Technische Universiteit Eindhoven University of Technology

Eindhoven University of Technology
2IO70 DBL Embedded Systems

Making the Sorting Machine

Member:	*Student №:*
Hofsté, M.L.	0996144
Lacquet, T.L.	1016388
Scheepers, Y.J.A.	1002370
Wordragen, G.C.	1024503

tutored by: ir. Huberts, T.P.M.M.
supervised by: dr.ir. Cuijpers, P.J.L.

June 4, 2020

Contents

1 Summary **3**

2 Introduction **4**

3 System Level Requirements and Use Cases **5**
3.1 Requirements . . . 5
3.2 Use Cases . . . 6
3.3 User Constraints . . . 6

4 Prototype 1 **8**
4.1 Machine specification . . . 8
4.2 Machine design . . . 8
4.3 Machine implementation . . . 8
4.4 Software specification . . . 9
4.5 Software design . . . 9
4.6 Software implementation . . . 10

5 Machine Specification **12**
5.1 Fault Detection . . . 12
5.2 Ejecting discs . . . 13
5.3 Moving discs . . . 13
5.4 Sorting discs . . . 13

6 Software Specification **14**

7 Machine Design **15**
7.1 Cam Plate . . . 15
7.2 Conveyor Belt . . . 15
7.3 Containers . . . 16

8 Software Design **18**
8.1 Last disc . . . 19
8.2 Emergency button . . . 20
8.3 Error detection . . . 20

9 Machine Implementation **21**
9.1 Cam Plate . . . 21
9.2 Conveyor Belt . . . 22
9.3 Sensors . . . 22
9.4 Containers . . . 22
9.5 Overcurrent protection . . . 23

10 Software Implementation **25**
10.1 Walk through . . . 25
10.2 Branch problem . . . 28

11 Testing **30**
11.1 Component Tests . . . 30
11.2 Unit Test . . . 30
11.3 System Test . . . 30
11.4 Validation Tests . . . 31

12 Conclusion **33**

13 Literature Overview **i**

14 Appendix **i**
14.1 Error Manual or Debug Manual i
14.2 UPPAAL model iii
14.3 Source code iii
14.4 Debug program min_max_brightness x
14.5 Logbook xii

1 Summary

For this assignment, a sorting machine was constructed made from FischerTechnik parts. It is powered and controlled by the "PP2" processor, which is given directions through a custom assembler in which the source code is written.
The machine is able to safely sort any amount of black and white discs the user has put in, into its two containers. To do this, it takes a little over two seconds per disc. It can operate under all conditions that can be considered "normal"; for example the (absence of) lighting does not affect the machine, as long as there is no strong lamp shining directly into a light sensor.
Furthermore, the sorting machine is able to detect almost all errors that can occur during runtime, both on the mechanical and on the electrical side. When such a fault occurs, the machine will, along with its displayed error state and accompanying error guide, run the user through the process of resolving it, after which the machine will once again be operative.
The machine consists of three subsystems: one that uses an egg-shaped disc to push discs out of the tube in a controlled manner and checks whether there are still discs to sort with a lamp and a light sensor; one that uses a conveyor belt to transport the discs from aforementioned subsystem to the next, taking note of its colour along the way; and one that consists of two containers that horizontally move in such a way that each disc falls in the container that corresponds to its colour. Each of these units makes use of a motor and has one or two limit switches present to keep track of this motor's movement.

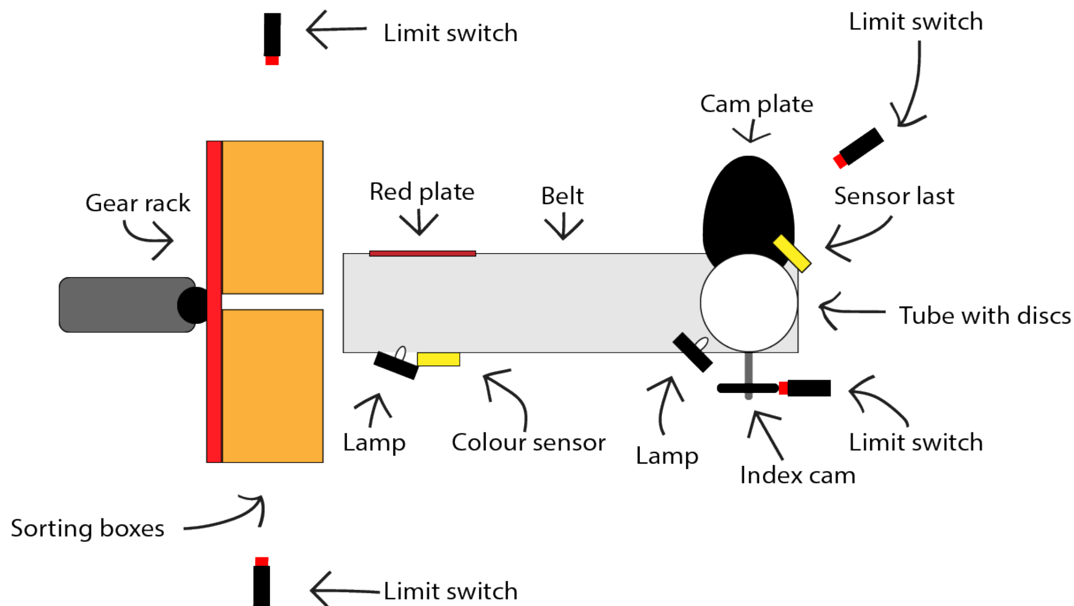

Figure 1: A systematic drawing of the sorting machine

2 Introduction

This report is for the course DBL Embedded Systems, provided by the University of Technology Eindhoven. The structure of the report follows the several stages of the V-model. For every stage of the V-model there is a section dedicated to explaining how it was used for the machine.
To get a better grasp at how everything works, a quick iteration of the entire V-model was followed during the first two weeks, doing every step very crudely. This helped find out how the hardware specifications should be and how the assembler code needs to look.
After this a second iteration V-model was carefully followed and our findings on it were documented. The report starts with a section about the User Requirements and Use Cases. In this section it is shown how the machine should be operated and what is and is not possible to do on the user side.

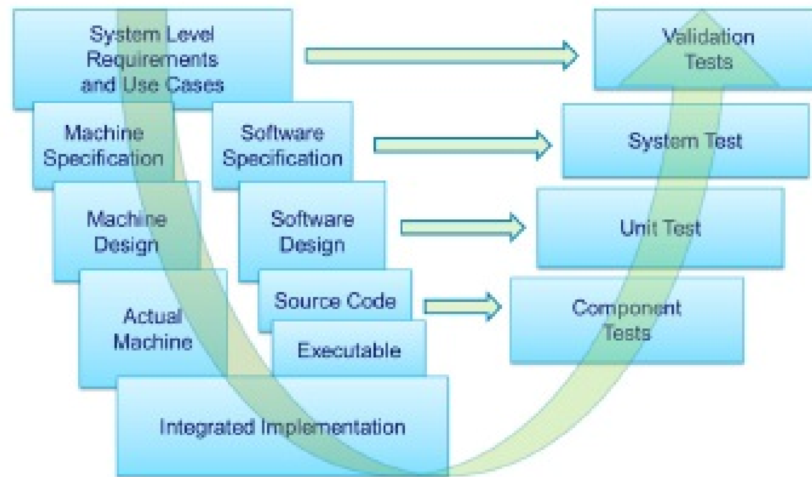

Figure 2: The V-Model

3 System Level Requirements and Use Cases

The System Level Requirements and Use Cases (SLRUC) are the same for both the different prototypes, since every prototype is built to do the same thing with the same rules. This is why it does not appear in the section on the first prototype as well, unlike most other parts of the V-model.
Our System Level Requirements are based on the general requirements that the Embedded Systems course's project guide provided.

3.1 Requirements

These are the requirements we have for our Artefact To Be Desired (ATBD) and for what use case the product is developed:

Mechanical movement with purpose
The machine mechanically moves white and black discs with a clear purpose. It is not allowed that the machine just moves discs without a clear reason. In the case of the sorting machine, it should be able to distinguish the black and white discs from each other and then move them to the appropriate container.

Sensible use of motors and sensors
The operation of the machine involves the use of motors and sensors in a meaningful way. In this case that means that the motors need to interact with the discs one way or another when a sensor detects that action is needed. This means for instance, that a sensor is used in detecting the presence of a disc and the colour of said disc. Another use for the sensors mentioned by the Project Guide is that it is allowed to use them to detect if the system is acting as it is supposed to, and to verify that all the parts of the machine are operating correctly.

Only use the PP2-processor and the FischerTechnik parts
It is not allowed to use any processor or coding language other than respectively the PP2 and Assembly. Nor is it allowed to use other parts than the FischerTechnik parts provided to us, since we are doing the standard sorting machine.

Start in resting state
From a high-level point of view, the machine starts in a resting state. The general idea here is that the machine starts in a state in which all the motors and lamps are off. In this state we are supposed to make the machine ready. This means removing any parts that stayed on the conveyor belt during the last run and filling the tube containing all the discs to be sorted.

Machine starts performing after signal (i.e. Start button)
While in the resting state, after the user is done preparing it, it can be given a signal in the form of a button push so that the machine knows that it can start sorting.

Emergency button
The machine has an emergency button which will bring it within 1 second into, what is perceived as, an emergency state. This state is allowed to be the resting state. It must do this in an as safe as possible manner. In our case this would be to immediately stop all motors and kill the lights. Adding to this the machine should be able to be restarted with human intervention. So manually removing discs from the machine and setting the components in the right position again to get the machine to function again is allowed.

Reports on its internal state
There are multiple states in which the machine can be, an UPPAAL model is constructed in which a visual representation describes the states and what they do. A state can for example be, to push a disc forward or to get the colour or presence of a certain disc. The 7-segment display available on the PP2 is used to show

in what internal state the machine is in.

Recognizes mechanical failure
Using additional sensors and time-outs the machine should be able to detect a mechanical failure. This could be a motor getting stuck, or a disc getting stuck somewhere in the machine. When the machine detects that there is something wrong it should enter the emergency state mentioned earlier and report this to the user. Since the PP2's display is not big enough to clearly report errors all by itself, we should use error codes and explain these in an accompanying guide.

Recognizes electrical failure
Using additional sensors and time-outs the machine should also be able to detect an electrical failure. Examples of this would be a disconnected wire. When this happens the machine should also enter the emergency state just like with a mechanical failure and also report this to the user. This reporting should be done the same way mechanical failures are reported.

Some sort of efficiency
The last requirement stated in the Project Guide is efficiency. The metrics they use are:

1. How many discs can be sorted per minute; Worst-case, bast-case, and on average?
2. Are any of the lamps and motors turned on more often than strictly necessary?

Furthermore they mention that these are not hard requirements, but that evidence needs to be provided for each of these metrics. In other words, documentation should exist on the speed of the machine under some different circumstances, and it needs to be reasonably efficient and fast.
All motors and lamps should be turned on as little as possible.

3.2 Use Cases

Before users are able to use the machine, they will need to connect the PP2 to a computer with the PP2 debugger program and the executable on it. The instructions in the Board User Guide will then help them connect the PP2 to the computer.

To connect the PP2 to the hardware, the PP2 needs to be positioned besides the hardware. Every input and output is labelled with numbers corresponding to the numbers on the PP2, allowing the user to easily find the matching pin for every cable. The input wires are labelled with blue numbers, while the output wires are labelled with black numbers.

After that the executable can be run, which will first let the machine wait in the start state, as required by the System Level Requirements.
Now the user can simply insert the discs to be sorted, while adhering to the User Constraints, and is to then press the start button.

In case of an emergency that the machine cannot detect, users should press the emergency button. If the machine did detect it, following the Error Manual will resolve the problem.

3.3 User Constraints

Before starting the machine, the colour sensor might need to be calibrated slightly. This is done with a program provided by us with the name "*min_max_brightness*", which also appears in the appendix.
The user is then expected to fill the tube with discs, with all discs oriented the same way and the flat side

up or down, and remove all discs from the rest of the machine.
Apart from pressing the start and emergency buttons, loading and unloading the discs, and resolving errors with help of the guide, the user should not interfere with the machine and its PP2 in any way, except when there is an emergency.
It is expected that only one electrical or mechanical fault occurs at a time. When such an error is detected, the user is supposed to follow the instructions listed in the error correction guide.

4 Prototype 1

To get more acquainted with the FischerTechnik pieces, the V-model, and the project as a whole, the decision was made to do a quick iteration of the V-model. Another reason this was done, was to find out which methods of sorting are and which are not feasible. This would help set up better specifications and make better designs for the next version.

4.1 Machine specification

For this prototype, the main focus was on speed. Besides that, the decision was made to not make it too complex, because it was already known that this would not be the final version. One constraint was to avoid using conveyor belts to come up with a more creative and less obvious solution. This constraint would also help in making sure the design is fast and somewhat simple. A side effect of these specifications is that the end product would end up very compact.

4.2 Machine design

The discs fall from the tube onto a small platform, after which a two-pronged, fork-like structure is used to directly push them into the correct box. This can be done very quickly and efficiently, which is in line with the focus on speed. The colour of the disc and whether or not there are still any left would be checked while it is still in the tube via an analogue light sensor and a lamp.
In the beginning, concrete ideas for error detection where avoided. This was not a big issue, since this iteration of the V-model was just for orientation purposes as mentioned earlier. There was, however, some time spend on thinking how error detection should be implemented.
This resulted in the realization that the new design needed to make the discs move a greater distance, upon which our next prototype is built.

4.3 Machine implementation

While implementing the machine design, one issue that presented itself was making sure the fork is correctly moved by the motor. The two possibilities were to have the motor on top of the gear rack and against it. The former was less efficient, because it required a much more complex supporting structure, which would have made it easier to make mistakes and caused the motor to slip semi-regularly. It is for this reason that the latter was used.
Another issue that proved to be difficult, was to add sensors for colour and fault detection to the machine. To prevent this from happening in the future, the future machine machine designs where made to leave as much space in them as possible.
For the motor to move in both directions it is necessary to use an H-bridge, which gives the power to choose in which direction the current flows through the motor. As it turns out from one of the PP2's manuals, the processor has special outputs specifically designed as H-bridges which we can connect the motor to.

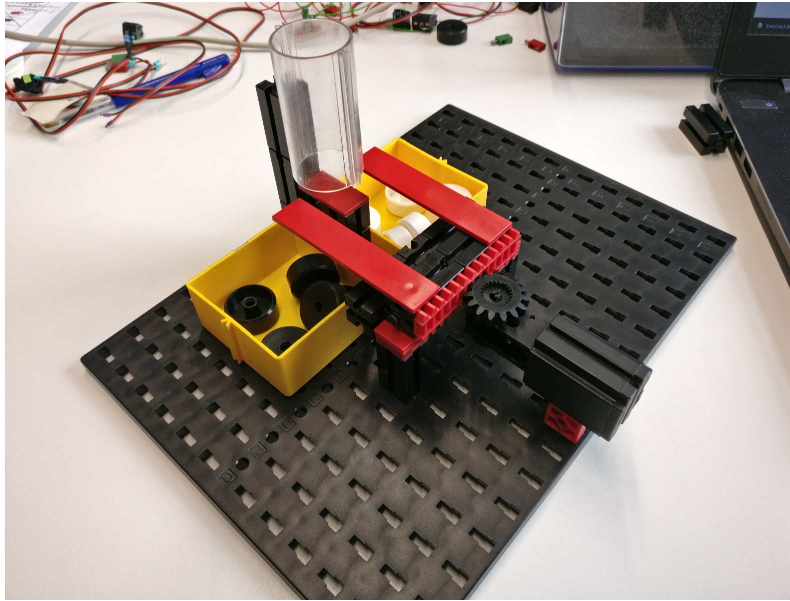

Figure 3: The first prototype

4.4 Software specification

The machine would not be built to an extent where an extensive software specification was necessary. To test it, all that was required was a simple program that gave a Pulse-Width Modulated output corresponding to the input buttons pressed (so pressing button X would power output X). Not using Pulse-Width Modulation (PWM) is not an option, as the project guide very clearly indicated that this would be bad for the motors and might cause them to break down. The PWM program mentioned is extensively written about in Software Implementation.
This program is used for manually controlling the motors and it is also a very important part in the final design of the sorting machine.

4.5 Software design

A first observation was that because the PWM program must be used to control the motors, supposedly simple features like a stop button can be hard to implement. When a new feature needs to be implemented the code inside the PWM subroutine must also be changed, which means the code can get complex very quickly. To circumvent this, the PWM subroutine was coded in such a way that this can be done more easily: by using a timer interrupt for the Pulse-Width Modulation, while also making sure the subroutine does not rely on any other subroutine and has its own places in memory to work with, it will stand completely on its own and the rest of the code can be written independently of the PWM implementation.

Even though not much progress on the actual assembly code was made and necessary, an actual UPPAAL model was made (shown below) for the code that would be needed to power this prototype. This was, like the rest of this iteration, mostly to get more acquainted with the program.
The basic idea of this automaton is as follows: first, the machine makes sure the fork is in its rightmost position, then it waits until the start button is pressed. In this position the fork blocks the discs from falling out of the tube, so these can be loaded safely. As soon as that happens, it moves the fork back to the middle, so a disc can drop onto the plate. Depending on the detected colour, the fork then moves to the right or left until it hits the relevant limit switch, before moving back to the middle and re-starting the cycle. Whenever it takes too long for a limit switch to be hit, we go to the error state corresponding to the motor that should

be working.

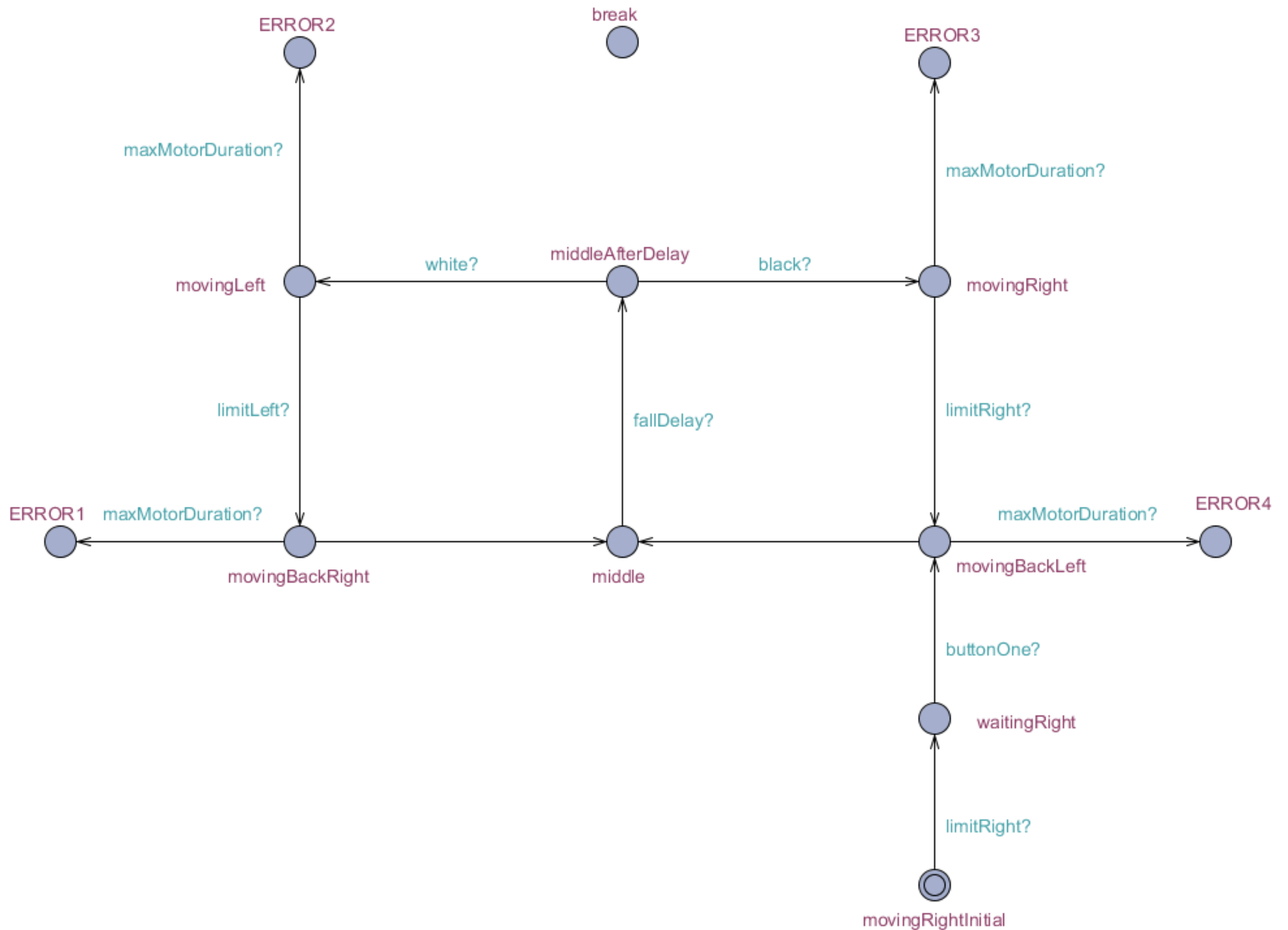

Figure 4: the UPPAAL model for our first prototype

4.6 Software implementation

For the simple PWM program, one main loop was used that sets the outputs that need to be powered based on the input, and a timer interrupt that then turns these outputs on and off at the right time to deliver a Pulse-Width Modulated output.

In the process, an attempt was made to set an interrupt for when the over-current protection is triggered, but even though the instructions from the documentation were followed, it did not work and instead made the program function incorrectly. Since it was not a vital part of the machine and the protection already has a visual way of letting the user know something went wrong, the feature was omitted.

The program's source code is shown below as pseudo-code. The original assembly code will be explained later on. The pseudo-code offers insight in how the PWM program functions.

PWMSTATE = false;

```
INTENSITY = 80;
ACTIVE = which lamps we want to use PWM on;

PWMwithinterrupt()
    EMERGENCY = emergency button state;
    SENSORANAL = analogue sensor value;
    if PWMSTATE
        lowstate()
    else
        highstate()
return

lowstate()
    timer += 100 - INTENSITY;
    PWMSTATE = false;
    output = off
    set timerinterrupt;
return

highstate()
    timer += INTENSITY;
    PWMSTATE = true;
    output = ACTIVE;
    set timerinterrupt;
return
```

5 Machine Specification

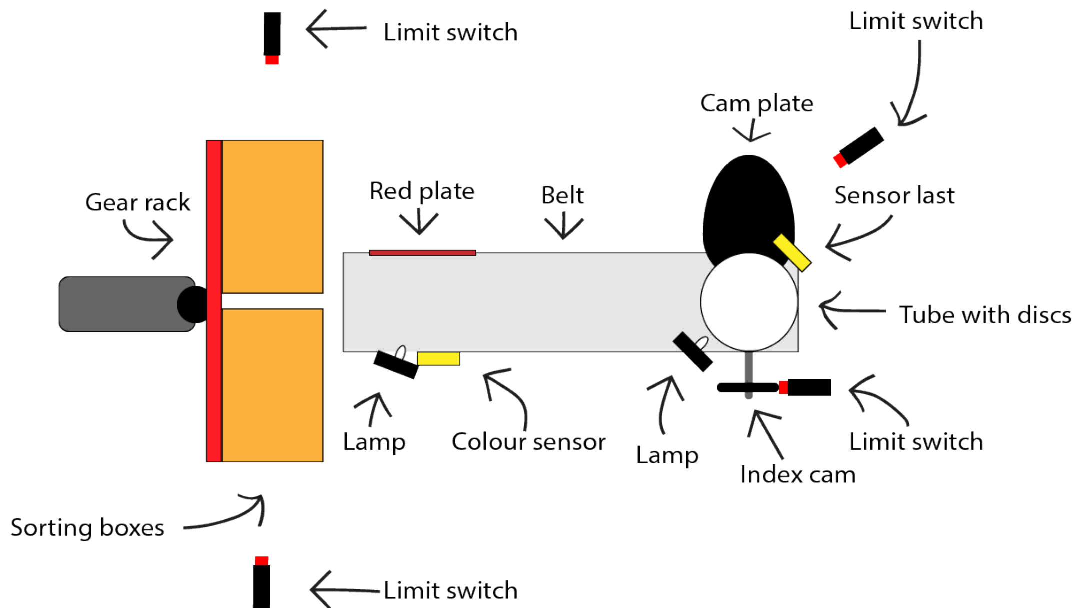

Figure 5: Sketch of the machine

For the final machine, some new specifications have been decided upon. Instead of focussing on speed, the focus has been shifted to fault detection. Experience with the first prototype has led to the conclusion that the machine should not be too compact, because that would make adding new sensors, systems and fault detection unnecessarily difficult.

For further specifications, the machine has been separated into three subsystems: a mechanism to extract the discs from the tube, one to transport the discs and check their colour and lastly, one to sort the discs. The main benefit of separating the machine into different parts is that it was now possible to work more efficiently, since it allowed three people to independently design and work on a single subsystem, instead of watching one person do all the work. This also simplified the design phase, as the subsystems can be designed independently of each other.

5.1 Fault Detection

The machine should be able to detect almost every mechanical or electrical error, assuming that only one part is malfunctioning, and see the difference between as many as possible. It can be assumed that only one part is malfunctioning at any given time because this is mentioned in the Project Guide.

One important requirement that has been decided upon was that between each of the three major parts of the system (ejecting the disc from the tube, transporting the disc, and getting the disc in the right box) the machine should detect if the disc is still there. In order to detect errors with the motors, at least one limit switch should be used for each motor to detect if it is still functioning properly.

5.2 Ejecting discs

The ejection mechanism should eject the discs from the tube one by one, so our machine has more control. When only one disc is moving in the machine at all time, the machine can conclude that when a disc is detected somewhere in the machine, it has to be the disc ejected earlier. Secondly, this part of the machine should check whether there are discs left that need to be sorted or not, so that the machine knows when it can stop and when there are still discs left to be sorted. This part of the machine or subsystem will later be referred to as Cam Plate later in the report, because that is the name of the egg-shaped FischerTechnik part that we will use for it.

5.3 Moving discs

Our machine should have a part in which the discs travel a sufficient distance. There are two main objectives that this system should accomplish. Firstly, it should move the discs from the ejection to the sorting mechanism. Secondly, during the movement of the discs, this system should be able to detect what colour the discs are and if there are even any discs passing by at all. An extra requirement is that there needs to be a way to know if the motor is working or not, so we can that to distinguish between errors.

5.4 Sorting discs

The sorting containers are the yellow boxes in which the discs need to end up; in our machine the discs need to end up in the right box according to the colour the PP2 received earlier through the sensors. Also, there needs to be a way to detect in which position this mechanism is, and it should even recognize if there is a failure with the motor, or one of the sensors. Besides that, it is important that there is no possibility that discs end up in the wrong container or neither and it is preferred that the system does not take up a lot of time.

6 Software Specification

The software should start in a resting state, just as the machine. In this state it should react to a 'signal', in our case a button press. After said button press the software should power the motor to eject discs from the tube, check the colour with a sensor, and move the motors that move boxes depending on that colour. During the transport of the discs, the motor for the conveyor belt should also be moving. Lastly, the software should go to a 'stop' state when there are no discs left. Whenever the emergency button is pressed, the machine should (immediately) come to a halt and return to the start state.
Next, there are some additional 'side' requirements our software should adhere to. It should be able to notice if a disc goes "missing" between one of the subsystems and it should notice if a sensor is disconnected, for as far as that is possible.
The motors from the cam shaft, the conveyor belt and the boxes have to be controlled with Pulse-Width Modulation, to make sure that the motors do not run too fast or too slow. Having them run too fast can do serious damage to the motors and will make the machine less predictable, because the discs can slide and get launched from the belt. Having them run too slow might lead to the motors not having enough force to move at all or systems not being ready on time. For example, if the motor for the sorting part moves slower than the transportation part, the discs will arrive at the sorting part and be sorted according to the colour of the previous disc. If you have a FischerTechnik part to restrict the movement of a disc to only a certain location based on its colour, but this part moves too slow, the disc will not be sorted properly.

As another restraint, lamps should only be turned on when they are needed, for example to check for discs moving by. Having them on at all times is generally bad design and also increases the chance that the overcurrent protection is triggered. Apart from that, it goes against the system requirements.

7 Machine Design

7.1 Cam Plate

For the extraction mechanism, a first consideration was having a belt move the discs from the tube. To test this a quick prototype was made and powered using a simple PWM program. This program is also mentioned in the software implementation. The results of this prototype were that the belt had too much leeway, and because of that, it jammed when extracting discs from the tube due to the inconsistency. Therefore this idea was discarded. After this three ideas were left:
Firstly, using a sort of claw, this idea was very crude and since the whole idea was too variable and complicated it was also discarded.
Secondly, we thought of using a fork to extract the discs. This idea was more or less constructed in our first prototype and from the results of testing it was concluded that the fork left too little space to put sensors.
Lastly, using a cam plate to extract the discs. The cam plate is an egg shaped part that rotates to push discs out of the tube. Ultimately this idea was chosen, because it would leave more room for sensors, and it would require only one limit switch to detect and limit its movement.

Since everything in the machine needed to be more spaced apart, a new design for ejection was required. According to the machine specification this part should be able to eject discs from the initial tube. As for the design of this part, a cam plate was chosen already, which will be powered by a motor. Since there is no direct way to power the cam plate, it needs to be attached to an axle with a gear attached to it. The motor can then power this gear and through that the cam plate.
It is also designed so that the machine can detect whether the cam plate is moving or not with a limit switch. This limit switch is spaced the ideal distance so that it only triggers once every rotation of the cam plate.
In this part of the machine the position of the tubes that contain the discs also needs to be determined. These two tubes need to be positioned so that they can possibly hold all the discs that we have at once. Below these two tubes there is a horizontal platform on which the discs will fall before the cam plate pushes them on the conveyor belt. Lastly, this part of the machine has to be able to detect if there are any discs left. This is also a requirement in the machine specification. To check this we will use a lamp and a light sensor.

7.2 Conveyor Belt

As mentioned in the section on Prototype 1 and the machine specification, the discs should cover some ground in order for the machine to have space for our sensors. For this, the possibility of using a conveyor belt, a fork, a claw and slopes were all considered and analysed. The problem with using a fork would be that one side of the path of the disc would be occupied by the fork, making it hard to place a sensor to see if discs are passing certain points. A claw was, as already mentioned earlier, really impractical since it was way too complicated, using too many motors and making error detection hard. A slope was also a possibility, but the results from our first prototype concluded that the machine is going to require a lot of FischerTechnik parts, so using this idea would deplete the supply too much. A belt would work fine, since it leaves plenty of room for other components (left, right and top) and is fairly reliable and doable, although it might not be as fast as some other systems. This design had the most advantages and the least disadvantages, so it was decided to go for the conveyor belt.

The sketch below indicates how the design was started. In this design a triangular shaped conveyor belt was used and the motor that powered the belt was inside this triangular shape. This design was not the design that was going to be implemented, because of two main reasons.
First of all, this implementation of the conveyor belt took too many building blocks of the FischerTechnik set. This is a problem because these blocks are needed for the other parts of the machine.
Secondly, the machine was not sturdy enough. Even though it was really easy to adjust every time on the spot, it is not handy to fix it after storing it away.

In this design two sensors were used instead of one, since the machine needs to detect if there are discs passing by and what colour they are. One sensor was used to detect discs passing by and one sensor to detect the colour of said disc.
In the next version of the transport belt the issue with the sturdiness was fixed and the conveyor belt design switched to a linear conveyor belt instead of the triangular one. One of the sensors was also removed (and one of the lamps) and the colour detecting sensor was left in place. The location of this sensor also changed from looking down on the conveyor belt to being at the side of the belt, looking at a red plate. This red plate was chosen so we can distinguish between a black, white and no disc. In the machine implementation there is a more elaborate explanation about this sensor.
The last thing that is mentioned in the machine specification about the conveyor belt, is that the machine needs to be able to detect if the motor is faulty. To make sure this can be possible the machine needs to make use of a limit switch that gets triggered if the belt is moving.

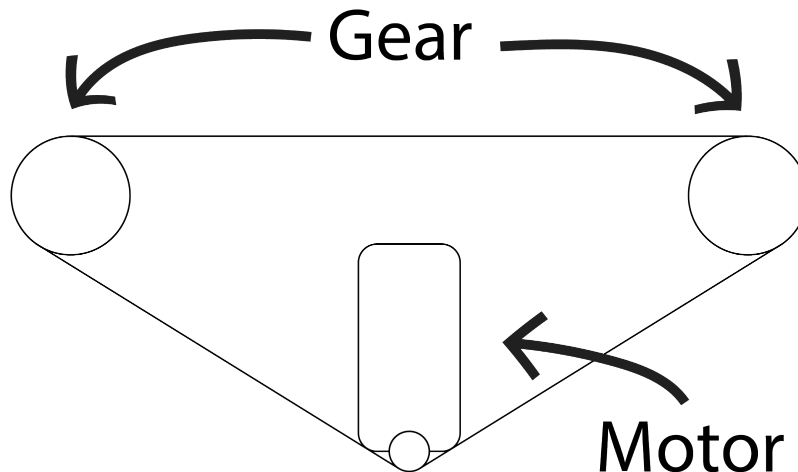

Figure 6: First design of the conveyor belt

7.3 Containers

The last mechanism of the machine is the part that actually sorts the discs. As mentioned already in the previous part, it does not have to detect whether it is are dealing with a black or white disc, since this is already checked in the previous part; everything that has to be done in this part is to divide the discs to different sections depending on the colour.
Various options regarding this were considered. Firstly, a wall that is able to move to either the left, in case of a white disc, or to the right, in case of a black disc. This wall then forces the movement of the disc and guides them to the appropriate container. Secondly, we considered using a second conveyor belt that moves to either side depending on the colour of the disc that falls on it. And thirdly, we have thought of containers that are able to move horizontally to the correct position with the aid of a motor. This last design was eventually chosen, because it seemed interesting and it could work without much issue, while not influencing the time it takes to sort.
Using this design, the discs immediately fall from the conveyor belt into the right container, because as soon as the colour is known the containers start moving in the necessary direction until the relevant limit switch is pressed.

We first designed the sorting mechanism as you can see in the picture below. As you can see, this design consisted of a very simple system. The boxes were simply attached to a gear rack and to each other.

The gear rack is then simply powered by a motor, which is attached to the PP2.

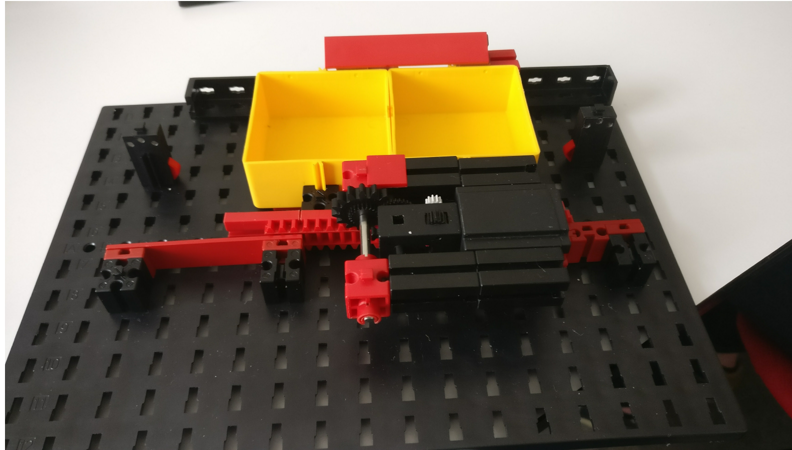

Figure 7: First design our sorting containers

The design did not really work out as the movement of the sorting containers was only restricted by physical FischerTechnik blocks placed on the board. This resulted in the machine getting stuck more often than wanted, and also made the motor work harder as there was friction between the surface of the board and the sorting containers to worry about.

After this a new design was being worked on, since the last one had a couple of flaws. The result can be read about in the machine implementation.

8 Software Design

To completely and accurately specify how the software should function for this machine, an automaton has been designed in UPPAAL. It is worth noting that this was not yet the final version.

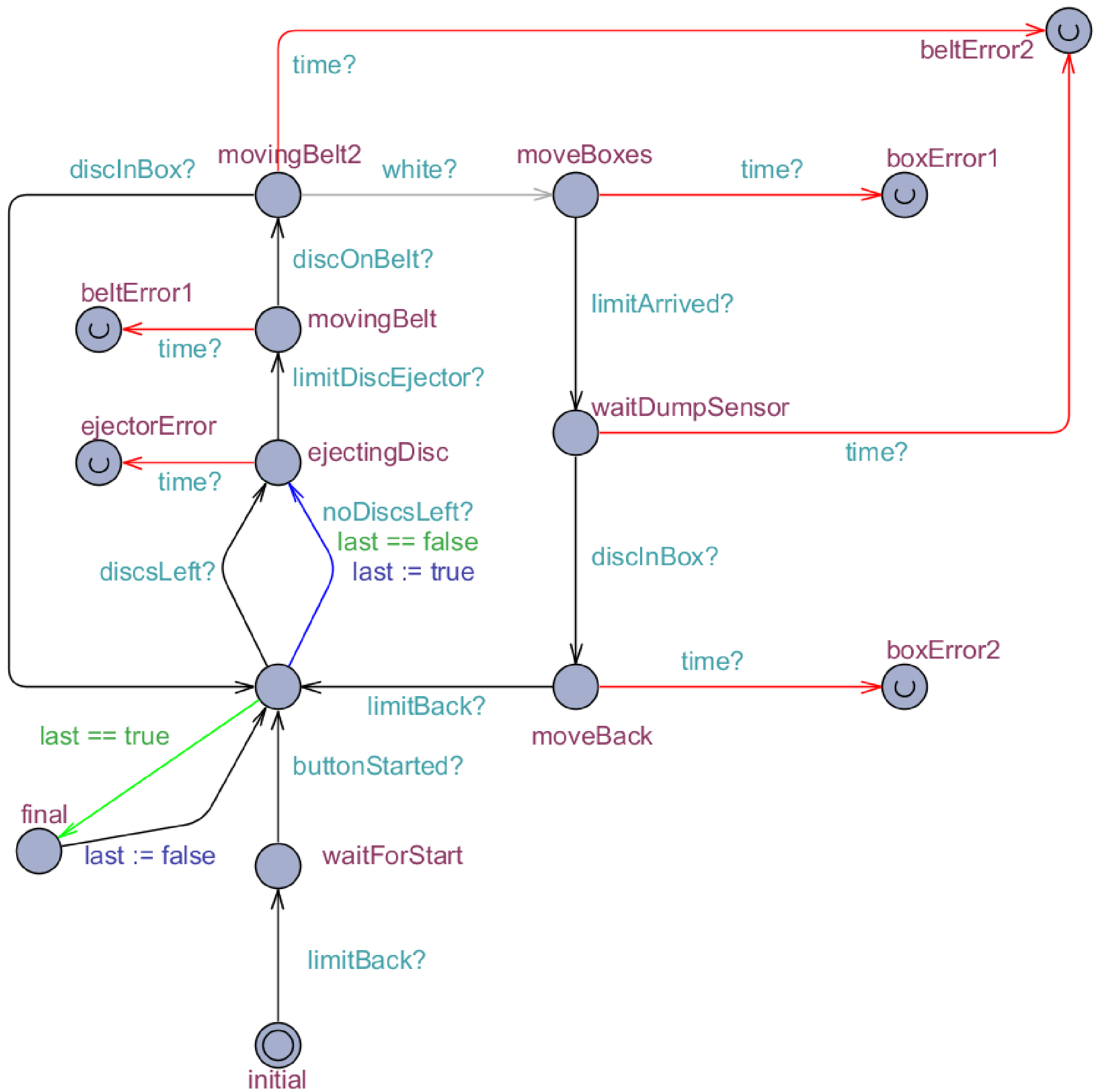

Figure 8: the second UPPAAL model

At the start of this program, the boxes should be in the correct position, such that when the boxes do not get moved the black discs fall into the right one. Then the machine waits for the start button to be pressed. Once it gets pressed, the program's main loop is entered: first, it has to check if there are discs left (see the subsection "Last disc"). Next, it has to eject the disc and wait for the relevant limit switch to get released

and pressed again; if this takes too long an error must have occurred. Once the disc has been ejected the conveyor belt starts moving, checking whether it passes the first pass sensor in time. Since a digital light sensor has been used in this design, it will either detect a white disc and move the containers accordingly, or detect the disc with the last pass sensor. In the latter case it was a black disc and landed in the right container and the loop can be started all over again.
In the other case, the containers will be moved until they touch the relevant limit switch to make sure white discs land in the other container. The boxes stay stationary until the last pass sensor has noticed the disc, after which they move back to their starting position and we go through the loop once more.

Later, the two "pass-sensors" have been removed and replaced with a more advanced analogue sensor. This led to the UPPAAL model below, in which error detection has been added too.

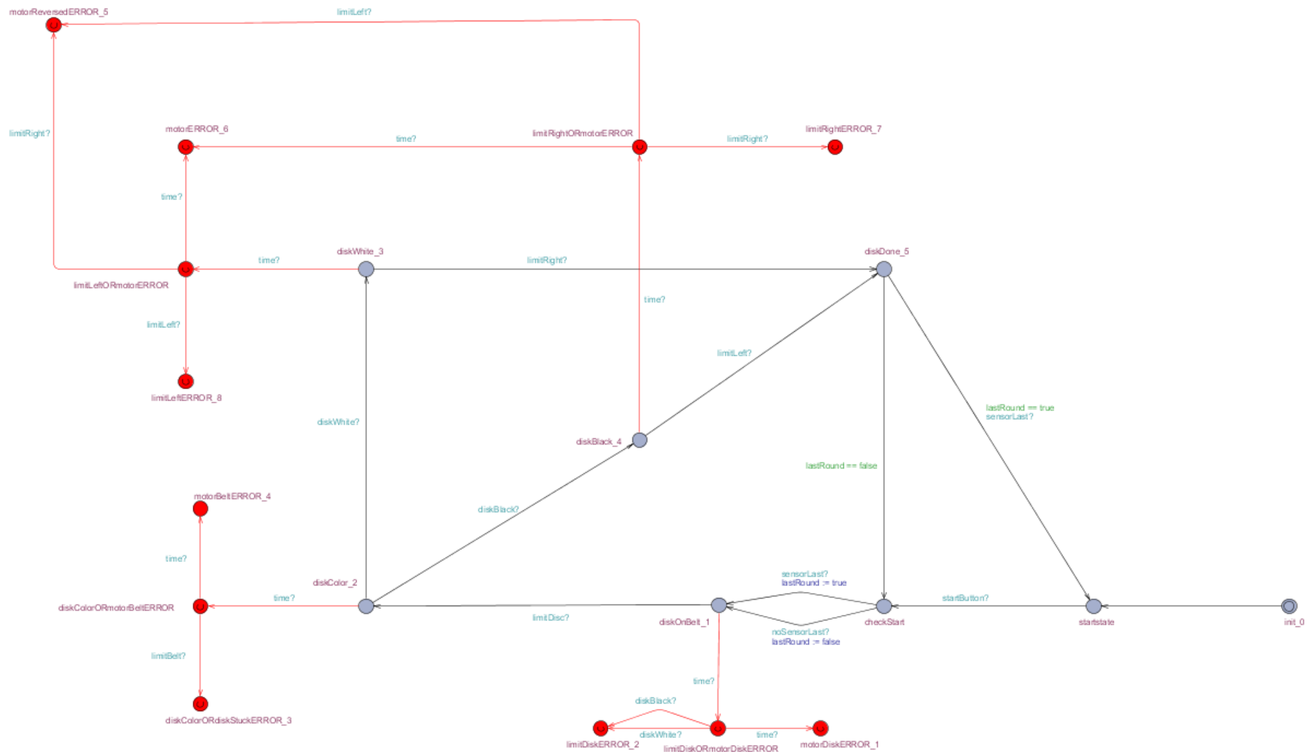

Figure 9: the third UPPAAL model

8.1 Last disc

To detect that the final disc has been ejected and to stop trying to sort any more discs, the cam plate subsystem has a sensor that can detect whether there are still discs left. This sensor is not able to detect the bottommost disc, but it can see whether there is a disc present above the bottommost disc. Therefore, once it does not detect a second last disc, it knows to run the sorting procedure one more time. To accomplish the above-mentioned, a variable is stored for whether or not the machine is in the last round of sorting. This way, the program will still run one more time if it does not detect a disc with the sensor, but quit after sorting this last disc.

8.2 Emergency button

The first plan for the emergency button was to add the code to handle it to the same interrupt as the Pulse-Width Modulation. It would check the emergency button for a press and then move into an emergency state to wait for the start button. Getting this to work would save us a lot of effort and keep our code more simple and clean. Sadly, simple testing showed that staying in the timer interrupt for extended amounts of time would disconnect the PP2 from the computer. This is not a huge problem in itself, but it also does not allow specifying where it returns to when the emergency has been taken care of.

For this reason, using a subroutine instead has been settled upon. It needs to be called in every state to make sure the machine can always be stopped, which leads to some slightly longer code, as expected. Having it as a subroutine is, on the other hand, shorter than having the code there directly and also gives an easy way to add more code to always execute and to modify the emergency button's code. From here it is possible to branch to any state without much issue, provided that the position RTS would have sent us to is pulled from the stack. This is necessary because for any BRS instruction an RTS is expected, but the behaviour can be mimicked by pulling the line number from the stack. If this is not done repeatedly using the emergency button would fill up the stack and function as a memory leak.

8.3 Error detection

In order to be able to detect most errors, a way is needed to check if too much time has passed and the next sensor has not detected something yet. In order to detect if sufficient time has passed, a counter has been created. Every time the timer interrupt runs, it updates this counter. Then this counter is used in various states in our program in order to detect if a motor or disc is taking too long to finish its current business and therefore it can be known when there is something wrong, so the machine can go to an error state.

9 Machine Implementation

Although the design and specifications of the second prototype was ready, the actual making of the machine proved to be rather challenging. A number of issues arose on all 3 of the parts mentioned in the Machine Specification. The subsequent subsections contain explanations on why and how the implementation of these parts was difficult.

9.1 Cam Plate

In this section the first part of the machine is discussed, namely the cam plate. Before the final implementation of this part it will have been numerous times. The first time of building it, the focus was on having it actually build to test if this was the best option. It was not sturdy at all and lacked some compactness (not that this was necessary). After testing it with the code written for it, it resulted in the expected behaviour, and that this was indeed the best approach for the first part of our machine. The part was then changed a bit to improve the sturdiness and make it more compact. The second time, a sensor was added to detect whether or not the machine was finished. This was not really as sturdy as necessary, but it contained every function which was expected from the machine. The next time the whole subsystem was rebuilt, compactness and sturdiness where key points. This part, together with the conveyor belt, is in a vertically higher position since the discs need to fall in the containers at the end. Therefore the whole construction needed to not only be sturdy and compact, but also balanced. The finished implementation of this can be seen in the following picture.

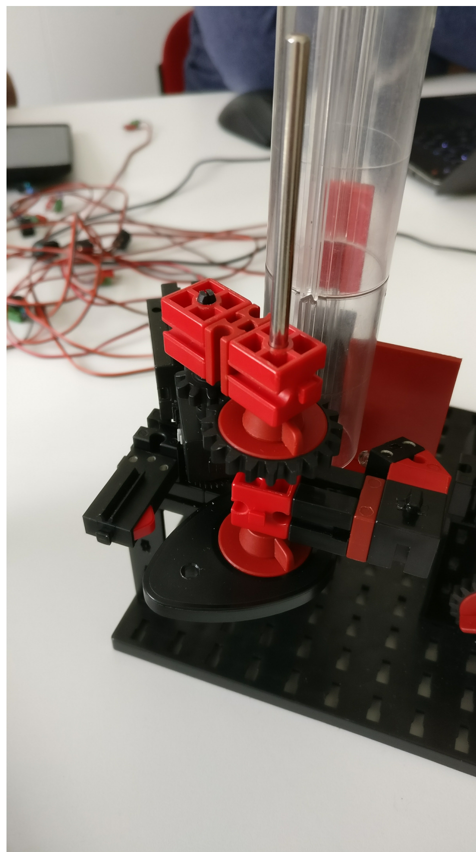

Figure 10: The cam plate

After this only very small changes were made. The metal axle was swapped with a smaller one, since the longer axle was needed for the containers.
The last thing that is mentioned in the machine design is the use of sensors to detect discs in the tube. For this the sensor and lamp were placed on opposite sides of the tube so the light of the lamp shines through the tube and either against a disc or through the air in the tube (in case of no disc) and finally hits the sensor.

9.2 Conveyor Belt

The implementation of the conveyor belt has been a process of building it and breaking it down again or changing it a lot. As already covered in the machine design of the conveyor belt, first the belt was mounted in the form of a triangle. This caused issues with placing the motor in between it. A lot of extra parts where used around the conveyor belt since we had to support the motor in that position.
The motor is rotating a gear on an axle that was connected to one of the gears, and that was in turn attached to the conveyor belt. As specified in the machine specification of the conveyor belt, we want to know if the motor of the belt works or not. In machine design it is stated that a limit switch is going to be used for this. A flat disc that has one half extending further out than the other half, called an ”index cam” by the Fischertechnik guide, is attached to the axle. This index cam presses against the limit switch to detect the movement of the conveyor belt. To save space and parts this construction is put on the side of the conveyor belt opposite to where the motor is.

9.3 Sensors

To allow for more error detection, the sensor which we use to determine the colour of the discs was connected to an analogue input. By using a simple program which prints the analogue sensor’s input values on the display, it was found that white discs give a value below 100, the black discs give a value between 100 and 180, and if there is no disc at all it gives a value above 180.
The advantage of using the analogue values is that the machine could now also detect when a black disc has passed the sensor, in contrast to an earlier prototype where the machine could not distinguish between a black disc or no disc at all. Some additional tests were conducted regarding the situation when the cable would be plugged in wrongly. If the cable is disconnected from the PP2, the input is a value of 3 or 4. If the cable is plugged in the other way around on the PP2, the input is 0. If the sensor is completely blocked, its polarity is reversed, or the cable is disconnected from the sensor, the input is 255. These values are then used to detect these situations and let the machine go to an error state.
After implementing this, another problem occurred: the light values for the black, white and no discs were fluctuating a little, and were dependent on belt speed and background lighting. This was solved by moving the light sensor to the side instead of the top, and placing a dark red plate behind it. The white discs always had a value of less than 50, and the black ones always had a value over 170. The plate (no disc) was approximately 100.

9.4 Containers

As mentioned in machine design there are several designs, and from these designs there exists a way so that the boxes are restricted in their movement, but not too much as this will be detrimental to the operation of the machine.
The result of this is the following implementation, in which the sorting containers movement is restricted by two metal rods. As can be seen in the picture below, the sorting containers are attached to and resting on top of the metal rods. In the first redesign of the sorting container part, blocks were used, which had a too narrow hole for the rods to go through, but this was easily resolved by using blocks with a wider opening. The motor still moves the boxes through the use of a gear on a gear rack mounted on the side.

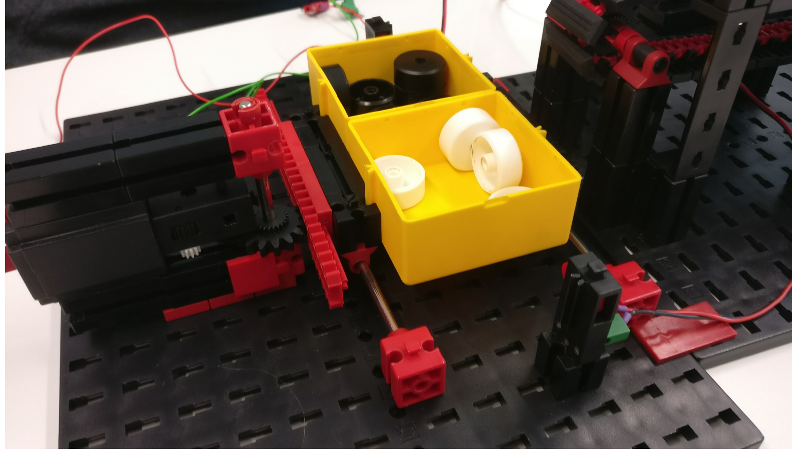

Figure 11: Final implementation of the boxes

9.5 Overcurrent protection

There was trouble with the overcurrent protection; it triggered with only two lamps and one motor connected to the same group. In response to that, a few tests were conducted using a multimeter. From these test it was found that the voltage drop over the lamps was 7V and that the current draw was 130 mA, when connected to an 11V output using the 33 ohm resistor. With the circuit below (in which the resistors represent the lamps) it was possible to connect all four lamps to one output of our PP2, with a total current draw of a little over 200mA.
Using this method of connecting the lamps, there was no longer any need to use wires with 33 ohm resistors in them, saving some power. There were only two disadvantages: when one lamp was connected wrongly or broke down, two lamps would get turned off and if the connector to the output would be disconnected, all four lamps would get turned off.
The final product does not use this method, since the current draw was too high on a single output. Instead, it was decided to avoid unnecessary activation of lights and reduce the amount of lamps and light sensors. This method has been useful for testing one of the first versions of the code, because one could simply connect this to the always on 12V port instead of one of the output ports, completely avoiding the overcurrent protection altogether.

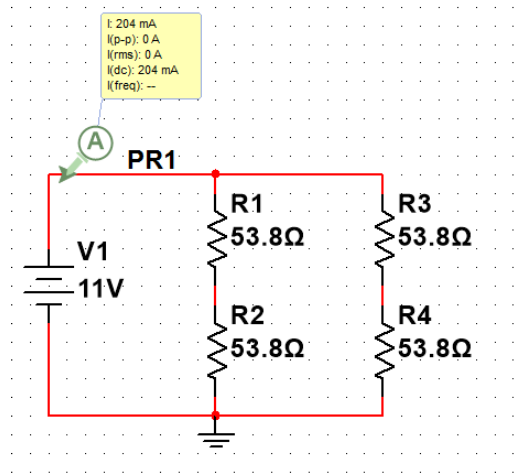

Figure 12: Schematic of our lamps

10 Software Implementation

10.1 Walk through

From the Software Specification it should be clear what it is the PP2 should do. Software Design explained how the code to be was envisioned and how UPPAAL was used to designed it. Next up is a walk through of the final implementation of the software based on those two sections; what each subroutine does and under which circumstances each is called.

First of all, the pulse-Width modulating part of the code. This is a part that is constantly running and has a simple function, namely to provide PWM to the motors so they do not burn down.
The general idea behind the PWM part of the code is that whenever a timer-interrupt happens the necessary outputs get turned on in the high state. The next time the interrupt comes it will enter the low state, setting the corresponding outputs and doing the corresponding calculations.

The code for the timer interrupt is shown below. The first thing it does is increase the step counter. It is necessary to manually keep track of the time because the interrupt already uses the physical timer. This works and is still reliable since the timer interrupt gets called exactly twice per 100 timer steps.
Next, it checks if the emergency button is pressed. That is done here and now, because this part of the code is run very often. This makes the emergency button press independent of the state the machine is in at the time of running this code, which is very useful.
The last thing this piece of code does is load the values of the AD-converter, format it to the correct value and store it again so it can easily be used elsewhere in the code. This makes the use of this value less confusing and keeps the rest of the code cleaner. The code is then redirected to either the high state or the low state, both mentioned earlier in the walk through.

```
timerinterrupt :
    LOAD    R0  [GB+COUNTER]        ; Increase the step counter
    ADD     R0  1
    STOR    R0  [GB+COUNTER]

    LOAD    R0  [R5+INPUT]
    AND     R0  EMERGENCY           ; Check whether the emergency button is pressed
    STOR    R0  [GB+EMERGENCY]

    LOAD    R0  [R5+ADCONVS]        ; Load the AD-converter and store its value in the memory
    AND     R0  COLORDISK
    STOR    R0  [GB+SENSORANAL]
    LOAD    R0  [GB+PWMSTATE]
    BEQ     highstate               ; PWMSTATE of 1 will go to lowstate and 0 will go to ↩
        highstate
```

The code in the high state is very simple. The first thing it does is load the INTENSITY from memory, which is a value with a scale of 0 to 100 determining how long the pulse should be. That value is then simply added to the timer, since every low and high state together is 100 timer steps. Next, the code stores 1 in PWMSTATE, so the machine will go to the low state the next time the interrupt goes off. Before closing off, the outputs defined elsewhere in the code are turned on. Lastly, the interrupt register gets reset and the code returns from the interrupt.

```
highstate:
    LOAD    R0  [GB+INTENSITY]      ; load initial time of the PWM
    STOR    R0  [R5+TIMER]
    LOAD    R0  1
    STOR    R0  [GB+PWMSTATE]
    LOAD    R0  [GB+ACTIVE]         ; load active ports
    OR      R0  [GB+PERMANENT]
    STOR    R0  [R5+OUTPUT]         ; active loaded ports
    SETI    NUMTIM
```

```
    RTE
```

The code for the low state is largely the same. There are only two differences; one is that it sets the PWMSTATE to 0, to make sure the machine goes to the high state next time. Another difference is that the amount added to the timer is the difference of the total - highstate = lowstate, where total is the total time spent in the high state of the interrupt plus the low state of the interrupt.

```
lowstate:
    LOAD    R0  DELTATIME           ; load remaining time of our PWM
    SUB     R0  [GB+INTENSITY]
    STOR    R0  [R5+TIMER]          ; bump timer by DELTATIME
    LOAD    R0  0;
    STOR    R0  [GB+PWMSTATE]       ; set variable so that it changes PWMstate
    LOAD    R0  [GB+PERMANENT]
    STOR    R0  [R5+OUTPUT]
    SETI    NUMTIM
    RTE
```

Next up are two other pieces of code which can be used and seen as independent of the machine, while still having a very important role.
The first one of these is the emergency subroutine. The subroutine starts with immediately stopping all motors and lamps as per the System Level Requirements. Next it loads the ERRORSTATE, converts it to seven segment display using a subroutine that will be explained later, and outputs it on the display. Once the start button is pressed everything should be resolved, so the program pulls the line number put on the stack by *BRS* and returns to the start state. As long as the start button has not been pressed this script will keep running.

```
emergstate:
    LOAD    R0  0
    STOR    R0  [GB+ACTIVE]         ; when in emergency mode everything is off
    STOR    R0  [GB+PERMANENT]
    STOR    R0  [R5+OUTPUT]

    LOAD    R0  [GB+ERRORSTATE]
    BRS     Hex7Seg                 ; translate (value in) R0 into a display pattern
    STOR    R1  [R5+DSPSEG]         ; and place this in the Display Element
    LOAD    R1  %01000              ; R1 := the bitpattern identifying Digit 3
    STOR    R1  [R5+DSPDIG]         ; activate Display Element nr. 3

    LOAD    R0  [R5+INPUT]
    AND     R0  STARTBUTTON
    BEQ     emergstate

    PULL    R0                      ; to fix the wrong stack pointer
    BRA     startstate
```

As seen above, the second separate subroutine to be looked at is Hex7Seg. This routine fills R1 with the seven-segment value of R0.
DISCLAIMER: This part, and this part alone, has been provided by the Technical University of Eindhoven. It is listed for the sake of completion.

```
Hex7Seg     :
    BRS     Hex7Seg_bgn             ; push address(tbl) onto stack and proceed at "bgn"
Hex7Seg_tbl :
    CONS    %01111110               ; 7-segment pattern for '0'
    CONS    %00110000               ; 7-segment pattern for '1'
    CONS    %01101101               ; 7-segment pattern for '2'
    CONS    %01111001               ; 7-segment pattern for '3'
    CONS    %00110011               ; 7-segment pattern for '4'
    CONS    %01011011               ; 7-segment pattern for '5'
    CONS    %01011111               ; 7-segment pattern for '6'
    CONS    %01110000               ; 7-segment pattern for '7'
    CONS    %01111111               ; 7-segment pattern for '8'
```

```
    CONS     %01111011                ; 7-segment pattern for '9'
    CONS     %01110111                ;  7-segment pattern for 'A'
    CONS     %00011111                ;  7-segment pattern for 'b'
    CONS     %01001110                ;  7-segment pattern for 'C'
    CONS     %00111101                ;  7-segment pattern for 'd'
    CONS     %01001111                ;  7-segment pattern for 'E'
    CONS     %01000111                ;  7-segment pattern for 'F'
Hex7Seg_bgn:
    MOD R0   16
    LOAD     R1   [SP++]              ; R1 := address(tbl) (retrieve from stack)
    LOAD     R1   [R1+R0]             ; R1 := tbl[R0]
    RTS
```

Now that the core of the program has been covered, the rest of the code is next, represented by the UPPAAL model below. (Figure 11) This part will use the model instead of the actual code to show what the code is meant to achieve without having to go into the details.

The automaton starts in the initial state, marked with the text $init_0$ in the bottom-right. Here the machine gets ready for operation. It loads certain values in memory and installs the timer interrupt routine.
Then it goes to the first state, which needs manual intervention. This is the *startstate*, in which the machine simply waits for the user to press the start button, as described in the project guide.
The next state is the *checkstart*. This state simply checks if there is a disc in the tube. If there is no disc in the tube the program runs one more time, because that means that last loaded disc still needs to be sorted, else it goes to the *disconbelt*.
Now in *disconbelt*, the machine operates the cam plate while detecting for possible errors. This is described in the machine design and implementation.
The next state, *disccolor*, is responsible for detecting the colour of the discs. It is also responsible for detecting the motor errors, since the disc would not reach the sensor if the motor does not work.
discwhite and *discblack* are the different states *disccolour* points to if the discs are white and black respectively. In these states the sorting containers are moved horizontally as described in machine design.
These two states lead to the last state *discdone*, where the machine has the containers in the correct place. Here the machine just waits until the disc on the belt actually dropped down in the box. When the disc is sorted, the machine goes back to *checkstart*.
This is the procedure the machine follows. The machine keeps working until the condition in *checkstart* is met.

Figure 13: UPPAAL model of the code

10.2 Branch problem

There was a weird problem, in which upon adding a certain amount of code, even code like "ADD R0 0", the debugger would give error 33. This error means that the stack has overflown. The step by step function of the debugger made clear that for some reason there was a difference between the hexadecimal file and the code we saw in the debugger. The debugger always branched one line above the one where it should, causing it to enter a branch loop, which in turn causes the stack to overflow. This can be seen in the two pictures below, the left side is the debugger, and the right side is the hex file, with the two lines executed in the loop marked with "—".

Figure 14: Part of the code with a wrong branch statement

Figure 15: Part of the code which was branched too wrongly

As can be seen in the top picture, "BRS checkemerg" got converted to "BRS 14f". Line 14f, however, corresponds to "BRA disconbelt" instead of the checkemerg label. For some reason, the debugger branches one line too high. The initial conclusion was that it was caused by the debugger, but after looking at the hexadecimal file more closely, it became clear that the hexadecimal code did not correspond to the comments next to it. Adding the meaningless instruction "ADD R0 0" above every label worked as a workaround, but those instructions also made the code a lot less clear and clean. This prompted us to look at the hexadecimal codes in the .hex file, which revealed more about the underlying problem: almost all branch instructions had the wrong amount of displacement. The last two hexadecimal characters should be the difference between the line numbers of the branch and the label to branch to, minus one. As it turns out this value is one less than it should be for some of the branches. Adding one to the displacement in the .hex file fixed the issue, confirming this theory. This means the issue had to be in Assembler9.jar.

After some digging, it became clear what the actual issue was, why it occurred and how to avoid it: the first instruction (which sets the timer interrupt to the right label) needed a long form instruction (because the displacement from the label did not fit in eight bits). Long form instructions take up two lines of code in the resulting .hex file and for some reason the Assembler did not account for this correctly, even though it seems to handle it correctly in other parts of the code. In order to find out more, we created a simple program with a lot of meaningless instructions to take up space, and force long form instructions. As it turns out, the assembler only handles long form instructions with labels incorrectly if they are LOAD instructions. This error occurred in the program because its first line read "LOAD R0 timerinterrupt", referring to the "timerinterrupt" label at the end of the code. An easy way to avoid this is to make sure the timer interrupt is not too far down into the program, so the instruction only takes up one word. This also explains why adding a single meaningless instruction could make or break the code, as that changes the displacement and can make it no longer fit in eight bits.

11 Testing

The following section gives an overview of all the tests conducted on the machine and its parts. As suggested by the Study Guide, the box was checked to make sure all FischerTechnik parts were present before doing any tests. To avoid confusion beforehand, following are some definitions: with 'component' the individual FischerTechnik parts are meant that interact with the PP2, for example, motors, lights and sensors. With 'unit', however, the subsystems of the machine are meant. These have been mentioned already in the machine specification and include the cam plate, conveyor belt and sorting containers.

11.1 Component Tests

The component tests were started with a simple Pulse-Width Modulation script. This script is extensively explained in the Software Implementation. The version used here is the same as the one used to test the first prototype and mimicked the input to the outputs. The only thing that needed to be tested for the PWM was the percentage width of the pulse.
According to the Technical Guide from this project we know that the motors can take a maximum of 9 volts. We also know from the Guide on the PP2 that the outputs of the processor supply 11 volts. Therefore the width of the pulse in our PWM program should be $\frac{9}{11} \cdot 100\% \approx 82\%$. This way the motors can be safely tested without the risk of damaging them.
In this phase all the lamps and limit switches have been tested. This is pretty straight forward as the lamps and limit switches work without any configuring and are thus easily tested. Just attaching them to the output with use of a cable with a resistor suffices.
Next, the light-dependent resistor has been tested. The digital input was very straight forward, like the limit switches. The analogue input, however, required a bit more work. The PWM program was modified to also display the value of the analogue input on the display. This analogue sensor was then tested by using a light close up to the sensor and also turning the lights in the room on and off to check if the values it returned were sufficiently accurate.
All this testing was done on the PP2 and it behaved as expected, so the PP2 has also been tested for the options needed.

11.2 Unit Test

Next on the V-model there is the Unit Test, where the different units are tested. The three main units of the sorting machine, as written about in previous parts of the V-model, are the cam plate, the conveyor belt, and the sorting containers.
To test the disc extraction system, or cam plate, a simple script was made which ejects one disc when a button is pushed. This script was based on the PWM program and had a small extension to incorporate aforementioned behaviour.
To test the conveyor belt, it is disconnected from the motors and then moved by hand to assess the friction it creates. If the conveyor belt does not move with little force, then check why the belt does not move as desired. Repeat this while the belt is not nearly frictionless. After this the belt is reconnected to the motor and the PWM program is run on the conveyor belt. Now the belt is easily inspected visually and if the belt moves at a steady pace, it is working as expected.
Lastly, the sorting containers were tested. The same procedure is used as for the conveyor belt; the only difference is that a special version of the test program is used that can move the boxes in two directions.

11.3 System Test

The UPPAAL model was used to run multiple system tests, for which the simulator in UPPAAL has been used. The machine has been simulated by clicking on the enabled transitions which correspond to a sensor, button or the timer. It has also been checked if all states in the model could be reached, which was indeed possible. After this, multiple scenarios were checked:

1. Normal operation of the machine with one disc: functioned properly.
2. Normal operation of the machine with 5 discs, white, white, black, white, black: functioned properly.
3. Operation of the machine with no discs inserted: machine would try to extract a disc and it would give the error that a disc is stuck.
4. Operation of the machine with a faulty motor for the disc extraction mechanism: the machine would give the error that the motor for the disc extraction mechanism is malfunctioning.
5. Operation of the machine with a fault limit switch for the disc extraction mechanism: the machine would give the error that the limit switch for the disc extraction mechanism is malfunctioning.
6. Operation of the machine with a faulty motor for the belt: the machine would give the error that the belt is malfunctioning.
7. Operation of the machine with a faulty limit switch for the belt: the machine would function properly and not notice the problem.
8. Operation of the machine where the disc gets stuck between the extraction and colour sensing mechanism: the machine would give the error that the disc is stuck.
9. Operation of the machine with a faulty motor for the sorting boxes: the machine functions properly when the boxes are already in the right place, so it does not have to move the boxes. It gives the error that the motor for the boxes is malfunctioning when it has to move the boxes.
10. Operation of the machine with a motor for the sorting boxes which has it's polarity reversed: the machine gives the error that the polarity of the sorting boxes is reversed.
11. Operation of the machine with a faulty lamp or sensor for the mechanism which checks if the machine is finished sorting: the machine keeps on going and gives the error that a disc is stuck.
12. Operation of the machine with a colour sensor stuck on white: the machine sorts as if there is a white disc.
13. Operation of the machine with a colour sensor stuck on black: the machine sorts as if there is a black disc.
14. Operation of the machine with a colour sensor stuck on no discs: the machine would give the error that a disc is stuck.
15. Operation of the machine with the left limit switch for the sorting boxes malfunctioning: the machine gives an error that the left limit switch is malfunctioning.
16. Operation of the machine with the right limit switch for the sorting boxes malfunctioning: the machine gives an error that the right limit switch is malfunctioning.

11.4 Validation Tests

For validation tests, the most important measurements are those made to assess the speed of the machine and calculate the standard deviation. To do this, the machine was loaded with 10 discs in an order that would bring about either the best or the worst case scenario. For the best case the machine was loaded with 5 white discs, and then with 5 black discs. This means that the least amount of movement is needed, and this is thus the best case. The test was performed 5 times as can be seen in the results in Table 1. It must be noted that out off the 50 total discs sorted all of them were sorted correctly. This is also a good indication of the reliability of the product.

Test no.	1	2	3	4	5
Time in s.	20.76	20.82	20.89	21.17	20.54

Table 1: The 5 best case results

This results in an average of 20.836 seconds with a standard deviation of 0.2281.

The worst case scenario is tested in the same way, but now the discs are placed in an alternating order with regards to the colour. Once again 10 discs were used each time and tested 5 times. The result can be seen in Table 2.

Test no.	1	2	3	4	5
Time in s.	22.83	23.69	23.41 s	23.49	22.76

Table 2: The 5 worst case results

This results in an average of 23.236 seconds with a standard deviation of 0.41603.

Now, as can been seen, the machine does perform slightly worse in the worst case scenario, but the difference in performance is only 10.3%.

After the speed measurements, the machine was tested with regard to the light sensors and how they function in different ambient lighting conditions. As mentioned in the component test, the values for the analogue sensor can differ based on these light conditions. Therefore, the machine in its completeness underwent multiple tests with the lights on and off to make sure the chosen thresholds would always work. The test results showed that the machine continues to work both in complete darkness and full light without changing the threshold values and thus passed these tests flawlessly.

Objects that were not supposed to be there were also inserted in the machine. In these cases the machine gave the correct error code.

12 Conclusion

During this quartile, we as a group have learned a lot about working as a group. We have put lots of time and effort in this project. Almost every week we sat together for the full length of our meetings to work and check each other's work. We practised skills needed for working in a group of people with different opinions. Some times we needed to just make a decision because we could not agree with each other. We all have gained useful knowledge and motivated each other.
Furthermore, we learned a lot of general project planning and documentation skills: on holding useful meetings, maintaining minutes on these meetings, planning out these meetings and projects with help from design processes such as the V-model, writing documentation with utilising minutes and looking at the (Belbin) roles people assume in a project.

13 Literature Overview

(1.) Dr. Meredith Belbin, Belbin Team Roles, Belbin, 2017,
http://www.belbin.com/about/belbin-team-roles/

14 Appendix

Included below are the Debug Manual (or Error Manual), the final version of the UPPAAL model, the final version of the source code and a debug program *min_max_brightness*. After that you can also find the logbook.

14.1 Error Manual or Debug Manual

Debug manual

Below follow all error identifiers with their respective issue descriptions and explanation how to resolve the issue. These identifiers are visible on the segment display of the PP2 processor. The identifiers for the errors are displayed on the third display. However, keep in mind that the identifiers for states are visible on the first display. Furthermore, we define the right side of the machine when looking from the container part towards the cam plate. The left side is trivially the other side. Whenever we say that a motor is disconnected we imply that the motor can be physically blocked as there is no difference to the PP2.

Errors

Below you can see a list of detectable errors and what you should do in that specific error to resolve the issue. Besides that, make sure the conveyer belt is empty, and in case the last disc has been sorted, remove it before resuming operation. Press the start button once everything is solved and if there are still discs left to be sorted.

ID	Issue	Fix
0	The machine is stopped by the emergency button	Resolve the problem.
1	The motor for the cam plate is not functioning correctly.	Check the wires of the motor and the connection on both ends, also check that there is no disc getting stuck in the tube.
2	The limit switch of the cam plate is not connected.	Check the wires of this limit switch and the connection on both ends. Now you have to get the cam plate in the correct position. This is easily done by pressing the start button once.
3	An unidentified object was detected	Remove the object from the machine and press the start button to let the machine resume action.
4	The motor for the belt is not connected.	Check the wires of this motor and check the connection on both ends.
5	The polarity of the motor of the container part is reversed.	Reverse the polarity of the motor by switching the red and black wire connection.
6	The motor of the container part is not connected or the container is blocked	Check the wires of the motor and the connection on both ends. Also check the switches on both ends.
7	The limit switch from the container part on the right side of the machine	Check the wires of this limit switch and the connection on both ends.

14.2 UPPAAL model

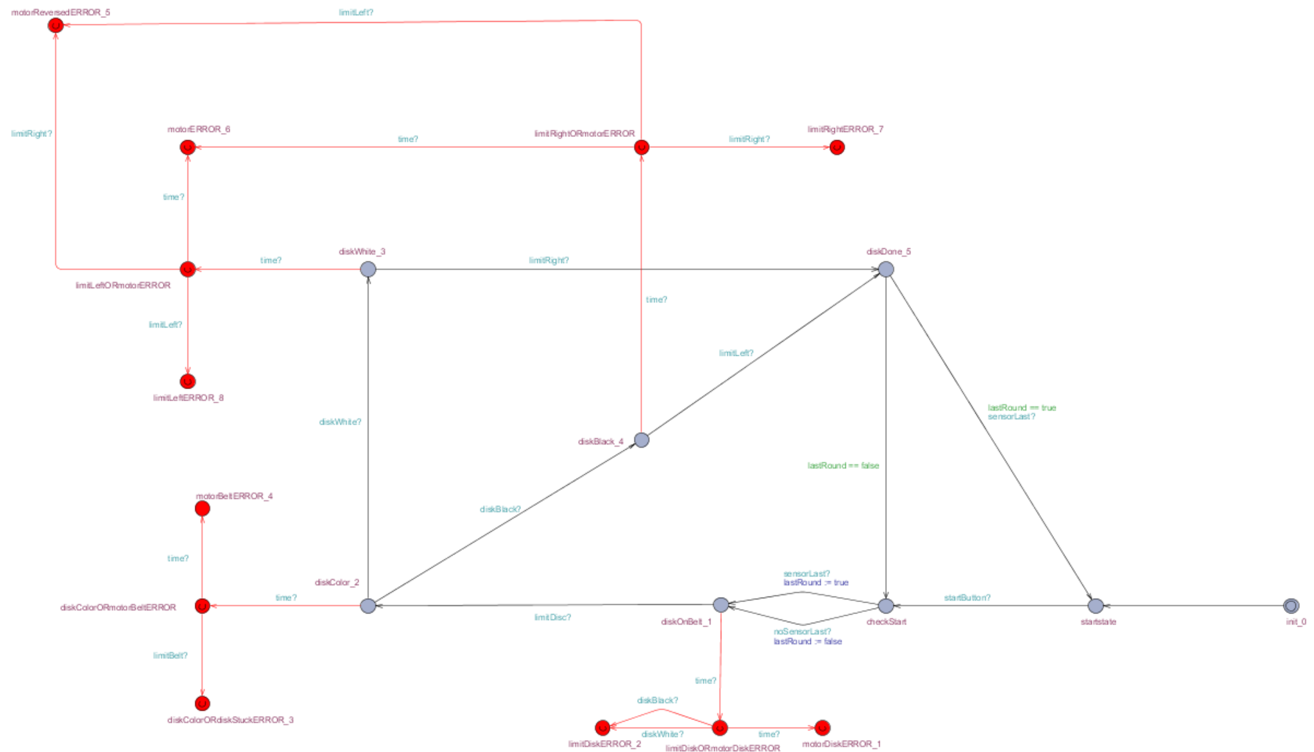

Figure 16: Final UPPAAL model

14.3 Source code

```
; first 3 errorstates are for overload
@DATA
    ACTIVE       DS      1           ; Contains what outputs will be active
    PERMANENT    DS      1           ; Contains what outputs will be active permanently
    ERRORSTATE   DS      1           ; Contains the state of the overload
    PWMSTATE     DS      1           ; Contains the state of the current PWM phase
    INTENSITY    DS      1           ; Contains the intensity of the leds
    LASTROUND    DS      1           ; Is 0 if not, else it is 1
    SENSORANAL   DS      1           ; Contains the value of the analogue ports
    COUNTER      DS      1           ; Holds the counter value
    CORRUPT      DS      1           ;
    BELTSTATUS   DS      1
@CODE
    IOAREA       EQU     -16         ; address of the I/O-Area, modulo 2^18
    INPUT        EQU     7           ; position of the input buttons (relative to IOAREA)
    OUTPUT       EQU     11          ; relative position of the power outputs
    DELTATIME    EQU     100         ; our time unit
    LEDS         EQU     10          ; the 3 LEDs above the 3 slide switches
    DSPDIG       EQU     9           ; relative position of the 7-segment display's digit ↩
        selector
    DSPSEG       EQU     8           ; relative position of the 7-segment display's segments
    ADCONVS      EQU     6           ; the outputs, concatenated, of the 2 A/D-converters

    TIMER        EQU     13          ; timer register
    TMRINT       EQU     16          ; place of the timer interrupt register
    NUMTIM       EQU     8           ; number of the timer interrupt
```

```
    WAITTIME    EQU     1000        ; Amount of cycles needed to before updating the display
    DISKWAIT    EQU     100         ; Amount of interrupt steps we wait before ejecting a new ↩
        disc
    WAITLAMP    EQU     30          ; Amount of time needed to turn on the lamp
    EJECTWAIT   EQU     200         ; Amount of time we can stay in the disc eject state before ↩
        we go to an error state
    BOXWAIT     EQU     100
    BELTWAIT    EQU     300

                                    ; The position of:
    STARTBUTTON EQU     %01         ; the start button on the PP2
    LIMITDISK   EQU     %010        ; the limit switch positioned at the cam plate
    EMERGENCY   EQU     %01000      ; the emergency button on the PP2
    LIMITBELT   EQU     %010000     ; the limit switch attached to the axle of the belt
    LIMITLEFT   EQU     %0100000    ; the limit switch on the left side as defined by the guide
    LIMITRIGHT  EQU     %01000000   ; the limit switch on the right side as defined by the guide
    SENSORLAST  EQU     %010000000  ; the sensor which detects if there is a disc left or not

    MOTORDISK   EQU     %01         ; the cam plate motor
    MOTORBELT   EQU     %010        ; the motor of the conveyor belt
    LAMPLAST    EQU     %0100       ; the lamp corresponding with SENSORLAST
    MOTORLEFT   EQU     %010000     ; the sorting containers motor moves to the left
    MOTORRIGHT  EQU     %0100000    ; the sorting containers motor moves to the right
    LAMPCOLOR   EQU     %010000000  ; the lamp above the belt besides the analogue photon ↩
        resistor

    COLORDISK   EQU     %011111111  ; used for converting the ADCONVS value to a usable value
    COLORWHITE  EQU     60          ; analogue threshold for white/no disk
    COLORBLACK  EQU     170         ; analogue threshold for no disk/black


init :
    LOAD    R0  timerinterrupt      ; relative position of the routine

    ADD     R0  R5                  ; memory adress of the interrupt

    LOAD    R5  IOAREA

    LOAD    R1  TMRINT              ; load interrupt register of timer
    STOR    R0  [R1]                ; store the place of routine in the register

    LOAD    R0  DELTATIME
    STOR    R0  [R5+TIMER]          ; initialise the timer
    LOAD    R0  0
    STOR    R0  [GB+PWMSTATE]       ; set initial PWM-phase to highstate = 0
    STOR    R0  [GB+ACTIVE]         ; set initial active inputs to none
    STOR    R0  [GB+PERMANENT]
    STOR    R0  [GB+ERRORSTATE]     ; set initial overload-state to 0
    STOR    R0  [GB+LASTROUND]
    LOAD    R0  80
    STOR    R0  [GB+INTENSITY]

    SETI    NUMTIM                  ; activating interrupt register

    LOAD    R2  0
    BRS     writestatus
    BRA     startstate

timerinterrupt :
    LOAD    R0  [GB+COUNTER]
    ADD     R0  1
    STOR    R0  [GB+COUNTER]        ; Increase the step counter

    LOAD    R0  [R5+INPUT]
    AND     R0  EMERGENCY           ; Check whether the emergency button is pressed
    STOR    R0  [GB+EMERGENCY]

    LOAD    R0  [R5+ADCONVS]        ; store sensible ADCONVS variable to use later
    AND     R0  COLORDISK
    STOR    R0  [GB+SENSORANAL]
    LOAD    R0  [GB+PWMSTATE]
    BEQ     highstate               ; PWMSTATE of 1 will go to lowstate and 0 will go to ↩
        highstate

lowstate:
    LOAD    R0  DELTATIME           ; load remaining time of our PWM
    SUB     R0  [GB+INTENSITY]
```

```
    STOR    R0  [R5+TIMER]          ; bump timer by DELTATIME
    LOAD    R0  0;
    STOR    R0  [GB+PWMSTATE]       ; set variable so that it changes PWMstate
    LOAD    R0  [GB+PERMANENT]
    STOR    R0  [R5+OUTPUT]
    SETI    NUMTIM
    RTE

highstate:
    LOAD    R0  [GB+INTENSITY]      ; load initial time of the PWM
    STOR    R0  [R5+TIMER]
    LOAD    R0  1
    STOR    R0  [GB+PWMSTATE]
    LOAD    R0  [GB+ACTIVE]         ; load active ports
    OR      R0  [GB+PERMANENT]
    STOR    R0  [R5+OUTPUT]         ; active loaded ports
    SETI    NUMTIM
    RTE

startstate:
    BRS     checkemerg              ; check wheter the emergency button is pressed

    LOAD    R0  0                   ; start with lamps off
    STOR    R0  [GB+PERMANENT]
    LOAD    R0  [R5+INPUT]
    AND     R0  STARTBUTTON         ; while the start button is not pressed stay in this state
    BEQ     startstate

    LOAD    R0  0
    STOR    R0  [GB+LASTROUND]      ; reset the LASTROUND variable
    LOAD    R2  1
    BRS     writestatus             ; display the state

    LOAD    R0  0
    STOR    R0  [GB+COUNTER]        ; reset the counter
    LOAD    R0  LAMPLAST            ; turn LAMPLAST on
    STOR    R0  [GB+PERMANENT]

checkstart:
    BRS     checkemerg              ; check wheter the emergency button is pressed

    LOAD    R0  [GB+COUNTER]        ; give the lamp a bit to start, since it is not instantly on
    CMP     R0  WAITLAMP
    BMI     checkstart

    LOAD    R0  [GB+LASTROUND]      ; go to startstate if we are done sorting
    BNE     startstate
    LOAD    R0  [R5+INPUT]
    AND     R0  SENSORLAST          ; if this is the last round said variables accordingly
    BNE     addone

    LOAD    R0  0
    STOR    R0  [GB+COUNTER]        ; reset the counter

diskonbelt:
    BRS     checkemerg              ; check wheter the emergency button is pressed

    LOAD    R0  LAMPCOLOR           ; make sure to also turn the checking lamp on
    STOR    R0  [GB+PERMANENT]
    LOAD    R0  MOTORDISK           ; start the motor for the camplate
    STOR    R0  [GB+ACTIVE]

    LOAD    R0  [GB+COUNTER]
    CMP     R0  EJECTWAIT
    BPL     countertozero_cam       ; We know something is wrong with either the cam plate or ↩
        the limit switch there.

    LOAD    R0  [R5+INPUT]
    AND     R0  LIMITDISK           ; Keep going while the limit switch is not pressed
    BEQ     diskonbelt

    LOAD    R2  2
    BRS     writestatus             ; display the state
    LOAD    R0  0
    STOR    R0  [GB+COUNTER]        ; reset the counter

diskcolor:
```

```
    BRS      checkemerg               ; check wheter the emergency button is pressed

    LOAD     R0   MOTORBELT           ; enable the motor of the conveyor belt
    STOR     R0   [GB+ACTIVE]

    LOAD     R0   [GB+SENSORANAL]
    CMP      R0   COLORWHITE
    BMI      countertozero_white      ; Sensor lower then 100, so disk is white

    LOAD     R0   [GB+SENSORANAL]
    CMP      R0   COLORBLACK
    BPL      countertozero_black      ; Sensor between 100 and 180, there is a disk but we don't ↩
        know which one
                                      ; if disk is white, we find out earlier, else go on with ↩
                                        black

    LOAD     R0   [GB+COUNTER]
    CMP      R0   BELTWAIT            ; if the disk does not reach the sensor in time go to ↩
        belterror
    BPL      countertozero_belt

    BRA      diskcolor

countertozero_black:
    LOAD     R0   0
    STOR     R0   [GB+COUNTER]        ; reset the counter
diskblack:
    BRS      checkemerg               ; check wheter the emergency button is pressed

    LOAD     R2   3
    BRS      writestatus              ; display the state

    LOAD     R0   [GB+COUNTER]        ; if the box does not get to the right in time
    CMP      R0   BELTWAIT                   ; go to boxblackerror
    BPL      boxblackerror

    LOAD     R0   MOTORRIGHT
    OR       R0   MOTORBELT           ; enable the motor for the belt and move boxes to the right
    STOR     R0   [GB+ACTIVE]
    LOAD     R0   [R5+INPUT]
    AND      R0   LIMITRIGHT          ; while the boxes are not in place repeat this subroutine
    BEQ      diskblack

    LOAD     R2   5
    BRS      writestatus              ; display the state
    LOAD     R0   0
    STOR     R0   [GB+COUNTER]        ; reset the counter
    BRA      diskdone

countertozero_white:
    LOAD     R0   0
    STOR     R0   [GB+COUNTER]        ; reset the counter
diskwhite:
    BRS      checkemerg               ; check wheter the emergency button is pressed

    LOAD     R2   4
    BRS      writestatus              ; display the state

    LOAD     R0   [GB+COUNTER]        ; if the box does not get to the right in time
    CMP      R0   BELTWAIT                  ; go to boxblackerror
    BPL      boxwhiteerror

    LOAD     R0   MOTORLEFT
    OR       R0   MOTORBELT           ; enable the motor for the belt and move boxes to the left
    STOR     R0   [GB+ACTIVE]
    LOAD     R0   [R5+INPUT]
    AND      R0   LIMITLEFT           ; while the boxes are not in place repeat this subroutine
    BEQ      diskwhite

    LOAD     R2   5
    BRS      writestatus              ; display the state

    LOAD     R0   0
    STOR     R0   [GB+COUNTER]        ; reset the counter

diskdone:
    BRS      checkemerg               ; check wheter the emergency button is pressed
```

```
    LOAD    R2  6
    BRS     writestatus             ; display the state
    LOAD    R0  MOTORBELT
    STOR    R0  [GB+ACTIVE]

    LOAD    R0  [GB+COUNTER]
    CMP     R0  DISKWAIT            ; wait for the disk to fall in the containers
    BMI     diskdone

    LOAD    R0  LAMPLAST            ; light the lamp
    STOR    R0  [GB+PERMANENT]

    LOAD    R0  0                   ; stop all the motors
    STOR    R0  [GB+ACTIVE]
    STOR    R0  [GB+COUNTER]        ; reset the counter

    BRA     checkstart

; Error detecting
boxwhiteerror:
    LOAD    R0  0
    STOR    R0  [GB+ACTIVE]         ; stop the motors
    STOR    R0  [GB+COUNTER]        ; reset the counter
    LOAD    R0  [R5+INPUT]
    AND     R0  LIMITRIGHT          ; when the containers move to the wrong side
    BNE     boxpolarityright        ; check if the polarity is reversed
boxwhiteleft:
    LOAD    R0  MOTORRIGHT
    STOR    R0  [GB+ACTIVE]         ; move the sorting container to the right
    LOAD    R0  [R5+INPUT]
    AND     R0  LIMITRIGHT          ; if it does arrive on the right then
    BNE     leftswitchkaduuk        ; the left switch does not work correctly
    LOAD    R0  [GB+COUNTER]
    CMP     R0  BOXWAIT
    BMI     boxwhiteleft            ; if the box do NOT arrive on the right in time then
    BRA     motorboxkaduuk          ; the motor does not work

boxblackerror:
    LOAD    R0  0
    STOR    R0  [GB+ACTIVE]         ; stop all the motors
    STOR    R0  [GB+COUNTER]        ; reset the counter
    LOAD    R0  [R5+INPUT]
    AND     R0  LIMITLEFT           ; when the containers move to the wrong side
    BNE     boxpolarityleft         ; check if the polarity is reversed
boxblackright:
    LOAD    R0  MOTORLEFT
    STOR    R0  [GB+ACTIVE]         ; move the sorting container to the left
    LOAD    R0  [R5+INPUT]
    AND     R0  LIMITLEFT           ; if it does arrive on the left then
    BNE     rightswitchkaduuk       ; the right switch does not work correctly
    LOAD    R0  [GB+COUNTER]
    CMP     R0  BOXWAIT
    BMI     boxblackright           ; if the box do NOT arrive on the left in time then
    BRA     motorboxkaduuk          ; the motor does not work

boxpolarityright:                   ; here we check to polarity of the motor of the sorting ↩
    containers
    LOAD    R0  MOTORRIGHT
    STOR    R0  [GB+ACTIVE]         ; move the boxes to the right
    LOAD    R0  [R5+INPUT]          ; if the boxes arrive left, the polarity is reversed
    AND     R0  LIMITLEFT
    BNE     boxswitch
    LOAD    R0  [GB+COUNTER]        ; if the boxes do NOT arrive in time on the left the motor ↩
        does not work
    CMP     R0  BOXWAIT
    BMI     boxpolarityright
    BRA     motorboxkaduuk
boxpolarityleft:                    ; here we check to polarity of the motor of the sorting ↩
    containers
    LOAD    R0  MOTORLEFT
    STOR    R0  [GB+ACTIVE]         ; move the boxes to the right
    LOAD    R0  [R5+INPUT]          ; if the boxes arrive right, the polarity is reversed
    AND     R0  LIMITRIGHT
    BNE     boxswitch
    LOAD    R0  [GB+COUNTER]        ; if the boxes do NOT arrive in time on the left the motor ↩
        does not work
```

```
    CMP     R0  BOXWAIT
    BMI     boxpolarityleft
    BRA     motorboxkaduuk

boxswitch:
    LOAD    R0  5
    STOR    R0  [GB+ERRORSTATE]
    BRA     emergstate

countertozero_cam:
    LOAD    R0  0
    STOR    R0  [GB+COUNTER]        ; reset the counter
camerror:
    BRS     checkemerg              ; check wheter the emergency button is pressed

    LOAD    R0  MOTORBELT
    STOR    R0  [GB+ACTIVE]         ; start the motor for the belt

    LOAD    R0  [GB+SENSORANAL]
    CMP     R0  COLORWHITE
    BMI     limitdiskerror          ; Sensor lower then 100, so disk is white
    CMP     R0  COLORBLACK          ; if the disk arrives in time at the lamp
    BPL     limitdiskerror          ; then limit switch with the camplate is not working ↩
        correctly

    LOAD    R0  [GB+COUNTER]
    CMP     R0  BELTWAIT            ; if the disk does not arrive at the sensor in time
    BMI     camerror                ; then the camplate does not turn

    BRA     motordiskerror

motordiskerror:
    LOAD    R0  1                   ; The motor for the camplate is not getting power
    STOR    R0  [GB+ERRORSTATE]
    BRA     emergstate

limitdiskerror:
    LOAD    R0  2                   ; The limitswitch by the camplate is not attached
    STOR    R0  [GB+ERRORSTATE]
    BRA     emergstate

countertozero_belt:
    LOAD    R0  0
    STOR    R0  [GB+COUNTER]        ; reset the counter
    LOAD    R0  [R5+INPUT]
    AND     R0  LIMITBELT           ; store the state of the index cam
    STOR    R0  [GB+BELTSTATUS]
belterror:
    LOAD    R0  [R5+INPUT]
    AND     R0  LIMITBELT
    CMP     R0  [GB+BELTSTATUS]     ; if the index cam value changes, then the belt is working
    BNE     diskhostage             ; but the disc is physically blocked

    LOAD    R0  [GB+COUNTER]
    CMP     R0  DISKWAIT            ; repeat this for a given time
    BMI     belterror

    LOAD    R0  4                   ; There is a disc stuck on the belt
    STOR    R0  [GB+ERRORSTATE]
    BRA     emergstate

motorboxkaduuk:
    LOAD    R0  6                   ; The motor for the boxes is not getting power
    STOR    R0  [GB+ERRORSTATE]
    BRA     emergstate

rightswitchkaduuk:
    LOAD    R0  7
    STOR    R0  [GB+ERRORSTATE]
    BRA     emergstate

leftswitchkaduuk:
    LOAD    R0  8
    STOR    R0  [GB+ERRORSTATE]
    BRA     emergstate

diskhostage:
```

```
    LOAD     R0   3                    ; The motor for the belt is not getting power
    STOR     R0   [GB+ERRORSTATE]
    BRA      emergstate
emergtozero:
    LOAD     R0   0
    STOR     R0   [GB+ERRORSTATE]
emergstate:
    LOAD     R0   0
    STOR     R0   [GB+ACTIVE]          ; when in emergency mode everything is off
    STOR     R0   [GB+PERMANENT]
    STOR     R0   [R5+OUTPUT]

    LOAD     R0   [GB+ERRORSTATE]
    BRS      Hex7Seg                   ; translate (value in) R0 into a display pattern
    STOR     R1   [R5+DSPSEG]          ; and place this in the Display Element
    LOAD     R1   %01000               ; R1 := the bitpattern identifying Digit 0
    STOR     R1   [R5+DSPDIG]          ; activate Display Element nr. 0

    LOAD     R0   [R5+INPUT]
    AND      R0   STARTBUTTON
    BEQ      emergstate

    PULL     R0                        ; to fix the shit we fucked up
    BRA      startstate

writestatus:
    LOAD     R0   R2
    BRS      Hex7Seg                   ; translate (value in) R0 into a display pattern
    STOR     R1   [R5+DSPSEG]          ; and place this in the Display Element
    LOAD     R1   %0100000             ; R1 := the bitpattern identifying Digit 0
    STOR     R1   [R5+DSPDIG]          ; activate Display Element nr. 0
    RTS

Hex7Seg      :
    BRS      Hex7Seg_bgn               ; push address(tbl) onto stack and proceed at "bgn"
Hex7Seg_tbl  :
    CONS     %01111110                 ; 7-segment pattern for '0'
    CONS     %00110000                 ; 7-segment pattern for '1'
    CONS     %01101101                 ; 7-segment pattern for '2'
    CONS     %01111001                 ; 7-segment pattern for '3'
    CONS     %00110011                 ; 7-segment pattern for '4'
    CONS     %01011011                 ; 7-segment pattern for '5'
    CONS     %01011111                 ; 7-segment pattern for '6'
    CONS     %01110000                 ; 7-segment pattern for '7'
    CONS     %01111111                 ; 7-segment pattern for '8'
    CONS     %01111011                 ; 7-segment pattern for '9'
    CONS     %01110111                 ;  7-segment pattern for 'A'
    CONS     %00011111                 ;  7-segment pattern for 'b'
    CONS     %01001110                 ;  7-segment pattern for 'C'
    CONS     %00111101                 ;  7-segment pattern for 'd'
    CONS     %01001111                 ;  7-segment pattern for 'E'
    CONS     %01000111                 ;  7-segment pattern for 'F'
Hex7Seg_bgn:
    MOD      R0   16
    LOAD     R1   [SP++]               ; R1 := address(tbl) (retrieve from stack)
    LOAD     R1   [R1+R0]              ; R1 := tbl[R0]
    RTS

addone:                                ; gets called if this is the last round
    LOAD     R0   1
    STOR     R0   [GB+LASTROUND]       ; store the variable LASTROUND accordingly
    LOAD     R2   0
    BRS      writestatus              ; write the value to the display
    BRA      diskonbelt

checkemerg:                            ; check if the emergency button is pressed
    LOAD     R0   [R5+INPUT]
    AND      R0   EMERGENCY
    BNE      emergtozero
    RTS
@END
```

14.4 Debug program min_max_brightness

```
@DATA
    MIN     DS      1
    MAX     DS      1

@CODE

    IOAREA      EQU   -16       ; address of the I/O-Area, modulo 2^18
    INPUT       EQU   7         ; relative position of the input buttons
    OUTPUT      EQU   11        ; relative position of the power outputs
    DSPDIG      EQU   9         ; relative position of the 7-segment display's digit selector
    DSPSEG      EQU   8         ; relative position of the 7-segment display's segments
    ADCONVS     EQU   6         ; relative position of the ADCONVS volt (255 = 5 volt)
    TIMER       EQU   13        ; timer register (relative to IOAREA)
    SECOND      EQU   10000     ; The amount of timersteps in 1 second

begin :
    LOAD    R5  IOAREA          ; R5 := "address of the area with the I/O-registers"
    LOAD    R0  0               ; R0 = 0;
    STOR    R0  [GB+MAX]
    LOAD    R0  %010000010
    STOR    R0  [R5+OUTPUT]     ; Output leds Off to start with

    LOAD    R0  255
    STOR    R0  [GB+MIN]

    BRA   main                ;  skip subroutine Dec7Seg

;
;       Routine Dec7Seg maps a number in the range [0..9] to its hexadecimal
;       representation pattern for the 7-segment display.
;       R0 : upon entry, contains the number
;       R1 : upon exit,  contains the resulting pattern
;
Dec7Seg     :  BRS  Dec7Seg_bgn  ;  push address(tbl) onto stack and proceed at "bgn"
Dec7Seg_tbl : CONS  %01111110    ;  7-segment pattern for '0'
              CONS  %00110000    ;  7-segment pattern for '1'
              CONS  %01101101    ;  7-segment pattern for '2'
              CONS  %01111001    ;  7-segment pattern for '3'
              CONS  %00110011    ;  7-segment pattern for '4'
              CONS  %01011011    ;  7-segment pattern for '5'
              CONS  %01011111    ;  7-segment pattern for '6'
              CONS  %01110000    ;  7-segment pattern for '7'
              CONS  %01111111    ;  7-segment pattern for '8'
              CONS  %01111011    ;  7-segment pattern for '9'
Dec7Seg_bgn:   MOD  R0  10       ;  R0 := R0 MOD 10 , just to be safe...
              LOAD  R1  [SP++]   ;  R1 := address(tbl) (retrieve from stack)
              LOAD  R1  [R1+R0]  ;  R1 := tbl[R0]
               RTS


;
;       The body of the main program
;

main:
    ; Get the value of potentiometer and display on output leds
    LOAD  R0  [R5+ADCONVS]    ; Load the potentiometer
    AND   R0  %011111111      ; Read only the lower 8 bits

    LOAD  R2  [R5+INPUT]
    AND   R2  %01
    BNE   reset

    LOAD    R3  [GB+MIN]
    CMP     R3  R0
    BLT     check_max
    STOR    R0  [GB+MIN]
    BRA     calc_value

check_max:
    ; Get the value of potentiometer and display on output leds
    LOAD  R1  [R5+ADCONVS]    ; Load the potentiometer
```

```
    AND   R1  %011111111      ; Read only the lower 8 bits

    LOAD  R2  [R5+INPUT]
    AND   R2  %01
    BNE   reset

    LOAD  R3  [GB+MAX]
    CMP   R3  R1
    BGT   calc_value
    STOR  R1  [GB+MAX]

    BRA     calc_value

reset:
    LOAD  R0  255
    STOR  R0  [GB+MIN]
    LOAD  R0  0
    STOR  R0  [GB+MAX]

calc_value:
    LOAD    R0  [GB+MIN]
    LOAD    R1  [GB+MAX]

    ; Separate / cycle through the digits
    ADD   R4 1                ; ADD 1 to the displaytimer
    MOD   R4 600              ; Make sure it loops at 600
    CMP   R4  0               ; if 0:
    BEQ   first             ;   display the first digit
    CMP   R4  100             ; if 100:
    BEQ   second              ;   display the second digit
    CMP   R4  200             ; if 200:
    BEQ   third             ;   display the third digit
    CMP   R4  300             ; if 300:
    BEQ   fourth          ;   display the first raw digit
    CMP   R4  400             ; if 400:
    BEQ   fifth            ;   display the second raw digit
    CMP   R4  500             ; if 500:
    BEQ   sixth           ;   display the third raw digit

    ; Loop back to beginning
    BRA   main                ; In case of no new digit selected, loop

first:
    DIV   R0  100             ; input / 100
    BRS   Dec7Seg             ; Turn this value into the display-value for 7-seg
    LOAD  R2  %100000         ; Select the first 7-seg display
    BRA   display             ; Display this digit

second:
    MOD   R0  100             ; input MOD 100
    DIV   R0  10              ; / 10
    BRS   Dec7Seg             ; Turn this value into the display-value for 7-seg
    LOAD  R2  %010000         ; Select the second 7-seg display
    BRA   display             ; Display this digit

third:
    MOD   R0  10              ; input MOD 10
    BRS   Dec7Seg             ; Turn this value into the display-value for 7-seg
    LOAD  R2  %001000         ; Select the third 7-seg display
    BRA   display             ; Display this digit

fourth:
    LOAD  R0  R1
    DIV   R0  100             ; input / 100
    BRS   Dec7Seg             ; Turn this value into the display-value for 7-seg
    LOAD  R2  %000100         ; Select the fourth 7-seg display
    BRA   display             ; Display this digit

fifth:
    LOAD  R0  R1
    MOD   R0  100             ; input MOD 100
    DIV   R0  10              ; / 10
    BRS   Dec7Seg             ; Turn this value into the display-value for 7-seg
    LOAD  R2  %000010         ; Select the fifth 7-seg display
    BRA   display             ; Display this digit

sixth:
```

```
    LOAD   R0   R1
    MOD    R0   10                ; input MOD 10
    BRS    Dec7Seg                ; Turn this value into the display-value for 7-seg
    LOAD   R2   %000001           ; Select the sixth 7-seg display
    BRA    display                ; Display this digit

; Updates the display with Pattern R1 on display R2
display:
    STOR   R1   [R5+DSPSEG]       ; Update the value that has to be displayed
    STOR   R2   [R5+DSPDIG]       ; Select the right 7-seg display unit
    BRA    main                   ; Loop forever

@END
```

14.5 Logbook

Meeting #	Activity	Minutes spend
1	Introduction and expectations	30
	Looking at the requirements, deadlines and scoring guidelines	45
	Ideas for alternative machine	30
	Tutor meeting	45
2	Presentations	60
	Menno and Geert tested the motors using the PP2 processor and their PWM script from computer systems	30
	Thijs and Yannick looked at the Belbin roles	30
3	Checking the inventory list	60
	We discussed the absence of Lisa with the tutor	30
	We discussed the peer reviews with the tutor	10
	We discussed some requirements for the alternative machine with our tutor	10
	We discussed building a tic-tac-toe machine and discarded it, and we decided to build a sorting machine	60
	We made a layout for the first prototype of our machine	30
4	Thijs looked at how the H-bridge works and how it needs to be implemented in the software	15
	Testing the prototype using a crude PWM program controlled by the input buttons	60
	Coming up with some requirements we discovered after the testing	90
5	We made a planning for the rest of the project	
	Geert missed something in his presentation of the V-model, so he made some remarks	
	We discussed Eric and Lisa leaving the group with our tutor	10
	The tutor made some suggestions about the Belbin roles and some general things	15
	We discussed the midterm and final presentations with our tutor	5
	Menno and Geert tried to implement an OVL-interupt for the overcurrent detection on the PP2 to be used in the PWM script and final script	60
	Thijs made an UPPAAL model of the software of the first prototype	60
	Menno made some corrections to his Belbin roles after the suggestions of the tutor	15
	We brainstormed for the new design of the machine	90

6	We discussed which components our machine needs	60
	Thijs worked on the tube and the mechanism which extracts the discs from the tube	120
	Geert worked on the mechanism which sorts the discs	120
	Menno and Yannick worked on the mechanism which includes the sensors and transports the discs	120
7	Thijs did some tests regarding the over current protection	60
	Yannick set up a github repository for the code	15
	The tutor came, but we didn't have anything to discuss	10
	Geert and Menno worked on the mechanism which moves the boxes	150
	Yannick made some improvements to the mechanism which extracts discs from the tube	150
	Thijs worked on the new UPPAAL model	90
8	Menno and Yannick tried to make the transport belt more resource efficient	90
	Thijs continued working on the new UPPAAL model	90
	Geert started working on the assembly program	90
	We discussed how we function as a group with the tutor	20
	We combined the three parts of the machine	120
	We discussed how we want to structure our final report	10
	Menno created the LaTeX document	10
9	Thijs revised the mechanism which moves the containers	120
	Yannick build a "wall" around the belt to prevent the discs from falling of, in this wall he also placed some sensors	120
	Geert and Menno did some work on the software	120
	We worked on the report	90
10	Geert and Menno continued working on the software	210
	Thijs hooked up all four lamps on the single 12v output	60
	Yannick worked on the report	210
	Thijs worked on the report	150
11	Geert and Menno continued working on the software	210
	Thijs and Yannick worked on the report	210

12	We discussed our problems with the over current detection with our tutor	10
	We discussed some things about the belbin roles we have to hand in with our tutor	10
	We discussed some things about the midterm presentation with our tutor	10
	We wrote the second reflection on our Belbin roles	60
	Geert, Yannick and Menno messed around with the analogue input of the colour sensor	150
	Thijs worked on the new UPPAAL model	150
13	Menno and Yannick worked on the midterm presentation	210
	Thijs and Geert worked on the final report	210
14	The tutor gave some feedback on our midterm presentation	30
	The tutor gave some feedback on our 2nd belbin roles report	5
	Menno and Yannick implemented the changes the tutor suggested for the presentation, and practiced it	210
	Thijs and Geert made some changes to the UPPAAL model, and also implemented some crude error detection	60
	Thijs and Geert worked on the software	150
15	We had trouble with the light sensor, Yannick and Meno made changes to the light sensor	150
	Thijs and Geert implemented the changes to the light sensor in the UPPAAL model and in the script	150
	We did some work on the report	90
16	Yannick and Menno practised the midterm presentation and also made some changes to it	90
	Menno and Geert made a counter to have some delay between the disc being detected and the next disc being inserted	60
	Thijs, and later the rest worked on a new moving box system	60
	Geert and Yannick implemented the code which stops the machine after the last disc has been sorted	60
	Thijs worked on the error detection in the UPPAAL model	120
17	We discussed last weeks presentations with our tutor	10
	Geert and Menno worked on implemented the emergency button	120
	Thijs and Yannick worked on the report, Geert and Menno joined later	240

18	Geert and Thijs worked on the implementation of the error detection in the UPPAAL model	120
	Menno did some performance tests	15
	Menno and Yannick worked on the report	120
	We spend the rest of the meeting trying to solve the "branch problem"	90
19	We spend the entire meeting trying to solve the "branch problem", we now now that the assembler creates and offset in the line numbers (when we use labels) for some reason, and managed to create a workaround	180
20	Yannick and Menno the exact reason why the "branch problem" occured	180
	Thijs and Geert worked on adding more fault detection	180
21	Yannick and Menno worked on error detection for the boxes	200
	We discussed some things about the final presentation with our tutor	20
	Thijs and Geert worked on the presentation	200
22	We took a look at the scoring guide and wrote down some areas in which we could improve	30
	Geert and Thijs tried to solve some issues with the colour sensor	150
	Menno and Yannick did some fixes to the error detection part of the code	120
	Menno and Yannick worked on making a debugging manual	30
23	Menno and Yannick worked on a problem regarding the error detection of the belt	60
	Geert and Thijs practised and worked on the final presentation	60
	We worked on the report	90
	We provided the tutor with evidence that our machine functioned correctly and that is could detect and identify errors	36
24	We had a lengthy tutor meeting	45
	We worked on proofreading and editing the report	150
25	We made sure the machine will work, replacing broken cables and checking all subsystems	60
	Thijs and Geert practiced their final presentation extensively	120

	Yannick and Menno worked on making the assembly code clearer and cleaner	120
26	We filled in the peer review	10
	We all read the report, and made small adjustements before handing it in	30