Final report 2IO70
aka
Reinhard Heinrich Bertram Vinzenz Freiherr von Pelden genannt Cloudt zu Lauersfort und Impel

Sergio van Amerongen	Stefan Cloudt	Daan de Graaf
0952200	0940775	0956112
Robert van Lente	Tom Peters	Berrie Trippe
0953343	0948730	0948147

Responsible:

Daan de Graaf

`d.j.a.d.graaf@student.tue.nl`

March 30, 2016

Contents

Chapter 1

Introduction

On an Easter morning, you are rudely woken by the merciless beeping of your alarm clock. In a reflex, you grab your phone from the nightstand to check up on your social media. You catch a glimpse of the time on your phone, and cry out in horror as you realise that your alarm clock does not account for daylight savings. Fueled by adrenalin you sprint downstairs, and are subsequently greeted by the smell of fresh coffee, which your coffee machine is programmed to make you every day. You resist your craving for caffeine and enter your car. You enter your sister's new address on your satellite navigation and drive off in silence, as your little one recently discovered the disc drive slot on the car stereo and decided it would make an ideal place to dispose of chewing gum. As much as that confused the car stereo, your GPS system is equally bewildered to find you taking a right turn where you should only have gone slightly right.

Nowadays, embedded systems are ubiquitous. They make sure you awake in time, satisfy your caffeine addiction and guide you from A to B. These devices all work flawlessly, were it not for one misbehaving part: Us. These machines have to work with our complicated time zones and daylight savings, deal with improper usage and even know the things we forget to tell them in order to do their jobs right! Surely, we can not expect any machine to account for all of this? Therefore, the best we can do is make machines as fault-tolerant as possible, and clearly communicate with the user when something inevitably goes wrong.

This project is about sorting black and white discs. The machine made in this project is not only able to sort, but it can also detect errors during its process and give an adequate response. The machine is therefore able to tell something about its operating state, which is considered difficult.

Chapter 2

Workplan

2.1   Introduction

This document contains the workplan. We have documented what will be done when and by who. It is assumed that we have 96 man hours available every week for 7 weeks, because we have 4 timeslots, 4 hours per slot and 6 people. According to this assumption, deadlines are chosen which are also documented in this document. For each deliverable document, one person is pointed responsible for that document.

2.2   Absence

All group members will be present on times the group decides to meet and during the tutor hours. Exceptions are noted in the table below. All of those cases of absence are approved by the tutor.

Who	When
Stefan	5-2-2016 after 11:00
Stefan	From 16-2-2016 untill 22-2-2016

2.3   Deadlines of deliverables

We have included two deadlines. One deadline for which we want to have a finished document and one deadline for handing in the document to the tutor. The time in between of those two dates is the time spend to solve unforeseen situations, for example illness or travel problems.

Document	Responsible	Finished	Delivery
Abstract	Individual	5-2-2016	5-2-2016
Self-Reflection	Individual	12-2-2016	12-2-2016
Workplan	Stefan, Tom	19-2-2016	19-2-2016
Machine Design	Robert	19-2-2016	26-2-2016
Software Specification	Tom	26-2-2016	4-3-2016
software Design	Stefan	4-3-2016	11-3-2016
Software Implementation	Berrie	11-3-2016	18-3-2016
System Validation and Testing	Sergio	18-3-2016	25-3-2016
Final Report	Daan	7-4-2016	7-4-2016

2.4   Presentations

There will be three internal presentations by the group members for the group members. These presentations are used to decide who will do the midterm presentations and who will do the final presentations.

Subject	Who	Date	Time
V-Model	Stefan, Tom	5-2-2016	9:15
Testing	Daan, Sergio	5-2-2016	9:30
UPPAAL	Berrie, Robert	15-2-2016	13:00

As decided on 15-2-2016 after the internal presentations, the following presentations will be as shown in the table.

Presentation	Who	Date
Midterm	Sergio, Tom and Berrie	23-2-2016, 25-2-2016 or 26-2-2016
Final	Stefan, Daan and Robert	29-3-2016, 31-3-2016 or 1-4-2016

2.5   Tasks

Machine Design.

Task	Estimated	Dependent
Experiment with LEGO	18	
Define SLR	36	
Define machine interface	12	SLR
Creating documents	6	All tasks of phase

Software Specification

Task	Estimated	Dependent
Define inputs	3	Machine interface
Define outputs	3	Machine interface
Define dependency of input output	6	Define inputs, define outputs
Finite State automaton	12	Define dependency of I/O
UPPAAL model	18	Finite state automaton
Creating documents	6	All tasks of phase

Software Design

Task	Estimated	Dependent
Class diagram	18	
State machines	6	
Message sequence charts	12	
Pseudo code of functions	12	All above
Creating documents	6	All above

Implementation

Task	Estimated	Dependent
Coding	1	
Writing code	24	Coding standard

System validation and testing

Task	Estimated	Dependent
Machine Design test	8	Machine design
Software Specification test	12	Software specification
Software Design test	6	Software design
Implementation test	12	Implementation
Fixing failed test	18	One of the above

Completion

Task	Estimated	Dependent
Final report	36	Everything of other phases
Final presentation	24	Everything of other phases

2.6   Role division

The roles of the group members are as shown inside the table.

Week	President	Secretary	QA Manager	Materials Manager
1	Daan	Sergio	Robert	Berrie
2	Tom	Sergio	Robert	Berrie
3	Stefan	Berrie	Robert	Berrie
4	Sergio	Stefan	Daan	Berrie
5	Robert	Tom	Daan	Berrie
6	Berrie	Daan	Sergio	Berrie
7	Daan	Robert	Sergio	Berrie

2.7   Tutor Meetings

Weekly two tutor meetings are planned. Wednesday at 14:15 and on Friday at 13:00.

Chapter 3

Machine design

3.1   Introduction

We started our project by determining the features that we wanted to implement in our sorting machine, like user interaction and exception handling. These design decisions form the foundation of our mechanical machine design. The purpose of this document is to record the design decisions we made in the first stage of the project, from a mechanical point of view as well as a user point of view.

3.2   Design requirements

In order to create a functioning sorting machine that can sort black and white discs, at least the following components are required.

- A container for storing unsorted discs

- Two containers for storing the sorted discs

- A sensor that can distinguish the different colour discs.

- A transporter that is able to move the discs from the container to the sensor.

- A transporter that is able to move the discs from the sensor to the appropriate container.

3.3   Final Design

The machine consists of a vertical tube in which the discs can be stored. A pin can be removed from the bottom of the tube, allowing the discs to fall on a platform below, one by one. A colour sensor is attached underneath this platform, such that it can detect the colour of the disc. Next to this platform, a step motor is attached that rotates a wheel with three arms. These arms can push the disc of the platform, either on the left or the right side. Below the platform, a seesaw is attached with both sides of the same length, such that the discs that fall on the platform can fall on these arms. A gyroscope is attached to the axis of this seesaw, such that rotations of the seesaw can be detected. The discs then slide off the seesaw, allowing it to stabilize, and fall into one of two mountable trays.

3.4   Design Decisions

The machine went through several iterations before we settled on the final design. The decisions that we made during the design process are listed below.

3.4.1 Position of the colour sensor

We had the option to attach the colour sensor in various ways to the machine. We chose to put it underneath the platform with the sensor pointing up, in such way that the arm of the motor could easily wipe the discs away. Placing the sensor below the discs makes the sensor more reliable, since the surface of the discs on the bottom is larger than the surface on the side of the discs. It would also have been significantly more difficult to mount the colour sensor in a sideways or upright position because of the structural limitations of the lego construction set.

3.4.2 Container

The decision to use a tube-like structure was made early on, as it is the most compact way to store the discs. The vertical storage also has the added benefit of removing the need to create an additional transporter to move the discs from the container to the sensor, since a new disc will simply fall in place when the previous disc is moved away.

The height between the tube and the platform is designed to be slightly higher than the height of one disc. This ensures that only one disc will be moved from the tube at a time.

3.4.3 Rotating arm

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

3.4.4 Seesaw

Initially, the plan was to use touch sensors to detect if a disc arrived in the correct tray. However, we quickly found that the discs did not weigh enough to trigger the touch sensors. Therefore we finally settled with the seesaw and a gyroscope attached to it.

3.4.5 Trays

We chose to use coffee cups for the trays in the machine, since these where readily and freely available. Other options would be buying plastic trays or building trays out of LEGO, which was not possible because there was not a sufficient number of parts in the construction set.
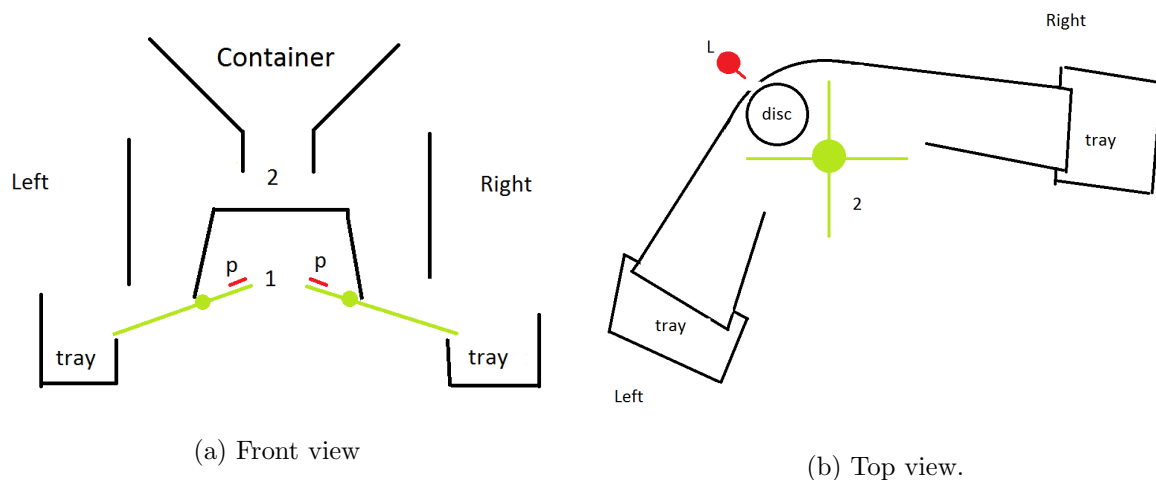

(a) Front view

(b) Top view.

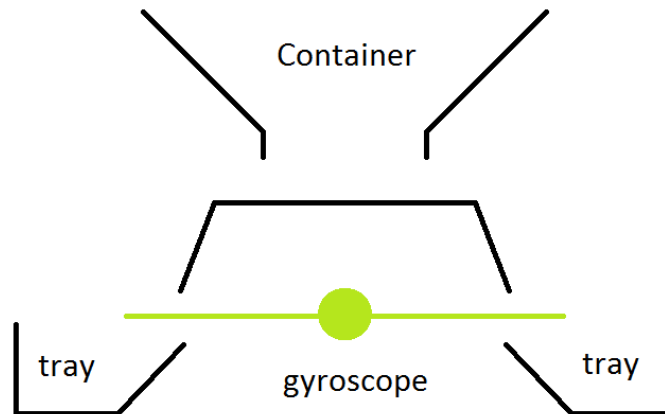Figure 3.1: The initial design of the machine.

Figure 3.2: The final front view of the machine.

3.5 System Level Requirements

3.5.1 Use Cases

The machine can be described as a finite automaton. This finite automaton is shown in figure 3.3. The machine has three buttons, which the user can utilize to control the machine. The START/PAUSE button is used to start and pause the machine. The ABORT button is used to stop the machine in case of emergency. The RESET button is used to reset the machine. The user is required to add black or white discs to the container before starting the machine. The machine, when finished without fatal error, produces two trays, one filled with white and one filled with black discs. The machine will also display information about its current state and progress on the display during the sorting process.

The sorting process is described by the following automaton. The initial state is the mode selection state. In this state, the mode in which the machine operates can be selected. These modes are explained further in the document. After the mode has been selected, the machine goes to the initial state of the sorting process. This initial state is the rest- ing state. The user is required to prepare the machine as described in the User Constraints segment further in the document. None of the buttons should have any effect, except for the START/PAUSE button, which starts the sorting process and the ABORT button, which triggers an abort. When this button is pressed, the machine should proceed to the operating state.

In this state, the machine sorts the discs. Internally, it has a state in which it checks the color of the disc, after which it moves to the correct sorting state in which it transports the disc to the correct tray. If the RESET button is pressed in this state, the machine should return to the resting state after finishing its current cycle.

While in the operating state, if the START/PAUSE is pressed, the process should halt after it has finished its current cycle and the machine should go into a paused state. When the START/PAUSE button is pressed again, the machine should return to the operating state at the point where it was when the START/PAUSE button was first pressed. If the RESET button is pressed in this pausing state, the machine should return to the resting state.

While in the operating state, if the ABORT button is pressed or a fatal exception is encountered, the machine should immediately go into an exception state. In this state, the machine must come to a full stop. The machine should only continue when the RESET button is pressed, after which it returns to the mode selection state.

When the machine finds that there are no more discs left in the storage, it should go into a finished state. Then, if the RESET button is pressed, the machine should return to the mode selection State.
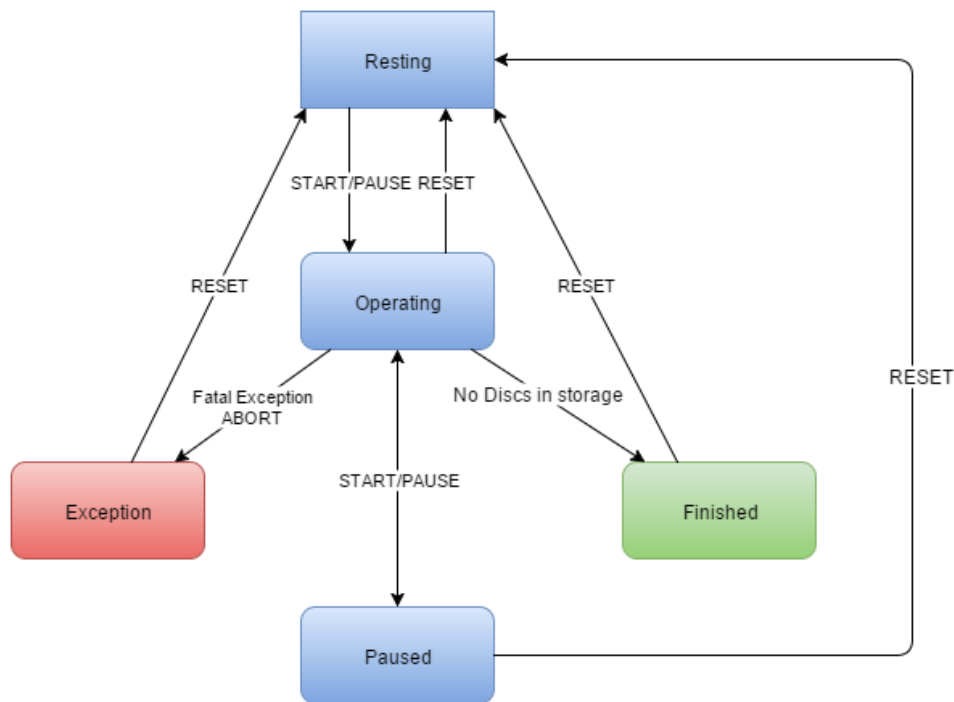
Figure 3.3: The finite state automaton of the machine.

3.5.2 User Constraints

Machine Preparation

Before starting the operation of the machine with the START/PAUSE button in the resting state, the user is expected to fill the disc storage with black and white colored discs that need to be sorted. Only discs are allowed in the disc storage and no other objects should be placed in it. The user must also ensure that no discs are present in other parts of the machine and that the disc trays are mounted to their mounting points prior to starting execution. The disc trays should be empty. At most 12 discs may be placed into the container. The user then has to proceed by starting the program. At this point, the machine will calibrate the arm to the standard position. After the calibration, the tube can be filled with discs by pulling out the plug.

After the machine has finished execution, the user will have access to two trays, one containing exclusively white discs, the other only contains black discs. When the machine indicates that it has finished its sorting procedure, the user should remove the trays from their mounts so he can dispose the discs somewhere else.

Exceptions

Possible errors are as follows:

Error type	Fatal (= go to abort state)
Disc does not reach the tray	Yes
Disc arrives in the wrong tray	No
Time to tray is higher than average	No
Wrong input (i.e. different colour disc)	No
Motor jams	Yes
Gyroscope does not stabilise	Yes
Battery low	Yes
Abort by user	Yes
Software exception	Yes

When a disc does not reach the tray, the machine should stop, as there is may be an obstruction in the path the disc takes from the sorting arm to the lever.

When a disc arrives in the wrong tray, the machine should stop, as something happened that caused the disc to move to the wrong tray, even though the sorting arm tried to move it to the correct tray. This means there must be a mechanical failure somewhere. However, because of the limitations the gyroscope and lejos give us, we can not thrust this check entirely. Therefore, we have decided to make this a non-fatal error instead, so the machine doesn't halt in the wrong cases.

The Wrong Input error occurs when a disc is being detected with another color than white or black. This could, for example, be a disc with the color red. This is not a fatal error because we can specify an action that will be taken when this exception occurs. We may for example want the machine to simply put the unknown disc in one of the trays anyway, say default the destination to white, or we dispose all unknown discs into a separate tray.

The machine will give a warning when the battery has less than 5% of charge left. This is done to prevent the machine running out of power in between operations and ending in an invalid state. The user should act appropriately when this happens.

When an error occurs in the machine that is indicated with fatal, the user must remove all discs from all parts of the machine. To help the user with this, the machine has a build-in function that spits out all the discs to the right when pressing the ABORT button in the mode selection state. The user should first make sure this can be done safely and without harming the machine. The user should then press the RESET button in order to put the machine in the mode selection state again. The user may then proceed to prepare the machine for execution again, as described above, if desired.

Safety properties The machine should have the following safety properties. These are our guarantees about the working and stopping conditions of the machine, in case something unexpected happens.

- The ABORT button must stop all moving parts within 10 ms. This to make sure that the user is able to quickly stop the machine in case something can not be handled by the Machine.

- When the machine starts with discs loaded into the container, then the discs will be sorted when it arrives in the finished state. The sorting process takes at most 5 minutes in the default safe mode. However software specification may specify more modes.

- A fatal exception must stop all moving parts within 10 ms. This ensures that the machine does not damage or even destroy itself when it is blocked in any way.

3.6   Machine Interface

3.6.1   Lejos API

The Lejos API provides access to buttons on the brick, sensors and actuators. Buttons can be queried directly, but references to sensors and actuators must be instantiated with a Port parameter indicating how they are connected to the brick. The following ports are used in our design:

Peripheral	Port
Colour sensor	Port 1
Gyroscope	Port 2
Touch sensor	Port 3
Motor	Port A

Motors expose methods that allow the developer to have to motor rotate an arbitrary number of degrees. Therefore, it is not necessary to implement pulse width modulation. One can call a method to fetch a sample from a sensor in the form of an array of floats. It is up to the developer to parse these in a meaningful way.

3.6.2   Peripherals

Motor rotating arm

This motor is connected to port A and rotates the arm which moves the discs. The arm has three legs with an angle of 120 degrees. The motor has basically three states: a resting state and two working states, one for every possible direction. The neutral position of the three legged arm is the position where the two arms closest to the place where the disc will land have an approximately identical distance to the disc.

When the motor is in the resting state the three legged wheel has this neutral position. When the motor is in a working state the arms rotate left or right until the next neutral position of the wheel (120 degrees further), where it will transition to the resting state of the motor. During the working state the motor should not be jammed or obstructed in any way, or the motor will be jammed and trigger an exception.

Touch sensor calibrator

The touch sensor, connected to port 3, is attached in such a way behind the sorting platform that the arms of the sorting wheel are able to fully press the sensor when it rotates. This way, it is possible to detect the position of the sorting wheel when it is rotating, which can be used to calibrate the sorting wheel through software.

Gyroscope with seesaw

To the seesaw a gyroscope is attached, which is wired to port 2. The seesaw has three states, a neutral state, a left-side down state and a right-side down state. The neutral state is the state in which the gyroscope when read returns a value indicating that it is in balance. The seesaw enters a left-side down state when a disc falls on the left side. The gyroscope will read a value indicating that it was unbalanced and that the left side went down. This is analogous for the right side down state. The gyroscope is unreliable at best: Its value tends to drift at an arbitrary rate, making the gyroscope unreliable if not unusable in some cases. While the seesaw is in motion no further discs should be sorted, as it will not be possible to accurately measure the angle rate change. Extensive testing suggests, although it is not documented anywhere, that

the gyro sensor is much more reliable if one at first calibrates the sensor by quickly switching from rate change mode to angle[1] and back. If the sensor is then kept completely stable, a lower drift rate may be acquired.

Colour sensor

The colour sensor, connected to port 1, gives a value indicating the grayscale of the surface above it. The colour sensor returns a float indicating the grayscale (e.g. 0 uptil 1) of the disc currently on the platform, or the grayscale of the environment, which means that there is no disc above it. After extensive testing, we found that the following values are usefull:

Colour disc	Value
None	0.12
Black	0.15
White	0.86

However, in the mode selection state, these values can be calibrated by pushing the button UP. The user has to put a black disc, then a white disc on the sensor and wait for the program to finish. The values will be changed according to the values found. If no calibration is done, the values of last calibration will be used.

The brick

The brick controls the sensors and actuators. The state of the buttons is either up or down, which may be queried using the Lejos API. The buttons used are shown in figure 3.4.
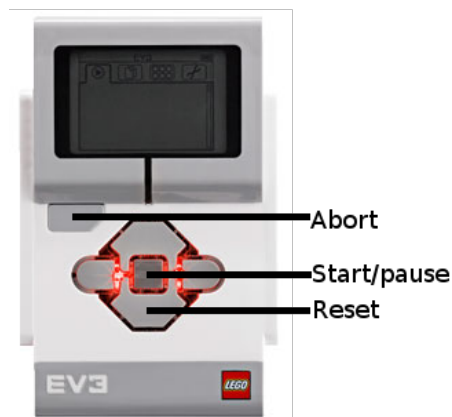

Figure 3.4: The buttons on the brick.

3.7 Conclusion

The design decisions recorded in this chapter will be built upon in the next phase of the project, where we are going to specify the software specifications. The software specifications are heavily dependent on the machine interface described in this chapter, since the machine interface describe our assumptions about the machine API and the mechanical implementation. The software specifications will describe how we are going to use this interface and how the machine will behave.

---

[1]The gyro sensor has both an angle and a rate change mode, which return the measured angle and the rate of change of this angle, respectively. The angle mode is so unreliable that it is considered unusable for our purposes

Chapter 4

Software specification

4.1   Introduction

After having designed the machine, we are able to specify how our software should behave. We do this by describing the software in a state machine with abstract states. This state machine also implements the different operation modes. After this we characterise the distinct inputs and outputs. We will convert the state diagram into a working UPPAAL model and explain the working systems in detail. Finally we explain the verification of our model. This chapter is called the software specification.

4.2   Operation modes

The gyroscope can be quite a bottleneck in the sorting process, as it is rather slow. If we want to sort very fast, we should skip the input from the gyroscope and sort the discs immediately after each other. Therefore, we want to implement 2 modes: a fast mode in which we dont check for input from the gyroscope and thus dont check if the disc actually reaches the tray, and a slower safe mode, where we check for input from the gyroscope and we can make sure that discs do reach the tray. Besides this, we want to make an additional incremental mode for debug purposes, in which the user has to press a button for the machine to sort the next disc.

4.2.1   Fast mode

The machine starts in a resting state. If the Start/Pause button is pressed in this state, the machine should go to the read color state. In this state, the machine checks which color disc is present on top of the light sensor. There are four different transitions the machine can make at this point, depended on the detected colour. If the machine is black or white, it should go to a respective state in which the motor is rotated accordingly. When the motor has finished moving, the machine simply returns to the read color state. The machine can be paused by pressing the Start/Pause button anytime during the sorting process. The machine then goes to a paused state as soon as it enters the read color state and waits for the Start/Pause again to return to the checking process.

4.2.2 Safe mode

The safe mode is exactly the same as the fast mode, with a few additional safety checks. This means that it has the same assumptions according to the input. There is still an arrow from every state except the initial state to the abort state for when the ABORT button is pressed. There are two additional states. One for each colour, black or white. After the motor is finished turning, it enters a state in which it waits for input from the gyroscope (G), which can indicate a disc that felt to the right (R) or to the left (L). When the input is correct, it goes to the state in which the colour is checked. When the input is wrong (Exception) (wrong side, no input at all after a while), the machine goes to the abort state.

4.2.3 Incremental mode

The incremental mode is again almost the same as the safe mode, so again the same assumptions about the input and the state machine, but now there is 1 extra state between the ingoing arrows to the state in which the colour is checked. In this state, the machine waits for a press on a button, the START/STOP button (S/P) to continue. If that button is pressed, it goes to the state in which the colour is checked. All the ingoing arrow that were going to the colour-check state are now going to the wait-for-button-press state. There is 1 exception for this. From the resting state, you go to the colour-check state immediately after a press on the button. It would be silly if you would have to press that button twice for something to happen.

4.3 Signals

4.3.1 Inputs:

Color c:

The input from the sensor with the following possible values: "N", "W", "B" and "U". The value "N" implies that no disk has been detected in front of the colour sensor. The value "W" implies that a white coloured disk is in front of the sensor. The value "B" implies that a black disk is in front of the sensor. Finally a value "U" means that an unknown disc is in front of the sensor.

Motor M:

The input from the motor can be that it is jammed, J, or that it is finished, F. We use these inputs as a method to detect if we can go to the next state, or to detect if there is a possible error.

Abort A:

This signal is high when the abort button has been pressed. This signal indicates that the machine should be halted nearly instantaneously and that the machine should enter the Abort state.

Reset R:

This signal is high when the Reset button is pressed. The signal will be used to bring the machine back in its resting state and reset all the statistics about the current sorting operation.

Start/Pause S/P:

This signal is high when the Start/Pause button is pressed. This signal will be used to start and pause the operation of the machine, depending on its current state.

Gyroscope G: This signal might have three values "L", "R" and "S". "L" indicates that gyroscope is left, "R" indicates that the gyroscope is right and "S" indicates that the gyroscope is stable.

4.3.2   Outputs

Colour Sensor CS:

Whether the colour sensor is currently reading the sensor values. In order to read the values, a light from within the sensor has to be activated in order for the detection to work. This means that this light is an output for the machine. While the light is activated, the sensor is constantly reading the sensor values, but we only read it in certain situations. The colour sensor is only active while the machine is in the operating state and is therefore disabled in the Resting state, Abort state, Finished state and the Paused state.

Motor M:

This signal has three possible values "L", "R" and "O". Where "L" is the motor moving to the left for 120 degrees, "R" is the motor moving to the right for 120 degrees, "O" is the motor turned off. The sorting arm moves, depending on the colour read.

LED Active E:

The EV3 controller brick has a built-in LED behind the buttons that we can use to give feedback to the user. We can specify the flashing pattern from a limited set of patterns and we can specify the colour. The LED will emit a constant green colour while the machine is in the Finished state. The LED will flash in red while it is in the Abort state. The LED will flash orange when it is in a warning state.

Sound:

The brick also features a built-in speaker which we can use to give even more feedback to the user. We will use this speaker to let the user now what happens in the machine. So when there is an error, the machine will sound an error. When the machine finishes sorting it will play tiroler music.
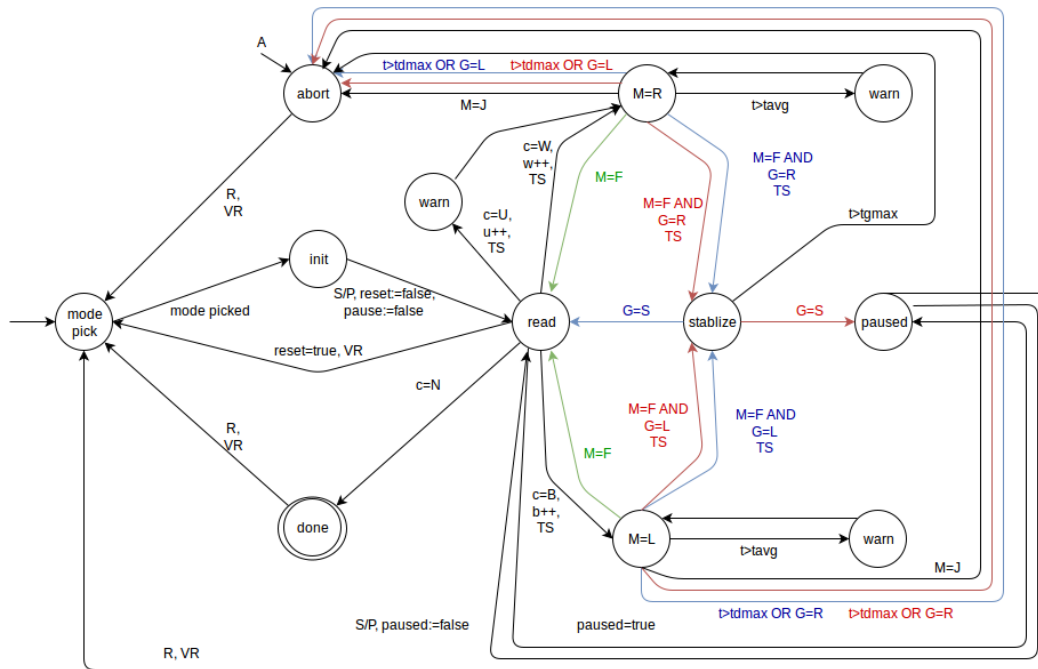
4.4 State Machine

Figure 4.1: This state machine defines the behaviour of the software. The three modes of our machine have been embedded into the figure. Black coloured transitions are done in all modes.

We have indicated the transitions for specific modes using a colour coding. Green transitions only apply to the fast mode. Red transitions only apply to the incremental mode. Blue transitions only apply to the safe mode.

    All symbols are inputs or outputs defined in the next chapter. However some symbols are not input or output but something else, these are documented here:

Symbol	Description
PS	Starts the peripherals in order to be able to use them. This means setting output CS to active and M=O
PR	Resets the peripherals to turn them off, sets the CS to non active and M=O
TS	Starts a timer which increments variable t at a fixed time rate
TR	Resets the timer by setting t to 0
t	The variable containing the current timer value
$tdmax$	The maximum timer value after which it is decided that a disk was lost
$tavg$	The average time of a disk to get in the basket
$tgmax$	The maximum time it should take to stabilize the gyroscope
$paused$	Variable indicating if the machine should go into the paused state or not
$reset$	Variable indicating if the machine should go into the reset state or not

4.5 UPPAAL model

An UPPAAL model consist of multiple systems that communicate over several channels. The systems in our UPPAAL model are Reinhard, our main system, along with several other peripheral systems.
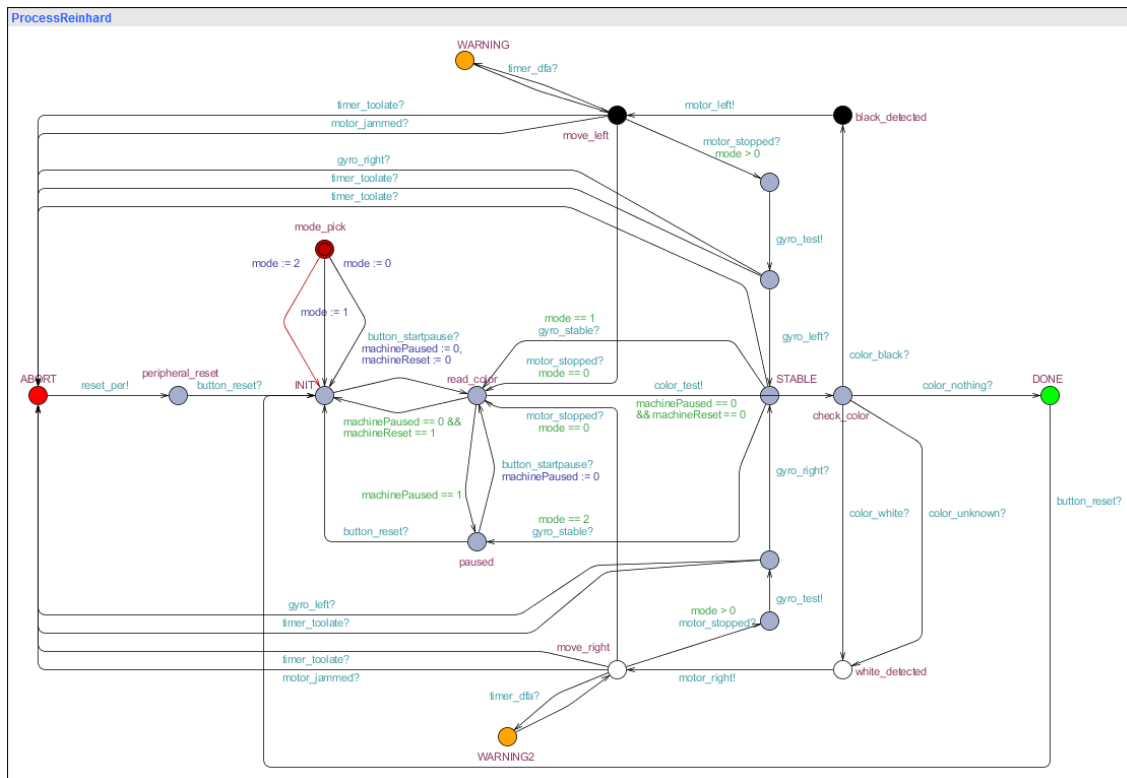
Figure 4.2: This state machine defines the behaviour of the software. The three modes of our machine have been embedded into the figure. Black coloured transitions are done in all modes.

4.5.1 Main process

Choosing the mode

The initial state is mode_pick, which is not present in the state machine. In this mode one of the three operation modes is chosen. This is important for the verification process, because the UPPAAL model uses an internal variable to store the mode, whereas the state machine indicates the different modes with colours. If it is not possible to set this variable in the model itself, the verification process will only verify if a query holds for one mode.

Initialiation

Upon choosing a mode, we enter the state INIT, which is equivalent to the initial state in the state machine. Here, the model waits for the button_startpause signal from ProcessButtons. Also, it sets the machinePaused and machineReset variable to 0, which ensures that the model does not immediately pause or reset.

Reading colours

Upon leaving the INIT state we enter the read_color state(read in the state machine). Whenever the machinePaused variable is 1 in the read_color state, the model goes to the paused state. If the machineReset variable is 1, the machine returns to the INIT state, but this has the lower priority. If both of these are 0, the color_test signal is send and the model transitions to the check_color state. In the check_color state it waits for a signal of the ColorSensor system depending on whether there is a disc and which colour the disc has. Depending on the colour, it goes to one of two parts of the main system, one for each distinct colour (black or white). We will describe one of these parts as they are mirrored. Upon receiving the signal of a colour, the

model transitions to a detected state. Here, it sends a signal to ProcessMotor to start moving in the appropriate direction and goes into a move state. The transition the model makes here is dependent on the mode variable.

Check sequence

If mode is not 0, indicating that the system is in the safe or incremental mode, it will add a check sequence using the gyroscope. After the signal from the motor is received that it has stopped, the main system will send a test signal to the GyroSensor waiting for a signal back. If the correct signal is received back, indicating that the disc has arrived in the correct tray, it will return to read_color state. If mode is 0, indicating that the system is in fast mode, this check is skipped and the model returns to the read_color state from the move state as soon as it receives the motor_stopped signal from ProcessMotor.

Finishing up

When the model receives the color_nothing signal from ProcessColorSensor, the machine is finished and the model goes to the DONE state. Here it waits for the button_reset signal and returns to the initial state.

Exceptions

In our machine design is it specified that there is an ABORT button present that will stop the whole machine almost immediately. However, we have chosen not to include the abort button in our UPPAAL model as a possible transition for every state because this would be rather cumbersome and just a small implementation. We have however chosen to include possible exceptions such as a jammed motor or a signal that has not arrived in decent time. These transitions will lead the system into the ABORT state. In this state the process waits for the reset button to give a signal, after which it will reset the other processes and return to the intitial state. Note that it is not required to actually reset the gyroscope in the actual machine, just in the UPPAAL model. Non-fatal exceptions are also represented in the model. Deviating average detection time is represented in the states WARNING and WARNING2, which the model reaches when the timer_dfa is send from ProcessTimer. Process ColorSensor can also send a color_unknown signal, which sends the machine to WARNING3. These warning states simply return to the state they were detected in and let the model continue.

4.5.2   Subprocesses

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

ProcessColorSensor

waits for the *color_test* signal. It can then send four different signals: *color_black*, *color_white*, *color_unknown*, and *color_nothing*.

ProcessMotor

waits for the *motor_left* or *motor_right* signal and moves to the respective state. From there, it can send either a *motor_stopped* or a *motor_jammed* signal. It also listens for a *reset_per* signal, after which it returns to the initial state.

ProcessGyroSensor

waits for the *gyro_test* signal from the main process. It can then send the following signals: *gyro_stable*, *gyro_left* and *gyro_right*. If either *gyro_left* or *gyro_right* is send, the machine goes to an additional state where it is able to send a *gyro_stable* signal, indicating that the gyroscope has returned to its initial position. The process also listens for the *reset_per* signal, for resetting purposes.

ProcessButtons

is continuously able to send button signals. This simulates that any button can be pressed at any time, which is necessary for the verifier to check all possible transitions.

ProcessTimer

works the same as ProcessButtons, but with signals that an internal timer would send if it detected one of the errors.

ProcessPauseListener

and **ProcessResetListener** continuously listen to the *button_startpause* and *button_reset* signals from ProcessButtons. When they detect the respective signals, they set the respective variables to 1 to alert the main process that the button is pressed.

4.6   Verification

The main reason for recreating the state machine in UPPAAL was to allow extensive verification of the model. With UPPAAL we can verify the events like deadlock will never occur. The UPPAAL query language is a minimalistic but functional language to specify these properties. We use the following queries to verify our model:

Abort state reachability

Query: E<> ProcessReinhard.ABORT
Description: There exists a path where the system reaches the ABORT state.
We want to ensure that the machine halts in case of a fatal error. This check ensures that it is theoretically possible for a fatal error to occur and jump to the ABORT state.

Peripheral reset

Query: A[] (ProcessReinhard.paused or ProcessReinhard.INIT) imply (ProcessMotor.INIT and ProcessGyroSensor.INIT)
Description: Whenever the system is in the paused or initialisation state, the motor and gyro sensor must be in their initial state.
The goal is this check is twofold: We want to ensure that all peripherals are idle when the machine is in the paused state or has not yet started, and we have to be certain that after reaching the ABORT state, all peripherals are correctly reset. If we would leave the gyroscope in its left or right state, we would cause deadlock in the next cycle. Worse still, leaving the motor running after an ABORT could harm to machine.
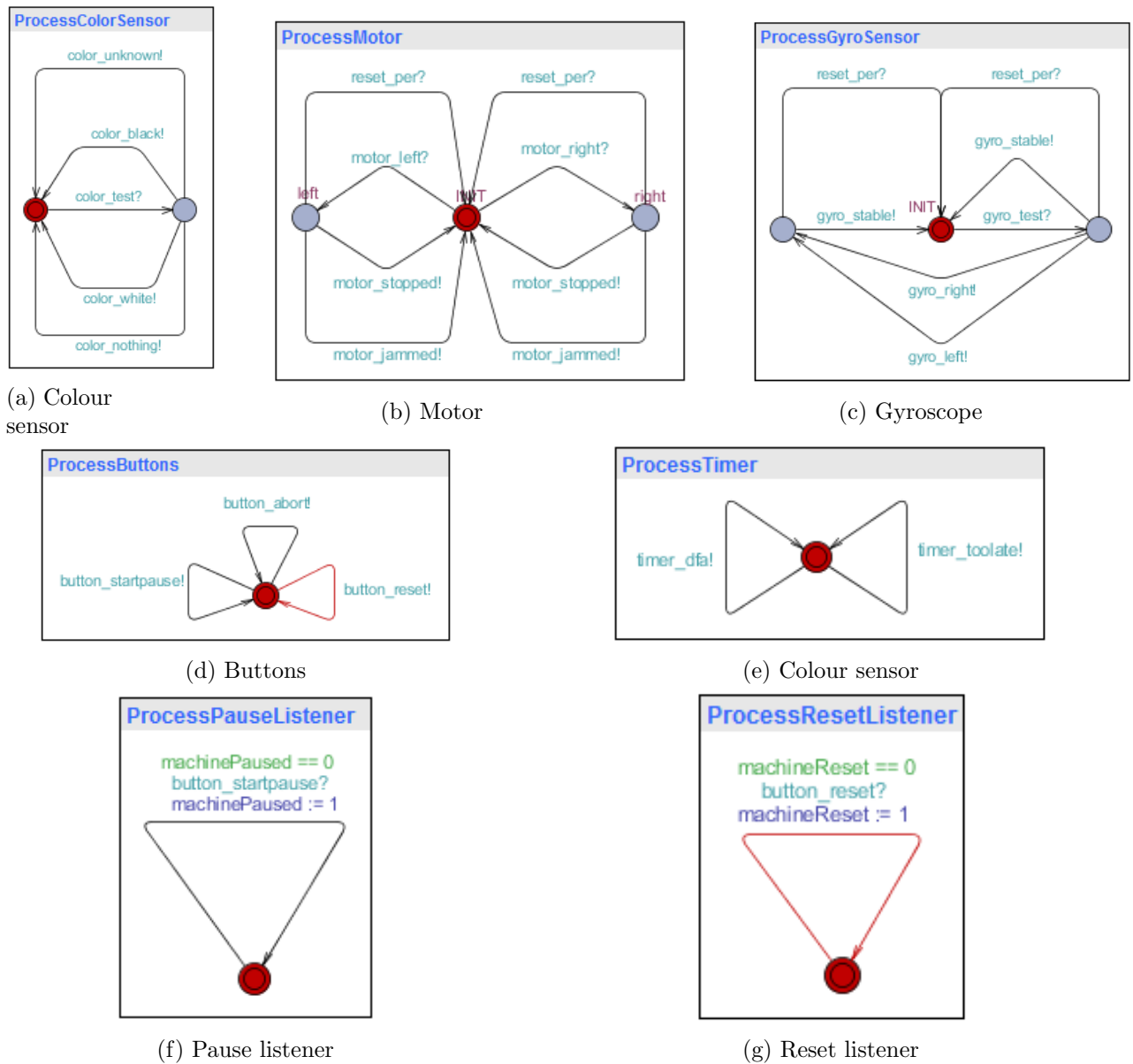
(a) Colour sensor

(b) Motor

(c) Gyroscope

(d) Buttons

(e) Colour sensor

(f) Pause listener

(g) Reset listener

Figure 4.3: Subprocesses

Full speed independent of gyroscope

Query: A[] mode == 0 imply ProcessGyroSensor.INIT
Description: When in full speed mode, the system must leave the gyroscope in its initial state. Full speed mode is supposed to ignore the gyroscope to speed up the sorting process. Here we check if the machine correctly uses the mode setting.

DONE state reachability

Query: E<> ProcessReinhard.DONE
Description: There exists a path where the system reaches the ABORT state
This check ensures that the machine can eventually finish the sorting process.

Machine pause justify

Query: A[] ((ProcessReinhard.paused and mode != 2) imply machinePaused==1)
Description: Whenever the system is in the paused state and not in incremental mode, machinePaused must be set to 1.
Verifies that in the full speed and safe modes, the machine only reaches the paused state when the pause button was pressed beforehand.

Valid mode specified

Query: A[] mode==0 or mode==1 or mode==2
Description: Mode must be one of: full speed, safe or incremental.
The mode value must be valid.

No deadlock

Query: A[] not deadlock
Description: The system never deadlocks in any case

4.7   Conclusion

this chapter provided the information about the behaviour of our software. We introduced our state machine diagram which we then implemented in UPPAAL. We clearly explained the working of our systems in UPPAAL and verified our model using this program. This part is used as the basis for the design of the software and is very important.

Chapter 5

Software design

5.1 Introduction

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

5.2 Object Oriented Design

We have chosen to implement the states of our UPPAAL model using the object oriented programming paradigm. The state implementations follow directly from the finite state machine described in the software specification. The class diagram of figure 5.1 provides an overview of all states and auxiliary classes. In the diagram the '+' symbol indicates a publicly accessible method or property, '-' indicates a private accessible method or property, bold indicates a constant property or a method which cannot be overridden and underlined indicates a property or method which is accessible through the class.
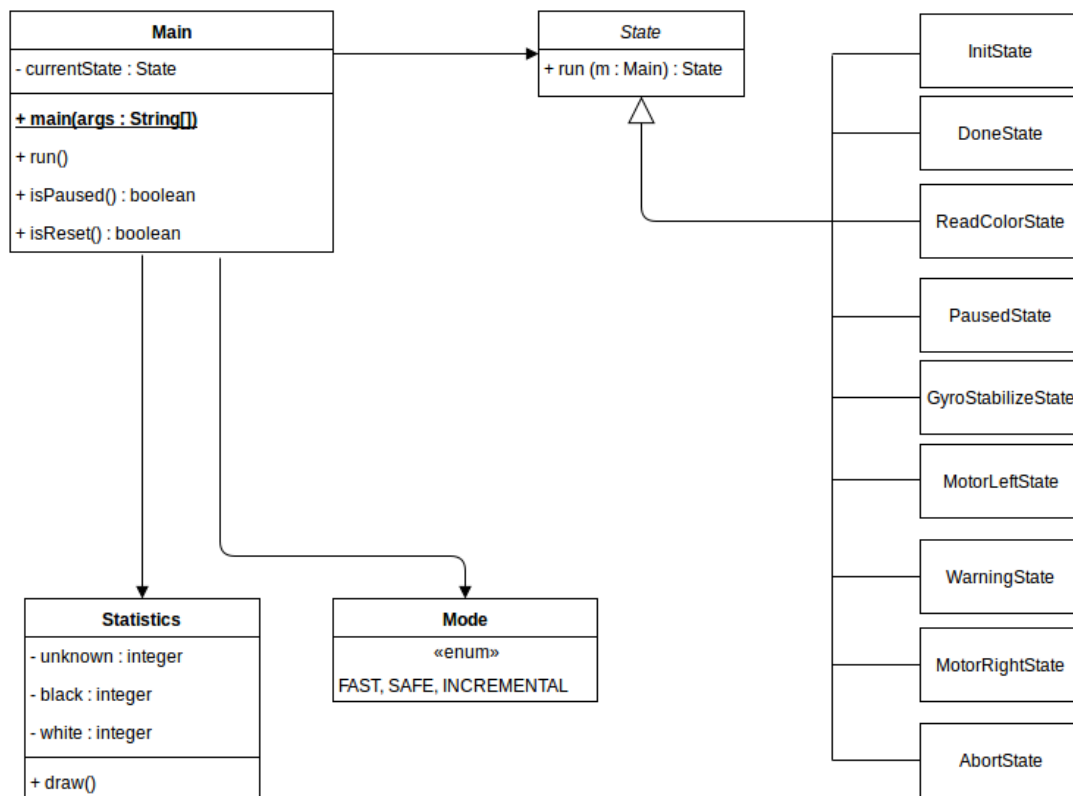
Figure 5.1: The class diagram

5.2.1 Main class

The main class handles button presses, updates the display and executes the 'run' method on the current state. It also exposes properties which provide access to the actuators and sensors. These are objects from the LeJOS API. The properties providing this access are motor, gyro, color, aButton (the abort button), spButton (the start/pause button) and rButton (the reset button). These properties are declared public to provide easy access, and these are declared final to make sure the properties cant be changed.

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

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

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

Run method

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

```
 1: while true do
 2:     if Abort button pressed and current state is not abort state then
 3:         Current state = new AbortState()
 4:     else if Start/pause button is pressed then
 5:         paused = true
 6:     else if reset equals true then
 7:         reset = true
 8:     end if
 9:     current state = current state.run(this)
10: end while
```

5.2.2 Statistics class

The statistics class is meant to store three counters for unknown, black and white discs. Although it is not shown in the class diagram of 5.1, it does provide methods to modify these counters.

5.2.3 Display class

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

5.2.4 Mode enumerable

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

5.2.5 Abstract class state

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

5.2.6 State implementation classes

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

WarningState class

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

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

AbortState class

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

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

5.3   Arguing correctness

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

1. The states are implemented as in the state machine

2. The button flags are set as required

3. The abort button triggers the system to go to the abort state if it is not yet in the abort state

5.3.1   State implementation

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

5.3.2   Button flags

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

5.3.3   Abort button

The loop first checks if the abort button is pressed and it checks if the system is not in the abort state. Then, if this is the case, then the current state pointer is set to a new abort state object and the run method of that abort state object is run and therefore the system is in the

abort state. Again we have the constraint that the run method should run in less time than an average human presses a button. However, this isnt a problem as said before.

5.4   Conclusion

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

Chapter 6

Software implementation

6.1 Introduction

Having designed our software it is time to actually implement it. This document will describe the implementation of the design described in the document Software Design. It will however, not contain the actual code itself, nor any information of how specific methods and classes are implemented, but only describe the general conventions of the designed software and a user manual of our machine in relation to the software. It will also briefly mention how we have implemented timers in our program.

6.2 Naming/Java conventions

Since we implement our software with Java as our programming language it is important to use the standard Java conventions for our code. We will therefore explain the most important conventions we stick to:

6.2.1 Proper indentation and brackets.

To keep the code clean and readable it is important to use proper indentation when nesting code.

Example:
```
if (...) {
    while (...) {
        indentation
    }
    indentation
}
```

As is visible we also use the standard conventions for the use of curly brackets, making loops and methods structurized.

6.2.2 Naming conventions

To clearly distinct variables, constants, methods and classes we use the Java conventions for naming them. Constants are written in uppercase letters with an underscore between the words.

Example:
public static final int $MOTOR\_TURN\_SPEED = 120;$

Variables are written according to the lowerCamelCase convention, where the first word has a first lowercase letter and the other words are written with first upper case letter.

Example:
```
public final GyroSensor gyroSensor;
```

Methods are also written in lowerCamelCase convention.

Example:
```
private void run() {
    indentation;
}
```

Classes are written according to the UpperCamelCase convention.

Example:
```
public class MotorLeftState extends MotorState {
    indentation;
}
```

6.3   User Manual

6.3.1   Machine preparation

Before using the machine and running any programs on it, the machine should be sufficiently prepared to ensure proper operation. The machine should be positioned on a level surface, in a not too disruptive lit room (e.g. no very powerful lights pointing directly or indirectly at the machine).

6.3.2   Starting the machine and program

After the machine has been prepared properly as described in the previous section, the machine can be booted up. This is done by holding down the center button on the main brick until the display shows the booting screen of LeJos. After the machine has been booted, the display will show a main menu. To start the sorting program, the user should navigate to the Programs item by using the left and right button on brick and then pressing the center button. Then the user should navigate to the Main program and press the center button to start it. The machine will now start the actual program.

6.3.3   Selecting a mode

After the program has been started, the user will be prompted with a mode selection screen. The user can choose a mode with the left, center and right buttons. The modes are explained in the machine design document. After a mode has been chosen, sensors and motors will be calibrated, so parts will start to move during this process.

6.3.4   Filling the input tube

After program initialization is done, the user is allowed to open the input tube by rotating it horizontally and pulling the yellow part on the top side of the tube. The user may need to keep

the tube in place with his other hand to prevent it from moving back vertically while opening it the tube. The grey bar attached to the yellow pull can now be rotated vertically to move it out of the way. Ensure that the black/grey pin is inserted in the bottom part of the tube. The tube can now be filled with black and white discs. After the tube has been filled, the tube must be closed by moving the yellow pull back to the top of the tube and attaching the black connectors to the side of the tube. Then the tube must be rotated back vertically to its position above the moving arm. The black/grey pin on the bottom of the tube can be removed. The pin can optionally be inserted in the small blue bar on the left side of the machine to store it.

6.3.5   Start the sorting operation

The center button must be pressed to start the sorting operation

6.3.6   Pausing and resuming the sorting operation

While the machine is sorting, the sorting operation can be paused by pressing the center button.While the machine is paused, the sorting operation can be resumed its sorting operation by pressing the center button again.

6.3.7   Stopping the sorting operation

While the machine is sorting, the sorting operation can be stopped by pressing the down button. The operation will be stopped completely and the user can start the operation again if desired.

6.3.8   Shutting down the machine

In order to shut down the machine, one must go back the mode selection menu, and afterwards press reset.

6.3.9   Troubleshooting

The machine supports detection of different kinds of errors that may occur during the sorting operation. When such an error occurs, a light will start to flash red or orange and the machine will show a message on the display describing the error that occurred. When the error is fatal (The led blinks red) has occurred, the user can return to the mode selection screen by pressing the down button. The user is expected to have resolved the problem before returning to the mode selection screen.

6.4   Timer

Some warnings depend on time. The errors and warnings like gyro does not stabilize, disc did not reach basket and disc took longer than average to reach basket depend on time. In code this is implemented by a Timer class which gives the time between two method calls. However we need to set the thresholds in order to be able to decide whether or not an error or warning should occur. To do this we made a simple app, TimerApp which sorts the discs and measures the time between the moment that the motor started to turn and the time the disc fell onto the lever. The simple app then calculates the average, minimal time and maximum time.

6.5   Conclusion

this chapter has explained the naming conventions important for our software implementation phase. Since the Java conventions are known we have used and explained these in this file.

We have also discussed the use of timers in our software implementation since these were not mentioned before. Lastly we have included a user manual that describes the user interaction in relation to the software. The next document will describe how we test our implementation and validate correctness.

Chapter 7

System validation and testing

7.1   Introduction

During the different project phases we had to ensure that every decision and design we made would result in a correctly working machine. This document describes how we verified that our designs and implementations function correctly.

7.2   Test cases

7.2.1   Manual tests

During the software specification and software implementation phases we defined and implemented tests which we can run manually to verify correct functioning of sensors and motor. We decided to exclude these tests from the unit testing suite, which will be described next, because these tests are also dependant on the hardware instead of only on software.

Colour sensor test   We created a test program which continuously reads the output of the colour sensor and shows those values on the screen. The LEGO Mindstorms colour sensor has various modes that detect different kinds of light. The test switches between the RGB, grayscale and ambient modes and then reports the values measured. This enabled us to determine what mode was best suited to distinguish different colours, as well as provide some initial settings for unknown lighting conditions.

Gyroscope test   The second test program tests the functionality of our gyroscope sensor, which we use to verify arrival of the discs in the sorting trays. The test measures the current rate of angle change of the sensor and reports the values to the LCD display. Next it uses the algorithm we implemented to report either the values LEFT, RIGHT or NEUTRAL based on the current angle and report that value to the display as well. We can use the test by starting the test program on the brick, making sure that the gyroscope is stable and level during initialisation. When initialisation is finished, the gyroscope can be moved in both directions and the values can be read from the display during the process. This way we can verify manually whether the reported values are correct.

7.2.2   Unit tests

Unit tests are small programs that test a small part of the main program, usually a single function or class. The input of the to be tested part of the code is simulated, or mocked, such that we can easily create conditions under which the tested code has to perform. These conditions are sometimes difficult to produce outside the tests. After the input and conditions

are created, the function or class is executed with these values and the output is compared with the expected output.

We chose to use JUnit as our unit testing framework. Unit tests will execute automatically when code is being changed in our IDE, eclipse. The unit testing framework will evaluate the code changes and run the tests for which the tested code is possibly affected by the changes. This ensures that we spot bugs and errors early on in the implementation, granted there are enough tests to cover the codebase.

Mocking

Several mock peripherals are needed to simulate inputs and outputs during unit tests:

Battery mock: We use this to simulate the voltage given by the battery. This way we can test whether the software handles a low battery exception properly.

MockGyroSensor: Used to test the usage of the gyroscope. We can simulate left or right turns to check whether code handles these situations correctly.

MockButton: Simulates button presses on the EV3 brick. Used to verify menu behaviour and user interaction.

MockColourSensor: Used to simulate detected colour values. This is useful when we want to verify behaviour regarding colour detection.

MockDisplay: Used to print values back to the console, instead of displaying to the screen on the EV3 brick.

MockMotor: Used to simulate functioning of the motor.

MockTouchSensor: Used to simulate input from the touch sensor.

Testing the sorting process

We implemented the following unit tests. The below unit test ensure that the machine can perform the sorting process correctly.

TestMotorCalibration: Start spinning the motor. It should keep spinning until the touch-sensor is being pressed. Then the motor should stop rotating.

TestcolourSensor: Simulate a colour input, let the statemachine react to the input and check whether we entered the correct state.

TestSortingProcess: Simulate the machine in the safe mode and test if it handles colour readings and gyroscope readings correctly.

TestReset: Test whether the machine correctly returns to the mode selection state whenever the reset button is pressed.

Testing the errors

The following unit tests ensure that all errors are handled correctly.

TestMotorJammed: Simulate a motor and test whether the program correctly gives an error when the motor is stalled.

TestWrongBasket: Simulate a gyroscope and let the current state be a motor state for a specific direction. Test if the program gives an error if the gyroscope detects that the disc fell the wrong way.

TestNoBasket: Let the current state be an arbitrary motor state. Let the test wait for the maximum waiting time and test if the machine gives an error if no disc has been detected yet.

TestWrongInput: Simulate a colour sensor and let the current state be the colour reading state. Test whether the machine gives an error when an unknown colour is detected.

TestBatteryLow: Test whether the machine gives an error in any state and in any mode when the battery level is lower than the permitted amount.

TestDeviatesFromAverage: Let the current state be an arbitrary motor state. Let the test wait for the average waiting time Test whether the machine gives a warning if no disc has been detected yet.

TestAbort: Test whether the machine gives an error in any state and in any mode whenever the abort button is pressed.

TestGyroDoesNotStabilize: Let the current state be the stabilization state. Let the test wait for the maximum stabilization waiting time. Test whether the machine gives an error if the gyroscope has not stabilized yet.

7.2.3 Test coverage

To ensure that all of our code works correctly and as intended, we have to cover our code with tests as much as possible. This means that, we have to try and get as many lines of our code tested. The ideal situation will be to strive for a 100% code coverage, but this has some inconveniences. A lot of the code is very trivial, for example, getter and setter methods that simply return or set a value of a variable. We dont need to test these parts of the code, as the code does not do anything particularly interesting and we can assume that the java runtime works correctly. We dont have to check whether setting a variable to value x, actually results in the variable being equal to x. Testing this would be pointless and would add additional testing code which adds more stuff to maintain. Therefore we aim to only test interesting interactions in the code.

7.3 Formal proofs

Now that we have shown through numerous test cases that the machine behaves as described in the specification, we will formally prove that the specification itself is correct. We do this by using the verification capabilities in UPPAAL on our UPPAAL model.

7.3.1 Properties

UPPAAL proves properties through exhaustive state space exploration. A property is tested to hold in every state in the model. The following properties will verify the correctness of our model:

- The system never deadlocks
  A[] not deadlock
  This property ensures that the machine is always operable. This is important as we cannot expect the user to be able to fix a broken machine.

- There is always a valid mode specified
  A[] mode == 0 or mode == 1 or mode == 2
  This property shows that the mode is always one of fast, safe or incremental. The machine is unable to handle any other value of mode, so this property ensures that mode is always valid.

- The sorting machine can finish sorting
  E<> ProcessReinhard.DONE
  This property proves that the machine is capable of finishing the sorting process. It is necessary that the machine can perform its task correctly.

- The sorting machine can encounter and handle fatal errors
  E<> ProcessReinhard.ABORT
  This property shows that the machine can halt in case of fatal errors by going to the ABORT state.

- It is possible that the system does not reach the ABORT state
  E[] not ProcessReinhard.ABORT
  This property proves that the machine can run without encountering a fatal error.

- In the initial and paused states all peripherals are in their idle state
  A[] (ProcessReinhard.paused or ProcessReinhard.peripheral_reset) imply (ProcessMotor.INIT and ProcessGyroSensor.INIT)
  This property ensures that all peripherals are ready for use when their functions are required. This is especially important when the machine goes through the ABORT state, which can abruptly interrupt the normal functionality of the sensors. It is of the essence that the machine can continue to function correctly when this happens.

- The gyroscope is never checked in fast mode
  A[] mode == 0 imply ProcessGyroSensor.INIT
  This property shows that the gyroscope is not used in the fast mode, as it is not required in that mode.

- In safe and incremental mode, the gyroscope is always used
  E<> (mode != 0) imply (not ProcessGyroSensor.INIT)
  This property shows that the gyroscope is always used in safe and incremental mode.

- When not in incremental mode, machinePaused is always 1 in the paused state
  A[] ((ProcessReinhard.paused and mode != 2) imply machinePaused==1)
  This property shows that when the machine runs in either fast or safe mode, the machine is only paused when the associated button is pressed.

All of these properties are proven to be satisfied by UPPAALs model verifier. Hence, we can confirm that the UPPAAL model is indeed correct. Since our machine performs its task precisely as described in the UPPAAL model, we can conclude that the machine adheres to the System Level Recuirements.

7.4   Conclusion

As shown in this chapter, we have tested our designs throughout the different phases of the project. We used several techniques to prove or show correctness of designs and implementations depending on the situation. We used manual tests for testing the hardware interface and hardware sensors, while we use automated unit tests for the software. While designing the software we verified the behaviour of our state machine in UPPAAL and proved the behaviour formally wherever we could. This resulted in a software implementation that has been thoroughly tested and shown to be correct.

Chapter 8

Conclusion

After extensive proofs in Uppaal, logical reasoning and unit tests, we could go on to say that our machine is extremely reliable. It passed all these tests with flying colours, yet we do not consider it to be dependable. The truth is that there are many ways in which the sorting process can be disrupted. Disconnecting one or more of the peripherals from the brick for instance is considered undefined behaviour, as there exists no straightforward way of detecting this. However, many of the possible errors due to the environment can be detected at run time and will either be handled automatically or are reported to the user, and the machine is immediately halted in case of a fatal or potentially dangerous error. While it is impossible to build a totally reliable machine, we have been able to create a fast and safe solution to the initial problem. The use of a Mindstorms kit instead of a PP2 with fischertechnik has been both a blessing and a curse: It was relatively simple to write the software, as we could use a high-level language. This did make us very dependent on the platform we used, which is still in beta and did not always suit our purposes. We had sensors that could be queried easily and were well-integrated with the platform, but were not designed for our small, light stones and only a single sensor of each type was provided, forcing us to design a rather clever contraption. All in all, this project has been a great adventure, resulting in a fast, safe and verbose sorting machine: One that, combined with better equipment, might even become incredibly reliable.