Init_E \wedge Init_M \wedge \Box[\mathcal{N}_E \vee \mathcal{N}_M]_{\langle e,m,v\rangle} \wedge (L_E \Rightarrow L_M)) \qquad (12)$$

Thus, $E \wedge M$ has precisely the form we expect for a closed system comprising two components with next-state actions $\mathcal{N}_E$ and $\mathcal{N}_M$.

Implementation means implication. A system with guarantee M implements a system with guarantee $\widehat{M}$, under environment assumption E, iff $E \Rightarrow M$ implies $E \Rightarrow \widehat{M}$. But this is logically equivalent to $E \wedge M$ implying $E \wedge \widehat{M}$. In other words, proving that one open system implements another is equivalent to proving the implementation relation for the corresponding closed systems. Hence, implementation for open systems reduces to implementation for closed systems.[9]

4.3 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 based on the following result, which is a slight generalization of Theorem 2 of [1].

Theorem 3 *If P, E, E_1, and E_2 are safety properties, M_1 and M_2 are arbitrary properties, and μ_1 and μ_2 are actions such that*

1. *(a) E_1 does not constrain μ_1, (b) E_2 does not constrain μ_2, and (c) E does not constrain $\mu_1 \vee \mu_2$,*
2. *$\mathcal{C}(M_1)$ constrains at most μ_1, and $\mathcal{C}(M_2)$ constrains at most μ_2,*
3. *$\mu_1 \vee \mu_2$ does not allow stuttering steps,*

then the following proof rule is valid.

$$\frac{P \wedge E \wedge \mathcal{C}(M_1) \wedge \mathcal{C}(M_2) \;\Rightarrow\; E_1 \wedge E_2}{\mathcal{R}_{\mu_1}(E_1 \Rightarrow M_1) \wedge \mathcal{R}_{\mu_2}(E_2 \Rightarrow M_2) \;\Rightarrow\; (E \Rightarrow (P \mathrel{-\!\!\triangleright} M_1) \wedge (P \mathrel{-\!\!\triangleright} M_2))}$$

This theorem differs from Theorem 2 of [1] in two significant ways:

- The assumption $\mu_1 \wedge \mu_2 = \emptyset$ is missing, and the conclusion of the proof rule has been weakened by removing the $\mathcal{R}_{\mu_1 \vee \mu_2}$. An examination of the proof of the theorem in [1] reveals that the assumption is not needed for this weaker conclusion.
- The hypothesis has been weakened to include the conjunct P and the conclusion weakened by adding the "$P \mathrel{-\!\!\triangleright}$"s. The original theorem is obtained by letting P be true. A simple modification to the argument in [1] proves the generalization.

[9] A similar argument shows that we can replace $L_E \Rightarrow L_M$ by $L_E \wedge L_M$ in (12) when proving that $E \wedge M$ implements $E \wedge \widehat{M}$.

5 Real-Time Open Systems

5.1 The Paradox Revisited

We now consider the paradoxical example of the introduction, illustrated in Figure 1. For simplicity, let the 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

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

and defining the untimed version of specification S_1 to be

$$\Box[\nu_1 \vee \mathcal{M}_y]_{(x,y)} \Rightarrow \Box[\neg\nu_1 \vee \mathcal{M}_x]_{(x,y)} \tag{13}$$

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. Remembering that initial conditions must appear in the environment assumption, we define

$$\begin{array}{lcl} \mathcal{N}_x & \triangleq & \mathcal{M}_x \wedge (now' = now) \\ \mathcal{N}_y & \triangleq & \mathcal{M}_y \wedge (now' = now) \\ T_x & \triangleq & \textbf{if } x \neq 0 \textbf{ then } 12 \textbf{ else } \infty \\ T_y & \triangleq & \textbf{if } y \neq 0 \textbf{ then } 12 \textbf{ else } \infty \end{array} \qquad \begin{array}{lcl} MT_x & \triangleq & MaxTime(T_x) \\ MT_y & \triangleq & MaxTime(T_y) \\ \mu_1 & \triangleq & \nu_1 \vee (now' \neq now) \\ & & \end{array}$$

$$\begin{array}{lcl} E_1 & \triangleq & (now = 0) \wedge (MT_x)_0 \wedge \Box[\mu_1 \vee \mathcal{N}_y]_{(x,y,now)} \\ M_1 & \triangleq & \Box[\neg\mu_1 \vee \mathcal{N}_x]_{(x,y,now)} \wedge (RT_{(x,y)})_\Box \wedge (MT_x)_\Box \end{array}$$

Adding timing constraints to (13) the same way we did for closed systems then leads to the following timed version of specification S_1.

$$E_1 \wedge MT_y \Rightarrow M_1 \tag{14}$$

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. This is easily done by rewriting (14) in the equivalent form

$$E_1 \Rightarrow (MT_y \Rightarrow M_1)$$

so the system guarantee becomes $MT_y \Rightarrow M_1$.[10] However, this guarantee is not a safety property. To make it one, we must replace $\Rightarrow$ by $\mathrel{-\!\!\triangleright}$, obtaining

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

[10]Because MT_y appears on the left of an implication, there is no need to put its initial condition in the environment assumption.

The specification S_2 of the second component in Figure 1 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.

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

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

where

$$E \;\triangleq\; (now = 0) \;\wedge\; (MT_x)_0 \;\wedge\; (MT_y)_0 \;\wedge\; \Box[\mu_1 \vee \mu_2]_{(x,y,now)}$$

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

$$\begin{aligned} \mathcal{R}_{\mu_1}(S_1) \wedge \mathcal{R}_{\mu_2}(S_2) \;\;\Rightarrow\;\; (E \;\Rightarrow\; &\wedge (MT_x \vee MT_y) \mathrel{-\!\!\triangleright} (MT_y \mathrel{-\!\!\triangleright} M_1) \\ &\wedge (MT_x \vee MT_y) \mathrel{-\!\!\triangleright} (MT_x \mathrel{-\!\!\triangleright} M_2)\,) \end{aligned}$$

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

$$\mathcal{R}_{\mu_1}(S_1) \;\wedge\; \mathcal{R}_{\mu_2}(S_2) \;\;\Rightarrow\;\; (E \;\Rightarrow\; (MT_y \mathrel{-\!\!\triangleright} M_1) \;\wedge\; (MT_x \mathrel{-\!\!\triangleright} M_2)) \tag{15}$$

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 $E_2 \Rightarrow M_2$. 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

$$\mathcal{R}_{\mu_1}(S_1) \;\wedge\; \mathcal{R}_{\mu_2}(S_3) \;\;\Rightarrow\;\; (E \;\Rightarrow\; (MT_x \mathrel{-\!\!\triangleright} M_1) \;\wedge\; M_2)$$

Since M_2 implies MT_x, this simplifies to

$$\mathcal{R}_{\mu_1}(S_1) \;\wedge\; \mathcal{R}_{\mu_2}(S_3) \;\;\Rightarrow\;\; (E \;\Rightarrow\; M_1 \;\wedge\; M_2)$$

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

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

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. By (11), proving that one specification in this form implements another reduces to the proof for the corresponding closed systems.

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* can't 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. This condition cannot be satisfied if S contains unrealizable behaviors. By Proposition 7, we can eliminate unrealizable behaviors by changing "$\Rightarrow$" to "$\rightarrow\!\!\!\!\triangleright$" in (16). Using (10), we then see that the appropriate definition of nonZenoness for an open system specification of the form (16), with M a safety property, is that $((E \wedge P) \rightarrow\!\!\!\!\triangleright M,\ NZ)$ be μ-machine realizable. This condition is proved using Theorem 2. To apply the theorem, one must show that M constrains at most μ. Any property of the form $\Box[\mathcal{N} \vee \mu]_v$ constrains at most μ. To prove that a formula with internal variables (existential quantification) constrains at most μ, one applies Proposition 5 with **true** substituted for L, since M constrains at most μ iff $(M, \textbf{true})$ is μ-machine realizable.

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. 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 1-1/2 pages of figures in [11].

All the fundamental concepts described in Sections 2 and 4, including machine closure, machine realizability, and the $\rightarrow\!\!\!\!\triangleright$ operator, have appeared before [1, 2]. However, they are expressed here for the first time in terms of TLA. These concepts are subtle, but they are important for understanding any concurrent system; they were not invented for real-time systems.

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. The proofs for our examples, propositions, and theorems occupy some sixty 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 5, 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 [11], 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 are a fairly recent development, and we have little practical experience with them. The simple idea of putting the environment's timing assumptions to the left of a $\mathrel{-\!\!\triangleright}$ in the system's guarantee came only after numerous failed efforts. We still have much to learn before reasoning about open systems becomes routine. However, the basic intellectual tools we needed to handle real-time open systems were all in place, and we have confidence in our basic approach to open-system verification.

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 only published work we know of that uses this idea is by Marzullo, Schneider, and Budhiraja [12].

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 1–41. Springer-Verlag, May/June 1989.

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

[3] Martín Abadi and Gordon Plotkin. A logical view of composition and refinement. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 323–332, January 1991.

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

[5] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.

[6] Arthur Bernstein and Paul K. Harter, Jr. Proving real time properties of programs with temporal logic. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 1–11, New York, 1981. ACM. *Operating Systems Review 15*, 5.

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

[8] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* PhD thesis, Carnegie Mellon University, February 1988.

[9] Leslie Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, 2(3):175–206, December 1982.

[10] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[11] Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, 1991. To appear.

[12] Keith Marzullo, Fred B. Schneider, and Navin Budhiraja. Derivation of sequential, real-time process-control programs. In Andreé M. van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing: Formal Specifications and Methods*, chapter 2, pages 39–54. Kluwer Academic Publishers, Boston, Dordrecht, and London, 1991.

[13] Peter G. Neumann and Leslie Lamport. Highly dependable distributed systems. Technical report, SRI International, June 1983. Contract Number DAEA18-81-G-0062, SRI Project 4180.

[14] Fred B. Schneider, Bard Bloom, and Keith Marzullo. Putting time into proof outlines. This volume.