

# Chapter 6

## Timed Models

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 uncontrolled 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 synchronized, 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 action corresponding to sensing every 5ms,” “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.”

### 6.1 Timed Processes

The formal model of computation for timed processes is similar to the model of *asynchronous processes* from Chapter 3. We will first illustrate the model with examples.

#### Timing-based light switch

Consider a light switch that uses a single switch, along with timing assumptions, 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 low intensity, and if it is pressed twice in rapid succession, the light is turned on at a bright intensity. Here, “rapid” means that the duration between the successive press events is less than 10ms. If the delay between the successive press events is more than 10ms, the second press event is interpreted as a command to switch the light off.

The system is modeled by the timed process `LightSwitch` of Figure 6.1. It has an input channel `press` on which it receives events corresponding to pressing of the switch. The dynamics is illustrated using the state-machine notation.

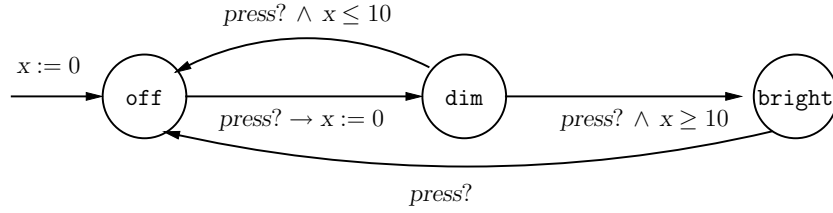


Figure 6.1: A timed model of a light switch

Such state machines have a special state variable  $s$ , called the *mode*. 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 transitions just like an asynchronous process, and during such transitions, clock variables are tested and updated in the same way as other state variables. In addition, a timed process also has *timed transitions* that capture elapse of time. During a timed transition of duration  $\delta$ , which may be any positive real number, each clock variable is incremented by  $\delta$ , and other variables stay unchanged.

For the process **LightSwitch**, initially the mode is **off**,  $x$  is 0, and the process is waiting for the input event *press*. Waiting for a time period  $\delta_1$  is modeled by a time transition of duration  $\delta_1$ . After such a transition, the mode is still **off** and the value of  $x$  is  $\delta_1$ . When the input event *press* is received, the process updates the mode to **dim** and resets the clock variable to 0. As the process waits in the mode **dim**, the value of  $x$  captures the time elapsed with respect to the time instance when the transition from **off** to **dim** occurred. If the process waits in the mode **dim** for a total of  $\delta_2$  time using a timed transition of duration  $\delta_2$ , then the value of  $x$  will be  $\delta_2$ . When the input event *press* happens, the value of  $x$  is used to decide if the process will update the mode to **bright** or to **off**, and this is captured by the conditions  $x \leq 10$  and  $x \geq 10$  on the two transitions. Note that if the value of  $x$  is exactly 10 (that is, the duration between the two successive *press* events is 10ms), both transitions are possible, and thus, the model behaves nondeterministically. When the process is in the mode **bright**, it switches back to **off** whenever it receives the next input event.

One possible execution of the process is shown below:

$$\begin{aligned}
 & (\text{off}, 0) \xrightarrow{2.3} (\text{off}, 2.3) \xrightarrow{\text{press?}} (\text{dim}, 0) \xrightarrow{0.2} (\text{dim}, 0.2) \xrightarrow{0.5} \\
 & (\text{dim}, 0.7) \xrightarrow{\text{press?}} (\text{bright}, 0.7) \xrightarrow{3.0} (\text{bright}, 3.7) \xrightarrow{\text{press?}} (\text{off}, 3.7)
 \end{aligned}$$

Note that during an execution two timed transitions may follow one another: the effect a timed transition of duration  $\delta_1$  immediately followed by a timed transition of duration  $\delta_2$  is the same as a single timed transition of duration  $\delta_1 + \delta_2$ .

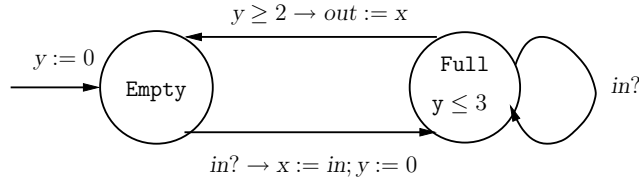


Figure 6.2: A timed buffer with a bounded delay

### Buffer with a bounded delay

As a second example, let us consider a timed buffer of capacity 1 with input channel  $in$  and output channel  $out$ . Whenever an input value  $v$  is supplied to the buffer, it is stored in the internal state  $x$ . Now the buffer becomes full, and it simply ignores (or loses) further inputs until it gets a chance to output the value on the output channel. The timing assumption is that the delay between receiving the input and the sending of the output is at least 2 and at most 3 time units. This form of lower and upper bounds on delays is a typical pattern. The timed process `TimedBuf` shown in Figure 6.2 captures the desired timing assumptions using one clock variable  $y$ . The mode 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 variable  $x$ , the mode is updated to `Full`, and the clock  $y$  is set to 0. As time elapses while the mode is `Full`, the value of the clock  $y$  captures the total time the process has been waiting in this mode. Input events received in the mode `Full` do not change the buffer state. The output transition is guarded with the condition  $y \geq 2$ , and thus captures the assumption that the buffer can issue the message on its output channel only after the lower bound 2 on the delay. The assumption concerning the upper bound, namely, the process is guaranteed to issue the output within 3 time units of receiving the input, is captured by the annotation  $y \leq 3$  on the mode `Full`. If the mode is `Full` and the value of the clock  $y$  is  $\delta$ , then a timed transition of duration  $\delta'$  is allowed only if the constraint  $y \leq 3$  holds throughout the transition as the value of  $y$  keeps increasing with time, that is, only if  $\delta + \delta' \leq 3$ . The process and its environment are synchronizing on passage of time during a timed transition. The process wants to issue the output before the clock  $y$  reaches 3, and thus, is willing to let time elapse only upto a certain limit.

### Timed process with multiple clocks

As an example of a timed process using two clocks, consider the process in Figure 6.3 with input channel  $in$  and output channels  $out_1$  and  $out_2$ . If an input event on channel  $in$  happens at time  $t$ , the process responds by producing an output event on  $out_1$  at time  $t_1$  followed by an output event on  $out_2$  at time  $t_2$  such that the delay  $t_1 - t$  is at least 1, the delay  $t_2 - t$  is at least 3, and the

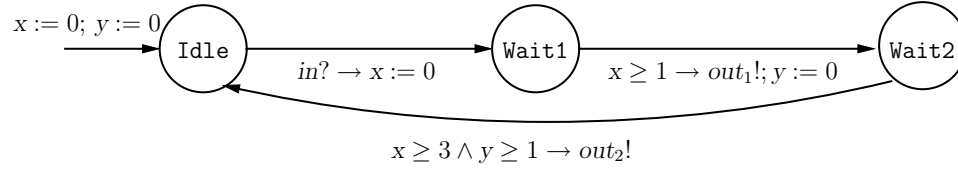


Figure 6.3: A timed process with two clocks

delay  $t_2 - t_1$  is at least 1 (the process does not accept inputs on channel *in* until it has issued both outputs). The desired constraints are expressed using two clocks  $x$  and  $y$ . Initially the mode is `idle`. When the input event occurs, the process sets the clock  $x$  to 0 and switches to the mode `Wait1`. The output event  $out_1$  can occur at any time instance when the condition  $x \geq 1$  is satisfied. At this point, the process switches to the mode `Wait2` and sets the clock  $y$  to 0. In the mode `Wait2` both clocks increase as time elapses, with the clock  $x$  capturing the time elapsed since the occurrence of input event and the clock  $y$  capturing the time elapsed since the occurrence of the first output event. The condition  $x \geq 3 \wedge y \geq 1$  associated with the output on  $out_2$  captures the desired timing constraint. Here is a sample execution of the process:

$$\begin{aligned}
 (\text{Idle}, 0, 0) &\xrightarrow{5.7} (\text{Idle}, 5.7, 5.7) \xrightarrow{\text{in}^?} (\text{Wait1}, 0, 5.7) \xrightarrow{2.6} (\text{Wait1}, 2.6, 8.3) \xrightarrow{\text{out}_1^!} \\
 &(\text{Wait2}, 2.6, 0) \xrightarrow{0.5} (\text{Wait2}, 3.1, 0.5) \xrightarrow{2} (\text{Wait2}, 5.1, 2) \xrightarrow{\text{out}_2^!} (\text{Idle}, 5.1, 2)
 \end{aligned}$$

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

## Formal Model

We can define the formal model for timed processes as an extension of asynchronous processes. The notion of input, output, and state variables, and input, output, and internal actions is identical. The additional notion is that of a clock invariant, which is a Boolean expression on state variables, and this is used to define timed transitions of duration  $\delta$ . We summarize the definition below. In this definition, given a state  $s$  and a positive real number  $\delta$ ,  $s + \delta$  denotes the state that assigns the value  $s(x) + \delta$  to every clock variable  $x$ , and the value  $s(y)$  to every variable  $y$  of type other than `clock`.

## TIMED PROCESS

A *timed process*  $TP$  consists of (1) an asynchronous process  $P = (I, O, S, Init, InActs, OutActs, Acts)$ , where some of the state variables are of type `clock`; and (2) a *clock invariant*  $ClockInv$  which is a Boolean expression over the state variables  $S$ . Inputs, outputs, states, initial states, internal transitions, input transitions, and output transitions of  $TP$  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 transition* of  $TP$  if  $s + t$  satisfies  $ClockInv$  for all  $0 \leq t \leq \delta$ .

For the timed process `TimedBuf` of Figure 6.2, 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 state variable  $s$  of enumerated type  $\{\text{Empty}, \text{Full}\}$ ,  $x$  of type `msg`, and  $y$  of type `clock`;
- the initialization expression is

$$y = 0 \wedge s = \text{Empty}$$

- the input action for the variable  $in$  is given by the expression

$$[s = \text{Empty} \wedge s' = \text{Full} \wedge x' = in \wedge y' = 0] \vee [s = \text{Full} \wedge \text{same}(s, x, y)]$$

- the output action for the variable  $out$  is given by the expression

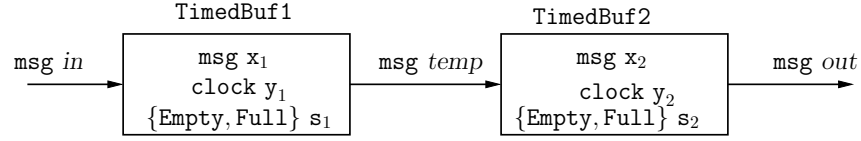
$$s = \text{Full} \wedge y \geq 2 \wedge out = x \wedge s' = \text{Empty} \wedge \text{same}(x, y)$$

- it has no internal actions;
- the clock invariant  $ClockInv$  is given by the expression

$$s = \text{Empty} \vee (s = \text{Full} \wedge y \leq 3)$$

Note that when the clock invariant annotation is missing on a mode in the state-machine representation, it is assumed to be the constant 1 (that is, when the mode is `Empty`, there is no constraint on how much time can elapse). The translation from the state-machine notation to the formal definition of timed processes can be automated.

Note that the formal definition of a time transition requires that starting in a state  $s$ , a timed transition of duration  $\delta$  is possible if the state  $s + t$  satisfies the clock invariant at every time  $t$  during the interval  $[0, \delta]$ . Typically the

Figure 6.4: Composition of two instances of `TimedBuf` processes

expressions used in clock invariants are convex, so it suffices to check that the final state  $s + \delta$  satisfies the clock invariant.

As in 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 either an input transition, or an output transition, or an internal transition, or a timed transition, at every step. Note that input, output, and internal transitions are interleaved as in an asynchronous process. However, during a timed transition the clocks belonging to different processes all increase together reflecting the passage of the same global time, and thus, a timed transition is executed synchronously. This is why sometimes this model is called a *partially synchronous model*.

### Process Composition

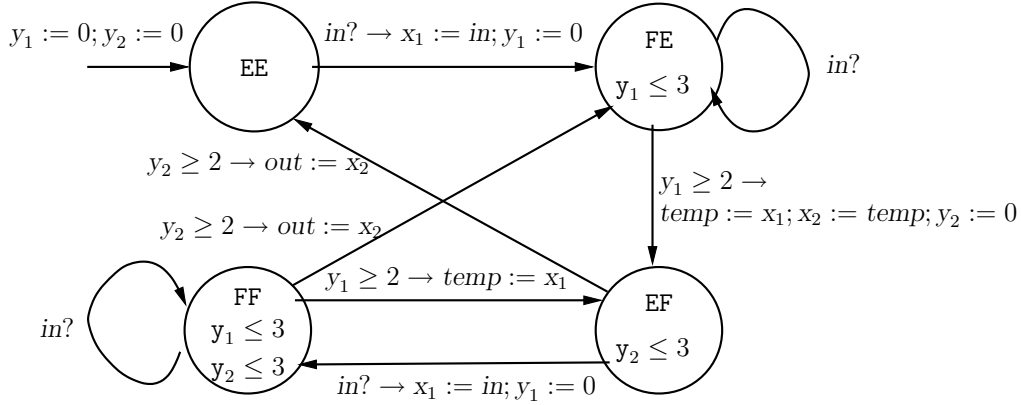
Timed processes can be put together using block diagrams. Operations such as instantiation, variable renaming, and output hiding are defined in the obvious way. Let us consider the operation of composing timed processes. To compose two timed processes, we compose the corresponding asynchronous processes using the composition operation for asynchronous processes, and the clock invariant for the composed process is simply the *conjunction* of the clock invariants of the component processes.

#### PROCESS COMPOSITION

Consider two compatible timed processes  $TP_1 = (P_1, ClockInv_1)$  and  $TP_2 = (P_2, ClockInv_2)$ . Then the parallel composition  $TP_1 | TP_2$  is the timed process whose asynchronous process is  $P_1 | P_2$  and whose clock invariant is  $ClockInv_1 \wedge ClockInv_2$ .

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 clock invariants means that a timed transition of duration  $\delta$  is possible in the composite process only if it is acceptable for both component processes to wait for a duration of  $\delta$ . The number of clocks in the composition is sum of the number of clocks in component processes.

To understand how the composition works, let us describe composition of two instances of timed processes `TimedBuf` connected in a series. Figure 6.4 shows

Figure 6.5: State machine for composition of two `TimedBuf` processes

two instances with their variables renamed appropriately. The behavior of the parallel composition of the two processes is captured by the state machine shown in Figure 6.5. Since each component has two possible modes, the composite process has 4 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 `FE` (that is, `s1` is `Full` and `s2` is `Empty`). The variables `x1` and `y1` are updated according to the input action of the first process, and the variables `x2` and `y2` stay unchanged. The switch from the mode `FE` to `EF` occurs when the output event for the first process is synchronized with the input event for the second.

The mode `FF` corresponds to the case when both the processes are in the `Full` mode. The clock invariant of this mode is the conjunction  $y_1 \leq 3 \wedge y_2 \leq 3$ , and reflects the synchronization of the two component processes on timed transitions. The mode can change in two ways: if the second component issues its output on `out`, the mode changes to `FE`, and if the first component issues its output on `temp`, the mode changes to `EF`. In the latter case, the second process simply ignores the value output by the first process.

## 6.2 Timing-based Protocols

### 6.2.1 Timing-based Distributed Coordination

Timing assumptions can be used to solve distributed coordination problems. In Chapter 3, 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 this bound can be used to solve consensus using only atomic registers. Below

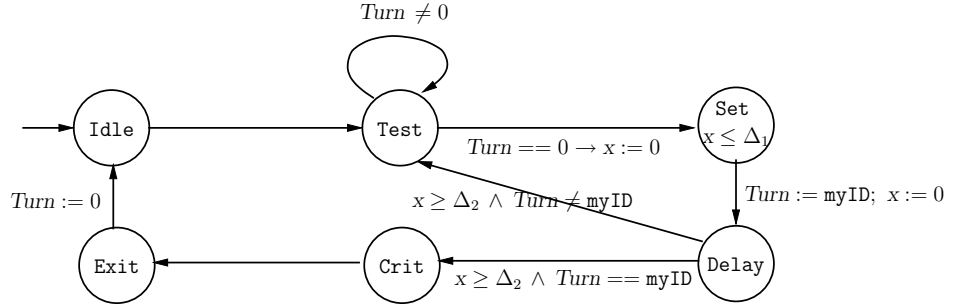


Figure 6.6: Timing-based Mutual Exclusion

we describe a timing-based solution to another classical coordination problem—mutual exclusion, and the same ideas can be used to solve consensus.

Suppose we have a set of asynchronous processes that need access to a critical shared resource. At any time, only one process should be using the shared resource. The allocation of the resource is not governed by a central coordinator, but processes need to coordinate among themselves to ensure such a mutually exclusive access. We assume that each process has a unique identifier that is different from 0, and processes can communicate using atomic registers. Initially, a process starts in the mode `Idle`. It accesses the shared resource in the mode `Crit`, classically known as the *critical section*. We want to design the *entry code* that the process should execute when it wants to switch from `Idle` to `Crit`, and the exit code that the process should execute when it is finished its job in the critical section before returning to the idle mode. The safety requirement is that no two processes should be in `Crit` mode 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.

Let us consider a simple solution that uses a single shared register `Turn` and timing assumptions as shown in Figure 6.6. Initially, `Turn` is 0. When a process  $P$  wants to enter the critical section, it tests whether `Turn` is 0 or not. If it finds `Turn` to be non-zero, it waits until `Turn` becomes 0. Then it sets `Turn` to its own identifier. Let  $t_1$  be the time when the process  $P$  finds `Turn` to be 0, that is, the time of the transition from the mode `Test` to the mode `Set`, and let  $t_2$  be the time when it sets `Turn` to its own identifier, that is, the time of the transition from the mode `Set` to the mode `Delay`. It is possible that some other process  $P'$  trying to enter the critical section also finds `Turn` to be 0 during the time interval from  $t_1$  to  $t_2$ . Additional timing-based synchronization is used to resolve contention among all such processes. The clock-invariant  $x \leq \Delta_1$  captures the assumption that the delay between the times  $t_1$  and  $t_2$ , that is, the time spent by the process  $P$  in the mode `Set`, is at most  $\Delta_1$ . After setting `Turn` to its identifier, the process  $P$  proceeds to check the value of `Turn` again. If it finds that the value is unchanged (that is, equal to its own identifier), it proceeds to the critical

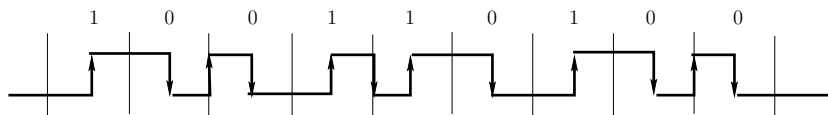


Figure 6.7: Manchester encoding of the bit sequence 100110100

section, but if it finds that the value has been overwritten by someone else, it retries by switching back to the **Test** mode. The switches out of the **Delay** mode are guarded with the condition  $x \geq \Delta_2$ , and this ensures that the process waits in the **Delay** mode for at least  $\Delta_2$  time before checking the value of *Turn*. Let  $t_3$  be this time when the process checks the value of *Turn* and leaves the mode **Delay**. Now it is easy to see that if  $\Delta_2 > \Delta_1$ , then any competing process that switched from **Test** to **Set** during the interval  $[t_1, t_2]$  would have left the mode **Set** by time  $t_3$ . We don't need to worry about processes that had not entered the mode **Set** by time  $t_1$  since they would be blocked in the mode **Test** since *Turn* is non-zero. Among all competing processes in the mode **Set**, if  $P'$  is the last process to set *Turn* to its identifier, then every process will find *Turn* to be the identifier of  $P'$  when it checks *Turn* after waiting in the mode **Delay** for  $\Delta_2$  time. Such a process  $P'$  will be the unique winner and will proceed to its critical section. When it leaves the critical section, it sets *Turn* back to 0 so that processes waiting in the mode **Test** can compete again to enter the critical section.

## 6.2.2 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.

A stream of bits is communicated using high and low voltage settings on a communication bus. Time is divided in slots of fixed length, and in each slot a single bit is sent in the middle of the slot by a change in the voltage. A 0 bit is sent as a falling edge from high voltage to low voltage, and a 1 bit is sent as a rising edge from low voltage to high voltage. If the bits to be sent in consequent slots are the same, 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 6.7.

The clocks of the sender and the receiver are not perfectly aligned and have a specified tolerance. The receiver does not know when the first time slot begins, but both the sender and the receiver know the agreed upon width of the slots. The sender and receiver synchronize the beginning of transmission by requiring low voltage when no information is exchanged, and assuming that each message begins with 1. The receiver does not know the length of the message in advance,

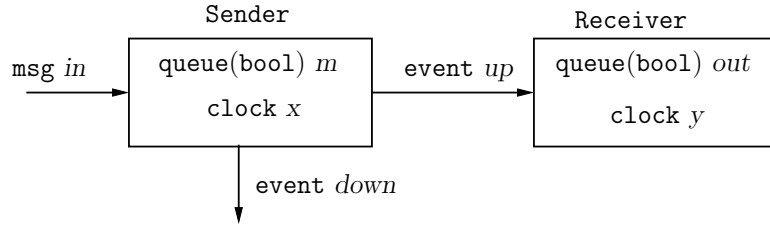


Figure 6.8: Block diagram for the audio control protocol

but can infer the end of the current message when it detects no information sent in a slot. The interesting aspect of the protocol is due to the constraint that the receiver cannot reliably detect a falling edge. Thus all decoding must happen 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, let us assume that all message end in 00.

For design and analysis of the required protocol, let us assume that the length of the time slot is 4 time units, and the error in the clocks of the sender and the receiver is  $\epsilon$  per time unit. What this means is that if sender keeps the voltage high for 4 time units according to its internal clock, the actual elapsed time may be anywhere between  $4 - 4\epsilon$  to  $4 + 4\epsilon$ . Similarly, if the receiver finds the delay between two consecutive rising edges to be less than 3 time units, we can only assume that the corresponding delay is less than  $3 + 3\epsilon$ .

For a given value of the clock tolerance  $\epsilon$ , the correctness requirement is that every message is correctly decoded by the receiver. More generally, we would like to determine the largest value of  $\epsilon$  for which the protocol works correctly.

The block diagram for the system composed of the sender and the receiver is shown in Figure 6.8. The state machine for the sender is shown in Figure 6.9. For the sender process, it receives the message to be transferred on the input channel *in*. The message type `msg` is sequence of Boolean values. It 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 *x* is used to capture the timing constraints. The state machine for the sender process is shown in Figure fig:audio-send.

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*, except the

first bit, which is required to be 1. The process sets its clock to 0, and switches to mode **B**. After waiting for 2 time units, it issues the *up* event to change the voltage to high in the middle of the slot. Note that all timing constraints are modified to reflect the possible error in the measurement of time. The process removes the next bit to be transmitted from  $m$ . If this bit is 1, it switches to mode **C**, where it waits for 2 time units, then changes the voltage to low by issuing a *down* event, and then returns to the mode **B** in order to transmit 1 in the following time slot. If the bit is 0, it switches to the mode **D**. In mode **D**, it waits for 4 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 message queue. If this bit is 1 (different from the last processed), it switches to the mode **E** where it waits for 4 time units before issuing the *up* event. If this bit 0 (same as the last processed), it waits in the mode **F** for 2 time units, raises the voltage, and then waits in the mode **G** for 2 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  $m$  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 6.9. In the description  $F(m)$  stands for the first element of the queue  $m$ , and the action  $\text{Deq}(m)$  removes the first element from the queue  $m$ . We use  $2^{-\epsilon}$  as an abbreviation for  $2 - 2\epsilon$ , and  $2^{+\epsilon}$  for  $2 + 2\epsilon$ .

The receiver process is shown in Figure 6.10. The receiver uses a clock  $y$ , 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, and 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, the exact duration till 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 to 5, 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  $\epsilon$  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  $[5, 7]$ , it means that the bit 0 is sent, and if the next rising edge is detected after a delay in the interval  $[7, 9]$ , it means that the bits 0 and 1 were sent (no rising edge is required for a 0 sandwiched between two 1's). In the mode **Last0**, similar logic is applied by partitioning expected delays until the next *up* event into different categories: between 3 and 5 means the next bit is 0, between 5 and 7 means that bits 0 and 1 are sent, and if no event is detected for 7 time units, the receiver concludes end of transmission (the message ends with 0) and returns to **Idle**.

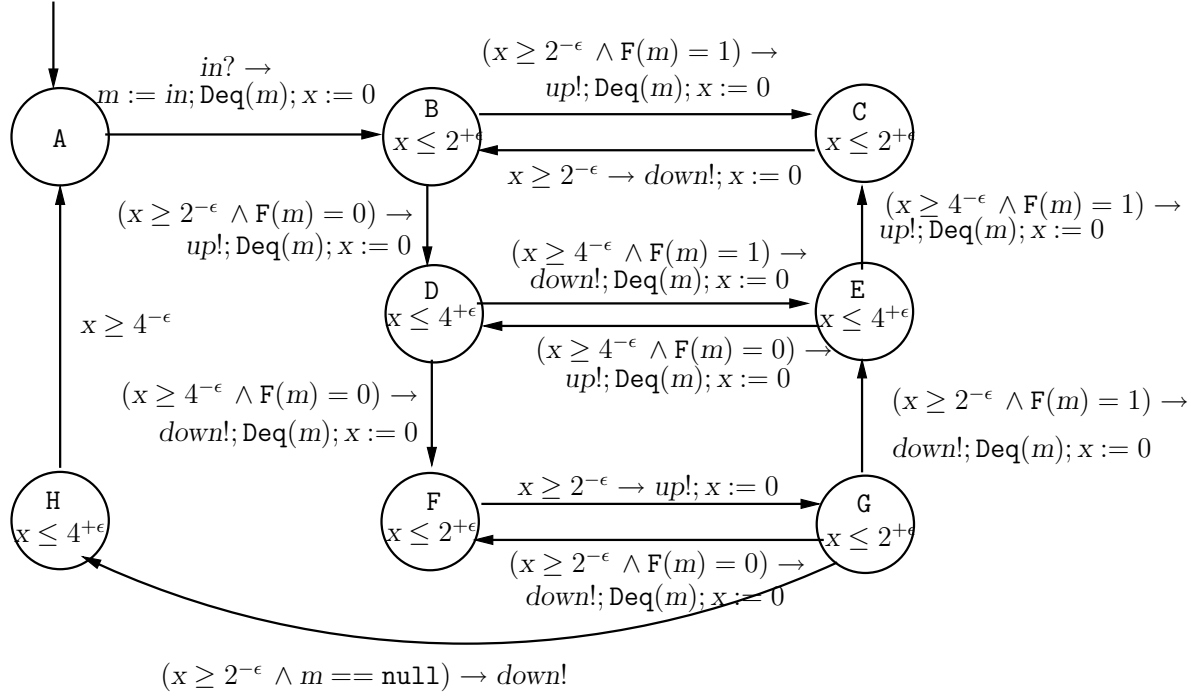


Figure 6.9: The sender process in the audio control protocol

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

### 6.3 Timing Analysis

Given a timed process *TP*, which may be expressed as a composition of a number of timed processes including a safety monitor, and a property  $\varphi$  over the state variables, the goal of the reachability analysis is to check whether  $\varphi$  is an invariant of *TP*, and if not, produce a counter-example. One challenge in applying on-the-fly enumerative depth-first-search (as studied in Chapter 2.3) is the fact that the state of a timed process includes the values for the clocks, which are real-valued variables. As a result, we must develop symbolic or constraint-based

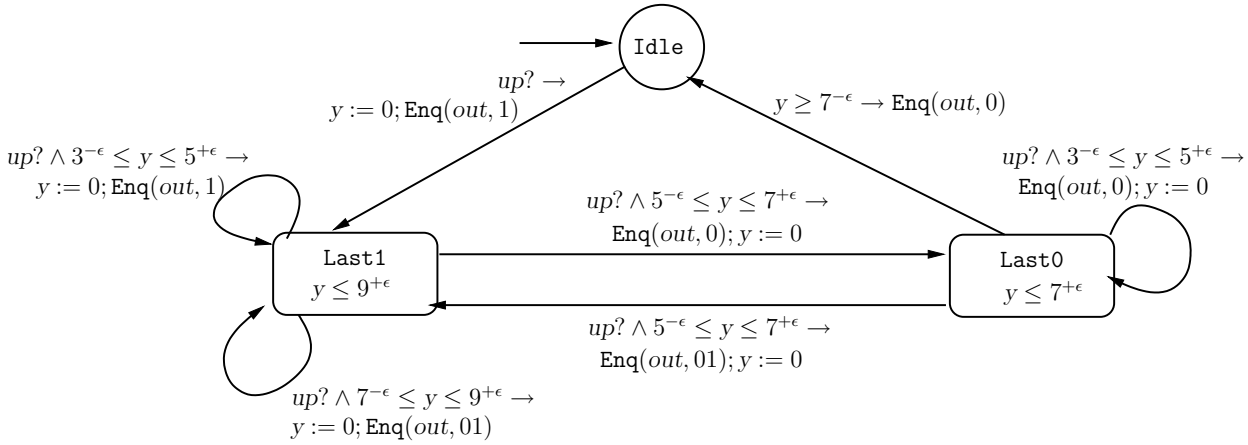


Figure 6.10: The receiver process in the audio control protocol

techniques to handle sets of values for clocks. In this section, we discuss an analysis technique based on a matrix-based representation for the most commonly occurring way in which clocks are used in modeling of timed systems.

We restrict ourselves to timed processes whose specification meets the following constraints:

1. The only value assigned to a clock is 0.
2. The only tests on clocks are of the form  $x \leq c$  and  $x \geq c$ , for some integer constant  $c$ , that is, the only atomic expressions used in the clock invariants and actions compare a clock 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 delays between events. The analysis can be easily adopted to handle rational constants. In the audio control protocol of Section 6.2.2, the model has constraints of the form  $5 - 5\epsilon \leq y \leq 5 + 5\epsilon$ , for  $\epsilon = 1/15$ . To handle such models we can simply multiply all constants by a factor of 15 to make them integers, without affecting changing the executions that are possible in the model. The analysis can also be extended to accommodate strict constraints of the form  $x < 2$ . This requires tagging each integral bound with a Boolean flag that indicates whether the associated constraint is strict or non-strict.

### Timing analysis example

To explain the analysis technique, let us consider the timed process shown in Figure 6.11. It uses two clocks,  $x$  and  $y$ , and the example shows only the clock invariants associated with the modes and tests on clock values used in the mode

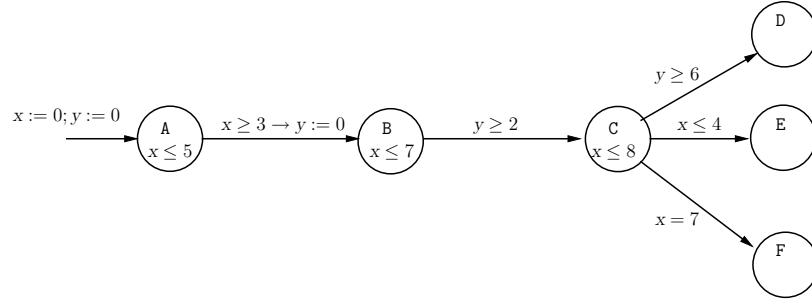


Figure 6.11: Example for analyzing timing feasibility of executions

switches. Examination of timing constraints reveals that the mode D is not reachable, that is, the path A, B, C, D cannot be traversed. Similarly, 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 clocks in the invariant of mode C and the guards in the switches out of C, but is based on the implied constraints on the values of  $x$  and  $y$  when the control reaches the mode C.

A state in this example consists of the mode which takes values from the enumerated set  $\{A, B, C, D, E, F\}$ , and the values for  $x$  and  $y$ , each of which can be a non-negative real number. We will capture a set of states using constraints of a particular form over the variables  $x$  and  $y$ , namely, bounds on the values of individual clocks and bounds on the differences between clocks.

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

$$x = 0 \wedge y = 0 \wedge x - y = 0.$$

This is shown as the set  $R_0$  in Figure 6.12. Given that the set of clock values upon entry to the mode A is described by the constraints  $R_0$ , we can calculate the set of states that can be reached using timed transitions as the process waits in the mode A. The value of  $x$  increases, but is bounded by 5 due to the clock invariant associated with the mode, and this gives the constraint  $0 \leq x \leq 5$ . Observe that during timed transitions, the difference in the clock values stays unchanged, so the constraint  $x - y = 0$  from  $R_0$  stays unchanged. These two constraints imply bounds on the value of  $y$ , and this gives the description of  $R_1$ :

$$0 \leq x \leq 5 \wedge 0 \leq y \leq 5 \wedge x - y = 0.$$

The set  $R_2$  describing the set of values upon entry to the mode B is calculated from the set  $R_1$  by intersecting it with the guard  $x \geq 3$ , and setting  $y$  to 0 to capture the effect of the assignment. The desired set  $R_2$  is described by the constraints

$$3 \leq x \leq 5 \wedge y = 0 \wedge 3 \leq x - y \leq 5.$$

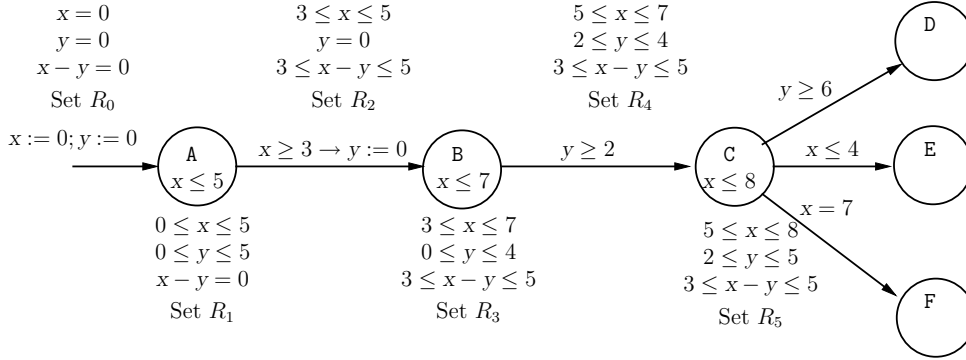


Figure 6.12: Inferring and propagating clock constraints

Again notice the implied constraint  $3 \leq x - y \leq 5$ .

This process can be repeatedly applied. The set  $R_3$  describes the set of states that are reachable as time elapses in the mode B, the set  $R_4$  describes the set of states upon entry to the mode C, and the set  $R_5$  describes the set of states that are reachable as time elapses in the mode C (see Figure 6.12). The set  $R_5$  described by

$$\leq x \leq 8 \wedge 2 \leq y \leq 5 \wedge 3 \leq x - y \leq 5$$

accurately captures the cumulative effect of timing constraints along the path leading to the mode C. Intersection of this set with the guard  $y \geq 6$  on the switch to mode D is empty, and this establishes that the mode D is unreachable. Similarly, the intersection of the set  $R_5$  with the guard  $x \leq 4$  is empty, and so the mode E cannot be reached. Intersecting the set  $R_5$  with the guard  $x = 7$  gives a non-empty set that describes the set of states upon entry to the mode F.

### Difference Bound Matrices

The most way natural of representing the constraints that arise during timing analysis is using a matrix. Let us assume that the timed process has  $k$  clocks:  $x_1, x_2 \dots x_k$ . We use a dummy clock  $x_0$  that is assumed to represent the constant 0. We also use  $\infty$  to represent a large value (that is greater than every integer), and this is used to represent absence of a bound. Then, a set of clock values is represented by a square matrix  $R$  of dimension  $k + 1$ : the  $(i, j)$ -th entry of the matrix gives the upper bound on the difference  $x_i - x_j$ . In other words, the matrix  $R$  represents the conjunction of constraints

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

The column 0 (that is, the entries  $R_{i0}$ ) give upper bounds on the clock  $x_i$ , and the row 0 (that is, the entries  $R_{0i}$ ) give upper bounds on  $-x_i$  (and thus

capture lower bounds on the value of  $x_i$ ). Such a matrix representing bounds on differences of clocks is called a *difference bounds matrix* (DBM).

Going back to our example from Figure 6.12, the set  $R_5$  is represented by the following DBM (assume that  $x$  is the first clock and  $y$  is the second clock):

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

Note that a lower bound of 3 on  $x - y$  shows up as the upper bound of -3 on the difference  $y - x$ .

The key insight regarding algorithmic and efficient inference of constraints (which is necessary to derive the constraint  $0 \leq y \leq 5$  from the constraints  $0 \leq x \leq 5$  and  $x - y = 0$  in the description of  $R_1$  in our example in Figure 6.12) is the following. Since  $R_{ij}$  is an upper bound on the difference  $x_i - x_j$  and  $R_{jl}$  is an upper bound on the difference  $x_j - x_l$ ,  $R_{ij} + R_{jl}$  is an *inferred* upper bound on the difference  $x_i - x_l$ . If the entry  $R_{il}$  is larger than  $R_{ij} + R_{jl}$  then we can “tighten” the upper bound  $R_{il}$  by replacing it with the inferred bound  $R_{ij} + R_{jl}$ . This inference reflects the transitivity of constraints, and can be readily understood by the alternative view of the DBM as a weighted directed graph. Consider the graph with  $k+1$  vertices  $x_0, x_1, \dots, x_k$ . For every pair of vertices  $x_i$  and  $x_j$ , there is an edge from  $x_i$  to  $x_j$  whose cost is equal to the entry  $R_{ij}$ . Adding  $R_{ij}$  and  $R_{jl}$  gives the cost of a path from  $x_i$  to  $x_l$  consisting of two edges. If this cost is smaller than the cost of the direct edge from  $x_i$  to  $x_l$ , we can replace the cost of this edge by the smaller value. In general, the shortest path between two vertices gives the “best” upper bound on the difference between the corresponding clocks.

Let us consider some operations that are useful on DBMs.

**Emptiness test** Given a DBM  $R$ , is the conjunction of all the constraints represented by  $R$  satisfiable? It turns out that the matrix  $R$  represents unsatisfiable set of constraints, and thus, empty set of clock values, precisely when the corresponding graph has a cycle with a negative cost. For example, consider the constraints  $x - y \leq 2$  and  $x - y \geq 3$ . The first constraint gives an edge from  $x$  to  $y$  with cost 2. The second constraint is the same as  $y - x \leq -3$ , and gives an edge from  $y$  to  $x$  with cost  $-3$ . Together they lead to a negative cost cycle, and indeed the two constraints are unsatisfiable. Any standard algorithm for detecting negative cost cycles in a graph can be used for the emptiness test.

**Canonicalization** The DBM  $R$  is said to be *canonical* iff

$$\text{for all } 0 \leq i, j, l \leq k, R_{il} \leq R_{ij} + R_{jl}.$$

That is, in a canonical matrix, every entry  $R_{il}$  represents the shortest path from  $x_i$  to  $x_l$ , and thus, the tightest bound that can be inferred on

the difference  $x_i - x_l$ . Every non-empty DBM can be converted into a canonical one by executing the shortest path algorithm (or equivalently, the transitive closure construction) on the matrix.

**Intersection** Consider two DBMs  $R$  and  $R'$  both in canonical forms. To compute the intersection of the sets of clock values 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. The intersection operation is useful for capturing the effect of clock invariants and of tests in guards on mode switches.

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

**Clock reset** Given a canonical DBM  $R$  and a clock  $x_i$ , for  $1 \leq i \leq k$ , we can define an operation on the DBM so that the result captures the set of states that can be obtained by assigning the clock  $x_i$  to 0 starting from a state in  $R$ . The details of this operation are left for the reader.

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

### Reachability analysis

To verify safety requirements of timed protocols, we can now adopt the on-the-fly depth-first-search algorithm of Section 2.3. A state of the system is now represented as a pair  $(s, R)$ , where  $s$  records the values of all the non-clock variables and  $R$  is a non-empty canonical DBM that captures a set of clock values. The basic search mechanism stays the same. In particular, states are explored and examined on demand, and the algorithm terminates as soon as a violation of the safety property is encountered. The main differences are noted below:

**Timed Transitions** For a state  $(s, R)$ , one possible successor state is obtained by considering the effect of letting time elapse using a timed transition. For this purpose, the algorithm first intersects  $R$  with the clock invariant corresponding to  $s$ , then updating the matrix to reflect elapse of time (by setting  $(i, 0)$  entries to  $\infty$ ), and again intersecting with the clock invariant corresponding to  $s$ . Note that the clock invariant corresponding to each  $s$  also needs to be represented as a DBM (and this is possible assuming it

is a conjunction of constraints that put bounds on clock values, as is the case in all our examples). At every step, the DBM is tested for emptiness, and if non-empty, is made canonical.

**Transitions** For a state  $(s, R)$ , the successor corresponding to a mode switch is computed in the following steps. First, we compute the intersection of  $R$  and the DBM that captures the constraints on clock values for the switch to happen. If this intersection is empty, then this switch is not feasible, and else, the result is made canonical. Then, the component  $s$  corresponding to non-clock variables is updated according to the description of the action corresponding to the switch. If the action involves setting a clock to 0, then the clock reset operation is applied to the DBM part.

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

Observe that the algorithm has a mix of enumerative search and symbolic search: non-clock variables are processed by explicitly enumerating their values and clock variables are manipulated using constraints. The algorithm is implemented in tools such as the model checker UPPAAL (see [www.uppaal.com](http://www.uppaal.com)) with many optimizations. The same ideas can also be applied to modify the nested depth-first-search algorithm of Chapter 4 for checking liveness properties of timed systems.