Timed Systems

**Concepts:**
programs that are concerned with **passage of time**
synchronize processes through **global clock**

**Models:**
model time through shared ‘**tick**’ action

**Practice:**
implement processes as **Timed objects**
control progress of time using **TimeManager thread**

Concurrency: timed systems
timed vs. real-time systems

So far we have not been concerned with passage of time: the correctness of the models/implementations depended on the order of actions, but not their duration.

With **timed systems**, the correctness *does* depend on performing actions by specific times. We make the simplifying assumption that program execution proceeds sufficiently quickly such that, when related to the time between external events, it can be ignored.

With **real-time systems**, we do take the duration of program execution into account, and we typically specify and subsequently guarantee an upper bound to execution time. Real-time systems are beyond the scope of this chapter.
12.1 Modeling timed systems

To model time, we adopt a discrete model of time introduces *timing uncertainty*, but can increase accuracy by allowing more ticks per second.

Passage of time is signaled by successive ‘tick’ s of a clock shared by all processes that need to be aware of passing of time.

Consider detection of double mouse clicks **within** D ticks:

```plaintext
DOUBLECLICK(D=3) =
  (tick -> DOUBLECLICK | click -> PERIOD[1]),
PERIOD[t:1..D] =
  (when (t==D) tick -> DOUBLECLICK
  |when (t<D) tick -> PERIOD[t+1]
  |click -> doubleclick -> DOUBLECLICK)
```

*LTS? Trace…*
timing consistency

Producer produces item every $T_p$ seconds and consumer consumes item every $T_c$ seconds.

CONSUMER($T_c=3$) = (item -> DELAY[1] | tick -> CONSUMER),
DELAY[$t:1..T_c$] = (when (t==T_c) tick -> CONSUMER |when (t<T_c) tick -> DELAY[t+1] ).
PRODUCER($T_p=3$) = (item -> DELAY[1]),
DELAY[$t:1..T_p$] = (when (t==T_p) tick -> PRODUCER |when (t<T_p) tick -> DELAY[t+1] ).

||SAME = (PRODUCER(2) || CONSUMER(2)).
||SLOWER = (PRODUCER(3) || CONSUMER(2)).
||FASTER = (PRODUCER(2) || CONSUMER(3)).

Safety?

Deadlock is a “time-stop”
maximal progress

Use a store for items to connect producer and consumer.

\[
\begin{align*}
\text{STORE}(N=3) &= \text{STORE}[0], \\
\text{STORE}[i:0..N] &= (\text{put} \rightarrow \text{STORE}[i+1] \\
&\quad \text{when } (i > 0) \text{ get} \rightarrow \text{STORE}[i-1]) \\
|\|\text{SYS} &= (\text{PRODUCER}(1)/\{\text{put/item}\} \\
&\quad |\|\text{CONSUMER}(1)/\{\text{get/item}\} \\
&\quad |\|\text{STORE} \\
\end{align*}
\]

If items are consumed at the same rate as they are produced, then surely the store should not overflow?
model analysis

Trace to property violation in STORE:

put
tick
put
tick
put

Consumer always chooses tick over get action and store overflows!

To ensure maximal progress of other actions, make the tick action low priority.

||NEW_SYS = SYS>>{tick}.

To ensure progression of time, make sure tick occurs regularly in an infinite execution.

progress TIME = {tick}

Concurrency: timed systems
ensuring progression of time

The following process violates the TIME progress property:

\[
\begin{aligned}
PROG &= (\text{start} \rightarrow \text{LOOP} \mid \text{tick} \rightarrow \text{PROG}), \\
\text{LOOP} &= (\text{compute} \rightarrow \text{LOOP} \mid \text{tick} \rightarrow \text{LOOP}).
\end{aligned}
\]

||CHECK = PROG>>\{\text{tick}\}.
progress TIME = \{\text{tick}\}.

To fix this, we can include an action that terminates the loop and forces a tick action.

\[
\begin{aligned}
PROG &= (\text{start} \rightarrow \text{LOOP} \mid \text{tick} \rightarrow \text{PROG}), \\
\text{LOOP} &= (\text{compute} \rightarrow \text{LOOP} \\
&\quad \mid \text{tick} \rightarrow \text{LOOP} \\
&\quad \mid \text{end} \rightarrow \text{tick} \rightarrow \text{PROG} \\
&\quad )).
\end{aligned}
\]
Modeling output in an interval

Produce an output at any time after \( \text{Min} \) ticks and before \( \text{Max} \) ticks.

\[
\text{OUTPUT}(\text{Min}=1, \text{Max}=3) = \\
\quad (\text{start} \rightarrow \text{OUTPUT}[1] \\
\quad | \text{tick} \rightarrow \text{OUTPUT} \\
\quad ),
\]

\[
\text{OUTPUT}[t:1..\text{Max}] = \\
\quad (\text{when} \ (t>\text{Min} \ \&\& \ t\leq\text{Max}) \ \text{output} \rightarrow \text{OUTPUT} \\
\quad | \text{when} \ (t<\text{Max}) \quad \quad \quad \quad \text{tick} \rightarrow \text{OUTPUT}[t+1] \\
\quad ).
\]
Modeling jitter

Produce an output at a predictable rate, but at any time within a given period.

\[
\text{JITTER}(\text{Max}=2) = \\
\quad (\text{start} \to \text{JITTER}[1] \\
\quad | \text{tick} \to \text{JITTER}) \\
\]

\[
\text{JITTER}[t:1..\text{Max}] = \\
\quad (\text{output} \to \text{FINISH}[t] \\
\quad | \text{when (t<Max) tick} \to \text{JITTER}[t+1]) \\
\]

\[
\text{FINISH}[t:1..\text{Max}] = \\
\quad (\text{when (t<Max) tick} \to \text{FINISH}[t+1] \\
\quad | \text{when (t==Max) tick} \to \text{JITTER}) \\
\]
Modelling timeout

Use of timeout to detect the loss of a message or failure in a distributed system. Use a separate TIMEOUT process:

\[
\text{TIMEOUT}(D=1) = (\text{setT0} \rightarrow \text{TIMEOUT}[0] \\
\quad | \{\text{tick}, \text{resetT0}\} \rightarrow \text{TIMEOUT})
\]

\[
\text{TIMEOUT}[t:0..D] = \\
\quad (\text{when} (t < D) \text{ tick} \rightarrow \text{TIMEOUT}[t+1] \\
\quad | \text{when} (t == D) \text{ timeout} \rightarrow \text{TIMEOUT} \\
\quad | \text{resetT0} \rightarrow \text{TIMEOUT})
\]

\[
\text{REC} = (\text{start} \rightarrow \text{setT0} \rightarrow \text{WAIT}),
\]

\[
\text{WAIT} = (\text{timeout} \rightarrow \text{REC} \\
\quad | \text{receive} \rightarrow \text{resetT0} \rightarrow \text{REC}).
\]

\[
||\text{RECEIVER}(D=2) = (\text{REC} || \text{TIMEOUT}(D))
\]

\[
\gg\{\text{receive, timeout, start, tick}\}
\]

\[
@\{\text{receive, timeout, start, tick}\}.
\]

Interface actions depend on the system into which RECEIVER is placed – so we should not apply maximal progress to these actions within the RECEIVER process but later at the system level. Consequently, we give interface actions the same priority as the tick action.

Minimized LTS?
12.2 implementing timed systems

- **Thread-based approach**
  - translate active entities in model into threads in implementation
  - use `sleep()` and timed `wait()` to synchronize with time

- **Event-based approach**
  - translate active entities in model into objects that respond to timing events
  - `tick` actions in model become events broadcast by a time manager to all program entities that need to be aware of passage of time

- **Use event-based approach in this chapter**
  - more direct translation from model to implementation
  - more efficient for timed system with many activities (avoids context-switching overheads)
timed objects

Each process which has a **tick** action in its alphabet becomes a **timed** object in the implementation.

```java
interface Timed {
    public void pretick() throws TimeStop;
    public void tick();
}
```

Time manager implements a two-phase event broadcast:
1. **pretick()**: object performs all output actions that are enabled in current state
2. **tick()**: object updates its state with respect to inputs and passage of time
countdown timer

COUNTDOWN(N=3) = COUNTDOWN[N],
COUNTDOWN[i:0..N] = (when (i>0) tick -> COUNTDOWN[i-1]
  |when (i==0) beep -> STOP
)

class TimedCountDown implements Timed {
  int i;  TimeManager clock;

  TimedCountDown(int N, TimeManager clock) {
    i = N; this.clock = clock;
    clock.addTimed(this);  // register with time manager
  }

  public void pretick() throws TimeStop {
    if (i == 0) {
      // do beep action
      clock.removeTimed(this);  // unregister = STOP
    }
  }

  public void tick() { --i; }
}
timed producer-consumer

class ProducerConsumer {
    TimeManager clock = new TimeManager(1000);
    Producer producer = new Producer(2);
    Consumer consumer = new Consumer(2);

    ProducerConsumer() {clock.start();}

    class Producer implements Timed {
        ...
    }

    class Consumer implements Timed {
        ...
    }
}
timed producer-consumer - class Producer

PRODUCER(Tp=3) = (item -> DELAY[1]),
DELAY[t:1..Tp] = (when (t==Tp) tick -> PRODUCER
|when (t<Tp) tick -> DELAY[t+1]
).
CONSUMER(Tc=3) = (item -> DELAY[1] | tick -> CONSUMER),
DELAY[t:1..Tc] = (when (t==Tc) tick -> CONSUMER
    |when (t<Tc) tick -> DELAY[t+1]
    ).

class Consumer implements Timed {
    int Tc,t; Object consuming = null;
    Consumer(int Tc) {
        this.Tc = Tc; t = 1;
        clock.addTimed(this);
    }
    public void item(Object x) throws TimeStop {
        if (consuming != null) throw new TimeStop();
        consuming = x;
    }
    public void pretick() {}
    public void tick() {
        if (consuming == null) { return; }
        if (t < Tc) { ++t; return; }
        if (t == Tc) { consuming = null; t = 1; }
    }
}
The `ImmutableList` class provides access to a list that does not change while it is enumerated.
Concurrent Systems

19

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Concurrent Systems

Conn
12.3 parcel router

Parcels are dropped in a chute and fall by gravity; each parcel has a destination code, which can be read so that the parcel is routed to the correct destination bin. A switch can only be moved if there is no parcel in its way.
parcel router – structure diagram

CONCRETE_ROUTER

GEN(T)  top:STAGE(1)  right:STAGE(0)  left:STAGE(0)  BIN(3)  BIN(2)  BIN(1)  BIN(0)

enter  enter  enter  right  right  dest(3)

left  left  left  left  dest(2)

right  right  right  right  dest(1)

dest(0)  dest(0)  dest(0)
parcel router – system specification

||PARCEL_ROUTER(T=4) =
(top:STAGE(1) || left:STAGE(0) || right:STAGE(0)
|| GEN(T) || forall[d:0..3] BIN(d)
)/{ enter/top.enter,
    top.left/left.enter, top.right/right.enter,
    dest[0]/left.left,   dest[1]/left.right,
    dest[2]/right.left,  dest[3]/right.right,
    tick/{top,left,right}.tick
}>>{tick}@{enter,dest,tick}
**parcel router – GEN process and BIN property**

**GEN** generates a parcel every T units of time. The destination of the parcel is chosen non-deterministically.

\[
\text{range Dest} = 0..3 \\
\text{set Parcel} = \{\text{parcel}[\text{Dest}]\}
\]

\[
\text{GEN}(T=3) = (\text{enter}[\text{Parcel}] \rightarrow \text{DELAY}[1] \mid \text{tick} \rightarrow \text{GEN}), \\
\text{DELAY}[t:1..T] = \\
(\text{tick} \rightarrow \text{if } (t<T) \text{ then } \text{DELAY}[t+1] \text{ else } \text{GEN}).
\]

A destination bin is modeled as the property **BIN**, which asserts that a parcel must be delivered to the correct destination bin.

\[
\text{property BIN}(D=0) = \\
(\text{dest}[D].\text{parcel}[D] \rightarrow \text{BIN}) + \{\text{dest}[D][\text{Parcel}]\}.
\]
STAGE (L) represents a part of a parcel router at level \( L \) with two chutes, a sensor, and a switch.

\[
\text{STAGE}(L) \quad \text{initialize} \quad \text{a:CHUTE} \quad \text{b:CHUTE} \quad \text{g:SWITCH} \\
\text{sense} \quad \text{setSwitch} \\
\text{enter} \quad \text{leave} \quad \text{leave}(1) \quad \text{right} \quad \text{left} \\
\text{enter} \quad \text{leave} \quad \text{leave}(0)
\]
parcel router – STAGE process

||STAGE (L=0) =

( a:CHUTE || b:CHUTE || g:SWITCH
 || s:SENSORCONTROLLER(L)
 )/{ enter/a.enter,   b.enter/{s.sense,a.leave},
     g.enter/b.leave, s.setSwitch/g.setSwitch,
     left/g.leave[0], right/g.leave[1],
     tick/{a,b,g}.tick
}

>>{enter,left,right,tick}

@{enter,left,right,tick}.}
**parcel router – CHUTE process**

**CHUTE** models the movement of a single parcel through a segment of a physical chute. Each chute can only handle one parcel, and a parcel stays in a chute for $T$ (default 2) time units.

\[
\text{CHUTE}(T=2) = \\
(\text{enter}[p:\text{Parcel}] \rightarrow \text{DROP}[p][0] \\
| \text{tick} \rightarrow \text{CHUTE}) \\
, \\
\text{DROP}[p:\text{Parcel}][i:0..T] = \\
(\text{when } (i<T) \text{ tick } \rightarrow \text{DROP}[p][i+1] \\
| \text{when } (i==T) \text{ leave}[p] \rightarrow \text{CHUTE}) \\
). \\
\]
parcel router – SENSORCONTROLLER process

SENSORCONTROLLER detects a parcel by the parcel moving from one chute to the next. To control where the parcel has to be sent, it uses the destination of the parcel and the level of the stage of which it is part (0 indicates left and 1 indicates right).

```plaintext
range DIR = 0..1   // Direction: 0 – left, 1 – right

SENSORCONTROLLER(Level=0)
   = (sense.parcel[d:Dest]
      -> setSwitch[(d>>Level)&1]->SENSORCONTROLLER).
```
parcel router – SWITCH process

SWITCH controls the direction in which the parcel leaves. It ignores commands from the SENSORCONTROLLER process when there is a parcel in the switch (since the physical switch can not move then).

\[
\begin{align*}
\text{SWITCH(T=1)} & = \text{SWITCH}[0], \\
\text{SWITCH}[s:\text{Dir}] & = \\
& (\text{setSwitch}[x:\text{Dir}] \rightarrow \text{SWITCH}[x] \\
& | \text{enter}[p:\text{Parcel}] \rightarrow \text{SWITCH}[s][p][0] \\
& | \text{tick} \rightarrow \text{SWITCH}[s]) \\
\end{align*}
\]

\[
\begin{align*}
\text{SWITCH}[s:\text{Dir}][p:\text{Parcel}][i:0..T] & = \\
& (\text{setSwitch}[\text{Dir}] \rightarrow \text{SWITCH}[s][p][i] \\
& | \text{when (i<T) tick} \rightarrow \text{SWITCH}[s][p][i+1] \\
& | \text{when (i==T) leave[s][p]} \rightarrow \text{SWITCH}[s]) \\
\end{align*}
\]
**parcel router – ANALYSIS**

- **PARCEL_ROUTER (3) leads to property violation**
  - trace to property violation in BIN(0):
    - `enter.parcel.0 -> tick -> tick -> tick -> enter.parcel.1 -> tick -> tick -> tick -> enter.parcel.0 -> tick -> tick -> tick -> enter.parcel.0 -> tick -> dest.0.parcel.0 -> tick -> tick -> enter.parcel.0 -> tick -> dest.0.parcel.1`
  - first parcel is in switch when sensor detects second parcel and attempts to change the switch

- **PARCEL_ROUTER (4) does not lead to property violation and satisfies the TIME progress property**
interface ParcelMover {
    void enter(Parcel p) throws TimeStop;
}

interface SwitchControl {
    void setSwitch(int Direction);
}
parcel router – CHUTE implementation

```java
public class Chute implements ParcelMover, Timed {
    protected int i, T, direction;
    protected Parcel current = null;
    ParcelMover next = null;

    Chute(int len, int dir) { T = len; direction = dir; }
    public void enter(Parcel p) throws TimeStop {
        if (current != null) throw new TimeStop();
        current = p; i = 0; // package enters chute
    }
    public void pretick() throws TimeStop {
        if (current == null) return;
        if (i == T) {
            next.enter(current); // package leaves chute
            current = null;
        }
    }
    public void tick() {
        if (current == null) return;
        ++i; current.move(direction);
    }
}
```
class Switch extends Chute implements SwitchControl {
  ParcelMover left = null;
  ParcelMover right = null;
  private ParcelCanvas display;
  private int gate;

  Switch(int len, int dir, int g, ParcelCanvas d)
  { super(len, dir); display = d; gate = g; }

  public void setSwitch(int direction) {
    if (current == null)
      // nothing passing through switch
      display.setGate(gate, direction);
    if (direction == 0)
      next = left;
    else
      next = right;
  }
}

class SensorController implements ParcelMover {
    ParcelMover next;
    SwitchControl controlled;
    protected int level;

    SensorController(int level) { this.level = level; }

    // parcel enters and leaves within one clock cycle
public void enter(Parcel p) throws TimeStop {
    route(p.destination);
    next.enter(p);
}

protected void route(int destination) {
    int dir = (destination>>level) & 1;
    controlled.setSwitch(dir);
}
}
parcel router – STAGE implementation

```java
ParcelMover makeStage(
    (ParcelMover left, ParcelMover right,
     int fallDir,    // movement direction for parcel display
     int level,      // 0 or 1 as in the model
     int gate,       // identity of gate for display purposes
)
{
    // create parts and register each with TimeManager ticker
    Chute a = new Chute(16,fallDir);
    ticker.addTimed(a);
    SensorController s = new SensorController(level);
    Chute b = new Chute(15,fallDir);
    ticker.addTimed(b);
    Switch g = new Switch(12,fallDir,gate,display);
    ticker.addTimed(g);
    // wire things together
    a.next = s; s.next = b;    s.controlled = g;
    b.next = g; g.left = left; g.right = right;
    return a;
}
```
Summary

◆ Concepts
  ● programs that are concerned with passage of time
  ● synchronize processes through global clock

◆ Models
  ● model time through shared ‘tick’ action

◆ Practice
  ● event-based approach: implement processes as Timed objects that respond to timing events
  ● TimeManager thread broadcasts passing of time to Timed objects