Final Document
OGO 2IO70
Version 0.3

Group 2
A.K. Bokharouss
a.k.bokharouss@student.tue.nl

October 31, 2016

Contents

Chapter 1

Preface

This technical report describes a sorting machine developed over the course of 8 weeks by Group 2 for the course 2IO70 DBL: Embedded Systems. This is a technical document written to explain the process of creating this embedded system, the sorting machine. This manual is intended towards everybody who wants to see how we developed the machine for this course.

This text uses one text convention. In the orientation phase, Italics are used to identify the names of the persons responsible for the different things that need to be done. In all the other phases, the same text convention was used to identify key-words in sentences.

The assignment that was given to us was to construct and program a machine that sorts 12 disks depending on their color(black or white). Given that we were one of the trail groups that had the chance to work with the EV3 processor, we decided to not only sort the disks, but also implement a lot of error detection and additional features.

Chapter 2

Summary

Upcoming chapters document each phase of the development process. This summary consists of a brief notion about what to expect in each phase. The first phase is the Orientation which describes how the project was intended to be executed. All the roles have been distributed for the whole period, a schedule was set and a date for every deliverable was put. Also, a description of the EV3 is in this phase.

The Machine Design phase describes the physical implementation of the sorting machine, its requirements, use-cases, user-constraints, safety properties and the interface. It also contains a walk-through of all the design decisions we have made.

The Software Specification phase describes the design of our software, which is base on a Finite State Automaton. The correctness is proven using UPPAAL.

The Software Design phase describes what functions are needed and the functions of each of these.

In the last construction phase, the Software Implementation and Integration, the software designed in the previous phase is translated in Java using the leJOS library.

The System Validation and Testing phase is used to test the functionality of our machine. Code review were made and we tested everything and described every test in the test cases. It also describes the proof that was made using UPPAAL.

Chapter 3

Orientation phase

3.1 Summary

This phase started with orientating on the requirements and the flow of the project by reading and making abstracts of the Project Guide (subsection 3.2.1). When we had a sufficient idea about the way the project was intended to be executed we started with the Work Plan (see section 3.3). This includes defining and distributing the roles of president, secretary, quality insurance manager, the author responsible for each phase, and the time schedule for the project itself and the deliverables.

For everyone who is not familiar with the EV3, there is a not too in-depth overview LEGO Mindstorm EV3 (see subsection 3.2.3) mostly about the physical aspects, the software side is covered in LeJOS (see section 7.3).

3.2 Orientation

3.2.1 Abstracts of the Project Guide

Every member of the group has individually written a short abstract of the project guide. This has really helped us in understanding the structure of the different levels of the project. It gives us a good overview of all the different phases of the project. We know where to start, and to where we're headed.

3.2.2 Work Plan

Part of the orientation phase is consists of writing the Work Plan. In the Work Plan is indicated explicitly who will be doing what and when. The Work Plan also contains the deadlines of all other documents that have been handed in by now. The Work Plan also contains a table that indicates which group member was responsible for handing in which document. Finally the Work Plan indicates which three group members have performed the midterm presentation and which three group members have performed the final presentation of the group. The deadlines that are included in the Work Plan are mostly chosen by us, and approved by our tutor. The deadlines in the Work Plan pertain to the final versions, so drafts were handed in beforehand, so that we could process the feedback from our tutor.

3.2.3 LEGO Mindstorm EV3

Overview

The LEGO Minstorm series combines the simplicity and quality of LEGO with robotics. The EV3 is the latest in the series and is equipped with an ARM processor, it's own screen, a speaker, 3 motors, multiple sensors, and a decent set of LEGO Technic parts to combine everything. These parts can be used to construct a wide range of robots, from simple beginner robots whom can only drive forward, more advanced designs like for example 3D milling machines, or even more complex designs. The EV3 can be linked with others to be able to control up to 16 motors with just one Mindstorm. If the CPU is the bottleneck, then (using third-party software) it is even possible to let a computer do (parts of) the calculations in stead.

The Brick

The Mindstorm brick is the central part, it does all calculations, stores all programs, supplies the connection between the computer, contains the needed batteries, and connects to all sensors and motors. It features a (non-color) screen, a couple of buttons, a speaker, a MicroSD slot to increase the internal storage, Bluetooth, and two USB connectors. One of the USB connectors is used to communicate with the computer (can also be done with Bluetooth), on the other one can 'normal' USB devices be used in combination with the brick, this should enable the brick to use other devices as input, for example a USB keyboard could be connected, although it should be noted that some third-party code is needed to make this compatible with their software.

Sensors

Touch The EV3 has two touch sensors, it returns the state (pressed or not) as a boolean value.

Color This sensor can measure the light intensity of 8 different colors, besides this it can also measure the amount of ambient light and reflected red light.

Gyroscope Measuring change in orientation - and thus rotation - can be done with the gyroscope.

Ultrasonic Measures distance with ultrasonic, it works best at detecting large objects and can measure distance between 1 and 255 cm.

Motors

The EV3 features 3 motors, two large ones and one 'Medium' one, one extra can be added since the EV3 brick supports up to 4 motors to be connected at the same time. All motors have a rotation sensor, which make it possible to control position, speed, and acceleration. The motor can also 'lock' the current position, in this mode it will try to keep itself in it's current position.

Programming

By default the EV3 comes with LEGOs own firmware and software. This gives the user a drag-and-drop interface to start building, although a lot of robots can be controlled with this, to enable the full potential of the robot third-party firmware and software is needed to enable more complex programming languages to be used to program the robot. There are multiple alternatives to the default Mindstorm programming kit, two examples are:

leJOS A free Java alternative, originally a fork of the TinyVM Project for the first Mindstorm, shortly after the release of the EV3 it got ported, it allows more complex programming structures like recurrence, arrays, objects, and many more.

RobotC A payed alternative for the original firmware, it is C-based and only supports Windows as development platform. Programs for the EV3 can be written in C, besides programming the physical EV3 it is also possible to develop, test, and debug (EV3) robots in a virtual environment.

3.2.4 Differences between EV3 and PP2

The EV3 contains a way more complex processor, compilers for this processor are available, and thus programming in a higher language is possible. These higher programming languages often have a extensive list of features, like for example object oriented programming and data types like floats.

Also there are libraries available which contain methods for handling a lot of things, like the interpretation of input of sensors and the correct way of generating output towards motors, meaning that the direct input and output can be handled by these libraries making it in most cases unnecessary to implement these yourself. Some libraries even contain classes which are abstract implementations of some common robot designs.

If we compare the sensor of the PP2 against the sensors of the EV3 then there are two main differences. First of all, the EV3 kit contains a lot more sensors than the PP2 kit where the other groups work with. Although every analogue signal - if the voltage is in the correct range - can function as an input for the PP2, the EV3 features something comparable since extra sensors can be made by third parties. This is however more complicated than the PP2 analogue input method, but they both feature the possibility to add sensors yourself.
The second difference between the sensors of both kits is that the sensors of the EV3 are a lot more evolved, not only do they all feature the same compact design, but also are most of them quite complex compared to the sensor of the PP2.

3.3 Work Plan

3.3.1 Phases

We have made a division of the phases over the weeks. The deliverables of each phase are to be handed in each Friday of the particular week of that phase (except Software Implementation and Integration which is done in two weeks). This way our tutor can have a look at it over the weekend and necessarily Monday so that he can give us feedback during the meeting of the following week (Tuesday). If necessary, changes are made on Tuesday, and then again shortly discussed on the Wednesday meeting. If it is approved, the final version of the particular deliverable will be handed in on that Wednesday. This way we'll have a structured way of processing feedback and handing in the final version of each deliverable.

Orientation

The first phase is the Orientation phase. In this phase we will deliver the Work Plan (That's due Friday week 2 at 5 o'clock) This will be made by *Jolan and Abdel* on Tuesday, *Adriaan* will make a list of deliverables. The Work Plan will be checked and finished on Wednesday. We will also make a checklist to keep track of all the documents we need to submit. This will be done by *Adriaan.* We will eventually add this to the Work Plan. The 2nd week on Wednesday we will also have three presentations. A presentation about the V-model will be done by *Abdel* (since *Bogdan* is absent), a presentation about Software specification and UPPAAL will be done by *Jolan and Valentin*, finally a presentation about System validation and testing will be done by *Ivo and Adriaan.*

Machine Design

The second phase is designing the sorting machine. In this phase we will try to build the machine itself using our Lego and we'll deliver a document concerning the machine design, this will be done by *Jolan* by Friday (19 February). If we do not succeed in finishing it on the 19th, then we will hand it in on the same day as the deliverable of the following week (Software Specification). In this document we have to define the System Level Requirements, consisting of the user cases, user constraints, and safety properties, and in addition the Machine Interface. The design of the physical machine and the system level requirements is an iterative process. We will deliver this document in week 2 as well and we'll work on it on Wednesday. If we don't succeed, we'll continue on Friday morning.

Software Specification

The third phase is Software specification. We will work on this on Tuesday and Wednesday in week 3 and deliver a document about this, made by *Valentin* together with the UPPAAL model made by us all. This will be done by Friday (26 February) that week (3). *[February 17] Valentin is quitting the Software Science program, Bogdan will be responsible for the document instead of Valentin.* From this point on we will unfortunately be with 5 people instead of 6. We thus have to make some adaptions to the draft of Work Plan [February 17]. This same week there will also be Midterm presentations on Wednesday by *Adriaan, Abdel and Ivo.*

Software Design

The fourth phase is Software Design. In this phase we'll write pseudo-code that will realize the specified functions of the sorting machine. This program serves as a stepping stone towards the leJOS (Java) code that will be constructed in the next phase: Software Implementation and Integration. If applicable, Exception Service Routines can be formulated as ordinary, parameter-less methods. In the document "Software Design" the design decisions and choices of the data representation must be explained and motivated. We will work on this phase in week 4 and hand in the deliverable before Friday (4 March). If we however do not succeed in handing in the deliverable in week 4, we will move it to week 5 since we have reserved two weeks for Software Implementation and Integration. This document is to be constructed by *Abdel.*

Software Implementation and Integration

The next phase is Software Implementation and Integration. The pseudo-code from previous phase must be implemented in language suitable for the EV3. We also have to consider the different possibilities that EV3 provides, language-wise, and weigh in mind which is the best choice for us. The data representation chosen and the coding standard are to be documented in a short document "Software Implementation". Finally the program must be compiled and integrated so that it can be run by the EV3. We will work on this phase in week 5 and week 6 and hand in the deliverable before Friday (18 march) (before the 18th if possible, because this way we'll have more time to process feedback and to debug). This document is to be constructed by *Adriaan.*

System Validation and Testing

We have one phase that is parallel to all the other phases and activities except the machine design phase. It is the System Validation and Testing phase. The goal of this activity is to ensure the high quality of the final product. An important objective of this activity is to ensure trace-ability from the System Level Requirements down to the integrated object code. The System Level Requirements document should also contain the choices made with relations to validation and testing, since it is impossible to thoroughly test, review and prove everything.

We also have to argue the trace-ability between system level requirements and the final code, to prove that we have designed the machine as carefully as we could. Note that it is not required to actually put all review, test, proof reports in the deliverable of this phase. But we will try to keep "log books" about these activities using a textual file. This text file will be submitted as a digital appendix to our deliverable (and also to the final report). Part of System Level Requirements is code review, we also have to deliver a "review report" (short, max 1 A4) as documentation of this, stating the date, people involved, the kind of activity, and how it was performed (Were bugs found? Were improvements made to the code? etc.)

The System Level Requirements should thus include description of reviews, test executions and properties and proofs in appendix. *Bogdan* will construct, and is responsible, for the deliverables of this phase. *[February 17] Since Valentin is quitting the program, we have moved Bogdan to Software Specification and have Jolan as replacement for constructing the System Validation and Testing document.* Since this phase is parallel to other phases, we will all have a big contribution to this document, so *Jolan* is the one who has the overview, but we will all add things to this document. As mentioned before, this phase is parallel to other phases, and these phases are from week 3 until and including week 7. The deliverable will be handed in before the final report, so that we can process feedback, if necessary.

Completion

The final phase is the completion phase. In this phase the documents resulting from the preceding phases must be integrated into a single document "Final Report". In addition we have the final presentation in the last week 8. They will be done by *Jolan and Bogdan.* Keeping the conclusions from the final presentations in mind, some time is reserved to solve unforeseen problems.
The final report presents the reader with a clear picture of the designed machine, the method of working followed, the specification, validation, and design of the software, and a motivation of the main design decisions. The conclusion must pay attention to what has been learned during this project and which complications have been encountered and how the group has handled these. *Ivo* is responsible for the final version of the final report. It will be handed in on Friday, 1 April. This way we'll have some time to process feedback, if necessary, before we hand in the final version on or before Friday, 8 April.

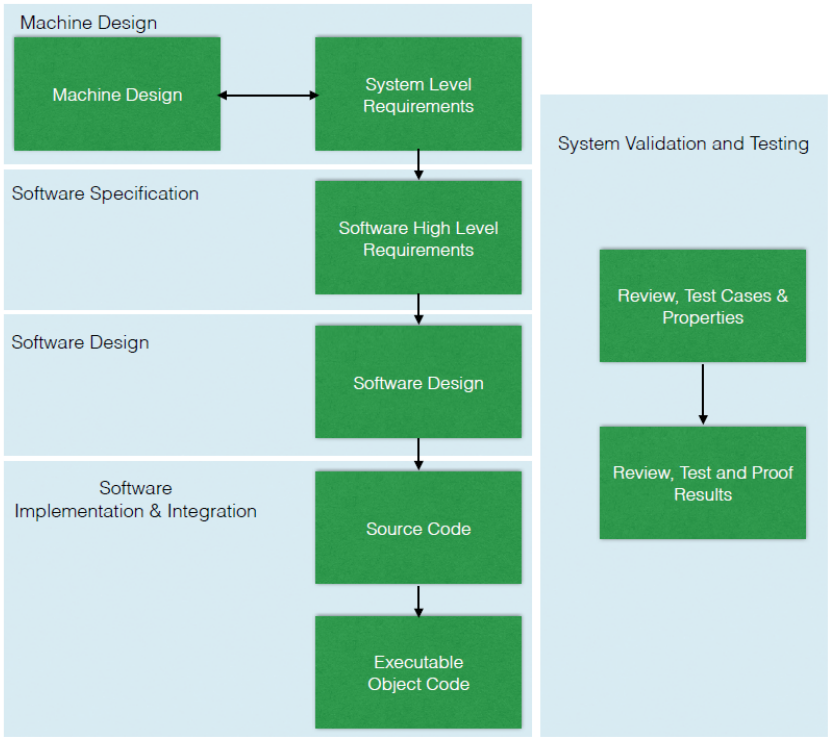

Figure 3.1: Overview of the software development phases

3.3.2 Quality Assurance Manager

The Quality Assurance Manager is responsible for the deliverables. He will not necessarily construct the particular document to be handed in, but he'll make sure that it is finished and handed in, in due time. He will also be responsible for the processing of feedback giving by our tutor into the final version of the particular deliverable.

During the project we will have three different Quality Assurance Managers. He monitors whether all communication between group members runs smoothly and whether all (intermediary and final) deliverables, documents and "products" are handed in time to other members, our *tutor or/and Pieter Cuijpers.* The Work Plan may have to be adjusted. The Quality Assurance Manager does not have to redistribute work, but makes sure the Work Plan is up to date. Finally, the Quality Assurance Manager monitors the quality of handed-in products, by checking whether the products meet the requirements.

3.3.3 Deliverables

What	When	Who
Abstract of the Project Guide	Friday week 1, 17:00 individual	Individual
Self-reflection	Friday week 7, 17:00 individual	Individual
Work Plan with assignment of tasks	Friday week 2, 17:00 group	Abdel
Machine Design	Firday week 2, 17:00	Jolan
Software Specification	Friday week 4, 17:00	Bogdan
Software Design	Friday week 6, 17:00	Abdel
Software Implementation	Friday week 6, 17:00	Adriaan
System Validation and Testing	according to Work Plan	Jolan
Midterm Presentations	Tuesday 23rd, Thursday 25th and Friday 26th Feb 2016 (Room MMP1)	Ivo, Abdel and Adriaan
Final Presentations and Competition	Tuesday 29th, Thursday 31st Mar 2016 (Room MMP1) and Friday 1st Apr 2016 (Room GEM Z)	Jolan, Adriaan and Bogdan
Final Report	Friday Apr 8th 2016, 17:00	Ivo

3.3.4 Distribution of Tasks

week	President	Secretary	Quality Insurance Manager
Week 1	-	-	-
Week 2	Abdel	Ivo	Adriaan
week 3	Jolan	Bogdan	Adriaan
week 4	Bogdan	Ivo	Adriaan
week 5	Adriaan	Abdel	Bogdan
week 6	Ivo	Jolan	Bogdan
week 7	Adriaan	Bogdan	Abdel
week 8	Jolan	Ivo	Abdel

The material manager during the entire course is *Abdel*, he will look after the key of the locker and the material.

3.4 Literature

- Project Guide 2016 Q3 - [2IO70] University of Technology Eindhoven
- [What is V-model- advantages, disadvantages and when to use it?] http://istqbexamcertification.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/

Chapter 4

Machine Design phase

4.1 Summary

At the beginning we stated the System Level Requirements which consists of different subjects like use-cases, user constraints and safety properties. We then have a look at the Machine Interface. Followed lastly by a summary and motivation of our design decisions.

4.2 System Level Requirements

The System Level Requirements (SLRs) specify what can be expected from the sorting machine, how it should be operated, and under which circumstances it will function as specified. Design decisions (see section 4.3)

4.2.1 Requirements

The system should meet the following requirements, if the Use-cases (see subsection 4.2.2) and User constraints (see subsection 4.2.3) are met:

> If sorting is started by the operator, it should sort 12 black and white disks, based on their difference in value of reflection of the visible light spectrum - thus grouping all black discs, and grouping all white discs. Within 3 minutes the machine should be finished and all discs should be sorted, both groups of discs should then be in their own dedicated container. Both containers have the specific purpose to contain one group, which is defined beforehand. When the sorting machine is powered on it should show in which state it is and show all detected problems it encounters.

4.2.2 Use-cases

To sort one disk, the machine needs to do the following: First the wheel is supposed to be calibrated to the right starting angle. Whether this is either of the five ways practically doesn't matter, however mathematically it can be convenient to start at a certain angle each time the machine starts. After the confirms that the tube is empty, when inserting a disk in the tube, a counter goes up. This counter holds the value of the number of disks in the tube. This will be displayed on the display of the EV3 brick. In the case of one disk, this value is supposed to be 1. When the disk then exits the tube it enters the wheel, blocking the colour sensor that was originally facing the gray outside of the wheel. We now get the value of the sensor. If the disk gets stuck in the tube, we can now detect this, since the value of the colour sensor will remain the same. Otherwise, we can detect whether the current disk is black or white and then move the wheel $\frac{360^\circ}{5}$ to the left or right. The disk now drops in the correct container, either the one for white disks or the one for the black ones. The display will now show 0 for the disks still in the tube and have white or black depending on the colour set to 1. If one of the disks gets stuck or somehow blocks the wheel, we can detect this using the sensors in the rotation motor. We will inform the user about this and ask what to do.

To sort more than one disk mainly is the same as sorting but one. The only difference is that the second one and those thereafter stay in the tube while the first one is being scanned. The display will still show the number of disks in the tube, how many black ones are already in the basked and how many white disks. When the first one is done, the wheel moves it to the left or right, unveiling a new slot to the tube where then the next disk can fall in, get scanned and so on. When no more disks slide in the wheel, the machine will stop. Then the machine will ask tell the user it's done and ask for a button press to confirm to go to the resting state again.

Our machine also will have the ability to calibrate. This happens when you turn it on or when being in the resting state and pressing enter. Of course the user will be informed by the display. It will try and remove all the disks currently in the tube (for example the disks that are left after an abort) and reset the wheel to the correct starting position using a pressure sensor and the wheel itself.

When the machine is sorting and the user presses the start/stop button, the machine should finish its cycle and keep still after the current disk has been pushed to the correct side. It will return to the resting state. When the user presses the abort button, the program should terminate and the machine should immediately stop.

4.2.3 User constraints

1. The user shouldn't unplug any of the sensors or the motor while running the program.
2. The tube should be empty when booting the program. Mostly the program can handle it when this is not the case, however this should be avoided.
3. The user should make sure, the motor is connected to one of the upper slots of the brick and the sensors to one of the bottom slots.
4. The user should make sure the batteries inside the EV3 brick have enough power, either with the provided battery, a set of AA batteries or by connecting it to the charger.
5. The user should check if all the teeth of the wheel are still ok. While not running the program, the user should manually turn the wheel using the handle and give all the teeth a push inwards to make sure the wheel won't get stuck with its teeth to the bottom.
6. Calibration of the machine is necessary before the machine can start sorting, if the ambient light of the environment of the system changes significantly a calibration might be necessary before the system will function as specified in the *Requirements* (see 4.2.1). The calibration will be done by moving the sorting wheel around until a certain extension on the wheel passes the colour sensor. Note again that this must be done *before* a disk is entered.
7. The user is required to put the disks upside down (hollow side upwards) into the machine. This is to lower the chance of the disk getting stuck in the tube as well as counting the disk as one and not as two disks (due to their form factor).
8. The user is required to enter the disks one by one since the pressure sensor might otherwise detect two disks as one.
9. In principle all lighting conditions should work, since the sensor looks at the direct reflection of the object in front of it neglecting ambient light. However, to ensure stability, extreme lighting conditions should be avoided.
10. When the user presses the abort button he should remove the disk if the disk is blocking the wheel to rotate or solve any problem that the machine cannot solve with calibrating. After this he should let the machine do a calibration.
11. The machine needs to be upright correctly and the tube has to have an angle (most likely this will be fixed). This is due to the machine using gravity to function. Also, the machine shouldn't be exposed to high accelerations.

12. When inserting the disks, the user might want to give them an extra push downwards the tube using the provided stick to get them past the pressure sensor. However the user should be careful not the press the sensor another time since then the machine will detect two disks while there is only one in the tube.

13. The user shouldn't press the touch sensor itself.

4.2.4 Safety Properties

1. After pressing the abort button the machine should stop immediately when running (This will be around 5ms).

2. After pressing the stop button, the machine should stop when running after its current cycle.

3. When the light sensor detects a colour different than gray, black or white. The wheel is either not calibrated or a disk or object of a different colour has come out of the tube. The user will be informed and asked if he wants to continue the machine or re-calibrate.

4. When the counter has a value of $counter \leq 0$ and the light sensor detects the colour white or black, it should ask the user that an error has occurred and ask whether to continue or to stop the machine.

5. When the counter has a value of $counter > 0$ and the light sensor detects the colour gray, it should inform the user a disk has gotten stuck in the tube.

6. When the wheel gets stuck (so when we give the motor a certain speed but it doesn't detect a change in angle), the program should abort and inform the user.

4.2.5 Machine Interface

The only attributes connected to the EV3 are the pressure sensors, the colour sensor and the motor. Since we can just connect them to the EV3 without any hassle, part will explain a bit about how they actually work and what values they deliver. There are also a number of six buttons on the EV3 brick itself which we will use.

The pressure sensors give a value of either 1 or 0 depending on their state. There's one pressure sensor at the start of the tube that delivers a 1 signal whenever a disk is passing by. The other one is attached near the wheel and returns a 1 whenever the wheel is in its initial position, so when the extended bit on the wheel touches it.

The light sensor returns a value between 0 and 100 depending on which colour it detects. We use its colour-detecting mode for this.

The motor is multiple things. First and mostly it's used to rotate a rod, some gears and then of course the wheel. This can be done in various speeds, various acceleration and of course various time. Secondly the motor is able to perceive its current angle and whether it's blocked or not.

4.3 Design decisions

As you can see from the video in the conclusion, we've chosen for a tilted design with a sorting wheel to sort the disks.

We first thought about a vertical design, however, due to the design of the Lego pieces, we couldn't come up with a clean way to make a vertical tube without the disks flipping over. With a tilted tube, the disks can lie flat and slide downwards. The tube had to be extended a bit, this had to be done, since not all twelve disks actually fitted inside it. Also, at the end of the tube, we higher-ed a part half a piece, to lessen the times for the disk getting stuck. To have the tube tilted, we designed a stand that would make sure the tube got tilted, but also to en-stable the machine itself.

We did change a lot about the wheel during the process. We moved it up- and downwards, changed the color behind the disks from gray to red (to improve the calibration part) and added the rubber pieces to make sure no disks slide past the wheel. The teeth of the wheel that prevent the disks from going sideways were also changed a lot in the design. Eventually, after some heavy testing, we gave them a height of 2.5 Lego pieces. In the end, this seemed to us to be the most stable construction after testing.

We did think about having a rod pushing the disk either left or right, or even up or down, but we didn't know how to realize it with Legos in a stable way. That is why we didn't go for such a design.

Also, we had to add the two pallets on the side since otherwise the disks would go flying off. We attached them like this to have the disks bounce off and fall downwards in one of the two containers

We put the color sensor underneath the wheel to have the best possible readings. The wheel blocks off the ambient light and due to the sensor being so close to the disks, we can get accurate readings.

On the top we put the sensor in this way and narrowed down the hole to make sure the disks actually press it.

Finally we attached the motor like this to have the most stable construction and because in this way it was possible to reach the wheel with gears from the motor. This reaching tended to be a bit of a problem at the first try, but after moving the motor 1 upwards, this was a lot easier. For testing purposes we also added a handle for being able to manually turn the wheel.

4.4 Conclusion

To conclude, we have now stated the requirements of the machine. We have shown the use-cases, so what the machine should be able to do in different cases. We have noted the user constraints and explained the safety properties and finally we have defined the machine interface.

A video of the workings of the machine can be seen in the following link: https://goo.gl/BG1Xey.

Chapter 5

Software Specification phase

5.1 Summary

In this phase we define the design of our software, the software is based upon a Finite State Automaton. Since the EV3's input and output comes from or goes to multiple different sensors we split them into multiple groups. The FSA is based on a Mealy machine, and uses the input and current state to define the output and next state. Afterwards UPPAAL was used to proof the correctness of the FSA.

5.2 Introduction

The software specification talks about the desired behavior of our machine. This part describes what the program does without telling how it does it.

5.3 Inputs and Outputs

The first part is to determine what the input of our program is, and after a careful analysis we categorized the input of the machine into three parts:

- the color sensor - the most important input -, which determines the color of the tile that is currently on top of the sensor to be able to tell the motor in which direction it should spin. It also determines if there is no tile there. This aspect will be used in error detection.

- the touch sensor. It is attached at the beginning at the tube and every time a tile is introduced in the tube, the input will change from 0 to 1. Immediately after the tile has passed, the input will change back to 0. We will use this aspect in order to count how many tiles are there in the tube. Every time the input reads 1, we will increment the counter.
- the buttons are the last part of the inputs. We have two buttons: one for START/STOP, which starts the machine and stops it after completing the current cycle, and the ABORT button which does exactly as it sounds and stops the machine immediately. After the ABORT button has been pressed, human intervention is needed.

The second part is what comes out of those inputs, the outputs:

- the motor is the most important output, it's what determines how the wheel moves and in what direction: the motor is entirely dependent on the color sensor input and it is also dependent on the calibration
- the screen is where we output all the errors and how many tiles there are in the tube, how many tiles have been sorted, how many are black and how many are white. Basically, it tells the state in which the program is

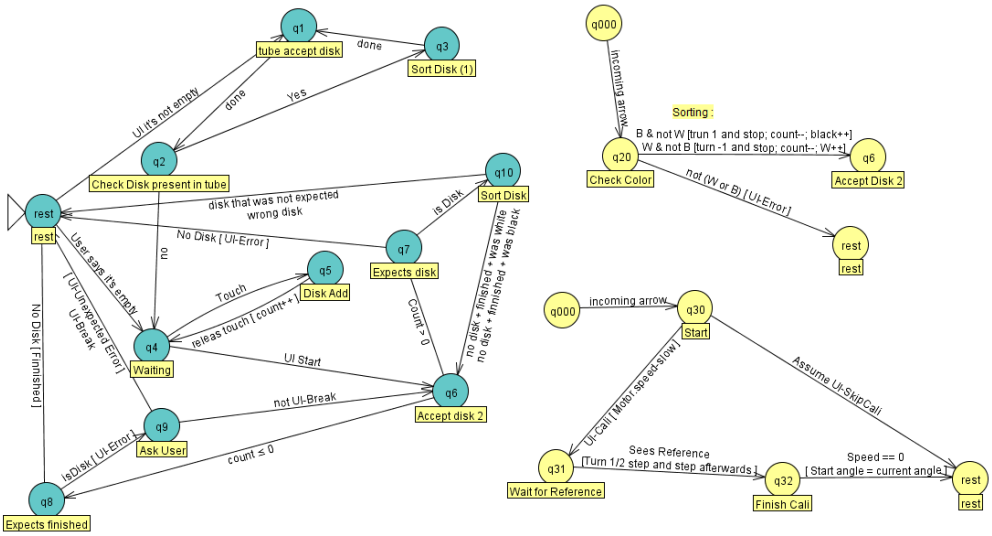

Figure 5.1: State Transition Diagram of the finite automaton made in JFLAP

5.4 Finite State Automaton

Now, the most important part of the program is how the outputs depend on the inputs, and for this part we will use a Finite State Automaton, which we constructed. The main finite state automaton is the one with the green circles. The yellow use are used for:

- the top one is used instead of the Sort Disk
- the bottom one is used instead of the Calibration

We decided to go with this implementation in order to not clutter the diagram.

Our software starts from rest, and then checks whether the disk tube is empty or not and proceeds accordingly to one of the two states that depend on this. If the tube is not empty, than the program sorts all the disks that are already in the tube and when all the disks are sorted, goes to the Waiting state. If there is no disk in the tube while in the rest state, the program expects it to be finished. If there is disk in the tube, the we have detected an error and the program expects the intervention of the user and than goes to Accept disk(if count is 0 or less then it will go to expects finished). The third part of the rest state is when the user specifically says that the tube is empty, the the program goes directly to the Waiting state. From the Waiting state, we start to add disks, when the touch sensor reads 1, we go to add disk and then, when it goes back to 0, the counter is incremented and the program goes back to the Waiting State.

Now, when the user presses Start, the program goes to Accept Disk, from which if the counter is bigger than 0, we go to the expects disk phase. Here, if there is no disk in the tube, we found an error and we go back to the Rest and if there are disks, we go on to sort them. Now we check the color of the disk. If it's white and not black than turn the wheel 1 time, increment the black counter and decrement the tiles counter to show that one was sorted. Then we move again to accept disk and we do this for every disk in the tube. From check color, if the color is not black and not white, then we detected and error and move to the rest phase. When there is no disk left, the program goes to rest.

The calibration phase is used to calibrate the wheel and it is called by the user. The user can either skip and the program goes to rest or choose to calibrate it and the program asks for a reference, calibrates the wheel and than goes to the resting phase, putting the speed at 0.

The calibration phase is not part of the final state diagram because at that time, the calibration was just an idea. So we will argue its correctness informally. This phase starts when the program is booted right before the Resting state, but it can also be called from the Resting state. We define a teeth as the part of the wheel the pushes the disk left or right. The first part of the program checks if a teeth is blocking the color sensor and thus the disks. In order to detect that, we decided to build the teeth with red pieces of Lego, making it easier to be distinguished from the disks. If that is the case, we turn the wheel half a teeth and go on to find the first calibration point. To find it, we just turn the wheel for half a teeth until one is found then go on to find the second one.

5.5 UPPAAL

In addition to the finite automaton given and described in the previous section(s) we have made a version of the finite automaton in UPPAAL. UPPAAL is a tool developed for the description of high-level state behavior. The finite automaton is however a simplification of the actual automaton. Our finite automaton is a Mealy Machine with a lot of string output (interaction with the user). We have omitted these things in the UPPAAL model since it would otherwise be a big mess. The UPPAAL model is here to help us with testing the system. We will go into further depth about this in the section System Validation and Testing (Chapter 8). There are however some transitions in the UPPAAL model that look superfluous, but they are there to show that each transitions will give a different output (message), even though it can be the case that we omitted these outputs. This is our UPPAAL model and as mentioned before we'll go into further depth into the model and the testing with it in chapter 8.

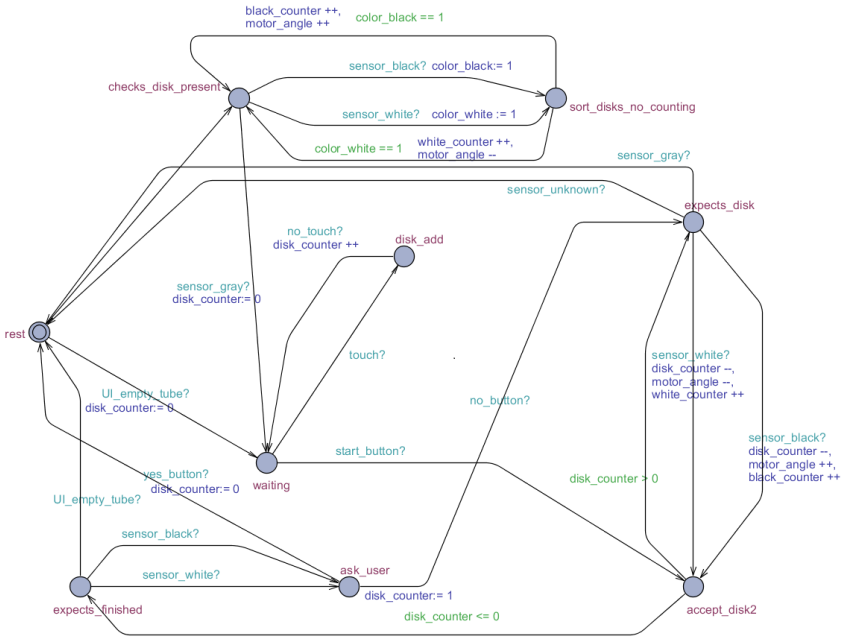

Figure 5.2: Our UPPAAL Model. It captures the most import things of what our software should do. It is thus a simplification of the actual finite automaton.

5.6 Conclusion

To conclude, this showed exactly how we plan to use the inputs in order to get the outputs using the Finite State Automaton.

Chapter 6

Software Design phase

6.1 Preface

In the project guide it was stated that we should construct a computer program that realizes the functions specified in the Software Specification deliverable during this phase/in this deliverable. The program should be written in (elementary) Java and does not even have to be compilable and executable. The idea is that this program written in elementary Java would serve as a stepping stone towards the Assembly Language program for the PP2 written in the next phase. The thing is that we are on of the LEGO MindStorm Trial groups. We are given a little bit more freedom of choice. Instead of the PP2 processor, we are given the EV3 brick (like mentioned before), and instead of the Fisher Technik kit, we are given the LEGO MindStorm kit.

In section 3.1.3 we have seen the various programming languages options for this EV3 brick. The EV3 comes by default with its own LEGO firmware and software. We have *chosen* to flash other firmware onto the brick so that we can make use of leJOS (instead of the drag-and-drop interface that comes with the official firmware). *The reason we have chosen* to program the sorting-machine in leJOS is because leJOS is Java based, and this gives us more options and freedom programming wise and allows more complex programming structures. This is however a little bit in conflict with the description of contents of the software design deliverable as stated in the project guide. Since it is the case that with the firmware replacement leJOS a Java virtual machine is included, we are able to program in Java. There are leJOS plugins for some Java IDE's like NetBeans and Eclipse (we are going to use the latter one). This is the reason of the conflict. The project guide states that during this phase the program should be written in (elementary) Java, *but since our actual source code for the sorting machine (see Software Implementation and Integration)* is written in Java, it

seems a little bit overkill to write the program first in elementary Java during this phase, and to write the source code in Java again during the next phase. We have *therefore chosen that we* are not going to write the source code first in elementary Java, but to do it in pseudo-code (checked with the tutor first if this was ok). This will then serve as a stepping stone to the actual source code, written in Java.

Since the implementation of our program is so close to the actual finite state automaton, the correctness of the program is mostly proven by the correctness of the finite automaton which we have seen in software specification and will be further discussed in the section System Validation and Testing.

6.2 Summary

The design of the software specifies which functions are needed. The functions are grouped based on function, for each group it is specified which other groups they are allowed to call (Structure, section 6.3). Every group of functions has multiple functions already worked out in pseudo-code, to create a guideline for the general structure of the functions in that group, to aid the implementation during the Software Implementation and Integration phase (see chapter 7).

6.3 Structure

Since we are not going to give an elementary Java program in this section, we provide some pseudo-code to give a nice overview of how the actual source code that will be written in Java should be structured and implemented. We have seen the finite automaton of our program at the previous section. We are going to try to stick to it as much as possible when writing the actual source code. To do this we need a nice structured way of implementing the automaton into software that simulates its behavior. The different components that we will actually need to simulate or to implement a finite automaton are listed down below. Our finite automaton is a Mealy Machine, which means that the output values are determined by both the current state of the automaton and the current input. To simulate we are going to use the following components:

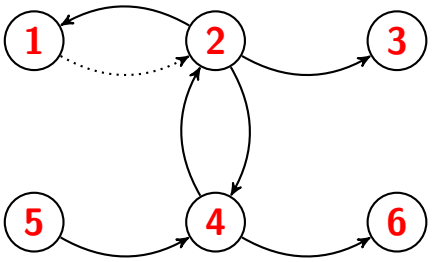

Figure 6.1: Software structure. The solid edges represent calls to a different class of functions. The dotted edges are there to show that we make a call to obtain data from one set of functions to use in another set of functions.

1. Condition functions
   Functions that check whether certain conditions (mainly boils down to checking input) are met. For example is the yes-button pressed, or is the reading value of the color sensor within a certain range.

2. Decision functions
   These functions are basically the transitions in the automaton that occur based on certain conditions (see condition functions). And since our automaton is a Mealy Machine. The output is only dependent on these conditions and the current state the automaton is in.

3. Output functions and state variables
   The output functions and the functions for the state variables are being called when certain conditions are met in certain states. The decision functions thus decide to which state we proceed (or that we should stay in the same state) and what the output functions and state variable functions should be called in this transition.

4. Update
   This is an update function of the states. It makes sure to update the states and the message on the screen of the brick

5. Initialization
   We calibrate the wheel first and initialize the input before we run the actual state machine.

6. Termination
   The program is terminated by the user of the sorting machine.

We are going to explain the different components of the program in more details below.

6.4 Condition Functions

The condition functions check whether certain conditions are met. We are going to be using condition functions to check the following things:

- to check whether the color sensor detects red
- to check whether the color sensor detects white
- to check whether the color sensor is detects black
- to check whether the push sensor at the beginning of the tube is pressed
- to check whether the start/stop button is pressed
- to check whether the yes button is pressed
- to check whether the no button is pressed
- to check whether there is no disk at all in front of the touch sensor
- to check whether one of the red teeth is in front of the touch sensor (for calibration purposes)

These are all based on inputs of the sorting machine.

The listed functions are all boolean functions. They will thus return true or false, based on whether a certain condition is met. In the decision functions we then call these condition functions, and the condition function will return a boolean value. We can then use this boolean function in the guard of an if statement for example. To give you an idea of how we have structured the condition functions, I am going to give some pseudo code of some of these functions.

pseudo-code/condition_functions.java

```
1 // return true if the disk in front of the sensor is
    white.
2   boolean function colorSensorWhite {
3   return true, if and only if the value given by the
     color sensor is within a certain range that is
     specific for the color white
4  }
5
6 // return true if the disk in front of the sensor is
    black.
7   boolean function colorSensorBlack {
8   return true, if and only if the value given by the
     color sensor is within a certain range that is
     specific for the color black
```

```
 }

// return true if the start/stop button is pressed,
    false if it isn't.
  boolean function buttonSSDown {
  return true, if and only if the enter button is
    pressed down
 }

 // return true if the yes button is pressed, false
    if it isn't.
 boolean function buttonYesDown {
   return true, if and only if the right button is
    pressed down
 }

// return true if the no button is pressed, false if
    it isn't.
 boolean function buttonNoDown {
  return true, if and only if the left button is
    pressed down
 }
```

6.5 Decision Functions

The decision functions basically represent the transitions between the states in the finite automaton. These transitions are different for each state, and the behaviour depends on the values returned by the condition functions. For example when we are in the resting state we ask the user whether the tube is empty or not. If the user then presses the yes button the disk counter will be set to zero (state variable) and we proceed to the Waiting state. If the no button is pressed, we proceed to the state CheckDiskPresent without calling any output functions or functions for the state variables. In addition if we insert disks in this state (check condition function that checks whether the push sensor at the beginning of the tube is pressed), then we proceed to the InsertedEarly state. If none of these conditions (checked by condition functions) are met, we stay in the resting state. In pseudo-code this will look something like:

pseudo-code/rest_state.java

```
Rest {
   if (yes button is pressed) {
   // check return value specific condition function
    call state variable function that sets the
   disk-counter to 0;
    return Waiting;
    // proceed to the Waiting state
   }
   else if(no button is pressed) {
   // check return value specific condition function
    return CheckDiskPresent;
    // proceed to the CheckDiskPresent state
   }
   else if (the touch sensor is pressed) {
   // check return value specific condition function
    return InsertedEarly;
    // proceed to the InsertedEarly state
    }

   return Rest;  // remain in the resting state
  }
 }
```

Here are some more examples of decision functions that we are going to implement in the next phase. The comments in the pseudo-code will explain the construction and structure of the decision function.

pseudo-code/inserted_early.java

```
InsertedEarly {
   call output function that outputs a message on th
   screen that notifies the user that disks were
   inserted too early;
   if (no button is pressed || yes button is pressed
   || start/stop button is pressed) {
    // check return values specific condition
   functions
    return CheckDiskPresent; // proceed to
   CheckDiskPresent state
   }
   return InsertedEarly; // remain in the
   InsertedEarly state
  }
 }
```

The next example of a decision function is the ExpectsDisk state. This is one of the states where the actual sorting happens. There are some comments added to explain the underlying structure etc.

pseudo-code/expect_disks.java

```
ExpectsDisk {
   if (the color sensor detects white) {
   // check return value specific condition function
    call the state variable function that decreases
   the disk counter;
    call the output function that rotates the motor
   in the right direction (white disk);
    call the state variable function that increases
   the white counter;
    return AcceptDisk2; // proceed to the AcceptDisk2
   state
   } else if (the color sensor detects black) {
   // check return value specific condition function
    call the state variable function that decreases
   the disk counter;
    call the output function that rotates the motor
   in the right direction (black disk);
    call the state variable function that increases
   the black counter;
    return AcceptDisk2; // proceed to the AcceptDisk2
   state
   } else if (the color sensor detects no black and
   no white) {
    call the output function that notifies the user
   that there is a disk stuck in the tube;
    if (the  start/stop button is pressed) {
      // check return value specific condition
   function
     call the output function that asks the user if
   the tube is empty;
     return Rest; // go back to the resting state
    }
   } else {  // If the color sensor is indecisive.
    call the output function that notifies the user
   that another color is detected;
    if (the start/stop button is pressed) {
      // check return value specific condition
   function
     call the output function that asks the user if
   the tube is empty;
```

```
26        return Rest; // go back to the resting state
27       }
28      }
29
30      return ExpectsDisk; // remain in the current state
31     }
32    }
```

6.6 Output functions and State Variables

The output functions and the functions for the state variables are being called when certain conditions are met in certain states. We have seen some examples of this in the previous section (the decision functions). If you look at the last piece of pseudo-code (of the ExpectsDisk state). The decision functions thus decides to which state we proceed (or that we should stay in the same state), and what the output functions and state variable functions should be called in this transition. We have output functions that simply send messages to the screen to notify the user or to ask the user something. At the same time we make use of the led-lights of the button. Wen we send a message we immediately set the color of the led-lights. When there is an error related message we make the light red. In addition the we make the light blink at a high frequency. When we ask for user-input we make the light orange and make it also blink, but at a lower frequency than with the error-related red light. At last we have screen messages that are output when the machine is just busy doing something (and everything goes fine). We then just make the light of the buttons green (no blinking). You can see some examples of output functions for the messages and the blinking of the lights below.

pseudo-code/output_functions_messages.java

```
output function stuckInTube{
  output message: "Earlier done than expected, disk
   stuck? Press enter to dismiss.";
  setLEDState: error;  // red light + high frequency
   blinking
}

output function enterToSort {
 output message: "Press Enter to start sorting.";
  setLEDState: userInput; // orange light + blinking
}

output functon isCalibrating {
  output message = "Calibration in process.";
  setLEDState: busy // green light }
```

We have furthermore output functions that notify the motor in what direction it should move, how far it should move and at what speed. We furthermore have some output functions that are specific for the calibration of the wheel. We calibrate the wheel so that the disks can fall exactly in the gaps of the wheel. You have read the workings of the calibration in the software specification part. You can see some examples of output functions related to the motor (rotations) and the calibration below.

pseudo-code/output_functions_motor.java

```
// When the color sensor detects a black disk, turn
    one teeth right.
output function motorSortBlack{
  motor.rotate turn one teeth right
 }

// When the color sensor detects a white disk, turn
    one teeth left.
output function motorSortWhite{
  motor.rotate turn one teeth left
 }

 // Let the motor turn a small step.
 output function motorTurnSmallStep(boolean left) {
  if(left) {
   motor.rotate.right (smallStepSize)
  } else {
   motor.rotate.left (smallStepSize)
 }

 // Let the motor run for half a teeth.
 output function motorTurnHalfTeeth(boolean left) {
  if(left) {
   motor.rotate turn half a teeth right
  } else {
   motor.rotate turn half a teeth left
  }
 }
```

We have various state variables that we use in our finite automaton based program. We have state variables for the disk counter, the black disk counter, the white disk counter and calibration points. We have functions for these state variables that set the counters to 0, functions that increase the counters, functions that decrease the counters, and functions that give the current values of the counter. We will again give some examples of these functions.

pseudo-code/state_variables.java

```
// increase the diskcounter
state variable function increaseCounter {
  diskCounter++;
 }

 // decrement the black disk counter by one.
 state variable function decreaseCounter {
  diskCounter--;
 }

 // Increment the white disk counter by one
 state variable function increaseWhiteCounter{
  whiteDiskCounter++;
 }

 // Reset the disk counter.
 state variable function setCounterToZero {
  diskCounter = 0;
 }


 // Return if disks are expected in the tube
 state variable function counterGreaterThanZero{
 // boolean function that returns true when
    diskcounter is greater than 0
    if (diskCounter > 0) {
      return true;
    } else {
      return false;
    }
  }

 //   SCREEN OUTPUT
 //   Return the value of the white disk counter.
 state variable function getWhiteDiskCount {
  return whiteDiskCounter;
 }
```

6.7 Update

We furthermore have an update function of the states. It makes sure to update the states and the message on the screen of the brick. We also have implemented the stop button in the update function. When this button is pressed we go back to the resting state. This was not part of our finite automaton since it would be a little bit useless to add a stop transition from each state to the resting state. The finite automaton (the UPPAAL model for example) is in that way a simplification of the actual finite automaton implemented in our source code. Because the finite automaton would otherwise would be very complex and you would have no overview at all.

pseudo-code/update_function.java

```
while(the abort button is not pressed) {
   update the current state;
   update the message drawn on the screen;

   // Implement the STOP button.
   if(the stop button is pressed) {
    go to the resting state;
    output a message to notify the user;
   }
  }
```

6.8 Initialization and Termination

Before we start running the actual program (that sorts the disks) we initialize the input (sensors etc.) and calibrate the wheel so that one of the five gaps where the disks can fall is parallel to the end of the tube. The code for the initialization is very java specific and includes a lot of things of the EV3 leJOS library. I will therefore not include any pseudo-code of this. In the next section (Software Implementation and Integration and the appendix) you can have a look at the code that initializes the input. We furthermore have the abort button that terminates the whole program. This button was sort of already implemented in the EV3 brick. When a program is running and you click the upper-left button (the one that is right under the screen), the program will be terminated and as a result everything will stop immediately. Since this is exactly what the abort button should do of our sorting machine, we decided to just make this the abort button and don't introduce another button as the abort button that would do the same thing. After you have pressed the abort button you can restart the program however (immediately or in the future) and the program will start from scratch. If there were still some disk left in the tube of the last run of the program, this is no problem, since this is taking care of in our program.

Chapter 7

Software Implementation and Integration phase

The source code created in this phase can be found in Appendix A: Source code.

7.1 Summary

In the last construction phase the software designed in the Software Design phase (see chapter 6) is translated into Java in combination with the leJOS library. A Coding standard (see section 7.4) was developed to make development easier and the code more consistent. When the FSA was implemented Additions (see section 7.5) - all within our Requirements (see subsection 4.2.1) - where made, to further extend the functionality or implement things which where not present in our FSA.

7.2 Structure

Since we are directly writing Java we first did some research into the implementation of Finite State Automaton into Java. We found a couple of examples and picked one which used an *enum* to switch between different implementations of a State class. This has the advantage that only one state can and will be active at once. This implementation of the State class has one update function, which get's called by a loop in the Main method, this function decides what the next state should be and what the output is. Since we are implementing a Mealy machine this works fine.

To handle input and output they both have their own classes. The update function of the states only calls functions of an object of both of these. These input and output classes consist of implementations of the input and output variables and methods from the Finite State Automaton, they also contain some other functions to support the input and output functions, these are all private to prevent the user from calling them.

This structure separates the input from the output and prevents any direct contact between these classes. *This caused one problem* for the counters, since a counter can be input, when it's for example read, or output when it's altered. *To solve this* an extra class is defined to which both the input and output object have access to, this state variables class stores all counters. Access to this object is not granted to the states directly, since they can only work with input and output.

The result is not a perfect implementation of a FSA, since transitions aren't instant, but this allows us to almost implement the FSA almost directly.

7.3 LeJOS

For our implementation we choose for LeJOS since it's free, seemingly complete, and Java based. Our experience with LeJOS was limited to non existing, only one of our group had used it before - briefly during the carnival brake - but on a NXT in stead of the EV3. Therefore there was still a lot to learn and explore for all of us.

7.3.1 Installation

Since LeJOS is third party software we still had to install it. It shouldn't have been that hard, but it still took an entire afternoon to get it working. Preparing the micro-SD card went fairly simple since the leJOS documentation has their own guide on how to do this. The first problems appeared when it didn't seem to detect the files on the card and we wanted to take it out. This turned out to be a bit harder than you would expect, the micro-SD slot had no spring to push the card out and there is to little space to pull the card out with your fingers. The internet isn't conclusive about this problem but the solution was to use pliers to get it out. We eventually found a workaround by using the point of some cutlery to retrieve it.

When we overcame this difficulty the next obstacle appeared to be the micro-SD card itself. Although this wasn't clear since it functioned without problems in Windows, but it gave read and/or write errors during the installation. That it was the card wasn't clear from the messages since the install error message was really generic. The internet wasn't really helpful in this situation either, especially when you have to filter between all the different versions of leJOS, and after a lot of different attempts we decided to try our luck with a different micro-SD card. This one didn't gave any problems at all and 10 minutes later leJOS was running on our brick.

Before we could actually run some sample code on our machine we first had to get it on there. The instructions of leJOS to achieve this where scattered over a couple of pages so it took some time to get it working. LeJOS uses a ethernet adapter over USB to push your programs on it, later on this caused problems since IP addresses weren't issued which made connection impossible. A solution is to give both devices static IP addresses, but since every USB port get's a different adapter we had to do this a couple of time, which can be quite frustrating.

7.3.2 The Library

The leJOS library is easy to use and nicely structured. It suited all our needs, although the lack of examples and sometimes even documentation did slow us down, there where multiple occasions where we had to look through the source code of the classes to find out what they offered and how they should be used. Luckily most simple things, like for example turning a motor or drawing text on the display, can be achieved with a single line of code, and there are plenty of examples for them. When we started to get the hang of it the experience improved a lot, all the features of Java and the leJOS library, in combination with the FSA structure made debugging and implementation of extra features a lot faster and easier.

The Eclipse plugin is worth mentioning, installing it is easy and makes the life of the developer a lot easier. It allows the content assist to show their documentation, which saved us a lot of time. Also running your program on the EV3 is now possible - once setup - with one button click which takes care of compiling, uploading, and launching of your program. It also features a debug mode which should allow usage of breakpoints and other Eclipse Debug features. All of this made our general experience a lot easier since it really supports the developer.

7.4 Coding standard

Since we could develop directly in Java we had no need to define any conversion standards to Assembly, but we had to standardize a lot of other things. First of all we agreed to keep the coding standard of the programming course in mind.

- The most important thing for our coding standard is that the earlier explained structure is honored. Example of abuse of the structure would be using input during output functions, which should actually be handled by the state which uses that output function. Only in some exceptional cases when we as a group cannot come up with a proper solution or it results in a significant simplification of the code we allow minor changes to the structure. An example for this is the use of the delay class which prevents the implementation of an extra state which stalls the program.
- Naming of states and public input and output functions should be similar to, or the same as, the names in the FSA.
- Comments should be used for all code that isn't directly derivable from the FSA.
- All functions should have their access level modifier set accordingly and have a description of the function. For functions with multiple parameters explicit description of these is strongly recommended.
- Functions may only be added if they have a direct need, this allows placeholders but prevents the creation of functions which are unused at creation.
- Code is not allowed to be directly copied from the internet. Rewriting code examples to our needs is preferred, although exceptions can be made as a group if it would result in a significant time saving and its usage would improve the final result.
- Extra classes should only be imported when used, and removed when obsolete.
- Related functions should be placed in groups, each group should have it's own caption to show where one group starts and the previous one ends. An example of a group can be all output functions which allow motor control.

7.5 Additions

A couple of things where added during this phase after we finished the development of the sorting machine. These made the machine easier to operate and more efficient.

7.5.1 Automatic wheel calibration

Automatic wheel calibration was our original plan but got pushed to the end of the implementation phase, because we first wanted to get the sorting itself stable, problems where caused by the color sensor which had problems detecting the difference between a close black disk and further away gray wheel. After changing the color of the wheel to red this became a lot easier and afterwards we started looking into implementing the calibration. This wasn't in the initial UPPAAL model to decrease the complexity, the calibration itself was something which we already created during the design of the FSA, therefore implementing the states wasn't that much of a problem, calibrating the input values of the color sensor eventually caused a lot more problems since their value can be quite unstable.

7.5.2 Sensorport detection

Support for using an arbitrary sensor port for both sensors, and the same for the motor, was one thing we added later on. This caused quite some problems since we first wanted to find the location of the sensor every time we polled the sensor. The lack of documentation made it quite an exploration before we came up with a full implementation for our sorting machine. This relied on two classes which had to access each port, only one of both classes could have an open connection with a port at the same time. Opening and closing a port caused so much time that it really slowed down the machine. Therefore we now only detect in which port everything is at the start of the program. Now it does not detect sensors and motors which get disconnected in real time. Which shouldn't really be a problem since the connectors of the cables aren't easy to disconnect. So it seems quite impossible to achieve disconnecting a motor or sensor by accident.

7.5.3 Motor stall detection

This was not in the original design but implemented at the end. This adds a state which is not described in the UPPAAL model, but since it is a simple state which only function is to show a message and wait for the user to dismiss this we assume it trivial. Without this detection a stalled motor will just keep on trying to achieve it's target angle until it reaches it. The motor is designed for this, but it gave the nasty result that after a couple of seconds of stagnation still trying to reach it's target angle seemed silly and gave the machine a 'dumb' impression.

After implementing the stalling detection we now check if the motor is stalled, and if it is the program switches to the motor stalled state. In which the user is asked to resolve the problem, afterwards the program goes back to calibrating and after this the machine should be operational again.

7.5.4 Enhanced UI

What we would specifically communicate to the user wasn't specified beforehand. During the process of designing the machine we came up a lot of things which where useful for debugging, now most of them are gone since they are - or should is maybe better - not necessary for the end user. This doesn't mean that their influence is entirely gone. Showing the counters and the current state where things which where first of all used for debugging and checking our program, but managed to make it into the final UI of the machine. Over time the layout of the UI has evolved to be more compact to get multiple space for multiline messages. Later on we coupled the LED status on the intention of the message drawn on the screen. They are divided into three groups: machine busy, in need of user interaction, and error, each with their own color and flashing rhythm.

Chapter 8

System Validation and Testing phase

8.1 Summary

The last phase focuses on guarantying the quality of our product. To achieve this Code Review (see section 8.2) and Test Cases (see section 8.3) where used.

Code Review intents to improve the quality of the code and minimize the amount of bugs. Test Cases aims at checking if the machine functions achieves all Requirements (see subsection 4.2.1), and stays within their limits.

8.2 Code Review

Here follows some code reviews we have done while writing the code. Of course there have been a lot of changes to the code individually, but in these phases we checked up on each others code:

- ***Pair Programming:*** *March March 16th, Adriaan and Abdel*

  Here Adriaan and Abdel were working together on improving the code overall, removing dead code, improving some messages, rewriting some states just to improve clarity and functionality. Also the detection of a stalled motor was added by Adriaan and approved by Abdel at this time.

- ***Pair Programming:*** *March 9th and 15th, Jolan and Adriaan*

  This pair programming was spread out across two days where Jolan on the first day found out the commands, wrote the most part and Adriaan on the second day corrected and fixed the problems Jolan faced. We wanted to add feedback from the LEDs to the user about what the machine is doing. Green being 'busy', orange being 'require attention' and red being 'error'. However Jolan ran into some problems where red wouldn't flash as intended in some states, Adriaan fixed this and cleaned up the code.

- ***Pair Programming:*** *March 8th, Ardiaan and Jolan*

  We found out we didn't have any detection for when a user entered a disk before the machine was actually in the correct state. So we made an extra state from the Resting state to the flushing part of the machine, so that when a user enters a disk too early (so in the Resting state), the machine will inform that it is too early and will prompt to empty the tube. The programming part mostly was done on Jolan's laptop while Adriaan was giving input and helped cleaning up.

- ***Walktrough:*** *March 4th, Adriaan, Abdel and Jolan*

  We added a way to go through the program step by step by pressing the up-button (later called the stop-button). In this way we could exactly view how the machine operated in each state and if it malfunctioned or displayed something weird, we changed bits and pieces of the code to solve this. For example, we here found out we didn't yet have a white and black disk counter on the screen, so we added this in the Output file (this was done on Adriaans laptop). We also noticed that not everywhere camel case was used for the method names, so we fixed that too.

- ***Pair Programming:*** *March 4th, Adriaan and Abdel*

  We were implementing the motor which needed some tweaking to have it turn exactly the amount of degrees to push off the disk in the right direction and end up in the correct position for the next one. First Abdel made it 216° and created the variable *turndegrees* for clarity. While this was going on Adriaan was watching and giving input to Abdel. Next, Abdel changed *turndegrees* to 288°, after redoing the calculations, however 216 appeared to be working better so he reverted.

- ***Pair Programming:*** *March 2nd, Bogdan and Jolan*

  Bogdan created the Output file and made it so that it contained some error messages we might need in the future. Jolan observed and co-operated and later on, while he was working on the States file, he added some more messages that were relevant for that state. Bogdan in his turn watched and agreed or disagreed on those messages.

8.3 Test Cases

Orientation phase:

- Testing light/color-sensor readings in general:
  We were wondering what our possibilities are, in detecting objects and colors with the one light/color-sensor we are given.
  So we tested what the light/color-sensor can detect and distinguish and also the distance at which it sill functions properly.
  From our tests we could conclude that:
  The light/color-sensor can detect the intensity of red, green and blue in front of the light/color-sensor as well as just object detection over a distance of a few centimeters. It can easily distinguish the white disks from the black disks. However, detecting the difference between the black disks and the gray LEGO components is less clear.

- Testing at which point the pressure-sensor gives the signal for pressed:
  We were wondering how sensitive the pressure-sensor is.
  So we tested at what point the pressure-sensor gives the signal of being pressed and get a feeling of the amount of force needed to get to the pressed state.
  From our tests we could conclude that:
  The pressure-sensor is clearly not sensitive enough to be triggered by the weight of the disks.

Machine Design phase:

- Testing light/color-sensor readings on LEGO components, that could be used for constructing the sorting wheel, in comparison to the readings we get from the disks:
  We were wondering if the light/color-sensor can distinguish the LEGO components from the disks.
  So we tested different colors of LEGO at different distances to see if we can detect them as not being a disk.
  From our tests we could conclude that:
  It is possible, but up close to the sensor the difference between gray LEGO parts and black disks is to small to be reliable.

- Testing the functionality of our tube design:
  We were wondering if the design of our tube works properly and if it was the right choice.
  So we tested a few designs. We first tested a few designs with the tube vertically up. Then we tested several designs with the tube at an angle.
  From our tests we could conclude that:
  It is hard to make a vertical tube with LEGO in which the disks won't flip into an unfavorable position. For a more horizontal tube to work, the angle between the tube and the ground should be great enough that gravity is enough to get all disks to the bottom without one getting stuck on the way.

- Testing different ways of how to count the amount of disks inserted into the tube:
  We were wondering what would be a good way of counting the amount of disks that we insert into the tube.
  So we tested a few designs with a pressure-sensor at the entrance of the tube.
  From our tests we could conclude that:
  For a tube design with a pressure-sensor at the entrance, you should make the height of the tube entrance small enough that you need to force the disks past the pressure-sensor to get it into the tube.

- Testing different ways of pushing disks into the tube:
  We were wondering what is a good and reliable way of inserting disks into the tube.
  So we tested several cases:

  a. A case where a disk gets pushed past the pressure-sensor with the next disk.
  b. A case where a disk gets pushed past the pressure-sensor using ones finger.
  c. A case where a disk gets pushed past the pressure-sensor with a stick made from LEGO.

  From our tests we could conclude that:

  1. You should not use the next disk to push the previous disk past the sensor, since it will not release the pressure-sensor in between, counting a hole batch of disk as 1 insertion.
  2. You better not use your finger to push a disk past the pressure-sensor, because there is a decent chance that you accidentally press the pressure-sensor an extra time.

3. It is advised to use the LEGO stick to press the disks past the pressure-sensor, since that is the most reliable way to insert disks while triggering the pressure-sensor the right amount of times.

- Testing how the system handles disks that get inserted with the open end down instead of up:
  We were wondering if inserting a disk with the open end up or down would make a difference.
  So we tested inserting a few disks with the open side down and a few disks with the open side up.
  From our tests we could conclude that:
  The systems works even if you insert disks with the open side down, although that would increase the change of troubles. Putting disks in with the the open end up is the more reliable option.

- Testing different designs of the sorting wheel:
  We were wondering what would be an efficient design of the sorting wheel
  So we tested several designs over the project.
  From our tests we could conclude that:
  We need a wheel of which the light/color-sensor can detect the LEGO parts between the openings for the disks and distinguish that from the disks itself. This doesn't work well with gray LEGO parts.

Software phases:

- Testing motor capabilities while integrated into the system:
  We were wondering what the capabilities of the motor are once integrated into the system.
  So using code, we tested the possibilities of the following actions.

  a. Testing the maximum velocity of the sorting wheel, by letting the motor draw the maximum power it can get from the battery.
  b. Testing the smallest angle the sorting-wheel can turn, by letting the motor turn with the smallest angle it can.
  c. Testing if the motor could detect small angle changes.
  d. Testing the amount of force the motor delivered on the wheel by blocking it with a finger.

  From our tests we could conclude that:

  1. The maximum velocity is dependent on the amount of charge in the battery.

2. The motor can turn at a very small angle. Resulting in an even smaller rotation of the sorting wheel, because of the gears between the motor and the wheel.
3. The motor is very capable of checking if it indeed turned the amount of degrees it should have turned.
4. The motor can deliver more than enough force than is needed to move the disks around.

- Testing light/color-sensor readings while sorting with our current machine design:
  We were wondering if the way placed the light/color-sensor in the system and the current sorting wheel design, can cause problems with reading the color values of the wheel.
  So we performed several tests to rule out the problems. From our tests we could conclude that:
  The, into the system integrated, light/color-sensor can correctly tell apart the colors of the wheel and the disks. Which are red, less red, black and white.

- Testing if all disks of the same color, out of black and white, get sorted to the correct side:
  We were wondering if it can happen that, after sorting there would be a black disk among the white disks or a with disk among the black disks.
  So every time we made significant changes in the design our code, we tested sorting all the disks.
  From our tests we could conclude that:
  In all test cases, where all disks got sorted without getting stuck, all disks got sorted to the correct side.

- Testing the display of the EV3 brick:
  We were wondering what we can display on the display of the brick and if the EV3 brick would display the right text when certain situation like errors would occur.
  So every time we improved the UI and when we added something like an error message or a counter, we tested and observed if all the characters and drawings were correctly displayed on the display of the brick.
  From our tests we could conclude that:
  Everything we put on the display was displayed exactly as we expected it to.

- Disks in the tube while not yet calibrated:
  We were wondering how the system would behave if there are disks in the tube while the wheel has not yet been calibrated.
  So we tested running the calibration code while there are disks in the tube.
  From our tests we could conclude that:
  The system can calibrate with disks in the tube, but it is advised not to do so, because it is unreliable.

- Testing the systems behavior when the pressure-sensor gets triggered more or less often as disks get inserted:
  We were wondering how the system would behave if you trigger the pressure-sensor a different amount of times than the amount of disks that will be in the tube. In other words, making the disk counter variable contain a different number than the actual amount of disks in the tube.
  To establish what this behavior is, we ran a couple of tests:

  a. A situation with "$diskcounter == 0$ while there are disks in the tube", by having a filled tube before running the insertion code.
  b. Situations with "$diskcounter <$ amount of disks in the tube", by already having a few disks in the tube before running the insertion code. once with a difference of 1 and once with a greater difference.
  c. A situation with "$diskcounter <$ amount of disks in the tube", by pushing a few disks in right after each other without releasing the pressure-sensor between them, while the insertion code is already running.
  d. Situations with "$diskcounter >$ amount of disks in the tube", by manually pressing the pressure-sensor a few times while the insertion code is running. Once with $diskcounter <= 12$ and once with $diskcounter > 12$.

  From our tests we could conclude that:
  Our machine gives the correct error messages in each of the mentioned situation above.

- Disk stuck on the pressure-sensor (as in keeps giving the signal for pressed):
  We were wondering how our system would behave if we would leave a disk stuck on the pressure-sensor instead of pushing it past it. So we tested it in combinations of the following situations:

  a. Without disks in the tube or with already disks in the tube.
  b. Already having the disk in place before the test starts or putting the disk in place during the test.
  c. Before, during or after running either the code for calibration, the code for flushing, the code for disk insertion or the code for sorting.

d. in the transitions from one part of the code to another, where that is possible with a disk on the pressure-sensor.

e. during or after an error.

f. In case of user input, one test for each possible button to press.

From our tests we could conclude that:
When in the waiting state, the machine just got to the state where a disk was added. When somewhere else, nothing happened until the resting state, there the machine reported that the user inserted a disk too early.

- Cables in different ports:
  We were wondering how the system would behave if we put the cables in different ports than we would usually use.
  So we tested inserting sensor cables in other input ports and the motor cable in another output port. After that we tested inserting sensor cables in output ports and the motor cable in an input port.
  From our tests we could conclude that:
  Putting a sensor cable or the motor cable in a different port gave errors. So after this we added the support for inserting cables in different ports and after that it only gave an error when, a sensor cable is plugged into a output port or when the motor cable is plugged into a input port.

- Testing the behavior of the system when Cables get pulled out, before and during execution:
  We were wondering how the system would behave if we would plug out a cable before our during the execution of our program. We were wondering if it Would give an error, just keep repeating its last action or if it would it just stop abruptly.
  So we first tested how the system behaved if we started it up with one or more cables plugged out. After that we tested unplugging either one cable or unplugging combinations of cables, during the execution of each type of executable code.
  From our tests we could conclude that:
  At first it would just give an error right away, but after we implemented the part of code for checking which port is connected to what, it ran without problem. Assuming you put the motor in one of the output ports and the rest in some of the input ports of the brick. If you however would plug in the motor into an input port or one of the sensor in an output port on the brick, it would give an error.

- Testing if all the disks will always be caught by one of the containers:
  We were wondering if after sorting or flushing, all disks landed in the containers and not just fell on the floor. For this we also needed to catch the disks that were more forcefully launched from the sorting wheel.
  So we first tested the paths the disks would follow when they leave the sorting wheel and see if these paths where consistent. Since the materials we received didn't include containers we first did tests using plastic cups. We placed them at different distances and different angles. After that we tested the systems with our current containers. We also tested if the wings would stop launched disks so they would nicely fall into the containers.
  From our tests we could conclude that:

  1. The path that the disks follow when leaving the sorting wheel is not consistent. Disks can fall very close to the wheel, but in some cases even get launched slightly upwards.
  2. If we would want to use only containers to catch the disks, without the wings, we would need to hold the containers ourselfs by hand our build a quite complex mounting system to be reliable.
  3. Our initial wings also needed to be blocked from above to be reliable.
  4. After some adjustments to the wing, our containers work reliably if placed close enough to the sorting wheel, with the open end facing towards the sorting wheel.

- Testing all interaction between the user and the EV3 brick, when asked for input:
  We were wondering if the behavior of the EV3 brick is what we expect it to be, when it asks something to the user and gets a replay from the user.
  So we tested all situations in which the user is asked for input and all possible inputs the user can give in those situations.
  From our tests we could conclude that:
  It behaves as we expect it to.

- Testing all interaction between the user and the EV3 brick, when there is no request for input from the user:
  We were wondering how the system would behave if the user presses the abort or stop button. We were also wondering if pressing some or all the other buttons on the brick, while there is no input request, would influence behavior the system.
  So we tested pressing the abort and stop button in as many situations as possible and we tested pressing the other buttons while the brick was not waiting for user input.
  From our tests we could conclude that:
  It behaves as we expect it to.

- Testing if all types of the error detection get triggered, when and only when, they are expected to happen:
  We were wondering if all types of error detection we have, can actually be triggered in the situations where they are expected to happen. We were also wondering if there is a situation where an unexpected error detection could happen.
  So we tested all the situations for which we wrote error detection code. We also tested if any unexpected error detection would appear during other test cases.
  From our tests we could conclude that:
  All types of error detection happen in the situations where they should happen and only in those situations.

- Testing the behavior of all the errors:
  We were wondering if all the possible errors behave in the way we expect them to behave.
  So we tested all the errors.
  From our tests we could conclude that:
  All errors behave as we expect them to.

- Testing a few normal runs right after each other to confirm reliability:
  We were wondering if after all the adapting and adjusting we did to our system, it will run smoothly and if we can be sure enough that it will also run smoothly on the next run.
  So we tested the system many times without triggering any errors and observed if it would run in a consistent way.
  From our tests we could conclude that:
  Our system is reliable enough that it is unlikely that something will go wrong if you correctly follow the user-cases and user constraints.

8.4 System under test and Formal Proofs

At the end of chapter 5 (Software Specification) we gave you a quick glance at our UPPAAL model. As we mentioned there, we are now going to give more details about the UPPAAL model and its contribution to our project. The construction of our UPPAAL model was a long, exhausting and frustrating process. This is the case because there were not a lot of tutorials online, and the tutorials of the models that were online were in no way comparable to the complexity of our finite automaton. It would therefore be nice if the university provided us with a little more documenting of how to use UPPAAL with an embedded system of this complexity. In the end we managed to make a model ourselves (with a lot of tweaking), but it has cost us a tremendous amount of time. That was something we wanted to get across. Now back to the model: We already showed a screenshot of the model in chapter 5, here are some other shots of vital components of the UPPAAL model.

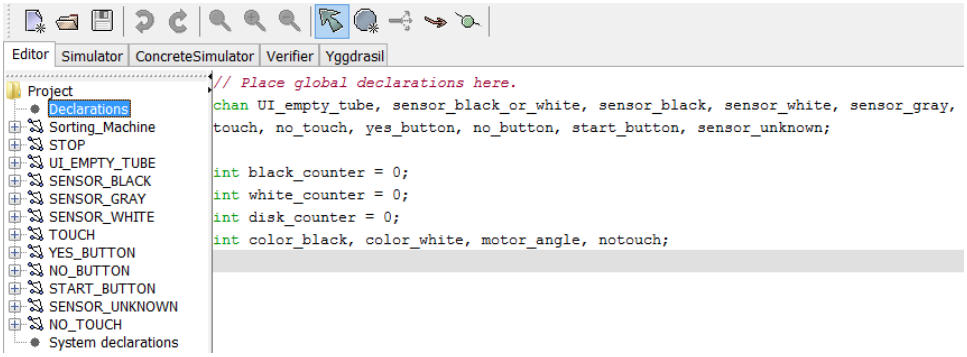

Figure 8.1: The global declarations of our UPPAAL model

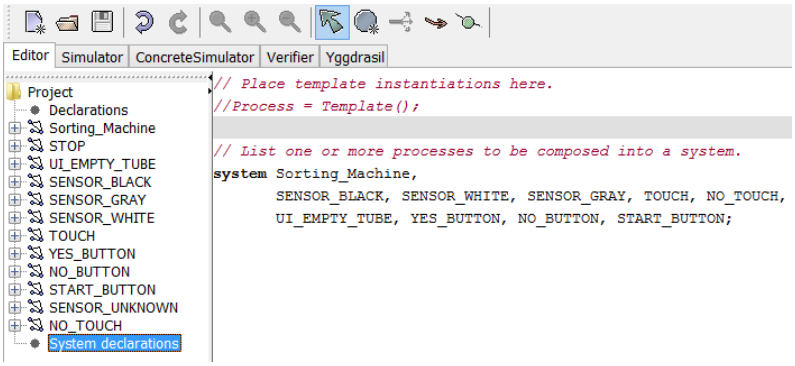

Figure 8.2: The system declarations of our UPPAAL model

As mentioned before our UPPAAL model is a simplification of the finite automaton which our software is actually based on. The UPPAAL model covers the most import things. It is where the actual sorting of the disks happens and a lot of error detection is covered in it. States that are implemented in our software, but not integrated in our model are for example the states responsible for the calibrating of the wheel. As we have seen at the previous section we have done a lot of test cases. Some of these test cases were at a stage where we had a complete working system. This is, a system under test with the entire machine and all its input, output and error-detection (combining software and hardware).

In week 6 for example we decided that we have implemented and tested the whole system enough and that it was ready to be considered as a 'snapshot' of our final product. So we did a test case with our sorting machine. We have of course already done a lot of test cases in which we tested the whole system, but we wanted to do like an 'official' test case where we repeatedly tested the entire machine. Here follows a brief summary of the test run (description of the execution of the test case) of this test case: The expectation was that it would have sorted the disks correctly and in the process of doing so showing correct output messages on the screen of the brick. We started the program on the brick. The calibrating was done correctly, and while doing so, showing correct screen messages. We then inserted 12 disks (after telling the machine that the tube was empty). While inserting the disks the screen messages were correct. We then told the machine to start with sorting the disks.
It sorted the disks in approximately 7 seconds. All the black disks (6) were in the right container and all the white disks (6) were in the left container.
While sorting the disk and when the sorting was done the machine outputted correct messages on the screen (that is, showing the correct state, notifications for the user and valid counter values).
We have run these full tests repeatedly and with different inputs and different 'paths' into our finite automaton. This is a brief summary of a test run. For more detailed descriptions of test runs we refer to Test Cases (see section 8.3).

Now lets go back to the UPPAAL model, because we can also consider a UPPAAL model to be a system under test (SUT). With UPPAAL you can simulate the behavior of the finite automaton. That is, you can specify the input and see if the output matches your expectations. UPPAAL provides several ways of simulating. The simulator of UPPAAL is a validation tool that enables examination of the possible dynamic executions of a system. It thus provides us an inexpensive mean of fault detection prior to verification by the model-checker. In the early phases of designing the model we exhaustively used the the step-by-step simulation as a validation tool. We could select from the enabled transitions ourselves and select next to take the transition. This helped us fixing faulty things of smaller components of our model (in the early phases of designing the model/ finite automaton). At later stages we also used the random simulator in which the program starts a random simulation where the simulator itself proceeds automatically by randomly selecting enabled inputs.

We have used this to check to see if our model contained mistakes and if it would get stuck somewhere. The random simulator is very fast so if you let it run for a while and observe its behavior it can be considered as a good validation tool for your model. When you know where to look of course. We made a video of our UPPAAL model, where we show some executions of the simulators (both step-by-step and random mode): https://goo.gl/WaFC95.

In addition we also used the verifier of UPPAAL as validation tool for our model. The verifier is to check safety and liveness properties by on-the-fly exploration of the state-space of a system in terms of symbolic states represented by constraints. We had at first a lot of problems doing these verification. When properties were not satisfied, it would give back the result: "property not satisfied", immediately, or in a couple of seconds. If this was not the case it would explore the state space until it would run out of memory. It would have passed millions of states (the verifier gives a past waiting list load of states) and it would then give an error stating that it we should consider reducing the amount of states etc, because the state space is too big. when doing some research on the internet we found that there were more people who encountered these problems. The verifier some times used up to 3/4 GB of the ram, but can not continue since even though the machine can have plenty of free memory (8GB) there is a limit as to how much as single process is allowed to address. This is caused by two things: UPPAAL is a 32-bit process. This means that there is now way that UPPAAL can address more than 4GB of memory and in addition the operating system only reserves a certain amount of address space for the user. On UPPAAL fora on the internet a software developer of UPPAAL advises to turn on aggressive state space reduction and the compact data structure representation.

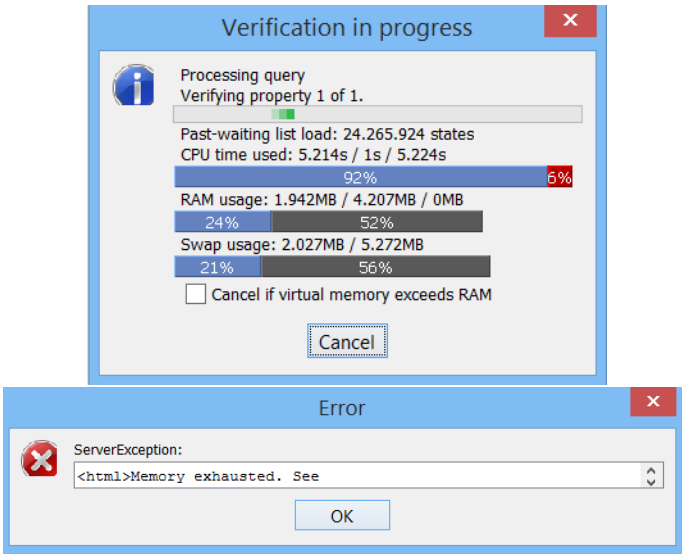

Figure 8.3: The verification process of some queries (A[]'s)

Because we have a big finite automaton with a lot of transitions, our state space is very big of course. We therefore had to use the advised options. UPPAAL gives us a lot of options: You can change the search order. This option influences the order in which the state space is explored. The options are Breadth first, Depth first and Random depth first. Based on the description Bread first should work best for us. UPPAAL in addition provided us a State Space Reduction option. You can choose between none, conservative and aggressive. We choose aggressive ofcourse, considering the size of our state space. It in addition also provides us an option for choosing the State Space Representation. We have tried Difference Bound Matrices (default), Compact Data Stucture (advised) and Under Approximation (and we then in addition chose the lowest hash table size). When tweaking these options the verifier would reach up to tens of millions states (it would take more than one hour to check one query). You can safely conclude that when reaching this stage the property is satisfied. Here follows a list and explanation of the statements (queries) we have verified using the UPPAAL model verifier:

- A[] not deadlock A state is a deadlock state if there are no outgoing action transitions neither from the state itself or any of its delay successors. Due to current limitations in UPPAAL, the deadlock state formula can only be used with reach-ability and invariantly path formula. A deadlock can be evidence of a design error.
- A[] whitecounter >= 0, E[] whitecounter >= 0 and A<> whitecounter >= 0. To check if the white counter cannot obtain negative values (should not be the case).

- A[] blackcounter >= 0, E[] blackcounter >= 0 and A<> blackcounter >= 0. Same goes for the black counter.
- A[] SortingMachine.waiting imply diskcounter >= 0, E[] SortingMachine.waiting imply diskcounter >= 0 and A<> SortingMachine.waiting imply diskcounter >= 0. While we are in the waiting state it will always be the case that there will either be no disks in the tube (diskcounter =0) or a positive number of disks.
- A[] SortingMachine.sortdisksnocounting imply (colorblack==1 || colorwhite==1), E[] SortingMachine.sortdisksnocounting imply (colorblack==1 || colorwhite==1) A<> SortingMachine.sortdisksnocounting imply (colorblack==1 || colorwhite==1). If we are in the SortDisksNoCounting state we have either seen a black disk or a white disk.
- A[] SortingMachine.expectsdisk imply diskcounter> 0, E[] SortingMachine.expectsdisk imply diskcounter> 0 and A<> SortingMachine.expectsdisk imply diskcounter> 0. When we are in the state expectsDisk it is always the case that the amount of disks that is counted is at least 1.

This concludes the verification and testing done with our UPPAAL model. If you want to check out our UPPAAL model yourself you can download the .xml file at:

https://drive.google.com/open?id=0B1zsOJUSLtRkWFo4TXhmZUxfOGc

Chapter 9

Conclusion

We have worked hard for several weeks, building and documenting our sorting machine. This was our first group project of this magnitude. Although our project may not have any money value, we still learned a lot these past weeks.

we learned how to

- plan these kind of group projects in a work plan and its added benefits.
- structurally work over the course of our entire project using the V-model.
- construct and share code together using Git and GitHub.
- improve our presentation skills.
- document these kind of projects.
- be creative with machine design to handle lack of components.
- work with the EV3 Brick and its associated motor and sensors.

Along the way we encountered some problems we had to overcome.

Somewhere during the early stages of our project, we were confronted with the fact that one of our group members was unfortunately no longer going to participate in this project. That left us us with 5 members for a project originally meant for 6. We solved this by adapting the work plan, dividing the workload among the 5 of us.

While working with the color sensor we encountered the problem that it can't reliably tell some of the LEGO components apart form the disks. We solved this by replacing those LEGO parts in our machine design with differently colored parts.

We encountered 2 problems regarding disk insertion. The first problem was that We weren't given a tube to insert our disks. We solved this problem by building a tube ourselves that was long enough to hold all 12 discs. The second problem was that we wanted to count the amount of disks that got inserted, but the pressure sensor isn't all that sensitive. We solved this problem by decreasing the height of the insertion point of the tube so that disks need to be pushed passed the sensor.

While working on the construction of the sorting wheel we encountered several problems. problems with telling the colors of the wheel apart form the disks, the wheel getting stuck, the disks sliding out and recognizing the teeth for calibration. After a lot of reconstructing and adapting we solve these problems. We ended up with a wheel constructed with red LEGO parts, of which the sensor could tell apart close by and further away, with teeth of 2.5 LEGO pieces high and rubber bars below the wheel it to prevent disks from sliding past it.

We ran into some problems while getting the EV3 to run self written code in general, since we didn't want to use the drag and drop system. We solve this problem by buying a SD card for the EV3 brick and flashing LEJOS Software on it.

In the end it when the machine was working properly and it could even sort 12 disks in under 6 seconds, we were very proud of what we had actually build and programmed. This is one of the most fun courses we have had up till now. We would like to thank Pieter Cuijpers for giving us the opportunity to be one of the (LEGO EV3) trial groups and we would like to thank Alberto Corvo for guiding and helping us along the road. It is all truly appreciated.

Appendices

Appendix A

Source code

A.1 Main

source-code/Main.java

```
1  import lejos.hardware.Button;
2  import lejos.hardware.lcd.LCD;
3  import lejos.utility.Delay;
4
5  public class Main {
6
7    public static void main(String[] args) {
8      State s = States.Initial;     // Set the initial
     state.
9      StateVariables sv = new StateVariables(); //
     Create new state variables object.
10     Input input = new Input(sv);    // Create the
     input object.
11     Output output = new Output(sv); // Create the
     output object.
12
13     while(!Button.ESCAPE.isDown()) {
14       // Update the state.
15       s = s.next(input, output);
16
17       // Update the message drawn on the screen.
18       output.setMessage(s);
19
20       // Implement the STOP button.
21       if(Button.UP.isDown()) {
```

```
22          s = States.Rest;
23          output.aborted();
24        }
25      }
26    }
27  }
```

A.2 States

source-code/State.java

```
1  interface State {
2    public State next(Input input, Output output);
3  }
```

source-code/States.java

```
1  /*
2   * Defines all states, and how input and the current
      state define
3   *  the output and the new current state.
4   */
5  enum States implements State {
6    Initial {
7      @Override
8      public State next(Input i, Output o) {
9        o.isCalibrating(); // Show a message that the
      machine is now calibrating.
10       return moveBetweenTeeth; // Start the
      calibration at boot.
11     }
12   },
13
14   /*
15    * If there is a teeth in front of the sensor, then
      turn the
16    * wheel for half a teeth to prevent the teeth to
      be in front
17    * of the sensor.
18    */
19   moveBetweenTeeth {
20     @Override
21     public State next(Input i, Output o) {
22       i.updateColor();
```

```
23
24        if(i.colorSensorTeeth()) {
25          o.motorTurnHalfTeeth(true);
26
27          if(i.getMotorStalled()) {
28            o.messageMotorStalled();
29            return MotorStalled;
30          }
31        }
32
33        return findFirstPoint;
34      }
35    },
36
37    /*
38     *  Turns the wheel until a teeth is found, then
       stores the wheel
39     *  and starts the search for the position of the
       second teeth.
40     */
41
42    findFirstPoint {
43      @Override
44      public State next(Input i, Output o) {
45        i.updateColor(); // Update the color sensor
       data.
46
47        if(i.colorSensorTeeth()) { // Check if we found
       a teeth.
48          o.setFirstCaliPoint(); // If so, set this as
       the first calibration point.
49          o.motorTurnHalfTeeth(true); // And turn in
       the reversed direction past the current teeth.
50
51          if(i.getMotorStalled()) {
52            o.messageMotorStalled();
53            return MotorStalled;
54          } else {
55            return findSecondPoint;
56          }
57        } else {
58          o.motorTurnSmallStep(false); // If not, keep
       turning until a teeth is found.
59
60          if(i.getMotorStalled()) {
61            o.messageMotorStalled();
```

```
62            return MotorStalled;
63          } else {
64            return findFirstPoint;
65          }
66        }
67      }
68    },
69
70    /*
71     * Turns the wheel in search for the second teeth. It searches
72     * in the opposite direction compared to the search for the first
73     * teeth.
74     */
75    findSecondPoint {
76      @Override
77      public State next(Input i, Output o) {
78        i.updateColor(); // Update the color sensor data.
79
80        if(i.colorSensorTeeth()) { // Check if we found the second teeth.
81          o.setSecondCaliPoint(); // If so, store this as the second calibration point.
82          o.motorMoveInBetweenCaliPoints(); // Turn to the calibrated point.
83
84          if(i.getMotorStalled()) {
85            o.messageMotorStalled();
86            return MotorStalled;
87          }
88
89          o.askIfEmpty(); // Ask the user if the tube is empty.
90          return Rest;
91        } else {
92          o.motorTurnSmallStep(true); // If not, keep turning until the second teeth is found.
93          return findSecondPoint;
94        }
95      }
96    },
97
98    /*
```

```
99      * State prior to sorting, allows the user to tell
        the machine if the
100     * tube is:
101     *  - Contains disks (No button): Then the machine
        will sort disks as
102     * long as they are in front of the sensor.
103     *  - Empty (Yes button): The machine will start
        counting inserted disks.
104     *
105     *  Or if the user wants to calibrate, pressing
        Enter restarts the
106     *  calibration sequence.
107     *
108     *  If a disk is inserted during this state, then
        we will handle this
109     *  by showing an error message and sorting the
        tube without counting.
110     */
111    Rest {
112      @Override
113      public State next(Input i, Output o) {
114        if (i.buttonYesDown()) {
115          o.setCounterToZero();
116          return Waiting;
117        } else if(i.buttonNoDown()) {
118          return CheckDiskPresent;
119        } else if (i.touchDown()) {
120          o.tooEarly();
121          return InsertedEarly;
122        } else if (i.buttonEnterPressed()) {
123          o.isCalibrating();
124          return moveBetweenTeeth;
125        }
126
127        return Rest;
128      }
129    },
130
131    /*
132     * If a disk was inserted early, then wait until
        the user presses
133     * the Enter button and then start the sorting
        procedure.
134     */
135    InsertedEarly {
136      @Override
```

```
137     public State next(Input i, Output o) {
138       if (i.buttonEnterPressed()) {
139         return CheckDiskPresent;
140       }
141
142       return InsertedEarly;
143     }
144   },
145
146   /*
147    * Phase executed before sorting a disk, checks if
       as disk is present.
148    * If not, then return to the waiting state, else
       sort the disk.
149    */
150   CheckDiskPresent {
151     @Override
152     public State next(Input i, Output o) {
153       i.updateColor(); // Sample the sensor for new
      color data.
154
155       if (i.colorSensorIsNoDisk() &&
      !i.colorSensorBlack() && !i.colorSensorWhite()) {
156         o.setCounterToZero();
157         return Waiting;
158       } else if (i.colorSensorBlack() ||
      i.colorSensorWhite()) {
159         o.sorting();
160         return SortDisksNoCounting;
161       }
162
163       return CheckDiskPresent;
164     }
165   },
166
167   /*
168    * Sort the disks without using the counter.
169    */
170   SortDisksNoCounting {
171     @Override
172     public State next(Input i, Output o) {
173       i.updateColor(); // Sample the sensor for new
      color data.
174
175       if (i.colorSensorIsNoDisk()) {
176         return CheckDiskPresent;
```

```
177       } else if (i.colorSensorBlack()) {
178         o.sorting();
179         o.motorSortBlack();
180
181         if(i.getMotorStalled()) {
182           o.messageMotorStalled();
183           return MotorStalled;
184         } else {
185           return SortDisksNoCounting;
186         }
187       } else if (i.colorSensorWhite()) {
188         o.sorting();
189         o.motorSortWhite();
190         return SortDisksNoCounting;
191       } else {
192         if (i.buttonEnterPressed()) {
193           o.sorting();
194           o.motorSortWhite();
195           return SortDisksNoCounting;
196         } else {
197           o.anotherColor();
198           return SortDisksNoCounting;
199         }
200       }
201     }
202   },
203
204   MotorStalled {
205     @Override
206     public State next(Input i, Output o) {
207       if(i.buttonEnterDown()) {
208         o.setCounterToZero();
209         return Initial;
210       }
211
212       return MotorStalled;
213     }
214   },
215
216   /*
217    * Waits for the user to insert a disk, or to start
       the
218    * sorting procedure.
219    */
220   Waiting {
221     @Override
```

```
222     public State next(Input i, Output o) {
223       if (!i.counterGreaterThanZero()) {
224         o.waitForInput();
225       } else {
226         o.enterToSort();
227       }
228
229       if (i.touchDown()) {
230         o.increaseCounter();
231         return DiskAdd;
232       }
233
234       if (i.buttonEnterPressed()) {
235         return AcceptDisk2;
236       }
237
238       return Waiting;
239     }
240   },
241
242   DiskAdd {
243     @Override
244     public State next(Input i, Output o) {
245       if (!i.touchDown()) {
246         return Waiting;
247       }
248
249       return DiskAdd;
250     }
251   },
252
253   AcceptDisk2 {
254     @Override
255     public State next(Input i, Output o) {
256       if (i.counterGreaterThanZero()) {
257         return ExpectsDisk;
258       } else {
259         return ExpectsFinished;
260       }
261     }
262   },
263
264   ExpectsFinished {
265     @Override
266     public State next(Input i, Output o) {
```

```
267       i.updateColor(); // Sample the sensor for new
    color data.
268
269       if ((i.colorSensorBlack() ||
    i.colorSensorWhite()) && !i.colorSensorIsNoDisk())
    {
270         o.askUnexpectedDisk(); // Ask the user what
    to do with the unexpected disk.
271         return AskUser;
272       } else {
273         o.tubeEmpty(); // Press any button to continue
274
275         if (i.buttonYesDown() || i.buttonNoDown() ||
    i.buttonEnterPressed()) {
276           o.askIfEmpty(); // Ask the user if the tube
    is empty.
277           return Rest;
278         }
279       }
280
281       return ExpectsFinished;
282     }
283   },
284
285   AskUser {
286     @Override
287     public State next(Input i, Output o) {
288       if (i.buttonNoDown()) {
289         o.askIfEmpty(); // Ask the user if the tube
    is empty.
290         o.setCounterToZero();
291         return Rest;
292       } else if (i.buttonYesDown()) {
293         o.increaseCounter(); // Apparently one - at
    least - more disk was present than expected, take
    this into account.
294         return ExpectsDisk;
295       } else {
296         return AskUser;
297       }
298
299     }
300   },
301
302   ExpectsDisk {
303     @Override
```

```
304     public State next(Input i, Output o) {
305       i.updateColor(); // Sample the sensor for new
      color data.
306
307       if (i.colorSensorWhite() &&
      !i.colorSensorBlack() && !i.colorSensorIsNoDisk())
      {
308         o.decreaseCounter();
309         o.motorSortWhite();
310         o.increaseWhiteCounter();
311
312         if(i.getMotorStalled()) {
313           o.messageMotorStalled();
314           return MotorStalled;
315         } else {
316           return AcceptDisk2;
317         }
318       } else if (i.colorSensorBlack() &&
      !i.colorSensorWhite() && !i.colorSensorIsNoDisk())
      {
319         o.decreaseCounter();
320         o.motorSortBlack();
321         o.increaseBlackCounter();
322
323         if(i.getMotorStalled()) {
324           o.messageMotorStalled();
325           return MotorStalled;
326         } else {
327           return AcceptDisk2;
328         }
329       } else if (!i.colorSensorBlack() &&
      !i.colorSensorWhite() && i.colorSensorIsNoDisk()) {
330         o.stuckInTube();
331
332         if (i.buttonEnterPressed()) {
333           o.askIfEmpty(); // Ask the user if the tube
      is empty.
334           return Rest;
335         }
336       } else {// If the color sensor is indecisive.
337         o.anotherColor();
338
339         if (i.buttonEnterPressed()) {
340           o.askIfEmpty(); // Ask the user if the tube
      is empty.
341           return Rest;
```

```
342           }
343         }
344
345         return ExpectsDisk;
346       }
347     }
348 }
```

source-code/StateVariables.java

```
1  public class StateVariables {
2
3    private int diskCounter = 0; // A counter that
       stores the amount of disks currently in the tube.
4    private int whiteDiskCounter = 0; // A counter that
       stores the amount of sorted white disks.
5    private int blackDiskCounter = 0; // A counter that
       stores the amount of sorted black disks.
6
7    private float firstCaliPoint;
8    private float secondCaliPoint;
9
10   private boolean motorStalled = false;
11
12   public StateVariables() {}
13
14   //         INPUT
15
16   /*
17    * Return if disks are expected in the tube.
18    * @return  True if more disks where expected, else
       false.
19    */
20   public boolean counterGreaterThanZero() {  //
       boolean function that returns true when
       diskcounter is greater than 0
21     if (diskCounter > 0) {
22       return true;
23     } else {
24       return false;
25     }
26   }
27
28   public boolean getMotorStalled() {
29     return motorStalled;
30   }
```

```
31
32  //         OUTPUT
33
34  /**
35   * Set first calibration point
36   */
37  public void setFirstCaliPoint(float point) {
38    firstCaliPoint = point;
39  }
40
41  /**
42   * Set second calibration point
43   */
44  public void setSecondCaliPoint(float point) {
45    secondCaliPoint = point;
46  }
47
48  /**
49   * Returns the first calibration point
50   */
51  public float getFirstCaliPoint() {
52    return firstCaliPoint;
53  }
54
55  /**
56   * Returns the second calibration point
57   */
58  public float getSecondCaliPoint() {
59    return secondCaliPoint;
60  }
61
62  /*
63   * Increment the disk counter by one.
64   */
65  public void increaseCounter() { // increase the
     diskcounter
66    diskCounter++;
67  }
68
69  /*
70   * Decrement the black disk counter by one.
71   */
72  public void decreaseCounter() {
73    diskCounter--;
74  }
75
```

```
76    /*
77     * Increment the white disk counter by one.
78     */
79    public void increaseWhiteCounter() { // increase
       the whitecounter
80      whiteDiskCounter++;
81    }
82
83    /*
84     * Increment the black disk counter by one.
85     */
86    public void increaseBlackCounter() { // increase
       the blackcounter
87      blackDiskCounter++;
88    }
89
90    /*
91     * Reset the disk counter.
92     */
93    public void setCounterToZero() { // set diskcounter
       to 0 again
94      diskCounter = 0;
95    }
96
97    /*
98     * Reset the white disk counter.
99     */
100   public void setWhiteCounterToZero() {
101     whiteDiskCounter = 0;
102   }
103
104   /*
105    * Reset the black disk counter.
106    */
107   public void setBlackCounterToZero(){ // set
       blackcounter to 0 again
108     blackDiskCounter = 0;
109   }
110
111   //        SCREEN OUTPUT
112   /*
113    * Return the value of the white disk counter.
114    */
115   public int getWhiteDiskCount() {
116     return whiteDiskCounter;
117   }
```

```
118
119   /*
120    * Return the value of the black disk counter.
121    */
122   public int getBlackDiskCount() {
123     return blackDiskCounter;
124   }
125
126   /*
127    * Return the amout of disks.
128    */
129   public int getDiskCount() {
130     return diskCounter;
131   }
132
133   //        MOTOR
134
135   public void motorStalled(boolean isStalled) {
136     motorStalled = isStalled;
137   }
138 }
```

A.3 Input and output

source-code/Input.java

```
1  import lejos.hardware.Button;
2  import lejos.hardware.port.ConfigurationPort;
3  import lejos.hardware.port.Port;
4  import lejos.hardware.port.SensorPort;
5  import lejos.hardware.sensor.EV3ColorSensor;
6  import lejos.hardware.sensor.EV3TouchSensor;
7  import lejos.robotics.SampleProvider;
8  import lejos.hardware.sensor.EV3SensorConstants;
9
10 public class Input implements EV3SensorConstants {
11   // Store the state variables object.
12   private StateVariables sv;
13
14   // Initialize variables for the color variables.
15   private EV3ColorSensor colorSensor;
16   private SampleProvider colorRGB;
17   private float[] RGB = new float[3];
18   private float RGBAvg, redPercentage;
```

```
19
20  // Initialize all port variables. Needed to find in
     which port the motor is.
21  private Port[] ports = new Port[4];
22  ConfigurationPort[] configPorts = new
     ConfigurationPort[4];
23  int colorPort, touchPort;
24  EV3TouchSensor touchSensor;
25
26  // Initialize variables for the touch sensor and
     buttons.
27  private float[] isTouched = new float[1];
28  private boolean enterWasDown = false;
29
30  public Input(StateVariables sv) {
31    this.sv = sv;
32
33    initializeSensors();
34  }
35
36  //        HANDLE COLOR INPUT
37
38  /**
39   * Initialize sensors, it will search all four
     sensor ports
40   * for the touch and color sensor.
41   */
42  void initializeSensors() {
43    // Define all ports.
44    ports[0] = SensorPort.S1;
45    ports[1] = SensorPort.S2;
46    ports[2] = SensorPort.S3;
47    ports[3] = SensorPort.S4;
48
49    // Open all configuration ports.
50    for (int i = 0; i < ports.length; i++) {
51      configPorts[i] =
    ports[i].open(ConfigurationPort.class);
52    }
53
54    // Wait until both a touch and a color sensor are
    detected.
55    updateSensors();
56    while (touchPort == -1 || colorPort == -1) {
57      updateSensors();
58    }
```

```
59
60      // Close all ports, to allow the data of the
        sensors to be read.
61      for (int i = 0; i < configPorts.length; i++) {
62        configPorts[i].close();
63      }
64
65      // Set the touch and color sensor.
66      touchSensor = new
        EV3TouchSensor(ports[touchPort]);
67      colorSensor = new
        EV3ColorSensor(ports[colorPort]);
68
69      // Create the RGB color fetcher.
70      colorRGB = colorSensor.getRGBMode();
71    }
72
73    /**
74     * Update sensor position.
75     */
76    private boolean updateSensors() {
77      colorPort = touchPort = -1; // Reset all values.
78
79      for (int i = 0; i < configPorts.length; i++) {
80        int portType = configPorts[i].getPortType();
81
82        // Find which sensor currently is in this port.
83        switch (portType) {
84        case CONN_INPUT_UART:
85          if (colorPort != -1) {
86            // Error: Two light sensors?
87            return false;
88          }
89
90          colorPort = i;
91          break;
92
93        case CONN_INPUT_DUMB:
94          if (touchPort != -1) {
95            // Error: Two touch sensors?
96            return false;
97          }
98
99          touchPort = i;
100         break;
101
```

```
102         case CONN_ERROR:
103           // Error: Unexpected sensor for this port, motor
104           // in sensor port or different EV3 sensor than color
105           // or touch?
106           return false;
107         }
108       }
109
110       return true; // Return true if no errors where detected.
111     }
112
113     /**
114      * Fetches new colors from the sensor which replace the old values.
115      *
116      * Note: Call them before getting color input which relies on this data,
117      * else old information will be used.
118      */
119     public void updateColor() {
120       colorRGB.fetchSample(RGB, 0); // Get the RGB values.
121
122       RGBAvg = (RGB[0] + RGB[1] + RGB[2]) / 3;
123       redPercentage = 100 * RGB[0] / (3 * RGBAvg);
124     }
125
126     /**
127      * Returns if a white disk is in front of the sorting wheel.
128      * @return True if the disk in front of the sensor is white, else false.
129      */
130     public boolean colorSensorWhite() {
131       return RGBAvg >= .2;
132     }
133
134     /**
135      * Returns if a black disk is in front of the sorting wheel.
136      * @return True if the disk in front of the sensor is black.
137      */
```

```
138   public boolean colorSensorBlack() {
139     return .2 > RGBAvg && !colorSensorIsNoDisk();
140   }
141
142   /**
143    * Returns if no disk is in front of the sorting
       wheel.
144    * @return True if no disk is in front of the
       sensor, else false.
145    */
146   public boolean colorSensorIsNoDisk() {
147     return redPercentage > 25 && RGB[0] <= 0.04f;
148   }
149
150   /**
151    * Returns if a teeth of the wheel is in front of
       the sensor.
152    * @return True if a wheel teeth is in front of the
       color sensor.
153    */
154   public boolean colorSensorTeeth() {
155     return RGB[0] > 0.04f && redPercentage > 30;
156   }
157
158   /**
159    * Checks if the touch sensor is pressed.
160    * @return True if the sensor is pressed, false if
       it isn't.
161    */
162   public boolean touchDown() {
163     // Get new values from the touch sensor.
164     if (touchPort != -1) {
165       touchSensor.fetchSample(isTouched, 0);
166
167       // Convert result of the touch sensor to a
       boolean value.
168       if (isTouched[0] == 0) {
169         return false;
170       } else {
171         return true;
172       }
173     } else {
174       return false;
175     }
176   }
177
```

```
178   //         HANDLE TOUCH SENSOR INPUT
179
180   /**
181    * Returns if the start/stop button is pressed.
182    * @return True if the start/stop button is
       pressed, false if it isn't.
183    */
184   public boolean buttonEnterDown() {
185     return Button.ENTER.isDown();
186   }
187
188   public boolean buttonEnterPressed() {
189     boolean isDown = buttonEnterDown();
190     boolean isPressed = isDown && !enterWasDown;
191
192     enterWasDown = isDown;
193
194     return isPressed;
195   }
196
197   /**
198    * Returns if the Yes button is pressed.
199    * @return True if the Yes button is pressed, false
       if it isn't.
200    */
201   public boolean buttonYesDown() {
202     return Button.RIGHT.isDown();
203   }
204
205   /**
206    * Returns if the No button is pressed.
207    * @return True if the No button is pressed, false
       if it isn't.
208    */
209   public boolean buttonNoDown() {
210     return Button.LEFT.isDown();
211   }
212
213   // HANDLE STATE VARIABLES
214
215   /**
216    * Returns if the disk counter is zero.
217    * @return True if the disk counter is greater than
       zero, else false.
218    */
219   public boolean counterGreaterThanZero() {
```

```
220     return sv.counterGreaterThanZero();
221   }
222
223   /**
224    * Retrieves the motor stalled state variable.
225    * @return The value of the motor stalled state
       variable.
226    */
227   public boolean getMotorStalled() {
228     return sv.getMotorStalled();
229   }
230 }
```

source-code/Output.java

```
1  import lejos.internal.ev3.EV3LED;
2  import lejos.hardware.lcd.LCD;
3  import lejos.hardware.lcd.GraphicsLCD;
4  import lejos.hardware.motor.Motor;
5  import lejos.hardware.BrickFinder;
6  import lejos.hardware.LED;
7  import lejos.hardware.ev3.LocalEV3;
8  import lejos.hardware.motor.NXTRegulatedMotor;
9  import lejos.hardware.port.ConfigurationPort;
10 import lejos.hardware.port.MotorPort;
11 import lejos.hardware.port.Port;
12 import lejos.hardware.sensor.EV3SensorConstants;
13
14 public class Output implements EV3SensorConstants {
15   private StateVariables sv;  // Store the state
       variables object.
16
17   private String currentMessage = ""; // Contains the
       message which currently should be drawn, initially
       empty.
18
19   // Motor angle sizes.
20   private int turndegrees = 216; // 360 degrees * (24
       gear teeth / 8 gear teeth) gear multiplier / 5
       teeth = 216 degree / wheel teeth
21   private int smallStepSize = 1; // Define the size
       of a small step as an angle in degrees.
22
23   // LED variables.
24   private LED led; // The LED object which is needed
       to control the back light color of the buttons.
```

```
25    private int lastLEDSpeed = -1;
26    private String lastLEDColor = null;
27
28    // Graphics object used to draw shaped on the LCD.
29    private GraphicsLCD g =
       BrickFinder.getDefault().getGraphicsLCD();
30
31    // Allocate variables for motor related things,
       like port detection.
32    private Port[] ports = new Port[4];
33    private ConfigurationPort[] configPorts = new
       ConfigurationPort[4];
34    private int motorPort;          // Stores the
       number of the motor port on which the motor is
       detected.
35    private NXTRegulatedMotor motor;    // The motor
       object which is connected to the wheel.
36
37    // The constructor of this class.
38    public Output(StateVariables sv) {
39      this.sv = sv;
40
41      // Disable the auto refresh of the screen, we
       will take care of it ourself.
42      LCD.setAutoRefresh(false);
43
44      initializeMotor();
45    }
46
47    /**
48     * Initialize the motor by first looking in which
       port it is.
49     */
50    private void initializeMotor() {
51      // Define all ports.
52      ports[0] = MotorPort.A;
53      ports[1] = MotorPort.B;
54      ports[2] = MotorPort.C;
55      ports[3] = MotorPort.D;
56
57      // Open all configuration ports.
58      for(int i = 0; i < ports.length; i++) {
59        configPorts[i] =
       ports[i].open(ConfigurationPort.class);
60      }
61
```

```
62      // Wait until a motor is detected.
63      do {
64        updateMotorPort();
65      } while(motorPort == -1);
66
67      // Close all ports, to allow to open motor ports
        on it.
68      for(int i = 0; i < configPorts.length; i++) {
69        configPorts[i].close();
70      }
71
72      // Select - and therefore open - the appropriate
        motor port.
73      switch(motorPort) {
74      case 0:
75        motor = Motor.A;
76        break;
77
78      case 1:
79        motor = Motor.B;
80        break;
81
82      case 2:
83        motor = Motor.C;
84        break;
85
86      case 3:
87        motor = Motor.D;
88        break;
89      }
90
91      // Maximize the motor speed for even faster
        sorting.
92      motor.setSpeed(motor.getMaxSpeed());
93
94      // Defines stalled as pushed out of position for
        2 degrees and 500 ms.
95      motor.setStallThreshold(2, 500);
96    }
97
98    /**
99     * Update motor port position.
100    */
101   private boolean updateMotorPort() {
102     motorPort = -1; // Reset all values.
103
```

```
104     // Go through all ports and read the port type
    from them.
105     for(int i = 0; i < configPorts.length; i++) {
106       int portType = configPorts[i].getPortType();
107 
108       // Find sensor currently is in this port.
109       switch(portType) {
110       case TYPE_TACHO:
111       case TYPE_MINITACHO:
112       case TYPE_NEWTACHO:
113       case CONN_OUTPUT_TACHO:
114         if(motorPort != -1) {
115           // Throw error: Two motors connected?
116           return false;
117         }
118 
119         motorPort = i;
120         break;
121 
122       case CONN_ERROR:
123         // Throw error: Unexpected sensor for this
    port, motor in sensor port or vice versa?
124         return false;
125       }
126     }
127 
128     return true; // Return true if no errors where
    detected.
129   }
130 
131   //        HANDLE MESSAGE CONTROL
132 
133   public void tubeEmpty() {
134     currentMessage = "Tube should be empty! Press
    Enter to resume.";
135     setLEDState("userInput");
136   }
137 
138   public void waitForInput() {
139     currentMessage = "No disks counted, inserted
    disks will now be counted. Press Enter to cancel.";
140     setLEDState("userInput");
141   }
142 
143   public void askIfEmpty() {
```

```
144       currentMessage = "Prior to sorting check if the
          tube is empty, press Yes or No. Enter: Calibrate";
145       setLEDState("userInput");
146     }
147
148     public void tubeNotEmpty() {
149       currentMessage = "Tube is not empty. Sorting
          disks now.";
150       setLEDState("busy");
151     }
152
153     public void sorting() {
154       currentMessage = "Sorting.";
155       setLEDState("busy");
156     }
157
158     public void askUnexpectedDisk() {
159       currentMessage = "Unexpected disk detected, press
          Yes to sort it, No to reset and go back to the
          main menu.";
160       setLEDState("error");
161     }
162
163     public void breakMachine() {
164       currentMessage = "Break. Resting..";
165       setLEDState("busy");
166     }
167
168     public void notBreak() {
169       currentMessage = "No break. Sorting..";
170       setLEDState("busy");
171     }
172
173     public void aborted() {
174       currentMessage = "Machine stopped, to start
          sorting: press Yes if the tube is empty, else
          press No.";
175       setLEDState("error");
176     }
177
178     public void start() {
179       currentMessage = "Starting..";
180       setLEDState("busy");
181     }
182
183     public void noDisk() {
```

```
184       currentMessage = "No disk detected";
185       setLEDState("error");
186     }
187
188     public void tooEarly() {
189       currentMessage = "Disk inserted too early, press
          enter to flush.";
190       setLEDState("error");
191     }
192
193     public void anotherColor() {
194       currentMessage = "Color not detected, press enter
          to flush as white disk?";
195       setLEDState("error");
196     }
197
198     public void stuckInTube() {
199       currentMessage = "Earlier done than expected,
          disk stuck? Press enter to dismiss.";
200       setLEDState("error");
201     }
202
203     public void enterToSort() {
204       currentMessage = "Press Enter to start sorting.";
205       setLEDState("userInput");
206     }
207
208     public void isCalibrating() {
209       currentMessage = "Calibration in process.";
210       setLEDState("busy");
211     }
212
213     public void messageMotorStalled() {
214       currentMessage = "Motor was stalled. Solve the
          problem and press Enter to restart, or Abort to
          exit the program.";
215       setLEDState("error");
216
217       // Reset the motor stalled variable.
218       sv.motorStalled(false);
219     }
220
221     /**
222      * Draws a string in a specific area.
223      * @param The string to be drawn.
224      * @param x start coordinate of the segment.
```

```
225     * @param y start coordinate of the segment.
226     * @param Width of the segment.
227     * @param Height of the segment.
228     */
229    private void textSegment(String input, int x, int
        y, int width, int height) {
230      String[] words = input.split(" ");
231      int line = y;
232      int lineLength = 0;
233      String currentLine = "";
234
235      // Loop through all words, until all words are
        drawn or the screen is segment if full.
236      for(int i = 0; i < words.length && line + 1 <
        height; i++) {
237        // Check if the next word fits on the current
        line, if it doesn't then draw the previous line
        and resume on the next one.
238        if(lineLength + words[i].length() >= width) {
239          // Draw the current line.
240          LCD.drawString(currentLine, x, line);
241
242          // Resume to the next line.
243          line++;
244          currentLine = "";
245          lineLength = 0;
246        }
247
248        // Add the word evaluated in this iteration to
        the current line.
249        lineLength += words[i].length() + 1;
250        currentLine += words[i] + " ";
251      }
252
253      LCD.drawString(currentLine, x, line); // Draw the
        last line.
254    }
255
256    /**
257     * Draw the currently active message on the screen.
258     * @param The current state.
259     */
260    public void setMessage(State s) {
261      LCD.clear();                  // Clear the screen
        prior drawing.
```

```
262     textSegment(currentMessage, 0, 0, 19, 5);   // Draw the message.
263
264     // Define the size and position of the disk bar.
265     int barWidth = 92;
266     int barHeight = 10;
267     int x = 84;
268     int y = 82;
269
270     g.drawRect(x, y, barWidth, barHeight); // Draw the outline.
271     g.fillRect(x, y, sv.getDiskCount() * barWidth / 12, barHeight); // Fill the bar.
272
273     // Draw the disk counters on the screen.
274     LCD.drawString("Disks:" + sv.getDiskCount(), 0, 5);
275     LCD.drawString("#Sorted W:" + sv.getWhiteDiskCount(), 0, 6);
276     LCD.drawString("B:" + sv.getBlackDiskCount(), 13, 6);
277
278     // Draw the current state on the screen.
279     LCD.drawString("CS: " + (States)s, 0, 7);
280
281     LCD.refresh(); // Refresh the screen to update the content on it.
282   }
283
284   /**
285    * Set the current LED state.
286    * @param State as a string, options are "error", "userInput", and "busy".
287    */
288   public void setLEDState(String state) {
289     switch(state) {
290     case "error":
291       setLED("red", 2);
292       break;
293     case "userInput":
294       setLED("orange", 1);
295       break;
296     case "busy":
297       setLED("green", 0);
298       break;
299     }
```

```
300   }
301
302   /**
303    * Sets the LED color and speed.
304    * @param The color of the LED as a string, options
       are "red", "orange", and "green".
305    * @param speed
306    */
307   public void setLED(String color, int speed) {//
       Requires a color and a speed 0, 1 or 2
308     led = LocalEV3.ev3.getLED();
309     speed *= 3; // Is needed for the right number
310
311     if (speed != lastLEDSpeed &&
       !color.equals(lastLEDColor)){ // To make sure it's
       only called the first time
312       lastLEDSpeed = speed;
313       lastLEDColor = color;
314       if (color.equals("green")) {
315         led.setPattern(speed + EV3LED.COLOR_GREEN);
316       } else if (color.equals("red")) {
317         led.setPattern(speed + EV3LED.COLOR_RED);
318       } else if (color.equals("orange")) {
319         led.setPattern(speed + EV3LED.COLOR_ORANGE);
320       } else {
321         led.setPattern(0);
322       }
323     }
324   }
325
326   //         HANDLE MOTOR CONTROL
327
328   /**
329    * When the color sensor detects a black disk, turn
       one teeth right.
330    */
331   public void motorSortBlack() {
332     turnMotor(turndegrees);
333   }
334
335   /**
336    * When the color sensor detects a white disk, turn
       one teeth left.
337    */
338   public void motorSortWhite() {
339     turnMotor(-turndegrees);
```

```
340   }
341
342   /**
343    * Let the motor turn a small step, it will not
       wait for the motor to finish.
344    * @param True if it should turn left, false if it
       should turn right.
345    */
346   public void motorTurnSmallStep(boolean left) {
347     if(left) {
348       motor.rotate(smallStepSize, true);
349     } else {
350       motor.rotate(-smallStepSize, true);
351     }
352   }
353
354   /**
355    * Let the motor run for half a teeth.
356    * @param True if it should turn left, false if it
       should turn right.
357    */
358   public void motorTurnHalfTeeth(boolean left) {
359     if(left) {
360       turnMotor(turndegrees / 2);
361     } else {
362       turnMotor(-turndegrees / 2);
363     }
364   }
365
366   /**
367    * Moves the wheel in between the two calibrated
       points.
368    */
369   public void motorMoveInBetweenCaliPoints() {
370     int targetAngle = (int) ((sv.getFirstCaliPoint()
       + sv.getSecondCaliPoint()) / 2f);
371     motor.rotateTo(targetAngle, false);
372   }
373
374   /**
375    * Turn the motor while keeping track of a stalling
       motor.
376    * @param The angle which it should turn.
377    */
378   private void turnMotor(int angle) {
```

```
379     motor.rotate(angle, true); // Turn the motor,
    return immediately.
380
381     while(!motor.isStalled() && motor.isMoving()) {}
382
383     if(motor.isStalled()) {
384       messageMotorStalled();
385       sv.motorStalled(true);
386     }
387
388   }
389   //         HANDLE STATE VARIABLES
390
391   public void decreaseCounter() {
392     sv.decreaseCounter();
393   }
394
395   public void increaseBlackCounter() {
396     sv.increaseBlackCounter();
397   }
398
399   public void increaseWhiteCounter() {
400     sv.increaseWhiteCounter();
401   }
402
403   public void increaseCounter() {
404     sv.increaseCounter();
405   }
406
407   public void setCounterToZero() {
408     sv.setCounterToZero();
409   }
410
411   public void setFirstCaliPoint() {
412     sv.setFirstCaliPoint(motor.getPosition());
413   }
414
415   public void setSecondCaliPoint() {
416     sv.setSecondCaliPoint(motor.getPosition());
417   }
418 }
```

Appendix B

Logbook

B.1 Adriaan

Up to date until 19th of March.

Date	Time spend	What
2nd of February	2h	First group meeting.
4th of February	2h	Worked on abstract.
5th of February	2h 20m	Finished abstract.
5th of February	2h 20m	Work on self-reflection.
Subtotal	8h 40m	**Pre-orientation phase**
14th of February	50m	Research into online LaTeXcollaboration tool.
16th of February	1h	Made a small presentation about my experiences with the Mindstorms NXT.
16th of February	1h	Worked on templating.
16th of February	7h 40m	Worked on the project as a group.
16th of February	1h	Prepared presentation about software Validation and Testing.
17th of February	1h 30m	Wrote first draft System Requirements.
17th of February	3h 20m	Worked on the project as a group.
17th of February	1h 40m	Wrote EV3 overview.
19th of February	3h 20m	Worked on the project as a group.
19th of February	1h 10m	Worked on logbook template.
Subtotal	22h 30m	**Orientation phase**
22nd of February	1h	Resumed work on logbook, now converted to LaTeXpackage.
23rd of February	1h	Preparing midterm presentation.
24th of February	50m	Added some things for the midterm presentation.

24th of February	9h 30m	Worked on the project as a group (midterm presentation and machine design).
25th of February	3h 30m	Worked on the project as a group (flashing EV3, experimenting with leJOS software).
26th of February	2h	Worked on the project as a group (specified Final-State Machine and global structure of software design).
Subtotal	17h 50m	**Machine Design phase**
1st of March	8h 20m	Started on software implementation and worked on software design.
2nd of March	4h	Resumed software design and implementation.
4th of March	4h	Worked on the software implementation of the input and output.
4th of March	40m	Cleaned up some code.
7th of March	50m	Made agenda for meeting.
8th of March	7h	Worked on the Software Implementation and made agenda for next meeting.
9th of March	3h 30m	Resumed work on Software Implementation.
14th of March	3h 20m	Worked on Software Implementation (implementing wheel calibration).
15th of March	7h 30m	Worked on Software Implementation.
16th of March	30m	Added some comments to the code.
16th of March	3h 40m	Debugging of calibration and added motor stall support.
17th of March	20m	Improved comments of the code.
18th of March	40m	Worked on the Software Implementation phase documentation.
Subtotal	44h 20m	**Software Design and Implementation Phases**
22nd of March	7h 30m	Worked on general improvements and summaries of the documentation.
23rd of March	2h	Started on the reflection.
24th of March	50m	Finished my reflection.
Subtotal	10h 20m	**Finalization of the documentation**
31st of March	4h	Attended final presentations and their last preparations.
Total	107h 40m	

B.2 Abdel

Up to date until 22nd of March.

Date	Time spend	What
2nd of February	2h	First group meeting.
3rd of February	2h 15m	Read the Project Guide
3rd of February	1h 15m	Started working on the abstract
4th of February	1h 30m	Finished the abstract.
4th of February	50m	Wrote my self-reflection
Subtotal	7h 50m	**Pre-orientation phase**
16th of February	7h 40m	Worked on the project as a group: Discuss Work Plan and started to work on the internal presentation about the V-Model for Sotware Development
16th of February	1h 40m	Prepared presentation about the V-Model
17th of February	3h 20m	Worked on the project as a group: We did the internal presenations + started designing the sorting machine
17th of February	1h 20m	Worked on Work plan at home
17th of February	2h 20m	Continued working on Work plan at home
18th of February	1h 20m	Finished orientation phase deliverable
19th of February	3h 20m	Worked on the project as a group: Continued working on the machine design
Subtotal	21h	**Orientation phase**
22nd of February	2h 15m	Preparing midterm presentation.
23rd of February	9h 30m	Did the midterm presentations + Worked on the project as a group: Started with the software Specification (finite automaton) + made some changes to the machine design
24th of February	3h 20m	Worked on the project as a group: Flashing the custom firmware onto the brick (mostly done by Adriaan) + Discussed the machine design deliverable + made some changes to the machine design
24th of February	30m	Implemented my logbook into the LaTeX file
Subtotal	15h 35m	**Machine Design phase**
26th of February	2h	Worked on the project as a group: Specifying the software structure based on the finite automaton
26th of February	1h 45m	Meeting Training (STU) + Tests of inputs
29th of February	1h 15m	Read about UPPAAL / studied tutorials at home

1st of March	6h 40m	Group meeting/Worked on the project as a group: Worked on the UPPAAL Model and started with Software Design
Subtotal	11h 40m	**Software Specification phase**
2nd of March	3h	Group meeting/Worked on the project as a group: Software Design / Implementation
4th of March	3h 40m	Group meeting/worked on the project as a group (some group-members worked from home): Software implementation and testing the software
8th of March	7h 40m	Group meeting/Worked on the project as a group: Software Design / Implementation
9th of March	3h 50m	Group meeting/Worked on the project as a group: Software Design / Implementation. Focus was on the UPPAAL Model
9th of March	2h 15m	Continued working on the UPPAAL Model at home
14th of March	2h 40m	Worked at home on the software design deliverable and did some tests with the UPPAAL model + changed some things / read some documents
15th of March	7h 30m	Group meeting/Worked on the project as a group: Software Design (worked on the software design deliverable) / Implementation (did some testing / added some things) + made pictures and videos for the final presentation
15th of March	2h 50m	Continued working on the software design deliverable at home + updated logbook + added minutes form last week to overleaf
16th of March	3h 40m	Group meeting/Worked on the project as a group: Continued working on the software design deliverable / added some pseudo code + tweaked the machine design a little bit
17th of March	3h 20m	Continued to work on the software design deliverable at home
18th of March	3h 10m	Had a group discussion on hangout and continued working on the software design deliverable and started working on system validation and testing (UPPAAL)
19th of March	3h 30m	Finished the software design deliverable + my part of the system validation and testing + made some changes/added some things to software specification

22nd of March	7h 30m	Group meeting/Worked on the project as a group: Processed the feedback given by our tutor (final report)
23rd of March	3h 40m	Group meeting/Worked on the project as a group: documentation, discussing presentations
28th of March	3h 20m	Made some changes / improvements to software design deliverable
30th of March	1h 20m	Made some changes / improvements to the final report
31st of March	4h 30m	Attended the final presentations of the project
1st of April	1h 40m	Worked on self reflection
7th of April	1h 25m	Worked on the final report one last time
Subtotal	70h 30m	**Software Design, Integration and Implementation phase**
Total	126h 35m	

B.3 Ivo

Up to date until 8th of April.

Date	Time spend	What
2nd of February	2h	First group meeting.
5th of February	10h	Finished abstract.
12th of February	2h 20m	Worked on self-reflection.
17th of February	30m	Finished and handed in self-reflection
Subtotal	14h 50m	**Pre-orientation phase**
16th of February	7h	Worked on the project as a group.
16th of February	1h	Working on personal and group logbook.
16th of February	2h	Prepared presentation about software Validation and Testing.
17th of February	3h 20m	Worked on the project as a group.
19th of February	3h 20m	Worked on the project as a group.
19th of February	1h	Stayed to keep working on the minutes
19th of February	40m	Worked some more on the minutes
20th of February	40m	Finished the minutes of week 2 and informed the group about it.
22nd of February	45m	reading things for the presentations and the presentations form the others so far.
22nd of February	1h 40m	excluding the UPPAAL part, completed my part of the presentations.
22nd of February	50m	worked on the UPPAAL part of the presentation.
23rd of February	3h	Meeting with tutor + preparing for presentation + did the midterm presentations.
Subtotal	25h 15m	**Orientation phase**
23rd of February	6h 40m	Worked on the project as a group: Started with the software Specification (finite automaton) + made some changes to the machine design
23rd of February	10m	Stayed another 10 min to work a bit on the Finite State Automaton.
24th of February	3h 30m	Worked on the project as a group: Flashing the custom firmware onto the brick (mostly done by Adriaan) + Discussed the machine design deliverable + made some changes to the machine design + making a first build of a complete Finite State Automaton for our sorting machine.
Subtotal	10h 20m	**Machine Design phase**
26th of February	2h	Worked on the project as a group: Specifying the software structure based on the finite automaton

26th of February	1h 45m	Meeting Training (STU) + Tests of inputs
1st of March	7h 30m	Group meeting/Worked on the project as a group: Worked on the UPPAAL Model and started with Software Design
2nd of March	4h	Resumed software design and implementation.
4th of March	3h 30m	Worked on the software implementation of the input and output. (Left a bit earlier then the rest because I had to catch my train.)
8th of March	6h 40m	Meeting with tutor + I updated the minutes + I converted my personal logbook from the informal version in word into the LaTeX file shared on overleaf.
9th of March	3h 50m	Worked on the project as a group
15th of March	7h 30m	Meeting with tutor + Worked on the project as a group. I made the agenda's of this week and the minutes of today. + I wrote UPPAAL queries for verification.
15th of March	30m	Worked on the UPPAAL queries for another 30 min.
16th of March	3h 40m	Group meeting/Worked on the project as a group: Continued working on the software design deliverable / added some pseudo code + tweaked the machine design a little bit + started on system validation and testing.
17th of March	2h 30m	Worked on peer review and System Validation and testing.
18th of March	3h 20m	Worked on System Validation and Testing.
19th of March	1h 30m	worked on System Validation and Testing.
22nd of March	30m	Worked 30 min on System Validation and Testing before the meeting.
22nd of March	6h 30m	Meeting and worked on the project with the group.
22nd of March	3h 20m	Still working on System Validation and Testing.
23rd of March	4h 20m	Meeting + I finished documenting my part of System Validation and testing. + updated my logbook.
25th of March	3h	Finished my final self-reflection document.
7th of April	3h 35m	worked on the conclusion of the project file.
7th of April	5m	finalized my logbook.
Subtotal	69h 35m	**Software, verification and completion phases**
Total	120h	

B.4 Jolan

Up to date until 7th of April.

Date	Time spend	What
2nd of February	2h	First group meeting.
3rd of February	1h 30m	Read the Project Guide
4th of February	2h	Finished the abstract
5th of February	1h	Wrote my self-reflection
Subtotal	6h 30m	**Pre-orientation phase**
16th of February	7h	First real meeting with the group, worked on workplan and presentations (uppaal and software design)
17th of February	3h 20m	Worked on the project as a group: We did the internal presentations + started designing the sorting machine
19th of February	3h 20m	Meeting: extended the sorting wheel, connected the motor
Subtotal	13h 40m	**Orientation phase**
22nd of February	30m	Prepared agenda for next meeting
23rd of February	10h	Listened to midterm presentations, worked on the machine, almost finished the machine design document (this needs checking with the group). In the evening, made agenda for next meeting
24th of February	3h 20m	Meeting, Discussed the machine design deliverable, matched it up with the machine itself
Subtotal	13h 50m	**Machine Design phase**
26th of February	2h 30m	Made agenda for the meeting, did the meeting, finishing up on the machine design
26th of February	1h 45m	Meeting, Training (STU) + Tests of inputs
1st of March	6h 40m	Meeting
Subtotal	10h 55m	**Software Specification phase**
2nd of March	3h	Meeting, started programming the states
4th of March	4h	Worked from home on states and in- and output, almost finished
8th of March	7h 40m	Meeting, worked further on software implementation and design
9th of March	3h 50m	Meeting, worked on coloring the buttons
15th of March		I was ill
16th of March	3h 40m	Meeting, tweaked machine design, started on the presentation and started on system validation and testing
18th of March	4h	Worked from home on System validation and testing, finished code review

22nd of March	9h 30m	Working on test cases, fixed up logbook, added machine design choices
23rd of March	4h	Meeting, worked on presentation
25th of March	4h	Worked at home on self reflection and presentation
26th of March	2h	Worked on presentation
27th of March	2h	Worked on presentation
29th of March	3h	Worked on presentation
30th of March	3h 50m	Met with Bogdan and Ivo, working on presentation
31st of March	4h	Practiced for presentation and gave it
Subtotal	58h 30m	**Software Design, Integration and Implementation phase**
Total	103h 25m	

B.5 Bogdan

Up to date until 23rd of March.

Date	Time spend	What
2nd of February	2h	First group meeting.
3rd of February	2h	Read the Project Guide
3rd of February	1h	Started working on the abstract
4th of February	1h 30m	Finished the abstract.
4th of February	2h	Wrote my self-reflection
Subtotal	8h 30m	**Pre-orientation phase**
16th of February		absent the whole week because I was not in the country due to my flight being canceled
Subtotal		**Orientation phase**
23rd of February	9h 30m	Listened to the midterm presentations + Worked on the project as a group: Started with the software Specification (finite automaton) + made some changes to the machine design
24th of February	3h 20m	Worked on the project as a group: Flashing the custom firmware onto the brick (mostly done by Adriaan) + Discussed the machine design deliverable + made some changes to the machine design
24th of February	10m	Implemented my logbook into the LaTeX file
Subtotal	13h	**Machine Design phase**
26th of February	2h	Worked on the project as a group: Specifying the software structure based on the finite automaton
26th of February	1h 45m	Meeting Training (STU) + Tests of inputs
29th of February	5h	Worked on my deliverable, Software Specification
1st of March	6h 40m	Group meeting/Worked on the project as a group: Worked on the UPPAAL Model and started with Software Design
3rd of March	8h 20m	Finished Software Specification and implemented it in the final report and emailed it to our tutor
Subtotal	23h 45m	**Software Specification phase**
2nd of March	3h	Group meeting/Worked on the project as a group: Software Design / Implementation
4th of March	3h 40m	Group meeting/worked on the project as a group (some group-members worked from home): Software implementation and testing the software

8th of March	4h 20m	Group meeting/Worked on the project as a group: Software Design / Implementation
9th of March	3h 50m	Group meeting/Worked on the project as a group: Software Design / Implementation. Focus was on the UPPAAL Model
14th of March	2h 40m	Worked at home on the software design deliverable and did some tests with the UPPAAL model + changed some things / read some documents
15th of March	7h 30m	Group meeting/Worked on the project as a group: Software Design (worked on the software design deliverable) / Implementation (did some testing / added some things) + made pictures and videos for the final presentation
16th of March	3h 40m	Group meeting/Worked on the project as a group: Continued working on the software design deliverable / added some pseudo code + tweaked the machine design a little bit
18th of March	3h 10m	Had a group discussion on hangout and continued working on the software design deliverable and started working on system validation and testing (UPPAAL)
22nd of March	5h 50m	Group meeting/Worked on the project as a group: Processed the feedback given by our tutor (final report)
22nd of March	3h 20m	Finished the PowerPoint presentation
23rd of March	3h 20m	Prepared my speech for the final presentation
Subtotal	44h 20m	**Software Design, Integration and Implementation phase**
Total	89h 35m	