Mandatory Assignment 1

Baggage Sorting Facility

1 Overview

This assignment deals with a baggage sorting facility such as the ones found in airports. It must transport bags from different check-in counters to different destinations depending on which flight the bags should go. In this assignment we work with (a LEGO model of) a simple such baggage sorting facility. An overview of this system is shown in figure 1.

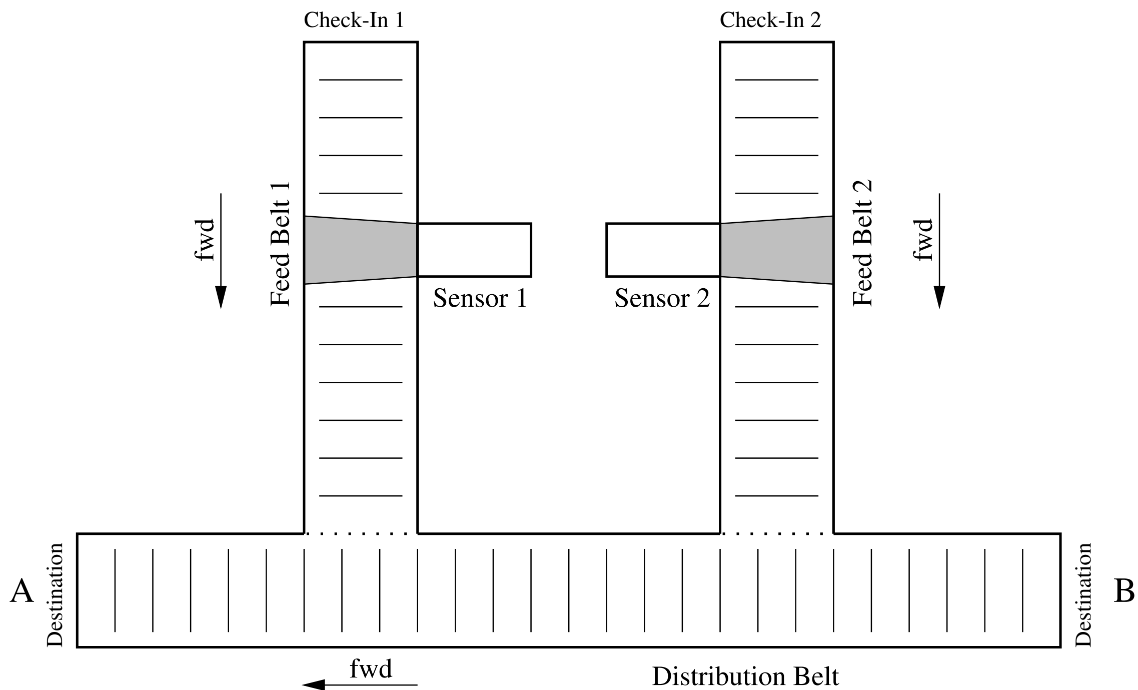

Figure 1: Overview of the baggage sorting facility

Bags are loaded into the baggage sorting facility at the Check-Ins either on Feed Belt 1 or on Feed Belt 2. The baggage sorting facility must transport bags to either Destination A or Destination B using the Distribution Belt.

The feed belts and the distribution belt may move bags either forward or backwards or they may be stopped. The forward direction is indicated on figure 1. On both feed belts a light sensor is mounted. It can be used to detect when a bag passes the sensor. Furthermore, it can detect the colour of the bag. In this system we assume that all bags are either yellow or black.

The overall task of the baggage sorting facility is simple: All yellow bags must go to Destination A and all black bags must go to Destination B.

1.1 Sensors

On each feed belt a sensor is mounted. This sensor is used to detect that bags pass the sensor. The sensor reads the light intensity, which are represented by integer values, which are roughly in the interval from 30 to 80.

Whenever the sensor is clear, i.e. when no bag is placed in front of the sensor, the sensor value is at its largest (approximately 75). When a yellow bag is placed in front of the sensor the sensor value drops to a lower value given in table 1. When a black bag is placed in front of the sensor the sensor value drops to an even lower value.

Status	Sensor value
Sensor is clear	>70
Yellow brick	50-60
Black brick	30-45

Table 1: Sensor values

There is a chance that the sensor cannot correctly distinguish between a black and a yellow bag. The problem arises when a bag does not completely block the sensor, as illustrated on figure 2. The sensor value may then drop below the yellow threshold, but some light may still reach the sensor. At this moment, the sensor value may still be above the black threshold even when the bag is black.

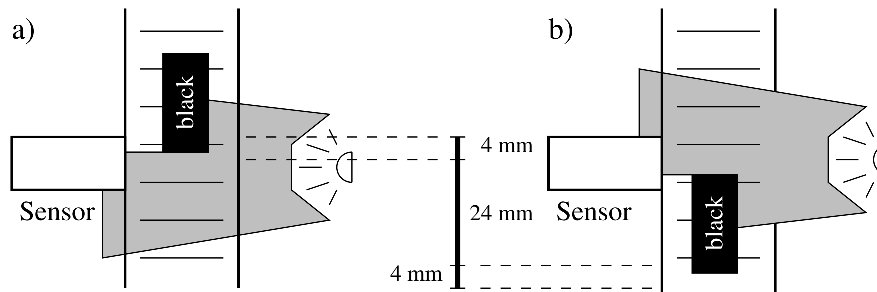

Figure 2: The sensor cannot distinguish between black and yellow bags when a bag is not fully in front of the sensor.

There is a small section of 4 mm of the feed belt in front of the sensor on which the sensor can detect that a bag is present but cannot correctly detect its colour. This phenomenon will occur when the bag moves in front of the sensor as illustrated on figure 2a. From that point on the bag will be able to move down the feed belt for 24 mm before the phenomenon occurs again when the bag moves out of the sensor's range as illustrated in figure 2b.

1.2 Timings of the System

There are only the two sensors in the system which can be used to detect where bags are located. When bags are anywhere else in the system we must calculate their positions based on timings of the system. Figure 3 shows different sections of the system for which timings are given. Note that Feed Belt 1 and Feed Belt 2 are symmetric.

There is no timing for the section from the point where the bags are placed on a feed belt and to the point where it reaches the sensor. For a bag to move through this section of the feed belt you must simply keep the belt running and use the sensor to detect when the bag passes the sensor. Section a) starts at the earliest point where a bag can be detected.

The timings are given for the system under the assumption that the belts are running, so the bags will actually move. The speed of the belts be must set to the predefined constants for the timing values to apply. The details of these constant will be given later, when it is explained how to write programs that work with the LEGO system.

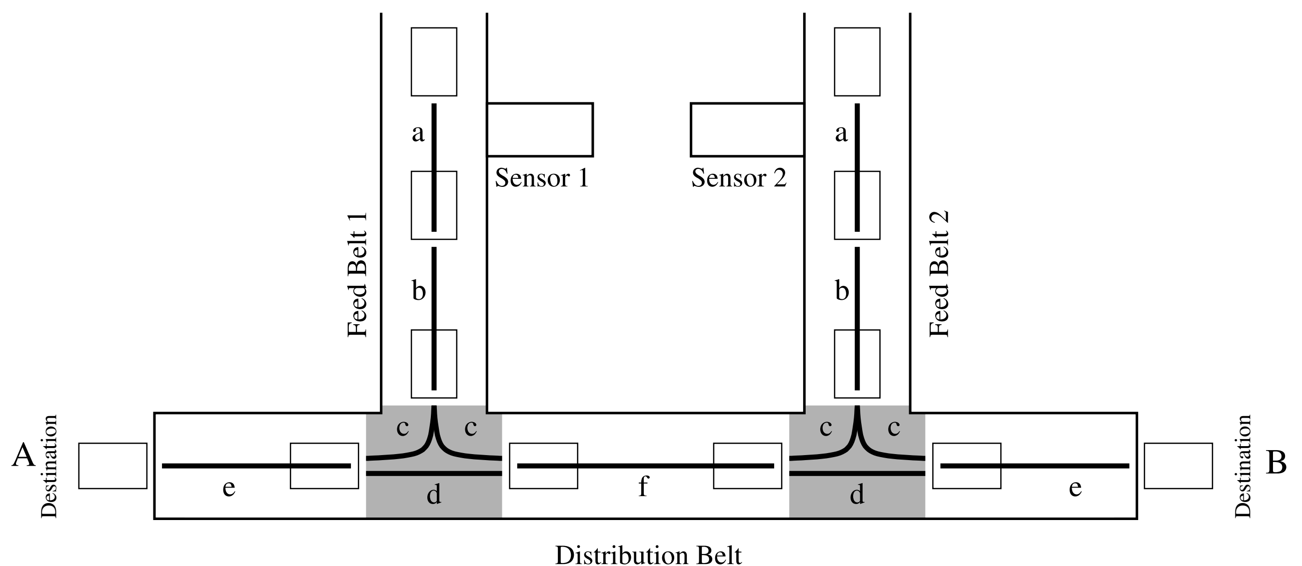

Figure 3: The system is divided into section for which timings are given. The location of the bag at the end of a section is indicated by the rectangles on the figure.

Section a This section starts at the point where it is possible to detect that a bag is present. The section ends when the bag is clear of the sensor.

Section b The section begins where the bag is clear of the sensor and ends just before the bag starts turning on to the distribution belt.

Section c In this section the bag turns on to the distribution belt. The section ends when the bag is clear of the feed belt. Note that both the feed belt and the distribution belt *must be running* for the bag to be turn properly.

Section d When the bag is already on the distribution belt it may pass one of the feed belts. The timing of this section includes all the time when the bag blocks the feed belt.

Section e The section goes from the point where the bag is clear of the feed belt till it has reached Destination A or Destination B (where it drops down from the distribution belt).

Section f The section between the two feed belts. At the end points of this section the bag is still clear of the feed belts.

For section a) and section b) we give the timings based on the length of the section and the speed of feed belt. The feed belt may be assumed to move at a constant speed of 20 mm/s. Section a) is 32 mm long. It is the entire section, where the sensor can detect that a bag is moving in front of the sensor i.e. the section shown in figure 2. Remember that the sensor value in the first and the last 4 mm of the section cannot be used to decide the colour of the bag. Section b) is 48 mm long.

The timing for the remaining sections are given as explicit timing values. Table 2 gives the time taken for a bag to move from one end of a section to the other in terms of upper and lower bounds.

Section	Lower bound	Upper bound
c	2.4	2.7
d	2.2	2.3
e	1.0	1.2
f	2.5	3.1

Table 2: Timings in the system. Timing values are given in seconds.

The variation of section c) is due to the turning process. The variation of section f) is primarily due to the fact that the distribution belt is composed of two sections and passing between them takes varying time.

Collisions and bumping

A *collision* may occur if a bag is fed to the distribution belt while another bag is passing. To be on the safe side, bags must not be be simultaneously present in sections c) and d) belonging to the same feed belt. Several bags on a c)-section or a d)-section are allowed, subject to the bumping constraint described in the following.

Also to be avoided is the phenomenon of a bag *bumping* into its predecessor. This may occur in sections c), d), e), and f) where a fast bag may arrive at the end of the section before the previous has completely left it. With the actual distribution belt speed, bags should be at least 1.4 seconds apart on the distribution belt — especially when arriving at the destination. To avoid bumping, a minimum separation of bags should be observed at the Check-Ins.

2 The LEGO RCX Box

The Lego RCX-box launched in 1998 is large LEGO brick with a (for that period) standard 8-bit microprocessor (Hitachi H8/3292) equipped with:

- 32 K RAM
- 3 input ports: 1, 2, 3
- 3 output ports: A, B, C
- 1 bidirectional serial infra-red port
- 1 LCD-display
- 1 "loud-speaker"
- Some control buttons
- A simple "BIOS" in ROM (for loading firmware).

A number of sensors can be connected to the input ports such as *buttons*, *rotation sensors* and *light sensors*. Their value is read by an A/D converter and mapped into memory locations.

The output ports may be connected to actuators. The only actuators available from LEGO are *motors* and *lamps*. Motors are controlled by assignment to special memory locations. The speed of the motors can be controlled by pulse-width modulation.

The RCX-box may be programmed in *RCX-code* similar to Java byte-code. The interpreter for the virtual RCX machine (the so-called *firmware*) can be downloaded by the "BIOS".

A number of programming environments have been developed for the RCX virtual machine. LEGO has provided the graphical environments RCX and Robolab. Others have developed textual languages for generating RCX-code, such as *NQC* (Not Quite C) which is available for many platforms.

The RCX-code has rather limited means of expression. Therefore people around the world have developed other environments for programming the RCX-box. One approach is to program in C compiling to native Hitachi code running on a multi-tasking real-time kernel called *LegOS/BrickOS*.

Another approach is to replace the RCX-code interpreter with a *Java Virtual Machine* executing standard Java byte code such that the RCX-box can be programmed in Java. This approach is further described below.

The RCX-brick was superseded by the ARM7-based NXT-model launched by LEGO in 2006 followed by the ARM9-based EV3 model launched in 2013 and the ARM-Cortex M4 based Spike Prime Hub in 2019.

3 Introduction to LeJOS

LeJOS is an implementation of a Java Virtual Machine on the RCX-processor enabling Java programming of the LEGO RCX brick. Due to the limited amount of memory, not all language features are available, but for many small embedded systems, this will pose no major problem. Amongst the features provided are:

- All language constructs except switch-statements.
- Java threads with synchronization and preemptive scheduling.
- Floating point types.
- A package `josx.platform.rcx` that provides high-level access to the RCX devices.

On the other hand, the virtual machine lacks garbage collection and almost all standard Java packages.

LeJOS was started by Jose Solorzano in 1999, and has been developed as an open source project located at:

`http://lejos.org/`

The home page is the place to look for information on LeJOS. There is a detailed documentation of the API and an online tutorial. Development for RCX seems to have ceased in 2006 though.

In the following we give a small introduction to some of the key facilities of LeJOS, which should be enough to get you started. In section 4.2 we show a small example of a program which uses some of the features described.

The facilities for real-time programming in LeJOS are partly to be found in the Java language itself and partly in the packages provided by LeJOS.

The reader is assumed to be familiar with Java in general and multi-threaded Java programming in particular.

Information on how to install LeJOS and compile Java programs for LeJOS can be found via the course home page.

3.1 Threads and synchronization

Threads are created as instances of classes that extend `Thread` as usual. Threads may share data via monitors in the usual way. Also conditional synchronization may be accomplished by the usual `wait`/`notify` mechanism. [Avoid `static` monitors which do not work.]

3.2 Motors

Motors are controlled through the class `Motor` found in the `josx.platform.rcx` package. The `Motor` class provides a number of instance operations of which the most useful ones

are:

`setPower(int n)`	Set the speed of the motor to a value n between 0 and 7. In the Sorting Facility, the speed of all motors should be set to 5 for the timings to be correct.
`forward()`	Let the motor run forward at the speed set.
`backward()`	Let the motor run backward at the speed set.
`stop()`	Stop the motor abruptly (brake).
`flt()`	Float. Stop the motor, but let it spin.
`reverseDirection()`	Reverse the direction of the motor (if running).

The current state of the motor can be interrogated with the boolean functions `isForward()`, `isBackward()`, `isStopped()`, and `isFloating()`.

The `Motor` class cannot be instantiated, but provides three static instances (`A`, `B`, and `C`) of itself representing the outputs of the RCX-box. E.g. to start the A-motor, you should issue:

```
Motor.A.forward();
```

3.3 Light Sensor

The light sensors are accessed through the `Sensor` class of the `josx.platform.rcx` package. There are many different types of sensors. However, by default a sensor is assumed to be a *light sensor* giving *percentage* values. This complies with the sensors of the current Baggage Sorting Facility. Among the many `Sensor` object operations, the following suffice for this assignment:

`activate()`	Activates the sensor. For a light sensor, this lits the red LED built into the sensor making it less sensitive to the lighting conditions of the surroundings.
`passivate()`	Deactivates the sensor turning off the LED.
`int readValue()`	Reads the current sensor value. Only valid, if the sensor is activated.

As for the motors, the `Sensor` class provides three static instances (`S1`, `S2`, and `S3`) of itself corresponding the the three inputs of the RCX-box.

The sensor value may be polled at regular intervals by the program or a specific value may be waited for using the `Poll` class described in Section 3.5.

3.4 Buttons

The buttons of the RCX-box are accessible through a `Button` class. Operations are provided to test whether a button is pressed and to wait for a press-and-release. For details, see the LeJOS API-documentation.

3.5 Awaiting Events

Very often, a thread should wait for a certain external event to happen before it can proceed. This should not be accomplished by busy waiting since it may use up all CPU-power. Rather LeJOS provides a simple mechanism in which the virtual machine polls the sensors and buttons at regular intervals (currently every 3 ms) and wakes up any thread that waits for certain input values to change. The mechanism is provided through the class `Poll`. An instance of `Poll` has the operation:

`int poll(int` m`, int` t`)` Waits for change of the inputs given by the bitmask m. t is a timeout value in milliseconds with $t = 0$ disabling timeout. The return value indicates which inputs have changed.

The class provides constants defining the various bit masks, e.g. `SENSOR2_MASK`.

Reading sensors only when they change may drastically reduce processor usage. For example, waiting for Sensor 2 to drop below the value 72, may be implemented by:

```
Poll e = new Poll();
while (Sensor.S2.readValue() >= 72) { e.poll(Poll.SENSOR2_MASK,0);}
```

As an alternative to the polling mechanism, events may be awaited by providing *event listeners* as known from full Java. For the current assignment, however, this mechanism is not recommended. Details may be found in the LeJOS API-documentation.

3.6 LCD Display

The LCD display of the RCX-box can be used to write numbers and simple strings. To access the display, an `LCD` class is given. It may come in handy for debugging purposes. For details see the LeJOS API-documentation.

3.7 Timing

The current time in terms of milliseconds elapsed since start of the virtual machine is read by the standard operation `long System.currentTimeMillis()`. The resolution of this clock is currently 3 ms. Note that LeJOS does not support operations on `long`. Hence these have to be casted to `int` to be operated upon.

A thread may delay itself for n milliseconds by calling `Thread.sleep(`n`)`. Again the granularity of the timing is 3ms.

4 The LEGO System

4.1 Controlling the LEGO System

You will be asked to write a program which controls a version of the baggage sorting facility built in LEGO. This program must be written in Java and work under LeJOS.

The Feed Belts and the Distribution Belt are controlled by motors. All the devices in the LEGO system are connected to a single LEGO RCX brick in the following manner:

Device	**RCX port**
Feed Belt 1	Motor A
Feed Belt 2	Motor B
Distribution Belt	Motor C
Sensor 1	Sensor 1
Sensor 2	Sensor 2

The sensors must be activated (which turns on their built-in light) before being read. This is e.g. done for Sensor 1 by the call `Sensor.S1.activate()`. The sensor value can then be read by `Sensor.S1.readValue()`

For the timings of the system to apply, the feed belts and the distribution belt must be set to run at a constant speed of `5` and stopped by `stop()`.

4.2 An Example Program

In this section we show a program, `SingleSort.java`, which is a small example of how to write Java programs for the LEGO RCX brick under LeJOS. The program works for the given LEGO system but can only sort bags for one check-in at a time.

The main program sets the motor speed and starts up a thread for each feed belt passing its motor and sensor objects as parameters. Each Feeder thread starts is feed belt and awaits that a bag passes its sensor. When this happens it waits for 0.8 seconds and then reads the colour of the bag. It then advances the bag further to the middle of section b) where it decides whether to stop or not. If the current direction of the distribution belt fits the destination of the bag, it just lets the bag continue. Otherwise the thread stops the feed belt, waits sufficently long time for the last bag to have left the distribution belt, changes the direction of that and starts the feed belt again. Concurrently, the main thread waits for a press on the `RUN`-button to stop the system.

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

class Feeder extends Thread {

  static final int BLOCKED = 70, YELLOW  = 60, BLACK   = 48;

  Motor  myMotor;
  Sensor mySensor;

  public Feeder(Motor m, Sensor s) { myMotor = m;  mySensor = s; }
```

```
  public void run() {
    try {
      final boolean closeToA = (myMotor == Motor.A);
      final int myMask = closeToA ? Poll.SENSOR1_MASK : Poll.SENSOR2_MASK;

      Poll e = new Poll();
      int done = (int) System.currentTimeMillis(); // When last bag is through

      myMotor.forward();

      while (true) {
        // Await arrival of a bag
        mySensor.activate();
        while(mySensor.readValue() > BLOCKED) { e.poll(myMask,0); }

        Thread.sleep(800);            // Wait for colour to be valid

        boolean destA = (mySensor.readValue() > BLACK);   // Determine destination
        mySensor.passivate();

        Thread.sleep(2000);           // Advance beyond sensor
        if (Motor.C.isForward() != destA) {  // Must stop
          myMotor.stop();
          int now = (int) System.currentTimeMillis();
          if (now < done) Thread.sleep(done-now);
          Motor.C.reverseDirection();
          myMotor.forward();
        }

        done = ((int) System.currentTimeMillis()) + 6000;
        if (Motor.C.isForward() != closeToA) done = done + 6000;  // Long path

        Thread.sleep(1200);                   // Follow to end of feed belt
       }
    } catch (Exception e) { }
  }
}

public class SingleSort {

  static final int BELT_SPEED = 5;            // Do not change

  public static void main (String[] arg) {
      Motor.A.setPower(BELT_SPEED);
      Motor.B.setPower(BELT_SPEED);
      Motor.C.setPower(BELT_SPEED);  Motor.C.forward();

      Thread f1 = new Feeder(Motor.A, Sensor.S1);  f1.start();
      Thread f2 = new Feeder(Motor.B, Sensor.S2);  f2.start();

      try{ Button.RUN.waitForPressAndRelease();} catch (Exception e) {}
      System.exit(0);
  }
}
```

This program only works if bags arrive at one check-in at a time and with a proper separation. Here the program uses the *activation light* to inform the operator that a new bag may be put on the feed belt (provided the sensor is clear). The program waits until the previous bag has reached the end of the feed belt before allowing a new bag.

5 The Assignment

You are expected to model the baggage sorting system using the simple control strategy provided by the `SingleSort` program and *verify* that bags on one feed belt are correctly and efficiently sorted.

[In the follow-up assignment option you will be asked to design and implement your own control strategy catering for both feed belts and verify that it sorts correctly.]

Specifically you must accomplish the following tasks:

1. Model the physical system (i.e. the LEGO construction) in UPPAAL.
2. Model the given simple control program `SingleSort.java` (see 4.2) in UPPAAL.
3. Formalize and verify the following properties:
   - Bags are delivered at the right destination.
   - Every bag is eventually delivered.
   - No collisions take place (when the system is properly used).
   - No bumping occurs. [Although not possible in this assignment given the separation of bags when a single feed belt is used, the property should be formulated and verified.]
   - While a bag is turning (in section c), neither the feed belt nor the distribution belt is stopped or reversed. [This is a precondition for the timings to be valid.]
   - The distribution belt is never stopped or reversed when it carries a bag. [Another timing precondition.]
   - Any further erroneous behaviour that you may have identified does not occur.

   For the given program, these properties should hold when only one of the check-ins is used and bags are put on the feed belt only when i) the feedbelt is moving forward, ii) section a) is clear of other bags, and iii) the sensor is active.
4. Demonstrate that some of these properties may fail to hold (or erroneous conditions occur), when bags are allowed to arrive at both check-ins.
5. Show that he system mail fail to sort correctly if the colour of the bag is read within one of the uncertainty zones shown in Figure 2. [In the given program, you may change the two first sleep periods to 1500 and 1300 respectively preserving the stop position at the middle of section b).]
6. Determine by analysis the minimum and maximum handling time (from arrival to delivery) for both yellow and black bags arriving at Check-in 1 and use the UPPAAL model to validate these numbers by showing appropriate properties.
7. [Optional, see below]
   The timings used in the `SingleSort.java` program are not tight. Determine by analysis the best worst-case bag handling time that can be achieved by adjusting

the timings of the program (i.e. the sleep/timeout periods, not the program structure) and validate this in UPPAAL. Of course, the system should still sort bags correctly.

8. [Optional, see below]
   Show that the system (with the given non-optimal timings) will work correctly when the check-ins are used one at a time with at proper inactivity period when switching from one check-in to another. Determine by analysis the minimum inactivity duration needed and validate this using UPPAAL.

9. [Optional, see below]
   In the given program, the bags are stopped in the middle of section b) and are separated by at least 4.0 seconds. Analyze whether another stop position might allow bags to be entered with less separation time and demonstrate your finding using UPPAAL verifications.

10. [Optional, see below]
    Make a *green version* of the system (with the given timings) such that the distribution belt stops if no bags are checked in for some period. State and verify this property on a UPPAAL model of the modified system. Also, a corresponding Java implementation must be made and handed in.

11. [Optional, see below]
    Enable simultaneous check-in at both feed belts. The solution should be as simple as possible and might be very inefficient. The solution must be modelled in UPPAAL with a corresponding implementation in Java (both to be handed in). Verify that the solution works in UPPAAL and test the implementation using the simulator.

Each group member must choose exactly one of the optional task 7.–11. and carry this one out on his/her own. The choices of a group must be distinct, i.e. no two members should do the same optional task.

The work must be described in a report as detailed in Section 8.

6 Modelling Guidelines

Physical Model

You should start by modelling the **physical system** (i.e. the Lego version of the sorting facility). The purpose of the physical model is to enable verification of a variety of system controllers. Therefore, the model should reflect both the intended behaviour of the system *as well as relevant undesired behaviour*. In other words, the model should allow for things to go wrong such as stopping the Distribution Belt prematurely etc.

You may consider the following *monkey test*: Assume that the RCX brick is replaced by a monkey controlling the outputs (presumably in an arbitrary way). What could happen

in the physical system? Does your model cater for that?

Erroneous Behaviour

Beware that you should not (and cannot) model any conceivable behaviour in all details. First of all, you may decide that certain control possibilities should not be modelled at all. E.g. you may assume that the speed settings of the motors is always fixed (corresponding to a setting of 5).

For other control aspects you may need to limit the control behaviour for which the model will work correctly. For instance, you may constrain the possible control solutions by making the (very reasonable) assumption that it makes no sense to move the feed belts backwards. If they ever are, this should be considered *erroneous (control) behaviour* and it should be verified that the actual controller never attempts to do so.

In task 3. above, two instances of erroneous behaviour have already been identified in order to rely on the timings and avoid uncertainty in the model:

- While a bag is turning (in one of the c-sections) neither the relevant feed belt nor the distribution belt must be stopped.
- Once a bag is on the distribution belt, it should not be stopped nor reversed.

In the report you must discuss and identify which further control behaviour you consider erroneous and explain how you have checked that such behaviour does not occur.

Environment

The physical model should also comprise an *environment/user* component that reflects the assumptions you make on the *use* of the system — e.g. when bags are loaded onto the feed belts. It may be assumed that the user can observe the activation lights of the sensors and the occupation of the sensors in order to determine whether a bag may be entered. The user component should be able to generate different usage scenarios.

The Stop Problem

A particular modelling issue is the (feed belt) *stop problem*. When a bag is stopped within some section the feed belt, its precise position cannot be recorded and when resumed, it should therefore be treated as being somewhere within the section. As discussed at the lectures, possible modelling approaches are:

- to allow stopping of bags only at specific places at the feed belt
- to use use constant time stop periods (see examples at the project page)
- to use time discretization (with risk of state explosion)

For some of these approaches to work, you may have to make usage assumption about the system, e.g. by allowing at most one bag at a feed belt at any time ensuring this through the environment/user component.

[Since the `SingleSort` program uses a fixed stop position, the first option is the obvious choice for this assignment. If needed by your own control program, you may use another approach in the optional follow-up assignment.]

Sensors

The two sensors form part of the interface to the control model. Basically it should be possible for the control model to read a sensor at any time getting a value that reflects the color of any bag that might be in front of the sensor.

As a first approximation, you may model the sensor as a shared variable set by the bag while passing.

However, also the *activation state* of the sensor should be properly modelled in order to enable the user to react to the *activation light* of the sensor. Ideally, it should also be modelled that sensor readings may not be valid when the sensor is inactive.

Motors

Also the three motors form part of the interface to the control model. The current state of each motor should be accessible for the bags in order to determine their movements.

The motors should not constrain the controller. In particular, it must be possible to apply any motor command at any time.

Control Model

Once the physical model is satisfactory, you may construct a **control model** which should control the physical systems in order to sort the bags. The control model should represent the `SingleSort.java` program that will eventually be running on the RCX brick and must thus be clearly separated from the physical model. For example, the control model may only refer to information obtained from the sensor readings.

The interface between the physical model and the control model should be *asynchronous*, i.e. it must be possible to change the motor settings at any time and it must be possible to read the sensor values at any time.

If you use channel synchronization at the interface, you must make sure that the above asynchrony is ensured.

The control model must clearly reflect the structure of the control program: Each concurrent thread should be represented by a UPPAAL process and the control structure of each thread must be readily recognized in the automaton.

The `Poll` mechanism in LejOS for awaiting events/conditions should not be mimicked in details, but rather be represented in a more abstract way.

The timeout mechanism used the program (setting the `done` mark) is not feasible in UPPAAL. Instead a similar timing effect must be achieved using UPPAAL clocks.

> You may assume that the control processes react *immediately* to changes of the input values and that *sleeping is precise*. In reality there will be some *jitter* but that will be order of a few milliseconds if the processor is not heavily loaded (hence no busy waiting!) and may be neglected.

7 Verification Guidelines

In order to make the verification feasible, you should:

- Be sure to avoid variables which may grow without bounds (check!).
- Limit the number of bags considered simultaneously.
- Avoid unnecessary non-determinism, such as allowing bags to enter in arbitrary order. Alternatively, you may use *scalar types* for bags as described on the *project page*.
- Use properly bounded types for all integer variables.
- Avoid using fine-grained timing. Instead, use conservative time constants based on a suitable time unit.
- Use *committed states* for sequences of local actions in the control model.
- You may have to restrict the verification to a finite number of bags (i.e. not recycling them), and perhaps even imposing particular timing constraints on their check-ins.

If you cannot verify in short finite time (say a few minutes), it is very likely that the state space has become too large for UPPAAL to handle in the available RAM. You can follow the memory usage in the tool. Once the machine starts to swap, you might as well give up.

Try to use some of the verification options or to reduce the state space as indicated above. If nothing helps, discuss this in the report.

8 Report Requirements

Your work on this assignment must be documented by a report. The report should comprise the following:

- A brief introduction including your work distribution (see below).
- An overview (accompanied by an illustration) of how the various model components are divided into a physical and a control part and how the components interact. Especially the interface between the physical and the control parts must be specified.
- A description of the physical modelling including:
  - The overall modelling principle.
  - Your deliberations on difficult modelling issues, in particular the *stop problem*.
  - Identification of control behaviour considered erroneous.
  - Your assumptions on the environment/user behaviour.
  - Argumentation that the physical model does not constrain the control model.
  - Argumentation that the control model does not unrealistically constrain the physical model.
- A description of your modelling of the simple sorting program — especially how it reflects the Java program structure and how the program timing has been modelled.
- A description of the your verification attemps and the results of these (whether successful or not).
- A discussion of the coverage of the verification: Does it cater for specific scenarious only, or does it cover a more general class of possible uses?
- A description of your findings for the analysis tasks 4., 5. and 6.
- A section for each of the optional tasks describing your deliberations and results.
- A conclusion in which you should summarize your findings of the assignment.

Report Form

The report must be uploaded to the assignment at DTU Learn as a *pdf-file*.

The report must have a front page identifying the course, the assignment, the group number, and the participating students.

The report should be consise and preferably not exceed 10–12 pages. To this you may add appendices. Printouts of your automata and listings of declarations should occur in the report or as an appendix. Large automata should be presented by making appropriate excerpts.

Also the UPPAAL model(s) (with queries) and possibly Java programs must uploaded to DTU Learn in a separate *zip-file*.

General

The report and model files must be uploaded to DTU Learn no later than **Friday, March 22, 2024 at 23.59**.

Please note:

- There should be an introductory section declaring the contributions of the participants to the various parts of the work and the report. Each group member must be the sole contributor to one of the optional tasks. Otherwise contribution may be (evenly) shared among the participants. By participating in the group upload, you agree to this declaration.
- Any collaboration with other groups on smaller parts of the assignments must be declared and clearly identified. Collaboration on major parts is not acceptable.
- Any undeclared use of others' work is *strictly prohibited*.
- If you for some (very good) reason cannot meet the deadline you must contact HHL in due time before the deadline. Any excess of the deadline will be taken into account for the assessment.

Also note that amendments and practical details will be put on the *project page* found via the course home page. You should consult this if you encounter problems.