System Validation & Testing

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

Responsible:
Sergio van Amerongen
s.w.j.v.amerongen@student.tue.nl

March 25, 2016

1 Introduction

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

2 Test cases

2.1 Manual tests

During the software specification and software implementation phases we defined and implemented some 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.

Color sensor test We created a test program which reads out the sensor values of the color sensor continuously and shows those values on the screen. The LEGO Mindstorms color 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. The test was created before we defined what values we used exactly to distinguish black from white discs. This way we base our decisions on the results of initial testing and we are able to verify that those values are still correct when the machine is put in a different lighting environment.

Gyroscop 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 angle 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 initialization. When initialization is finished, the gyroscope can be moved in both directions and the values can be read from the display during the proces. This way we can verify manually whether the reported values are correct.

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.

2.2.1 Mock

We have got several unit tests, for which we use several mocks to simulate input.

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.

MockColorSensor: Used to simulate detected color values. This is useful when we want to verify behaviour regarding color 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.

2.2.2 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.

TestColorSensor: Simulate a color 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 color readings and gyroscope readings correctly.

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

2.2.3 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 color sensor and let the current state be the color reading state. Test whether the machine gives an error when an unknown color 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.

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.

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.

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.

4 Conclusion

As shown in this document, 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.