Chapter 8

Real-time Specification

None of the properties we have considered relate to the actual speed of the hardware used in the implementation, or to any high level quantitative timing requirements. Such real-time requirements pose new difficulties in expression and verification. Real-time requirements are most common in what are known as hybrid systems, where software processes are used to control, monitor, or augment physical elements such as sensors or electronic devices. Man-machine interfaces are another fertile application area with real-time requirements.

The most common type of real-time specification is to declare an upper bound on the time allowed to take a particular step in the system. The step may appear atomic at the highest level of specification, but be composed of several steps at lower levels. It is also common to specify lower bounds. This is often expressed as a delay time that must pass before execution can proceed from certain states. For example, such a requirement might be necessary to allow a user sufficient time to react to an announcement on the screen. Specifications also may include limits on the time between key states or events that have many other intervening states.

Real-time requirements can be used instead of liveness properties to guarantee that particular steps are taken. In fact, liveness properties can be viewed as an abstraction of a real-time property where the precise length of time until an event occurs is irrelevant. Thus we might have specified

$$\square(p \Rightarrow \Diamond q)$$
to say that whenever \( p \) is true, \( q \) becomes true sometime later. At a later stage of development, the implementation may actually ensure that \( q \) will become true within 5 seconds of \( p \) becoming true. Thus the liveness requirement is replaced by a real-time requirement when additional information is available.

In many other specifications, real-time properties and liveness requirements co-exist both on the specification and on the implementation level. That is, some aspects are treated using real-time, while others are treated as liveness properties. Thus methods are needed that allow expressing both, and showing that real-time limits of execution times guarantee liveness properties such as eventually reaching a state that satisfies a given property.

Although real-time requirements can be added as an extension to any of the notations we have considered, we will concentrate on adding them to temporal logic, and to statecharts. It should be noted that many of the additions mentioned below are only proposals set forth in the literature, and have not been extensively used for specification, as have the languages seen in previous chapters.

### 8.1 Real-time in Temporal Logic

One straightforward approach to adding real-time involves using a special built-in free variable, which will be called \( \text{time} \) in the continuation. In the simplest version, this variable is assumed to be updated to the actual physical time arbitrarily often. It is used in conjunction with fixed event times or bounds on the time. Thus if \( \text{start} \) is intended to represent the time when a statement or module \( s \) is begun, \( \text{upper} \) is the bound on the maximal time for completing \( s \), and \( \text{lower} \) is the minimal time for completing \( s \), then we can express the requirements as

\[
\square (\text{in}(s) \Rightarrow (\text{time} - \text{start}) \leq \text{upper})
\]

(if \( s \) is executing, since \( \text{start} \) no more than \( \text{upper} \) time has passed.)

\[
\square (\text{after}(s) \Rightarrow \text{time} > \text{start} + \text{lower})
\]

(when \( s \) has just finished, \( \text{time} \) is at least \( \text{lower} \) units greater than the \( \text{start} \) time.)

\[
\square (\text{at}(s) \Rightarrow \text{time} \geq \text{start})
\]
(when \( s \) is about to begin, the time is at least \( \text{start} \).)

Note that the three formulas above can be taken as the formal definitions of the intended meaning of the constants \( \text{start} \), \( \text{upper} \), and \( \text{lower} \), in terms of the variable \( \text{time} \) and the predicates \( \text{in} \), \( \text{after} \), and \( \text{at} \) for \( s \). Surprisingly, all of the statements above are safety properties, involving only the \( \Box \) temporal operator. Yet these can be used to show liveness properties. This is possible because the \( \text{time} \) variable itself satisfies temporal properties. These properties need to be added as axioms so that desired liveness can be deduced from the above assertions along with the new axioms.

In expressing the key properties of the \( \text{time} \) variable, it is not sufficient to state that time advances (monotonically increases). If it increases by a smaller amount (say half of the previous amount) each time it increases, the value of \( \text{time} \) could still be bound from above. What is needed is the so-called non-Zeno property: for any constant \( r \)

\[
\Diamond (\text{time} > r)
\]

This means that for any constant value, the \( \text{time} \) variable will eventually be larger than that value. This property is named after the classic Zeno's paradox, in which a hare never overtakes a tortoise because the distance between them is always cut in half after a positive amount of time passes.

It is interesting that this single liveness requirement on the time variable is sufficient, along with safety properties connecting the program states to the time variable, to prove any other liveness property. For example, to establish the liveness property \( \text{in}(s) \Rightarrow \Diamond \neg \text{in}(s) \), the first equation above can be used to show that whenever \( \text{in}(s) \) is true, \( \text{time} \) is less than the fixed bound \( \text{start} + \text{upper} \). By the non-Zeno assumption, eventually there is a state in which \( \text{time} \) is greater than \( \text{start} + \text{upper} \), so in that state \( \neg \text{in}(s) \) must hold.

If various components have such time bounds, they can be checked for internal consistency. As a simple example, consider a module \( s \) that is the sequential composition of \( a \) and then \( b \). If \( \text{lower}_a \), \( \text{upper}_a \), \( \text{lower}_b \), \( \text{upper}_b \), \( \text{lower}_s \), and \( \text{upper}_s \) are all defined, and satisfy the required formulas expressing that they are lower and upper bounds on the time
to execute the corresponding module, then

\[ \text{lower}_a + \text{lower}_b \leq \text{upper}_s \]

must be true in order for there to be any possible timing. Otherwise, the sum of the minimal times to execute the sequential components is larger than the maximal time, so not all of the bounds can be satisfied. Similarly, it also must be true that

\[ \text{upper}_a + \text{upper}_b \geq \text{lower}_s \]

Otherwise, the longest time the components could take would still be less than the minimum time required for \( s \).

If \( s \) consists of \( a \) and \( b \) executing in parallel (with no synchronization), \( \text{upper}_s \) must be at least the maximum of \( \text{lower}_a \) and \( \text{lower}_b \) (i.e., substituting \( \max \) for addition in the first formula above), and the minimum of \( \text{upper}_a \) and \( \text{upper}_b \) must be at least \( \text{lower}_s \).

The use of a special variable to represent time is not universally accepted. It does not fit into the formal framework smoothly, since it is unclear how and when it is updated. The constants and marked times seen in \( \text{upper} \) or start are also not fully integrated into a formal framework. For example, confusion can result when the module \( s \) can be executed more than once. Beyond such technical objections, there may be separate local clocks that differ from physical time, and it may be necessary to coordinate these clocks with no reference to the “actual” time. In a relativistic approach, there is no absolute time against which to measure.

Another approach adds bounds to the temporal modalities involved in expressing liveness, without explicitly referring to a special time variable. Most commonly, bounds are added to the \( \Diamond \) and Until operators where \([a, b]\) means that at least \( a \) time units and at most \( b \) units could pass until the eventuality becomes true, or until the right hand side of the Until statement is satisfied. For example, the assertion

\[ \square(p \Rightarrow \Diamond[1, 5]q) \]

means that \( q \) will be true in a state at least 1 and at most 5 units after \( p \) has become true. Similarly, for the strong Until operator, we could
have some operators restricted by real-time constraints, and others not. The assertion

\[ \neg \text{in}(CS0) U (\text{in}(CS0) U [0, 4] (\neg \text{in}(CS0) U [3, 7] \text{in}(CS1))) \]

has the first \( U \) without any time restriction, while the second says that there is an upper bound of 4, and the third has both an upper and a lower bound. This assertion requires that not being in critical section CS0 is followed by being in that critical section, then within 4 time units not being in that section, followed by being in CS1, in not less than 3 and not more than 7 units.

As previously, the requirements can immediately be checked for internal consistency. As a trivial example, the requirement

\[ (\Diamond [2, 4] p) \land (\Diamond [5, 7] p) \]

is never true, because the time intervals do not overlap. The first conjunct requires that \( p \) become true in no more than four time units, while the second one requires that at least five units must pass before \( p \) becomes true. The lower bound on the eventuality operation seems problematic, because it means that the following predicate is not true until the lower bound has passed. It therefore resembles an Until statement. But even if such lower bounds are not used with the \( \Diamond \) operator, similar inconsistencies can be composed using Until.

Adding bounds, without an explicit time variable, has the added advantage of being easier to integrate with model checking algorithms. An arbitrary time variable is not finite state, and can make model checking impossible.

Yet a third proposal for incorporating real-time to temporal logic involves adding “bound variables” to unary temporal modalities such as \( \Box, \Diamond, \) and \( \lozenge \). The idea is to freeze the value of the time when the predicate following the modality is true. This resembles the use of bound variables with the usual \( \forall \) and \( \exists \) operators. Thus,

\[ \Diamond s. (x > y \land s < 7) \]

means that there is eventually a state in which \( x \) is greater than \( y \), and the time then (represented by \( s \)) is less than 7. Several significant
times, perhaps separated by intermediate states, can also be compared in this notation. The expression

$$\Box t. \ (x \geq y \Rightarrow (\Diamond (a > b \land \Diamond s. \ (x > b \land s - t \leq 8))))$$

means that whenever $x$ is greater or equal to $y$, $t$ is frozen to a value such that at some later state $a$ is greater than $b$ and then later there is a state with time $s$ such that $x$ is greater than $b$ and no more than 8 time units have passed in all, so that $s - t \leq 8$. The bound variables are added only to freeze significant times, and expressions involving program variables and bound variables (both from usual quantifiers and the temporal ones) are freely mixed. The basic non-Zeno requirement becomes the assertion

$$\forall r. \ \Diamond t. \ (t > r)$$

This proposal is an elegant generalization of the earlier temporal notation, and is more expressive than only adding time bounds. Yet it is easier to reason about automatically than the full use of an external time variable. In fact it has been shown that automatic model checking techniques can also be adapted to handle this generalization.

### 8.2 Interleaving and Tick Operations

The two approaches discussed above add real-time to temporal operators without mentioning a time variable, either by adding lower and/or upper ranges to the operators, or by adding bound variables that can be used indirectly to restrict the possible executions. In both approaches the progress of the “clock” or time variable seems to be intimately linked to the transitions from state to state, where the execution is viewed as a sequence of states. This is problematic because it goes against the basic rationalization for viewing executions of a system as a collection of sequences.

As will be recalled, the interleaving of actions from different processes to form an execution sequence is justified on the grounds that the steps in different processes are often independent. Whether independent steps in different processes occur at the same time or are arbitrarily interleaved is irrelevant to considerations of specification or analysis.
As long as there was no notion of absolute time, this seemed reasonable. But now, it does make a difference whether six time units have passed between two steps, or only one. Thus identifying the passage of time with the execution of an individual transition is problematic. It could be that three independent steps occur within one time unit. A linearization in which there is only one step per transition and the transition indicates that a unit of time has passed no longer correctly abstracts the system. Similar considerations arise in all specification methods that do not deal with an explicit time variable.

Even when explicit time is used, it is often necessary to calibrate the clock, i.e., to express the rate at which time changes in various parts of the system.

One way to overcome such problems is to view the progress of the clock as a separate action to be specified. That is, there is a tick action that can be specified as causing a transition to a new state. The other transitions are viewed as taking 0 time, and do not advance the time variable (or, for the alternative approaches, either are not counted in the ranges, or do not change the value of the bound variables associated with temporal operators). On the other hand, a tick transition does advance the time, but does not change the rest of the state. Of course, it then becomes necessary to globally specify restrictions on the use of the tick operation.

Note that the non-Zeno property must still hold. One way to guarantee this is if ticks advance the clock by a full unit of time and are guaranteed to occasionally occur. If we define a predicate tick that is true in a state immediately following a tick operation, then a “unit tick” may be expressed as

$$\Box t. \bigcirc s. ((\text{tick} \Rightarrow s = t + 1) \land ((\neg \text{tick}) \Rightarrow s = t))$$

That is, in any state with a time $t$, if in the next state the time is $s$, and a tick has just occurred, $s$ is one larger than $t$, while otherwise $s = t$. The desirable guarantee that ticks will occur follows from the temporal statement

$$\Box t. \Diamond s. (s > t)$$

This means that for any state there is a later state with a larger time value, so a tick must have occurred. The non-Zeno property follows from the two temporal characterizations above.
However, it is not always appropriate to require a unit clock, and in that case, the non-Zeno property can be directly required along with occasional ticks. In other words, time must exceed each value, but without specifying a particular rate of increase. There are also notions of time that increase by non-discrete amounts, whenever a tick occurs. In these it is necessary to require that every possible time is ‘passed’ as time increases, so that a deadline cannot be missed by jumping past the indicated time in an atomic step. Specifications of such continuous-time clocks are more complex, but can be integrated with hybrid systems where part of the system is considered analogue rather than digital, and is (typically) described by differential equations.

In addition, restrictions are needed that require ticks at reasonable intervals, e.g., that a single process cannot take too many steps ‘at the same time,’ or that a tick must occur between sequentially executed local steps. The possibility of expressing such requirements adds flexibility to the specification, and allows modeling processors with different speeds or ‘coarse-grained’ clocks. In fact, a ‘gallery’ of clocks (actually, of notions of real-time and rates of progress) could be developed as a library of augmented temporal assertions. These then could be incorporated into particular specifications as needed, and used in model checking or other analysis tools. As is so often the case in formal specifications, sometimes it becomes difficult to recognize that the formal requirements correspond to the desired intuition of real-time.

### 8.3 Real-time statecharts

In the present version of statecharts, real-time restrictions are the only way to specify liveness, and in particular that designated transitions must occur. Otherwise, statecharts only specify possible behaviors, but do not require that any transition take place. An implementation of a statechart could simply “rest” in some state, and be considered correct if there were no real-time requirements.

Time is viewed in statecharts as an absolute global value updated continuously outside of the system. Recall that some transitions are activated as actions associated with taking another transition, and are known as micro-steps. These micro-steps are assumed to occur at the
same time as the original transition. This somewhat supports the “tick” view, where not every transition advances time, but there is no real calibration of how long a transition should take, or whether all time passes inside states. There is a built-in condition for transitions known as timeout that forces a transition to be followed when a given time has elapsed. This can be seen in the watch example that deals with the beep sound, or the mode change to view the date.

It is also convenient to add upper or lower time bounds to states. By writing in a corner of the state “5 ≤” the requirement is expressed that control will remain in the state, once it is entered, at least 5 time units. Similarly, “≤ 8” means that no more that 8 units will pass from the time the state is entered until a transition out of the state will be taken.

Along with other keywords used in the Statemate system, these notations allow reasonable specifications of many real-time properties. As noted previously, such restrictions are especially common in specifying hybrid systems where software is used to control or drive physical devices, or for user interfaces where response time is vital. Avionic systems that control all modern aircraft are a particularly crucial example where real-time has been used with statecharts. Since these are application areas where statecharts have been commonly used, it is not surprising that built in notations for real-time are already incorporated into statecharts.

One natural extension adds upper and lower time bounds to transitions, after the conditions for taking the transition. This extension can be understood as implying that a transition itself takes time, and during that time the system is neither in the source nor the target state. Another interpretation could be that the transition itself is instantaneous, but will be taken only after an appropriate delay to satisfy the time bounds. This interpretation is actually the commonly accepted one, because it is easier to handle for purposes of analysis or simulation.

Under either interpretation, consistency of the real-time conditions can again be checked as inequalities involving upper and lower bounds. Now the consistency conditions are more complex, because of the activation of transitions in parallel components. Recall that a single step could be a joint transition or could lead to micro-steps in parallel components. The conditions for activation also add real-time dependencies.
For example, if a transition must occur within $c$ time units, but has a condition that it can only occur when a parallel component is in a state $R$, the timing constraints for entering or leaving $R$ are relevant. Such an analysis can be extremely useful in revealing real-time dependencies. Even if time bounds on transitions are allowed, real-time in state-charts deals with individual transitions or states (that may be super-states). Therefore it can be difficult to express more global restrictions on the time allowed for a collection or long sequence of actions. For example an elevator system may require that each floor is visited by an elevator within 5 minutes of a request, if the stop on each other floor takes no longer than 30 seconds. For such requirements, one of the real-time versions of temporal logic seems necessary.

### 8.4 Difficulties in real-time

These and other real-time additions to existing specification methods are becoming increasingly widespread. There is research on adding timing properties to LOTOS, or to other process algebras. For Larch it is most natural to add such considerations at the interface language rather than in the LSL shared language part, and again there is work demonstrating how this can be done. Similar proposals exist for the many specification notations not treated in detail in this book.

All of the proposals for real-time specification suffer from similar difficulties: there is no methodology for top-down development or splitting of states that takes into account the real-time restrictions. The problem is that the requirements are expressed in terms of the high level specification (e.g., “a status check must be completed within 0.3 seconds”), while whether the requirement is satisfied depends on extremely low level considerations. Among the factors that determine whether a real-time requirement is satisfied are the clock cycle times of the machines on which the code is executing, communication times on channels, the number of cycles needed for each operation, and the number of machine code operations executed in each processor. This information is usually not available in advance.

Often, the real-time requirements are simply passed from level to level in a top down development. When, for example, a module is im-
plemented by sequential composition of two lower level components, the division of how much time each should take from the total is arbitrary. Only when the final implementation is found not to execute within the time bounds, can adjustments be made to either change the requirement, or redistribute the loads on component processors. So one of the most valuable benefits of formal specifications—discovering problems at an early stage—is difficult to achieve in this context.

Real-time requirements seem especially relevant and easy to use in descriptions of synchronous hardware or firmware designs. In this case the components are synchronous, and take steps according to a global clock cycle. Thus here it is justified to associate a transition with a unit update of the global clock. One component can depend on the number of cycles that have occurred to obtain information on the status of other components. Errors due to hidden real-time dependencies are especially common in synchronous hardware designs. The added value of providing error analysis, feedback, and the ability to investigate alternative timing options during design seems impressive. Extensions to CTL and to linear temporal logic have been made that allow incorporating real-time requirements about synchronous state machines. As already noted, if bound variables or ranges on the temporal operators are used, these extensions may still employ model checking tools to determine whether a design (a state machine) satisfies the requirements.

Real-time requirements of varying strictness are becoming widespread, from software to control cars and washing machines, to video games, to industrial robots. It is clear that formal notations to deal with real-time issues will find their way into robust versions of specification methods. However, significant research remains to be done in this area, especially for systems involving asynchronous software processes.

8.5 Bibliographic remarks

Extensions of linear temporal logic to handle real time can be found in [7, 6].

The use of discrete time steps, and in particular using the bound variables with temporal modalities was proposed in Henzinger, Manna, Pnueli [5] and Alur and Henzinger [3] for timed transition systems. The
use of upper and lower bounds on temporal operators appears in [4].

Timing considerations have been added to the process algebra $CSP$ in [8]. Efficient and modular model checking techniques have been developed especially for the timed automata model introduced by Alur and Dill [1]. Translations from a metric temporal logic to timed automata, with model checking procedures, are shown in [2]. As noted in the chapter, especially intensive research is ongoing on methods for specification and verification of real-time properties, so new techniques and tools can be expected.
Bibliography


