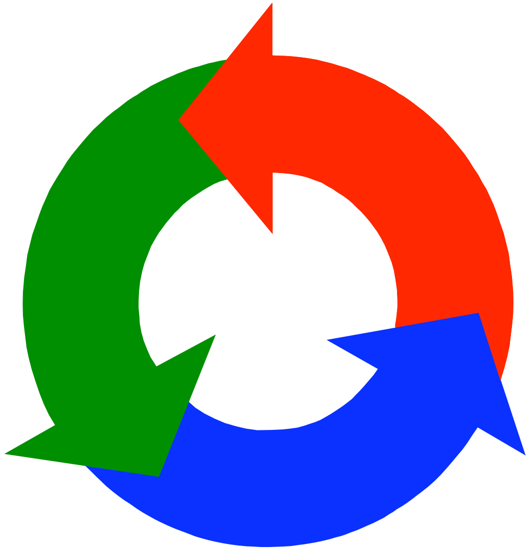
## Chapter 12

# Timed Systems



Acknowledgement: Thanks to Paul Strooper for a first draft of these slides.

# 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**

# 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:

```
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 Tp seconds and consumer consumes item every Tc seconds.

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

```
STORE(N=3) = STORE[0],
STORE[i:0..N] = (put -> STORE[i+1]
                |when (i>0) get -> STORE[i-1]
                ).
||SYS = ( PRODUCER(1)/{put/item}
        ||CONSUMER(1)/{get/item}
        ||STORE
        ).
```

*Safety?*

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
     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}
```

## ensuring progression of time

The following process violates the **TIME** progress property:

```
PROG = (start   -> LOOP | tick -> PROG),
LOOP = (compute -> LOOP | tick -> LOOP).

||CHECK = PROG>>{tick}.
progress TIME = {tick}.
```

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

```
PROG = (start   -> LOOP | tick -> PROG),
LOOP = (compute -> LOOP
       |tick    -> LOOP
       |end -> tick -> PROG
       ).
```

# Modeling output in an interval

Produce an output at any time after Min ticks and before Max ticks.

```
OUTPUT(Min=1,Max=3) =
     (start -> OUTPUT[1]
     |tick  -> OUTPUT
     ),
OUTPUT[t:1..Max] =
     (when (t>Min && t<=Max) output -> OUTPUT
     |when (t<Max)           tick -> OUTPUT[t+1]
     ).
```

*LTS? Trace…*

## Modeling jitter

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

```
JITTER(Max=2) =
     (start -> JITTER[1]
     |tick  -> JITTER
     ),
JITTER[t:1..Max] =
     (output -> FINISH[t]
     |when (t<Max)  tick -> JITTER[t+1]
     ).
FINISH[t:1..Max] =
     (when (t<Max)  tick -> FINISH[t+1]
     |when (t==Max) tick -> JITTER
     ).
```

*LTS? Trace…*

# Modeling timeout

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

```
TIMEOUT(D=1) = (setT0 -> TIMEOUT[0]
               |{tick,resetT0} -> TIMEOUT
               ),
TIMEOUT[t:0..D] =
     (when (t<D)  tick -> TIMEOUT[t+1]
     |when (t==D) timeout -> TIMEOUT
     |resetT0 -> TIMEOUT
     ).
REC = (start -> setT0 -> WAIT),
WAIT = (timeout -> REC
       |receive -> resetT0 -> REC).

||RECEIVER(D=2) = (REC || TIMEOUT(D))
                  >>{receive,timeout,start,tick}
                  @{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.

```
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]
                 ).
```

```
class Producer implements Timed {
  int Tp,t;
  Producer(int Tp) {
    this.Tp = Tp; t = 1;
    clock.addTimed(this);
  }
  public void pretick() throws TimeStop {
    if (t == 1) consumer.item(new Object());
  }
  public void tick() {
    if (t < Tp) { ++t; return; }
    if (t == Tp) { t = 1; }
  }
}
```

# timed producer-consumer - class Consumer

```
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; }
  }
}
```

## time manager

```
class TimeManager extends Thread
                 implements AdjustmentListener {
  volatile int delay;
  volatile ImmutableList clocked = null;

  TimeManager(int d) { delay = d; }
  public void addTimed(Timed el) {
    clocked = ImmutableList.add(clocked,el);
  }
  public void removeTimed(Timed el) {
    clocked = ImmutableList.remove(clocked,el);
  }
  public void adjustmentValueChanged(AdjustmentEvent e) {
    delay = e.getValue();
  }
  ...
}
```

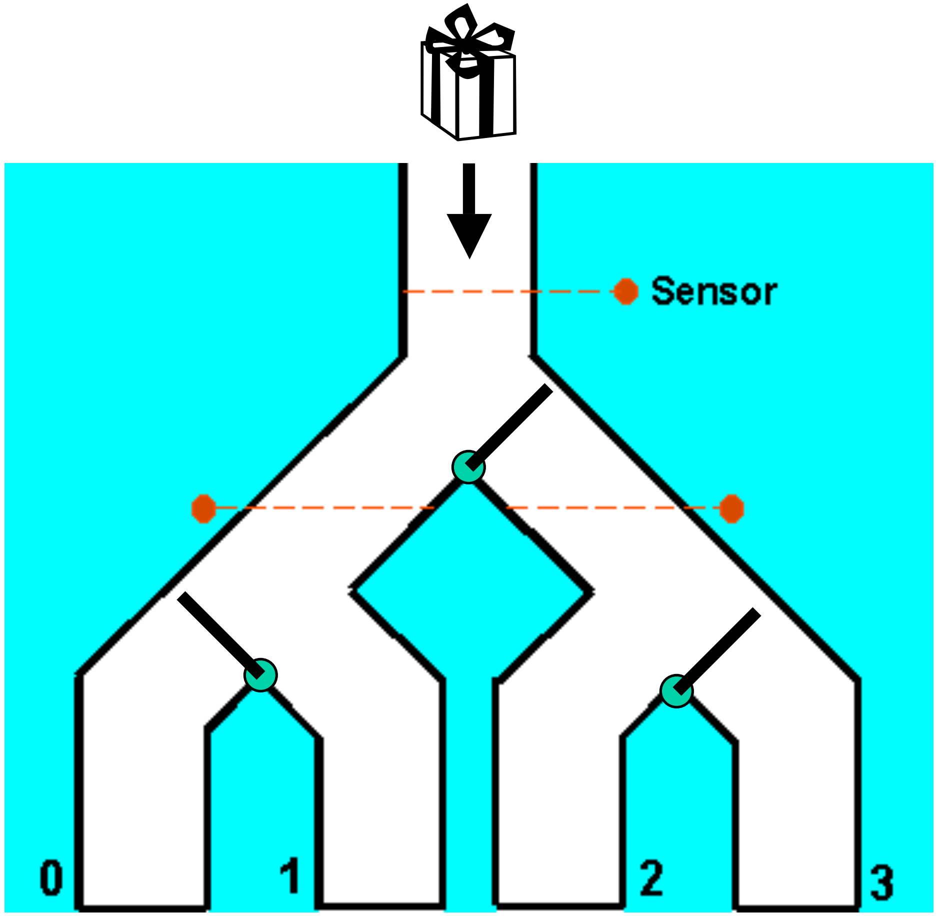
The **ImmutableList** class provides access to a list that does not change while it is enumerated.

# time manager – run  method

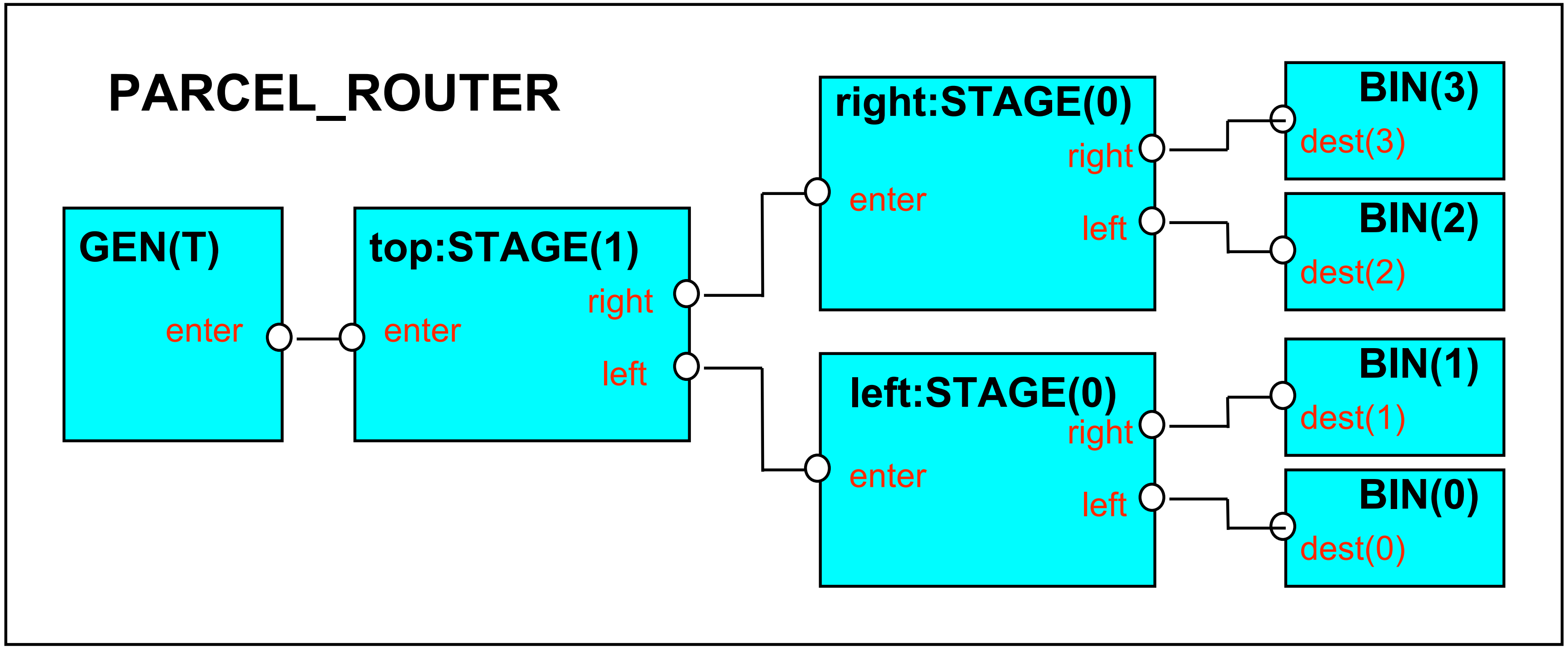
```
public void run() {
  try {
    while (true)
      try {
        Enumeration e = ImmutableList.elements(clocked);
        while (e.hasMoreElements())
          ((Timed)e.nextElement()).pretick();
        e = ImmutableList.elements(clocked);
        while (e.hasMoreElements())
          ((Timed)e.nextElement()).tick();
      } catch (TimeStop s) {
        System.out.println(“*** TimeStop”);
        return;
      }
      Thread.sleep(delay);
    }
  } catch (InterruptedException e){}
}
```

# 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



# 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.

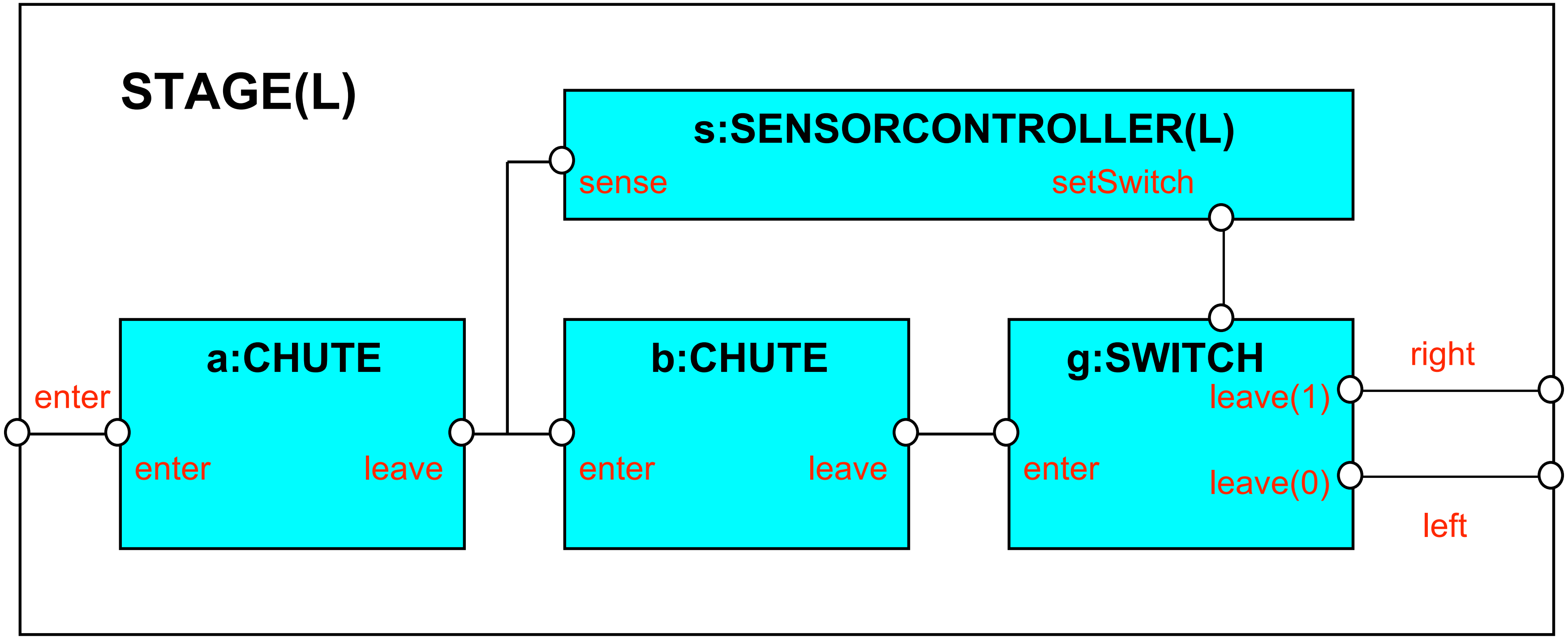
```
range Dest = 0..3
set Parcel = {parcel[Dest]}

GEN(T=3) = (enter[Parcel] -> DELAY[1] | tick -> GEN),
DELAY[t:1..T] =
     (tick -> if (t<T) then DELAY[t+1] else GEN).
```

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

```
property BIN(D=0) =
    (dest[D].parcel[D] -> BIN)+{dest[D][Parcel]}.
```

# parcel router – STAGE structure diagram

`STAGE(L)` represents a part of a parcel router at level `L` with two chutes, a sensor, and a switch.



# 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.

```
CHUTE(T=2) =
   (enter[p:Parcel] -> DROP[p][0]
   |tick            -> CHUTE
   ),
DROP[p:Parcel][i:0..T] =
   (when (i<T)  tick     -> DROP[p][i+1]
   |when (i==T) leave[p] -> 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).

```
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).

```
SWITCH(T=1)    = SWITCH[0],
SWITCH[s:Dir] =
   (setSwitch[x:Dir] -> SWITCH[x]
   |enter[p:Parcel]  -> SWITCH[s][p][0]
   |tick             -> SWITCH[s]
   ),
SWITCH[s:Dir][p:Parcel][i:0..T] =
   (setSwitch[Dir]              -> SWITCH[s][p][i]
   |when (i<T) tick             -> SWITCH[s][p][i+1]
   |when (i==T)leave[s][p] -> SWITCH[s]
   ).
```

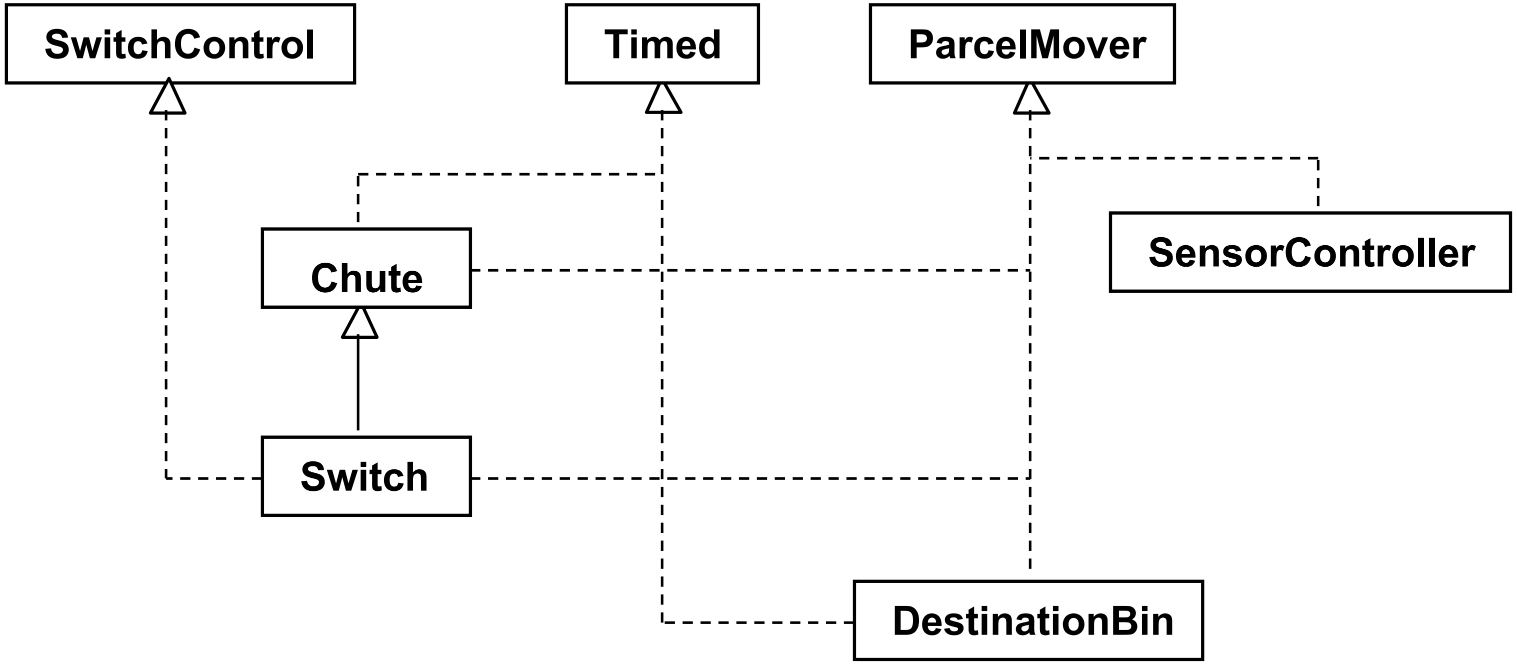
# parcel router – ANALYSIS

- **`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

# parcel router – implementation



```
interface ParcelMover {
      void enter(Parcel p) throws TimeStop;
}

interface SwitchControl {
      void setSwitch(int Direction)
}
```

# parcel router – CHUTE implementation

```
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);
  }
}
```

# parcel router – SWITCH implementation

```java
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;
      }
    }
  }
}
```

# parcel router – SENSORCONTROLLER implementation

```
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

```
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**