Chapter 9

Real Time

With a liveness property, we can specify that a system must eventually respond to a request. We cannot specify that it must respond within the next 100 years. To specify timely response, we must use a real-time property.

A system that does not respond within our lifetime isn't very useful, so we might expect real-time specifications to be common. They aren't. Formal specifications are most often used to describe what a system does rather than how long it takes to do it. However, you may someday want to specify real-time properties of a system. This chapter tells you how.

9.1 The Hour Clock Revisited

Let's return to our specification of the simple hour clock in Chapter 2, which asserts that the variable *hr* cycles through the values 1 through 12. We now add the requirement that the clock keep correct time. For centuries, scientists have represented the real-time behavior of a system by introducing a variable, traditionally t, whose value is a real number that represents time. A state in which $t = -17.51$ represents a state of the system at time -17.51, perhaps measured in seconds elapsed since 00:00 UT on 1 January 2000. In TLA$^+$ specifications, I prefer to use the variable *now* rather than t. For linguistic convenience, I will usually assume that the unit of time is the second, though we could just as well choose any other unit.

Remember that a state is an assignment of values to all variables.

Unlike sciences such as physics and chemistry, computer science studies systems whose behavior can be described by a sequence of discrete states, rather than by states that vary continuously with time. We consider the hour clock's display to change directly from reading 12 to reading 1, and ignore the continuum of intermediate states that occur in the physical display. This means that we pretend that the change is instantaneous (happens in 0 seconds). So, a real-time specification of the clock might allow the step

$$\begin{bmatrix} hr & = & 12 \\ now & = & \sqrt{2.47} \end{bmatrix} \rightarrow \begin{bmatrix} hr & = & 1 \\ now & = & \sqrt{2.47} \end{bmatrix}$$

The value of *now* advances between changes to *hr*. If we wanted to specify how long it takes the display to change from 12 to 1, we would have to introduce an intermediate state that represents a changing display—perhaps by letting *hr* assume some intermediate value such as 12.5, or by adding a Boolean-valued variable *chg* whose value indicates whether the display is changing. We won't do this, but will be content to specify an hour clock in which we consider the display to change instantaneously.

The value of *now* changes between changes to *hr*. Just as we represent a continuously varying clock display by a variable whose value changes in discrete steps, we let the value of *now* change in discrete steps. A behavior in which *now* increases in femtosecond increments would be an accurate enough description of continuously changing time for our specification of the hour clock. In fact, there's no need to choose any particular granularity of time; we can let *now* advance by arbitrary amounts between clock ticks. (Since the value of *hr* is unchanged by steps that change *now*, the requirement that the clock keep correct time will rule out behaviors in which *now* changes by too much in a single step.)

What real-time condition should our hour clock satisfy? We might require that it always display the time correctly to within ρ seconds, for some real number ρ. However, this is not typical of the real-time requirements that arise in actual systems. Instead, we require that the clock tick approximately once per hour. More precisely, we require that the interval between ticks be one hour plus or minus ρ seconds, for some positive number ρ. Of course, this requirement allows the time displayed by the clock eventually to drift away from the actual time. But that's what real clocks do if they are not reset.

We could start our specification of the real-time clock from scratch. However, we still want the hour clock to satisfy the specification *HC* of module *HourClock* (Figure 2.1 on page 20). We just want to add an additional real-time requirement. So, we will write the specification as the conjunction of *HC* and a formula requiring that the clock tick every hour, plus or minus ρ seconds. This requirement is the conjunction of two separate conditions: that the clock tick at most once every $3600 - \rho$ seconds, and at least once every $3600 + \rho$ seconds.

To specify these requirements, we introduce a variable that records how much time has elapsed since the last clock tick. Let's call it t for *timer*. The value of t is set to 0 by a step that represents a clock tick—namely, by an *HCnxt* step. Any step that represents the passing of s seconds should advance t by s. A step represents the passing of time iff it changes *now*, and such a step represents the passage of $now' - now$ seconds. So, the change to the timer t is described by the action

$$TNext \triangleq t' = \text{IF } HCnxt \text{ THEN } 0 \text{ ELSE } t + (now' - now)$$

We let t initially equal 0, so we consider the initial state to be one in which the clock has just ticked. The specification of how t changes is then a formula asserting that t initially equals 0, and that every step is a *TNext* step or else leaves unchanged all relevant variables—namely, t, hr, and now. This formula is

$$Timer \;\triangleq\; (t = 0) \land \Box[TNext]_{\langle t,\, hr,\, now \rangle}$$

The requirement that the clock tick at least once every $3600 + \rho$ seconds means that it's always the case that at most $3600 + \rho$ seconds have elapsed since the last *HCnxt* step. Since t always equals the elapsed time since the last *HCnxt* step, this requirement is expressed by the formula

$$MaxTime \;\triangleq\; \Box(t \leq 3600 + \rho)$$

(Since we can't measure time with perfect accuracy, it doesn't matter whether we use $<$ or $\leq$ in this formula. When we generalize from this example, it is a bit more convenient to use $\leq$.)

The requirement that the clock tick at most once every $3600 - \rho$ seconds means that, whenever an *HCnxt* step occurs, at least $3600 - \rho$ seconds have elapsed since the previous *HCnxt* step. This suggests the condition

In the generalization, $\geq$ will be more convenient than $>$.

$$(9.1) \quad \Box(HCnxt \Rightarrow (t \geq 3600 - \rho))$$

However, (9.1) isn't a legal TLA formula because $HCnxt \Rightarrow \ldots$ is an action (a formula containing primes), and a TLA formula asserting that an action is always true must have the form $\Box[A]_v$. We don't care about steps that leave hr unchanged, so we can replace (9.1) by the TLA formula

$$MinTime \;\triangleq\; \Box[HCnxt \Rightarrow (t \geq 3600 - \rho)]_{hr}$$

The desired real-time constraint on the clock is expressed by the conjunction of these three formulas:

$$HCTime \;\triangleq\; Timer \land MaxTime \land MinTime$$

Formula *HCTime* contains the variable t, and the specification of the real-time clock should describe only the changes to hr (the clock display) and now (the time). So, we have to hide t. Hiding is expressed in TLA$^+$ by the temporal existential quantifier $\boldsymbol{\exists}$, introduced in Section 4.3 (page 41). However, as explained in that section, we can't simply write $\boldsymbol{\exists}\, t : HCTime$. We must define *HCTime* in a module that declares t, and then use a parametrized instantiation of that module. This is done in Figure 9.1 on page 121. Instead of defining *HCTime* in a completely separate module, I have defined it in a submodule named *Inner* of the module *RealTimeHourClock* containing the specification of the real-time hour clock. Note that all the symbols declared and defined in the main module up to that point can be used in the submodule. Submodule *Inner* is instantiated in the main module with the statement

$$I(t) \triangleq \text{INSTANCE } Inner$$

The t in *HCTime* can then be hidden by writing $\exists\, t : I(t)!HCTime$.

The formula $HC \land (\exists\, t : I(t)!HCTime)$ describes the possible changes to the value of *hr*, and relates those changes to the value of *now*. But it says very little about how the value of *now* can change. For example, it allows the following behavior:

$$\begin{bmatrix} hr & = & 11 \\ now & = & 23.5 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = & 11 \\ now & = & 23.4 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = & 11 \\ now & = & 23.5 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = & 11 \\ now & = & 23.4 \end{bmatrix} \rightarrow \cdots$$

Because time can't go backwards, such a behavior doesn't represent a physical possibility. Everyone knows that time only increases, so there's no need to forbid this behavior if the only purpose of our specification is to describe the hour clock. However, a specification should also allow us to reason about a system. If the clock ticks approximately once per hour, then it can't stop. However, as the behavior above shows, the formula $HC \land (\exists\, t : I(t)!HCTime)$ by itself allows the clock to stop. To infer that it can't, we also need to state how *now* changes.

We define a formula *RTnow* that specifies the possible changes to *now*. This formula does not specify the granularity of the changes to *now*; it allows a step to advance *now* by a microsecond or by a century. However, we have decided that a step that changes *hr* should leave *now* unchanged, which implies that a step that changes *now* should leave *hr* unchanged. Therefore, steps that change *now* are described by the following action, where *Real* is the set of all real numbers:

$$\begin{aligned} NowNext \;\triangleq\; & \land\; now' \in \{r \in Real : r > now\} \\ & \land\; \text{UNCHANGED } hr \end{aligned}$$

now' can equal any real number $> now$.

Formula *RTnow* should also allow steps that leave *now* unchanged. The initial value of *now* is an arbitrary real number (we can start the system at any time), so the safety part of *RTnow* is

$$(now \in Real) \land \Box[NowNext]_{now}$$

The liveness condition we want is that *now* should increase without bound. Simple weak fairness of the *NowNext* action isn't good enough, because it allows "Zeno" behaviors such as

Weak fairness is discussed in Chapter 8.

$$[now = .9] \rightarrow [now = .99] \rightarrow [now = .999] \rightarrow [now = .9999] \rightarrow \cdots$$

in which the value of *now* remains bounded. Weak fairness of the action $NowNext \land (now' > r)$ implies that eventually a *NowNext* step will occur in which the new value of *now* is greater than r. (This action is always enabled, so weak fairness implies that infinitely many such actions must occur.) Asserting

MODULE *RealTimeHourClock*

EXTENDS *Reals*, *HourClock*

VARIABLE *now* — The current time, measured in seconds.

CONSTANT *Rho* — A positive real number.

ASSUME $(Rho \in Real) \wedge (Rho > 0)$

MODULE *Inner*

VARIABLE t

$TNext \triangleq t' = \text{IF } HCnxt \text{ THEN } 0 \text{ ELSE } t + (now' - now)$

$Timer \triangleq (t = 0) \wedge \Box[TNext]_{\langle t,\, hr,\, now \rangle}$ — t is the elapsed time since the last *HCnxt* step.

$MaxTime \triangleq \Box(t \leq 3600 + Rho)$ — t is always at most $3600 + Rho$.

$MinTime \triangleq \Box[HCnxt \Rightarrow t \geq 3600 - Rho]_{hr}$ — An *HCnxt* step can occur only if $t \geq 3600 - Rho$.

$HCTime \triangleq Timer \wedge MaxTime \wedge MinTime$

$I(t) \triangleq$ INSTANCE *Inner*

$NowNext \triangleq \wedge\ now' \in \{r \in Real : r > now\}$
$\qquad\qquad\quad \wedge$ UNCHANGED hr

A *NowNext* step can advance *now* by any amount while leaving *hr* unchanged.

$RTnow \triangleq \wedge\ now \in Real$ — *RTnow* specifies how time may change.
$\qquad\quad \wedge\ \Box[NowNext]_{now}$
$\qquad\quad \wedge\ \forall r \in Real : \text{WF}_{now}(NowNext \wedge (now' > r))$

$RTHC \triangleq HC \wedge RTnow \wedge (\exists\, t : I(t)!HCTime)$ — The complete specification.

Figure 9.1: The real-time specification of an hour clock that ticks every hour, plus or minus *Rho* seconds.

this for all real numbers r implies that *now* grows without bound, so we take as the fairness condition[1]

$$\forall r \in Real : \text{WF}_{now}(NowNext \wedge (now' > r))$$

The complete specification *RTHC* of the real-time hour clock, with the definition of formula *RTnow*, is in the *RealTimeHourClock* module of Figure 9.1 on this page. That module extends the standard *Reals* module, which defines the set *Real* of real numbers.

[1]An equivalent condition is $\forall r \in Real : \Diamond(now > r)$, but I like to express fairness with WF and SF formulas.

9.2 Real-Time Specifications in General

In Section 8.4 (page 96), we saw that the appropriate generalization of the liveness requirement that the hour clock tick infinitely often is weak fairness of the clock-tick action. There is a similar generalization for real-time specifications. Weak fairness of an action A asserts that if A is continuously enabled, then an A step must eventually occur. The real-time analog is that if A is continuously enabled for ϵ seconds, then an A step must occur. Since an *HCnxt* action is always enabled, the requirement that the clock tick at least once every $3600 + \rho$ seconds can be expressed in this way by letting A be *HCnxt* and ϵ be $3600 + \rho$.

The requirement that an *HCnxt* action occur at most once every $3600 - \rho$ seconds can be similarly generalized to the condition that an action A must be continuously enabled for at least δ seconds before an A step can occur.

The first condition, the upper bound ϵ on how long A can be enabled without an A step occurring, is vacuously satisfied if ϵ equals *Infinity*—a value defined in the *Reals* module to be greater than any real number. The second condition, the lower bound δ on how long A must be enabled before an A step can occur, is vacuously satisfied if δ equals 0. So, nothing is lost by combining both of these conditions into a single formula containing δ and ϵ as parameters. I now define such a formula, which I call a *real-time bound condition*.

The weak fairness formula $\mathrm{WF}_v(A)$ actually asserts weak fairness of the action $\langle A \rangle_v$, which equals $A \land (v' \neq v)$. The subscript v is needed to rule out stuttering steps. Since the truth of a meaningful formula can't depend on whether or not there are stuttering steps, it makes no sense to say that an A step did or did not occur if that step could be a stuttering step. For this reason, the corresponding real-time condition must also be a condition on an action $\langle A \rangle_v$, not on an arbitrary action A. In most cases of interest, v is the tuple of all variables that occur in A. I therefore define the real-time bound formula $RTBound(A, v, \delta, \epsilon)$ to assert that

- An $\langle A \rangle_v$ step cannot occur until $\langle A \rangle_v$ has been continuously enabled for at least δ time units since the last $\langle A \rangle_v$ step—or since the beginning of the behavior.
- $\langle A \rangle_v$ can be continuously enabled for at most ϵ time units before an $\langle A \rangle_v$ step occurs.

$RTBound(A, v, \delta, \epsilon)$ generalizes the formula $\exists\, t : I(t)!HCTime$ of the real-time hour-clock specification, and it can be defined in the same way, using a submodule. However, the definition can be structured a little more compactly as

For the TLA$^+$ specification, I have replaced δ and ϵ by D and E.

$$\begin{array}{lll} RTBound(A, v, D, E) \triangleq & \text{LET} & Timer(t) \triangleq \ldots \\ & & \ldots \\ & \text{IN} & \exists\, t : Timer(t) \land \ldots \end{array}$$

We first define $Timer(t)$ to be a temporal formula asserting that t always equals the length of time that $\langle A \rangle_v$ has been continuously enabled since the last $\langle A \rangle_v$ step. The value of t should be set to 0 by an $\langle A \rangle_v$ step or a step that disables $\langle A \rangle_v$. A step that advances *now* should increment t by $now' - now$ iff $\langle A \rangle_v$ is enabled. Changes to t are therefore described by the action

$$\begin{array}{lcl} TNext(t) & \triangleq & t' = \textsc{if}\ \langle A \rangle_v \lor \neg(\textsc{enabled}\ \langle A \rangle_v)' \\ & & \qquad\ \textsc{then}\ 0 \\ & & \qquad\ \textsc{else}\ \ t + (now' - now) \end{array}$$

We are interested in the meaning of $Timer(t)$ only when v is a tuple whose components include all the variables that may appear in A. In this case, a step that leaves v unchanged cannot enable or disable $\langle A \rangle_v$. So, the formula $Timer(t)$ should allow steps that leave t, v, and *now* unchanged. Letting the initial value of t be 0, we define

$$Timer(t) \ \triangleq\ (t = 0) \land \Box[TNext(t)]_{\langle t,\, v,\, now \rangle}$$

Formulas *MaxTime* and *MinTime* of the real-time hour clock's specification have the obvious generalizations:

- $MaxTime(t)$ asserts that t is always less than or equal to E:

  $$MaxTime(t) \ \triangleq\ \Box(t \leq E)$$

- $MinTime(t)$ asserts that an $\langle A \rangle_v$ step can occur only if $t \geq D$:

  $$MinTime(t) \ \triangleq\ \Box[A \Rightarrow (t \geq D)]_v$$

  (An equally plausible definition of $MinTime(t)$ is $\Box[\langle A \rangle_v \Rightarrow (t \geq D)]_v$, but the two are, in fact, equivalent.)

We then define $RTBound(A, v, D, E)$ to equal

$$\exists\, t : Timer(t) \land MaxTime(t) \land MinTime(t)$$

We must also generalize formula *RTnow* of the real-time hour clock's specification. That formula describes how *now* changes, and it asserts that *hr* remains unchanged when *now* changes. The generalization is the formula $RTnow(v)$, which replaces *hr* with an arbitrary state function v that will usually be the tuple of all variables, other than *now*, appearing in the specification. Using these definitions, the specification *RTHC* of the real-time hour clock can be written

$$HC \land RTnow(hr) \land RTBound(HCnxt, hr, 3600 - Rho, 3600 + Rho)$$

The *RealTime* module, with its definitions of *RTBound* and *RTnow*, appears in Figure 9.2 on page 125.

Strong fairness strengthens weak fairness by requiring an A step to occur not just if action A is continuously enabled, but if it is repeatedly enabled. Being repeatedly enabled includes the possibility that it is also repeatedly disabled. We can similarly strengthen our real-time bound conditions by defining a stronger formula $SRTBound(A, v, \delta, \epsilon)$ to assert that

- An $\langle A \rangle_v$ step cannot occur until $\langle A \rangle_v$ has been enabled for a total of at least δ time units since the last $\langle A \rangle_v$ step—or since the beginning of the behavior.
- $\langle A \rangle_v$ can be enabled for a total of at most ϵ time units before an $\langle A \rangle_v$ step occurs.

If $\epsilon < Infinity$, then $RTBound(A, v, \delta, \epsilon)$ implies that an $\langle A \rangle_v$ step must occur if $\langle A \rangle_v$ is continuously enabled for ϵ seconds. Hence, if $\langle A \rangle_v$ is ever enabled forever, infinitely many $\langle A \rangle_v$ steps must occur. Thus, $RTBound(A, v, \delta, \epsilon)$ implies weak fairness of A. More precisely, $RTBound(A, v, \delta, \epsilon)$ and $RTnow(v)$ together imply $\mathrm{WF}_v(A)$. However, $SRTBound(A, v, \delta, \epsilon)$ does not similarly imply strong fairness of A. It allows behaviors in which $\langle A \rangle_v$ is enabled infinitely often but never executed—for example, A can be enabled for $\epsilon/2$ seconds, then for $\epsilon/4$ seconds, then for $\epsilon/8$ seconds, and so on. For this reason, *SRTBound* does not seem to be of much practical use, so I won't bother defining it formally.

9.3 A Real-Time Caching Memory

Let's now use the *RealTime* module to write a real-time versions of the linearizable memory specification of Section 5.3 (page 51) and the write-through cache specification of Section 5.6 (page 54). We obtain the real-time memory specification by strengthening the specification in module *Memory* (Figure 5.3 on page 53) to require that the memory responds to a processor's requests within *Rho* seconds. The complete memory specification *Spec* of module *Memory* was obtained by hiding the variables *mem*, *ctl*, and *buf* in the internal specification *ISpec* of module *InternalMemory*. It's generally easier to add a real-time constraint to an internal specification, where the constraints can mention the internal (hidden) variables. So, we first add the timing constraint to *ISpec* and then hide the internal variables.

To specify that the system must respond to a processor request within *Rho* seconds, we add an upper-bound timing constraint for an action that becomes enabled when a request is issued, and that becomes disabled (possibly by being executed) only when the processor responds to the request. In specification *ISpec*, responding to a request requires two actions—$Do(p)$ to perform the operation internally, and $Rsp(p)$ to issue the response. Neither of these actions is the one we want; we have to define a new action for the purpose. There is a pending request for processor p iff $ctl[p]$ equals "rdy". So, we assert that the

MODULE *RealTime*

This module declares the variable *now*, which represents real time, and defines operators for writing real-time specifications. Real-time constraints are added to a specification by conjoining it with $RTnow(v)$ and formulas of the form $RTBound(A, v, \delta, \epsilon)$ for actions A, where v is the tuple of all specification variables and $0 \leq \delta \leq \epsilon \leq Infinity$.

EXTENDS *Reals*

VARIABLE *now* — The value of *now* is a real number that represents the current time, in unspecified units.

$RTBound(A, v, \delta, \epsilon)$ asserts that an $\langle A \rangle_v$ step can occur only after $\langle A \rangle_v$ has been continuously enabled for δ time units since the last $\langle A \rangle_v$ step (or the beginning of the behavior), and it must occur before $\langle A \rangle_v$ has been continuously enabled for more than ϵ time units since the last $\langle A \rangle_v$ step (or the beginning of the behavior).

```
RTBound(A, v, D, E) ≜
    LET TNext(t) ≜ t' = IF ⟨A⟩_v ∨ ¬(ENABLED ⟨A⟩_v)'
                         THEN 0
                         ELSE t + (now' − now)
        Timer(t) ≜ (t = 0) ∧ □[TNext(t)]_⟨t, v, now⟩
        MaxTime(t) ≜ □(t ≤ E)
        MinTime(t) ≜ □[A ⇒ (t ≥ D)]_v
    IN  ∃ t : Timer(t) ∧ MaxTime(t) ∧ MinTime(t)
```

$Timer(t)$ asserts that t is the length of time $\langle A \rangle_v$ has been continuously enabled without an $\langle A \rangle_v$ step occurring.

$MaxTime(t)$: Asserts that t is always $\leq E$.

$MinTime(t)$: Asserts that an $\langle A \rangle_v$ step can occur only if $t \geq D$.

$RTnow(v)$ asserts that *now* is a real number that is increased without bound, in arbitrary increments, by steps that leave v unchanged.

```
RTnow(v) ≜ LET NowNext ≜ ∧ now' ∈ {r ∈ Real : r > now}
                         ∧ UNCHANGED v
           IN  ∧ now ∈ Real
               ∧ □[NowNext]_now
               ∧ ∀ r ∈ Real : WF_now(NowNext ∧ (now' > r))
```

Figure 9.2: The *RealTime* module for writing real-time specifications.

following action cannot be enabled for more than *Rho* seconds without being executed:

$$Respond(p) \triangleq (ctl[p] \neq \text{"rdy"}) \wedge (ctl'[p] = \text{"rdy"})$$

The complete specification is formula *RTSpec* of module *RTMemory* in Figure 9.3 on the next page. To permit variables *mem*, *ctl*, and *buf* to be hidden, the *RTMemory* module contains a submodule *Inner* that extends module *InternalMemory*.

Having added a real-time constraint to the specification of a linearizable memory, let's strengthen the specification of the write-through cache so it sat-

```
──────────────────────────── MODULE RTMemory ────────────────────────────
A specification that strengthens the linearizable memory specification of Section 5.3 by requiring that a response
be sent to every processor request within Rho seconds.

EXTENDS MemoryInterface, RealTime

CONSTANT Rho

ASSUME (Rho ∈ Real) ∧ (Rho > 0)
  ──────────────────────────── MODULE Inner ────────────────────────────
  We introduce a submodule so we can hide the variables mem, ctl, and buf.

  EXTENDS InternalMemory

  Respond(p) ≜                                   Respond(p) is enabled when a request is received from p; it is
      (ctl[p] ≠ "rdy") ∧ (ctl′[p] = "rdy")       disabled when a Respond(p) step issues the response.

  RTISpec ≜ ∧ ISpec                                        We assert an upper-bound delay
            ∧ ∀ p ∈ Proc : RTBound(Respond(p), ctl, 0, Rho) of Rho on Respond(p), for all
            ∧ RTnow(⟨memInt, mem, ctl, buf⟩)               processors p.
  ─────────────────────────────────────────────────────────────────────

Inner(mem, ctl, buf) ≜ INSTANCE Inner

RTSpec ≜ ∃ mem, ctl, buf : Inner(mem, ctl, buf)!RTISpec
─────────────────────────────────────────────────────────────────────────
```

Figure 9.3: A real-time version of the linearizable memory specification.

isfies that constraint. The object is not just to add any real-time constraint that does the job—that's easy to do by using the same constraint that we added to the memory specification. We want to write a specification of a real-time algorithm—a specification that tells an implementer how to meet the real-time constraints. This is generally done by placing real-time bounds on the original actions of the untimed specification, not by adding time bounds on a new action, as we did for the memory specification. An upper-bound constraint on the response time should be achieved by enforcing upper-bound constraints on the system's actions.

If we try to achieve a bound on response time by adding real-time bounds to the write-through cache specification's actions, we encounter the following problem. Operations by different processors "compete" with one another to enqueue operations on the finite queue *memQ*. For example, when servicing a write request for processor p, the system must execute a $DoWr(p)$ action to enqueue the operation to the tail of *memQ*. That action is not enabled if *memQ* is full. The $DoWr(p)$ action can be continually disabled by the system performing *DoWr* or *RdMiss* actions for other processors. That's why, to guarantee liveness—that each request eventually receives a response—in Section 8.7 (page 107) we had to assert strong fairness of *DoWr* and *RdMiss* actions. The only way to ensure that a $DoWr(p)$ action is executed within some length of time is to use lower-bound constraints on the actions of other processors to ensure that they cannot perform $DoWr$ or $RdMiss$ actions too frequently. Although such a specification is possible, it is not the kind of approach anyone is likely to take in practice.

The usual method of enforcing real-time bounds on accesses to a shared resource is to schedule the use of the resource by different processors. So, let's modify the write-through cache to add a scheduling discipline to actions that enqueue operations on $memQ$. We use round-robin scheduling, which is probably the easiest one to implement. Suppose processors are numbered from 0 through $N-1$. Round-robin scheduling means that an operation for processor p is the next one to be enqueued after an operation for processor q iff there is not an operation for any of the processors $(q+1) \% N$, $(q+2) \% N$, ... , $(p-1) \% N$ waiting to be put on $memQ$.

To express this formally, we first let the set $Proc$ of processors equal the set $0 \,..\, (N-1)$ of integers. We normally do this by defining $Proc$ to equal $0 \,..\, (N-1)$. However, we want to reuse the parameters and definitions from the write-through cache specification, and that's easiest to do by extending module $WriteThroughCache$. Since $Proc$ is a parameter of that module, we can't define it. We therefore let N be a new constant parameter and let $Proc = 0 \,..\, (N-1)$ be an assumption.[2]

To implement round-robin scheduling, we use a variable $lastP$ that equals the last processor whose operation was enqueued to $memQ$. We define the operator $position$ so that p is the $position(p)^{\text{th}}$ processor after $lastP$ in the round-robin order:

$$position(p) \;\triangleq\; \text{CHOOSE } i \in 1 \,..\, N : p = (lastP + i) \% N$$

(Thus, $position(lastP)$ equals N.) An operation for processor p can be the next to access $memQ$ iff there is no operation for a processor q with $position(q) < position(p)$ ready to access it—that is, iff $canGoNext(p)$ is true, where

$$\begin{aligned} canGoNext(p) \;\triangleq\; \forall\, q \in Proc : (position(q) < position(p)) \Rightarrow& \\ \neg \text{ENABLED } (RdMiss(q) \lor DoWr(q))& \end{aligned}$$

We then define $RTRdMiss(p)$ and $RTDoWr(p)$ to be the same as $RdMiss(p)$ and $DoWr(p)$, respectively, except that they have the additional enabling condition $canGoNext(p)$, and they set $lastP$ to p. The other subactions of the next-state action are the same as before, except that they must also leave $lastP$ unchanged.

For simplicity, we assume a single upper bound of $Epsilon$ on the length of time any of the actions of processor p can remain enabled without being executed—except for the $Evict(p, a)$ action, which we never require to happen. In general, suppose $A_1, \ldots, A_k$ are actions such that (i) no two of them are

[2] We could also instantiate module $WriteThroughCache$ with $0 \,..\, (N-1)$ substituted for $Proc$; but that would require declaring the other parameters of $WriteThroughCache$, including the ones from the $MemoryInterface$ module.

ever simultaneously enabled, and (ii) once any A_i becomes enabled, it must be executed before another A_j can be enabled. In this case, a single *RTBound* constraint on $A_1 \vee \ldots \vee A_k$ is equivalent to separate constraints on all the A_i. We can therefore place a single constraint on the disjunction of all the actions of processor p, except that we can't use the same constraint for both $DoRd(p)$ and $RTRdMiss(p)$ because an $Evict(p, a)$ step could disable $DoRd(p)$ and enable $RTRdMiss(p)$. We therefore use a separate constraint for $RTRdMiss(p)$.

We assume an upper bound of *Delta* on the time *MemQWr* or *MemQRd* can be enabled without dequeuing an operation from *memQ*. The variable *memQ* represents a physical queue between the bus and the main memory, and *Delta* must be large enough so an operation inserted into an empty queue will reach the memory and be dequeued within *Delta* seconds.

We want the real-time write-through cache to implement the real-time memory specification. This requires an assumption relating *Delta*, *Epsilon*, and *Rho* to assure that the memory specification's timing constraint is satisfied—namely, that the delay between when the memory receives a request from processor p and when it responds is at most *Rho*. Determining this assumption requires computing an upper bound on that delay. Finding the smallest upper bound is hard; it's easier to show that

$$2 * (N + 1) * Epsilon + (N + QLen) * Delta$$

is an upper bound. So we assume that this value is less than or equal to *Rho*.

The complete specification appears in Figure 9.4 on the following two pages. The module also asserts as a theorem that the specification *RTSpec* of the real-time write-through cache implements (implies) the real-time memory specification, formula *RTSpec* of module *RTMemory*.

9.4 Zeno Specifications

I have described the formula $RTBound(HCnxt, hr, \delta, \epsilon)$ as asserting that an *HCnxt* step must occur within ϵ seconds of the previous *HCnxt* step. However, implicit in this description is a notion of causality that is not present in the formula. It would be just as accurate to describe the formula as asserting that *now* cannot advance by more than ϵ seconds before the next *HCnxt* step occurs. The formula doesn't tell us whether this condition is met by causing the clock to tick or by preventing time from advancing. Indeed, the formula is satisfied by a "Zeno" behavior:[3]

$$\begin{bmatrix} hr & = & 11 \\ now & = & 0 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = & 11 \\ now & = & \epsilon/2 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = & 11 \\ now & = & 3\epsilon/4 \end{bmatrix} \rightarrow \begin{bmatrix} hr & = & 11 \\ now & = & 7\epsilon/8 \end{bmatrix} \rightarrow \cdots$$

[3]The Greek philosopher Zeno posed the paradox that an arrow first had to travel half the distance to its target, then the next quarter of the distance, then the next eighth, and so on; thus it should not be able to land within a finite length of time.

MODULE *RTWriteThroughCache*

EXTENDS $\mathit{WriteThroughCache},\ \mathit{RealTime}$

CONSTANT N

ASSUME $(N \in \mathit{Nat}) \land (\mathit{Proc} = 0 \,..\, N-1)$

We assume that the set *Proc* of processors equals $0 \,..\, N-1$.

CONSTANTS $\mathit{Delta},\ \mathit{Epsilon},\ \mathit{Rho}$

Some real-time bounds on actions.

$$
\begin{aligned}
\textsc{assume}\ &\land (\mathit{Delta} \in \mathit{Real}) \land (\mathit{Delta} > 0)\\
&\land (\mathit{Epsilon} \in \mathit{Real}) \land (\mathit{Epsilon} > 0)\\
&\land (\mathit{Rho} \in \mathit{Real}) \land (\mathit{Rho} > 0)\\
&\land 2 * (N+1) * \mathit{Epsilon} + (N + \mathit{QLen}) * \mathit{Delta} \leq \mathit{Rho}
\end{aligned}
$$

We modify the write-through cache specification to require that operations for different processors are enqueued on *memQ* in round-robin order.

VARIABLE lastP

The last processor to enqueue an operation on *memQ*.

$\mathit{RTInit} \triangleq \mathit{Init} \land (\mathit{lastP} \in \mathit{Proc})$

Initially, *lastP* can equal any processor.

$\mathit{position}(p) \triangleq$

p is the $\mathit{position}(p)^{\text{th}}$ processor after *lastP* in the round-robin order.

$\quad$ CHOOSE $i \in 1 \,..\, N : p = (\mathit{lastP} + i) \,\%\, N$

$\mathit{canGoNext}(p) \triangleq$

True if processor p can be the next to enqueue an operation on *memQ*.

$\quad \forall q \in \mathit{Proc} : (\mathit{position}(q) < \mathit{position}(p)) \Rightarrow \lnot \textsc{enabled}\ (\mathit{RdMiss}(q) \lor \mathit{DoWr}(q))$

$$
\begin{aligned}
\mathit{RTRdMiss}(p) \triangleq\ &\land \mathit{canGoNext}(p)\\
&\land \mathit{RdMiss}(p)\\
&\land \mathit{lastP}' = p
\end{aligned}
$$

Actions $\mathit{RTRdMiss}(p)$ and $\mathit{RTDoWr}(p)$ are the same as $\mathit{RdMiss}(p)$ and $\mathit{DoWr}(p)$ except that they are not enabled unless p is the next processor in the round-robin order ready to enqueue an operation on *memQ*, and they set *lastP* to p.

$$
\begin{aligned}
\mathit{RTDoWr}(p) \triangleq\ &\land \mathit{canGoNext}(p)\\
&\land \mathit{DoWr}(p)\\
&\land \mathit{lastP}' = p
\end{aligned}
$$

$$
\begin{aligned}
\mathit{RTNext} \triangleq\ &\lor \exists p \in \mathit{Proc} : \mathit{RTRdMiss}(p) \lor \mathit{RTDoWr}(p)\\
&\lor \land \lor \exists p \in \mathit{Proc} : \lor \mathit{Req}(p) \lor \mathit{Rsp}(p) \lor \mathit{DoRd}(p)\\
&\hphantom{\lor \land \lor \exists p \in \mathit{Proc} : }\lor \exists a \in \mathit{Adr} : \mathit{Evict}(p, a)\\
&\hphantom{\lor \land }\lor \mathit{MemQWr} \lor \mathit{MemQRd}\\
&\hphantom{\lor }\land \textsc{unchanged}\ \mathit{lastP}
\end{aligned}
$$

The next-state action *RTNext* is the same as *Next* except with $\mathit{RTRdMiss}(p)$ and $\mathit{RTDoWr}(p)$ replaced by $\mathit{RdMiss}(p)$ and $\mathit{DoWr}(p)$, and with other actions modified to leave *lastP* unchanged.

$\mathit{vars} \triangleq \langle \mathit{memInt}, \mathit{wmem}, \mathit{buf}, \mathit{ctl}, \mathit{cache}, \mathit{memQ}, \mathit{lastP} \rangle$

Figure 9.4a: A real-time version of the write-through cache (beginning).

```
RTSpec ≜
  ∧ RTInit ∧ □[RTNext]_vars
  ∧ RTBound(MemQWr ∨ MemQRd, vars, 0, Delta)
  ∧ ∀ p ∈ Proc : ∧ RTBound(RTDoWr(p) ∨ DoRd(p) ∨ Rsp(p),
                              vars, 0, Epsilon)
                 ∧ RTBound(RTRdMiss(p), vars, 0, Epsilon)
  ∧ RTnow(vars)
├──────────────────────────────────────────────────────────────┤
RTM ≜ INSTANCE RTMemory
THEOREM RTSpec ⇒ RTM!RTSpec
└──────────────────────────────────────────────────────────────┘
```

We put an upper-bound delay of *Delta* on *MemQWr* and *MemQRd* actions (which dequeue operations from *memQ*), and an upper-bound delay of *Epsilon* on other actions.

Figure 9.4b: A real-time version of the write-through cache (end).

in which ϵ seconds never pass. We rule out such Zeno behaviors by conjoining to our specification the formula *RTnow*(*hr*)—more precisely by conjoining its liveness conjunct

$$\forall\, r \in \mathit{Real} : \mathrm{WF}_{\mathit{now}}(\mathit{Next} \land (\mathit{now}' > r))$$

which implies that time advances without bound. Let's call this formula *NZ* (for Non-Zeno).

Zeno behaviors pose no problem; they are trivially forbidden by conjoining *NZ*. A problem does exist if a specification allows *only* Zeno behaviors. For example, suppose we conjoined to the untimed hour-clock's specification the condition $\mathit{RTBound}(\mathit{HCnxt}, \mathit{hr}, \delta, \epsilon)$ for some δ and ϵ with $\delta > \epsilon$. This would assert that the clock must wait at least δ seconds before ticking, but must tick within a shorter length of time. In other words, the clock could never tick. Only a Zeno behavior, in which ϵ seconds never elapsed, can satisfy this specification. Conjoining *NZ* to this specification yields a formula that allows no behaviors—that is, a formula equivalent to FALSE.

This example is an extreme case of what is called a *Zeno specification*. A Zeno specification is one for which there exists a finite behavior σ that satisfies the safety part but cannot be extended to an infinite behavior that satisfies both the safety part and *NZ*.[4] In other words, the only complete behaviors satisfying the safety part that extend σ are Zeno behaviors. A specification that is not Zeno is, naturally enough, said to be *non-Zeno*. By the definition of machine closure (in Section 8.9.2 on page 111), a specification is non-Zeno iff it is machine closed. More precisely, it is non-Zeno iff the pair of properties consisting of the safety part of the specification (the conjunction of the untimed specification, the real-time bound conditions, and the safety part of the *RTnow* formula) and *NZ* is machine closed.

[4]Recall that, on page 112, a finite behavior σ was defined to satisfy a safety property P iff adding infinitely many stuttering steps to the end of σ produces a behavior that satisfies P.

A Zeno specification is one in which the requirement that time increases without bound rules out some finite behaviors that would otherwise be allowed. Such a specification is likely to be incorrect because the real-time bound conditions are probably constraining the system in unintended ways. In this respect, Zeno specifications are much like other non-machine-closed specifications.

Section 8.9.2 mentions that the conjunction of fairness conditions on subactions of the next-state relation produces a machine closed specification. There is an analogous result for *RTBound* conditions and non-Zeno specifications. A specification is non-Zeno if it is the conjunction of (i) a formula of the form $Init \wedge \Box[Next]_{vars}$, (ii) the formula $RTnow(vars)$, and (iii) a finite number of formulas of the form $RTBound(A_i, vars, \delta_i, \epsilon_i)$, where for each i

- $0 \leq \delta_i \leq \epsilon_i \leq Infinity$
- A_i is a subaction of the next-state action *Next*.
- No step is both an A_i and an A_j step, for any A_j with $j \neq i$.

The definition of a subaction appears on page 111.

In particular, this implies that the specification *RTSpec* of the real-time write-through cache in module *RTWriteThroughCache* is non-Zeno.

This result does not apply to the specification of the real-time memory in module *RTMemory* (Figure 9.3 on page 126) because the action $Respond(p)$ is not a subaction of the next-state action *INext* of formula *ISpec*. The specification is nonetheless non-Zeno, because any finite behavior σ that satisfies the specification can be extended to one in which time advances without bound. For example, we can first extend σ to respond to all pending requests immediately (in 0 time), and then extend it to an infinite behavior by adding steps that just increase *now*.

INext is defined on page 53

It's easy to construct an example in which conjoining an *RTBound* formula for an action that is not a subaction of the next-state action produces a Zeno specification. For example, consider the formula

$$(9.2) \quad HC \wedge RTBound(hr' = hr - 1, hr, 0, 3600) \wedge RTnow(hr)$$

where *HC* is the specification of the hour clock. The next-state action *HCnxt* of *HC* asserts that *hr* is either incremented by 1 or changes from 12 to 1. The *RTBound* formula asserts that *now* cannot advance for 3600 or more seconds without an $hr' = hr - 1$ step occurring. Since *HC* asserts that every step that changes *hr* is an *HCnxt* step, the safety part of (9.2) is satisfied only by behaviors in which *now* increases by less than 3600 seconds. Since the complete specification (9.2) contains the conjunct *NZ*, which asserts that *now* increases without bound, it is equivalent to FALSE, and is thus a Zeno specification.

When a specification describes how a system is implemented, the real-time constraints are likely to be expressed as *RTBound* formulas for subactions of the next-state action. These are the kinds of formulas that correspond fairly directly to an implementation. For example, module *RTWriteThroughCache* describes an algorithm for implementing a memory, and it has real-time bounds on subactions of the next-state action. On the other hand, more abstract, higher-level specifications—ones describing what a system is supposed to do rather than how to do it—are less likely to have real-time constraints expressed in this way. Thus, the high-level specification of the real-time memory in module *RTMemory* contains an *RTBound* formula for an action that is not a subaction of the next-state action.

9.5 Hybrid System Specifications

A system described by a TLA$^+$ specification is a physical entity. The specification's variables represent some part of the physical state—the display of a clock, or the distribution of charge in a piece of silicon that implements a memory cell. In a real-time specification, the variable *now* is different from the others because we are not abstracting away the continuous nature of time. The specification allows *now* to assume any of a continuum of values. The discrete states in a behavior mean that we are observing the state of the system, and hence the value of *now*, at a sequence of discrete instants.

There may be physical quantities other than time whose continuous nature we want to represent in a specification. For an air traffic control system, we might want to represent the positions and velocities of the aircraft. For a system controlling a nuclear reactor, we might want to represent the physical parameters of the reactor itself. A specification that represents such continuously varying quantities is called a *hybrid system specification.*

As an example, consider a system that, among other things, controls a switch that influences the one-dimensional motion of some object. Suppose the object's position p obeys one of the following laws, depending on whether the switch is off or on:

$$\begin{aligned} (9.3)\quad & d^2p/dt^2 \;+\; c * dp/dt \;+\; f[t] \;=\; 0 \\ & d^2p/dt^2 \;+\; c * dp/dt \;+\; f[t] \;+\; k * p \;=\; 0 \end{aligned}$$

where c and k are constants, f is some function, and t represents time. At any instant, the future position of the object is determined by the object's current position and velocity. So, the state of the object is described by two variables—namely, its position p and its velocity w. These variables are related by $w = dp/dt$.

We describe this system with a TLA$^+$ specification in which the variables p and w are changed only by steps that change *now*—that is, steps representing the passage of time. We specify the changes to the discrete system state and any real-time constraints as before. However, we replace $RTnow(v)$ with a formula having the following next-state action, where *Integrate* and D are explained below, and v is the tuple of all discrete variables:

$$\begin{array}{l} \land\ now' \in \{r \in Real : r > now\} \\ \land\ \langle p', w' \rangle = Integrate(D, now, now', \langle p, w \rangle) \\ \land\ \text{UNCHANGED}\ v \quad \text{The discrete variables change instantaneously.} \end{array}$$

The second conjunct asserts that p' and w' equal the expressions obtained by solving the appropriate differential equation for the object's position and velocity at time now', assuming that their values at time now are p and w. The differential equation is specified by D, while *Integrate* is a general operator for solving (integrating) an arbitrary differential equation.

To specify the differential equation satisfied by the object, let's suppose that *switchOn* is a Boolean-valued state variable that describes the position of the switch. We can then rewrite the pair of equations (9.3) as

$$d^2p/dt^2 + c * dp/dt + f[t] + (\text{IF}\ switchOn\ \text{THEN}\ k * p\ \text{ELSE}\ 0) = 0$$

We then define the function D so this equation can be written as

$$D[t, p, dp/dt, d^2p/dt^2] = 0$$

Using the TLA$^+$ notation for defining functions of multiple arguments, which is explained in Section 16.1.7 on page 301, the definition is

$$\begin{array}{l} D[t, p0, p1, p2 \in Real] \triangleq \\ \quad p2 + c * p1 + f[t] + (\text{IF}\ switchOn\ \text{THEN}\ k * p0\ \text{ELSE}\ 0) \end{array}$$

We obtain the desired specification if the operator *Integrate* is defined so that $Integrate(D, t_0, t_1, \langle x_0, \ldots, x_{n-1} \rangle)$ is the value at time t_1 of the n-tuple

$$\langle x, dx/dt, \ldots, d^{n-1}/dt^{n-1} \rangle$$

where x is a solution to the differential equation

$$D[t, x, dx/dt, \ldots, d^nx/ct^n] = 0$$

whose 0^{th} through $(n-1)^{\text{st}}$ derivatives at time t_0 are $x_0, \ldots, x_{n-1}$. The definition of *Integrate* appears in the *DifferentialEquations* module of Section 11.1.3 (page 174).

In general, a hybrid-system specification is similar to a real-time specification, except that the formula $RTnow(v)$ is replaced by one that describes the changes to all variables that represent continuously changing physical quantities. The *Integrate* operator will allow you to specify those changes for many hybrid systems. Some systems will require different operators. For example, describing the evolution of some physical quantities might require an operator for describing the solution to a partial differential equation. However, if you can describe the evolution mathematically, then it can be specified in TLA$^+$.

Hybrid system specifications still seem to be of only academic interest, so I won't say any more about them. If you do have occasion to write one, this brief discussion should indicate how you can do it.

9.6 Remarks on Real Time

Real-time constraints are used most often to place an upper bound on how long it can take the system to do something. In this capacity, they can be considered a strong form of liveness, specifying not just that something must eventually happen, but when it must happen. In very simple specifications, such as the hour clock and the write-through cache, real-time constraints usually replace liveness conditions. More complicated specifications can assert both real-time constraints and liveness properties.

The real-time specifications I have seen have not required very complicated timing constraints. They have been specifications either of fairly simple algorithms in which timing constraints are crucial to correctness, or of more complicated systems in which real time appears only through the use of simple timeouts to ensure liveness. I suspect that people don't build systems with complicated real-time constraints because it's too hard to get them right.

I've described how to write a real-time specification by conjoining *RTnow* and *RTBound* formulas to an untimed specification. One can prove that all real-time specifications can be written in this form. In fact, it suffices to use *RTBound* formulas only for subactions of the next-state action. However, this result is of theoretical interest only because the resulting specification can be incredibly complicated. The operators *RTnow* and *RTBound* solve all the real-time specification problems that I have encountered; but I haven't encountered enough to say with confidence that they're all you will ever need. Still, I am quite confident that, whatever real-time properties you have to specify, it will not be hard to express them in TLA$^+$.