7

Timed Model

In the synchronous model of computation, all components execute in lock-step, and the production of outputs by a component is synchronized with the reception of inputs. In the asynchronous model of computation, all processes execute at independent speeds, and there is an unspecified delay between the reception of inputs and the production of outputs by a process. Now we turn our attention to a *timed* model of computation where processes are not tightly synchronized to execute in a sequence of rounds but rely on the global physical time to achieve a loose form of synchronization. The timed model allows us to express phenomena such as "execute the task corresponding to sensing of temperature every 5ms," "the delay between the reception of an input value and the corresponding output response is between 2ms to 4ms," and "if an acknowledgment is not received within 4ms, resend."

7.1 Timed Processes

The formal model of computation for timed processes is a variation of the model of *asynchronous processes* from chapter 4. We will first illustrate the model with examples.

7.1.1 Timing-Based Light Switch

Consider a light switch that uses a single push-button, along with a built-in timer, to control a light bulb with two intensity levels. The switch is initially off. When it is pressed once, it turns on the light at a dim intensity, and if it is pressed twice in rapid succession, then the light is turned on at a bright intensity. Here, *rapid* means that the duration between the successive press events is less than one second. If the delay between the successive press events is more than one second, the second press event is interpreted as a command to switch the light off.

The system is modeled by the timed process `LightSwitch` shown in figure 7.1. It

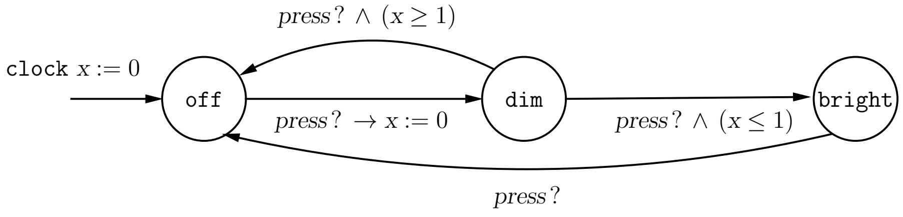
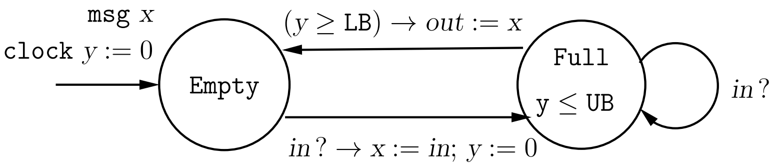


Figure 7.1: A Timed Model of a Light Switch

has an input channel *press* on which it receives events corresponding to pressing of the switch. The dynamics is illustrated using the extended-state machine notation. In this example, the mode can be either `off`, `dim`, or `bright`. The process uses a state variable x whose type is `clock`. A timed process has input, internal, and output actions just like an asynchronous process: an input action receives an input value on an input channel, an output action produces an output value on an output channel, and an internal action updates only the state variables and does not involve any input or output channels. During each such action, the clock variables are tested and updated in the same way as other state variables. The distinguishing new feature of the model is that a timed process also has *timed actions* that capture elapse of time. During a timed action of duration δ, which may be any positive real number, the value of each clock variable is incremented by the amount δ. A state variable, such as the variable *mode* for the process `LightSwitch`, of a type other than `clock` is called a *discrete* variable. A discrete state variable stays unchanged during a timed action.

For the process `LightSwitch`, initially the mode is `off`, the clock variable x is 0, and the process is waiting for the input event *press*. Waiting for a time period of δ_1 is modeled by a timed action of duration δ_1. After such an action, the mode is still `off`, but the value of the clock variable x equals δ_1. When the input event *press* is received, the process updates the mode to `dim` and resets the clock variable x to 0. As the process waits in the mode `dim`, the value of the clock variable x captures the time elapsed since the time instance when the mode-switch from `off` to `dim` occurred. Waiting in the mode `dim` for a total duration of length δ_2 corresponds to a timed action of duration δ_2, and the value of the clock variable x after such a timed action equals δ_2. When the subsequent input event `press` occurs, the value of the clock variable x is used to decide if the process updates the mode to `bright` or to `off`, and this is captured by the conjuncts $(x \leq 1)$ and $(x \geq 1)$ in the guard conditions of the two mode-switches out of the mode `dim`. Note that if the value of x is exactly 1 (that is, the duration between the two successive *press* events is one second), both these mode-switches are enabled, and thus the model behaves nondeterministically switching either to the mode `off` or to the mode `bright`. When the process is in the mode `bright`, it switches back to the initial mode `off` whenever it



msg x
clock $y := 0$
Empty
Full
$y \leq$ UB
$(y \geq \texttt{LB}) \to out := x$
$in? \to x := in;\ y := 0$
$in?$

Figure 7.2: A Timed Buffer with a Bounded Delay

receives the next input event.

One possible execution of the process is shown below, where each state is specified by listing the mode and the value of the clock variable x:

$$(\texttt{off}, 0) \xrightarrow{2.3} (\texttt{off}, 2.3) \xrightarrow{press?} (\texttt{dim}, 0) \xrightarrow{0.2} (\texttt{dim}, 0.2) \xrightarrow{0.5}$$
$$(\texttt{dim}, 0.7) \xrightarrow{press?} (\texttt{bright}, 0.7) \xrightarrow{3.0} (\texttt{bright}, 3.7) \xrightarrow{press?} (\texttt{off}, 3.7).$$

Note that during an execution, two timed actions may follow one another. In such a case, the effect of a timed action of duration δ_1 immediately followed by a timed action of duration δ_2 is identical to a single timed action of duration $\delta_1 + \delta_2$.

7.1.2 Buffer with a Bounded Delay

As a second example, let us consider a timed buffer of capacity 1 with an input channel *in* and an output channel *out*, both of type `msg`. Whenever an input value v is supplied to the buffer, it is stored in the internal discrete state variable x. Now the buffer becomes full, and it simply ignores (or loses) further inputs until it gets a chance to output the stored value on the output channel. The timing assumption is that the delay between the reception of an input value and the transmission of the corresponding output value is at least `LB` and at most `UB` time units, where the constants `LB` and `UB` capture the lower and upper bound, respectively, on the delay. This form of lower and upper bounds on delays is a commonly occurring pattern for timed systems.

The timed process `TimedBuf` shown in figure 7.2 captures the desired timed behavior using one clock variable y. The mode of the state machine indicates whether the buffer is empty or full. Initially the mode is `Empty`. When the input is received on the channel *in*, the message value is stored in the state variable x, the mode is updated to `Full`, and the clock y is set to 0. As time elapses while the mode of the process equals `Full`, the value of the clock y captures the duration of time the process has been waiting in this mode. Input events received in the mode `Full` do not change the buffer state, and this is modeled by the mode-switch corresponding to the self-loop on the mode `Full`. The mode-switch corresponding to the output actions is guarded with the condition

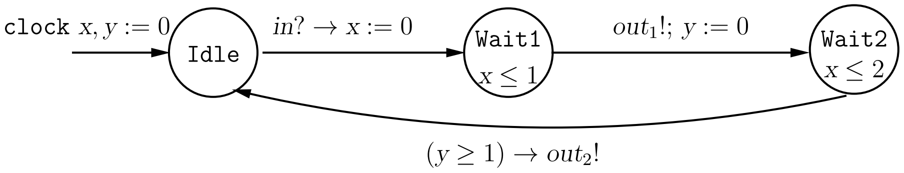


Figure 7.3: A Timed Process with Two Clocks

$(y \geq$ `LB`$)$ and thus captures the assumption that the buffer can issue the message on its output channel only after the lower bound `LB` on the delay.

The assumption concerning the *upper bound*, namely, that the process is guaranteed to produce the output within `UB` time units of receiving an input, is captured by the annotation $y \leq$ `UB` associated with the mode `Full`. If the mode is `Full` and the value of the clock y equals δ, then a timed action of duration δ' is allowed only if the constraint $y \leq$ `UB` holds throughout the transition as the value of the clock y keeps increasing with time, that is, only if the condition $\delta + \delta' \leq$ `UB` holds. The process and its environment synchronize on the passage of time during a timed action. The process wants to issue the output before the clock y reaches `UB` and thus is willing to let time elapse only up to a certain limit. The condition $y \leq$ `UB` associated with the mode `Full` is called a *clock invariant*. The clock invariant associated with a mode and the guard condition associated with the switches out of that mode together ensure lower and upper bounds on delays.

In general, each mode has an associated clock invariant, which is an expression over the clock variables, along with possibly other discrete state variables, of the process. When the clock invariant associated with a mode is the constant 1, it means that there is no upper bound on how long the process can wait in this mode. In such a case, we omit the annotation, as is the case for the mode `Empty` of the process `TimedBuf` and also for all the modes of the process `LightSwitch` of figure 7.1.

7.1.3 Multiple Clocks

As an example of a timed process that uses two clocks, consider the process shown in figure 7.3 with an input channel *in* and the output channels out_1 and out_2. If an input event on the channel *in* happens at time t, then the process responds by producing an output event on the channel out_1 at time t_1 followed by an output event on the channel out_2 at time t_2 such that (1) the delay $t_1 - t$ is at most 1, (2) the delay $t_2 - t$ is at most 2, and (3) the delay $t_2 - t_1$ is at least 1. Thus, the output event on the channel out_2 is constrained to occur within an interval that depends on the timing of the preceding input event on the channel *in* as well as the output event on the channel out_1. The process does not accept inputs on channel *in* until it has issued both output events.

`nat` $x_1 := 0;\ x_2 := 0$
`clock` $y_1 := 0;\ y_2 := 0$
$CI:\ (y_1 \leq 2) \wedge (y_2 \leq 2)$
$A_1:\ (y_1 \geq 1) \rightarrow \{x_1 := x_1 + 1;\ y_1 := 0\}$
$A_2:\ (y_2 \geq 1) \rightarrow \{x_2 := x_2 + 1;\ y_2 := 0\}$

Figure 7.4: Timed Process `TimedInc` with Parallel Increments

The desired constraints are expressed using two clock variables x and y. Initially the mode is `Idle`, and both clock variables equal 0. Waiting in the mode `Idle` for a duration of time δ is modeled by a timed action of duration δ, which updates both clock variables to the value δ. When an input event occurs, the process sets the clock x to 0, the clock y remains δ, and the mode is updated to `Wait1`. Waiting in the mode `Wait1` for a duration of time δ' increments the clock x to δ' and the clock y to $\delta + \delta'$. Such a timed action is allowed only as long as the value of x does not exceed 1 due to the clock-invariant $x \leq 1$ associated with the mode `Wait1`. The output event out_1 can occur at any time instance before the value of the clock x exceeds 1. At this point, the process switches to the mode `Wait2`, the clock y is reset to 0, and the value of the clock x indicates the time spent in the mode `Wait1`. When the mode of the process equals `Wait2`, both clocks increase with time, with the value of the clock x capturing the time elapsed since the occurrence of the input event and the value of the clock y capturing the time elapsed since the occurrence of the output event on the channel out_1. The clock-invariant $x \leq 2$ associated with the mode `Wait2` and the guard condition $(y \geq 1)$ associated with the mode-switch from the mode `Wait2` to the mode `Idle` capture the desired timing constraints on the occurrence of the output event on the channel out_2. Below is a sample execution of the process, where each state is specified by listing the mode, the value of the clock variable x, and the value of the clock variable y:

$$(\texttt{Idle}, 0, 0) \xrightarrow{5.7}, (\texttt{Idle}, 5.7, 5.7) \xrightarrow{in?} (\texttt{Wait1}, 0, 5.7) \xrightarrow{0.6} (\texttt{Wait1}, 0.6, 6.3) \xrightarrow{out_1!}$$
$$(\texttt{Wait2}, 0.6, 0) \xrightarrow{0.5} (\texttt{Wait2}, 1.1, 0.5) \xrightarrow{0.8} (\texttt{Wait2}, 1.9, 1.3) \xrightarrow{out_2!} (\texttt{Idle}, 1.9, 1.3).$$

Note that if we restrict the process to use only one clock, then the desired timing constraints cannot be expressed accurately.

As another example of a timed process, consider the process `TimedInc` shown in figure 7.4, which is a modified version of the asynchronous process `AsyncInc` of figure 4.2. The process `TimedInc` has two discrete state variables x_1 and x_2, both of which are initialized to 0 and are incremented by the internal tasks A_1 and A_2, respectively. However, unlike the asynchronous process `AsyncInc`, the order in which these two tasks execute is no longer completely unconstrained.

The time delay between the successive executions of the task A_1 is at least one and at most two time units. This constraint is specified using the clock variable y_1: initially its value is 0, the task A_1 can be executed only when the guard $(y_1 \geq 1)$ is satisfied, its execution resets the clock y_1 to 0, and the clock-invariant has a conjunct $(y_1 \leq 2)$, which ensures that the task A_1 must get executed before more than two time units elapse since its last execution. The clock y_2 is used in a similar manner to ensure that the time delay between the successive executions of the task A_2 is at least one and at most two time units.

Although the two tasks A_1 and A_2 do not have any variables in common, the fact that the two clocks y_1 and y_2 need to increase by the same amount of duration during a timed action constrains the relative frequencies at which the two tasks execute. In particular, consider an execution of the process in which the task A_1 executes twice without executing the task A_2. Such an execution then has the form below (the state lists the variables x_1, y_1, x_2, and y_2 in that order):

$$(0,0,0,0) \xrightarrow{\delta_1} (0,\delta_1,0,\delta_1) \xrightarrow{A_1} (1,0,0,\delta_1) \xrightarrow{\delta_2} (1,\delta_2,0,\delta_1{+}\delta_2) \xrightarrow{A_1} (2,0,0,\delta_1{+}\delta_2)$$

Based on the guard of the task A_1, we know that $\delta_1 \geq 1$ and $\delta_2 \geq 1$, and based on the clock-invariant, we can conclude that $\delta_1 + \delta_2 \leq 2$. These constraints can be satisfied only if $\delta_1 = \delta_2 = 1$, and thus in the state after executing the task A_1 twice, the clock y_2 must be 2. It means that in this state, time cannot elapse, and the only possible action is the execution of the task A_2. Thus, the variable x_2 must be incremented at least once before the variable x_1 is incremented thrice.

7.1.4 Formal Model

Recall the definition of an asynchronous process from section 4.1: an asynchronous process P has:

1. a finite set I of typed input channels defining the set of inputs of the form $x\,?\,v$ with $x \in I$ and a value v for x;

2. a finite set O of typed output channels defining the set of outputs of the form $y\,!\,v$ with $y \in O$ and a value v for y;

3. a finite set S of typed state variables defining the set Q_S of states;

4. an initialization *Init* defining the set $[\![Init]\!] \subseteq Q_S$ of initial states;

5. for each input channel x, a set $\mathcal{A}_x$ of input tasks, each described by a guard condition over S and an update from the read-set $S \cup \{x\}$ to the write-set S defining a set of input actions of the form $s \xrightarrow{x\,?\,v} t$;

6. for each output channel y, a set $\mathcal{A}_y$ of output tasks, each described by a guard condition over S and an update from the read-set S to the write-set $S \cup \{y\}$ defining a set of output actions of the form $s \xrightarrow{y\,!\,v} t$; and

7. a set $\mathcal{A}$ of internal tasks, each described by a guard condition over S and an update from the read-set S to the write-set S defining a set of internal actions of the form $s \xrightarrow{\varepsilon} t$.

We can define the formal model for timed processes as an extension of the above definition of asynchronous processes. The notions of input, output, and state variables, and input, output, and internal tasks and actions stay unchanged. The additional notion is that of a clock invariant, which is a Boolean expression over state variables, and is used to define timed actions of duration δ. Given a state s, which is a valuation of all the state variables, and a positive real number δ, let $s + \delta$ denote the state that assigns the value $s(x) + \delta$ to every clock variable x and the value $s(y)$ to every discrete state variable y. The state resulting from the timed action of duration δ starting in a state s is the state $s + \delta$. Such a timed action is allowed only if the Boolean condition specified by the clock invariant holds in all the states encountered during this interval. The definition of a timed process is now summarized below.

Timed Process

A *timed process TP* consists of:

- an asynchronous process P, where some of its state variables can be of type `clock`; and
- a *clock invariant CI*, which is a Boolean expression over the state variables S.

Inputs, outputs, states, initial states, internal actions, input actions, and output actions of the timed process *TP* are the same as that of the asynchronous process P. Given a state s and a real-valued time $\delta > 0$, $s \xrightarrow{\delta} s+\delta$ is a *timed action* of *TP* if the state $s + t$ satisfies the expression *CI* for all values $0 \leq t \leq \delta$.

For the timed process `TimedBuf` of figure 7.2, the various components are listed below:

- it has a single input channel *in* of type `msg`;
- it has a single output channel *out* of type `msg`;
- it has a (discrete) state variables *mode* of enumerated type $\{$`Empty`, `Full`$\}$ and *x* of type `msg`, and a variable *y* of type `clock`;
- the initial value of the clock variable *y* is 0, the initial value of the mode variable *mode* is `Empty`, and the initial value of the variable *x* is unconstrained;
- there is a single input task for processing the input channel *in*, its guard condition is 1 (that is, the task is always enabled), and the update corresponding to the extended-state machine of figure 7.2 can be equivalently written as:

$$\texttt{if}\ (mode = \texttt{Empty})\ \texttt{then}\ \{\ mode := \texttt{Full};\ x := in\};$$

- there is a single output task for producing outputs on the channel *out*, has the guard condition $(mode = \texttt{Full})\ \wedge\ y \geq \texttt{LB}$, and the corresponding update code is $out := x$;
- it has no internal tasks;
- the clock invariant *CI* is given by the expression

$$(mode = \texttt{Full})\ \rightarrow\ (y \leq \texttt{UB}).$$

Note that the clock invariant puts no constraints when the mode is `Empty`. The translation from the extended-state machine notation to the formal definition of timed processes can be automated.

The definition of a timed action requires that starting in a state s, a timed action of duration δ is possible if the state $s + t$ satisfies the clock invariant at every time t during the interval $[0, \delta]$. Typically the expressions used in clock invariants are *convex* functions of values of the clock variables, so it suffices to check that the starting state s and the final state $s + \delta$ both satisfy the clock invariant.

As in the case of asynchronous processes, the operational semantics of a timed process can be captured by defining its executions. An execution starts in an initial state and proceeds by executing an input action, an output action, an internal action, or a timed action at every step. Note that input, output, and internal actions are interleaved as in an asynchronous process. However, during a timed action, the clocks belonging to different processes all increase together, reflecting the passage of the same global time, and thus a timed action is executed synchronously. This is why sometimes this model is called a *partially synchronous model.*

Exercise 7.1 : Consider a timed process with an input event x and two output events y and z. Whenever the process receives an input event on the channel x, it issues output events on the channels y and z such that (1) the time delay between $x?$ and $y!$ is between two and four units, (2) the time delay between $x?$ and $z!$ is between three and five units, and (3) while the process is waiting to issue its outputs, any additional input events are ignored. Design a timed state machine that exactly models this description. ■

Exercise 7.2 : Consider a timed process with two input events x and y and an output event z. Initially, the process is waiting to receive an input event $x?$. If this event occurs at time t, then the process waits to receive an input on the channel y. If the event $y?$ occurs before time $t+2$ or does not occur before time $t + 5$, then the process simply returns to the initial state, and if the event $y?$ is received at some time t' between times $t + 2$ and $t + 5$, then the process issues an output event on z at some time between times $t' + 1$ and $t + 6$ and returns to the initial state. Unexpected input events (e.g., the event y in the initial mode) are ignored. Design a timed state machine that exactly models this description. ■

Exercise 7.3: Consider an asynchronous OR gate with Boolean input variables x and y and Boolean output variable z. Assume that initially all the variables have value 0. The event x? denotes toggling of the input wire x, and similarly the event y? denotes the toggling of the input wire y. The gate can change its output by issuing the event z!. The desired timing behavior is specified by the following rules:

1. When an input variable changes at time t_1, if this change warrants a change in the output (according to the standard logic of OR gate), then the output should be issued at time t_2 such that the delay $t_2 - t_1$ is between two and four time units (unless the inputs change again during the interval from t_1 to t_2; if so, see the rules below).

2. While a change in the output is pending in the interval $[t_1, t_2]$, if one of the input variables changes again, but this change is consistent with the output change about to happen at time t_2, then the output should change as scheduled.

3. While a change in the output is pending in the interval $[t_1, t_2]$, if one of the input variables changes at time t in a manner so as to make the change in output inconsistent with the revised inputs, then the behavior depends on the relative difference $t - t_1$: if this difference is less than 1, then the pending output change is canceled; if it is more than 1, then the output change will occur as scheduled at time t_2, and at that time, another output event is scheduled with a delay of two to four time units.

Based on this description, design a timed process (as an extended-state machine with one clock variable) that models the OR gate. The process should be input-enabled: it should allow input events to happen at all times. ■

7.1.5 Timed Process Composition

Timed processes can be composed together using block diagrams. Operations such as input-output variable renaming and output hiding are defined in the usual manner. Let us consider the operation of composing timed processes. To compose two timed processes, we first compose the corresponding asynchronous processes using the composition operation for asynchronous processes as described in section 4.1. The clock invariant for the composed process is simply the *conjunction* of the clock invariants of the component processes.

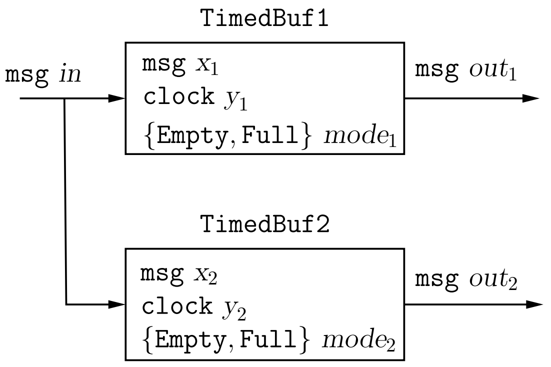


Figure 7.5: Composition of Two Instances of the Process `TimedBuf`

TIMED PROCESS COMPOSITION

For two timed processes $TP_1 = (P_1, CI_1)$ and $TP_2 = (P_2, CI_2)$, such that the output channels of the two processes are disjoint, the *parallel composition* $TP_1 \,|\, TP_2$ is the timed process whose asynchronous process is $P_1 \,|\, P_2$ and whose clock invariant is $CI_1 \wedge CI_2$.

Thus, the internal, input, and output actions of the composite process are obtained from the corresponding actions of the component processes using the asynchronous composition. The conjunction of the clock invariants means that a timed action of duration δ is possible in the composite process only if it is acceptable for each component process to wait for a duration of δ. For states s_1 and s_2 of the processes TP_1 and TP_2, respectively, and a time duration $\delta > 0$, $(s_1, s_2) \xrightarrow{\delta} (s_1 + \delta, s_2 + \delta)$ is a timed action of the composite process $TP_1 \,|\, TP_2$ exactly when $s_1 \xrightarrow{\delta} s_1 + \delta$ is a timed action of TP_1 and $s_2 \xrightarrow{\delta} s_2 + \delta$ is a timed action of TP_2.

Product of Timed State Machines

To understand how the composition works, let us describe the composition of two instances of the timed process `TimedBuf` connected in parallel with a common input channel. Figure 7.5 shows two instances with their variables renamed appropriately. The process `TimedBuf1` responds to the input event on the channel *in* by producing an output event on the channel out_1 after a delay of at least $\texttt{LB}_1$ and at most $\texttt{UB}_1$ time units, and the process `TimedBuf2` responds to the input event on the channel *in* by producing an output event on the channel out_2 after a delay of at least $\texttt{LB}_2$ and at most $\texttt{UB}_2$ time units. Instead of compiling the extended-state machine of figure 7.2 for each process into a description consisting of tasks and then computing the tasks for the composite process using the composition operation, let us construct the extended-state

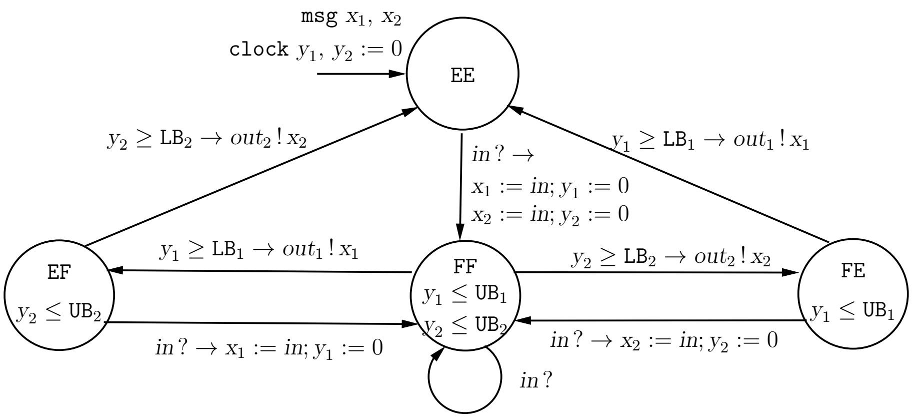

msg x_1, x_2
clock y_1, $y_2 := 0$
EE
$y_2 \geq$ LB$_2 \rightarrow out_2\,!\,x_2$
$in\,? \rightarrow$
$x_1 := in; y_1 := 0$
$x_2 := in; y_2 := 0$
$y_1 \geq$ LB$_1 \rightarrow out_1\,!\,x_1$
EF
$y_2 \leq$ UB$_2$
$y_1 \geq$ LB$_1 \rightarrow out_1\,!\,x_1$
FF
$y_1 \leq$ UB$_1$
$y_2 \leq$ UB$_2$
$y_2 \geq$ LB$_2 \rightarrow out_2\,!\,x_2$
FE
$y_1 \leq$ UB$_1$
$in\,? \rightarrow x_1 := in; y_1 := 0$
$in\,? \rightarrow x_2 := in; y_2 := 0$
$in\,?$

Figure 7.6: State Machine for Composition of Two `TimedBuf` Processes

machine capturing the behavior of the composition by taking a *product* of the state machines for the component processes.

The behavior of the parallel composition of the two processes is captured by the extended-state machine shown in figure 7.6. Since each component has two possible modes, the composite process has four modes. The initial mode is `EE` indicating that both component processes start in the mode `Empty`. When an input on the channel *in* is processed, the mode changes to `FF` (that is, the variable $mode_1$ is `Full` and the variable $mode_2$ is `Full`). The variables x_1 and y_1 are updated according to the input action of the first process, and the variables x_2 and y_2 are updated according to the input action of the second process.

The mode `FF` corresponds to the case when each process is in the mode `Full`. The clock-invariant of this mode is the conjunction ($y_1 \leq$ UB$_1 \wedge y_2 \leq$ UB$_2$). Thus, the composite process can wait in this mode only as long as the clock y_1 does not exceed UB$_1$ and the clock y_2 does not exceed UB$_2$. This conjunctive constraint reflects the synchronization of the two component processes on timed actions. The mode can change in two ways depending on which component process produces the output first. If the second component issues its output on the channel out_2, the mode changes to `FE` (that is, the variable $mode_1$ is `Full` and the variable $mode_2$ is `Empty`). This switch is guarded by the condition ($y_2 \geq$ LB$_2$), corresponding to the guard of the output action of the second component, and the variables of the first component stay unchanged during this switch. The clock-invariant in the mode `FE` is ($y_1 \leq$ UB$_1$) since the second component does not impose any constraints on how long the process can wait in this mode. In the mode `FE`, if the first component produces its output, then the mode changes to `EE`, and if an input event is received, then the mode switches back to `FF`.

Note that the values of the parameters that capture lower and upper bounds

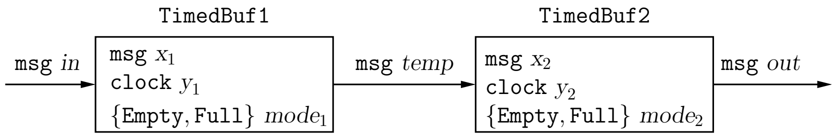


Figure 7.7: Composition of Two Instances of `TimedBuf` Processes in Series

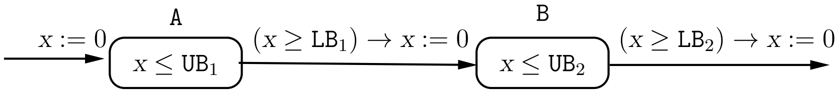
on delays determine the possible executions of this composite process. For example, if the upper bound $\mathtt{UB}_1$ is strictly smaller than the lower bound $\mathtt{LB}_2$, then in response to an initial input event, the first component is guaranteed to produce its output *before* the second component produces its output. That is, a mode-switch from the mode `EE` to the mode `FF` is guaranteed to be followed by the mode-switch to the mode `EF` since the guard condition $(y_2 \geq \mathtt{LB}_2)$ cannot be satisfied before the clock invariant $(y_1 \leq \mathtt{UB}_1)$ gets violated. Similarly, if the upper bound $\mathtt{UB}_2$ is strictly smaller than the lower bound $\mathtt{LB}_1$, then a mode-switch from the mode `EE` to the mode `FF` is guaranteed to be followed by the mode-switch to the mode `FE` corresponding to the output by the second component. If the intervals $[\mathtt{LB}_1, \mathtt{UB}_1]$ and $[\mathtt{LB}_2, \mathtt{UB}_2]$ overlap, then following a mode-switch from the mode `EE` to the mode `FF`, both scenarios, the first process producing its output before the second, and vice versa, are feasible. The goal of the timing analysis, to be discussed in section 7.3, is to discover which event sequences are consistent with the timing constraints.

Exercise 7.4: Figure 7.6 shows the *product* extended-state machine that captures the behavior of the composition of two instances of the timed process `TimedBuf` shown in figure 7.5. Now consider the composition of two instances of the timed process `TimedBuf` connected in series as shown in figure 7.7. Draw the extended-state machine with four modes and two clocks that captures the behavior of this composite process. ■

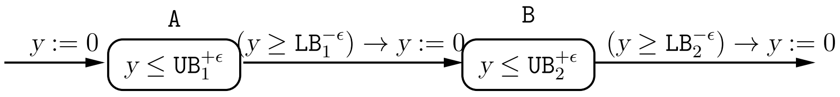
Exercise 7.5: For the timed process `TimedInc` of figure 7.4, argue that both the properties $(x_1 \leq 2x_2 + 2)$ and $(x_2 \leq 2x_1 + 2)$ are invariants of the system. ■

7.1.6 Modeling Imperfect Clocks *

In our model of timed processes, the value of a clock variable increases to accurately capture the amount of time elapsed. Now consider a timed process P that has a clock variable x that can measure time only imperfectly. The error is specified using a per-unit drift, say 0.01. This means that if the value of the clock x increases by 1, then the actual time elapsed may be any value in the interval $[0.99, 1.01]$. In general, if the drift is ϵ and the process P resets the clock x to 0 at time t and finds the constraint $\mathtt{LB} \leq x \leq \mathtt{UB}$ to be satisfied at a later time instance t', then it can conclude that the elapsed time $t' - t$ is at least $\mathtt{LB}(1-\epsilon)$ and at most $\mathtt{UB}(1+\epsilon)$.


A
B
$x := 0$
$x \leq \texttt{UB}_1$
$(x \geq \texttt{LB}_1) \rightarrow x := 0$
$x \leq \texttt{UB}_2$
$(x \geq \texttt{LB}_2) \rightarrow x := 0$

Timed process P with an imperfect clock x with drift ϵ


A
B
$y := 0$
$y \leq \texttt{UB}_1^{+\epsilon}$
$(y \geq \texttt{LB}_1^{-\epsilon}) \rightarrow y := 0$
$y \leq \texttt{UB}_2^{+\epsilon}$
$(y \geq \texttt{LB}_2^{-\epsilon}) \rightarrow y := 0$

Equivalent timed process P' with (perfect) clock y

Figure 7.8: Simulating an Imperfect Clock with a Drift by a Perfect Clock

Although our basic model does not allow modeling of such imperfect clocks explicitly, we can capture the resulting errors by changing the timing constraints. As an example, consider the timed process P shown in figure 7.8 that measures time using an imperfect clock x with drift ϵ. The clock-invariants associated with the modes and the guards on the mode-switches imply that the process spends between $\texttt{LB}_1$ and $\texttt{UB}_1$ time units in the mode `A` and between $\texttt{LB}_2$ and $\texttt{UB}_2$ time units in the mode `B`, as measured according to its imperfect clock x. We can capture the same behavior by the timed process P' that uses a perfect clock y. The clock-invariants associated with the modes and the guards of the mode-switches are modified by scaling upper bounds upward by a factor of ϵ and scaling lower bounds downward by a factor of ϵ. In figure 7.8, we abbreviate $\texttt{LB}(1-\epsilon)$ by $\texttt{LB}^{-\epsilon}$ and $\texttt{UB}(1+\epsilon)$ by $\texttt{UB}^{+\epsilon}$. The timing constraints of the process P' imply that the process spends between $\texttt{LB}_1(1-\epsilon)$ and $\texttt{UB}_1(1+\epsilon)$ time units in the mode `A` and between $\texttt{LB}_2(1-\epsilon)$ and $\texttt{UB}_2(1+\epsilon)$ time units in the mode `B`. As a result, the possible interactions of the process P with an imperfect clock and that of the process P' with a perfect clock with other processes are the same.

Exercise 7.6 : We have argued that, in figure 7.8, the timing behavior of the timed process P with an imperfect clock x with drift ϵ and the timed process P' with a perfect clock y with modified clock-invariants and guards are equivalent. Suppose in figure 7.8 we remove the resetting of the clock to 0 on the mode-switch from the mode `A` to the mode `B`. That is, consider the processes P and P' with the updates $x := 0$ and $y := 0$, respectively, on the mode-switch from the mode `A` to the mode `B` omitted. Are the resulting processes still equivalent in terms of the relative timings of when the mode-switches can occur? ■

7.2 Timing-Based Protocols

In this section, we illustrate the formal design of timed systems using three case studies: achieving distributed coordination by relying on timing assumptions,

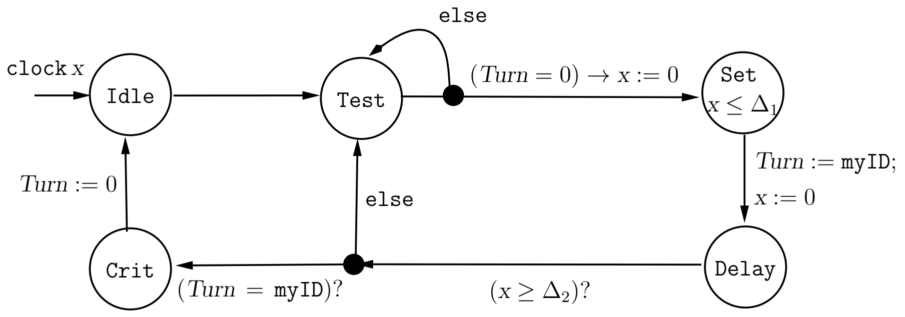


Figure 7.9: Timing-based Mutual Exclusion

achieving reliable transmission in presence of imperfect timing measurements, and design of a pacemaker to deliver timely pulse to the heart to maintain its rhythmic pulsation.

7.2.1 Timing-Based Distributed Coordination

In chapter 4, we studied how to solve problems that require coordination among distributed processes in the asynchronous model of computation, that is, without making any assumptions about the relative speeds of the participating processes. Now we turn our attention to solving such problems in the timed model of computation, where timing assumptions about the relative speeds of concurrent processes can be relied on to design solutions. In chapter 4, we established that there is no solution to the consensus problem if we restrict shared variables to atomic registers. However, if we assume that delays between successive steps of a process are bounded, then the knowledge of these bounds can be used to solve consensus using only atomic registers. Below we describe a timing-based solution to the classical coordination problem of *mutual exclusion*, and the same ideas can be used to solve consensus.

Recall the mutual exclusion problem aimed at allowing asynchronous processes to access a critical shared resource in a safe manner. The allocation of the resource is not governed by a central coordinator, but processes need to coordinate among themselves using atomic registers. The safety requirement is mutual exclusion: no two processes should be in the critical section simultaneously, and the liveness requirement is deadlock freedom: if some process wants to enter the critical section, then some process should be allowed to enter the critical section.

The timing-based solution, known as the Fischer's protocol, uses a single shared register *Turn*, and each process, with identifier `myID`, executes the timed state machine shown in figure 7.9. Initially, the register *Turn* is 0 and the mode is `Idle`. There is no clock invariant associated with the initial mode `Idle`, and thus the process may spend an arbitrary amount of time in this mode. When the process wants to enter the critical section, it switches to the mode `Test`. Then it reads the shared register *Turn*: if *Turn* equals 0, then the process proceeds to the mode `Set`, or else returns to the mode `Test` in order to read the shared register again.

In the mode `Set`, the next step of the process is to update the value of the shared register *Turn* to its own identifier. Observe that if a process P' tests *Turn after* a process P has set *Turn*, then the process P' will have to wait in the mode `Test`. However, if two processes test *Turn* before either has set it, then the protocol needs to resolve contention between them to allow only one to proceed to the critical section. The protocol assumes that writing to the shared register *Turn* takes at most Δ_1 time units. This is specified using the clock variable x and the invariant $(x \leq \Delta_1)$ associated with the mode `Set`. After updating *Turn*, a process waits in the mode `Check` for at least Δ_2 time units; this delay is ensured by the guard $(x \geq \Delta_2)$ on the mode-switch out of `Delay`. When a process leaves the mode `Delay`, it reads the shared register *Turn* again. At this time, assuming $\Delta_2 > \Delta_1$, we can conclude that all the processes that tested *Turn* to be zero, and thus proceeded to set the register, have finished their updates. If a process finds *Turn* unchanged, that is, equal to its own identifier, it proceeds to the critical section, but if it finds that the value has been overwritten by some other process, then it retries by switching back to the mode `Test`. Once in the critical section, that is, in the mode `Crit`, a process can spend an arbitrary amount of time, and upon exit, it resets *Turn* to 0 and returns to the initial mode.

To prove mutual exclusion, consider the sequence of events in a typical execution depicted in figure 7.10. Let t_1 be the time when the process P leaves the mode `Test` and enters the mode `Set`, that is, the time when the process reads the register *Turn* and finds it to be 0. Let t_2 be the time when it enters the mode `Delay`, that is, the time when it sets the register *Turn* to its own identifier. Let t_3 be the time when the process P leaves the mode `Delay` and reads the value of the register *Turn* again. The timing constraints ensure that the condition $t_3 - t_2 \geq \Delta_2$ holds. It is possible that some other process P' attempting to enter the critical section also reads the shared register *Turn* to be 0 sometime during the time interval from t_1 to t_2, say at time t'_1 (see figure 7.10). We don't need to worry about processes that had not entered the mode `Set` before time t_2 since the value of the register *Turn* is guaranteed to be non-zero at all times from t_2 onward, and thus each such process would read a non-zero value from the register *Turn*. Due to the upper bound on time a process spends in the mode `Set`, the process P' must execute the mode-switch from the mode `Set` to the mode `Delay` sometime during the interval $[t'_1, t'_2]$, where $t'_2 = t'_1 + \Delta_1$. Now it is easy to see that if $\Delta_2 > \Delta_1$, then $t'_2 < t_3$, meaning that any competing process that switched from the mode `Test` to the mode `Set` during the interval $[t_1, t_2]$ would have left the mode `Set` by time t_3. In our illustrative scenario of figure 7.10, the process P modifies the register *Turn* to its identifier at time t_2, and the process P' modifies the register *Turn* to its own identifier at some time after time t'_1 and before time t'_2. If this write by the process P' occurs before

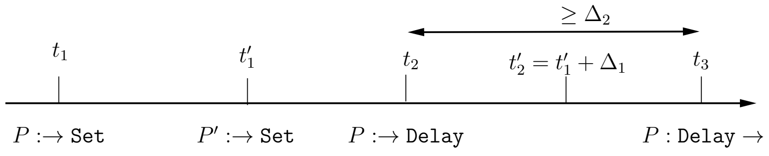


Figure 7.10: Illustrating a Timed Execution of the Mutual-exclusion Protocol

time t_2, then this update is overwritten by that of the process P, and if it occurs after time t_2, then it overwrites the update by the process P. In either case, the test ($Turn = \texttt{myID}$) cannot succeed for both, and it is not possible for both to proceed to the critical section.

In general, among all the competing processes that succeed in entering the mode `Set`, if the process P' is the last process to set the register *Turn* to its identifier, then every process will find the register *Turn* to be equal to the identifier of the process P' when it checks the register *Turn* when it leaves the mode `Delay`. Such a process P' will be the unique winner and will proceed to its critical section. When it leaves the critical section, it sets the register *Turn* back to 0 so that processes waiting in the mode `Test` can compete again to enter the critical section.

From the discussion above, for $\Delta_2 > \Delta_1$, it follows that the protocol satisfies both mutual exclusion and deadlock freedom. More precisely, consider the timed process obtained by composing arbitrarily many instantiations of the process P of figure 7.9, each with its own value of the identifier `myID` and the process modeling the atomic register *Turn*. For the composite process, the mutual exclusion property

$$\varphi_{me} : \ \neg\,(P.mode = \texttt{Crit} \ \wedge \ P'.mode = \texttt{Crit})$$

is an invariant, for every pair P and P' of processes. The composite process also satisfies the deadlock freedom property: if $P.mode = \texttt{Test}$ for a process P, then eventually $P'.mode = \texttt{Crit}$ for some process P'.

Exercise 7.7: For the timing-based mutual exclusion protocol of figure 7.9, consider the starvation-freedom requirement “if a process P enters the mode `Test`, then it will eventually enter the mode `Crit`.” Does the system satisfy the starvation-freedom requirement? If not, show a counter-example. ■

Exercise 7.8: Describe a protocol for solving the consensus problem described in section 4.3.3 using atomic registers and timing assumptions. State the timing assumptions explicitly. Describe the protocol in the state machine notation (using the mutual exclusion protocol of figure 7.9 as a guide). Argue why the protocol meets all three requirements of the consensus problem. ■

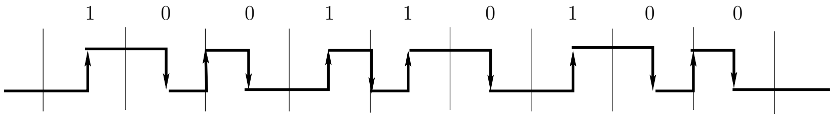


Figure 7.11: Manchester Encoding of the Bit Sequence 100110100

7.2.2 Audio Control Protocol*

We now consider a timing-based protocol for transferring a sequence of bits from the sender to the receiver using imperfect clocks. The encoding used in this protocol is called the Manchester encoding, and this protocol is based on an audio control protocol used by Philips Inc.

Problem Description

A stream of bits, that is, values of Boolean type, is communicated using high- and low-voltage settings on a communication bus. Time is divided into slots of fixed length, and in each slot, a single bit is communicated by changing the voltage in the middle of the slot. The value 0 is encoded as a falling edge from high voltage to low voltage, and the value 1 is encoded as a rising edge from low voltage to high voltage. If the bits to be sent in consecutive slots are the same, then there must be an intermediate change in the voltage, and this happens at the end of a slot. The voltage pulse corresponding to the encoding for the bit sequence 100110100 is shown in figure 7.11.

The clocks of the sender and the receiver are imperfect and have a specified drift. The receiver does not know when the first time slot begins, but both the sender and the receiver know the agreed-on width of the slots. The sender and the receiver synchronize the beginning of the transmission by requiring low voltage when no information is exchanged and by agreeing that each message begins with the bit 1. The receiver does not know the length of the message in advance but can infer the end of the current message when it detects that no information has been communicated during a slot. A challenge for the protocol designer is the constraint that the receiver cannot reliably detect a falling edge. Thus, all decoding must be inferred based purely on the relative timings of the rising edges. As a result, the receiver cannot resolve the ambiguity between messages ending in 10 and in 1. This is because even when the message ends with 1, the sender sets the voltage to low to ensure that the voltage is low when no information is being transmitted. The delay between the last falling edge and the preceding rising edge is a full time slot for messages ending with 10 and is only a half time slot for messages ending with 1. Since the receiver cannot detect a falling edge, it cannot differentiate between these two cases. To resolve this ambiguity, it is assumed that each message ends in 00.

For the design and analysis of the desired protocol, let us assume that the length of the time slot is four time units and the drift of the clocks of the sender and

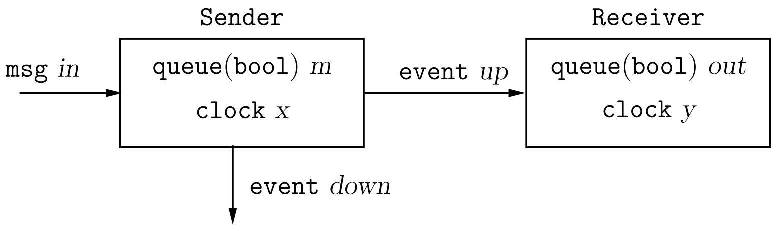


Figure 7.12: Block Diagram for the Audio Control Protocol

the receiver is ϵ per time unit. What this means is that if the sender keeps the voltage high for four time units according to its internal clock, then the actual elapsed time may be anywhere between $4 - 4\epsilon$ and $4 + 4\epsilon$ time units. Similarly, if the receiver finds the duration between two consecutive rising edges to be less than three time units, then it can only assume that the corresponding actual delay is less than $3 + 3\epsilon$ time units.

The correctness requirement is that every message is correctly decoded by the receiver, assuming that the drift in the clock values is bounded by a given ϵ per time unit. More generally, once we design a protocol, we would like to determine the largest value of the drift rate ϵ for which the protocol works correctly.

The Sender Process

The block diagram for the system composed of the sender and the receiver is shown in figure 7.12. The sender process receives the message to be transferred on the input channel *in*. A single message is a sequence of Boolean values. The sender process uses an internal queue *m* to store the sequence of bits to be transferred. We model the rising and falling edges of the voltage as output events *up* and *down*, respectively. The clock variable *x* is used to specify the timing constraints. The state machine for the sender process is shown in figure 7.13.

The process starts in the mode `A`, where it is waiting to receive the input. When the input is received on the channel *in*, it is stored in the queue *m*. The first bit is immediately dequeued for transfer, and it is assumed to be 1 (it is easy to modify the state machine for the sender so that it checks if the first bit is 1 and enters an error state if the check fails). The process sets its clock variable *x* to 0 and switches to the mode `B`. After waiting for two time units, it issues the event *up* to change the voltage to high in the middle of the slot. The process removes the next bit to be transmitted from the queue *m*. If this bit is 1, then it switches to the mode `C`, where it waits for two time units, changes the voltage to low by issuing the event *down*, and then returns to the mode `B` in order to transmit 1 in the following time slot. If the bit is 0, then it switches to the mode `D`. In mode `D`, it waits for four time units till the middle of the next time slot, and then changes the voltage to low. It then examines the next bit from the

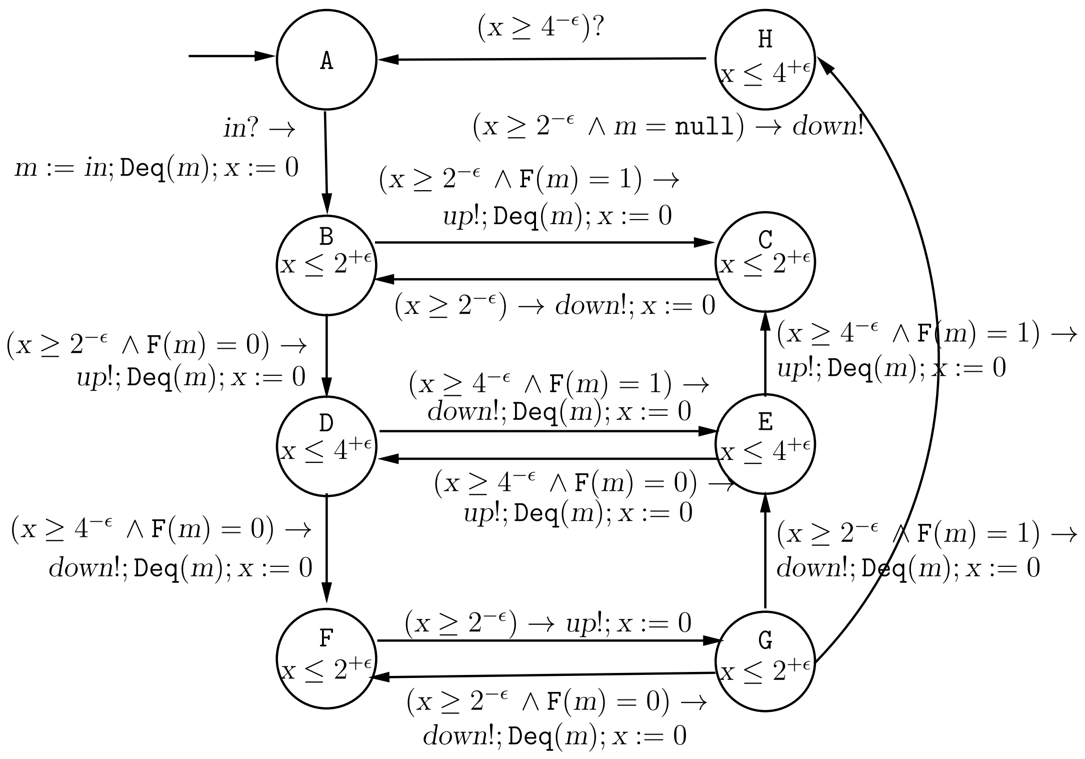

A
$(x \geq 4^{-\epsilon})?$
H
$x \leq 4^{+\epsilon}$
$in? \rightarrow$
$m := in; \texttt{Deq}(m); x := 0$
$(x \geq 2^{-\epsilon} \wedge m = \texttt{null}) \rightarrow down!$
$(x \geq 2^{-\epsilon} \wedge \texttt{F}(m) = 1) \rightarrow$
$up!; \texttt{Deq}(m); x := 0$
B
$x \leq 2^{+\epsilon}$
C
$x \leq 2^{+\epsilon}$
$(x \geq 2^{-\epsilon}) \rightarrow down!; x := 0$
$(x \geq 2^{-\epsilon} \wedge \texttt{F}(m) = 0) \rightarrow$
$up!; \texttt{Deq}(m); x := 0$
$(x \geq 4^{-\epsilon} \wedge \texttt{F}(m) = 1) \rightarrow$
$up!; \texttt{Deq}(m); x := 0$
$(x \geq 4^{-\epsilon} \wedge \texttt{F}(m) = 1) \rightarrow$
$down!; \texttt{Deq}(m); x := 0$
D
$x \leq 4^{+\epsilon}$
E
$x \leq 4^{+\epsilon}$
$(x \geq 4^{-\epsilon} \wedge \texttt{F}(m) = 0) \rightarrow$
$up!; \texttt{Deq}(m); x := 0$
$(x \geq 4^{-\epsilon} \wedge \texttt{F}(m) = 0) \rightarrow$
$down!; \texttt{Deq}(m); x := 0$
$(x \geq 2^{-\epsilon} \wedge \texttt{F}(m) = 1) \rightarrow$
$down!; \texttt{Deq}(m); x := 0$
$(x \geq 2^{-\epsilon}) \rightarrow up!; x := 0$
F
$x \leq 2^{+\epsilon}$
G
$x \leq 2^{+\epsilon}$
$(x \geq 2^{-\epsilon} \wedge \texttt{F}(m) = 0) \rightarrow$
$down!; \texttt{Deq}(m); x := 0$

Figure 7.13: The Sender Process in the Audio Control Protocol

message queue. If this bit is 1 (different from the last bit processed), it switches to the mode E, where it waits for four time units before issuing the event *up*. If this bit is 0 (same as the last bit processed), then it waits in the mode F for two time units, raises the voltage, waits in the mode G for two time units, and then lowers it again to send the 0 bit. Each time the next bit is removed from the message queue and decisions are made based on whether the next bit is the same or different compared to the bit most recently sent. The last two bits of the message are guaranteed to be 00. The process will be in the mode G when the message ends, and when the queue m is empty, it returns to the idle location A after waiting in the mode H for a duration of a time slot without changing the output.

The detailed state machine for the sender appears in figure 7.13. In the description $\texttt{F}(m)$ stands for the first element of the queue m, and the action $\texttt{Deq}(m)$ removes the first element from the queue m. Note that all timing constraints are modified to reflect the possible errors in the measurement of time, as explained in section 7.1.6. For example, to specify that the process waits in the mode H for four time units we need to associate the clock-invariant $(x \leq 4)$ with the

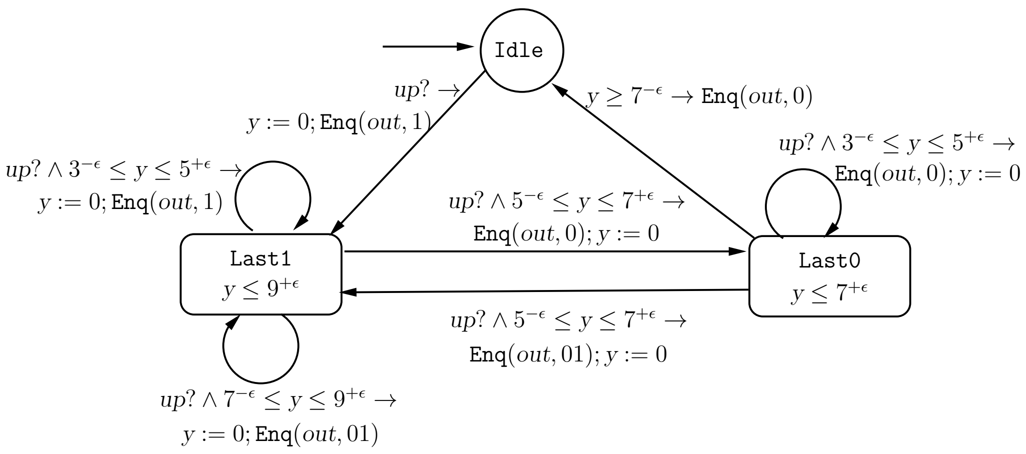

Idle
$up? \rightarrow$
$y := 0; \texttt{Enq}(out, 1)$
$y \geq 7^{-\epsilon} \rightarrow \texttt{Enq}(out, 0)$
$up? \wedge 3^{-\epsilon} \leq y \leq 5^{+\epsilon} \rightarrow$
$y := 0; \texttt{Enq}(out, 1)$
$up? \wedge 3^{-\epsilon} \leq y \leq 5^{+\epsilon} \rightarrow$
$\texttt{Enq}(out, 0); y := 0$
$up? \wedge 5^{-\epsilon} \leq y \leq 7^{+\epsilon} \rightarrow$
$\texttt{Enq}(out, 0); y := 0$
Last1
$y \leq 9^{+\epsilon}$
Last0
$y \leq 7^{+\epsilon}$
$up? \wedge 5^{-\epsilon} \leq y \leq 7^{+\epsilon} \rightarrow$
$\texttt{Enq}(out, 01); y := 0$
$up? \wedge 7^{-\epsilon} \leq y \leq 9^{+\epsilon} \rightarrow$
$y := 0; \texttt{Enq}(out, 01)$

Figure 7.14: The Receiver Process in the Audio Control Protocol

mode `H`, and associate the guard $(x \geq 4)$ with the mode-switch from the mode `H` to the mode `A`. To capture the errors in time measurement, the clock-invariant $(x \leq 4)$ is changed to the constraint $(x \leq 4 + 4\epsilon)$, and the guard $(x \geq 4)$ is changed to the constraint $(x \geq 4 - 4\epsilon)$. We use $4^{-\epsilon}$ as an abbreviation for $4 - 4\epsilon$ and $4^{+\epsilon}$ for $4 + 4\epsilon$.

The Receiver Process

The receiver process is shown in figure 7.14. The receiver uses a clock variable y and an output buffer *out* to store the decoded message. It starts in the mode `Idle`. When it receives the first *up* event, it initiates the message to be 1. The mode `Last1` corresponds to the case that the last decoded bit is 1; analogously, the mode `Last0` corresponds to the case that the last decoded bit is 0. The clock y is used to measure the duration between successive *up* events. In the mode `Last1`, if the next bit is 1, then the exact duration until the next event is expected to be 4. Since the receiver simply needs to distinguish among various cases, if the duration is any time between 3 and 5, then it considers the next bit to be 1. The delay is measured using the receiver's imperfect clock. The check $(3^{-\epsilon} \leq y \leq 5^{+\epsilon})$ is an abbreviation for the check $(3 - 3\epsilon \leq y \leq 5 + 5\epsilon)$, and this accounts for the fact that the receiver's clock has a potential drift of ϵ per time unit compared to the physical elapsed time. In the mode `Last1`, if the next rising edge is detected after a delay in the interval $[5, 7]$, then the bit 0 is sent, and if the next rising edge is detected after a delay in the interval $[7, 9]$, then the bits 0 and 1 were transmitted (no rising edge is required for a 0 sandwiched between two 1s). In the mode `Last0`, similar logic is applied by partitioning the expected delays until the next *up* event into different categories: a delay between 3 and 5 means the next bit is 0, a delay between 5 and 7 means that

Time	Event	x	Sender	Queue m	y	Receiver	Queue *out*
0			`B`	00110100		`Idle`	`null`
2.07	*up*	2.07	`D`	0110100		`Last1`	1
5.97	*down*	3.9	`F`	110100	3.9	`Last1`	1
7.97	*up*	2	`G`	110100	5.9	`Last0`	10
9.92	*down*	1.95	`E`	10100	1.95	`Last0`	10
14.08	*up*	4.16	`C`	0100	6.11	`Last1`	1001
16.1	*down*	2.02	`B`	0100	2.02	`Last1`	1001
18	*up*	1.9	`D`	100	3.92	`Last1`	10011
22.05	*down*	4.05	`E`	00	4.05	`Last1`	10011
25.91	*up*	3.86	`D`	0	7.91	`Last1`	1001101
30.01	*down*	4.1	`F`	`null`	4.1	`Last1`	1001101
32.11	*up*	2.1	`G`	`null`	6.2	`Last0`	10011010
34.16	*down*	2.05	`H`	`null`	2.05	`Last0`	10011010
38.29		4.13	`A`	`null`	6.18	`Last0`	10011010
39.39		1.1	`A`	`null`	7.28	`Idle`	100110100

Figure 7.15: An Execution of the Audio Control Protocol

bits 0 and 1 are sent, and if no event is detected for seven time units, then the receiver concludes that the transmission has ended (recall that the message ends with 0) and returns to the mode `Idle`.

Example Execution

Figure 7.15 shows a possible execution of the protocol when the message string 100110100 is supplied to the sender at time 0, where the error rate ϵ equals 0.05. Then at time 0, the sender switches to the mode `B`, setting its variable m to 00110100 and its clock x to 0. Each row in the table shows the time at which the transition occurs, the event issued by the sender during this transition, the value of the clock variable x of the sender at the time of the transition (before it gets updated), the mode of the sender after the transition, the value of the internal message queue m after the transition, the value of the clock variable y of the receiver at the time of the transition (before it gets updated), the mode of the receiver after the transition, and the value of the output message queue *out* after the transition. Note that at the end of this execution, both processes have returned to their respective initial modes, and the value of the output queue equals the original input message 100110100.

Analysis

The parallel composition of the timed processes for the sender and the receiver can be analyzed to check whether the protocol works correctly, that is, whether the message received on the channel *in* by the sender equals the final value of the buffer *out*. This requirement can be captured by a safety monitor. We would also like to find out what is the maximum value of the error rate ϵ for which the protocol works correctly. It turns out the industrial design by Philips allowed an error of 5% (that is, $\epsilon = 1/20$), and the protocol meets the correctness requirement for this error rate. A formal analysis using model checking tools such as HyTech and Uppaal established that the protocol is resilient for errors upto $\epsilon = 1/15$.

Exercise 7.9: Demonstrate that the audio control protocol cannot tolerate the error rate $\epsilon = 0.25$ by showing an incorrect execution corresponding to the input string 100110100. ■

7.2.3 Dual Chamber Implantable Pacemaker

The design and implementation of software for medical devices is challenging due to their rapidly increasing functionality and the tight coupling of computation, control, and communication. The safety-critical nature of such devices make them an ideal domain for exploring applications of formal modeling and analysis. In this section, we use a dual chamber implantable pacemaker to illustrate the modeling of control software and specification of correctness requirements for such devices. We begin with an overview of the basic functionality of a pacemaker.

Pacemaker Basics

The human heart is an excellent example of a naturally occurring timed system. It spontaneously generates electrical impulses that organize the sequence of muscle contractions during each heart beat. The underlying timing pattern of these impulses is key to the proper functioning of the heart. The implantable cardiac pacemaker is a rhythm management device that monitors these patterns and corrects them via external means when needed.

Controlled by the nervous system, a specialized tissue, called the *SinoAtrial node*, at the top of the right atrium periodically generates electrical pulses. These pulses cause both atria to contract, forcing blood into the ventricles. The electrical conduction gets delayed at the *AtrioVentricular node*, allowing the ventricles to fill fully, but then spreads rapidly across the ventricular muscles, resulting in their coordinated contraction that pumps the blood out of the heart.

A common heart disease, called *bradycardia*, is due to failures in either impulse generation or impulse propagation and results in slow heart rate, leading to insufficient pumping of blood. Bradycardia can be treated by an implantable pacemaker that monitors the heart rate and delivers timely external electrical pulses to maintain an appropriate heart rate as well as atrio-ventricular coordination. Such a pacemaker usually has two leads fixed on the wall of the right atrium and the right ventricle. Activation of local tissue is sensed by these leads, and these sensing events act as inputs to the pacemaker. If these sensed events

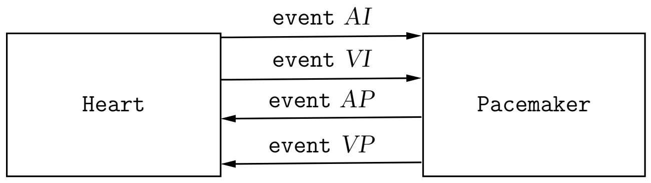


Figure 7.16: Interaction between the Heart and the Pacemaker

do not occur in a timely manner, then the pacemaker responds by producing pacing events that trigger electrical stimuli to the heart.

A modern pacemaker responds to a variety of heart conditions and can operate in different modes. We focus on a mode called DDD: the first character describes the pacing locations and D means that the pacemaker is pacing both the atrium and the ventricle, the second character describes the sensing locations and D means that both chambers are being sensed, and the third character specifies how the pacemaker software responds to sensing and D means that sensing can both activate or inhibit further pacing. While models corresponding to other commonly used modes (for instance, the VDI mode in which the pacemaker paces only the ventricle, senses both chambers, and sensing causes inhibition of pacing) are similar, the decision logic for switching from one mode to another causes additional complexity for the pacemaker software and is not reflected in our model for the DDD mode.

Design Overview

Figure 7.16 shows the top-level block diagram for communication between the two timed processes: the plant process `Heart` and the controller process `Pacemaker`. The events *AI* and *VI* are inputs to the pacemaker, while its outputs correspond to the events *AP* and *VP*. The event *AI* is sensed by the lead placed in the wall of the right atrium and corresponds to electrical potential exceeding a specific threshold value. The event *VI* is analogous and denotes electrical activation in the right ventricle. The pacing events *AP* and *VP* induce contractions of the muscles of the atrium and the ventricle, respectively. Note that all communication variables are modeled as events and thus have no associated data values. The behavior of the pacemaker depends on the timing delays between these events.

The design of the pacemaker is composed of four processes as shown in figure 7.17. The event *VI* denotes raw sensory input indicating electrical activity in the ventricle. The timed process `FilterV` outputs the event *VS*: the sequence of *VS* events corresponds to a filtered version of the sequence of *VI* events, and these events are used for deciding when to produce the pacing events. Similarly, the timed process `FilterA` outputs the event *AS*, a filtered version of the raw sensory event *AI*. The processes `PaceA` and `PaceV` produce the pacing events *AP*

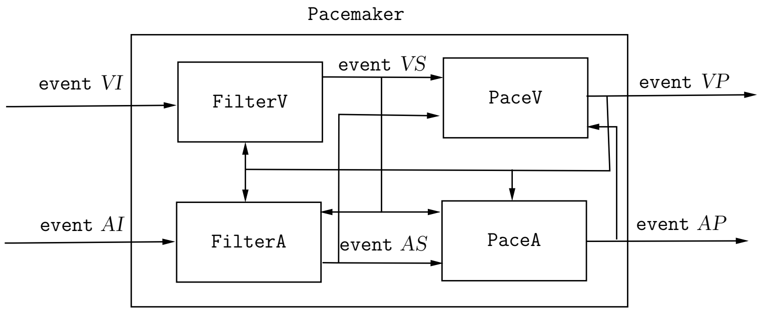


Figure 7.17: Block Diagram for the Pacemaker Subcomponents

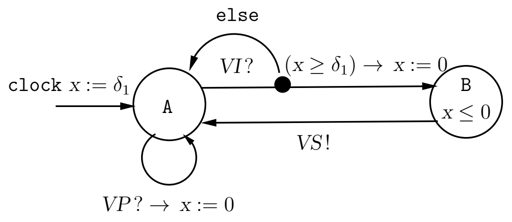
and *VP*, respectively, based on the timing pattern of the filtered sense events *AS* and *VS*. The inputs to each of these four subprocesses are as shown in figure 7.17, and we proceed to explain their designs.

Event Sensing

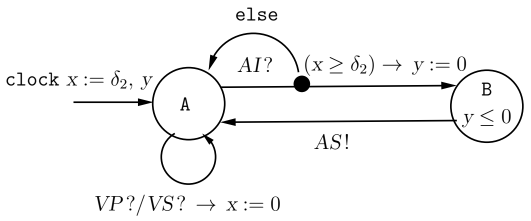
The timed process `FilterV` of figure 7.18 filters the sequence of *VI* events to remove noise. After each ventricular event, there is a blanking period, called *Ventricular Refractory Period* (VRP), during which additional ventricular events are not considered to be new activity and, hence, ignored. The programmable parameter δ_1 in the description of the process `FilterV` corresponds to VRP, and a typical value of this parameter is 100 ms.

The process `FilterV` uses a clock variable x to measure the timing delay since the last ventricular event. When the process receives an input event *VI*, if the clock x has not exceeded δ_1, then it means that the VRP has not elapsed, and no output is produced. If the clock x exceeds δ_1, then the process wants to immediately respond by producing the event *VS* and by resetting the clock x to mark the beginning of a new VRP. Note that the underlying formal model is that of asynchronous processes, and thus the output transition producing the event *VS* has to be decoupled from the input transition that receives the event *VI*. To ensure that this output transition immediately follows the input transition without a delay, on receiving the input, the process sets the clock x to 0 and switches to the mode `B` with the associated clock-invariant ($x \leq 0$). This ensures that this mode `B` is transient, and the system cannot spend a nonzero amount of time while the process `FilterV` is in this mode. Note that the clock x is reset to 0 also when the ventricular pacing event *VP* is received, and this ensures that an input event *VI* within VRP of such a pacing event will be ignored.

The process `FilterA` is responsible for filtering the sequence of sensed atrial events. Its functioning is defined by the following rule: an input event *AI* should be viewed as the filtered event *AS* if it is not within the *Post Ventricular Atrial*



Process `FilterV`



Process `FilterA`

Figure 7.18: Timed Processes for Filtering Sense Events

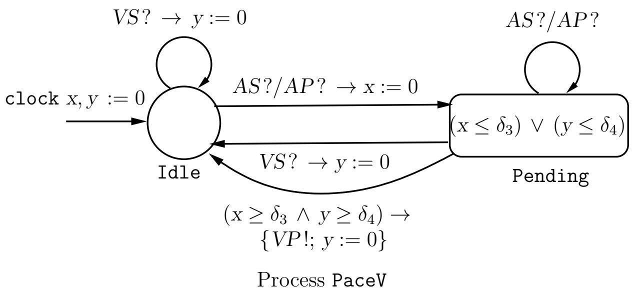
Refractory Period (PVARP) since the last ventricular event. The parameter δ_2 in the description of the process `FilterA` corresponds to this PVARP, and its typical value is 100 *ms*.

The clock x of the process `FilterA` measures the delay since the last ventricular event and is reset to 0 whenever either of the ventricular input events VS or VP is received. When the event AI occurs, if the value of x is below the threshold δ_2, then it does not result in any output event. Otherwise the process responds by producing the output event AS without any delay. To ensure that no time elapses between the reception of AI and production of AS, it sets a clock variable y to 0 when the input is received and switches to the transient mode `B` that has the associated clock-invariant $(y \leq 0)$ and a single outgoing transition that produces the desired output.

Timing of Pacing Events

The timed processes `PaceA` and `PaceV` shown in figure 7.19 implement the basic functionality of the pacemaker to keep the heart rate above a basic minimum.

The function of the process `PaceV` is to ensure that a ventricular event occurs within a maximum delay of *Atrio-Ventricular Interval* (AVI) since the last atrial event. The clock x is set to 0 when the event AS or AP occurs, and the process switches to the mode `Pending`. Note that the mode-switch $AS?/AP? \to x := 0$ is a short-hand for two mode-switches one triggered by the condition $AS?$ and one triggered by the condition $AP?$. The parameter δ_3 corresponds to AVI, and



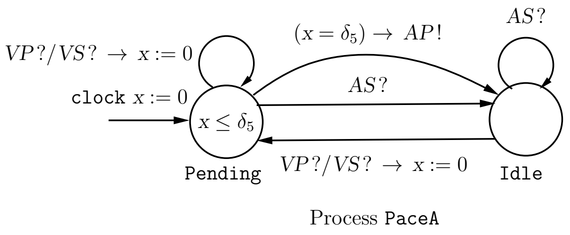


Figure 7.19: Timed Processes for Generating Pacing Events

its typical value is 150 *ms*. If the ventricular sensing event *VS* does not occur before the clock x exceeds this threshold δ_3 while waiting in the mode `Pending`, then the process should issue the pacing event *VP*. However, in order to prevent the pacemaker from pacing the ventricle too fast, the pacing event is issued only when at least *Upper Rate Interval* (URI) time has elapsed since the most recent ventricular event. To capture this constraint, the process uses another clock y that is reset to 0 on every ventricular event. The parameter δ_4 corresponds to URI that enforces a lower bound on the times between consecutive ventricular events, and its typical value is 400 *ms*. The condition to issue the pacing event *VP* is the conjunction $(x \geq \delta_3) \wedge (y \geq \delta_4)$. The disjunctive clock-invariant associated with the mode `Pending` is the negation of this guard and ensures that when both time limits expire, the pacemaker responds by pacing.

The process `PaceA` encodes the logic to generate the atrial pacing event *AP*. In the specification of a pacemaker, *Lower Rate Interval* (LRI) refers to the longest allowed interval between two ventricular events. In the initial mode `Pending`, if the process does not receive a ventricular event or the atrial sensing event *AS* within δ_5 time units since the previous ventricular event, then it issues the atrial pacing event *AP*. The value of the parameter δ_5 is chosen to be the difference LRI − AVI (with the assumption that the process `PaceV` paces the ventricle after a delay of AVI time units with respect to an atrial event). With an atrial event, the process `PaceA` switches to the mode `Idle` and switches back on the subsequent ventricular event. A typical value of LRI is 1000 *ms*.

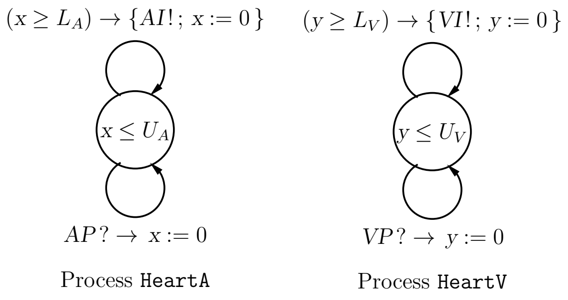


Figure 7.20: A Nondeterministic Model of the Heart

Heart Modeling

To analyze the functioning of the pacemaker, we need a model of the heart that generates the sensory events *AI* and *VI*. Figure 7.20 shows such an abstract model. The timed process `HeartA` generates the atrial sensing event *AI* after a nondeterministically chosen delay in the interval $[L_A, U_A]$ since the previous atrial event. The timed process `HeartV` is symmetric and generates the ventricular event *VI* with a nondeterministic delay in the interval $[L_V, U_V]$. The process `Heart` is the parallel composition of the timed processes `HeartA` and `HeartV`.

For establishing basic safety requirements of the pacemaker design, the simplified models in figure 7.20 suffice. However, this modeling does not capture the corelation in the timings of the atrial and ventricular events. A more faithful model can be constructed as a composition of processes, some capturing nodes in the heart tissue and some corresponding to the conduction pathways connecting these nodes.

Illustrative Execution

Let us illustrate the behavior of the pacemaker using a sample execution shown in figure 7.21. At time t_1, the process `HeartA` outputs the atrial event *AI*. The pacemaker component `FilterA` considers this to be a new atrial event and generates the sensing event *AS* without any delay. Given the absence of a ventricular sensing event, the process `PaceV` generates the pacing event *VP* at time t_2, which is the maximum of δ_4 and $t_1 + \delta_3$ (the parameters δ_3 and δ_4 correspond to the periods AVI and URI, respectively).

Following the ventricular event at time t_2, the subsequent atrial pulsation is generated by the heart at time t_3. However, this event is ignored by the process `FilterA` since $t_3 < t_2 + \delta_2$, where the parameter δ_2 is set to PVARP.

Meanwhile, the process `PaceA` expects the subsequent atrial sensing within a period of δ_5 corresponding to the difference LRI $-$ AVI after the ventricular

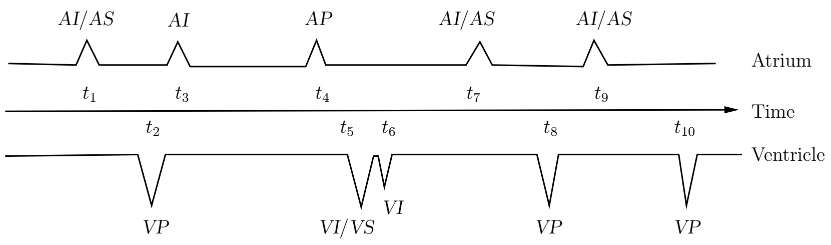


Figure 7.21: An Illustrative Execution of the Pacemaker

event at time t_2. Since such an event does not occur in this sample execution, it responds by generating the event *AP* at time $t_4 = t_2 + \delta_5$.

The heart generates the following ventricular event *VI* at time t_5, which gets mapped to the event *VS* by the filtering process `FilterV` without any delay. The subsequent ventricular puslation is generated by the heart at time t_6, and since $t_6 < t_5 + \delta_1$, where the parameter δ_1 corresponds to VRP, the event *VI* at time t_6 is ignored by the pacemaker. The next atrial event is generated by the heart at time t_7, and this event is translated to the atrial sensing event *AS* without any delay (assuming t_7 exceeds $t_5 + \delta_2$).

The process `PaceV` expects the subsequent ventricular event within δ_3 time units of the atrial event at time t_7. At time $t_8 = t_7 + \delta_3$, it turns out that $t_8 > t_5 + \delta_4$, implying that sufficient time (URI) has elapsed since the most recent *VS*, and hence the process `PaceV` generates the pacing event *VP*.

The following atrial event *AI* occurs at time t_9 and is translated to *AS* without any delay. Subsequently, the process `PaceV` expects a ventricular sensing before time $t_9 + \delta_3$. However, in this case, $t_9 + \delta_3 < t_8 + \delta_4$, implying that insufficient time has elapsed since it generated the most recent pacing event *VP*. As a result, it keeps waiting, and only at time $t_{10} = t_8 + \delta_4$ (which is greater than $t_9 + \delta_3$) it paces the heart again by generating the event *VP*

Requirements

The most basic functionality of a pacemaker is to treat bradycardia by maintaining the ventricular rate above the threshold of LRI (Lower Rate Interval). Thus, the closed-loop system consisting of the parallel composition of `Heart` and `Pacemaker` (see figure 7.16) should satisfy the following safety requirement: the delay between two successive ventricular events should not exceed LRI. This property cannot be directly captured as an invariant of the system, but we can use the timed process `LRIMonitor` of figure 7.22 as a safety monitor. The monitor observes the events *VS* and *VP* and enters the error state `E` if more than LRI time units have elapsed since the last occurrence of a ventricular event. Verifying safety of the pacemaker now corresponds to checking

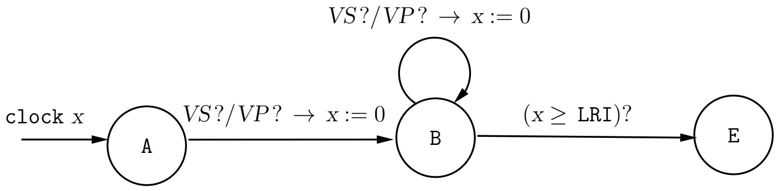


Figure 7.22: Safety Monitor for the Pacemaker

whether the property ($\texttt{LRIMonitor}.mode \neq E$) is an invariant of the system $\texttt{Heart} \,|\, \texttt{Pacemaker} \,|\, \texttt{LRIMonitor}$.

For the pacemaker model we have discussed, along with the specified values of the programmable delay parameters and for the nondeterministic heart model of figure 7.20, irrespective of the values of the parameters L_A, U_A, L_V, and U_V, it is the case that the pacemaker is safe (that is, the monitor `LRIMonitor` cannot enter the error mode). This can be established by a manual proof by identifying a suitable inductive invariant or by using a model checker such as Uppaal that implements the algorithmic reachability analysis discussed in section 7.3.

There are a number of other requirements that a pacemaker design is expected to satisfy. For example, a pacemaker should pace the ventricles beyond a maximum rate specified as the upper rate limit. A number of timing patterns of atrial and ventricular events are considered undesirable, and the pacemaker is expected to take a corrective action in response. While the modeling, specification, and analysis techniques we have discussed are suitable for the formalization of such requirements, we omit such detailed modeling from this case study.

Exercise 7.10: In the modeling of the timed process `PaceV` (see figure 7.19), we have used a *disjunctive* clock-invariant expression for the mode `Pending`. Construct an alternative model where the mode `Pending` is split into two modes, one with the clock-invariant ($x \leq \delta_3$) and one withe the clock-invariant ($y \leq \delta_4$), such that the resulting process has identical input/output behavior as the process `PaceV`. ■

7.3 Timed Automata

Given a timed process *TP*, which may be expressed as a parallel composition of a number of timed processes, including a safety monitor, and a property φ over the state variables, the goal of the reachability analysis is to check whether φ is an invariant of the process *TP* and, if not, produce a counter-example. A key obstacle in adapting the on-the-fly enumerative depth-first search algorithm for invariant verification (as studied in section 3.3) is the fact that the state of a timed process includes the values for the real-valued clock variables. As a result, we must develop symbolic or constraint-based techniques to handle sets of values for clocks. In this section, we present analysis techniques that are applicable for the most commonly occurring pattern in which clocks are used in the modeling of timed systems.

7.3.1 Model of Timed Automata

In chapter 3, we established that verification problems admit algorithmic solutions provided the system being analyzed is a finite-state system. A timed process is typically not a finite-state system due to the presence of clock variables. However, algorithmic analysis is possible if we restrict the way the clock variables are used. For such an analysis, the use of the clock variables is restricted in the following way. First, the only value assigned to a clock variable is 0. Second, the only tests involving the clock variables are of the form $x \leq k$ and $x \geq k$, for some integer constant k, that is, the only atomic expressions used in the clock-invariants and updates compare a clock variable to a constant. All the examples we have considered so far in this chapter obey these restrictions. Such updates and tests are adequate to express lower and upper bounds on the delays between events.

Timed processes with such restricted usage of clock variables are called *timed automata*. In particular, the timed process `TimedBuf` (see figure 7.2) is a timed automaton: the clock y is reset to 0 on the switch from the mode `Empty` to the mode `Full` and the test ($y \leq$ `UB`) in the clock-invariant of the mode `Full` and the test ($y \geq$ `LB`) in the guard of the mode-switch from the mode `Full` to the mode `Empty`, compare the clock variable with the constants corresponding to the upper and lower bounds on the delay, respectively.

Timed Automaton

A timed process *TP* is said to be a *timed automaton* if for every clock variable x of the process:

1. the only assignment to the variable x occurring in the update description of any of the tasks of the process *TP* is of the form $x := 0$; and

2. each atomic expression involving the variable x occurring either in the clock invariant of the process *TP*, or in the guards or update descriptions of any of the tasks of the process *TP*, is of the form $x \leq k$ or $x \geq k$, where k is an integer constant.

Note that a test of the form $(x = k)$, for an integer constant k, can be defined since it is equivalent to the conjunction $(x \leq k) \wedge (x \geq k)$, and so can be a test of the form $(x < k)x$, which is equivalent to $\neg\,(x \geq k)$.

Since the analysis treats clock variables in a special way, it is convenient to consider a state of a timed automaton *TP* as a pair (s, ν), where the *discrete-state* s assigns values to all the discrete variables, and the *clock-valuation* ν assigns values to the clock variables. Note that if a timed automaton has n

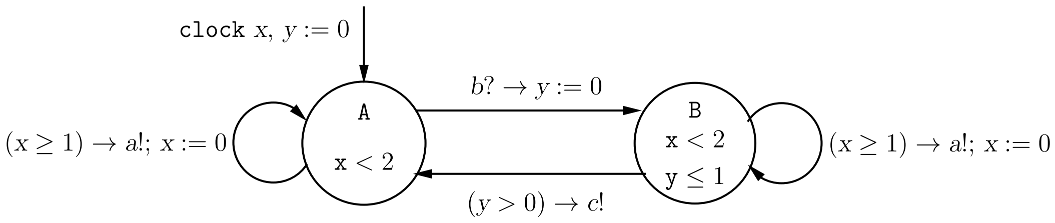


Figure 7.23: Example Timed Automaton for Illustrating Region Equivalence

clock variables, then a clock-valuation is an element of $\texttt{time}^n$, where `time` is the set of non-negative real numbers.

7.3.2 Region Equivalence*

Example Partition

To explain the analysis technique for timed automata, consider the automaton in figure 7.23 with modes `A` and `B`, clocks x and y, input channels a and b of type `event`, and an event output channel c. The constraints involving the clock x imply that the process issues the event a periodically such that the delay between two consecutive such events is greater than or equal to 1 and strictly less than 2. The constraints involving the clock y imply that whenever the process receives an input event b, it issues an output event c with a delay strictly greater than 0 and less than or equal to 1.

Although this timed automaton has only two discrete states, its state-space is infinite since there are infinitely many clock-valuations. The basic idea of the analysis algorithm is to cluster these clock-valuations into finitely many equivalence classes so that equivalent states behave *similarly*. This equivalence, called the *region equivalence*, is depicted in figure 7.24 for our example automaton. In this example, a clock-valuation is a point in the first quadrant of the two-dimensional x/y plane. This space is split into finitely many clusters by drawing vertical, horizontal, and diagonal lines.

Intuitively, whenever we find a line such that the clock-valuations that lie on the line and that lie in the halves on either side of the line lead to different behaviors, we need to split the space by drawing such a line. To limit the number of partitions, we should draw such lines only when needed. The guards and invariants of the automaton compare clock variables with integer constants, and this motivates drawing horizontal and vertical lines. For example, in the mode `A`, if the clock variable x is less than 1, then the event a cannot be issued immediately; if the clock variable x is 1 or more, then the event a can be issued immediately; in the mode `B`, if the clock y is 0, then the event c is not possible without waiting a non-zero amount of time; and if the clock y equals 1, then the event c must be issued immediately. In this example, the largest constant that

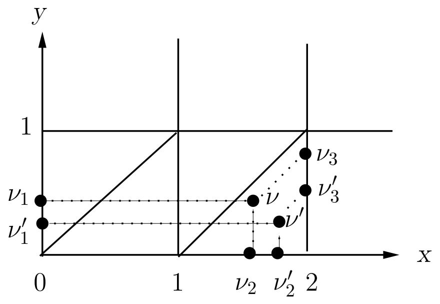


Figure 7.24: Clock Regions

the clock x is ever compared to is 2, and the largest constant that the clock y is ever compared to is 1. This suggests that the actual value of the clock x is not relevant for determining the set of possible future behaviors once it exceeds 2; similarly, once the value of the clock y exceeds 1, it need not be tracked accurately. As a result, to obtain the desired partitioning, we draw the vertical lines $x = 0$, $x = 1$, and $x = 2$ and the horizontal lines $y = 0$ and $y = 1$.

In a given mode, the effect of a timed action of duration δ is adding δ to the values of both clocks. Thus, during a timed action, the clock-valuation evolves along the *diagonal* direction. In our example, in the mode B, consider a clock-valuation where the variable x is between 0 and 1 and the variable y is between 0 and 1. This information is adequate to determine which guards are satisfied and, thus, which mode-switches can be executed immediately. If the constraint $(x > y)$ holds, then as time elapses, the value of the clock x reaches 2 before the clock y reaches 1, at which instance the output event a is enabled. However, if the value of x is less than that of y, then as time elapses, the value of the clock y reaches 1 before the clock x reaches 1, and this implies that the event b is guaranteed to be issued before the event a can be issued. To account for such effects, we split the partition further by drawing the two diagonal lines.

The resulting partition is shown in figure 7.24. Two clock-valuations are considered *region-equivalent* if they belong to the same partition, and each partition consisting of all equivalent clock-valuations is called a *clock-region*. In our example, there are 28 clock-regions: 6 corner points such as $(0, 1)$ and $(2, 0)$; 14 open line segments such as the segment $(0 < y < 1)$ on the line $x = 0$ and the segment $(0 < y < 1)$ on the diagonal line $x = (y + 1)$; and 8 open regions such as the triangular region $(0 < x < y < 1)$ and the unbounded region specified by the constraints $(1 < x < 2)$ and $(y > 1)$.

Two states are region-equivalent if their corresponding discrete states are identical and their clock-valuations are region-equivalent. To understand why region-equivalent states behave similarly, consider two clock-valuations ν and ν' belonging to the open region specified by the constraints $1 < x < 2$ and $0 < y < 1$

and $(x - y) > 1$ shown in figure 7.24. Consider an atomic clock constraint that compares the clock x with 0, 1, or 2 and compares the clock y with 0 or 1. Such a clock constraint is satisfied by the clock-valuation ν if and only if it is satisfied by the clock-valuation ν'. Thus, if a mode-switch is enabled in the state $(\texttt{A}, \nu)$, then it is also enabled in the region-equivalent state $(\texttt{A}, \nu')$, and if a mode-switch is enabled in the state $(\texttt{B}, \nu)$, then it is also enabled in the region-equivalent state $(\texttt{B}, \nu')$. If during such a mode-switch the clock x gets reset, then the resulting clock-valuations after the mode-switch are ν_1 and ν_1' (see figure 7.24), which are again region-equivalent; and if the clock y gets reset, then the resulting clock-valuations after the mode-switch are ν_2 and ν_2', which are again region-equivalent. Starting in the clock-valuation ν, as time elapses, the mode stays the same, and both the clocks increase along the diagonal line. The clock-valuation during such a transition remains region-equivalent to ν until the value of the clock x reaches 2, leading to the clock-valuation ν_3. The effect of a timed action starting from clock-valuation ν' is similar: the clock-valuation evolves along the diagonal line leading to the clock-valuation ν_3', which is region-equivalent to the clock-valuation ν_3. Note that the duration of the timed action leading the clock-valuation ν to ν_3 is different from the duration of the timed action leading the clock-valuation ν' to ν_3', but the order in which different clock-regions are encountered as time elapses is identical for two equivalent clock-valuations.

In summary, whatever action the process can take from a state can be matched by taking a corresponding action starting in a region-equivalent state, resulting in states that are equivalent, from which the same argument can be applied again. Thus, any execution starting in a state can be matched by a corresponding execution starting in a region-equivalent state, such that the two executions have matching sequence of input/output/internal/timed actions, with the only difference being in the exact durations used for timed actions.

As an example, consider the following execution of the timed automaton of figure 7.23:

$$(\texttt{A}, 0, 0) \xrightarrow{0.6} (\texttt{A}, 0.6, 0.6) \xrightarrow{b?} (\texttt{B}, 0.6, 0) \xrightarrow{0.5} (\texttt{B}, 1.1, 0.5) \xrightarrow{c!} (\texttt{A}, 1.1, 0.5)$$
$$\xrightarrow{0.2} (\texttt{A}, 1.3, 0.7) \xrightarrow{a!} (\texttt{A}, 0, 0.7) \xrightarrow{1.25} (\texttt{A}, 1.25, 1.95) \xrightarrow{0.61} (\texttt{A}, 1.86, 2.56).$$

Now suppose at the first step the duration of the timed action is changed to 0.1, resulting in the state $(\texttt{A}, 0.1, 0.1)$ that is region-equivalent to the state $(\texttt{A}, 0.6, 0.6)$. Below is another execution whose first step is the timed action of duration 0.1 such that, at every step of the execution, the state remains region-equivalent to the corresponding state of the execution above:

$$(\texttt{A}, 0, 0) \xrightarrow{0.1} (\texttt{A}, 0.1, 0.1) \xrightarrow{b?} (\texttt{B}, 0.1, 0) \xrightarrow{0.91} (\texttt{B}, 1.01, 0.91) \xrightarrow{c!} (\texttt{A}, 1.01, 0.91)$$
$$\xrightarrow{0.05} (\texttt{A}, 1.06, 0.96) \xrightarrow{a!} (\texttt{A}, 0, 0.96) \xrightarrow{1.25} (\texttt{A}, 1.25, 2.21) \xrightarrow{0.61} (\texttt{A}, 1.86, 2.82).$$

Region Equivalence

Now let us define the notion of region equivalence formally for the general case. Consider a timed automaton *TP*. For two clock-valuations ν and ν' to be considered equivalent, the following conditions must hold. Consider a clock variable x. The clock-valuations ν and ν' must agree on whether the clock x is 0, the clock x is between 0 and 1, the clock x is 1, the clock x is between 1 and 2, and such relationships with respect to the lines parallel to the x-axis. If k_x is the largest constant that the clock x is compared with in the atomic constraints that appear in a guard, update description, or the clock-invariant of *TP*, then once the value of the clock x exceeds k_x, its actual value does not matter. This condition can be summarized by requiring that the clock-valuations ν and ν' agree on all constraints of the form $(x = d)$ and $(x < d)$ for every integer d between 0 and k_x. The second condition accounts for constraints on the differences of clock values. Consider two clocks x and y such that both of them are assigned values that do not exceed their respective thresholds k_x and k_y. Then the values assigned by the clock-valuations ν and ν' must agree on the relationship with respect to the diagonal lines. This can be formalized by requiring that the ordering of the fractional parts of x and y must be identical according to the clock-valuations ν and ν'. For example, in figure 7.24, in each square, for the clock-valuations on the diagonal line, the fractional parts of the clocks x and y are equal; for the clock-valuations in the lower triangle, the fractional part of the clock x exceeds that of the clock y; and for the clock-valuations in the upper triangle, the fractional part of the clock y exceeds that of the clock x.

Two states are region-equivalent if they assign the same values to all the discrete variables and, thus, have identical discrete states, and their clock-valuations are region equivalent. This definition of the region equivalence is captured below.

Region Equivalence

Given a timed automaton *TP*, for each clock variable x, let k_x be the largest integer constant that the variable x is compared with in the atomic constraints appearing in the guards, update descriptions, and the clock-invariant of the automaton *TP*. Two clock-valuations ν and ν' of the timed automaton *TP* are said to be *region-equivalent* if the following conditions hold:

1. for every clock variable x and for every integer $0 \leq d \leq k_x$, $\nu(x) = d$ if and only if $\nu'(x) = d$ and $\nu(x) < d$ if and only if $\nu'(x) < d$; and

2. for every pair of clock variables x and y such that $\nu(x) \leq k_x$ and $\nu(y) \leq k_y$, the fractional part of $\nu(x)$ is less than or equal to the fractional part of $\nu(y)$ if and only if the fractional part of $\nu'(x)$ is less than or equal to the fractional part of $\nu'(y)$.

Two states $s = (t, \nu)$ and $s' = (t', \nu')$ of the timed automaton *TP* are said to be region-equivalent if (1) the discrete states t and t' are the same, and (2) the clock-valuations ν and ν' are region-equivalent.

When a process has three clocks, the desired partitioning is obtained by creating a three-dimensional grid using axis-parallel planes up to a certain relevant constant on each axis. If we consider a cube given by $(0 < \mathrm{x} < 1)$ and $(0 < \mathrm{y} < 1)$ and $(0 < \mathrm{z} < 1)$, then it is further split by diagonal planes into multiple cells. Examples of such clock regions include $(\mathrm{x} < \mathrm{y} < \mathrm{z})$, $(\mathrm{x} = \mathrm{y} < \mathrm{z})$, $(\mathrm{x} = \mathrm{y} = \mathrm{z})$, and $(\mathrm{y} < \mathrm{x} = \mathrm{z})$. Each such clock region can be described by giving the relative ordering of the fractional parts of the three clocks.

As illustrated in figure 7.24, if two states s and t are region-equivalent, then the transition from one state can be matched by a transition from the other such that the resulting states continue to be region-equivalent. This is formalized by the following theorem:

Theorem 7.1 [Region Equivalence] *Consider a timed automaton TP and two states s and t of the automaton TP such that s and t are region-equivalent. Then (1) if $s \xrightarrow{\alpha} s'$ is an input, or output, or internal action of TP, then there exists a state t' such that $t \xrightarrow{\alpha} t'$ holds and states s' and t' are region-equivalent; and (2) for every real-valued time duration $\delta > 0$ such that $s \xrightarrow{\delta} s + \delta$ is a timed action of TP, there exists a duration $\delta' > 0$ such that $s' \xrightarrow{\delta'} s' + \delta'$ is a timed action of TP and the states $s + \delta$ and $s' + \delta'$ are region-equivalent.*

Proof. Let *TP* be a timed automaton. Consider two states s and t that are region-equivalent. We want to prove that every input/output/internal/timed action from the state s can be matched by a corresponding action from the state t such that the target states continue to be region-equivalent.

Observe that if two states are region-equivalent, then every expression that appears in the guard or update code of the automaton has the same value in both states.

Consider an internal action obtained by executing an internal task A with the guard condition *Guard* and update code *Update* from the state s. Since $s(\mathit{Guard}) = t(\mathit{Guard})$, the task A is enabled in the state t also. Now consider the execution of the update code in the two region-equivalent states s and t. At every step of the execution, a conditional expression evaluates to the same value in both states. Executing an assignment of the form $\mathrm{y} := e$, for a discrete variable y, preserves the region-equivalence. If a statement involves a nondeterministic choice, then it can be resolved exactly the same way in both executions. Furthermore, executing an assignment to a clock variable of the form $\mathrm{x} := 0$ preserves the region-equivalence: it is easy to establish that whenever two clock-valuations ν and ν' are region-equivalent, so are the clock-valuations $\nu[\mathrm{x} \mapsto 0]$ and $\nu'[\mathrm{x} \mapsto 0]$. It follows that if the execution of the update code *Update* starting from the states s and t results in the states s' and t', respectively, then it must be the case that states s' and t' are region-equivalent.

The case of input and output actions is similar.

Consider a timed action $s \xrightarrow{\delta} s'$ where $s' = s + \delta$ for $\delta > 0$. Suppose the choice of the duration δ is such that for every $0 \leq \epsilon \leq \delta$, the state $s + \epsilon$ is region-equivalent to either the starting state s or the end-state s' (that is, the time duration is short enough so that multiple regions are not encountered along the way). We want to find a duration δ' so that the states $s + \delta$ and $t + \delta'$ are region-equivalent.

Let us say that a clock x is *integral* in the state s if $s(x)$ is an integer value not greater than the threshold k_x. Similarly, a clock x is *fractional* in the state s if $s(x)$ is not an integer (and thus has a nonzero fractional part) and does not exceed the threshold k_x.

Suppose in the state s there is some clock, say x, that is integral. Then for any $\epsilon > 0$, in the state $s + \epsilon$, the value of the clock x is no longer an integer, and such a state is not region-equivalent to s and, by assumption, must be region-equivalent to s'. Among all the clocks that are fractional in the state s, let y be the clock whose fractional part is the highest in the state s, and let ϵ_s be this fractional value (if there are multiple choices for y with equal fractional values, then we can choose any of them, and if there is no fractional clock in the state s, then this has to be handled as a separate simpler case). Then starting in the state s, if we let $1 - \epsilon_s$ time elapse, then the clock y will have an integer value, and the resulting state will no longer be region-equivalent to s'. Thus it must be the case that $0 < \delta < \epsilon_s$. Since the state t is region-equivalent to the state s, it agrees with the state s in terms of which clocks have integer values, which have fractional values, and the relative ordering of these fractional values. In particular, the clock x has an integer value in the state t also, and for any small $\epsilon > 0$, the state $t + \epsilon$ is no longer region-equivalent to t. Furthermore, among the fractional clocks in the state t, the clock y has the highest fractional value, and let this fraction be ϵ_t. Then verify that for every $0 < \delta' < \epsilon_t$, the state $t + \delta'$ is region-equivalent to the state $s + \delta$, and thus the desired duration can be any value in the interval $(0, \epsilon_t)$.

Now suppose there is no integral clock in the state s, but there is a fractional clock. Then let x be the clock with the highest fractional part, denoted ϵ_s. Then starting in the state s, as time elapses, the state stays region-equivalent to the state s for any duration less than $1 - \epsilon_s$, and at time $1 - \epsilon_s$, the clock x becomes an integer, triggering a change in the region. In this case, the duration δ must be equal to $1 - \epsilon_s$. Now the state t exhibits a similar behavior. If ϵ_t denotes the fractional part of the clock x in the state t, then we can choose $\delta' = 1 - \epsilon_t$ and verify that the state $t + \delta'$ is region-equivalent to the state s'.

When the state s has no integral or fractional clocks, the value of each clock x already exceeds the threshold k_x. In this case, for all values of δ and δ', the states s, $s + \delta$, t, and $t + \delta'$ are all region-equivalent.

Thus, the proof is complete for the case that the delay δ is short enough so that the states s and $s + \delta$ belong to adjacent regions. As exercise 7.13 establishes, in the general case, a timed action of duration δ can be split into a sequence of timed actions such that the duration of each part is short enough: there exist states $s = s_0$, s_1, $\ldots$ $s_n = s'$ and delays $\delta_1, \ldots \delta_n$ with $\delta_1 + \cdots + \delta_n = \delta$ such that for each i, $s_i = s_{i-1} + \delta_i$ and for every $0 \le \epsilon \le \delta_i$, the state $s_{i-1} + \epsilon$ is region-equivalent to either s_{i-1} or s_i. Then starting from the state $t_0 = t$ that is region-equivalent to the state $s_0 = s$, by applying the argument above n times, we can find delays $\delta'_1, \dots \delta'_n$ and states $t_i = t_{i-1} + \delta'_i$ such that each state t_i is region-equivalent to s_i. Thus, the desired duration δ' is the sum $\delta'_1 + \cdots + \delta'_n$, and this ensures that the state $t + \delta'$ is region-equivalent to the state $s' = s_n = s + \delta$. ■

Let us now establish a bound on the number of clock regions. For a clock variable x, a clock region specifies whether the value of the clock x equals an integer d, where the possible choices of d are $0, 1, \dots k_x$, whether the value of the clock x exceeds k_x, or whether the value of the clock x is strictly between the integers $d-1$ and d, where the possible choices of d are $1, 2, \dots k_x$. This gives a total of $2k_x + 2$ choices in terms of the constraint for the clock x. This means that the total number of partitions due to the axis-parallel constraints is given by the product $\Pi_x(2k_x + 2)$. In terms of the ordering of the fractional parts, if the timed automaton has m clock variables, then we get $m!$ many possible orderings. Finally, for every pair of adjacent clock variables x and y in such an ordering, the clock region makes a distinction based on whether the fractional part of the clock x is strictly smaller than that of the clock y or whether the two coincide. This gives additional 2^m choices. Altogether the number of possible clock regions is at most $2^m \cdot m! \cdot \Pi_x(2k_x + 2)$. This bound is not precise. In particular, this calculation considers orderings of fractional parts of all the clocks in every region, but when a clock x equals an integer d, its fractional part is guaranteed to be 0, and thus the actual number of clock regions is smaller than this bound. However, this bound is useful to get an understanding of how the number of clock regions varies: this number grows exponentially with the number of clocks and grows proportional to the product of the constants used in the description.

Search Using Region Equivalence

We have studied how region equivalence can be used to partition the infinite space of clock valuations into finitely many clock regions. We can now adopt the on-the-fly depth-first search enumerative algorithm for reachability of figure 3.16 to timed automata. Instead of enumerating states, the algorithm now enumerates regions, where each region specifies a discrete state and a clock region.

Figure 7.25 shows some of the regions that are found to be reachable for the timed automaton of figure 7.23. The initial region is described by $[\mathtt{A}, x = y = 0]$, which consists of a single state with the mode A and the clock region corresponding to the single clock-valuation $(0, 0)$. In this region, the state can change in two ways: either due to the input event b, leading to the region described by $[\mathtt{B}, x = y = 0]$, or due to a timed action of duration at most 2, leading to one of the regions $[\mathtt{A}, 0 < x = y < 1]$, $[\mathtt{A}, x = y = 1]$, or $[\mathtt{A}, 1 < x < 2, y > 1]$. Edges corresponding to successors due to timed actions are labeled with the symbol

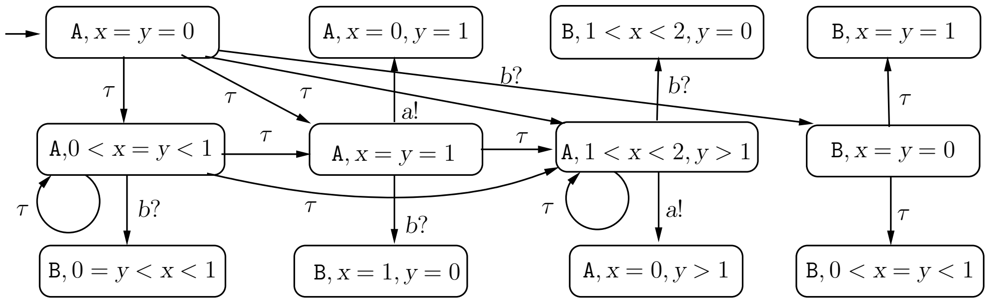


Figure 7.25: Search Using Clock Regions

τ in figure 7.25. This figure shows successors of each of these four regions. For example, consider the region [A, $1 < x < 2, y > 1$]. If the input event b occurs, then the mode changes to B and the clock y gets reset, leading to the region [B, $1 < x < 2, y = 0$]. Since the guard $(x \geq 1)$ is enabled, the output event a can be issued, and this resets the clock x, leading to the region [A, $x = 0, y > 1$]. The third possibility is a timed action. Given a state s that satisfies the constraints $(1 < x < 2)$ and $(y > 1)$, we can find a time duration $\delta > 0$ such that the state $s + \delta$ also satisfies the constraints $(1 < x < 2)$ and $(y > 1)$, and this explains the τ-labeled self-loop on the region [A, $1 < x < 2, y > 1$]. Due to the clock-invariant $(x < 2)$, the region [A, $x = 2, y > 1$] is not reachable using a timed action.

Recall the two illustrative executions for the timed automaton for figure 7.23 such that at every step, the corresponding states were region-equivalent. Both of these executions correspond to the following execution that records regions instead of concrete states:

$$
\begin{aligned}
&[\mathtt{A}, x = y = 0] \xrightarrow{\tau} [\mathtt{A}, 0 < x = y < 1] \xrightarrow{b?} [\mathtt{B}, 0 < x < 1, y = 0]\\
&\xrightarrow{\tau} [\mathtt{B}, 0 < y < 1 < x < y + 1] \xrightarrow{c!} [\mathtt{A}, 0 < y < 1 < x < y + 1]\\
&\xrightarrow{\tau} [\mathtt{A}, 0 < y < 1 < x < y + 1] \xrightarrow{a!} [\mathtt{A}, x = 0 < y < 1]\\
&\xrightarrow{\tau} [\mathtt{A}, 1 < x < 2 < y] \xrightarrow{\tau} [\mathtt{A}, 1 < x < 2 < y].
\end{aligned}
$$

In this example, there are two discrete states (the mode can be either A or B), and there are at most 28 clock regions as shown in figure 7.24. This implies that the depth-first search algorithm can explore only 56 possible regions. In general, if the types of variables that are not clocks are finite, then there are only finitely many discrete states, and hence only finitely many regions, and the depth-first search algorithm exploring regions is guaranteed to terminate.

Given a property φ of a timed automaton *TP*, which in general is a Boolean expression over its state variables, when can we use the search over regions to determine whether the property φ is an invariant of the automaton *TP*? If the property φ refers only to discrete variables, then since the search over regions keeps track of the discrete state, it is adequate to check if the property φ holds in every reachable region. If the property refers to the clock variables also, then the region-based search is adequate as long as these references are consistent with the partitioning, that is, every atomic constraint involving clock variables in the property φ is of the form $x \leq d$ or $x \geq d$ for some integer constant $d \leq k_x$. In other words, let φ be a property such that whenever two states s and t are region-equivalent, either both states s and t satisfy φ or both do not satisfy φ. Hence, such a property φ is called a *region-invariant* property. For such a region-invariant property φ, to check whether all reachable states of the timed automaton satisfy φ, it suffices to check whether all reachable regions satisfy φ using a search over regions.

The analysis for timed automata can be easily adopted to handle rational constants. In the audio control protocol of section 7.2.2, the model has constraints of the form $(5 - 5\epsilon \leq y \leq 5 + 5\epsilon)$, where ϵ is a rational-valued constant such as $1/15$. To handle such models, we can simply multiply all constants by a factor of 15 to make them integers without changing the executions that are possible in the model.

We conclude this section by noting that defining region-equivalence, and using the resulting equivalence classes for analysis (more specifically, for depth-first search), is an instance of the general concept of *abstraction*. Concrete states such as $\langle \mathtt{A}, 0.2, 0.3\rangle$ are replaced by regions such as $[\mathtt{A}, 0 < x < y < 1]$. Such a mapping removes some details and, hence, is called an abstraction, and the regions are called *abstract states*. Since many concrete states get mapped to the same abstract state, searching through abstract states is more efficient. Theorem 7.1 says that, in the case of timed automata, this specific abstraction using regions retains enough information so that a search over the abstract states accurately captures which sequences of input-output events are feasible and whether a region-invariant property is an invariant of the system.

Exercise 7.11: Consider the timed automaton of figure 7.23. List all possible regions that are successors of the region $[\mathtt{A}, x = 0, y > 1]$ and all possible regions that are successors of the region $[\mathtt{B}, 0 < x = y < 1]$. ■

Exercise 7.12: Consider a timed automaton with 3 clocks x, y, and z, with $k_x = k_y = k_z = 1$. List all possible clock regions for this automaton. ■

Exercise 7.13*: Prove that every timed action in a timed automaton can be split into a sequence of timed actions such that during every subaction, the state is region-equivalent to either the start or the end state of this subaction. Formally, consider a state s of a timed automaton and a duration δ. Prove that there exist states $s = s_0, s_1, \ldots s_n = s + \delta$ and delays $\delta_1, \ldots \delta_n$ with $\delta_1 + \cdots + \delta_n = \delta$ such that for each i, $s_i = s_{i-1} + \delta_i$, and for every $0 \leq \epsilon \leq \delta_i$, the state $s_{i-1} + \epsilon$ is region-equivalent to either s_{i-1} or s_i. Find a bound b as a function of the number m of the clock variables of the automaton so that every timed action can be split into at most b subactions of this desired form? ■

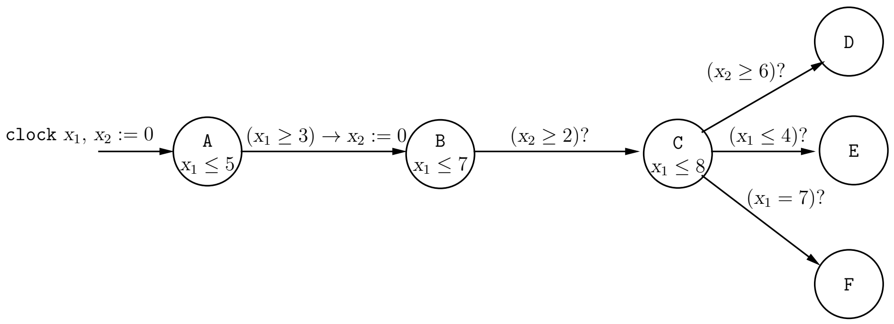


Figure 7.26: Example for Analyzing Timing Feasibility of Executions

Exercise 7.14*: Suppose we modify the definition of a timed automaton so that clock variables can be reset to constant values (that is, for a clock variable x, the allowed assignments are of the form $x := d$, where d is a non-negative integer constant), and tests can also compare differences of clock variables with constants (that is, each atomic expression involving clock variables is of the form $x \leq k$, $x \geq k$, or $x - y \leq k$, where k is an integer constant). How would you modify the definition of the region-equivalence over clock valuations so that there are only finitely many clock regions, and the analog of theorem 7.1 continues to hold? ■

7.3.3 Matrix-Based Representation for Symbolic Analysis

Region equivalence allows partitioning of the infinite space of clock-valuations into finitely many regions, and we have seen how it can be used for invariant verification of timed automata using a search algorithm that enumerates all reachable regions. While theorem 7.1 implies that it is sufficient to keep track of clock regions to verify region-invariant properties, such a fine partitioning may not be necessary to solve a particular invariant verification problem. For example, consider the search for reachable regions from the initial region $[\mathtt{A}, x = y = 0]$ shown in figure 7.25. Instead of enumerating the three regions $[\mathtt{A}, 0 < x = y < 1]$, $[\mathtt{A}, x = y = 1]$, and $[\mathtt{A}, 1 < x < 2, y > 1]$, which are successors of the initial region corresponding to timed actions, we can represent the result of a timed action from the initial region by the single set $[\mathtt{A}, 0 < x = y < 2]$. In this section, we consider a symbolic representation called *clock zones* that allows analyzing clock regions in clusters instead of enumerating them in a manner that allows an efficient representation and manipulation.

Example of Timing Analysis Using Clock Zones

To explain the symbolic analysis technique, let us consider the timed automaton shown in figure 7.26. Examination of timing constraints reveals that the mode D is not reachable, that is, the path A, B, C, D cannot be traversed. Similarly,

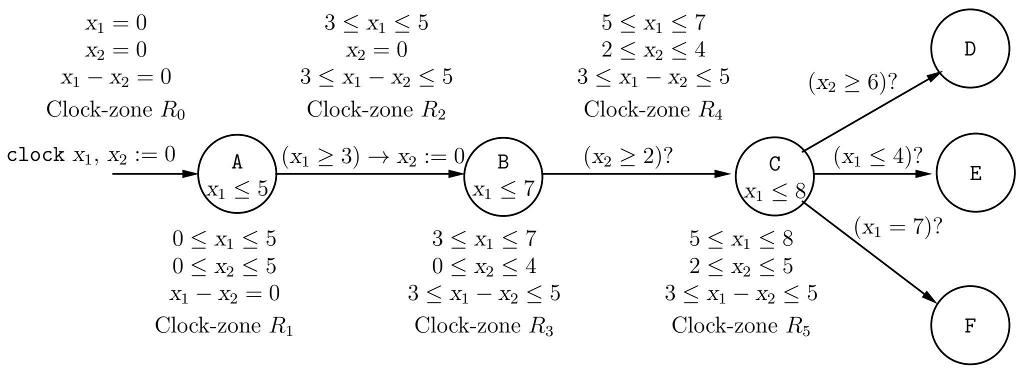


Figure 7.27: Inferring and Propagating Clock Constraints

the mode `E` cannot be reached, but it is possible to reach the mode `F`. This is not evident by a local examination of the constraints on the clock values in the clock-invariant of the mode `C` and the guards associated with the switches out of the mode `C` but is based on the implied constraints on the values of the clocks x_1 and x_2 when an execution reaches the mode `C`. This is illustrative of the nature of analysis that is needed to check whether the timing-based mutual exclusion protocol satisfies the mutual exclusion requirement: in the parallel composition of multiple instances of the timed process of figure 7.9, is it possible to reach the state with two processes in the mode `Crit` while satisfying the timing constraints imposed by the guards and the clock-invariants at every step?

A state in this example consists of the mode that takes values from the enumerated set $\{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}\}$ and the values for the clock variables x_1 and x_2, each of which can be a non-negative real number. We can use finite-state analysis by partitioning the space of clock-valuations into clock regions, but for the automaton of figure 7.26, $k_1 = 7$ and $k_2 = 6$, and as a result, there are many clock regions (140, to be precise), and we wish to avoid considering all such clock regions individually. For this purpose, we generalize the notion of a clock region to a *clock zone*, which is a set of clock-valuations that is represented using constraints of a particular form over the variables x_1 and x_2, namely, bounds on the values of individual clock variables and bounds on the differences between the values of clock variables.

Initially, both clocks are 0, and this leads to the constraint

$$(x_1 = 0) \;\wedge\; (x_2 = 0) \;\wedge\; (x_1 - x_2 = 0).$$

This is shown as the clock zone R_0 in figure 7.27. Given that the set of clock-valuations upon entry to the mode `A` is described by the constraints R_0, we can calculate the set of clock-valuations that can be reached using timed actions as the process waits in the mode `A`. The value of the clock x_1 increases but cannot

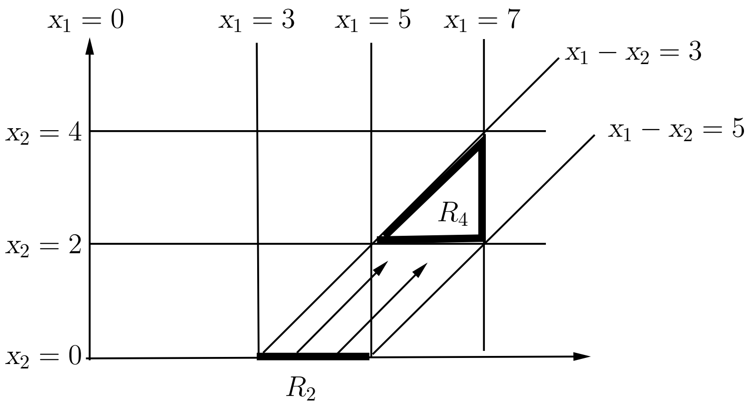


Figure 7.28: Illustrating Clock Zone Manipulations

exceed 5 due to the clock-invariant associated with the mode, and this gives the constraint $0 \leq x_1 \leq 5$. Observe that during timed actions, the difference in the clock values stays unchanged, so the constraint $(x_1 - x_2 = 0)$ from R_0 stays unchanged. These two constraints imply bounds on the value of the clock x_2, and this gives the description of the clock zone R_1:

$$(0 \leq x_1 \leq 5) \;\wedge\; (0 \leq x_2 \leq 5) \;\wedge\; (x_1 - x_2 = 0).$$

Note that the clock zone R_1 consists of multiple clock regions such as the corner point $x_1 = x_2 = 3$ and the line segment $3 < x_1 = x_2 < 4$. Also observe that the number of clock regions that can be reached from the clock zone R_0 due to a timed action is proportional to the constant 5 appearing in the clock-invariant of the mode `A`, while there is always a single clock zone that captures all the clock-valuations that are reachable using a timed action starting in the clock zone R_0.

The set R_2 describing the set of clock-valuations upon entry to the mode `B` is calculated from the clock zone R_1 by intersecting it with the guard condition $(x_1 \geq 3)$ and setting the clock x_2 to 0 to capture the effect of the assignment. The desired clock zone R_2 is described by the constraints:

$$(3 \leq x_1 \leq 5) \;\wedge\; (x_2 = 0) \;\wedge\; (3 \leq x_1 - x_2 \leq 5).$$

Again notice the implied constraint $(3 \leq x_1 - x_2 \leq 5)$.

This process can be repeatedly applied. The clock zone R_3 describes the set of clock-valuations that are reachable as time elapses in the mode `B`, and the clock zone R_4 describes the set of clock-valuations upon entry to the mode `C`. To get some intuition about this calculation, see figure 7.28. The clock zone R_2 is the segment of the line $x_2 = 0$ between $x_1 = 3$ and $x_1 = 5$. As time evolves, this line segment moves diagonally between the lines $(x_1 - x_2 = 3)$ and $(x_1 - x_2 = 5)$. The vertical line $x_1 = 7$ captures the clock-invariant and restricts the reachable clock-valuations in the mode B. The clock zone R_3 is thus the trapezoid between the lines $(x_2 = 0)$, $(x_1 = 7)$, $(x_1 - x_2 = 3)$, and $(x_1 - x_2 = 5)$. The guard condition of the mode-switch from the mode B to the mode C means that this trapezoid should be intersected with the constraint $(x_2 \geq 2)$, leading to the triangular clock zone R_4.

The clock zone R_5 describes the set of clock-valuations that are reachable as time elapses in the mode C (see figure 7.27) and is described by the constraints:

$$(5 \leq x_1 \leq 8) \ \wedge \ (2 \leq x_2 \leq 5) \ \wedge \ (3 \leq x_1 - x_2 \leq 5).$$

This accurately captures the cumulative effect of timing constraints along the path leading to the mode C. Intersection of this clock zone with the guard condition $(x_2 \geq 6)$ on the switch to the mode D is the empty set, and this establishes that the mode D is unreachable. Similarly, the intersection of the clock zone R_5 with the guard condition $(x_1 \leq 4)$ is also the empty set, and so the mode E cannot be reached. Intersecting the set R_5 with the guard condition $(x_1 = 7)$ gives a non-empty set, namely, $(x_1 = 7) \wedge (2 \leq x_2 \leq 4)$, that describes the set of clock-valuations upon entry to the mode F.

Difference Bounds Matrices

The most natural way of representing the constraints that arise during timing analysis is using a matrix-based representation. Let us assume that the timed automaton has m clock variables: $x_1, x_2, \ldots x_m$. We use a dummy clock x_0 that is assumed to represent the constant 0. Then a clock zone is represented by a square matrix R of dimension $m + 1$: the (i, j)th entry of the matrix gives the upper bound on the difference $(x_i - x_j)$.

We use the symbolic constant ∞ to denote a large value, and this is used to represent absence of a bound. More specifically, let `Bounds` be the set `int` of integers together with the symbolic constant ∞. The usual operations of comparison, minimum, and addition over integers are extended to the set `Bounds` in the following way: for every integer n, $n \leq \infty$ holds and $\min(n, \infty) = n$ and $n + \infty = \infty$.

A clock zone is represented by a square matrix R of dimension $(m + 1)$, with entries in `Bounds`, which represents the conjunction of constraints

$$\bigwedge_{0 \leq i \leq m, 0 \leq j \leq m} (x_i - x_j) \ \leq \ R_{ij}.$$

The column 0 (that is, the entries R_{i0}) gives the upper bounds on the clocks x_i, and the row 0 (that is, the entries R_{0i}) gives the upper bounds on the values of $-x_i$ (and thus, the negations of these entries capture lower bounds on the clocks x_i). Such a matrix representing bounds on the differences of clock values is called a *difference bounds matrix* (DBM).

Going back to our example from figure 7.27, the clock zone R_1 is represented by the following DBM:

$$\begin{bmatrix} 0 & 0 & 0 \\ 5 & 0 & 0 \\ 5 & 0 & 0 \end{bmatrix}$$

and the clock zone R_5 is represented by the following DBM:

$$\begin{bmatrix} 0 & -5 & -2 \\ 8 & 0 & 5 \\ 5 & -3 & 0 \end{bmatrix}.$$

Note that a lower bound of 5 on x_1 shows up as the upper bound -5 on $-x_1$ in the first row, and a lower bound of 3 on $(x_1 - x_2)$ shows up as the upper bound of -3 on the difference $(x_2 - x_1)$.

The representation (and the zone-based analysis of the example) discussed so far assumes that constraints on the clock values do not occur inside negation. A negated constraint such as $\neg(x_1 \geq 2)$ is equivalent to the strict inequality $(x_1 < 2)$. In the presence of such constraints, we need to distinguish between a non-strict upper bound of 2 (generated by the constraint $x_1 \leq 2$) and a strict upper bound of 2 (generated by the constraint $x_1 < 2$). This requires tagging each integral bound with a Boolean flag that indicates whether the associated constraint is strict or non-strict. The representation using DBMs and the techniques for manipulating DBMs can be adopted to handle this distinction (see exercise 7.19).

DBM Manipulation

The key insight regarding algorithmic and efficient inference of constraints (which is necessary to derive the constraint $(x_2 \leq 5)$ from the constraints $(x_1 \leq 5)$ and $(x_1 - x_2 = 0)$ in the description of the clock zone R_1 in our example in figure 7.27) is the following. Since the entry R_{il} is an upper bound on the difference $(x_i - x_l)$ and the entry R_{lj} is an upper bound on the difference $(x_l - x_j)$, the sum $(R_{il} + R_{lj})$ is an *inferred* upper bound on the difference $(x_i - x_j)$. If the entry R_{ij} is larger than the sum $(R_{il} + R_{lj})$, then we can *tighten* the upper bound R_{ij} by replacing it with the inferred bound $(R_{il} + R_{lj})$.

The DBM R is said to be *canonical* if and only if

$$\text{for all } 0 \leq i, j, l \leq m, \quad R_{ij} \;\leq\; (R_{il} + R_{lj}).$$

That is, in a canonical matrix, every entry R_{ij} represents the tightest bound that can be inferred on the difference $(x_i - x_j)$. Figure 7.29 shows an algorithm that computes the canonical version of an input DBM.

The tightening of upper bounds and the computation of the algorithm can be readily understood by an alternative view of the DBM as a weighted directed graph. Consider the graph with $m + 1$ vertices $x_0, x_1, \ldots x_m$. For every pair of

```
Input: $(m+1) \times (m+1)$ DBM $R$ with entries in Bounds.
Output: Empty if $R$ is empty, else canonical version of $R$.

for $l = 0$ to $m$ {
  for $i = 0$ to $m$ {
    for $j = 0$ to $m$ {
      $R[i,j] := \mathtt{min}\,(R[i,j], R[i,l] + R[l,j])$
    };
    if $R[i,i] < 0$ then return Empty
  }
}
return $R$.
```

Figure 7.29: Algorithm for Canonicalization of DBMs

vertices x_i and x_j, there is an edge from the vertex x_i to the vertex x_j whose cost is equal to the entry R_{ij}. Adding the entries R_{il} and R_{lj} gives the cost of a path from the vertex x_i to the vertex x_j consisting of two edges. If this cost is smaller than the cost of the direct edge from x_i to x_j, then we can replace the cost of this edge by the smaller value. In general, the path with the smallest cost between two vertices gives the tightest upper bound on the difference between the corresponding clocks. Then a DBM can be converted into a canonical one by executing the shortest path algorithm (or, equivalently, the transitive closure construction) on the matrix. The algorithm of figure 7.29 is indeed the classical Floyd-Warshall shortest-path algorithm. In the outer-most loop, the value of the variable l changes from 0 to m, and at every iteration, for all pairs (i,j), the entry $R[i,j]$ captures the shortest path from the vertex x_i to the vertex x_j using only vertices indexed $\leq l$, that is, the tightest upper bound on $(x_i - x_j)$ using constraints that involve variables with indices $\leq l$.

As an example, suppose $m = 3$ and consider the clock zone given by the constraints

$$(1 \leq x_1 \leq 3) \ \wedge \ (x_2 \geq 0) \ \wedge \ (0 \leq x_3 \leq 3) \ \wedge \ (x_2 - x_3 = 1) \ \wedge \ (x_2 - x_1 \geq 2).$$

Translating these constraints to the DBM representation gives the matrix R:

$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ 3 & 0 & -2 & \infty \\ \infty & \infty & 0 & 1 \\ 3 & \infty & -1 & 0 \end{bmatrix}.$$

The corresponding graph representation is shown in figure 7.30 on the left. Note that whenever a matrix entry is ∞, the corresponding edge is not shown as this indicates absence of a constraint. Also, self-loops with cost 0 are not shown.

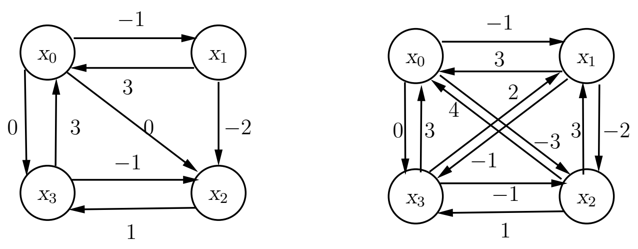


Figure 7.30: Illustrating Canonicalization

The result of canonicalization leads to the matrix

$$\begin{bmatrix} 0 & -1 & -3 & -2 \\ 2 & 0 & -2 & -1 \\ 4 & 3 & 0 & 1 \\ 3 & 2 & -1 & 0 \end{bmatrix}.$$

This matrix corresponds to the graph on the right in figure 7.30. Verify that for the graph on the right, the cost of an edge between a pair of vertices corresponds to the shortest (in terms of total cost) path between those two vertices in the left graph (for example, the shortest path from the vertex x_1 to the vertex x_0 is the path x_1, x_2, x_3, x_0 and has cost $-2 + 1 + 3 = 2$).

The algorithm for canonicalization also needs to address the following question: given a DBM R, is the conjunction of all the constraints represented by R satisfiable? It turns out that the matrix R represents an unsatisfiable set of constraints, and thus the empty set of clock valuations precisely when the corresponding graph has a cycle with a negative cost. For example, consider the constraints $(x_1 \geq 1)$ and $(x_2 \leq 2)$ and $2 \leq (x_1 - x_2) \leq 3$. These constraints are unsatisfiable, corresponding to the empty clock zone. In the DBM representation, due to the first constraint, we set R_{01} to -1; due to the second constraint, we set R_{20} to 2; and the third constraint gives $R_{12} = -2$ and $R_{21} = 3$. Adding up R_{01} and R_{12} implies that R_{02} must be tightened to -3, and adding up R_{02} and R_{20} then implies that R_{00} must be tightened to -1. In the graph view, the edge with cost -1 from x_0 to x_1, the edge with cost -2 from x_1 to x_2, and the edge with cost 2 from x_2 to x_0 form a cycle with a total cost that is negative. In such a case, repeating this cycle lowers the cost further and further, and the DBM cannot be made canonical. In the algorithm of figure 7.29, if some entry R_{ii} is lowered from 0 to some negative value, the algorithm has detected a cycle with a negative cost and returns with the answer that the input DBM corresponds to the empty clock zone. If the input DBM represents a non-empty clock zone, then the algorithm tightens all the entries as much as possible, and the output DBM is the canonical version of the input DBM.

Let us consider some operations that are useful on DBMs.

- **Atomic constraints:** Consider an atomic constraint $(x_i \leq k)$ for a constant k. To obtain the DBM R representing this constraint, we first set all the diagonal entries R_{jj} to 0, for $0 \leq j \leq m$; set the entry R_{i0} to k to reflect the upper bound on the difference $(x_i - x_0)$; set the entry R_{0j} to 0, for $0 \leq j \leq m$, to reflect the implicit assumption that $(x_j \geq 0)$ holds for every clock variable; and set all the remaining entries R_{jl} to ∞ to indicate absence of explicitly stated bounds. We then use the algorithm of figure 7.29 to convert this DBM into a canonical one.

- **Intersection:** Consider two DBMs R and R' both in canonical forms. To compute the intersection of the clock zones represented by these matrices, we simply set the (i, j)th entry of the result to be the minimum of R_{ij} and R'_{ij}. Then, we can test if the resulting matrix is empty, and if not, make it canonical using the algorithm of figure 7.29. The intersection operation is useful for capturing the effect of clock-invariants and of tests in guards appearing on mode-switches.

- **Time elapse:** Given a canonical DBM R representing a clock zone, to compute the set of clock-valuations that can be reached starting in the set R using timed actions (without accounting for the upper bounds imposed by the clock-invariants), we simply set the entry R_{i0}, for $1 \leq i \leq m$, to ∞. As time elapses, clock values increase, so the upper bounds on individual clock values are changed to ∞. Lower bounds on clock values and bounds on differences on clocks do not change because of timed actions.

- **Clock reset:** Given a canonical DBM R and a clock x_i, for $1 \leq i \leq m$, we can define an operation on the DBM so that the result captures the set of clock-valuations that can be obtained by assigning the clock x_i to 0 starting from a clock-valuation in R (see exercise 7.17 to develop details of this operation).

- **Subset test:** If R and R' are two canonical (non-empty) DBMs, then the clock zone represented by the DBM R is a subset of the clock zone represented by the DBM R' precisely when for every $0 \leq i, j \leq m$, $R_{ij} \leq R'_{ij}$. In particular, two canonical (non-empty) DBMs represent the same clock zone precisely when all of their respective entries match.

Reachability Analysis

To verify safety requirements of timed systems, we can now adopt the on-the-fly depth-first search algorithm of section 3.3 using clock zones. A *zone* is now represented as a pair (s, R), where the discrete state s records the values of all the discrete variables and R is a non-empty canonical DBM that captures a set of clock-valuations. The basic search mechanism stays the same. In particular, zones are explored and examined on demand, and the algorithm terminates as soon as a violation of the safety property is encountered. As in the case of the search using clock regions, the clustering of clock-valuations using clock zones is adequate as long as the property being checked is region-invariant.

For a zone (s, R), one possible successor zone is obtained by considering the effect of letting time elapse using a timed action. For this purpose, the algorithm first intersects the DBM R with the clock-invariant corresponding to the discrete state s, updates the matrix R to reflect elapse of time (by setting the entries in the 0-th column to ∞), and then again intersects it with the clock-invariant corresponding to the discrete state s. Note that the clock-invariant corresponding to each discrete state s also needs to be represented as a DBM, and such a DBM can be obtained using the constructions corresponding to atomic constraints and intersection. At every step, the resulting DBM is tested for emptiness and, if non-empty, is made canonical.

For a zone (s, R), the successor corresponding to a discrete transition, that is, execution of either an input, an output, or an internal task, is computed using the following steps. First, we compute the intersection of the DBM R and the DBM that captures the constraints on clock values for the guard condition of the corresponding task. If this intersection is an empty set, then this task is not enabled; otherwise the resulting DBM is made canonical. Then the discrete state s for the discrete variables is updated according to the update description of the task. If the update involves setting a clock variable to 0, then the clock reset operation is applied to the DBM part.

Zones of the form (s, R) are stored in the hash-table *Reach* that contains the zones visited so far. While examining a zone (s, R), the algorithm considers it as visited if a zone of the form (s, R'), where the DBM R is a subset of the DBM R', has been visited before. To implement this check, given a discrete state s, there needs to be an efficient way to access the set of all DBMs R such that the zone (s, R) has been encountered before.

Observe that the search algorithm has a mix of enumerative and symbolic flavors: discrete variables are processed by explicitly enumerating their values and clock variables are manipulated using constraints represented as DBMs. For finite-state timed automata, the number of choices for the discrete states s is bounded a priori, and the zone-based search is guaranteed to terminate.

The search algorithm using clock zones is implemented in tools such as the model checker UPPAAL (see `www.uppaal.com`) with many optimizations. The same ideas can also be applied to modify the nested depth-first search algorithm of chapter 5 for checking liveness properties of timed systems.

Exercise 7.15: Suppose a timed automaton has two clocks x_1 and x_2. Before entering a mode `A`, suppose we know that $(3 \leq x_1 \leq 4)$ and $(1 \leq x_1 - x_2 \leq 6)$ and $(x_2 \geq 0)$:

1. Show the DBM corresponding to the given constraints.

2. Is the DBM in part (1) canonical? If not, obtain an equivalent canonical form.

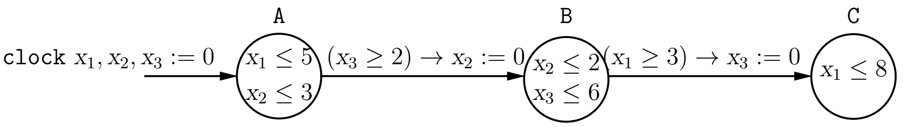

A
B
C
clock $x_1, x_2, x_3 := 0$
$x_1 \leq 5$
$x_2 \leq 3$
$(x_3 \geq 2) \to x_2 := 0$
$x_2 \leq 2$
$x_3 \leq 6$
$(x_1 \geq 3) \to x_3 := 0$
$x_1 \leq 8$

Figure 7.31: DBM Exercise

3. Suppose the clock-invariant of mode `A` is $(x_2 \leq 5)$. Compute the canonical DBM that captures the set of clock values that can be reached as the process waits in mode `A`.

4. Consider a mode-switch out of mode `A` with guard $(x_1 \geq 7)$ and update $x_1 := 0$. Compute the canonical DBM that captures the set of clock values that are possible after taking this transition.

■

Exercise 7.16: Consider the timed process shown in figure 7.31 with three clocks. Compute the sets R_A, R'_A, R_B, R'_B, R_C, and R'_C of clock values represented as canonical DBMs such that each of the DBMs R_A, R_B, R_C captures the possible clock values when the corresponding mode is entered, and each of the DBMs R'_A, R'_B, R'_C captures the possible clock values as the process waits in the corresponding mode. ■

Exercise 7.17: Consider a non-empty canonical DBM R and an index $1 \leq i \leq m$. Describe clearly how to compute the DBM R' that captures the effect of setting the clock x_i to 0. That is, R' should represent the set of all clock-valuations v such that $v = u[x_i \mapsto 0]$ for some $u \in R$. ■

Exercise 7.18: For $m = 3$, consider the constraints given by

$$x_1 \leq 3 \ \wedge \ x_3 \geq 1 \ \wedge \ 4 \leq x_1 - x_2 \leq 10 \ \wedge \ x_1 - x_3 \leq 2 \ \wedge \ x_3 - x_2 \leq 2.$$

Draw the weighted directed graph with four vertices that captures these constraints. Then draw the graph corresponding to the canonical DBM in which the weights reflect the shortest paths in the original graph. ■

Exercise 7.19*: The DBMs we have discussed cannot capture *strict* inequalities such as $(x_1 < 2)$. For this purpose, we can change the set `Bounds` to contain, in addition to the symbolic constant ∞, pairs of the form (k, b), where k is an integer and b is a Boolean value. The entries of the DBM now range over this new type. Then to capture the constraint $(x_1 < 2)$, we set R_{10} to the value $(2, 0)$, and to capture the constraint $(x_1 \leq 2)$, we set R_{10} to the value $(2, 1)$. The concepts such as tightening of bounds and canonicalization continue to work provided we extend the operations of comparison, minimum, and addition over this new set of bounds. Define these operations precisely. Over two clocks, consider the constraints $(3 < x_1 \leq 6)$ and $(1 \leq x_1 - x_2 < 4)$ and $(x_2 \geq 0)$. Show the DBM corresponding to these constraints. Is this DBM canonical? If not, obtain an equivalent canonical form. ■

Bibliographic Notes

Since the 1980s, there have been many proposals for incorporating timing constraints in formal models of reactive computation (see, for instance, timed I/O automata as an example of a well-developed model [KLSV10]). The model presented in this textbook is based on timed automata [AD94], which has been studied extensively, resulting in a wealth of theoretical results and practical applications.

The data structure of difference-bounds matrices for analysis of timing constraints was introduced in [Dil89], and the concept of regions for finite partitioning of the state-space of timed models was introduced in [AD94]. Model checkers that implement these analysis techniques include KRONOS [HNSY94], RED [Wan04], and UPPAAL [LPY97], which now supports different forms of efficient analysis tools for real-time systems and has been used in industrial case studies (see `www.uppaal.org` and [BDL$^+$11]).

The mutual exclusion algorithm of figure 7.9 is due to Fischer (see [Lam87] and [Lyn96] for solving distributed coordination problems relying on timing delays). The formal modeling and analysis of the audio control protocol in section 7.2.2 is based on [HW95] (see [BGK$^+$96] for an automated analysis of the protocol using UPPAAL). Applying formal methods to the design and verification of a pacemaker is described as a challenge at `sqrl.mcmaster.ca/pacemaker.html` (see also [LSC$^+$12] for a survey of formal modeling and analysis of medical cyber-physical systems). The pacemaker design in section 7.2.3 is based on [JPAM14].