Final Report

# Introduction

Embedded Systems are everywhere. Think of coffee machines, lights control and ovens for example. But in the area of industrial engineering embedded systems are everywhere. Embedded systems are used to control machines. This project is about sorting black and white discs. The machine made in this project is not only able to sort, but it can also detect errors during its process and gives an adequate response. The machine is therefore able to tell something about its operating state. Something that is considered difficult.

Though this machine only sorts by color, there might be a wide variety of applications possible when using another comparison mechanism. At the supermarket you can bring empty bottles and get a deposit for it. These bottles need to be sorted for example by company such that Coca Cola gets the Coca Cola bottles and not those of Pepsi. Another use of a sorting machine might be used by companies like ASML, to pick erroneous wafers out of the production process. When using a weight sensor a sorting machine might be applied to sort luggage at an airport. Many applications of a sorting machine are used in daily life, though you might not notice it at home.

Sorting machines are thus widely used around the world in various industries. Many solutions to the sorting problem are already given, nevertheless is it a problem which fits quite well for first year students.

# Table of contents

LATEX YOLO

# 

# 

# Machine Design

## Introduction

We started our project by determining the features that we wanted to implement in our sorting

machine, like user interaction and exception handling. These design decisions form the founda-

tion of our mechanical machine design. The purpose of this document is to record the design

decisions we made in the first stage of the project, from a mechanical point of view as well as

a user point of view.

## Design requirements

In order to create a functioning sorting machine that can sort black and white discs, at least

the following components are required.

* A container for storing unsorted discs
* Two trays for storing the sorted discs
* A sensor that can distinguish the different color discs.
* A transporter that is able to move the discs from the container to the sensor.
* A transporter that is able to move the discs from the sensor to the appropriate tray.

## Current Design

The machine consists of a vertical tube in which the discs can be stored. A pin can be removed from the bottom of the tube, allowing the discs to fall on a platform below. A color sensor is attached underneath this platform, such that it can detect the color of the disc.

Next to this platform, a step motor is attached that rotates a

The disc is then shoved either left or right, depending on the color the sensor has detected, by the sorting wheel. The disc then falls of the

platform onto a seesaw. A gyroscope is attached to the axis of this seesaw. This gyroscope will

detect on which side of the seesaw this disc falls. The disc then slides off the seesaw and falls

into a mountable tray.

## Design Decisions

**Position of the color Sensor**

We had the option to attach the color sensor in various ways onto the machine. We chose to put it with the sensor pointing up, in such way that the arm of the motor could easily wipe the discs away. Placing the sensor below of the discs makes the sensor more reliable because the surface of the disc is larger than the sides of the discs. Also mounting the color sensor in any other position was structural more difficult than placing the color sensor below.

**The container**

The container is made in such way that the discs are stacked vertically. When the arm rotates and moves a disc away then due to gravity the discs will fall down. Therefore the transporter to move the discs from the container to the sensor is not needed anymore.

The height between the sensor and the color sensor is made a little bit higher than the height of one disc. This is done to ensure that only one disc is transported to the trays.

**Motor rotating arm**

The wheel containing the arms are used to transport the discs from the sensor to the trays. We had options to put 3 or 4 arms on the wheel. We chose to go for 3 arms because then we would be able to attach a touch sensor which is used for calibration.

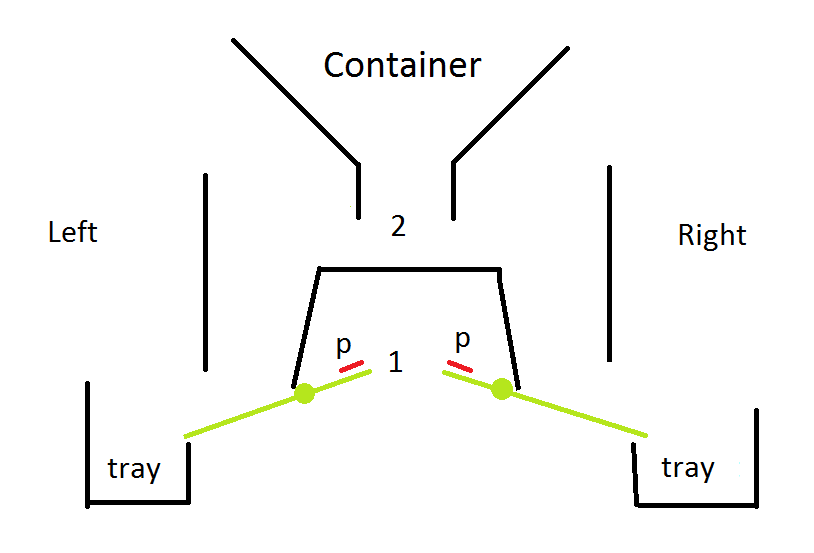
**The seesaw**

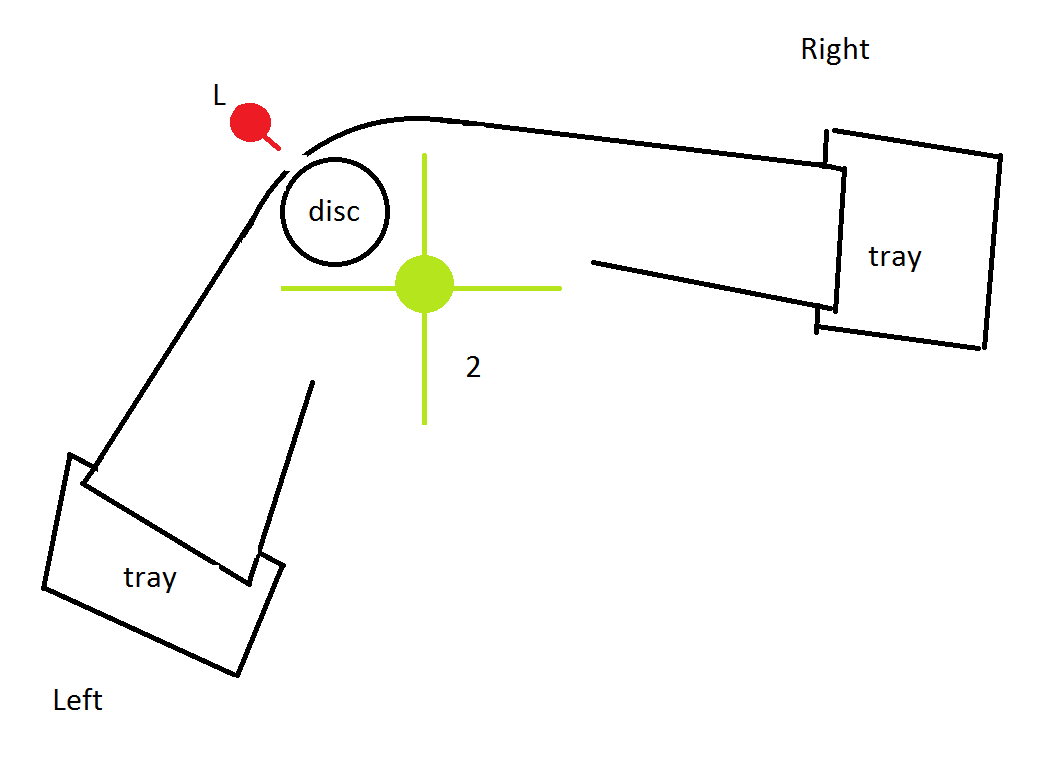
The seesaw is used to detect whether or not a disc arrives at the tray. Because the seesaw is placed below of the color sensor the discs will fall onto it by gravitational force. A gyroscope attached to the seesaw detects on which side of the seesaw a disc fell.

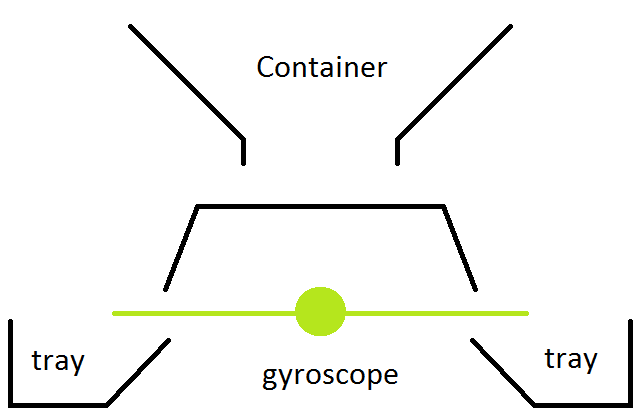
First we had a system which used touch sensor to detect if a disc arrived at the correct tray. The discs however had too less weight to trigger the touch sensor. Therefore we settled finally with the seesaw and a gyroscope attached to it.

**The trays**

We chose to use coffee cups from the university canteen. These cups are free as long as you buy a cup of coffee. These are easily replaceable when they break down for some reason. Convenient about these cups is that they have some additional promotional texts on it. Also the main reason to pick those was that they were the most easy to get. Other options such as tupperware boxes would require us to go to the city centre to buy those. We also had the option to build trays out of LEGO, this was not an option because we would lack LEGO beams.







## System Level Requirements

### Use Cases

The machine can be described as a finite automaton. This finite automaton is shown in figure

3. The machine has three buttons, which the user can utilize to control the machine. The

START/PAUSE button is used to start and pause the machine. The ABORT button is used to

stop the machine in case of emergency. The RESET button is used to reset the machine. The

user is required to add black or white discs to the container before starting the machine.

The machine, when finished without fatal error, produces two trays, one filled with white

and one filled with black discs. The machine will also display information about its current

state and progress on the display during the sorting process.

The sorting process is described by the following automaton. The initial state is the rest-

ing state. The user is required to prepare the machine as described in the User Constraints segment further in the document. None of the buttons should have any effect, except for the

START/PAUSE button. When this button is pressed, the machine should proceed to the

operating state.

In this state, the machine sorts the discs. Internally, it has a state in which it checks the

color of the disc, after which it moves to the correct sorting state in which it transports the disc

to the correct tray. If the RESET button is pressed in this state, the machine should return to

the resting state.

While in the operating state, if the START/PAUSE is pressed, the process should halt

and the machine should go into a paused state. When the START/PAUSE button is pressed

again, the machine should return to the operating state at the point where it was when the

START/PAUSE button was first pressed. If the RESET button is pressed in this pausing state,

the machine should return to the resting state.

While in the operating state, if the ABORT button is pressed or a fatal exception is encoun-

tered, the machine should immediately go into an exception state. In this state, the machine

must come to a full stop. The machine should only continue when the RESET button is pressed,

after which it returns to the resting state.

When the machine finds that there are no more discs left in the storage, it should go into a

finished state. Then, if the RESET button is pressed, the machine should return to the resting

State.

Machine Design.png

### User Constraints

Machine Preparation Before starting the operation of the machine with the START/PAUSE

button in the resting state, the user is expected to fill the disc storage device with black and

white colored discs that need to be sorted. Only discs are allowed in the disc storage and no

other objects should be placed in it. The user must also ensure that no discs are present in

other parts of the machine and that the disc trays are mounted to their mounting points prior

to starting execution. The disc trays should be empty. At most 12 discs may place into the

container. The user then has to proceed by starting the program. The tube can be filled with

discs after the machine has finished calibrating the moving arm.

After the machine has finished execution, the user will have access to two trays, one con-

taining exclusively white discs, the other only contains black discs. When the machine indicates

that it has finished its sorting procedure, the user should remove the trays from their mounts

so he can dispose the discs somewhere else.

Exceptions

Possible errors are as follows:

|  |  |
| --- | --- |
| Error Type | Fatal (to abort state requires user input to continue) |
| Disc does not reach the tray | Yes |
| Disc arrives in the wrong tray | Yes |
| Time to tray is higher than average | No |
| Wrong input (i.e. different color disc) | No |
| Motor jams | Yes |
| Gyroscope does not stabilize | Yes |
| Connection to peripheral lost | Yes |
| Battery low | Yes |
| Abort by user | Yes |
| Software exception | Yes |

When a disc does not reach the tray, the machine should stop, as there is may be an

obstruction in the path the disc takes from the sorting arm to the lever.

When a disc arrives in the wrong tray, the machine should stop, as something happened

that caused the disc to move to the wrong tray, even though the sorting arm tried to move

it to the correct tray. This means there must be a mechanical failure somewhere.

The Wrong Input error occurs when a disc is being detected with another color than white

or black. This could, for example, be a disc with the color red. This is not a fatal error because

we can specify an action that will be taken when this exception occurs. We may for example

want the machine to simply put the unknown disc in one of the trays anyway, say default the

destination to white, or we dispose all unknown discs into a separate tray.

The connection to peripherals lost exception is marked as a fatal exception, because a defect

in the controlling parts of the machine prevent it from functioning properly for obvious reasons.

The user has to fix this by hand before the machine can be used again.

The machine will stop operation when the battery has less than 5% of charge left. This

is done to prevent the machine running out of power in between operations and ending in an

invalid state.

When an error occurs in the machine that is indicated with fatal, the user must remove all

discs from all parts of the machine. The user should then press the RESET button in order

to put the machine in its resting state. The user may then proceed to prepare the machine for

execution again, as described above, if desired.

Safety properties

The machine should have the following safety properties. These are our

guarantees about the working and stopping conditions of the machine, in case something unex-

pected happens.

• The ABORT button must stop all moving parts within 10 ms. This to make sure that

the user is able to quickly stop the machine in case something cant be handled by the

Machine.

• When the machine starts with discs loaded into the container, then the discs will be sorted

when it arrives in the finished state. The sorting process takes at most 5 minutes in the

default safe mode. However software specification may specify more modes.

• A fatal exception must stop all moving parts within 10 ms. This ensures that the machine

does not damage or even destroy itself when it is blocked in any way.

## Machine Interface

Lejos API

The Lejos API provides access to buttons on the brick, sensors and actuators.

Buttons can be queried directly, but references to sensors and actuators must be instantiated

with a ’Port’ parameter indicating how they are connected to the brick. The following ports

are used in our design:

|  |  |
| --- | --- |
| Peripheral | Port |
| Color sensor | Port 1 |
| Gyroscope | Port 2 |
| Touch sensor | Port 3 |
| Motor | Port A |

Motors expose methods that allow the developer to have to motor rotate an arbitrary number

of degrees. Therefore, it is not necessary to implement pulse width modulation. One can call

a method to fetch a sample from a sensor in the form of an array of floats. It is up to the

developer to parse these in a meaningful way.

Motor rotating arm

This motor is connected to port A and rotates the arm which moves

the discs. The arm has three legs with an angle of 120 degrees. The motor has basically three

states: a resting state and two working states, one for every possible direction. The neutral

position of the three legged arm is the position where the two arms closest to the place where

the disc will land have an approximately identical distance to the disc. When the motor is in

the resting state the three legged wheel has this neutral position.

When the motor is in a working state the arms rotate left or right until the next neutral

position of the wheel (120 degrees further), where it will transition to the resting state of the

motor. During the working state the motor should not be jammed or obstructed in any way, or

the motor will be jammed and trigger an exception.

Touch sensor calibrating

The touch sensor, connected to port 3, is attached in such a way

behind the sorting platform that the arms of the sorting wheel are able to fully press the sensor

when it rotates. This way, it is possible to detect the position of the sorting wheel when it is

rotating, which can be used to calibrate the sorting wheel through software.

Gyroscope with seesaw

To the seesaw a gyroscope is attached, which is wired to port 2.

The seesaw has three states, a neutral state, a left-side down state and a right-side down state.

The neutral state is the state in which the gyroscope when read returns a value indicating that

it is in balance. The seesaw enters a left-side down state when a disc falls on the left side. The

gyroscope will read a value indicating that it was unbalanced and that the left side went down.

This is analog for the right side down state.

Color sensor

The color sensor, connected to port 1, gives a value indicating the color of the

surface above it. The color sensor returns an integer indicating the color of the disc (e.g. 0

for red, 1 for green etc.) currently on the platform, or -1 if only ambient light was reaches the detector, indicating that there is no disc on the platform. The sensor is able to distinguish 15

different colors, as shown in the following table:

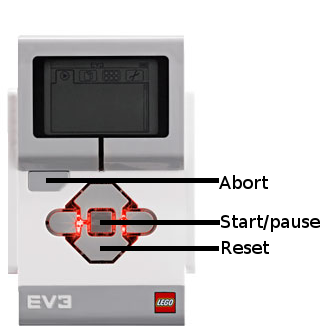
|  |  |
| --- | --- |
| Constant Field | Value |
| [BLACK](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#BLACK) | 7 |
| [BLUE](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#BLUE) | 2 |
| [BROWN](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#BROWN) | 13 |
| [CYAN](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#CYAN) | 12 |
| [DARK\_GRAY](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#DARK_GRAY) | 11 |
| [GRAY](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#GRAY) | 9 |
| [GREEN](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#GREEN) | 1 |
| [LIGHT\_GRAY](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#LIGHT_GRAY) | 10 |
| [MAGENTA](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#MAGENTA) | 4 |
| [NONE](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#NONE) | -1 |
| [ORANGE](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#ORANGE) | 5 |
| [PINK](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#PINK) | 8 |
| [RED](http://www.lejos.org/ev3/docs/lejos/robotics/Color.html#RED) | 0 |
| WHITE | 6 |
| YELLOW | 3 |

The brick

The brick controls the sensors and actuators. The state of the buttons is either

up or down, which may be queried using the Lejos API. The buttons used are shown in figure

4.



## Conclusion

The design decisions recorded in this document will be built upon in the next phase of the

project, where we are going to specify the software specifications. The software specifications

are heavily dependent on the machine interface described in this document, since the machine

interface describe our assumptions about the machine API and the mechanical implementation.

The software specifications will describe how we are going to use this interface and how the

machine will behave.

# 

# 

# Software Specification

## Introduction

After having designed the machine, we are able to specify how our software should behave. We

do this by describing the software in a state machine with abstract states. This state machine

also implements the different operation modes. After this we characterize the distinct inputs

and outputs. We will convert the state diagram into a working UPPAAL model and explain

the working systems in detail. Finally we explain the verification of our model. This document

is called the software specification.

## Operation Modes

Because the gyroscope is rather slow and has to stabilize, we have to wait for it to stabilize before

we can sort the next disc, thus causing a bottleneck. If we want to sort very fast, we should skip

the input from the gyroscope and sort the discs immediately after each other. Therefore, we

want to implement 2 modes: a fast mode in which we dont check for input from the gyroscope

and thus dont check if the disc actually reaches the tray, and a slower safe mode, where we

check for input from the gyroscope and we can make sure that discs do reach the tray. Besides

this, we want to make an additional incremental mode for debug purposes, in which the user

has to press a button for the machine to sort the next disc.

### Fast Mode

The machine starts in a resting state. If the Start/Pause button is pressed in this state, the

machine should go to the checking state. In this state, the machine checks which color disc is

present on top of the light sensor. There are four different transitions the machine can make at

this point, depended on the detected color. If the machine is black or white, it should go to a

respective state in which the motor is rotated accordingly. When the motor has finished moving,

the machine simply returns to the checking state. The machine can be paused by pressing the

Start/Pause button anytime during the checking process. The machine then goes to a paused

state as soon as it enters the checking state and waits for the Start/Pause again to return to

the checking process.

### Safe Mode

The safe mode is exactly the same as the fast mode, with a few additional safety checks. This

means that it has the same assumptions according to the input. There is still an arrow from

every state except the initial state to the abort state for when the ABORT button is pressed.

There are two additional states. One for each color, black or white. After the motor is finished

turning, it enters a state in which it waits for input from the gyroscope (G), which can indicate

a disc that felt to the right (R) or to the left (L). When the input is correct, it goes to the state

in which the color is checked. When the input is wrong (Exception) (wrong side, no input at

all after a while), the machine goes to the exception state.

### Incremental Mode

The incremental mode is again almost the same as the safe mode, so again the same assumptions

about the input and the state machine, but now there is 1 extra state between the ingoing arrows

to the state in which the color is checked. In this state, the machine waits for a press on a button,

the START/STOP button (S/P) to continue. If that button is pressed, it goes to the state in

which the color is checked. All the ingoing arrow that were going to the color-check state are

now going to the wait-for-button-press state. There is 1 exception for this. From the resting

state, you go to the color-check state immediately after a press on the button. It would be silly

if you would have to press that button twice for something to happen.

## State Machine

statemachine.png

We have indicated the transitions for specific modes using a color coding. Green transitions only

apply to the fast mode. Red transitions only apply to the incremental mode. Blue transitions

only apply to the safe mode.

All symbols are inputs or outputs defined in the next chapter. However some symbols are

not input or output but something else, these are documented here:

|  |  |
| --- | --- |
| Symbol | Description |
| TS | Starts a timer which increments variable "*t*" at a fixed time rate. |
| TR | Resets the timer by setting "*t*" to 0 |
| R | Reset button is pressed |
| VR | Full reset, reset of statistics and buttons variables |
| *t* | The variable containing the current timer value |
| *tdmax* | The maximum timer value after which it is decided that a disk was lost |
| *tavg* | The average time of a disk to get in the tray |
| *tgmax* | The maximum time it should take to stabilize the gyroscope. |
| *paused* | Variable indicating if the machine should go into the paused state or not. |

## Signals

### Inputs:

Color c: The input from the sensor with the following possible values: ”N”, ”W”, ”B” and

”U”. The value ”N” implies that no disk has been detected in front of the color sensor. The

value ”W” implies that a white colored disk is in front of the sensor. The value ”B” implies

that a black disk is in front of the sensor. Finally a value ”U” means that an unknown disc is

in front of the sensor.

Motor M: The input from the motor can be that it is jammed, J, or that it is finished, F.

We use these inputs as a method to detect if we can go to the next state, or to detect if there

is a possible error.

Abort A: This signal is high when the abort button has been pressed. This signal indicates

that the machine should be halted nearly instantaneously and that the machine should enter

the Abort state.

Reset R: This signal is high when the Reset button has been pressed. The signal will be

used to bring the machine back in its resting state and reset all the statistics about the current

sorting operation.

Start/Pause S/P: This signal is high when the Start/Pause button has been pressed. This

signal will be used to start and pause the operation of the machine, depending on its current

State.

Gyroscope G: This signal might have three values ”L”, ”R” and ”S”. ”L” indicates that

gyroscope is left, ”R” indicates that the gyroscope is right and ”S” indicates that the gyroscope

is stable.

### Outputs

Color Sensor CS: Whether the color sensor is currently reading the sensor values. In order

to read the values, a light from within the sensor has to be activated in order for the detection

to work. This means that this light is an output for the machine. While the light is activated,

the sensor is constantly reading the sensor values, but we only read it in certain situations. The

color sensor is only active while the machine is in the operating state and is therefore disabled

in the Resting state, Abort state, Finished state and the Paused state.

Motor M: This signal has three possible values ”L”, ”R” and ”O”. Where ”L” is the motor

moving to the left for 120 degrees, ”R” is the motor moving to the right for 120 degrees, ”O”

is the motor turned off. The sorting arm moves, depending on the color read.

LED Active E: The EV3 controller brick has a built-in LED behind the buttons that we

can use to give feedback to the user. We can specify the flashing pattern from a limited set of

patterns and we can specify the color. The LED will be emit a constant green color while the

machine is in the Finished state. The LED will flash in red while it is in the Abort state. The

LED will flash orange when it is in a warning state.

Sound: The brick also features a built-in speaker which we can use to give even more feedback

to the user. We will use this speaker to play an error sound when the machine enters the Abort

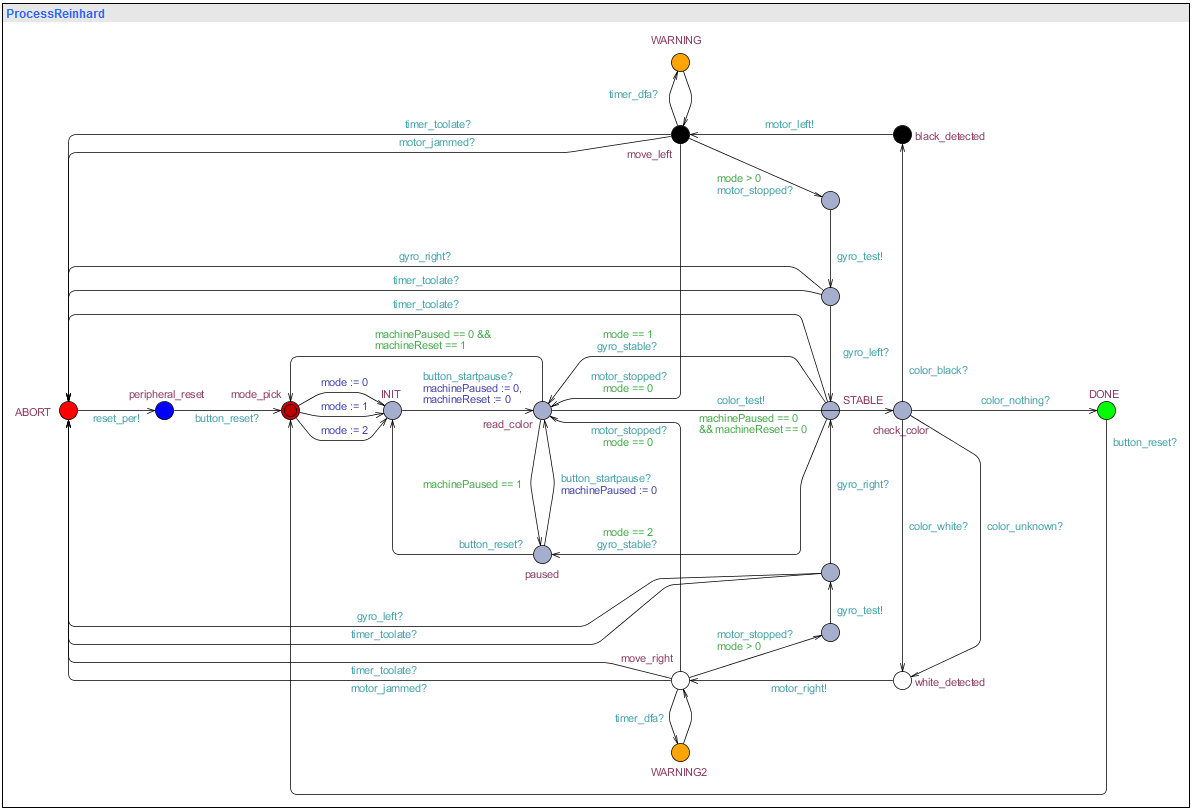
state. A sound will also be played when the machine enters the Finished state.

## UPPAAL Model

An UPPAAL model consist of multiple systems that communicate over several channels. The

systems in our UPPAAL model are Reinhard, our main system, along with several other pe-

ripheral systems.



### Main Process

**Choosing the mode** The initial state is mode pick, which is not present in the state machine.

In this mode one of the three operation modes is chosen. This is important for the verification

process, because the UPPAAL model uses an internal variable to store the mode, whereas the

state machine indicates the different modes with colors. If it is not possible to set this variable

in the model itself, the verification process will only verify if a query holds for one mode.

Initialiation Upon choosing a mode, we enter the state INIT, which is equivalent to the

initial state in the state machine. Here, the model waits for the button startpause signal from

ProcessButtons. Also, it sets the machinePaused and machineReset variable to 0, which ensures

that the model does not immediately pause or reset.

**Reading colors** Upon leaving the INIT state we enter the read color state(read in the state

machine). Whenever the machinePaused variable is 1 in the read color state, the model goes

to the paused state. If the machineReset variable is 1, the machine returns to the INIT state,

but this has the lower priority. If both of these are 0, the color test signal is send and the

model transitions to the check color state. In the check color state it waits for a signal of

the ColorSensor system depending on whether there is a disc and which color the disc has.

Depending on the color, it goes to one of two parts of the main system, one for each distinct

color(black or white). We will describe one of these parts as they are mirrored. Upon receiving

the signal of a color, the model transitions to a detected state. Here, it sends a signal to

**ProcessMotor** to start moving in the appropriate direction and goes into a move state. The

transition the model makes here is dependend on the mode variable.

**Check sequence** If mode is not 0, indicating that the system is in the safe or incremental

mode, it will add a check sequence using the gyroscope. After the signal from the motor is

received that it has stopped, the main system will send a test signal to the GyroSensor waiting

for a signal back. If the correct signal is received back, indicating that the disc has arrived in

the correct tray, it will return to read color state. If mode is 0, indicating that the system is in

fast mode, this check is skipped and the model returns to the read color state from the move

state as soon as it receives the motor stopped signal from ProcessMotor.

**Finishing up** When the model receives the color nothing signal from ProcessColorSensor, the

machine is finished and the model goes to the DONE state. Here it waits for the button reset

signal and returns to the initial state.

**Exceptions** In our machine design is it specified that there is an ABORT button present that

will stop the whole machine almost immediately. However, we have chosen not to include the

abort button in our UPPAAL model as a possible transition for every state because this would

be rather cumbersome and just a small implementation. We have however chosen to include

possible exceptions such as a jammed motor or a signal that has not arrived in decent time.

These transitions will lead the system into the ABORT state. In this state the process waits for

the reset button to give a signal, after which it will reset the other processes and return to the

intitial state. Note that it is not required to actually reset the gyroscope in the actual machine,

just in the UPPAAL model. Non-fatal exceptions are also represented in the model. Deviating

average detection time is represented in the states WARNING and WARNING2, which the

model reaches when the timer dfa is send from ProcessTimer. Process ColorSensor can also

send a color unknown signal, which sends the machine to WARNING3. These warning states

simply return to the state they were detected in and let the model continue.

### Subprocesses

The following are all subprocesses that the main process depends on:

**ProcessColorSensor** waits for the color test signal. It can then send four different signals:

color black, color white, color unknown, and color nothing.

**ProcessMotor** waits for the motor lef t or motor right signal and moves to the respective

state. From there, it can send either a motor stopped or a motor jammed signal. It also listens

for a reset per signal, after which it returns to the initial state.

**ProcessGyroSensor** waits for the gyro test signal from the main process. It can then send

the following signals: gyro stable, gyro lef t and gyro right. If either gyro lef t or gyro right

is send, the machine goes to an additional state where it is able to send a gyro stable signal,

indicating that the gyroscope has returned to its initial position. The process also listens for

the reset per signal, for resetting purposes.

**ProcessButtons** is continuously able to send button signals. This simulates that any button

can be pressed at any time, which is necessary for the verifier to check all possible transitions.

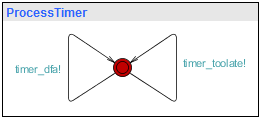
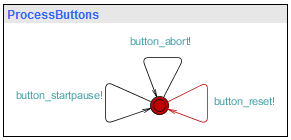
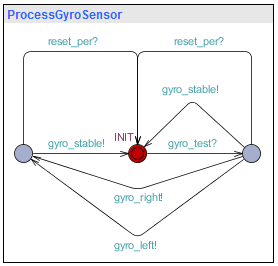
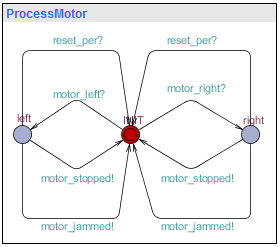
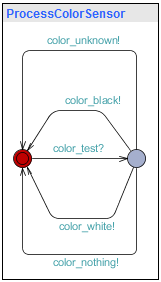
**ProcessTimer** works the same as ProcessButtons, but with signals that an internal timer

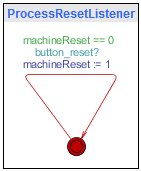
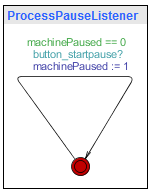
would send if it detected one of the errors.

**ProcessPauseListener** and **ProcessResetListener** continuously listen to the button startpause

and button reset signals from ProcessButtons. When they detect the respective signals, they

set the respective variables to 1 to alert the main process that the button is pressed.





## Verification

The main reason for recreating the state machine in UPPAAL was to allow extensive verification

of the model. With UPPAAL we can verify the events like deadlock will never occur. The

UPPAAL query language is a minimalistic but functional language to specify these properties.

We use the following queries to verify our model:

**Abort state reachability** Query: E<> ProcessReinhard.ABORT

Description: There exists a path where the system reaches the ABORT state.

We want to ensure that the machine halts in case of a fatal error. This check ensures that it is theoretically possible for a fatal error to occur and jump to the ABORT state.

**Peripheral reset** Query: A[] (ProcessReinhard.paused or ProcessReinhard.INIT) imply (Pro-

cessMotor.INIT and ProcessGyroSensor.INIT)

Description: Whenever the system is in the paused or initialization state, the motor and gyro

sensor must be in their initial state.

The goal is this check is twofold: We want to ensure that all peripherals are idle when the

machine is in the paused state or has not yet started, and we have to be certain that after

reaching the ABORT state, all peripherals are correctly reset. If we would leave the gyroscope

in its left or right state, we would cause deadlock in the next cycle. Worse still, leaving the

motor running after an ABORT could harm to machine.

**Full speed independent of gyroscope** Query: A[] mode == 0 imply ProcessGyroSen-

sor.INIT

Description: When in full speed mode, the system must leave the gyroscope in its initial state.

Full speed mode is supposed to ignore the gyroscope to speed up the sorting process. Here we

check if the machine correctly uses the mode setting.

**DONE state reachability** Query: E<> ProcessReinhard.DONE

Description: There exists a path where the system reaches the ABORT state

This check ensures that the machine can eventually finish the sorting process.

**Machine pause justify** Query: A[] ((ProcessReinhard.paused and mode != 2) imply ma-

chinePaused==1)

Description: Whenever the system is in the paused state and not in incremental mode, ma-

chinePaused must be set to 1.

Verifies that in the full speed and safe modes, the machine only reaches the paused state when

the pause button was pressed beforehand.

**Valid mode specified** Query: A[] mode==0 or mode==1 or mode==2

Description: Mode must be one of: full speed, safe or incremental.

The mode value must be valid.

**No deadlock** Query: A[] not deadlock

Description: The system never deadlocks in any case

## Conclusion

This document provided the information about the behavior of our software. We introduced

our state machine diagram which we then implemented in UPPAAL. We clearly explained the

working of our systems in UPPAAL and verified our model using this program. This part is

used as the basis for the design of the software and is very important.

# Software design

## Introduction

This document will describe the design decisions made while building the software, based on the UPPAAL model specified during the software specification phase. It will also include an early draft of the software that will control the machine. This software will be written in Java, which means that we are able to use the object oriented programming paradigm. The object oriented design of our program is described in a class diagram.

## Object Oriented Design

We have chosen to implement the states using the object oriented programming paradigm. The state implementations follow directly from the finite state machine described in the software specification. The class diagram of figure 1 provides an overview of all states and auxiliary classes. In the diagram the '+' symbol indicates a publicly accessible method or property, “-” indicates a private accessible method or property, bold indicates a constant property or a method which cannot be overridden and underlined indicates a property or method which is accessible through the class.

# classdiagram.png

### Main class

The main class handles button presses, updates the display and executes the 'run' method (REF to run) on the current state. It also exposes properties which provide access to the actuators and sensors. These are objects from the LeJOS API. The properties providing this access are *motor, gyro, color, aButton* (the abort button), *spButton* (the start/pause button) and *rButton* (the reset button). These properties are declared public to provide easy access, and these are declared final to make sure the properties can’t be changed.

Then we have two more properties, *paused* and *reset*. These two booleans correspond to the paused and reset flags of the software specification. These flags can be accessed by getter and setter methods.

Furthermore we have a property storing the statistics object of type Statistics. (REF)This is a final and global property.

Finally a property indicating the current mode of the system is provided. For this property a getter is provided.

### The run method

The main method of the class Main is the method which is called when the program is started. That main method calls the run method of the Main class. The run method contains the main loop of the algorithm.

The run method

1. while(true)
   1. If abort button is pressed && current state is not abort state
      1. Then current state = new AbortState()
   2. Else if start pause button is pressed
      1. Then paused = true
   3. Else if reset button is pressed
      1. Then reset = true
   4. current state = current state.run(this)

### Statistics class

The statistics class is meant the store three counters for unknown, black and white discs. Although it is not shown in the class diagram of figure 1(REF), it does provide methods to modify these counters.

### Display class

The display class has some methods to draw output on the LCD screen of the brick. It has a method for normal running, for errors and warnings and for the mode-pick menu.

### Mode enumerable

The mode enumerable can be set to either fast, safe or incremental. It stores the current mode in the Main class, which can be referenced by all states for use in a transition guard.

### Abstract class 'State'

This abstract class is the base class for all states. It defines the *nextState* method and provides a default display showing the current state and the number of sorted discs by colour. Other states can also opt to override this method and implement a custom display, such as a warning. The *nextState* method returns the next state of the machine, which may also be the same state if none of the guards are met.

### State implementation classes

Nearly all states in the Uppaal state machine have a counterpart in java, although some of them have been merged, as we do not have to transition to a state to send a signal.

### The WarningState class

In the finite automaton of the Software Specification there are multiple warning states. All these warning states have in common that they have one or more transitions to the warning state, then a warning state which displays the warning on the display of the Brick, and after that it goes with one transition without a guard to some other state. The WarningState class takes on construction of a new object a parameter of the type Warning and a parameter indicating to which state to go after the warning. The warning object contains information about the warning and the text to display on the display.

The finite automaton contains multiple warning states. All of these have in common that they are transitioned into from one of the operating states, then display a warning message and return to the operating state they were entered from. We have implemented this using one *WarningState*, to which we pass a the relevant warning type and the state to return to. The warning state will then display the warning and transition to the return state.

### The AbortState class

The abort state class differs in the sense that it has an additional parameter on construction indicating the type of error which occurred. The abort state object then uses the information inside that object to display the appropriate message.

The *AbortState* takes a parameter indicating the type of error to report. It overrides the *displayUpdate* method to display the error message until the reset button is pressed.

## Arguing correctness

We need to argue three things to argue that the software design is correct and will work:

1. the states are implemented as in the state machine
2. the button flags are set as required
3. the abort button triggers the system to go to the abort state if it is not yet in the abort state

### 1. States implementation

The Main class holds a pointer to the current state. On initialization the current state is set to a new ModeSelectionState object. This is done in the constructor of the Main class. Therefore the system starts in the initial state just like the finite automaton of the Software Specification. Then the main loop iterates forever and sets the current state to the result of the method run of the current state object. The run method of a state object should as documented do its task, then check the guards of the transitions and after that do a transition if the guard is satisfied. Doing a transition implies returning a state object which is the state the system transitions to. Then the current state pointer is set to that returned state object and next iteration the run method corresponding to the next state is executed. When the run method of a state object behaves as described and implements the state, its guards and transitions correctly, then it follows that the system behaves as expected, because in that case it is the same as the state machine we already have proven.

### 2. Button flags are set as required

We want to argue that when the buttons are pressed, the flags are set in a short amount of time. The loop of the run method first checks if the abort button is pressed. Then if the abort button is not pressed or if the current state is the abort state then the start pause button should be checked. Since the abort button has a higher priority than the start pause button this is no problem. So the paused flag is set to true when the start paused button is pressed, when the abort button is not pressed or when the current state is the abort state. Then if all of these cases do not occur, then the reset flag needs to be set if the reset button is pressed. Therefore the loop sets the flags as required, given that the run method takes less time than it takes to press a button. If a user presses a button longer than the running time of the run method, then the flags are set correctly. Because the run method is very short, the time it takes to press a button will always exceed the time it takes to execute one iteration of the run method.

### 3. Pressing the abort button triggers the system to go into the abort state, if it is not yet already in the abort state.

The loop first checks if the abort button is pressed and it checks if the system is not in the abort state. Then, if this is the case, then the current state pointer is set to a new abort state object and the run method of that abort state object is run and therefore the system is in the abort state. Again we have the constraint that the run method should run in less time than an average human presses a button. However, this isn’t a problem as said before.

### Conclusion

This document has provided a diagram depicting the class structure of our object oriented program. This diagram serves as the basis for the software that will be fully constructed in the software implementation phase. We also created an initial version of this software and proved its correctness in this document. This is an important step in finalizing the software that will control our machine.

# Conclusion

# Sources

Color table: http://www.lejos.org/ev3/docs/ - Constant Field Values