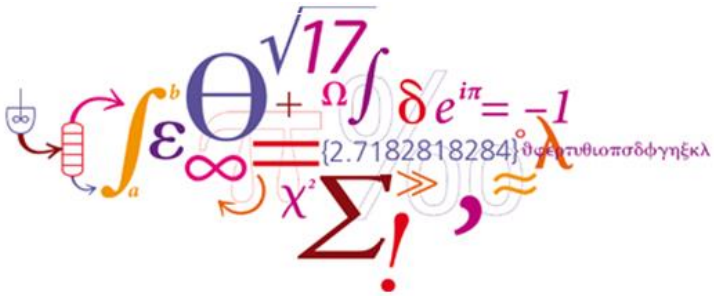
Technical University of Denmark

02224 REAL TIME SYSTEMS
Assignment 2



Group 5
Ioannis Oikonomidis (s083928)
Alexandru Malus (s131188)
Dimitrios Kosmadakis (s131200)

May 12, 2014

All members equally contributed to the report

Contents

1. Introduction

Having got some feedback from the first report we started by applying it. We made our automata simpler and we improved the Bag and the `FeedController` models. After removing unnecessary models like the `FeedBelt`, the Sensor and the `DistController` we focused on making the `Bag` model less abstract. That helped us solve the stopping problem using fixed positions. The `StoppedInitially` state was introduced in `DistributionBelt` model that helped us check with for reversal of the distribution belt and not only stopping. The updated models are used as is in this assignment and can be seen in Figure 3, Bag and Figure 2, Distribution Belt. Next task was to make the `FeedController` more realistic, making it looks as an exact mirror of the given `SingleSort` program. Figure 1, SingleSort FeedController Improved Version displays this model below.

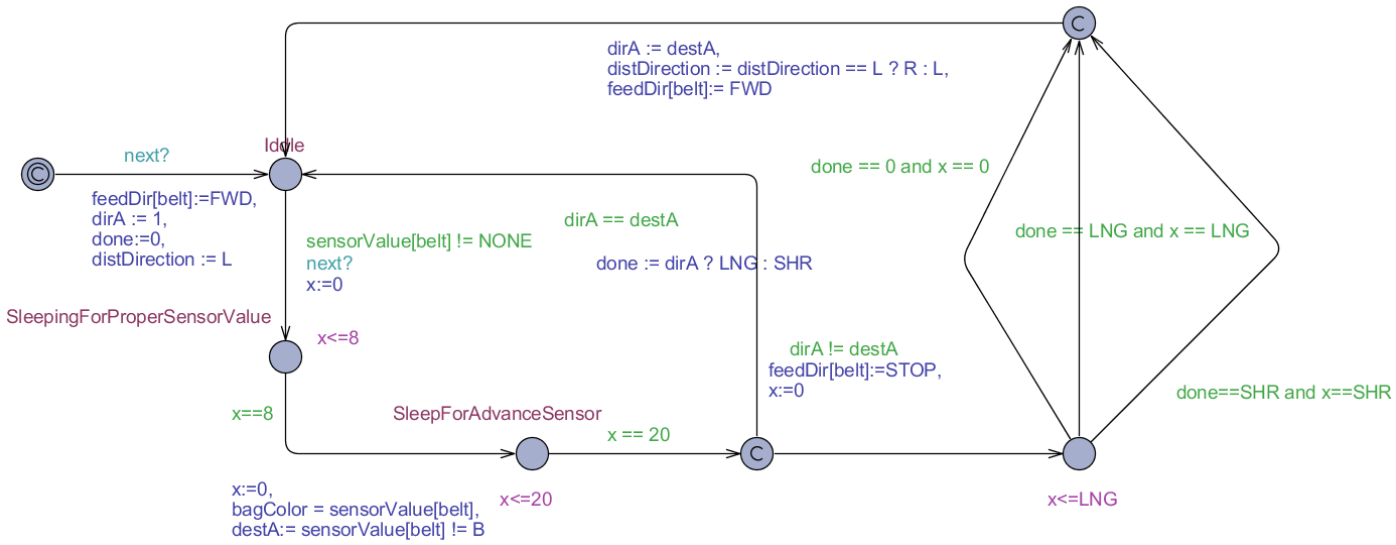


Figure 1, SingleSort FeedController Improved Version

2. Analysis

The system is a baggage sorting facility which sorts bags regarding their color (black and yellow). There are two feed belts which accept the bags, pass them into color sensors and deliver them into the distribution belt. The distribution belt is responsible for sorting them (yellow bags are delivered to the left side and black ones to the right). The sorting facility is handled by one or two users, which can either stand on the left check-in or on the right one. The two feed belts and the distribution belt are controlled by two feed belt controllers. The bag's position in the system is represented by the bag model. The goal of this assignment is to model this facility using UPPAAL modeling tool and after verifying all required properties, implement it in java and integrate it in a LEGO system using leJOS framework and a JVM RCX brick.

The time unit used in our modeling approach is 0.1 seconds (1 second = 10 time units)

2.1 Physical Parts

We describe below the physical components of the model.

2.1.1 DistributionBelt

The distribution belt model is shown in Figure 2, Distribution Belt below.

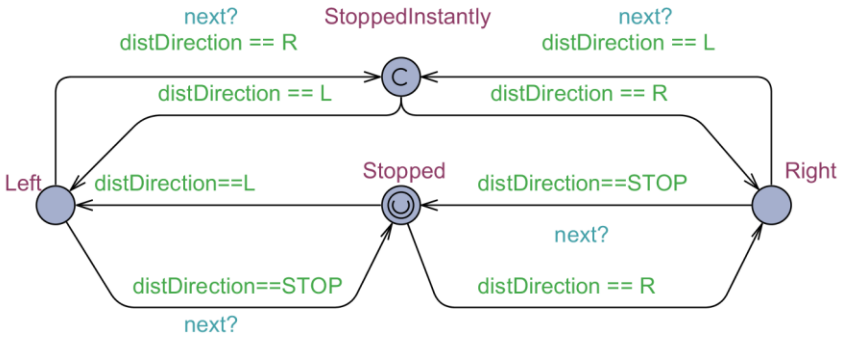


Figure 2, Distribution Belt

2.1.2 Bag

The bag model is displayed in the Figure 3, Bag below.

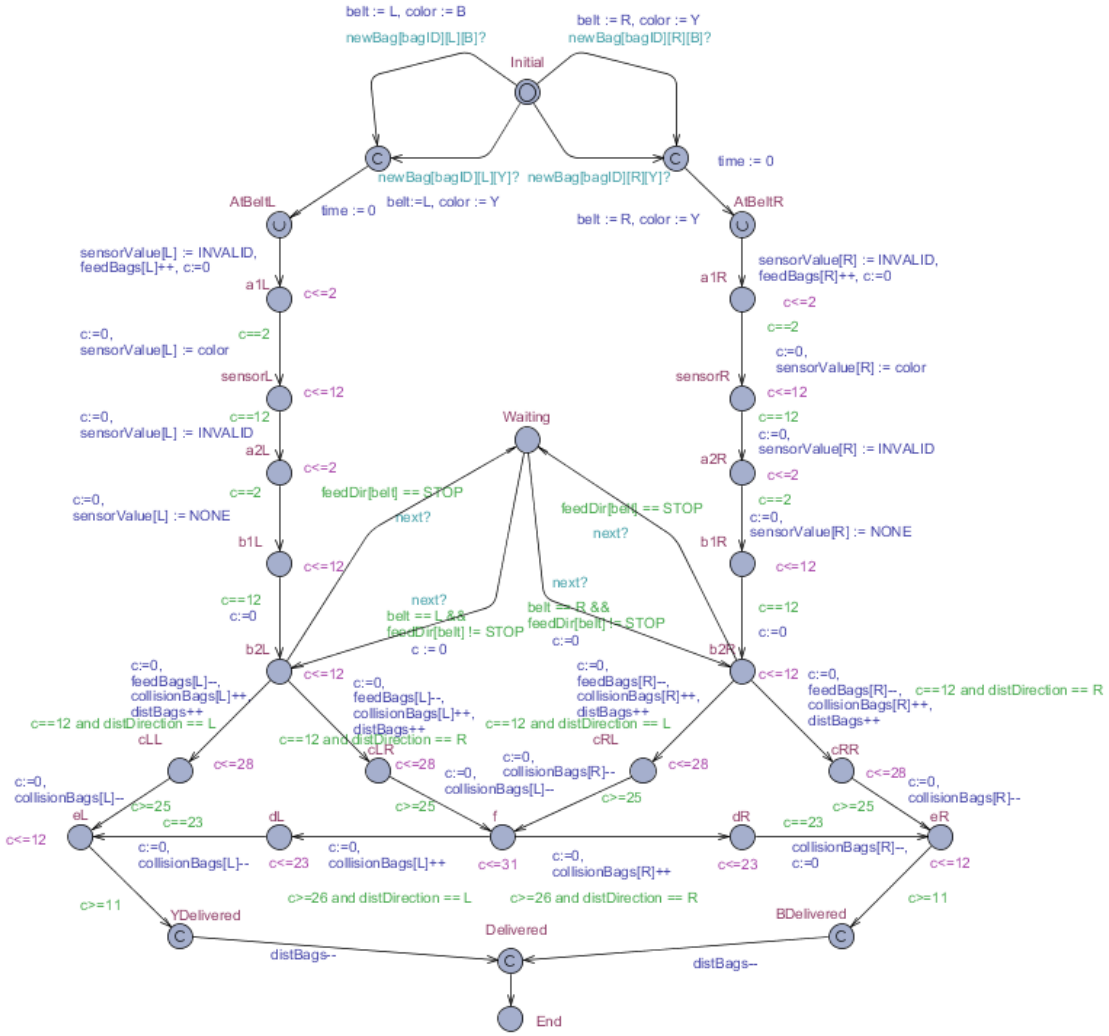


Figure 3, Bag

The simplified bag model is responsible for keeping track of the position of a baggage and setting a proper sensor value regarding its color and its position. All the guards used, are based on the timings provided in the course and on the directions of the feed belt and the distribution belt. The only communication channel between a control part of the system is the channel which synchronizes the insertion of a baggage triggered by a user. It is a three dimensional array channel which allows us to simulate and control the color, the `bagID` and the position of the inserted baggage. As seen above a bag can only stop when it is either at `b2L` or at `b2R` states.

2.2 Control Parts

We describe below the control components of the model.

2.2.1 FeedController and the Suggested Control Strategy

The feed controller model is displayed in the Figure 4, FeedController below.

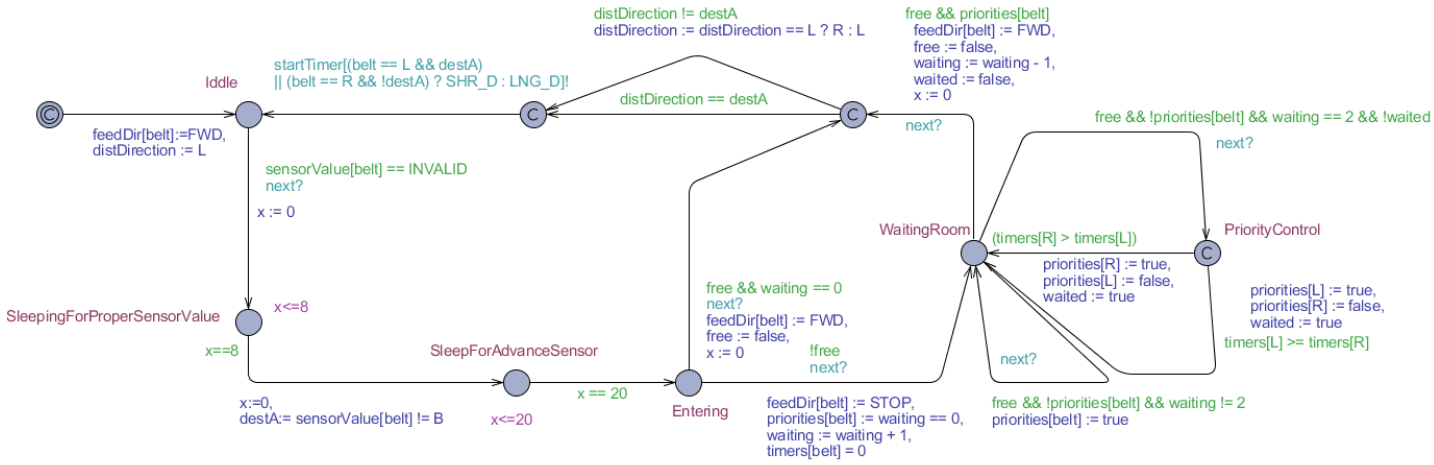


Figure 4, FeedController

The above controller is a mirror of the java code provided in the course. The controller is polling the `sensorValue` until it is not 'clear' which means a baggage is crossing the sensor. Afterwards it sleeps giving the time to the baggage to take a proper position in front of the sensor and when it wakes up it reads a valid `sensorValue`. Then it sleeps again until the bag advances the sensor. Here the bag enters a critical section (states `c, d, e, f`). That means that only one baggage is allowed to stand in this section. Other bags arriving will wait until the section is free (the feed belt will stop moving). The process follows a first in first out priority. After entering the section, if the baggage direction differs from the previous one it reverses the distribution belt and sets a timer that will timeout in a sufficient time for the baggage in the critical section to leave the system. On timeout, the timer triggers a leave event, which frees the critical section. The timeout delay time can either be the time needed for a baggage to cross a short or a long path[1].

The stopping problem is solved using the `free` variable which holds the status of the critical section. In order for our solution to be fair we implemented a first in first out queue for the waiting bags. The `waiting` variable holds the number of bags that are currently waiting for getting access to enter the critical section and together with `timers` and `priorities` variable they are responsible for maintaining the FIFO priority. When two bags are waiting in the

[1] By 'short path' we denote the b-c-e and by 'long path' the b-c-f-d-e sections path

WaitingRoom state, when the critical section gets free, one of them will enter the PriorityControl state. There, the bag that waited the most, gets priority to enter the section. The waited flag is only used in order to avoid entering an infinite loop (WaitingRoom -> PriorityControl -> WaitingRoom -> PriorityControl -> …). The free flag is initially set to true and as long as a baggage captures one of the c sections it is set to false. In this way stop may only occur on b states. The startTimer communication channel is used in order to properly set the timer. In order to simulate the channel as a function, a SHR_D or a LNG_D flag is passed as an argument and the delay variable is properly set to SHR or LNG respectively. Once the timer timeouts after delay time units it sets the free flag to true. Figure 5, Timer displays the timer's model.

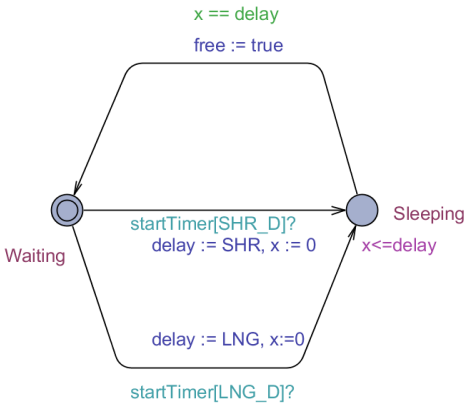


Figure 5, Timer

2.2.2 User

The user model is displayed in the Figure 6, User below.

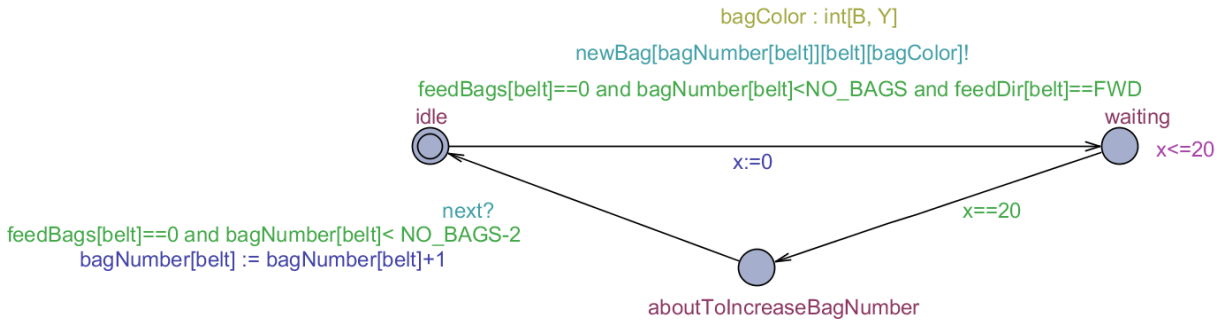


Figure 6, User

The user is responsible for providing bags to the system assigning a unique identifier (bagNumber). He waits until the feed belt does not have any baggage on it and then he introduces a new baggage. He stops when the maximum number of bags is reached (NO_OF_BAGS) and communicates with the bag model through the newBag channel.

3. Verification

In this section we verify some properties of the system using Uppaal verifier. For chapters 3.1 to 3.9, we assume that four bags are introduced to the system successively. The color is arbitrary and it is the user that defines it. After each property, a verifier command follows, along with a justification.

3.1 Bags are delivered at the right destination

```
A[] forall(i:t_id) (not ((Bag(i).color == B and Bag(i).YDelivered) or
(Bag(i).color == Y and Bag(i).BDelivered)))
```

We define a type `t_id` as `int[0,NO_OF_BAGS-1]`. The query above ensures that at any time all bags will not be delivered at the wrong destination. But this is not enough as it does not ensure that they will eventually be delivered. The last is shown in 3.6.

3.2 No collisions take place

```
A[] (collisionBags[R]<=1 and collisionBags[L]<=1)
```

In the bag model we use the `collisionBags` counter in order to verify the property above. We increase the counter every time a baggage enters the collision area (sections c and d) and we decrease it when it leaves the area. We can then ensure that no collisions take place as at any time the `collisionBags` counter should be less than or equal to one.

3.3 No bumping occurs

As mentioned in the problem description we can assume that bumping is possible to occur in sections c,d,e and f of the distribution belt when two bags are less than 1.4 seconds apart. In order to cover all possible scenarios we only focus on final destination e section and its predecessors d and c. If the property holds there, we can ensure that it holds in other parts of the distribution belt by induction. The improved query, compared with assignment 1 where we had verified the property for successive bags only, is shown below

```
A[] forall(i:t_id) forall(y:t_id)
      not(Bag(i).eR and Bag(y).dR and Bag(y).c>=9) and
      not(Bag(i).eR and Bag(y).cRR and Bag(y).c>=11) and
      not(Bag(i).eL and Bag(y).cLL and Bag(y).c>=11) and
      not(Bag(i).eL and Bag(y).dL and Bag(y).c>=9)
```

Since bumping may occur in successive bags only, we verify that at any time there is no possibility that a baggage ‘b1’ is at e section and another baggage ‘b2’ is at section c or d and ‘b2’ is about to leave its current position in less than or equal to 14 time units.

3.4 While a bag is at section c, no stops or reversions occur

```
A[] forall(i:t_id)
not(
      (
            Bag(i).cLL and (DistBelt.StoppedInstantly or feedDir[L] == STOP)
            and Bag(i).c<28
      )
      or
      (
            Bag(i).cLR and (DistBelt.StoppedInstantly or feedDir[L] == STOP)
            and Bag(i).c<28
      )
      or
      (
          Bag(i).cRR and (DistBelt.StoppedInstantly or feedDir[R] == STOP)
          and Bag(i).c<28
      )
      or
      (
          Bag(i).cRL and (DistBelt.StoppedInstantly or feedDir[R] == STOP)
          and Bag(i).c<28
      )
)
```

As displayed above we ensure that neither the distribution belt nor the feed belt is reversed or stopped while a bag is at `cLL, cLR, CRL, CRR.`

3.5 The dist belt is never stopped or reversed when it carries a bag

```
A[] forall(i:t_id)
not(
     (Bag(i).eL   and   ((DistBelt.StoppedInstantly   and   Bag(i).c<12)   or
DistBelt.Stopped)) or
     (Bag(i).dL and (DistBelt.StoppedInstantly or DistBelt.Stopped)) or
     (Bag(i).f and (DistBelt.StoppedInstantly or DistBelt.Stopped)) or
     (Bag(i).dR and (DistBelt.StoppedInstantly or DistBelt.Stopped)) or
     (Bag(i).eR   and   ((DistBelt.StoppedInstantly   and   Bag(i).c<12)   or
DistBelt.Stopped))
)
```

The distribution belt passes from a `StoppedInstantly` state while it is being reversed and from a `Stopped` state when it is stopped. Using the query above we ensure that the required property is satisfied.

3.6 Every bag is eventually delivered

```
(Bag(0).AtBeltL or Bag(0).AtBeltR) -->Bag(0).Delivered
(Bag(1).AtBeltL or Bag(1).AtBeltR) -->Bag(1).Delivered
(Bag(2).AtBeltL or Bag(2).AtBeltR) -->Bag(2).Delivered
(Bag(3).AtBeltL or Bag(3).AtBeltR) -->Bag(3).Delivered
```

The four queries above ensure that once a bag is introduced to the system it is eventually delivered.

3.7 Maximum number of bags that can be handled simultaneously

```
A[] feedBags[L]+feedBags[R]+distBags<=3
E<> feedBags[L]+feedBags[R]+distBags==3
```

The first query above verifies that at any time there are at most 3 bags in the system. The second one verifies that 3 bags are eventually occupying the system. Therefore the maximum number of bags that can be handled simultaneously by the system is 3, which makes sense since the system was design in such a way so that at most two bags can wait until a third one free the critical section.

3.8 Fastest possible handling of a bag

```
A[] forall(i:t_id) not((Bag(i).Delivered and Bag(i).time < 76))
E<> ((Bag(0).Delivered and Bag(0).time==76))
```

The first query above verifies that there is no possibility that a baggage is at state `Delivered` in less than 76 time units. The second one verifies that a bag will eventually be at state `Delivered` in 76 time units. Therefore the fastest possible handling of a bag is 76 time units, which is the time needed for a bag to cross the feed belt and the short path of the distribution belt without waiting.

3.9 Maximum handling time of any bag

```
A[] forall(i:t_id) not((Bag(i).Delivered and Bag(i).time > 134 + 106 + 66))
E<>  ((Bag(0).Delivered and Bag(0).time==134 + 106 + 66))
```

The first query above verifies that there is no possibility that a baggage is at state `Delivered` in more than 306 time units. The second one verifies that a bag will eventually be at state `Delivered` in 306 time units. Therefore the maximum handling time of any bag is 306 time units. This is logical since it corresponds to the sum of the time that a baggage can cross the feed belt and the long path of the distribution without waiting (134), the time that a baggage can wait for another waiting baggage with higher priority to cross the long path of the distribution belt (106) and the time that a third baggage needs to exit the distribution belt and release the critical section (66).

For chapters 3.10 to 3.12 we calculate the throughput of the system (bags per minute) for three different scenarios, considering the worst case scenario.

3.10 Sequence of yellow arriving at Check-In 1.

```
A[] forall(i:t_id) not((Bag(i).Delivered and Bag(i).time > 76+16))
E<> (Bag(1).Delivered and Bag(1).time==76+16)
E<> (Bag(2).Delivered and Bag(2).time==76+16)
```

The throughput of the system for the given scenario is 60/9.2 bags / min = 6.52 bags / min

3.11 Sequence of black arriving at Check-In 1.

```
A[] forall(i:t_id) not((Bag(i).Delivered and Bag(i).time > 134+66))
E<>  (Bag(2).Delivered and Bag(2).time==134+66)
E<>  (Bag(3).Delivered and Bag(3).time==134+66)
```

The throughput of the system for the given scenario is 60/20 bags / min = 3 bags / min

3.12 Alternating seq of bags for A and B arriving at Check-In 1.

```
A[] not (Bag(3).Delivered and Bag(3).time>134 + 12)
E<> (Bag(3).Delivered and Bag(3).time==134 + 12)
A[] not (Bag(2).Delivered and Bag(2).time>76 + 70)
E<> (Bag(2).Delivered and Bag(2).time==76+70)
```

For this specific scenario we test the third and fourth baggage introduced to the system and we calculate the average time that a bag needs to cross the facility. The throughput of the system for the given scenario is 60/14.6 bags / min = 4.11 bags / min

4. Java Implementation

After having modelled and verified the system, we started the java implementation. `MultiSort.java` is responsible for initializing our monitor, shared functions and variables. It starts the motors (the distribution motor will be started initially with a default moving direction) and two threads for the feed belt controllers.

`Shared.java` is our main monitor and its main tasks are to handle the synchronization in the waiting room. We document below the main variables used.

- `waiting` - reflects the number of bags waiting for the distribution belt to be free
- `free` - shows the status of the distribution belt,
- `delay` - time needed until a bag exits the critical section;
- `priorities` - the system is built in a way that, while a bag is in the distribution belt, 2 bags can be in the waiting section, 1 for each feed belt. Depending on who entered first, a bag receives priority over the other. If no bag is in the distribution belt, the first one to arrive gets a free pass.
- `timestamps` - used to compare the different times two bags landed on the `WaitingRoom`. We have used this approach on our Uppaal model in which we compare the two timers, with the difference that we had to avoid an infinite loop by introducing the variable `waited`.

For entering the critical section, we introduce the `shared.enter()` synchronized function which blocks only when the section is not free or the current waiting thread does not have priority. The section is released using a timer thread (`ThreadTimer.java`) and a timer monitor (`TimerMonitor.java`). Just before reentering the while loop, the `FeedBeltController.java` sets a timer with the proper delay, as modeled in FeedController and the Suggested Control Strategy. The timer, sleeps for this delay and when it wakes up it calls the synchronized `shared.leave()` notifying all waiting threads in the waiting room. When a thread wakes up it defines its priority for entering the section regarding how long it waited in the waiting room compared with other possible waiting thread. The code is available in Appendices.

5. Simulation and Physical Testing

The simulation of the system running the java implementation was working properly. No warnings or errors were outputted and all bags were delivered in the correct destination. But when we did the physical testing we realized that there was a major problem: a baggage could stay at the waiting room forever without updating its priority. Placing LCD outputs in different parts of our code helped us realize that the problem was related to the priority mechanism. Our logic for introducing bags in the system did not allow us to catch this miscalculation in UPPAAL. This step allowed us to make further improvements both in the modelling and the implementation. The bug was fixed by introducing `System.currentTimeMillis()` in order to keep track of the waiting time of each thread.

6. Conclusion

This part of the assignment proved to be a reality check for our project. All the timings were tweaked and perfected in order to fulfill strict demands in performance. We successfully managed to create a model for the baggage sorting facility, meeting the requirements for the separation of the physical and control parts of the system. The physical testing not only gave us a unique chance in seeing how the system actually behaves but also, together with the verification described earlier, allowed us to argue about the trustworthiness of the final system.

Appendices

FeedBeltController.java

```
package team05assignment02;
import josx.platform.rcx.Motor;
import josx.platform.rcx.Poll;
import josx.platform.rcx.Sensor;

class FeedBeltController extends Thread {
    static final int BLOCKED = 65;
    static final int YELLOW  = 60;
    static final int BLACK   = 45;
    static final int SHORTPATH   = 6000;
    static final int LONGPATH   = 11000;
    private int belt;
    private Sensor sensor;
    private Motor motor;
    private short sensorMask;
    private boolean destA;   // Required destination is A
    private TimerMonitor timerMonitor;
    private Shared shared;

    public FeedBeltController(TimerMonitor timerMonitor, Shared shared) {
        this.timerMonitor = timerMonitor;
        this.shared = shared;
    }

    public void setBelt(int belt) {
        this.belt = belt;
    }

    public int getBelt() {
        return belt;
    }

    public void run() {
        try {
            Poll e = new Poll();
            if (this.belt == MultiSort.LEFT) {
                sensor = Sensor.S1;
                sensorMask = Poll.SENSOR1_MASK;
                motor = Motor.A;
            } else {
                sensor = Sensor.S2;
                sensorMask = Poll.SENSOR2_MASK;
                motor = Motor.B;
            }
            sensor.activate();
            motor.forward();
            int delay;
            while (true) {
                while(sensor.readValue() > BLOCKED) {e.poll(sensorMask,0);} // Wait for bag
                Thread.sleep(800); // wait for proper sensor value
                destA = sensor.readValue() > BLACK;   // determine destination
                Thread.sleep(2000); // advance sensor
                shared.enter(motor); // enter mutex area
                if (shared.getDistDirA() != destA) {  // Decide whether to reverse distribution
                    Motor.C.reverseDirection();
                    shared.setDistDirA(destA);
                }
                delay = (belt == MultiSort.LEFT && destA) ||
                        (belt == MultiSort.RIGHT && !destA) ? SHORTPATH : LONGPATH;
                timerMonitor.startTimer(delay);
            }
        } catch (Exception e) { }
    }
}
```

MultiSort.java

```
package team05assignment02;
import java.lang.System;
import josx.platform.rcx.*;

public class MultiSort {
    static final int BELT_SPEED = 3;
    static final int LEFT = 1;
    static final int RIGHT = 2;

    public static void main (String[] arg) {

        TimerMonitor timerMonitor = new TimerMonitor();
        Shared shared = new Shared();
        ThreadTimer threadTimer = new ThreadTimer(timerMonitor, shared);
        threadTimer.start();
        Motor.A.setPower(BELT_SPEED);  Motor.C.forward();
        Motor.B.setPower(BELT_SPEED);
        Motor.C.setPower(BELT_SPEED);
        shared.setDistDirA(true);
        Thread fL = new FeedBeltController(timerMonitor, shared);
        ((FeedBeltController) fL).setBelt(LEFT);
        fL.start();
        Thread fR = new FeedBeltController(timerMonitor, shared);
        ((FeedBeltController) fR).setBelt(RIGHT);
        fR.start();
        try{ Button.RUN.waitForPressAndRelease();} catch (Exception e) {}
        System.exit(0);
    }
}
```

ThreadTimer.java

```
package team05assignment02;

import josx.platform.rcx.*;

public class ThreadTimer extends Thread {

    private TimerMonitor timerMonitor;
    private Shared shared;

    public ThreadTimer(TimerMonitor timerMonitor, Shared shared) {
        this.timerMonitor = timerMonitor;
        this.shared = shared;
    }

    public void run() {
        while (true) {
            timerMonitor.waitStartTimer();
            try {
                Thread.sleep(timerMonitor.getDelay());
            } catch (InterruptedException ie) {
                LCD.showNumber(98); LCD.refresh();
            }
            System.out.println("timedout!");
            shared.leave();
        }
    }
}
```

Shared.java

```
package team05assignment02;
import josx.platform.rcx.*;

public class Shared {
    private int waiting = 0;
    private boolean free = true;
    public int delay;
    private boolean distDirA;
    private boolean priorities[] = {true, true};
    private int timestamps[] = {0, 0};
    public boolean read(Sensor s) {
        synchronized (this) {
            return s.readValue() > FeedBeltController.BLACK;
        }
    }

    public void enter(Motor motor) {
        synchronized (this) {
            int index = (motor.getId() == 'A') ? 0 : 1;
            if (!free) {
                priorities[index] = (waiting == 0);
                motor.stop();
            }
            waiting++;
            while (!(free && priorities[index])) {      //while at least a bag is using Shared
                try {
                    timestamps[index] = (int) System.currentTimeMillis();
                    wait();
                    if (waiting == 2) {
                        priorities[0] = timestamps[0] < timestamps[1];
                        priorities[1] = timestamps[1] <= timestamps[0];
                    } else {
                        priorities[index] = true;
                    }
                } catch (InterruptedException e) {}
            }
            waiting--;
            free = false;                   // good morning! you are using shared
            motor.forward();
        }
    }

    public void leave() {
        synchronized(this) {
            System.out.println("leaving!");
            free = true;
            notifyAll();                    // notify other waiting bags
        }
    }

    public void setDistDirA(boolean destA) {
        synchronized(this) {
            distDirA = destA;
        }
    }

    public boolean getDistDirA() {
        synchronized(this) {
            return distDirA;
        }
    }
}
```

TimerMonitor.java

```
package team05assignment02;
import josx.platform.rcx.*;

public class TimerMonitor {
    private boolean startTimer = false;
    private int delay;

    public void waitStartTimer() {
        synchronized (this) {
            while (!startTimer) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    LCD.showNumber(99); LCD.refresh();
                }
            }
            startTimer = false;
        }
    }

    public void startTimer(int delay) {
        synchronized (this) {
            this.delay = delay;
            startTimer = true;
            notifyAll();
        }
    }

    public int getDelay()
    {
        return this.delay;
    }
}
```