FH Vorarlberg
University of Applied Sciences

Transferring the Python framework to Java for the robot "e-Puck" and V-REP as simulator

Bachelor Thesis
for the degree of

Bachelor of Science in Engineering (BSc)

Vorarlberg University of Applied Sciences
Computer Science – Software and Information Engineering

Advisor
Prof. (FH) Dr. Hans-Joachim Vollbrecht

Submitted by
Daniel Thomas Groß
Dornbirn, October 2018

Kurzreferat

Die Fachhochschule Vorarlberg verwendet derzeit zwei verschiedene Simulatoren, um Inhalte zu mobilen Robotern zu lehren. Die verwendeten Simulatoren sind V-REP und Webots. Im Masterstudium der Informatik wird der Simulator V-REP verwendet. Die Programmierung der Steuerung erfolgt mit der Programmiersprache Python. Die Fachhochschule Vorarlberg hat dafür ein Programmiergerüst in Python entwickelt. Im Bakkalaureats Studium der Informatik wird der Simulator Webots verwendet. Die Programmierung der Steuerung erfolgt mit Java. Diese Arbeit befasst sich mit der Implementierung des bestehenden Python Programmiergerüst in Java, um den Simulator V-REP auch im Bakkalaureats Studium verwenden zu können. Das bestehende Python Programmiergerüst wird analysiert und die Programmiersprachen Python und Java miteinander verglichen, um Python zu verstehen und die Funktionalität des Programmiergerüsts richtig in Java zu übernehmen. Anhand der Analyse des Python Programmiergerüst wird das Java Programmiergerüst implementiert und dann auf die Performanz getestet. Der Test der Performanz soll zeigen ob das Framework schnell genug ist um den Roboter ohne Nachteile zu steuern. Um die Verwendbarkeit des Frameworks zu testen, wird ein kleines Beispiel Programm präsentiert.

Abstract

The Vorarlberg University of Applied Sciences currently uses two different simulators to teach the students about mobile robots. During the Masters in Computer Science program, the simulator V-REP is used. The robot controllers are programmed using the programming language Python. Therefore, the Vorarlberg University of Applied Sciences developed a framework using Python. In the Bachelor of Computer Science program, the simulator Webots is used and the controllers are programmed using Java. The topic of this thesis is to translate the existing Python framework to Java, to be able to use the simulator V-REP during the Bachelor studies. The existing Python framework is analysed, and the programming languages Python and Java are compared to understand Python and be able to translate the framework correctly to Java. Based on the analysis of the Python framework, the Java framework gets implemented and is then tested regarding its performance. The performance test shall show if the framework is fast enough to control the robot without drawbacks. To test the usability of the framework an example application is created.

Contents

List of Figures

1. Introduction

With the help of simulators, robotic software development is greatly simplified. The development and testing of a software and directly testing it is made possible everywhere, and cost can be much lower. A physical robot can cost thousands of Euros. There are some simulators that can be as expensive as a physical robot, but there are also simulators that are available for free. Simulators have their own programming language to control a robot, and often they also provide an application programming interface (API). With an API it is possible to communicate with the simulator and also to control a robot inside the simulator. Such an API can be very complex to use and therefore frameworks are developed. A framework is not only used to simplify the control of a robot, but also to define the robot's functionalities.

1.1. Problem Statement

The Vorarlberg University of Applied Sciences currently uses two different simulators. During the Bachelor studies *Computer Science - Information and Software Engineering* the simulator Webots is used in the 5th semester [54]. For the elective course *Autonomous Systems*, of the Msters studies of Computer Science, the simulator V-REP is used [50]. The simulator V-REP is available to Universities and students for free as the version V-REP PRO EDU [49]. Webots, however, has high licence costs [49]. Therefore, this thesis is to reduce the costs. To be able to use the simulator V-REP during the Bachelor studies, it must be possible to control the virtual model of the robot e-Puck in Java as the students learn Java and not Python. V-REP offers a remote API for Java to communicate with the simulator, but there is currently no framework for Java to control the virtual robot e-Puck. Therefore, a framework is needed which makes it possible to program a controller in Java.

1.2. Aim of the Work

The aim of this work is to create a framework in Java to control the robot e-Puck. e-Puck is a robot for educational purpose [4]. It has two wheels and some sensors. The sensors on the robot are sound sensors, accelerometer, proximity sensors, a camera and ground sensors. In the major *Autonomous Systems* of the Computer Science Masters program the robot simulator V-REP and the real robot e-Puck are used. Therefore, a framework in Python was developed by the Vorarlberg University of Applied Sciences. As shown in Figure 1.1 the Python framework allows for the control of the virtual

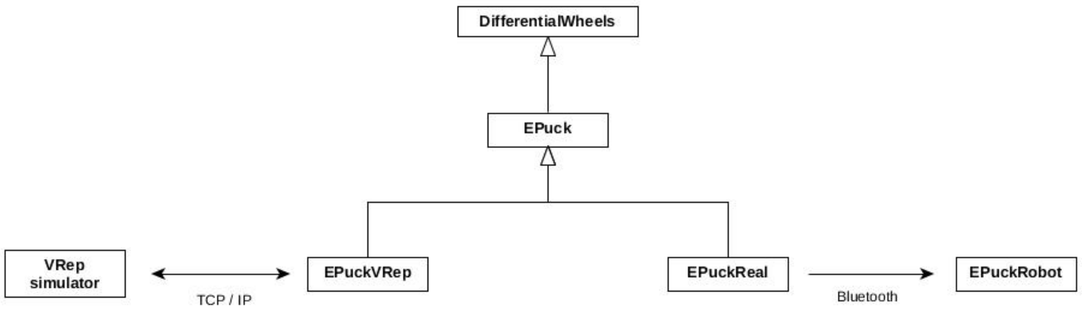


Figure 1.1.: Picture shows the design of the python framework

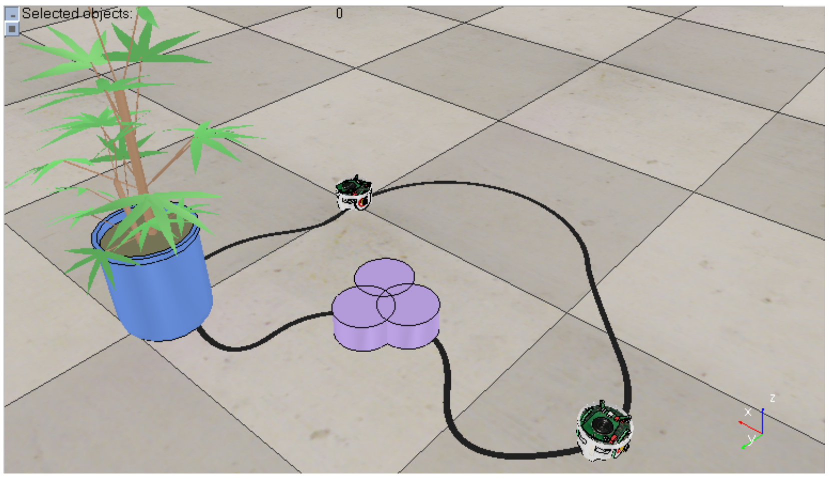


Figure 1.2.: Environment for the example application V-REP. Two robots following a line while driving towards each other. They also must avoid the obstacles on the path.

version of the robot and the real robot. The real robot is controlled by using a Bluetooth connection and the connection to V-REP based on TCP/IP. The new framework shall have the same functionality as the already existing framework in Python, but for now we only want to control a robot in the simulator V-REP. As the framework shall be extended with functionality to control the physical robot in the future, the architecture needs to be easy to extend. The aim is to replace the simulator Webots with the simulator V-REP. The simulator Webots is currently used during the 5th semester. Therefore, the new framework needs to be easy to understand and a good documentation needs to be provided for the students. Additionally, to the documentation a example application shall be provided for the 5th semester students. The example program shall show the basic principals of the framework to make it easier for the students to write their first program. Figure 1.2 shows how the environment for the example application looks like. The two robots are following a line while they are driving towards each other. When the robots meet they should pass each other and get back to the line and continue following it. In the path there are also some obstacles which the robots must avoid.

1.3. Methodological Approach

Without any prior knowledge of the Python framework for the robot e-Puck, a class diagram of the framework was created first. To analyse and understand the framework, it was necessary to learn the fundamentals of Python. Based on the class diagram of the Python framework, the Java framework was then designed with a class diagram. This class diagram was used as reference for implementation. Changes in the design during implementation, were immediately updated in the class diagram. Before proceeding with the example application, the performance and functionality of the framework was tested. The documentation was created by using Javadoc and adding a description of every methods of the framework.

2. State of the art

2.1. Controlling a robot

A software, which controls a robot, needs to process various information. The main task of a robotic software is to plan its future path. To plan this path some information is needed. One possible way of getting such information is through a sensor. Therefore, the software needs to be able to read data from a sensor. By interpreting this data, it is possible to decide what to do next. The next step, after interpreting the data, could be to set the speed of the wheels, if the robot is a wheeled robot. Another action, based on the results interpreted from the data, could be to move a camera in a certain direction or any other moveable part at the robot [41].

Mobile Robotics Architectures defines ways on how robots are controlled and therefore how data is requested, processed and applied. Section 2.1.1 discusses the requirements for Mobile Robotics Architectures and explains some of the common architectures. Following this, Section 2.1.2 explains where this framework is to be used in these architectures.

2.1.1. Architectures to control a robot

For an architecture, to control a robot, four different requirements can be defined:

1. Deliberative and reactive behaviour
   Deliberate and reactive implies that specific actions are taken to achieve a goal while also reacting to the environment. For example, a robot needs to follow a line but also avoid obstacles [41].

2. Allow uncertainty
   Allowing uncertainty means allowing the robot to be able to function at any time. The software must be able to handle information that might be incomplete, unreliable or contradictory at any time [41].

3. Account for danger
   The architecture must be able to handle certain circumstances like a door which is expected to be closed being open. It should be able to react and avoid a crash with the door. Such properties of an architecture relate to fault tolerance, performance and safety. [41].

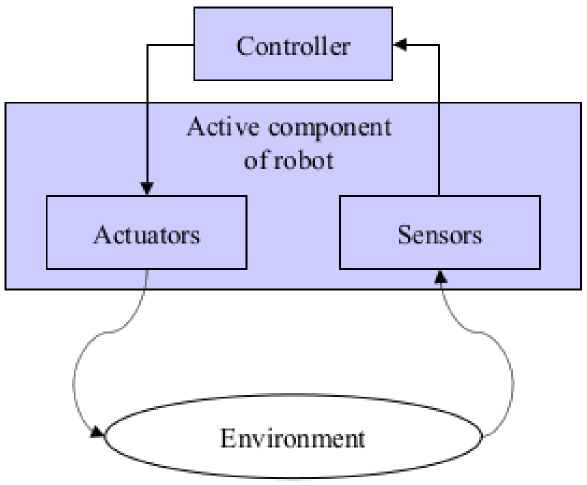


Figure 2.1.: Example of a loop control architecture [41].

4. Flexibility for the designer
   The architecture allows the software developer to replace main components like sensors or motors [41].

Based on these four requirements common architectures are: the *Control Loop Architecture*, *Layered Architecture*, *Task Control Architecture*, *Subsumption Architecture* and *Blackboard Architecture*.

Control Loop Architecture
The control loop architecture is defined by its continuous loop where a sensor value is read, values for actuators are being calculated and values are set. Because components interact with each other by providing data and because components are being executed when data is available, it may be described as a data flow architecture. Figure 2.1 shows a very simple example of a loop control architecture. The controller gets a new sensor value (for example values of a distance sensor), checks if there are any obstacles and then sets the motor speed [41, 25].

Layered Architecture
The idea of the layered architecture is to split up the control of the robot. Splitting up means to make abstract layers that connect to each other. Each of the layers has a specific area of responsibility and after a layer has processed the data, the results get passed on to the next layer. Instead of passing the results to the next layer, it is also possible to notify the next layer that there is new data available. For example, the responsibility of the layer *sensor interpretation* is to analyse the data from one sensor. The layer *sensor integration* performs a combined analysis of different sensor inputs. Different to business applications, where this happens from top-down, for robotics this happens bottom-up. The start therefore is for example a sensor that returns a new value [41, 26]. From top to bottom of the architecture, the layers are as following:

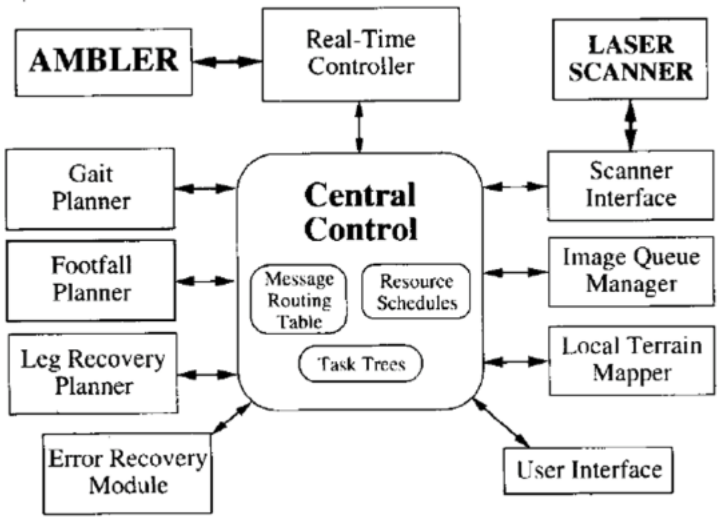


Figure 2.2.: Example of a central module with task modules [23].

1. Supervisor
2. Global planning
3. Control
4. Navigation
5. Real-world modeling
6. Sensor integration
7. Sensor interpretation
8. Robot control

Task Control Architecture
The task control architecture is a high-level operating system. It consists of a central module which is general and task-specific modules, shown in Figure 2.2. The central module is responsible for routing messages and maintaining task control information. The robot-dependent information is processed by the task-specific modules. Task trees, like the one shown in Figure 2.3, build the base for the task control architecture. Child tasks get initiated by its parent tasks. Between tasks temporal constraints can be defined. The communication between tasks and task-modules is done by sending messages to a central server. The central server redirects the message to tasks which have registered at the central to handle them [41, 27, 29, 23].

Subsumption Architecture
The subsumption architecture is a robot control system which uses a layering methodology. Each level describes a behaviour which is autonomous in itself. The

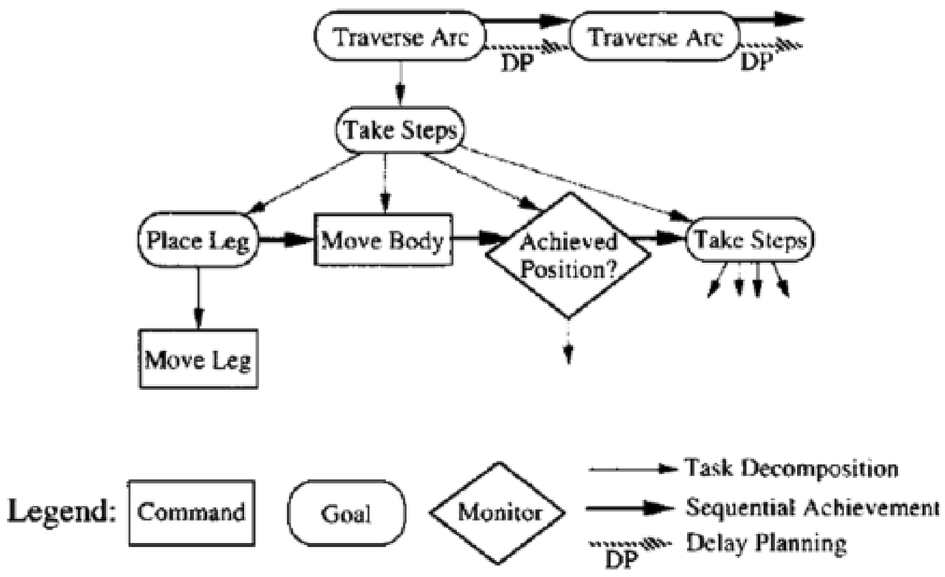


Figure 2.3.: Example of a task tree for autonomous walking [23].

activation of a level is done by an activation condition which is a Boolean. Each behaviour has an action as a result. The behaviour action of a higher level suppresses all the actions of the lower levels, which is indicated with the arrows pointing downwards in Figure 2.4. Figure 2.4 shows different levels of a subsumption architecture. For example, let the behaviour of level 0 be finding a box, pushing a box the behaviour of level 1 and detach from box the behaviour of level 2. If the robot is currently pushing the box and the activation condition for level 2 is fulfilled, the level 2 behaviour action will suppress the behaviour action of level 0 and level 1 [8].

Blackboard Architecture
The Blackboard Architecture approach uses a central unit that stores all information. Modules let the central unit, the blackboard, know which information they are interested in. If the requested data is available, it will be returned immediately. If the information of interest is not available, it will be returned when another module inserts this data [41, 28]. Figure 2.5 shows an example of the Blackboard Architecture.

2.1.2. Why a framework?

As all the architectures from Section 2.1.1 showed, there is a point in the architecture where a communication with the actuators and the sensors takes place. Without a framework the software engineer directly addresses the required actuator or sensor of the robot or the API of the simulator. The sequence diagram sketch, shown in Figure 2.6, shows a controller with the control loop architecture that calls the remote API of V-REP. The sequence diagram shows that the software engineer must look up how to use the API of V-REP. For all needed actuators and sensors, methods need to be implemented to write and read values. If the software engineer only uses a simulator

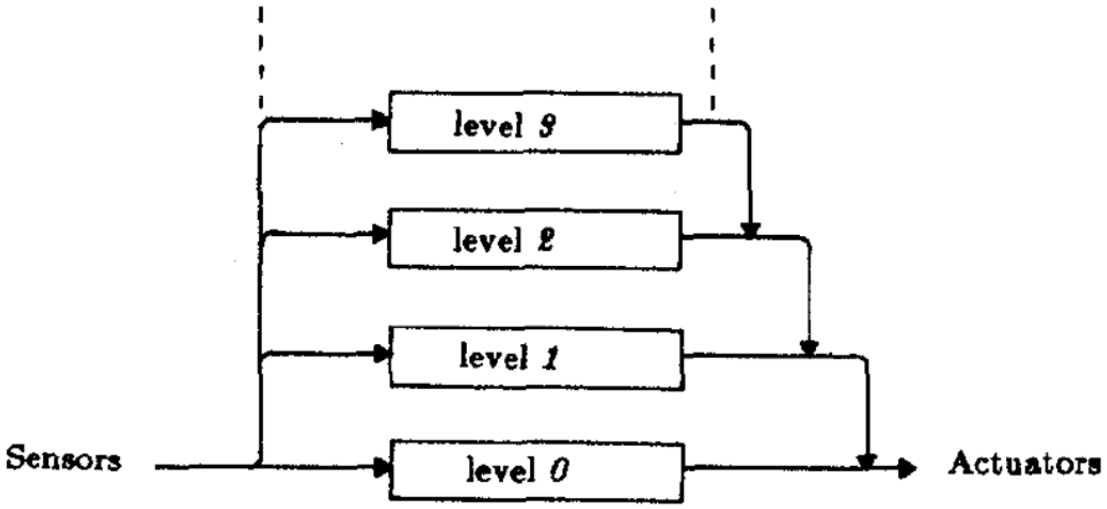


Figure 2.4.: Example of the layering of the subsumption architecture [8].

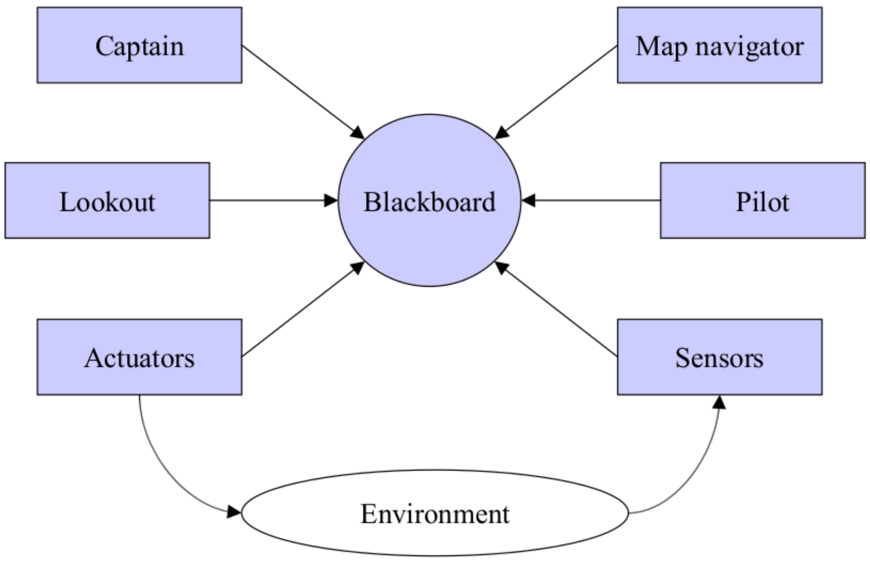


Figure 2.5.: Example of the blackboard architecture [10].

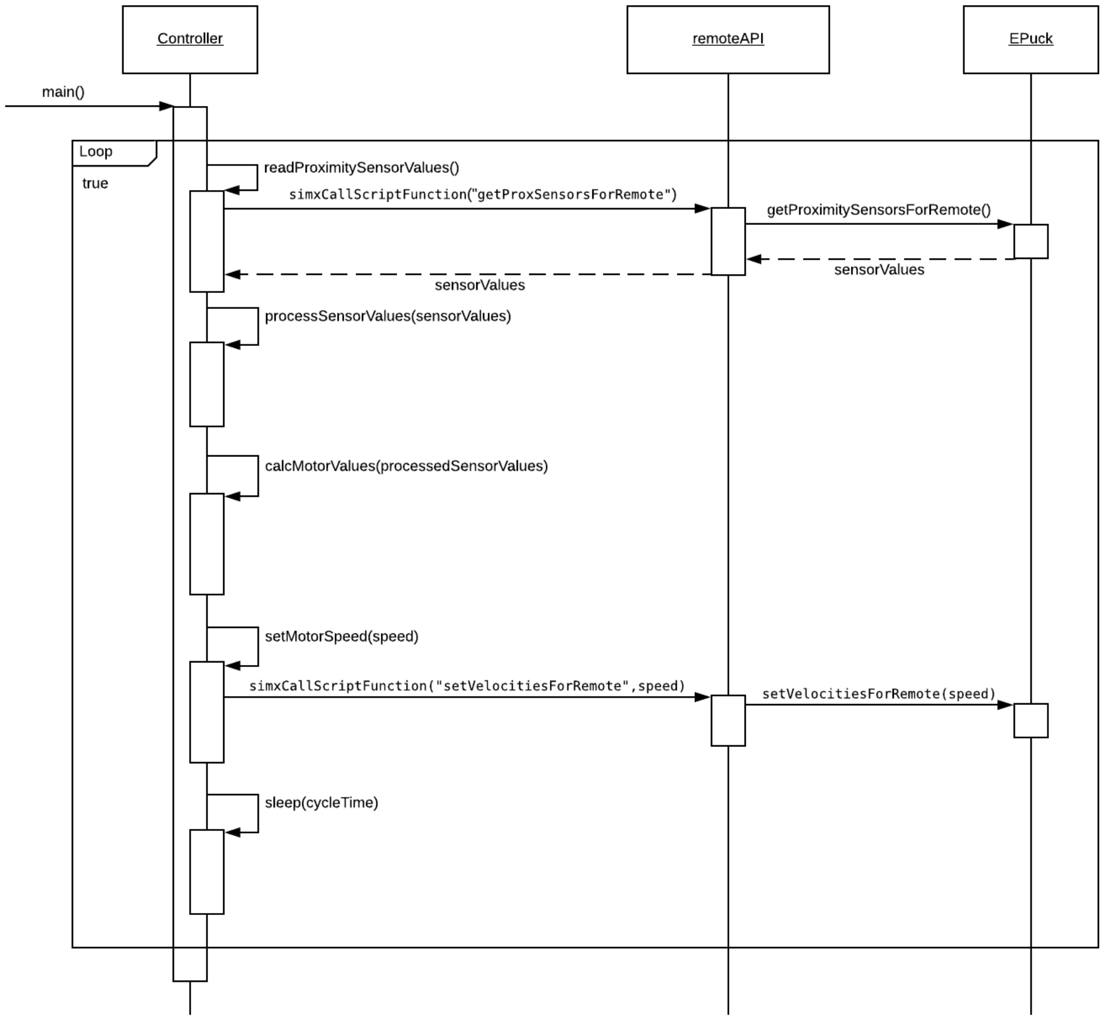

Controller
remoteAPI
EPuck
main()
Loop
true
readProximitySensorValues()
simxCallScriptFunction("getProxSensorsForRemote")
getProximitySensorsForRemote()
sensorValues
sensorValues
processSensorValues(sensorValues)
calcMotorValues(processedSensorValues)
setMotorSpeed(speed)
simxCallScriptFunction("setVelocitiesForRemote",speed)
setVelocitiesForRemote(speed)
sleep(cycleTime)

Figure 2.6.: Sketch of a sequence diagram of a controller calling remote API methods without a framework.

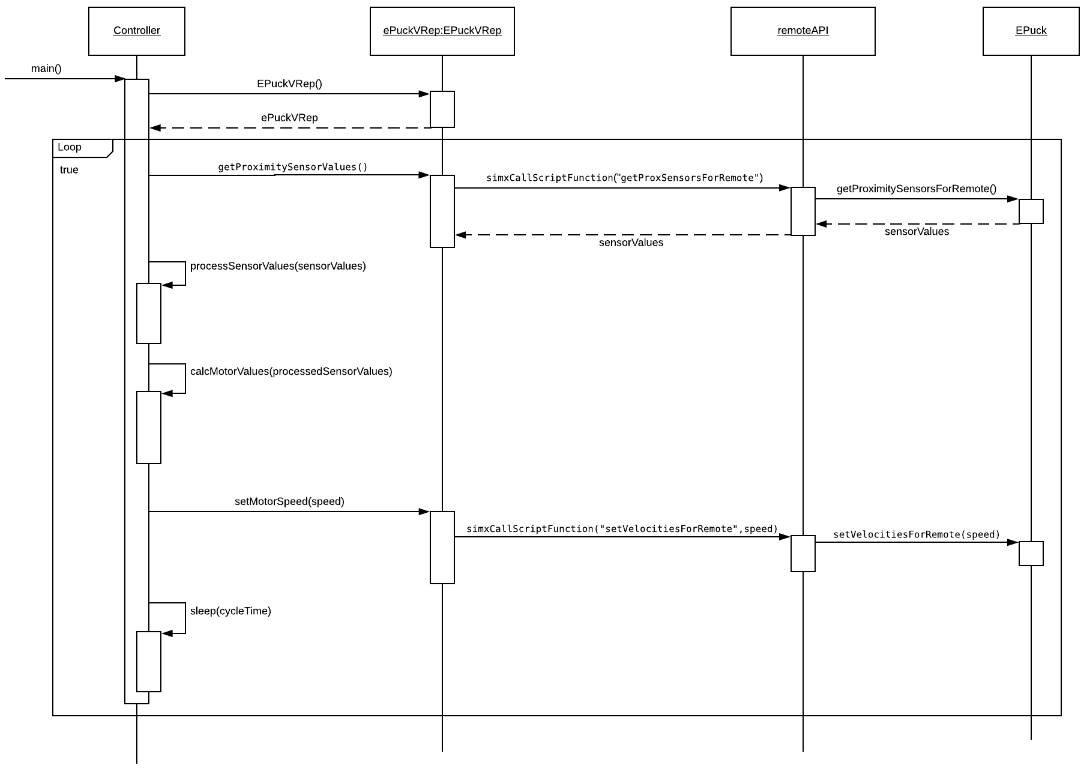

Controller
ePuckVRep:EPuckVRep
remoteAPI
EPuck
main()
EPuckVRep()
ePuckVRep
Loop
true
getProximitySensorValues()
simxCallScriptFunction("getProxSensorsForRemote")
getProximitySensorsForRemote()
sensorValues
sensorValues
processSensorValues(sensorValues)
calcMotorValues(processedSensorValues)
setMotorSpeed(speed)
simxCallScriptFunction("setVelocitiesForRemote",speed)
setVelocitiesForRemote(speed)
sleep(cycleTime)

Figure 2.7.: Sequence diagram of a controller calling remote API methods using the framework.

like V-REP, to run and test his program, this can be an acceptable solution. If the software engineer wants to test the controller on the real robot, the current code needs to be changed. Changing the code can be very time consuming. The software engineer needs to find out how to communicate with the real robot. New methods need to be implemented to get sensor values and set the values of actuators. This is where the use of a framework makes sense. Figure 2.7 shows a sequence diagram sketch of the controller from Figure 2.6 using the framework. The framework represents the robot and implements methods to communicate with actuators and sensors. Methods called by the software engineer are only those of the framework object *ePuckVRep*, every call to the right of the class *EPuckVRep* happens inside the framework. thus, the framework removes the problem of needing to cope with the real robot's API or the API of a simulator when programming the controller. The University's Python framework implements classes to control the real robot and the robot in the simulator V-REP. It also has an abstract representation of the robot e-Puck to make switching between the real and the virtual robot easy. The abstraction and implementation of the Python framework is discussed in detail in Section 2.4.

2.1.3. Real robot vs simulated robot

Controlling a virtual robot can be different from controlling a real robot. There will be a big difference in the values received from the real robot compared to the simulator. Sensor values inside a simulator are calculated values. Taking the proximity sensor as an example, the simulator exactly knows the distance between the robot and an obstacle and can calculate the exact proximity value. To make the value more realistic the simulator adds noise to the value. Even with this added noise the values will never be like the ones received from a real robot. This means a controller, developed and tested using a simulator, will not work the same on the real robot. Due to the real sensor values, which have more noise than the ones of the simulator, the controller will need some adjustment to make it work again. Slight differences of the traction of the wheels between the real robot and the simulated robot can make a difference when controlling it. A model of a robot in the simulator will get close to the real robot but will not replicate it exactly. An advantage of the simulator can be the step by step simulation. V-REP offers such a functionality where the simulator performs a specific amount of simulation steps and then pauses the simulation. By default, the remote API function calls of V-REP are asynchronousl. A simulation will advance or progress without taking into account the progress of the remote API client [46]. If for example V-REP is running on a different computer then the controller, a slow network can cause a bad simulation. A long delay will cause the controller to wait while the simulator is still running. With the synchronized option the simulator only performs a simulation step when the controller tells him to do so. Thus slow communication is not a problem anymore.

2.2. The robot e-Puck

e-Puck is the result of a project started at the Ecole Polytechnique Federale de Lausanne. It is a collaboration between the Laboratory of Intelligent System, Autonomous Systems Lab and Swarm-Intelligent System group. The goal of the project was to develop a robot for educational purpose at university level. They defined the following five features as important for a robot with educational purpose:

- The mechanical structure should be simple to understand and has a good clean modern system which is represented by the electronics, processor structure and software.
- To cover as many educational fields as possible it should be extendible, have many sensors and a fast processor.
- To be very user friendly it should be small, such that it can fit on the table next to the computer. It should be battery driven and require almost no wiring and optimal working comfort.
- Because it is for students it should be robust and in case of damage the repair cost should be small and the repair should be simple.

- It should be cheap to buy.

To give everyone the opportunity to improve the hardware of the robot, the documentation of the hardware is published with the Open Source Hardware Licence. At the time of writing this thesis, two versions of the robot are available. Version one was released in November 2015 and two years later version two was released [5].
The latest robot of the project is the e-Puck2 which is available since the beginning of 2018 [7].

All versions of e-Puck are equipped with distance sensors, an accelerometer and gyroscope, camera and several microphones. The e-Puck2 additionally has a compass. The actuators of the robot are two stepper motors, a speaker and LEDs. It is possible to upgrade the robot with a ground sensor, range and bearing sensor, RGB panel, Gumstix extension, omnivision or own design boards with sensors by using the I2C bus. Communication with the robot is possible via USB, Bluetooth 2.0, Bluetooth Low Energy as well as WiFi in the case of an e-Puck2. The initial version only allowed communication by RS232 and Bluetooth 2.0 [6, 16].

2.2.1. Virtual e-Puck of V-REP

V-REP comes with the virtual model of the robot e-Puck. All sensors and actuators of the basic e-Puck also exist in the virtual version of the robot. However, V-REP also supports a ground sensor which an e-Puck is not equipped with by default. In the V-REP model the ground sensor is implemented via a camera, which is shown in Code 2.1. There is no detail information about the e-Puck model of V-REP available, from the variable names of the code, the used vision sensor is a color sensor. The bottom left, bottom middle and bottom right pixel of the color sensor are passed on as the ground sensor values.

```
img=simGetVisionSensorImage(colorSensor)
data={img[1],img[22],img[94]}
simSendData(sim_handle_chain,0,'EPUCK_groundSens',simPackFloatTable(data))
```

Code 2.1.: Three pixels of the camera image are defined as the ground sensor values on the virtual e-Puck of V-REP.

2.3. Simulators V-REP and Webots

In this section an overview of the two simulators is given. First the simulator Webots is discussed and then the simulator V-REP.

2.3.1. Webots

Webots is the robot simulation software that is currently used in the 5th semester at the Vorarlberg University of Applied Sciences [54]. The students use this software to learn the basics of programming a robot for different tasks. Webots has APIs for C, C++, Java, Python and Mathlab to control a robot remotely [53]. For the students of the Vorarlberg University of Applied Sciences the possibility to program in Java is very important. Java is the programming language that is taught starting from the 2nd semester and therefore the main programming language during the bachelor's study. The installation of Webots includes a Java library as a JAR-file which is used to program in Java.

Programming the robot in Webots is not a straight forward process. Webots has a build-in code editor. The editor has a small issue, as it does not support auto completion which is a feature that makes development a lot easier. Some students experienced problems while compiling, as the system variables needed to be set properly such that they point to the Java JDK. Even with correct system variables there can be problems with the compiler. Most of the students therefore switched to another Integrated Development Environment (IDE) like Eclipse or Intellij. Programming in another IDE requires after compilation the *class-files* of Java to be copied from the project output to another directory. This directory then must have the same name as the java class which contains the main-function. Very important for students is the possibility to debug. Even if an IDE is used to program, it is not possible to debug the code because Webots requires the output files. Working with real robots debugging will not be possible as the program gets flashed on to the robot. But for a student this feature would be very helpful to locate problems faster and also to understand the robot better. Beside a missing debug feature another problem is the complicated use of an external JAR library with a controller. If a controller uses a library, it needs to be added to variable CLASSPATH manually like it is described on the Webots website [56].

Webots has the helpful feature monitor view, that lists all monitor values as shown in Figure 2.8. The window shows all the sensor values of e-puck making it very helpful to tune the program. In my case and for many other students this window almost never worked. Which is unacceptable for the high costs of this simulation software. In the control interface, shown in Figure 2.8, it is also possible to switch from the simulation to the real e-Puck. The commands are then sent by Bluetooth to the real robot.

Webots is based on a licence system where you can choose from different packages. The different packages have different features of the software unlocked [55]. The package with the highest cost is the *Webots PRO* for CHF 3450 (at the time of writing this thesis) which includes all features. *Webots EDU* has limited access to the features of Webots and costs CHF 320 per licence. One licence is valid for one computer running an instance of Webots. Hence for a class of 50 students, 50 licences are needed which results in very high costs as on education licence is CHF 320. Furthermore, the licences are only valid for a specific version of the softwareand do not qualify for software updates. Therefore the use of a new release of Webots requires the purchase of new licenses. Webots is available for Mac, Linux and Windows.

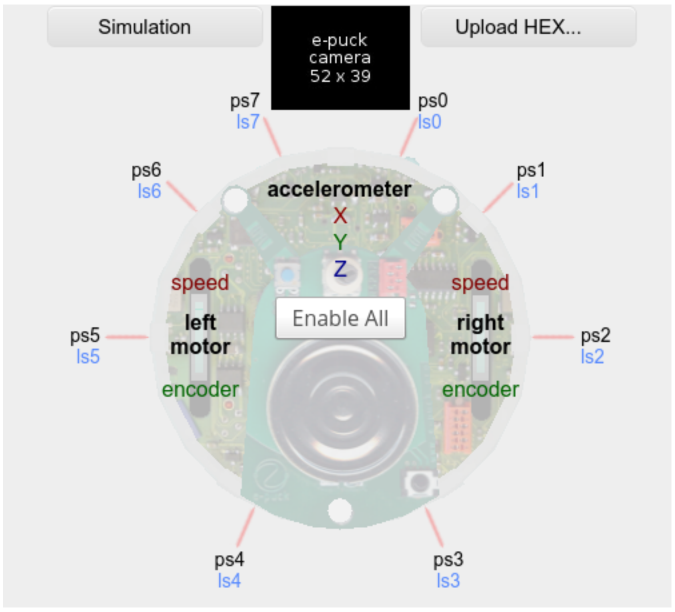


Figure 2.8.: Webots e-puck sensor monitor.

2.3.2. V-REP

Virtual Robot Experimentation Platform (V-REP) is a robot simulator with an integrated development environment [50]. This software is currently used at the Vorarlberg University of Applied Sciences theirs in Masters degree program in Computer Science in the specialisation autonomous systems. The programming is done by using the remote API for Python.

V-REP has multiple possibilities to control a robot simulation. Figure 2.9 shows an illustration of the V-REP framework, taken from their website [47]. With *Main Client Application* V-REP terms the executable. The Main Client Application handles the tasks of running the simulator, loading and unloading plugins, loading a scene or model and handles running simulations [44].
Native V-REP supports programming of the robots with Lua as an embedded script. "An embedded script is a script that is embedded in a scene (or model), i.e. a script that is part of the scene and that will be saved and loaded together with the rest of the scene (or model)" [43].
With plugins and add-ons, it is possible to extend the functionality of V-REP by writing own functions. The plugins and add-ons are loaded automatically by V-REP at the program start-up. For plugins, "the language can be any language able to generate a shared library and able to call exported C-functions (e.g. in the case of Java, refer to GCJ and IKVM). A plugin can also be used as a wrapper for running code written in other languages or even written for other microcontrollers (e.g. a plugin was written that handles and executes code for Atmel microcontrollers)" [45]. The add-ons of V-REP are written in Lua [42]. V-REP distinguishes between add-on functions and add-on scripts. The add-on functions will be executed once the user selects them.

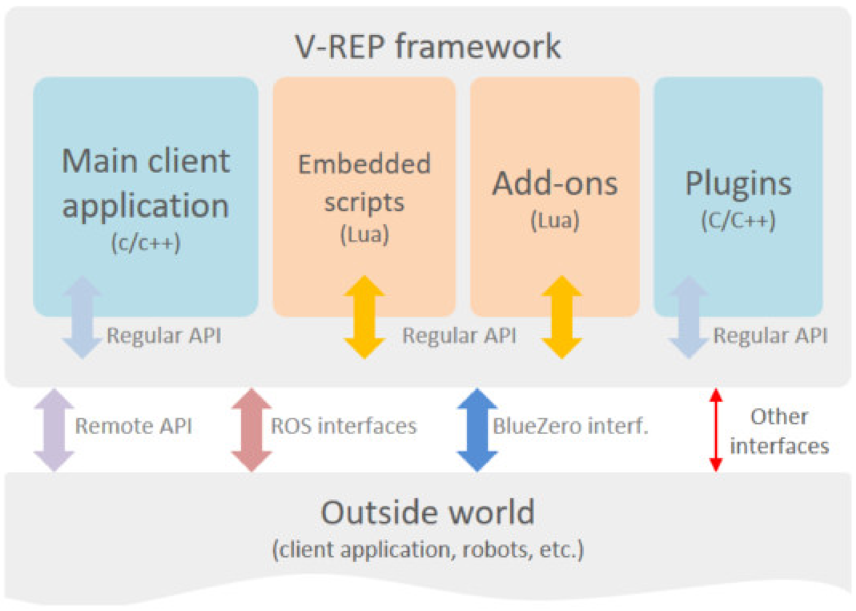


Figure 2.9.: V-REP API framework [47].

"Add-on scripts are executed constantly while the simulator is running, effectively running in the background. They should only execute minimalistic code every time they are called, since the whole application would otherwise slow down" [42].
With the remote API, ROS interface and the BlueZero interface V-REP is accessible from almost any possible external application or hardware according to their website [47]. "The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms" [22]. "BlueZero (in short, "B0") is a cross-platform middleware which provides tools for interconnecting pieces of software running in multiple threads, processes or machines. It has some similarities with ROS, although it only focuses on providing communication paradigms (client/server and publisher/subscriber) and message transport (based on ZeroMQ), while being agnostic to message serialization format or common protocols and data structures" [2].
To support more programming languages V-REP has a remote API [51, 52]. The remote API is available for Java, C/C++, Python, Matlab, Octave and Lua. "The remote API functions are interacting with V-REP via socket communication (or, optionally, via shared memory) in a way that reduces lag and network load to a great extent. All this happens in a hidden fashion to the user. The remote API can let one or several external applications interact with V-REP in a synchronous or asynchronous way (asynchronous by default), and even remote control of the simulator is supported (e.g. remotely loading a scene, starting, pausing or stopping a simulation for instance). The word Synchronous is used in the sense that each simulation pass runs synchronously with the remote API application" [51].

Since Lua is the main programming language in V-REP to program robots, it would be the easiest way to use this programming language. The students in the Bachelor studies, however, learn the programming language Java from the second semester onwards. V-REP will be used in the fifth semester. By then the students are very comfortable with Java and can focus on applying patterns and architectures. Changing the programming language could cause another difficulty and slow down the learning process as they first have to learn a new language. To avoid this problem the framework for java is needed.

When communicating with V-REP using the remote API it acts like a server. The functionality is implemented via a plugin, which is automatically loaded with V-REP at the program start-up. A client, which is programmed in one of the supported languages, can connect to the server and control a robot. It is possible that more than one client connects to the server. This makes it possible to control several robots in one V-REP instance. To control the robot, using the remote API, V-REP includes all needed libraries in the installation. Each supported programming language has its own set of libraries.

Other features of V-REP include a multiple physics engines that can be chosen from, inverse kinematic calculation, collision detection, distance calculation, building a scene, support for different proximity sensor types and modes of operation, vision sensors and recording simulations for later play back. The detailed information about all the features can be found on the website of V-REP [50].

V-REP is available in three different versions. There are two versions of V-REP that are free to use. The one version is the V-REP player which can play back simulations. V-REP Pro EDU is a non-limited version of V-REP for educational use. The educational licensing says it is free for Hobbyists, students, teachers, professors , schools and universities [49]. The commercial version is V-REP Pro which is for companies, research institutions, non-profit organisations and foundations. Also, the complete source code of V-REP is available on GitHub.

2.4. Python framework

The python framework developed by the Vorarlberg University of Applied Sciences consists of the four classes DifferentialWheels, ePuck, EPuckVRep and EPuckReal. The classes DifferentialWheels and EPuck are abstract classes. As the UML class diagram (in Figure 2.10) shows. The classes EPuckVRep and EPuckReal inherit from EPuck and EPuck from DifferentialWheels. To control the robot in the simulator V-REP the class EPuckVRep is used. With the class EPuckReal the real robot can be controlled.

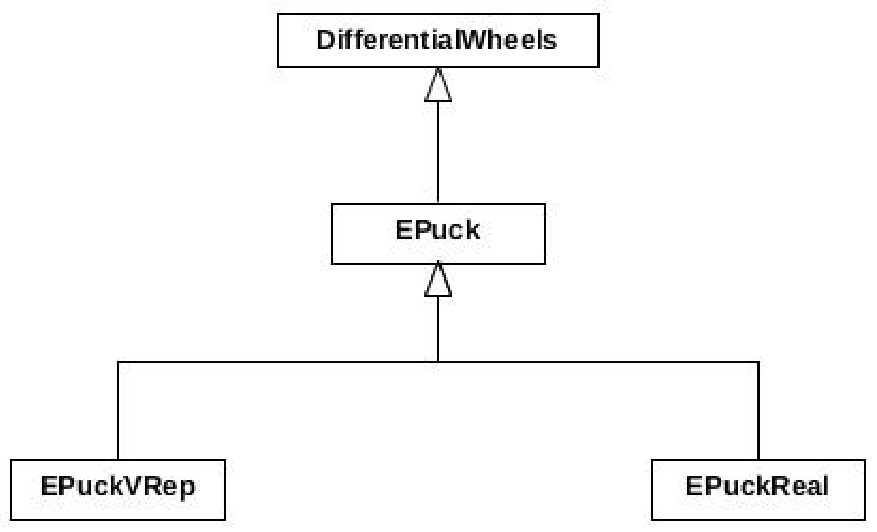


Figure 2.10.: Python framework inheritance

2.4.1. DifferentialWheels

The *DifferentialWheels* class is an abstract base class which describes the methods of a robot with differential wheel locomotion. Differential wheel locomotion means a robot with two fixed wheels, which can be driven independently. The implemented methods of this class are methods needed to control the robot e-Puck but also apply to any other robots with two wheels and the same sensors. Figure 2.11 shows the *DifferentialWheels* class with all its methods. Looking at the e-Puck's actuators, the wheels can be generalised. To control the wheels a method is needed to set the speed of the left motor and the right motor. Any other robot with two wheels can use this method as well. Therefore, the class implements a method to set the speed. As sensors like the distance sensor, accelerometer, gyroscope and camera can be expected to be found also on other robots and not only the e-Puck, *DifferentialWheels* defines methods to retrieve those values from a robot. Additionally, a robot might need activation of the sensors and thus it implements methods to activate them. Furthermore, the class implements an observer pattern for the sensor values and the camera image. This pattern is implemented to allow the software engineer to create, for example, a layered architecture to control a robot. The observer of sensor values is notified, which sensor values have changed. Camera image observers receive the camera image if it has changed.

2.4.2. EPuck

The class *EPuck* (Figure 2.12) represents the robot e-Puck and is a high-level proxy of the robot (virtual or real) for the controller. This class defines the methods, which return the sensor values and the camera image. These methods distinguish if the

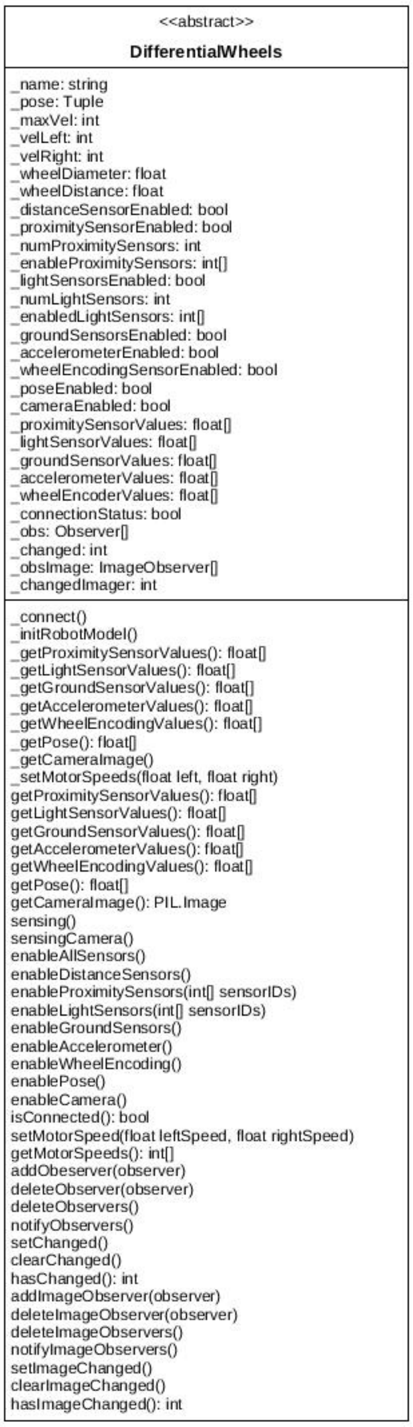

<<abstract>>
DifferentialWheels
_name: string
_pose: Tuple
_maxVel: int
_velLeft: int
_velRight: int
_wheelDiameter: float
_wheelDistance: float
_distanceSensorEnabled: bool
_proximitySensorEnabled: bool
_numProximitySensors: int
_enableProximitySensors: int[]
_lightSensorsEnabled: bool
_numLightSensors: int
_enabledLightSensors: int[]
_groundSensorsEnabled: bool
_accelerometerEnabled: bool
_wheelEncodingSensorEnabled: bool
_poseEnabled: bool
_cameraEnabled: bool
_proximitySensorValues: float[]
_lightSensorValues: float[]
_groundSensorValues: float[]
_accelerometerValues: float[]
_wheelEncoderValues: float[]
_connectionStatus: bool
_obs: Observer[]
_changed: int
_obsImage: ImageObserver[]
_changedImager: int
_connect()
_initRobotModel()
_getProximitySensorValues(): float[]
_getLightSensorValues(): float[]
_getGroundSensorValues(): float[]
_getAccelerometerValues(): float[]
_getWheelEncodingValues(): float[]
_getPose(): float[]
_getCameraImage()
_setMotorSpeeds(float left, float right)
getProximitySensorValues(): float[]
getLightSensorValues(): float[]
getGroundSensorValues(): float[]
getAccelerometerValues(): float[]
getWheelEncodingValues(): float[]
getPose(): float[]
getCameraImage(): PIL.Image
sensing()
sensingCamera()
enableAllSensors()
enableDistanceSensors()
enableProximitySensors(int[] sensorIDs)
enableLightSensors(int[] sensorIDs)
enableGroundSensors()
enableAccelerometer()
enableWheelEncoding()
enablePose()
enableCamera()
isConnected(): bool
setMotorSpeed(float leftSpeed, float rightSpeed)
getMotorSpeeds(): int[]
addObeserver(observer)
deleteObserver(observer)
deleteObservers()
notifyObservers()
setChanged()
clearChanged()
hasChanged(): int
addImageObserver(observer)
deleteImageObserver(observer)
deleteImageObservers()
notifyImageObservers()
setImageChanged()
clearImageChanged()
hasImageChanged(): int

Figure 2.11.: DifferentialWheels class

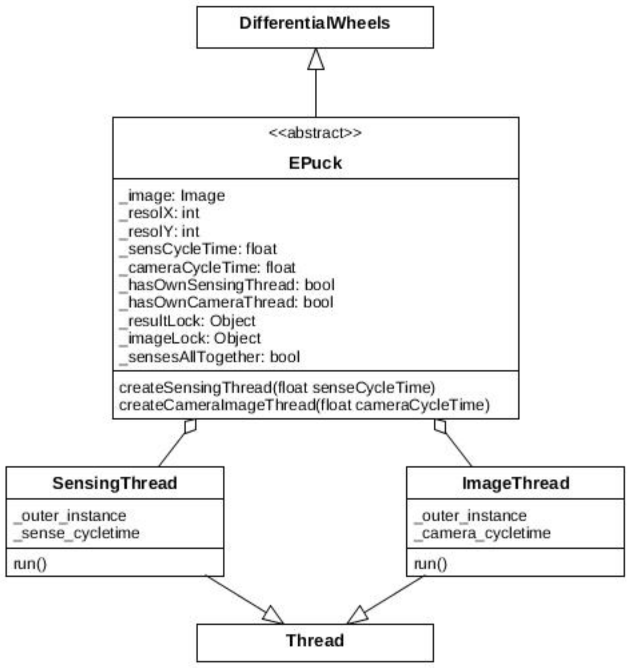


Figure 2.12.: UML diagram of the class EPuck

camera image or sensor values are updated by a thread, sensor values being requested all at once or if the sensor value or camera image are requested separately. Requesting separately means that only the value of a single sensor is requested. Requesting all at once means that only one call is made to get all values. As the Figure 2.12 shows, the class *EPuck* makes use of the two classes *SensingThread* and *ImageThread*. *EPuck* implements those two classes to be able to allow the use of the observer pattern which is defined by the class *DifferentialWheels*. The observer pattern allows the software engineer to use different architectures like it is explained in Section 2.4.1. For both threads the interval can be set when initialising the threads, which gives the opportunity to adjust it independently.

2.4.3. EPuckVRep

The class *EPuckVRep* (Figure 2.15) is a proxy of the virtual robot e-Puck of V-REP for the controller. Therefore, this class implements all abstract methods that are defined by its parent class *EPuck*. Using the remote API, sensor values and the camera image are requested, and the motor speed can be set. To be able to use the remote API functionality of V-REP the classes *vrep.py* and *vrepConst.py* are needed. Those two classes are provided by V-REP and need to be included in the folder of the controller to access the remote API. The library comes with the installation of V-REP and is the *remoteApi.so* for Unix, the *remoteApi.dll* for Windows or the *remoteApi.dylib* for Darwin. Depending on the platform the application is running on,

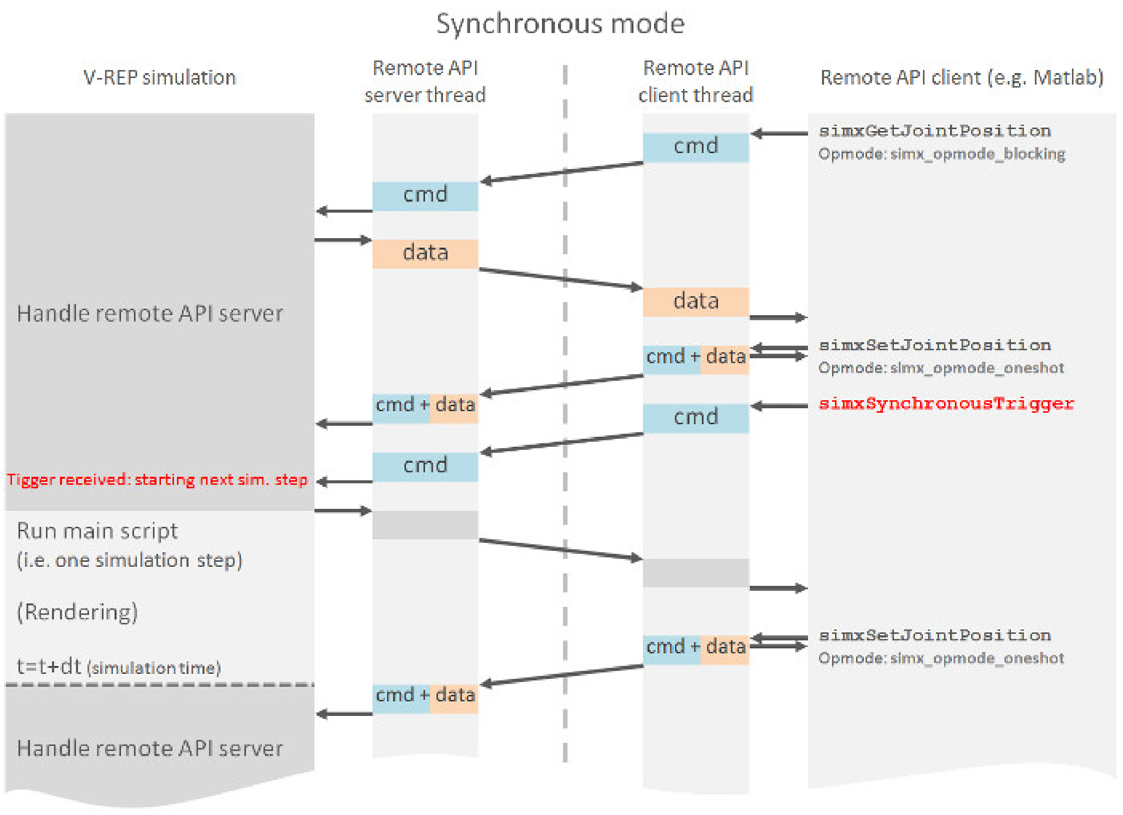


Figure 2.13.: Illustration of the V-REP synchronous mode [46].

one of the three remote Api files needs to be provided. *EPuckVRep* makes use of the possibility to run the simulation of V-REP in synchronous mode, which was mentioned in Section 2.3.2. Therefore, the methods *startsim* and *stepsim* are implemented. The method *startsim* needs to be called to initiate the simulation. With the method *stepsim* the amount of simulation steps performed by V-REP can be defined. Using this method, a very precise simulation, independent of the frame rate, can be achieved. Figure 2.13 shows an illustration of the synchronous mode, the illustration is taken from V-REP [46]. While the function *simxGetJointPosition* and *simxSetJointPosition* are called, the simulator is not running the simulation. The simulation starts running for one simulation step after calling *simxSynchronousTrigger*. After finishing one simulation step the simulator waits for the next command to continue the simulation. The calls made to the API by the framework are blocking function calls, like in the illustration for the synchronous mode. Figure 2.14 shows the illustration of the blocking function call. Using the blocking function call causes a delay on the client side. The block takes as long as it takes for the simulator to return the requested value or to acknowledge that the command was processed.

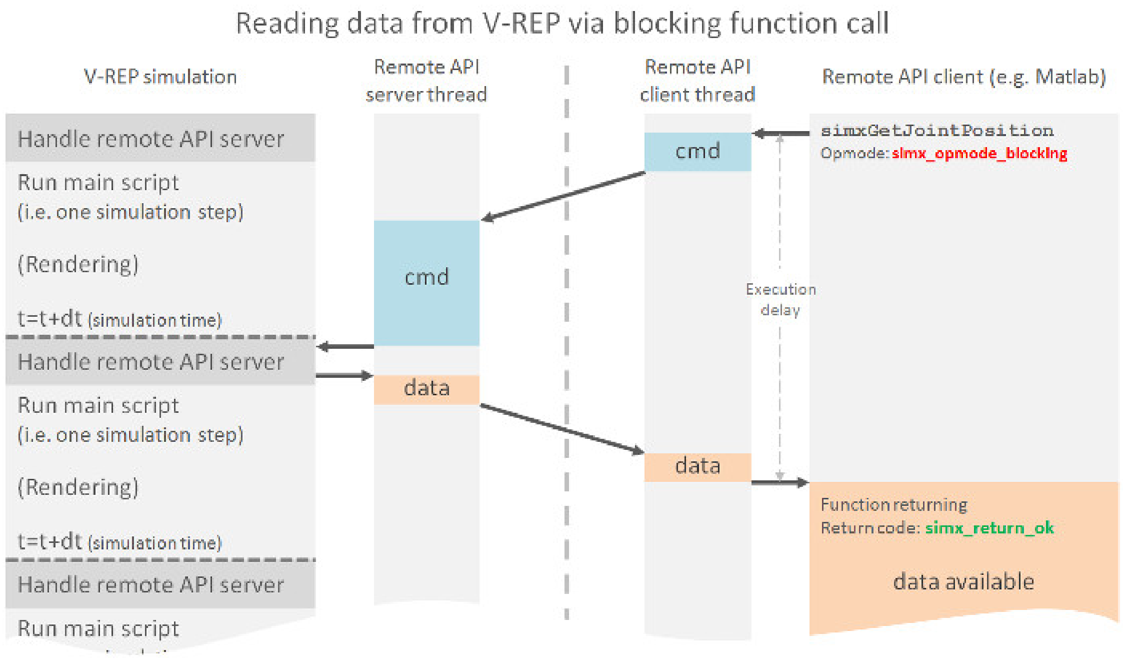


Figure 2.14.: Illustration of the V-REP blocking function call [46].

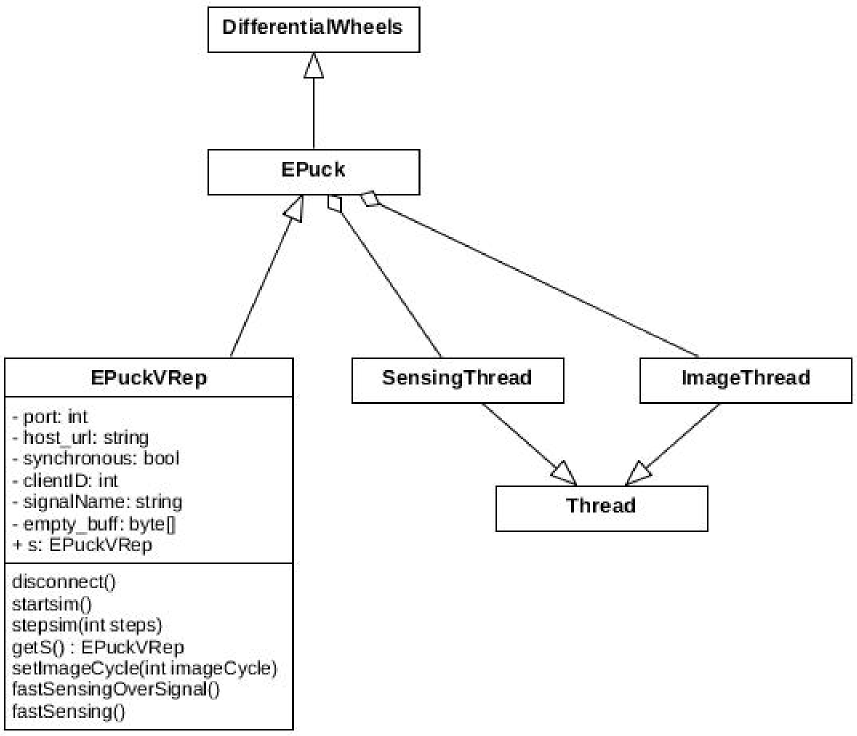


Figure 2.15.: UML diagram of the class EPuckVrep

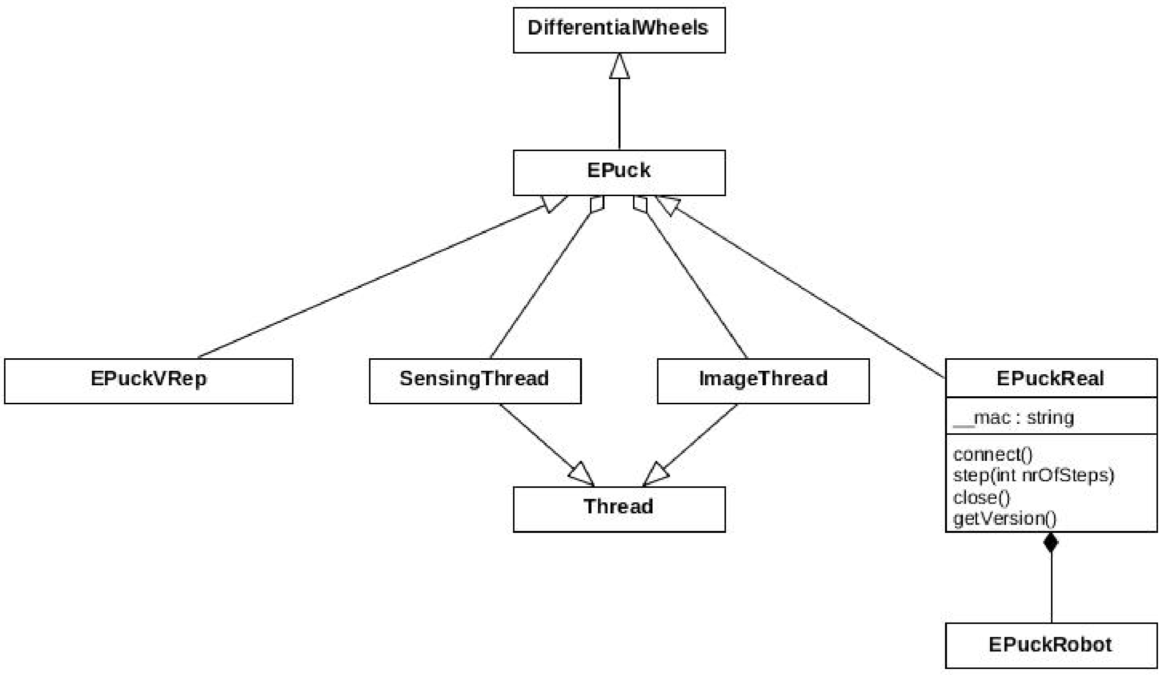


Figure 2.16.: UML diagram of the class EPuckReal.

2.4.4. EPuckReal

Since the class *EPuckRobot* was not developed by the Vorarlberg University of Applied Sciences it will not be ported to Java. Hence this class will not be covered in detail. The class *EPuckReal* (Figure 2.16) is the proxy of the real robot. Because this class inherits from the class *EPuck* all the abstract methods are implemented. The class *EPuckRobot* implements methods to send commands to the robot. The connection to the robot is established by Bluetooth. To connect to the robot, the class *EPuckRobot* needs the MAC address of the robot.

2.5. Java

This part covers the available libraries that can be used for the implementation of the framework. Existing libraries for V-REP to control the e-Puck robot, libraries for the camera image and the Java classes that are provided by V-REP will be discussed.

2.5.1. Existing libraries for the robot e-Puck for V-REP

At the time of writing this thesis only one project could be found that controls the robot e-Puck inside of V-REP. The project is available on github and was developed by the user *AurelienC* [1]. The implemented functionality is very limited and specific for a certain problem. The class *VRepHelper.java* that can be found in the directory *src/fr/esisar/px504/simulation* implements a few simple methods like reading sensors, move forward, turn, start / stop the simulation and moving to a specific point. For example, the method (Code 2.2) that reads the distance sensors returns the information if the sensor detected an obstacle. Since the functionality is very limited and too specific, this project was only taken as a reference on how to interact with the remote API.

```java
/**
 * Ge the state of sensors of E-Puck robot
 * @param operationMode The type of query (opMode in V-Rep)
 */
private void refrechSensorsWithSettings(int operationMode) {
  IntW ret = new IntW(0);

  vrep.simxGetIntegerSignal(clientId, "sensors", ret, operationMode);

  boolean[] bits = new boolean[8];
  for (int i = 7; i >= 0; i--) {
          bits[i] = (ret.getValue() & (1 << i)) != 0;
  }

  this.sensors = bits;
}
```

Code 2.2.: Method to read sensor values by AurelienC

2.5.2. Camera image

When requesting the image from the API, the return value is an array. A pixel is represented by three entries in the array for red, green and blue, respectively. For further use of the image it might be good to store the image in a format which makes it easy to use.

When determining the best data format to store the image a few factors have to be taken into account. When controlling a robot, it is very important that the access to the information of an image is fast. It is also very important that converting from the API format to the other format does not take too much time. The second factor is the maintenance.In case of choosing a third-party library to store the image, the framework is dependent on the maintainer of this library to fix problems. The time between reporting an error and publishing of a new library version where the error is fixed can be long. When developing a on library fixing errors can be done immediately and therefore make maintenance easier. The third factor is the compatibility to libraries like Java Advanced Imaging (JAI), Catalano or OpenCV [13, 15, 18]. A preference for a specific image processing library already determines the data format and can therefore make the decision much easier.

The data type that OpenCV uses, to store images, is their own type *Mat* as it can be found in the Java documentation of the library [17]. The type *Mat* is a matrix and the image can be loaded in two different approaches. The first approach is to create the *Mat* object from an image file. Code 2.3 shows an example from tutorialspoint on how to read an image from a file. The second approach is to create a new *Mat* object and then use the *put* method, shown in Figure 2.17, of the Java documentation, to assign the image [17]. The *put* method offers various data types on how the image can be provided. Code 2.4 shows the two lines of code which are needed to load and assign the image. Since saving the image to a file and then load it into OpenCV is time consuming, the e-Puck framework must offer a possibility to provide the image as a data type which can then be used in combination with the *put* method.

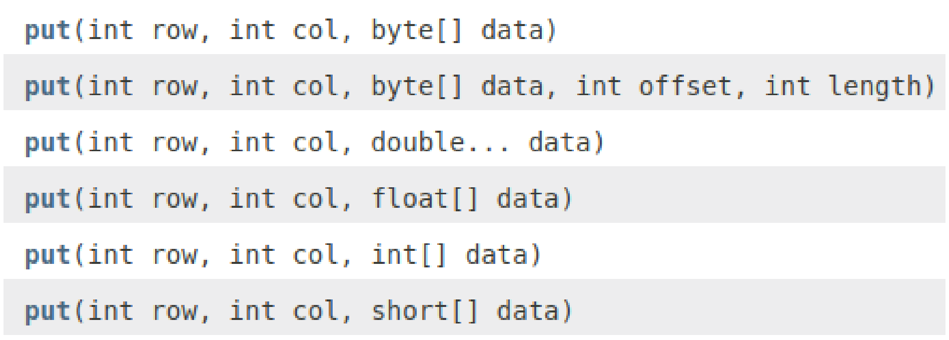
```java
import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.imgcodecs.Imgcodecs;

public class ReadingImages {
   public static void main(String args[]) {
      //Loading the OpenCV core library
      System.loadLibrary( Core.NATIVE_LIBRARY_NAME );

      //Instantiating the Imagecodecs class
      Imgcodecs imageCodecs = new Imgcodecs();

      //Reading the Image from the file
      String file ="C:/EXAMPLES/OpenCV/sample.jpg";
      Mat matrix = imageCodecs.imread(file);

      System.out.println("Image Loaded");
   }
}
```

Code 2.3.: Example of reading a image with OpenCV [38].

```java
Mat mat = new Mat(width, height, CvType.CV_8UC3);
mat.put(0, 0, data); //data is can be any type of the accepted data types
```

Code 2.4.: Using *put* to load the image in OpenCV.

JAI uses the data type *RenderedImage* to store images [12]. The most common way to load an image is shown in Code 2.5, which is an example from Oracle, where the image



```
put(int row, int col, byte[] data)
put(int row, int col, byte[] data, int offset, int length)
put(int row, int col, double... data)
put(int row, int col, float[] data)
put(int row, int col, int[] data)
put(int row, int col, short[] data)
```

Figure 2.17.: Different variations of the *put* method of the OpenCV class *Mat*

is loaded from a file [13]. Another possibility to create a *RenderedImage* is from the data type *ParameterBlock* [11]. With the method *add* of *ParameterBlock* a *BufferedImage* can be added [14]. By using the method *getRenderedSource* of *ParameterBlock* a *RenderedImage* can be generated. Code 2.6 shows an example how to get a *ReneredImage* from a *BufferedImage*.

```
String filename = "// path and name of the file to be read,
                     that is on an accessible filesystem //";

RenderedImage image = JAI.create("fileload", filename);
```

Code 2.5.: Loading a image from a file in JAI

```
ParameterBlock pb = new ParameterBlock();
pb.add(myBufferedImage);//myBufferedImage could be the image from the API
                        //converted to a BufferedImage */
RenderedImage renderedImage=paramBlock.getRenderedSource(0);
```

Code 2.6.: Creating a RenderedImage from a BufferedImage with JAI

The Java class *BufferedImage* is a good option for storing the camera image. It is possible to use it with JAI and also with OpenCV. The *BufferedImage* can be used with OpenCV because it provides a method which returns a byte array. The class also has methods to retrieve width, height and the RGB (red, green and blue) value for a specific pixel. However, for example, the RGB value of a specific pixel is returned as a single value. From the single value, the rgb values must be obtained by some calculations. A solution for this could be to create a class which holds a *BufferedImage* and offers methods to get the red, green and blue value of a pixel.

2.5.3. Java API provided by V-REP

The remote API is one of five different possibilities to communicate with the simulator [47]. V-REP lists a 6th possibility which is using plugins. It is listed separately because these plugins are not part of V-REP. V-REP provides some classes to be able to use the remote API. These classes come with the installation of V-REP. The subdirectory */programming/remoteApiBindings/* of the program directory contains directories for multiple programming languages. The supported programming languages for the remote API are:

- Java
- Lua
- Matlab
- Octave
- Python
- Urbi

V-REP provides two different Java API libraries for Linux and Windows. There is a remote API library file for the 32bit system and one for the 64bit system. For Mac it is only one file. Beside some example applications, which use the remote API of V-REP, the directory *Java* contains Java classes that are needed to work with the remote API. The most important class is *remoteApi.java.* This class implements methods to access the API of V-REP and also loads the library file. The other class files provided are data types. V-REP provides these classes since they don't use the standard Java data types as return values for the API functions.

2.6. Comparison Java and Python 2.7

Even though, Java and Python, are object-oriented programming languages, they have some differences in their implementation. This first section covers the definition of object-oriented programming. In the second section Python 2.7 and Java are compared to each other.

2.6.1. Object Oriented Programming

The Cambridge Dictionary describes *object-oriented* as "based on groups of information and their effects on each other, rather than on a series of instructions" [3].

Looking up OOP on Techterms, it says “refers to a programming methodology based on objects, instead of just functions and procedures. These objects are organized into classes, which allow individual objects to be grouped together. An "object" in an OOP language refers to a specific type, or "instance," of a class. Each object has a structure similar to other objects in the class but can be assigned individual characteristics. An object can also call functions, or methods, specific to that object. Object-oriented programming makes it easier for programmers to structure and organize software programs. Because individual objects can be modified without affecting other aspects of the program, it is also easier to update and change programs written in object-oriented languages." [33].

Techtarget defines it as: "Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data." [31].

2.6.2. Comparison

In this section the two programming languages are compared to each other to get an overview on how they work.

2.6.2.1. Basic classes

Code 2.7 shows how a class is defined in Python. A class is designed with *class* followed by the class name.

```
class foo:
    def __init__(self):
        #inside "__init__" declarations and initialisations can be done
```

Code 2.7.: Defining a class in Python

Code 2.8 shows the same class implemented in Java. Compared to Python there is a modifier which defines the class as public. This is the common way, but it is also possible to leave the *public* away which then also means that the class is public.

```
public class foo{
        public Foo(){
                //The constructor can be public or private
        }
}
```

Code 2.8.: Defining a class in Java

Now let us have a look at how instance and class variables are handled in both programming languages. Code 2.9 and Code 2.10 implement the same variables. But there is a considerable difference in the syntax.

```
class foo:
    x = 4711 #class variable

    def __init__(self):
            self._y = 959 #"private" instance variable
```

Code 2.9.: Instance and class variables in Python

```
public class Foo{
        public static int x1 = 4711; //Declaration and initialisation
        //of public class variable

        private int y; //Declaration of instance variable

        public Foo(){
                y = 959; //Initialisation of instance variable
        }
}
```

Code 2.10.: Instance and class variables in Java

One difference between Python and Java is that there are no data types for variables and no declaration of a variable in Python. In Python all variables outside the constructor, which is *__init__(self)*, are class variables and all variables initialised inside the constructor are instance variables. In Java a declaration of the variable with the data type is needed and then the variable can be initialised. The declaration and initialisation can also be done in one step like it is done for the class variable.
The next difference is that there are no modifiers like in Java to define if the variable is *public, private* or *protected*. Every variable is public in Python but there is a coding convention which says that you should not touch a variable which has an underscore before the name. So, the variable *_y* inside the constructor in Code 2.9 should not be touched from the outside. Defining a variable as a constant like in Java with *final* is also not possible. Again, this is handled by the coding convention which says that a variable with a name in capital letters should not be changed.

The declaration of methods and functions in Python is also very different. Code 2.11 shows an example class with methods and functions. Again, in Python there is no modifier to define if the method is *public, private or protected*. With the underscore before the name you can indicate that the method should not be used from the outside. Furthermore, the signature contains no information if the method has a return value and the parameters passed have no data type. However there is an option to set a default value for a parameter which is done by assigning a value to the parameter with *value=none* like it is done for function *f()*.

```
class foo:
    x = 4711

    def __init__(self):
            self._y = 959

    def f(self, value=none)#method is public
            if value is none:
                    value = [] #empty array
            # modify here

        def _sum(self, a,b): #method is "private"
                sum = a + b
                return a

        def incrementY(self): #public method increments _y
                _y++
```

Code 2.11.: Instance and class variables in Python

In Java the same class would look like shown in Code 2.12. Declarations of methods in Java have a modifier which defines if the method is *public*, *private* or *protected*. If the method has a return value the data type of it is defined. The data type of parameters is also defined. If a method or function returns nothing, this is indicated with *void*. There is no direct representation of the function *f()* from the Python code in Java as it is not possible to give a parameter a default value.

```
public class Foo{
        public static int x1 = 4711;

        private int y;

        public Foo(){
                y = 959;
        }

        //is private and returns an integer value
        private int sum(int a, int b){
                int sum = a + b;
                return sum;
        }

        //is public and increments y
        public void incrementY(){
                y++;
        }
}
```

Code 2.12.: Instance and class variables in Java

2.6.2.2. Inheritance

Inheritance is an important feature of object-oriented programming. Both programming language have inheritance and Code 2.13 shows the syntax for inheritance in Python. For the given syntax the derived class must be in the scope of the base class. It is also possible to inherit from a class in another module. Therefore, the module name is put before the class name like shown in Code 2.14.

```
class DerivedClassName(BaseClassName):
    def __init__(self)
            #declaration and initalisation of a variable could be done here
```

Code 2.13.: Inheritance in Python

```
class DerivedClassName(modname.BaseClassName):
        def __init__(self):
        super(DerivedClassName, self).__init__()
    .
    .
```

Code 2.14.: Inheritance from a class in an other module in Python

In terms of inheritance Python also allows multiple inheritance. Code 2.15 shows the syntax for multiple inheritance. The super constructor must be called before any other call in the constructor. The rule to search for an attribute is depth-first from left to right. This means if an attribute is not found in *DerivedClassName*, the *Base1* will be recursively searched and if it is not found it goes on to the next base class. This is the solution in Python for the so called *diamond problem* of multiple inheritance [21].

```
class DerivedClassName(Base1, Base2, Base3):
        def __init__(self):
        super(DerivedClassName, self).__init__()
    .
    .
    .
```

Code 2.15.: Multiple inheritance in Python

The diamond problem is a problem that occurs with multiple inheritance. The name comes from the shape of the class diagram that results out of multiple inheritance, as shown in Figure 2.18. Imagine *SuperClass* implements a method *foo()* that *ClassA* and *ClassB* override but is not overridden by *ClassC*. Which method is called when calling *foo()* on *ClassC*? Python solves this problem with the described rule.

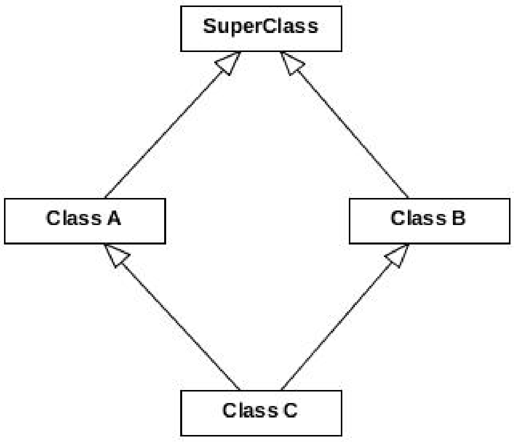


Figure 2.18.: Class diagram for multiple inheritance

In Java only single inheritance is allowed. The syntax for the inheritance is shown in Code 2.16. Important here is that the constructor of the super class must be the first call in the constructor of the derived class.

```
public class FooExtended extends BaseFoo{
        public FooExtended(){
                super();
        }
        .
        .
        .
}
```

Code 2.16.: Inheritance in Java

2.6.2.3. Interfaces

Java has the possibility to define interfaces. They are used to group a set of methods that belong together. All the methods defined in an interface have an empty body. An example for an interface is shown in Code 2.17 and the usage of it in Code 2.18. The compiler will check if the methods of the interface are implemented. If not all methods are implemented, the compilation will fail. In Python there does not exist something like an interface.

```
interface Interfaceable{
        void foo(int value);
}
```

Code 2.17.: Interface implementation in Java

```
public class ClassA implements Interfacable{
        private int i;
        public ClassA(){
                i = 0;
        }
        void foo(int value){
                i += value;
        }
}
```

Code 2.18.: Class which implements an interface in Java

2.6.2.4. Abstract classes

Abstract classes are useful as they make it possible to declare methods without an actual implementation of it. They are used to build up a class hierarchies where the first class is an abstract class and defines the behaviour which all subclasses have to implement. It is not possible to instantiate an abstract class. In Python there are a few steps needed to create an abstract class. To create a abstract class the class *ABCMeta* must be imported from the module *abc (abstract base class)*. This class has to be assigned to the variable *__metaclass__* which is a variable that describes the behaviour of the class. Code 2.19 shows an example for defining an abstract class.

```
from abc import ABCMeta
class BaseClass:
        __metaclass__ = ABCMeta
        .
        .
        .
```

Code 2.19.: Defining an abstract class in Python

To define an abstract method in Python a decorator needs to be added to an empty method. The decorator is also part of the ABCMeta class and is called *abstractmethod*. An empty method is created with the *pass* statement. When the *pass* statement is executed, nothing happens as it is null. Code 2.20 shows the use of the decorator and the pass statement.

```
from abc import ABCMeta
class BaseClass:
        __metaclass__ = ABCMeta

        @abstractmethod
        def foo(self)
                pass
```

Code 2.20.: Abstract class with an abstract method in Python

Java uses a class modifier and a method modifier to make it abstract. If we take a look at the Code 2.21 we can see how easy it is compared to Python. The method just consists of the declaration with the additional modifier *abstract*.

```
public abstract BaseClass{

        abstract void foo();
}
```

Code 2.21.: Abstract class with an abstract method in Java

3. Requirements

The current functionality available in the Python framework should also be available in the Java framework, with the exception that the Java framework does not need to be able to control the real robot. Controlling the real robot is at this point not needed as the students will only work with the simulator. Furthermore, including the functionality, of controlling a real robot, would exceed scope for a Bachelor Thesis. Since the students in the 5th semester often faced problems with the loading of the V-REP libraries, it would be wishful to be able to solve this problem with the Java framework. The Java framework therefore, should automatically detect the operating system and load the correct library.

For the framework it is important that the camera image format has good compatibility. The reason being that two different libraries for image processing, that are used at the Vorarlberg University of Applied Sciences. During the 5th semester of the Bachelor program, the students learn the basics of image processing using the library JAI. The library OpenCV is used in the elective course *Autonomous Systems* during the Masters program in Computer Science. For being capable for switching between the libraries, the camera image format needs to be compatible with both.

4. Implementation

Besides the implementation of the framework, creating a controller for a specific problem is also part of the thesis. This section covers details on the implementation of the framework and the application example.

4.1. Framework

This part covers how the framework for the robot e-Puck is structured and also shows how the calls to the remote API work.

The Java framework is built with the same structure as the Python framework. Figure 2.10 shows how the classes inherit in the Python framework. The difference in the Java framework is that the class for the real e-Puck is not implemented since it falls outside the scope of this thesis. Figure 4.1 shows the full class diagram for the Java framework. The aim, while designing the framework, was to make use of the object orientation of Java. Therefore, some new classes were introduced to handle some of the sensor values. In the next sections each part of the framework will be covered in detail.

4.1.1. DifferentialWheels

Like for the Python framework, the class *DifferentialWheels* describes a robot with differential wheel locomotion. The class is an abstract class as it just describes the robot and can be applied to any robot with a differential wheel locomotion. Figure 4.2 shows the class diagram for the *DifferentialWheel* class, with the other classes it uses. Compared to the Python framework the Java framework makes use of the object orientation for the values of some of the sensors.

In the Python framework arrays are used for the sensor values. Each index of the array has a specific value. Working with this array means, that knowledge of what index maps to which sensor value is always required. This problem can be solved by using the adapter pattern, also known as wrapper. Geeksforgeeks defines the adapter pattern as: "The adapter pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces." [9]. The classes *Acceleration*, *WheelEncode* and *Pose* use the adapter pattern to make the returned information easier accessible. The class *Acceleration* takes the array with the values and stores it. Three methods then make

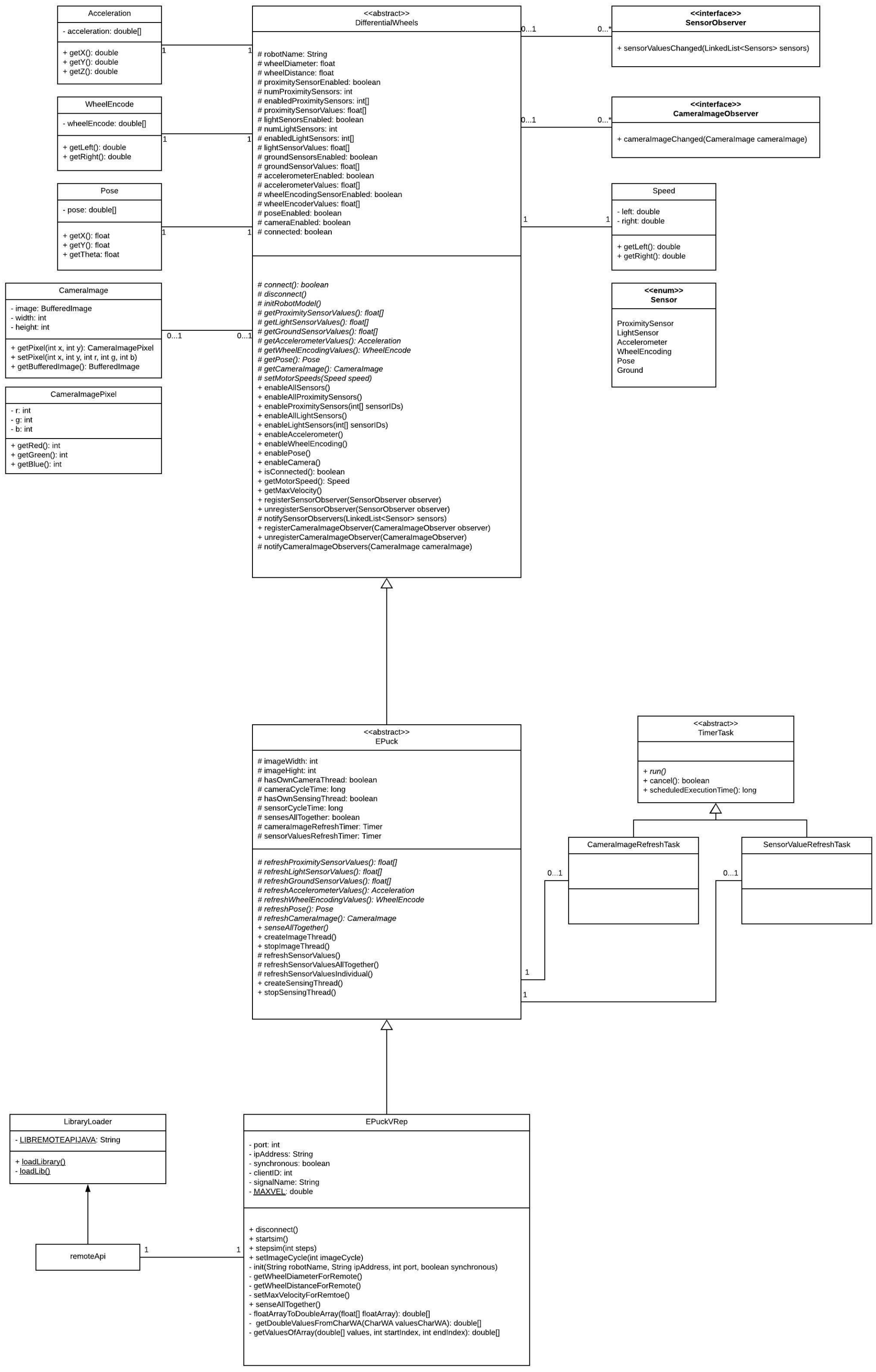

Acceleration
- acceleration: double[]
+ getX(): double
+ getY(): double
+ getZ(): double
WheelEncode
- wheelEncode: double[]
+ getLeft(): double
+ getRight(): double
Pose
- pose: double[]
+ getX(): float
+ getY(): float
+ getTheta: float
CameraImage
- image: BufferedImage
- width: int
- height: int
+ getPixel(int x, int y): CameraImagePixel
+ setPixel(int x, int y, int r, int g, int b)
+ getBufferedImage(): BufferedImage
CameraImagePixel
- r: int
- g: int
- b: int
+ getRed(): int
+ getGreen(): int
+ getBlue(): int
<<abstract>>
DifferentialWheels
# robotName: String
# wheelDiameter: float
# wheelDistance: float
# proximitySensorEnabled: boolean
# numProximitySensors: int
# enabledProximitySensors: int[]
# proximitySensorValues: float[]
# lightSensorsEnabled: boolean
# numLightSensors: int
# enabledLightSensors: int[]
# lightSensorValues: float[]
# groundSensorsEnabled: boolean
# groundSensorValues: float[]
# accelerometerEnabled: boolean
# accelerometerValues: float[]
# wheelEncodingSensorEnabled: boolean
# wheelEncoderValues: float[]
# poseEnabled: boolean
# cameraEnabled: boolean
# connected: boolean
# connect(): boolean
# disconnect()
# initRobotModel()
# getProximitySensorValues(): float[]
# getLightSensorValues(): float[]
# getGroundSensorValues(): float[]
# getAccelerometerValues(): Acceleration
# getWheelEncodingValues(): WheelEncode
# getPose(): Pose
# getCameraImage(): CameraImage
# setMotorSpeeds(Speed speed)
+ enableAllSensors()
+ enableAllProximitySensors()
+ enableProximitySensors(int[] sensorIDs)
+ enableAllLightSensors()
+ enableLightSensors(int[] sensorIDs)
+ enableAccelerometer()
+ enableWheelEncoding()
+ enablePose()
+ enableCamera()
+ isConnected(): boolean
+ getMotorSpeed(): Speed
+ getMaxVelocity()
+ registerSensorObserver(SensorObserver observer)
+ unregisterSensorObserver(SensorObserver observer)
# notifySensorObservers(LinkedList<Sensor> sensors)
+ registerCameraImageObserver(CameraImageObserver observer)
+ unregisterCameraImageObserver(CameraImageObserver)
# notifyCameraImageObservers(CameraImage cameraImage)
<<interface>>
SensorObserver
+ sensorValuesChanged(LinkedList<Sensors> sensors)
<<interface>>
CameraImageObserver
+ cameraImageChanged(CameraImage cameraImage)
Speed
- left: double
- right: double
+ getLeft(): double
+ getRight(): double
<<enum>>
Sensor
ProximitySensor
LightSensor
Accelerometer
WheelEncoding
Pose
Ground
<<abstract>>
EPuck
# imageWidth: int
# imageHight: int
# hasOwnCameraThread: boolean
# cameraCycleTime: long
# hasOwnSensingThread: boolean
# sensorCycleTime: long
# sensesAllTogether: boolean
# cameraImageRefreshTimer: Timer
# sensorValuesRefreshTimer: Timer
# refreshProximitySensorValues(): float[]
# refreshLightSensorValues(): float[]
# refreshGroundSensorValues(): float[]
# refreshAccelerometerValues(): Acceleration
# refreshWheelEncodingValues(): WheelEncode
# refreshPose(): Pose
# refreshCameraImage(): CameraImage
+ senseAllTogether()
+ createImageThread()
+ stopImageThread()
# refreshSensorValues()
# refreshSensorValuesAllTogether()
# refreshSensorValuesIndividual()
+ createSensingThread()
+ stopSensingThread()
<<abstract>>
TimerTask
+ run()
+ cancel(): boolean
+ scheduledExecutionTime(): long
CameraImageRefreshTask
SensorValueRefreshTask
LibraryLoader
- LIBREMOTEAPIJAVA: String
+ loadLibrary()
- loadLib()
remoteApi
EPuckVRep
- port: int
- ipAddress: String
- synchronous: boolean
- clientID: int
- signalName: String
- MAXVEL: double
+ disconnect()
+ startsim()
+ stepsim(int steps)
+ setImageCycle(int imageCycle)
- init(String robotName, String ipAddress, int port, boolean synchronous)
- getWheelDiameterForRemote()
- getWheelDistanceForRemote()
- setMaxVelocityForRemtoe()
+ senseAllTogether()
- floatArrayToDoubleArray(float[] floatArray): double[]
- getDoubleValuesFromCharWA(CharWA valuesCharWA): double[]
- getValuesOfArray(double[] values, int startIndex, int endIndex): double[]
1
0...1
0...*

Figure 4.1.: Full UML class diagram of the Java framework.

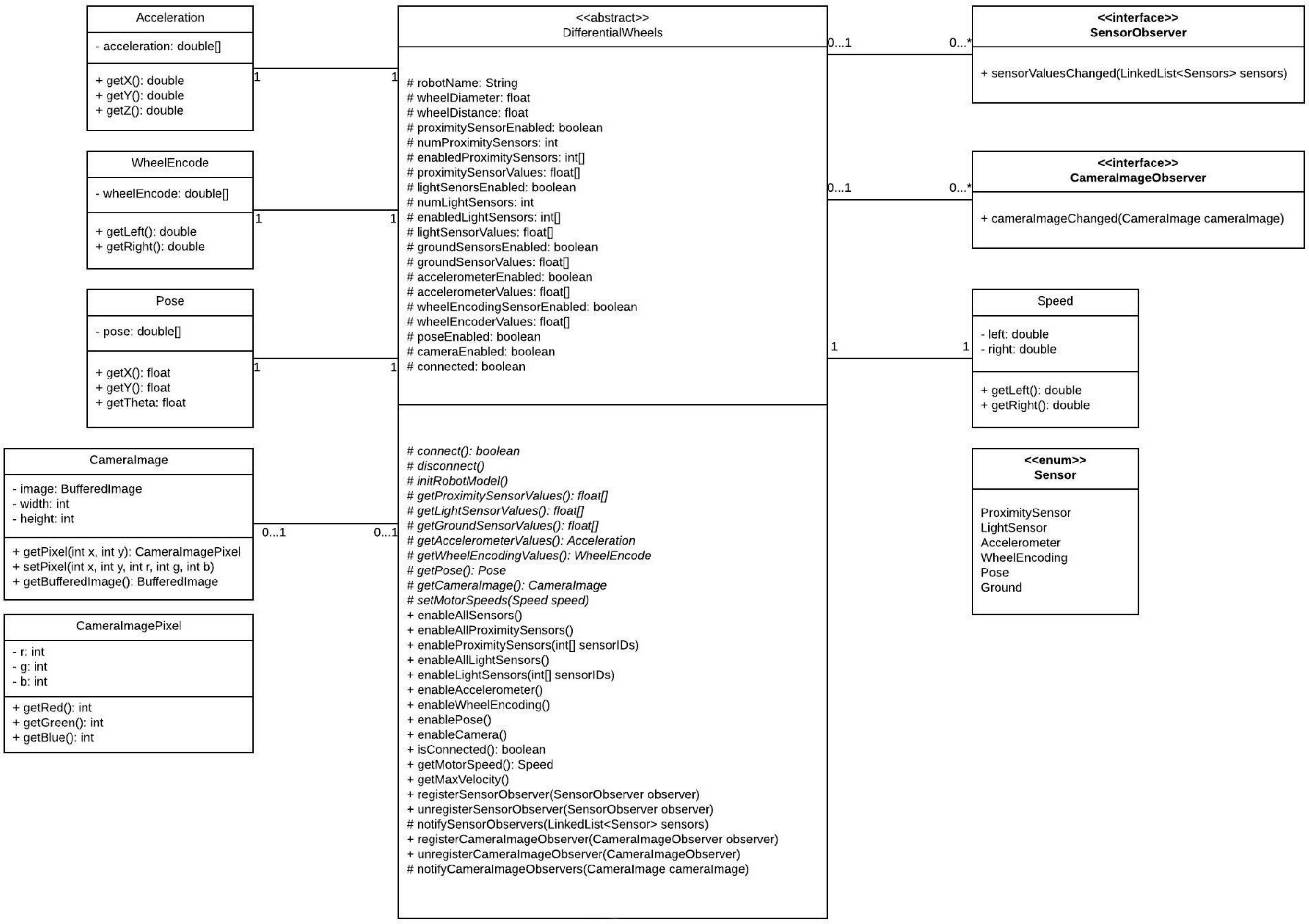

Acceleration
- acceleration: double[]
+ getX(): double
+ getY(): double
+ getZ(): double
WheelEncode
- wheelEncode: double[]
+ getLeft(): double
+ getRight(): double
Pose
- pose: double[]
+ getX(): float
+ getY(): float
+ getTheta: float
CameraImage
- image: BufferedImage
- width: int
- height: int
+ getPixel(int x, int y): CameraImagePixel
+ setPixel(int x, int y, int r, int g, int b)
+ getBufferedImage(): BufferedImage
CameraImagePixel
- r: int
- g: int
- b: int
+ getRed(): int
+ getGreen(): int
+ getBlue(): int
1
1
1
1
1
1
0...1
0...1
<<abstract>>
DifferentialWheels
# robotName: String
# wheelDiameter: float
# wheelDistance: float
# proximitySensorEnabled: boolean
# numProximitySensors: int
# enabledProximitySensors: int[]
# proximitySensorValues: float[]
# lightSenorsEnabled: boolean
# numLightSensors: int
# enabledLightSensors: int[]
# lightSensorValues: float[]
# groundSensorsEnabled: boolean
# groundSensorValues: float[]
# accelerometerEnabled: boolean
# accelerometerValues: float[]
# wheelEncodingSensorEnabled: boolean
# wheelEncoderValues: float[]
# poseEnabled: boolean
# cameraEnabled: boolean
# connected: boolean
# connect(): boolean
# disconnect()
# initRobotModel()
# getProximitySensorValues(): float[]
# getLightSensorValues(): float[]
# getGroundSensorValues(): float[]
# getAccelerometerValues(): Acceleration
# getWheelEncodingValues(): WheelEncode
# getPose(): Pose
# getCameraImage(): CameraImage
# setMotorSpeeds(Speed speed)
+ enableAllSensors()
+ enableAllProximitySensors()
+ enableProximitySensors(int[] sensorIDs)
+ enableAllLightSensors()
+ enableLightSensors(int[] sensorIDs)
+ enableAccelerometer()
+ enableWheelEncoding()
+ enablePose()
+ enableCamera()
+ isConnected(): boolean
+ getMotorSpeed(): Speed
+ getMaxVelocity()
+ registerSensorObserver(SensorObserver observer)
+ unregisterSensorObserver(SensorObserver observer)
# notifySensorObservers(LinkedList<Sensor> sensors)
+ registerCameraImageObserver(CameraImageObserver observer)
+ unregisterCameraImageObserver(CameraImageObserver)
# notifyCameraImageObservers(CameraImage cameraImage)
0...1
0...*
0...1
0...*
1
1
<<interface>>
SensorObserver
+ sensorValuesChanged(LinkedList<Sensors> sensors)
<<interface>>
CameraImageObserver
+ cameraImageChanged(CameraImage cameraImage)
Speed
- left: double
- right: double
+ getLeft(): double
+ getRight(): double
<<enum>>
Sensor
ProximitySensor
LightSensor
Accelerometer
WheelEncoding
Pose
Ground

Figure 4.2.: Abstract class DifferentialWheels of the Java framework.

information of the array accessible. Each of the methods accesses the corresponding index of the array and returns the value of it. The class *Speed* is a very simple class which is used to set the motor speed and also to return the current speed. The Python framework uses an array for this. Using an array is inconvenient as you always need to remember which index represents which side of the robot. The Speed class implements two methods for this purpose, each side has its own method.

Handling of the camera image is solved by creating a class which defines the camera image. The approach, of creating an own class, is chosen to keep the data type as basic as possible. A basic data type gives developer the opportunity work with the image processing framework of his choice. Therefore, *CameraImage* stores the image as *BufferedImage* which is a standard Java data type. *CameraImage* has methods to get a specific pixel, set the color value of a specific pixel and a method to get the image as a BufferedImage. The class *CameraImagePixel* holds the RGB value of a specific pixel. It is used to return the pixel color information. *BufferedImage* returns the color value of a pixel only as an integer. From the integer value the values for red, green and blue then need to be retrieved by performing bit operations. Setting a pixel of a *BufferedImage* is only possible with an integer value. The method *setPixel* takes the values for red, green and blue and creates this integer value and sets the pixel color. Code 4.1 shows how the conversion from the separate color values to an integer value is done within the *setPixel* method.

```java
/**
* Set a pixel of the image
* @param x x position of the pixel
* @param y y position of the pixel
* @param r Red value of the pixel
* @param g Green value of the pixel
* @param b Blue value of the pixel
*/
public void setPixel(int x, int y, int r, int g, int b){
     int rgb = 0;
    //Set red value
    //Red value from bit 16 till 23
    if(r > 255){
      //Shift 16 bits to the left
      rgb = rgb | (255<<16);
    } else if(r > 0){
      rgb = rgb | (r<<16);
    }

    //Set green value
    //green value from bit 8 till 15
    if(g > 255){
      //Shift 8 bits to the left
      rgb = rgb | (255<<8);
    } else if(g > 0){
      rgb = rgb | (g<<8);
    }

    //Set blue value
    //blue value from bit 0 till 7
    if(b > 255){
      rgb = rgb | 255;
    } else if(b > 0){
      rgb = rgb | b;
    }

    _image.setRGB(x,y,rgb);
  }
```

Code 4.1.: Method setPixel of CameraImage

Figure 4.3 shows what the binary operations in Code 4.1 do. Using bit operations the red, green and blue value of *rgb* are set. The bits for the red value are from bit 16 to 23, green from bit 8 to 15 and blue from 0 to 7. To set the red value, first a shift of 16 bits to the left is performed on either the *r* value or 255. With *«* from the right *0* are added, for red 16. For green a shift of 8 bits is needed and blue does not need a shift.

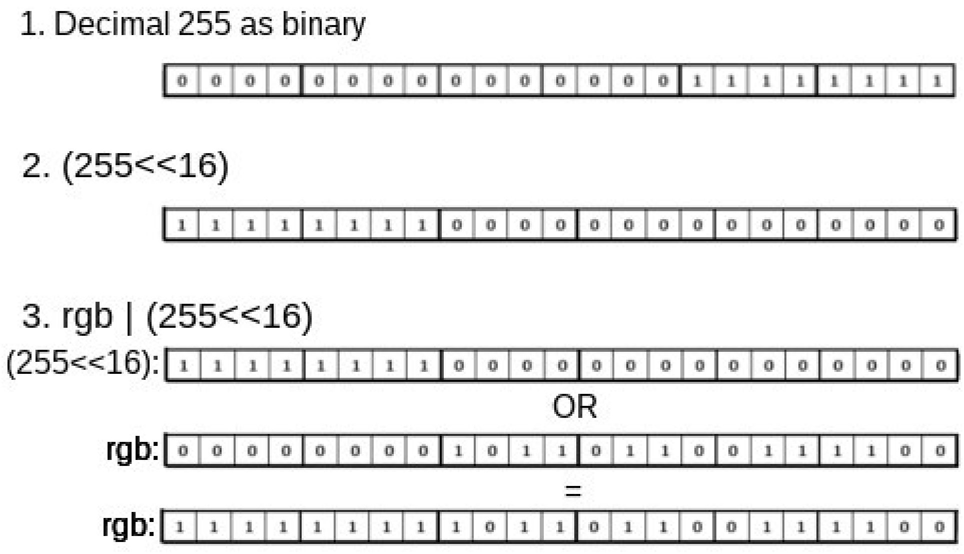


Figure 4.3.: Binary operation with shifting and linking.

The resulting value is then *OR* linked with *rgb*. As a result the value of *r* is now stored in *rgb* from bit 16 to 23.

When a pixel is requested by calling the method *getPixel*, first the integer value is retrieved from the BufferedImage and this integer value is then passed to the constructor of *CameraImagePixel*. Inside the constructor the integer value of red, green and blue value are extracted and saved separately. Code 4.2 shows the constructor of *CameraImagePixel*.

```
/**
* Constructs new camera image pixel from the given rgb int value.
* @param rgb Integer value from a BufferedImage with
* BufferedImage.TYPE_INT_RGB
*/
public CameraImagePixel(int rgb){
 _r= (rgb>>16)&255;
 _g= (rgb>>8)&255;
 _b= (rgb)&255;
}
```

Code 4.2.: Constructor of CameraImagePixel

The class *DifferentialWheels* has the possibility to be observed. The interfaces *SensorObserver* and *CameraImageObserver* define what other classes need to implement to be able to register as an observer. Observers get the information on

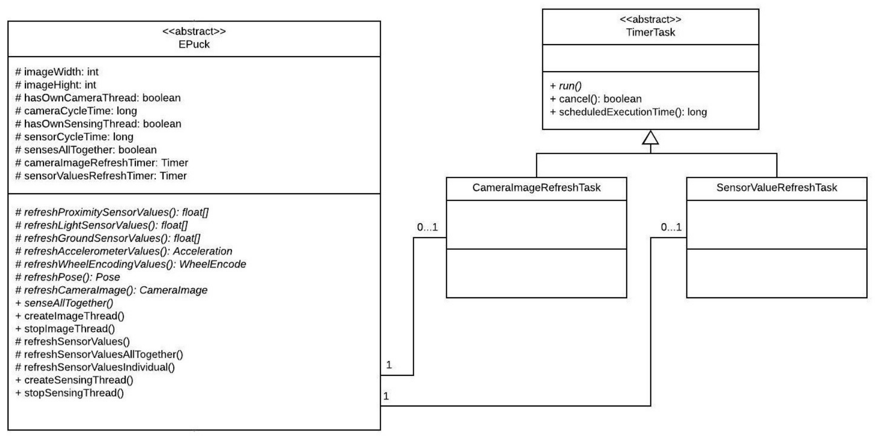

<<abstract>>
EPuck
# imageWidth: int
# imageHight: int
# hasOwnCameraThread: boolean
# cameraCycleTime: long
# hasOwnSensingThread: boolean
# sensorCycleTime: long
# sensesAllTogether: boolean
# cameraImageRefreshTimer: Timer
# sensorValuesRefreshTimer: Timer
# refreshProximitySensorValues(): float[]
# refreshLightSensorValues(): float[]
# refreshGroundSensorValues(): float[]
# refreshAccelerometerValues(): Acceleration
# refreshWheelEncodingValues(): WheelEncode
# refreshPose(): Pose
# refreshCameraImage(): CameraImage
+ senseAllTogether()
+ createImageThread()
+ stopImageThread()
# refreshSensorValues()
# refreshSensorValuesAllTogether()
# refreshSensorValuesIndividual()
+ createSensingThread()
+ stopSensingThread()
<<abstract>>
TimerTask
+ run()
+ cancel(): boolean
+ scheduledExecutionTime(): long
CameraImageRefreshTask
SensorValueRefreshTask
0...1
0...1
1
1

Figure 4.4.: Abstract class EPuck of the Java framework.

which sensor values have changed or get the latest camera image. There is an enum defining all the sensors to let the observer know which sensors have changed. The enum makes it easy to check if the sensor value of interest has changed. *DifferentialWheels* itself only implements methods where no information about the robot is needed. Implemented are methods to enable sensors and methods which are needed to implement the observer pattern. The website Tutorialspoint says that the observer pattern is used, when there are multiple objects (observers) that require information if something changed about one object (observable) [37]. In case of a modification all the objects are then notified. The class *DifferentialWheels* is in this case the observable and a controller could be the observer. Code A.1 and Code A.2 show the methods of *DifferentialWheels* to allow to be observed.

4.1.2. EPuck

The abstract class *EPuck* describes the robot and inherits from the abstract class *DifferentialWheels*. Figure 4.4 shows the class *EPuck* in detail. The class defines abstract methods for refreshing the sensor values and camera image. The methods are abstract as the refreshing of the sensor values and camera image is different for the real e-Puck and the e-Puck in a simulator. Implemented methods are those that return the sensor values to, for example, a controller and those for the threading. *EPuck* implements methods to update the sensor values and the camera image within a thread.

The use of threads simplifies a controller, because it is only necessary to start the thread at the beginning. After the thread is started, the only call, for the sensor values and camera image, is the call of the method which returns the value. Code 4.3 shows the method to return the proximity sensor values. It is important for all the methods, that the returned sensor values are synchronised. Not only accessing needs to be done synchronized, synchronization is also important for writing to the variable. This is important when the values are updated by using the thread. Using a thread, it can happen that the controller requests the sensor value while the thread writes the new values. This is especially important for the motor values. This is also mentioned in Section 4.1.3.4 which covers setting of the motor speed as this is specific for the real e-Puck and virtual e-Puck.

```java
@Override
public double[] getProximitySensorValues() throws
RobotFunctionCallException, SensorNotEnabledException {
    synchronized (_proximitySensorValues) {
    if (!_hasOwnSensingThread && !_senseAllTogetherEnabled) {
        _proximitySensorValues = refreshProximitySensorValues();
    }

    return _proximitySensorValues;
  }
}
```

Code 4.3.: Method of the class EPuck to return the current proximity sensor values

The threads for updating the sensor values and camera image are implemented by using *Timer* of *java.util* [35]. *Timer* can be used to schedule tasks which then are executed in a background thread. A scheduled task needs to be inherited from *TimerTask* [36]. *Timer* offers multiple ways of scheduling. One of the possible ways of scheduling is fixed-delay execution where the task is repeated periodically with a delay. The chosen delay for the task is 0ms as shown in Code 4.4 for the method which creates the image refreshing thread. The thread to refresh the sensor values, is created in the same way.

```java
public void createImageThread() {
  if (!_hasOwnCameraThread) {
      _cameraImageRefreshTimer.schedule(new CameraImageRefreshTask(this),
      0, _cameraCycleTime);
      _hasOwnCameraThread = true;
 }
}
```

Code 4.4.: Method to create the thread which refreshes the camera image.

```
/**
 * This tasks refreshes the camera image.
 */
public class CameraImageRefreshTask extends TimerTask {

private EPuck _ePuck;

/**
* Constructs new task
* @param ePuck Instance of the ePuck of which
* the camera image needs to refreshed
*/
public CameraImageRefreshTask(EPuck ePuck){
  _ePuck = ePuck;
}

@Override
public void run() {
  try {
    CameraImage img = _ePuck.refreshCameraImage();
    _ePuck.setCameraImage(img);
  } catch (CameraNotEnabledException e) {
    e.printStackTrace();
  } catch (RobotFunctionCallException e){
    e.printStackTrace();
  }
 }
}
```

Code 4.5.: Class CameraImageRefreshTask

The classes *CameraImageRefreshTask* and *SensorValueRefreshTask* are two very simple classes. Both inherit from *TimerTask* and implement the method *run()* which is defined by the abstract class *TimerTask*. Code 4.5 shows the class *CameraImageRefreshTask*. When the timer is started it periodically calls *run()*. Inside this method the latest camera image is requested and then saved. For the class *SensorValueRefreshTask* the method *run()* calls the method *refreshSensorValues()* of the class *EPuck*, as shown in Code 4.6.

```
@Override
public void run() {
  _ePuck.refreshSensorValues();
}
```

Code 4.6.: Method *run()* of the class SensorValueRefreshTask

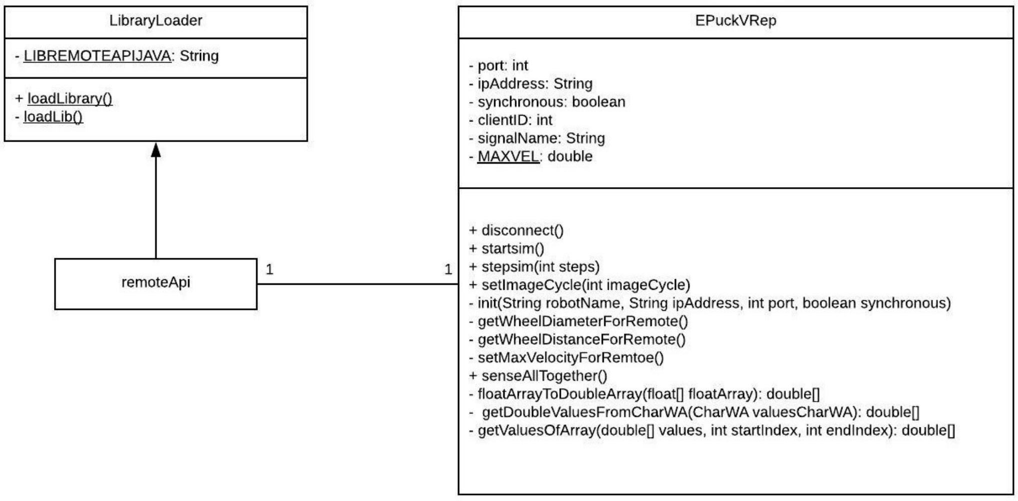

LibraryLoader
- LIBREMOTEAPIJAVA: String
+ loadLibrary()
- loadLib()
remoteApi
1
1
EPuckVRep
- port: int
- ipAddress: String
- synchronous: boolean
- clientID: int
- signalName: String
- MAXVEL: double
+ disconnect()
+ startsim()
+ stepsim(int steps)
+ setImageCycle(int imageCycle)
- init(String robotName, String ipAddress, int port, boolean synchronous)
- getWheelDiameterForRemote()
- getWheelDistanceForRemote()
- setMaxVelocityForRemtoe()
+ senseAllTogether()
- floatArrayToDoubleArray(float[] floatArray): double[]
- getDoubleValuesFromCharWA(CharWA valuesCharWA): double[]
- getValuesOfArray(double[] values, int startIndex, int endIndex): double[]

Figure 4.5.: Class EPuckVRep of the Java framework.

4.1.3. EPuckVRep

Like in the Python framework the class *EPuckVRep* is a proxy of the virtual robot e-Puck of V-REP for the controller. The implemented functionality is the same as it is in the Python framework. Methods are provided to get the sensor values, get the camera image and set the motor speed. The possibility of running the simulation in synchronous mode, as it is explained in Section 2.4.3, is also implemented. Special to this class is the integrated loading of the library. The automatic loading of the library greatly simplifies the use of the remote API. Section 4.1.3.1 covers the library loading in detail.

4.1.3.1. Loading the remote API library

As shown in Figure 4.5 the class *EPuckVRep* uses the remote API library. To make the framework easier to use, the *LibraryLoader* was developed. The *LibraryLoader* checks which operating system the application is running on and loads the correct library file. As shown in the class diagram in Figure 4.5 the *LibraryLoader* consists of two static methods. The public method *loadLibrary* calls the private method *loadLib* based on specific operating system (OS) the application is running on. Code B.1 shows the method *loadLibrary*. If a supported OS is detected, the file name of the library is constructed and passed on to the *loadLib* method. The detection of the OS is done with the class *OSValidator*.

OSValidator uses environment variables to detect the OS type and architecture the application is running on. It has two methods to detect Windows, one method to detect MacOSX and three methods to detect Linux systems. Code 4.7 shows the variables for the OS and the architecture type. The variable *OS* stores the OS type and the variables *windowsArch* and *windowsWow64Arch* are used to detect if it is a 64bit Windows OS or a 32bit Windows OS.

```java
private static String OS = System.getProperty("os.name").toLowerCase();

private static String windowsArch =
                              System.getenv("PROCESSOR_ARCHITECTURE");

private static String windowsWow64Arch =
                              System.getenv("PROCESSOR_ARCHITEW6432");
```

Code 4.7.: Variables of the OSValidator for the architecture type and OS type.

To define the OS and architecture the strings are tested on certain criteria. Code B.2 shows how the Windows OS and the architecture is detected. The application is running on a Windows OS if the value of the variable *OS* starts with *win*. To detect if it is a 32bit or a 64bit system the variables *windowsArch* and *windowsWow64Arch* are tested if one of those ends with *64*.

A Linux system can be detected by checking the value, of variable *OS*, to end with *nix*, *nux*, *aix* or *sunos* as shown in code B.3. If the Linux OS is a 32bit or 64bit can be detected by checking the environment variable *os.arch*. If this variable ends with *64* the OS is a 64bit OS. The MacOS is detected by checking if the value of the variable *OS* contains *mac*, as it is shown in Code B.4.

After detecting the OS, the correct library is loaded from the resources of the framework. The library is then copied into a folder as a temporary file which will be deleted when the application closes. After the copy process is finished the library is loaded so it can be used by the application. The method *loadLib*, shown in Code 4.8 does the loading from the resources, copying and loading.

```java
/**
* Puts library to temp dir and loads to memory
*/
private static void loadLib(String path, String fileExtension) {
   try {
        //Create a stream containing the library which needs to be written
        //to the drive
        InputStream in = LibraryLoader.class.getResourceAsStream(path);
        //Path to the folder with lib
        String libFolder = System.getProperty("java.io.tmpdir")
                          +File.separator+"vrep";
        //Create the folder for lib file
        new File(libFolder).mkdirs();
        //Write to a tmp file so that multiple instances can be run
        File fileOut = File.createTempFile(LIBREMOTEAPIJAVA,"."
                     + fileExtension,new File(libFolder));

        //Define that this temporary file needs to be deleted when the
        //applications exits
        fileOut.deleteOnExit();
        System.out.println("Writing library to: "
                          + fileOut.getAbsolutePath());

        //Create a output stream to temporary file and start copying
        OutputStream out = FileUtils.openOutputStream(fileOut);
        IOUtils.copy(in, out);

        //Close all streams
        in.close();
        out.close();
        try {
             //Load the library
             System.load(fileOut.toString());
        } catch (Exception ex){
             System.out.println(ex);
             System.exit(0);
        }
    } catch (Exception e) {
       System.out.println("Failed to load required library \n"
                          + e.getMessage());
    }
}
```

Code 4.8.: Method loadLib() of the class LibraryLoader.

To automatically load the correct library when a new object of the remote API is instantiated, a small change needed to be made in the remoteAPI.java class. The remoteAPI.java class contains a static area where the library is loaded. The method gets replaced with the loadLibrary method of LibraryLoader. Code B.5 shows the old type of loading the library and code B.6 shows the new version using the LibraryLoader. In the old version with *System.loadLibrary("remoteApiJava")* on the system is searched for the library which needed to be registered manually beforehand.

4.1.3.2. Reading the sensor values

The sensor values can be requested in different ways. Either each sensor value can be retrieved on its own or all the sensor values are requested at once. For this purpose the remote API methods *simxGetStringSignal* and *simxCallScriptFunction* are used. Both of the remote API methods call the Lua script methods of the model in the simulator. In both cases the method is called as a blocking call such that the execution continues when the results are received. Code 4.9 shows part of the method *senseAllTogether*. When calling the remote API method an ID, in this example stored in the variable *_clientID*, needs to be provided. This ID is returned when the connection to the simulator gets established by using the remote API method *simxStart*. This method needs to be called once before starting the communication. With *_signalName+"_allSens"* the model and the method of its Lua script is defined. The variable *_signalName* is constructed from the model name, in this case *epuck*, and the port number on which the model is mapped to in the simulator V-REP. All the remote API methods return a integer error code value. By checking the integer value *returnCode*, it is possible to determine if the call was successful. To return the result values, the remote API makes use of call by reference for objects. The method *simxGetStringSignal* needs an empty char array object as parameter. The object is defined by V-REP and contains a char array and some other methods to modify the char array. When the call was successful, the object contains the return value of the Lua script method called. It is important for the remote API calls to make sure that only one call at a time is made. Therefore, the Java method *synchronized* with the object *lockAPI* is used for every remote API call. From the returned array then the proximity sensor values, light sensor values, ground sensor values, accelerometer values and wheel encoder values can be extracted and stored.

```java
//Create the array for the return values
CharWA inCharWA = new CharWA(1);
int returnCode = 0;
synchronized (lockAPI) {
returnCode = _vrepRemote.simxGetStringSignal(_clientID, _signalName +
 "_allSens", inCharWA, _vrepRemote.simx_opmode_buffer);
}
```

Code 4.9.: Requesting all sensor values with one remote API call in Java

When the values of a single sensor type are requested, the remote API method *simxCallScriptFuntion* is used. This method calls specific function of the Lua script from the robot model. Code 4.10 shows part of the method *refreshProximitySensorValues* for requesting the proximity sensor values. The *_clientID* is again the ID returned from establishing the connection and the *_robotName* is the name of the robot model in the simulator. All the models in a simulation have a unique name by which they can be identified. Besides defining the type of Lua scripted called and the definition that the call should act blocking, the function of the Lua script needs to be specified and a return object provided. The return object provided needs to match with the defined return value of the Lua script function. If the return code indicates that the call was successful the value of the return object can be stored.

```java
FloatWA outFloat = new FloatWA(_numProximitySeonsors);
int returnCode = 0;
synchronized (lockAPI) {
 returnCode = _vrepRemote.simxCallScriptFunction(_clientID, _robotName,
 _vrepRemote.sim_scripttype_childscript, "getProxSensorsForRemote", null,
 null, null, null, null, outFloat, null, null,
 _vrepRemote.simx_opmode_blocking);
}
```

Code 4.10.: Example of the values of one sensor type are requested from the remote API in Java.

4.1.3.3. Getting the camera image

The camera image is also requested with the remote API method *simxCallScriptFunction* as described in Section 4.1.3.2. For the camera image the Lua script function *getCameraSensorsForRemote* is called. This function returns a float array of pixel colors. One pixel of the image uses three entries in the array as every color value has its own entry. Therefore, the size of this array is *image-width*image-hight*3*. To properly store the image some additional calculation is necessary. Code 4.11 shows how the data of the array is converted such that it matches the image formate of the framework. First a new image object is created with the correct size. Then the RGB value for a pixel is determined and stored in the image object at the correct position. Since the array consists of a concatenation of pixel rows the position of a pixel in the array and its RGB values can be easily calculated. The start of a new pixel row in an array is calculated with *y*_imageWidth*. A specific pixel of a row can be select by adding the x value so the calculation is *y*_imageWidth+x*. Because a pixel uses three entries in the array now, the value needs to be multiplied with 3 and the new calculation looks as like, *3*(y*_imageWidth+x)*. To get the corresponding color value, the offset needs to be added. For example, the index in the array for the color red of a pixel can then be calculated with

3(y*_imageWidth+x)+0*. Since the values returned are between 0 and 1, a multiplication by 255 is needed to map it correctly. The framework also corrects the orientation of the image when storing it. Originally the image returned is up-side-down, to make the developer's life easier the image is stored rotated by 180°.

```
//Create a new image object and set each pixel
CameraImage tmpImg = new CameraImage(_imageWidth, _imageHeight);
for (int y = 0; y < _imageHeight; y++) {
 for (int x = 0; x < _imageWidth; x++) {
 //The values from V-REP are between 0 and 1 so it is necessary to multiply
 //with 255 to get the correct value
 int r = (int) (rgbFloatValues[3 * (y * _imageWidth + x) + 0] * 255);
 int g = (int) (rgbFloatValues[3 * (y * _imageWidth + x) + 1] * 255);
 int b = (int) (rgbFloatValues[3 * (y * _imageWidth + x) + 2] * 255);
 //Set the pixel with the corresponding colour. The image returned from
 //V-REP is up-side-down so and with "(_imageHeight - 1) - y" this is
 //corrected so the image has the correct orientation.
 tmpImg.setPixel(x, (_imageHeight - 1) - y, r, g, b);
 }
}
```

Code 4.11.: Converting the array representation of the image to an image object.

4.1.3.4. Setting the motor speed

Setting the motor speed is done by using the remote API method *simxCallScriptFunction*. The name of the Lua script function called is *setVelocitiesForRemote*. Code 4.12 shows part of the framework method to set the motor speed. As the speed values for the motors are expected as a *FloatWA*, which is defined by the remote API, an object containing these values has to be created. The value for the left motor has to be stored at index one and the value for the right motor at index two. By checking the return code it is possible to determine if the call was successful. There is no other return value by the function *setVelocitiesForRemote*. For this method there are two reasons why the synchronisation is needed. The first reason is the requirement of the remoteAPI, as it was mentioned earlier. More important is, to avoid that multiple threads set the motor speed at the same time. Only on thread should be allowed to set the motor speed at a time. If every thread would set the motor speed at the same time, it result in unwanted behaviour of the robot since some values might never get processed by the robot.

```
//setVelocitiesForRemote float[] --> [0]= left , [1] = right
FloatWA speedFloatWA = new FloatWA(2);
float[] speedFloats = speedFloatWA.getArray();
speedFloats[0] = (float) speed.getLeft();
speedFloats[1] = (float) speed.getRight();

int returnCode = 0;
synchronized (lockAPI) {
 //Send command to VRep
 returnCode = _vrepRemote.simxCallScriptFunction(_clientID, _robotName,
 _vrepRemote.sim_scripttype_childscript, "setVelocitiesForRemote", null,
 speedFloatWA, null, null, null, null, null, null,
 vrepRemote.simx_opmode_blocking);
}
```

Code 4.12.: Setting the motor speed of the robot.

4.2. Example application

The example application is used to test the functionality and usability of the developed framework. First the problem is explained, followed by the analysis of the controller provided by V-REP and then how the problem was solved.

4.2.1. Problem

The scenario for the example application, which is shown in Figure 1.2, consists of two robots with the task to follow the line. Those to robots in the scenario are driving towards each other. At one point the two robots will meet on their path, have to pass each other and continue following the line. Besides passing each other they also have to avoid two obstacles and continue following the line. V-REP provides a controller written in Lua for this scene. However, the provided controller is not working properly. Following the line and avoiding the fixed obstacles works without any problems. Avoiding a moving obstacle, in this case the other robot, is not working. When the two robots meet and detect each other, they rotate in the same direction and continue driving parallel like it is shown in Figure 4.6. The existing controller shall be analysed to find the cause of the unexpected parallel driving and then implement the improved version of the controller in Java.

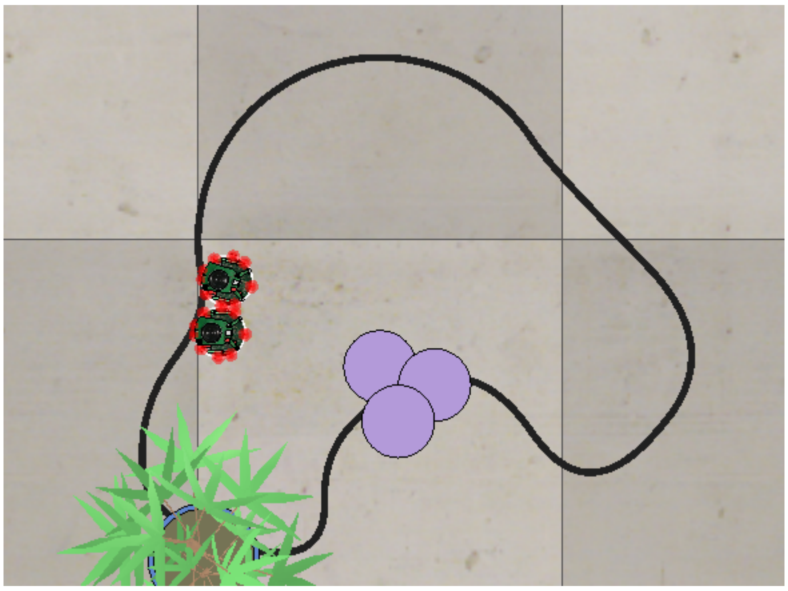


Figure 4.6.: With the default controller of V-REP in the example application the robots turn to each other and drive in parallel.

4.2.2. Analysing the existing controller

The controller that comes with the scene distinguishes three cases for the robot. Code 4.13 shows the controller as pseudo code. The first case (code line 7 till 18) is that the ground sensors are read, a line detected and no obstacle is detected in front of the robot. To detect an obstacle in front of the robot the proximity sensors 1,2,3 and 4, shown in Figure 4.7, are used. The second case (code line 21 till 34) gets activated when there are no obstacles in front of the robot. If there is no obstacle in front, the sides of the robots are checked for obstacles (code line 34 till 40). To check if an obstacle is on the side of the robot the sensors 1 and 6 are used. If the first two case do not get activated the robot has an obstacle in front and avoids it. The Braitenberg vehicle concept is used for avoiding the obstacle in front and also to keep a safe distance to the obstacle to the left or right. A Breitenberg vehicle uses a sensor to motor connection. This simple connection leads to a seemingly cognitive behaviour, as it is explained in the course materials of the University of Sussex [40]. Keeping a constant distance to the object to the left or right works flawlessly. Avoiding the obstacle in front of the robot does not work for every scenario. If all the sensors, used to detect the obstacle in front, deliver approximately the same value, the robot will not move around the obstacle. The robot will get stuck, it is moving but rather driving into the obstacle then trying to avoid it. This is happening as the applied Braitenberg accelerates the wheels on the side where the obstacle is detected and lets the opposite wheels run reverse. The robot detects an obstacle on the right side (sensor 3 and 4) and then tries to move to the left. At the same time an obstacle is detected on the left (sensor 1 and 2) and the robot tries to drive to the right side. This Braitenberg behaviour is called *fear*. The applied Braitenberg behaviour also explains the situation

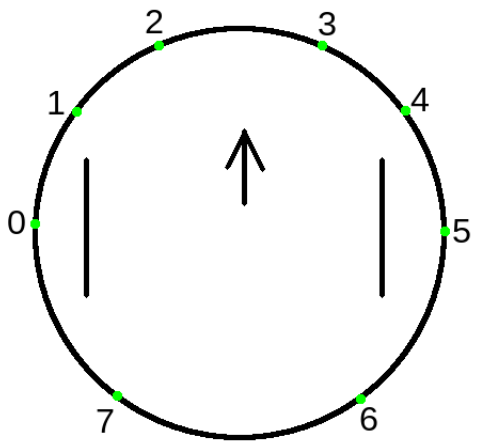


Figure 4.7.: Position of the proximity sensors on the robot e-Puck.

shown in Figure 4.6 occurs. To make the two robots pass each other, a new solution needs to be found for the detection and avoidance of an obstacle in the front. The solution is discussed in Section 4.2.3.

```
//Braitenberg weights for the 4 front prox sensors (avoidance):
int[] braitFrontSens_Motor = new int[]{1,2,-2,-1}
//Braitenberg weights for the 2 side prox sensors (following):
int[] braitSideSens_Motor = new int[]{-1,0}
double speedLeft = 0
double speedRight = 0
if(groundSensorValuesUpdated && lineDetected && !obstacleInFrontOfRobot){
  if(groundSens[0] > 0.5){
    speedLeft = maxSpeed
  } else {
    speedLeft = maxSpeed * 0.25
  }
  if(groundSens[2] > 0.5){
    speedRight = maxSpeed
  } else {
    speedRight = maxSpeed * 0.25
  }
} else {
  speedLeft = maxSpeed
  speedRight = maxSpeed
  if(!obstacleInFrontOfRobot){
    if(proxSensDist[0]>(0.25 * 0.05)){
     speedLeft = speedLeft+maxSpeed*braitSideSens_Motor[0] *
             (1-proxSensDist[0]/0.05))
     speedRight = speedRight+maxSpeed*braitSideSens_Motor[1] *
             (1-proxSensDist[0]/0.05)
    }
    if(proxSensDist[5]>(0.25 * 0.05)){
     speedLeft = speedLeft+maxSpeed*braitSideSens_Motor[1] *
             (1-proxSensDist[5]/0.05)
     speedRight = speedRight+maxSpeed*braitSideSens_Motor[0] *
              (1-proxSensDist[5]/0.05)
    }
  } else {
   for(int i=0; i < braitFrontSens_Motor.length; i++){
     speedLeft=speedLeft+maxSpeed*braitFrontSens_Motor[i] *
             (1-proxSensDist[1+i]/0.05)
     speedRight=speedRight+maxSpeed*braitFrontSens_Motor[3-i] *
     (1-proxSensDist[1+i]/0.05)
   }
  }
}
```

Code 4.13.: Controller for the example application provided by V-REP as pseudo-code.

4.2.3. New controller

As the analysis of the existing controller in Section 4.2.2 shows, the detection and avoidance of an object in front of the robot needs to be improved. Using Braitenberg, to drive alongside the obstacle, is a very good solution and therefore should stay. The problem of the current version is, that due to Braitenberg the robots can turn in the same direction. So the first robot detects the other robot to its left (sensor 1 and 2 of Figure 4.7) and starts turning to the right. The second robot detects the other robot to its right (sensor 3 and 4) and starts turning left which leads to the situation shown in Figure 4.6. To perfectly pass each other the robots should turn in different direction. The initial turn of the robot should be always in the same direction, seen from the direction of motion. This can either be a left or right turn. The important point is that the obstacle is always on the same side for Braitenberg. Code 4.14 shows the improved version of the controller as pseudo-code. Line 29 till 39 shows the improved avoidance of an obstacle in front of the robot. If an obstacle is detected by one of the two front sensors (sensor 2 and 3) the robot always does a turn to the left. With this initial turn the Braitenberg avoidance will always behave the same way. Both robots will continue turning to the left until sensor 3 and sensor 4 do not detect any obstacle. Sensor 5 is then used to keep a safe distance to the obstacle while driving (line 22 till 27). Using the sensor 5, both robots will try to drive around the obstacle. As both are moving, this ends up in a rotation which stops as soon as the robots detect a line and start following it. Line 3 till 19 show the new line following algorithm. The section is adapted to make the start of the line following more reliable. The changes are between line 5 and 16. If the middle and right sensor detect a line, a more aggressive right turn is performed then during the normal line following. The same happens if the middle and left sensor detect the line, but now a left turn is made. If all ground sensors detect the line a rotation to the left is done to make sure that the robot is able to get back on the line. With these changes the robots are now able to pass each other, avoid other none moving objects and follow the line.

```
double speedLeft = maxSpeed
double speedRight = maxSpeed
if(groundSensorValuesUpdated && lineDetected && !obstacleInFrontOfRobot){
  // middle and right sensor detect the line => right turn
  if((groundSensorValues[0] > 0.5 && groundSensorValues[1] < 0.5 &&
      groundSensorValues[2] < 0.5)){
      speedRight = -0.75 * maxSpeed
  }//the middle and the left sensor detect the line => left turn
  else if((groundSensorValues[0] < 0.5 && groundSensorValues[1] < 0.5 &&
      groundSensorValues[2] > 0.5)){
      speedLeft = -0.75 * maxSpeed
  }//All sensors detect line => make left turn
  else if(groundSensorValues[0] < 0.5 && groundSensorValues[1] < 0.5 &&
          groundSensorValues[2] < 0.5) {
      speedLeft = -1 * maxSpeed
  }else{
   if (groundSensorValues[0] < 0.5) { speedLeft = maxSpeed * 0.20 }
   if (groundSensorValues[2] < 0.5) { speedRight = maxSpeed * 0.20 }
  }
} else {
  if(!obstacleInFrontOfRobot){
    if(proxSensDist[5]>(0.25 * 0.05)){
     speedLeft = speedLeft+maxSpeed*braitSideSens_Motor[1] *
             (1-proxSensDist[5]/0.05)
     speedRight = speedRight+maxSpeed*braitSideSens_Motor[0] *
              (1-proxSensDist[5]/0.05)
    }
  } else {
   if(proxSensDist[2] > (0.25 * 0.05) ||
      proxSensDist[3] > (0.25 * 0.05)){
     speedRight = maxVelocity
     speedLeft = -1 * maxVelocity / 2
   } else {
    for(int i=0; i < braitFrontSens_Motor.length; i++){
     speedLeft=speedLeft+maxSpeed*braitFrontSens_Motor[i] *
      (1-proxSensDist[1+i]/0.05)
     speedRight=speedRight+maxSpeed*braitFrontSens_Motor[3-i] *
      (1-proxSensDist[1+i]/0.05)
    }
   }
  }
}
```

Code 4.14.: Improved controller for the example application provided by V-REP as pseudo-code.

5. Evaluation

This chapter covers the performance test of the framework, how I experienced the framework while using it for the example application, the evaluation of the framework for using it in the 5$^{\text{th}}$ semester and discusses the V-REP client/server separation.

5.1. Performance of the framework

The performance of the framework is a main factor for the simulation. If the framework has a slow performance it can lead to an imprecise simulation when the simulator is running in asynchronous mode. If the simulator is running in synchronous mode it will be always precise. In the synchronous mode the simulator gets told how many simulations steps it should make. After the simulator has done the amount of steps it waits until it gets the instruction to perform the next simulation steps. This part covers how performance testing is defined, how the performance of the framework was tested and what the results are.

5.1.1. Definition of performance testing

On TechTarget it says, "Performance testing is the process of determining the speed or effectiveness of a computer, network, software program or device. This process can involve quantitative tests done in a lab, such as measuring the response time or the number of MIPS (millions of instructions per second) at which a system functions. Qualitative attributes such as reliability, scalability and interoperability may also be evaluated. Performance testing is often done in conjunction with stress testing". "Performance testing can also be used as a diagnostic aid in locating communications bottlenecks. Often a system will work much better if a problem is resolved at a single point or in a single component. For example, even the fastest computer will function poorly on today's Web if the connection occurs at only 40 to 50 Kbps (kilobits per second)." [32].

Software Testing Fundamentals defines performance testing as followed: "Performance Testing is a type of software testing that intends to determine how a system performs in terms of responsiveness and stability under a certain load." [24].

"Performance testing, a non-functional testing technique performed to determine the system parameters in terms of responsiveness and stability under various workload. Performance testing measures the quality attributes of the system, such as scalability, reliability and resource usage", is the definition by TutorialsPoint [39].

Testing Performance defines performance testing as, "Performance Testing is associated with a number of interchangeable names. The performance test can also referred to as a stress test, load testing or volume testing and is the application of a process that verifies the ability of a system to handle varying degrees of concurrent users and system transactions. The Goals of performance testing are driven by a number of factors that could include business volumetric requirements and service level agreements (SLA) as well as perceived and actual performance risk." [34].

For this thesis the performance testing is determining how long methods take to return or set a value. The shorter the time passed between calling a method to get or set a value the better the performance is.

5.1.2. Performance testing the framework

For the framework it is important that the method called which refresh or set a value using the remote API, take as little time as possible. The time a method call takes is crucial for how precise the robot can be controlled. Therefore, the testing shall give an overview of how long the method calls take and also reveal where the bottlenecks are, that can cause problems during controlling the robot. Following scenarios were tested:

- Test 01 - Requesting a sensor value by directly addressing the sensor
- Test 02 - Requesting a sensor value by getting all sensor values with one call
- Test 03 - Requesting two sensor values by directly addressing each sensor
- Test 04 - Requesting two sensor values by getting all sensor values with one call
- Test 05 - Requesting all sensor values with one call by using a thread
- Test 06 - Requesting the camera image and sensor values (all sensor values at once) without threads
- Test 07 - Requesting the camera image by using a thread
- Test 08 - Only using threads to get sensor values and camera image

For profiling there are several software available on the market. Most of these require a license but some also offer a period of trial use. Such a software, which requires a license, is JProfiler [30]. This tool gives detail information on how long a method execution takes. It also has a plugin for Intellij which makes the profiling easy. The tool was used with the trial license that gives access for 10 days.

Java VisualVM is a profiling tool that comes with the Java framework and is free to use. The tool shows how much time was spent in a method in total and how often the method was called. The missing information the calls of the API methods. These do not show up and therefore the use of this tool was stopped during the testing.

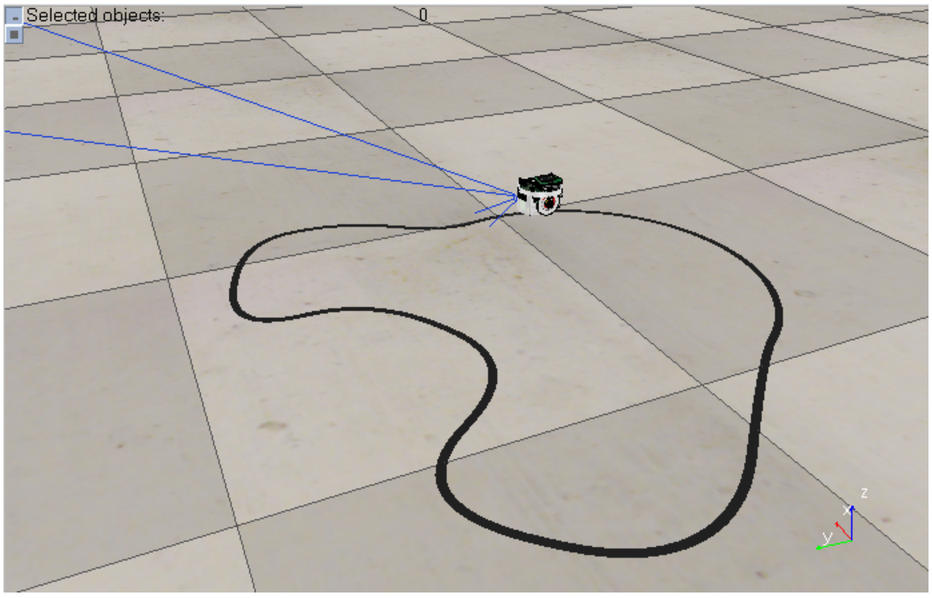


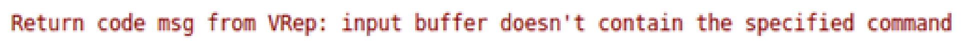
Figure 5.1.: Scene for performance test.

The main interest was on how long the execution of a method took, the use of the Java method *System.currentTimeMillis()* can be used to determine the execution time [20]. The method returns the time difference between the 1st January 1970 00:00 UTC and the current time. The time difference is given in milliseconds. By calling this method before and after a method it is possible to determine how long the execution took.

To determine the performance the tool JProfiler and the Java method *System.currentTimeMillis()* have been used. The scene for the test contains a line which the robot follows, as shown in Figure 5.1. During testing the focus was on the execution time of the methods which get the data from the simulator and if the robot follows the line well. The chosen simulation mode in V-REP was the real-time mode. In the real-time mode one second inside the simulator is also one second in the real world. There are other modes were the simulation is adapted according to the system the simulator is running on. These other modes try to keep the frame rate high to have a smooth simulation but the time inside the simulator maybe pass faster than the time in the real world.

5.1.3. Testing environment

The tests were run on a single hardware with an Intel i7-6700HQ, 16GB memory and a Nvidia GeForce GTX 960M graphics card. V-REP uses the graphics card for the rendering and handling of vision sensors [48]. During testing it was checked on which core the simulator and the Java program were running. For the best test results both programmes should run on different cores. The Java program was running on the same core during a single test while the simulator switched between different cores. If both,



```
Return code msg from VRep: input buffer doesn't contain the specified command
```

Figure 5.2.: Error during testing with threads.

the simulator and the Java program, were to run on the same core it could come to in accurate test results due to high load on one single core, which slows it down.

5.1.4. Problems encountered during testing

During test 05, where the sensor values are updated by using a thread, immediately after the start of the controller it came to a crash with the error message shown in Figure 5.2.

According to this message, the most logical reason is that the API call of the thread overlaps with the call to set the motor speed. As the calls worked for the previous tests, where the calls were made sequentially the problem seems to be the API of the simulator. A possible reason is that the simulator cannot handle multiple requests at the same time for one robot. The company's website however gives no information about such a restriction. The problem was solved by using the synchronized statement of java which was applied to all API method calls [19]. The synchronized statement prevents the execution of a method from multiple threads at the same time so only one thread at any given time can execute the method. Code 4.12 shows how the synchronized statement is applied to the method call which sets the motor speed. The object lockAPI is a simple object and is the intrinsic lock.

While making sure that the synchronization is applied to every API method call, I came across an implementation error in the framework. The methods which return the sensor value to the controller did not have synchronization, which is important when the sensor values are updated in a thread. Using a thread to update the sensor values can result in the thread updating the value while the controller accesses the value. To prevent this situation, synchronization was added to all methods that return or update a sensor value. To see if this change influences the API calls, the added synchronization to all the API calling methods were removed. Running the same test again the previous error did not appear. All the test were then run again with only the synchronization for the methods which return or update the sensor values. During test 07 and 08 the error occurred again. Test 07 uses a thread to update the camera image while the sensor value update is triggered by the controller. The threads only approach is used in test 08. Sensor values and the camera image are updated in separate threads in test 08. After checking the V-REP remote API documentation again, the problem appeared to be the blocking API calls. Because of the threads, there is a possibility that two API calls overlap and the API processes the call that arrived earlier and the other gets the error message shown in Figure 5.2. By adding synchronization to all the API calls the problem was solved.

5.1.5. Test results

- Test 01 - Requesting a sensor value by directly addressing the sensor
  - Simulation run time: ca. 10min
  - Average time to get ground sensor value from API (*simxCallScriptFunction*): 69.73ms
  - Average time to set motor speed (*simxCallScriptFunction*): 70.12ms
  - Average loop rung time: 139.94ms

  Figure C.1 and C.2 show the results of this test.

- Test 02 - Requesting a sensor value by getting all sensor values with one call
  - Simulation run time: ca. 10min
  - Average time to get all sensor values from API (*simxGetStringSignal*): 0.05ms
  - Average time to get ground sensor value: 0.001ms
  - Average time to set motor speed (*simxCallScriptFunction*): 67.89ms
  - Average loop rung time: 67.96ms

  Figure C.3 and C.4 show the results of this test.

- Test 03 - Requesting two sensor values by directly addressing each sensor
  - Simulation run time: ca. 10min
  - Average time to get pose from API (*simxCallScriptFunction*): 69.46ms
  - Average time to get ground sensor values from API (*simxCallScriptFunction*): 67.77ms
  - Average time to set motor speed (*simxCallScriptFunction*): 68.58ms
  - Average loop rung time: 205.86ms

  Figure C.5 and C.6 show the results of this test.

- Test 04 - Requesting two sensor values by getting all sensor values with one call
  - Simulation run time: ca. 10min
  - Average time to get all sensor values from API (*simxGetStringSignal*): 0.05ms
  - Average time to get proximity sensor values: 0.003ms
  - Average time to get ground sensor values: 0.001ms
  - Average time to set motor speed (*simxCallScriptFunction*): 72.65ms
  - Average loop rung time: 72.71ms

  Figure C.7 and C.8 show the results of this test.

- Test 05 - Requesting all sensor values with one call by using a thread

  - Simulation run time: ca. 10min
  - Average time to get all sensor values from API (*simxGetStringSignal*): 0.02ms
  - Average time to get ground sensor values: 0.005ms
  - Average time to set motor speed (*simxCallScriptFunction*): 71.09ms
  - Average loop rung time: 71.11ms

  Figure C.9 and C.10 show the results of this test.

- Test 06 - Requesting the camera image and sensor values (all sensor values at once) without threads
  - Simulation run time: ca. 10min
  - Average time to get camera image from API (*simxCallScriptFunction*): 77.39ms
  - Average time to get all sensor values from API (*simxGetStringSignal*): 0.04ms
  - Average time to get ground sensor values: 0.002ms
  - Average time to set motor speed (*simxCallScriptFunction*): 95.89ms
  - Average loop rung time: 173.34ms

  Figure C.11 and C.12 show the results of this test.

- Test 07 - Requesting the camera image by using a thread
  - Simulation run time: ca. 10min
  - Average time to get camera image from API (*simxCallScriptFunction*): 71.24ms
  - Average time to get all sensor values from API (*simxGetStringSignal*): 0.01ms
  - Average time to get camera image: 0.006ms
  - Average time to get ground sensor values: 0.002ms
  - Average time to set motor speed (*simxCallScriptFunction*): 71.33ms
  - Average loop rung time: 83.19ms

  Figure C.13 and C.14 show the results of this test.

- Test 08 - Only using threads to get sensor values and camera image
  - Simulation run time: ca. 10min
  - Average time to get camera image from API (*simxCallScriptFunction*): 71.46ms
  - Average time to get all sensor values from API (*simxGetStringSignal*): 0.02ms
  - Average time to get camera image: 0.004ms
  - Average time to get ground sensor values: 0.001ms
  - Average time to set motor speed (*simxCallScriptFunction*): 83.36ms
  - Average loop rung time: 83.37ms

5.1.6. Analysing the test results

The tests show that there is a big difference in the performance between the remote API methods *simxCallScriptFunction* and *simxGetStringSignal*. Using the remote API method *simxCallScriptFuntion* to get or set a value will take about 72ms for each call. A remote API call using the method *simxGetStringSignal* in comparison will take about 0.02ms. For controlling a robot this makes a difference in terms how precise the robot can be controlled. The website of V-REP does not give any information on why there is such a big difference in the execution time for these methods. To make the methods, which update a sensor value individually, usable without drawbacks in terms of the response time, they need to be changed to *simxGetStringSignal*. This change has not been done yet and is mention in the Chapter 6.

Using threads to get the values from the remote API makes the control loop in the controller faster, as the controller does not have to wait for the response of the API. The only limiting factor in a controller using threads is the *setMotorSpeed* since this method uses *simxCallScriptFuntion* to set the speed.

Based on this test a controller has the best performance if threads are used for the camera image and the sensor values. The sensor values should be requested all in one call using *senseAllTogether*.

5.2. Own experience

For the example application I used the framework like the students in the 5th semester would use it. A feature I really liked during the development process was the step simulation. It helped me to learn what value range the different sensors return. Being able to debug the controller and having the simulation stopped makes a precise analysis of the controller possible. This is the feature that I would have liked for building my first controllers. To update the values I used the method which updates all sensor values in one call. This is the most logic approach, in my opinion, as all values should represent the same time frame to be able to make decisions. The automatically loading of the remote API library worked without any problem. This feature of the framework removes a potential error source while developing.

5.3. Evaluation for the 5th semester

Comparing the framework of Webots with this framework I cannot see differences which could cause troubles for 5th semester students. The developed framework features all the functionality that is available in Python. In the Masters program the Python framework is already successfully in use. The only problem that occurred often in the Masters program was that the remote API library was not loaded. This problem is solved with the Java framework and the students do not need to worry about it and can fully focus on the development of the controller. With the example application and the framework documentation as JavaDoc the students can check how to use the framework. The example application illustrates how the framework is used and with the JavaDoc they can check the methods of the classes.

5.4. Client / Server separation

Comparing the process of simulating between V-REP and Webots I did not recognize any disadvantages. For me the development was easier with V-REP then it was with Webots. With Webots I used to develop the controller with Intellij and then had to copy the output files to another directory. I switched to another integrated development environment (IDE) to develop the controller because the controller editor in Webots was very minimalistic. With V-REP you just start the simulation in V-REP and after that you run the controller from the IDE of your choice. The separation of client and server also gives new opportunities during the lectures. For example presenting the exercises during the lecture gets easier. With Webots every student needs to connect their notebook to the projector, which time consuming. With V-REP the lecturer can start the simulator on his notebook. They then give the students the IP-address of their machine so they can set-up the connection to the remote API. Each student can then run the controller on their machine and control the robot in the simulator running on the lecturer's notebook. With this set-up it is also possible to make competitions. By adding robots to the simulation of the lecturer multiple students can run their controllers at the same time.

6. Future work

The current version of the framework only allows controlling the virtual robot of V-REP. Future work can extend the framework with the possibility to control the real version of the robot e-Puck. Furthermore, the Lua script of the virtual e-Puck can be modified. The script has to be changed to enable the use of the remote API method *simxGetStringSignal* for all calls.

Bibliography

[1] Aurélien. *This code provide tools for command and get status E-Puck robot in V-Rep simulation tool.: AurelienC/vrep-java-epuck-interface.* original-date: 2017-05-03T20:58:19Z. May 3, 2017. URL: `https://github.com/AurelienC/vrep-java-epuck-interface` (visited on 09/20/2018).

[2] BlueZero. *Middleware for distributed applications. Contribute to blueworkforce/bluezero development by creating an account on GitHub.* original-date: 2017-10-05T07:44:08Z. Aug. 21, 2018. URL: `https://github.com/blueworkforce/bluezero` (visited on 08/30/2018).

[3] cambridge oop definition. *object-oriented Meaning in the Cambridge English Dictionary.* Aug. 16, 2018. URL: `https://dictionary.cambridge.org/dictionary/english/object-oriented` (visited on 08/16/2018).

[4] e-puck eduaction robot. *e-puck education robot.* Aug. 17, 2018. URL: `http://www.e-puck.org/` (visited on 08/17/2018).

[5] ePuck: Spirit of the e-puck project. *Project.* Aug. 28, 2018. URL: `http://www.e-puck.org/index.php?option=com_content&view=article&id=6&Itemid=3` (visited on 08/28/2018).

[6] ePuck: wiki. *e-puck2 - GCtronic wiki.* Aug. 28, 2018. URL: `http://www.gctronic.com/doc/index.php/e-puck2` (visited on 08/28/2018).

[7] ePuck2. *e-puck2.* Feb. 28, 2018. URL: `http://www.e-puck.org/index.php?option=com_content&view=article&id=55&Itemid=42` (visited on 08/28/2018).

[8] T Ryan Fitz-Gibbon. "Brooks' Subsumption Architecture". In: (), p. 19. URL: `https://pdfs.semanticscholar.org/presentation/6c4d/f7e9dcd023c74fab72e22c8e51cd9ef7e` `pdf`.

[9] GeeksforGeeks. *Adapter Pattern.* GeeksforGeeks. May 3, 2016. URL: `https://www.geeksforgeeks.org/adapter-pattern/` (visited on 07/28/2018).

[10] *General Information.* URL: `https://cs.uwaterloo.ca/~gweddell/cs446/` (visited on 08/20/2018).

[11] JAI doc ParameterBlock. *ParameterBlock (Java Platform SE 7).* Aug. 15, 2018. URL: `https://docs.oracle.com/javase/7/docs/api/java/awt/image/renderable/ParameterBlock.html` (visited on 08/15/2018).

[12] JAI RenderedImage. *RenderedImage (Java Platform SE 7).* Aug. 15, 2018. URL: `https://docs.oracle.com/javase/7/docs/api/java/awt/image/RenderedImage.html` (visited on 08/15/2018).

[13] Java Advanced Imaging Oracle. *Java Advanced Imaging API Home Page*. Aug. 15, 2018. URL: https://www.oracle.com/technetwork/java/iio-141084.html (visited on 08/15/2018).

[14] Java Oracle BufferedImage. *BufferedImage (Java Platform SE 7)*. Aug. 15, 2018. URL: https://docs.oracle.com/javase/7/docs/api/java/awt/image/BufferedImage.html (visited on 08/15/2018).

[15] Marcos Diego Catalano. *Catalano-Framework: Framework*. original-date: 2013-08-24T16:59:48Z. Aug. 4, 2018. URL: https://github.com/DiegoCatalano/Catalano-Framework (visited on 08/15/2018).

[16] Francesco Mondada et al. "The e-puck, a Robot Designed for Education in Engineering". In: (Aug. 28, 2018), p. 7.

[17] Open CV documentation. *Overview (OpenCV 3.4.2 Java documentation)*. Aug. 15, 2018. URL: https://docs.opencv.org/3.4/javadoc/index.html (visited on 08/15/2018).

[18] OpenCV. *OpenCV library*. Aug. 15, 2018. URL: https://opencv.org/ (visited on 08/15/2018).

[19] Oracle. *Intrinsic Locks and Synchronization (The Java^{TM} Tutorials > Essential Classes > Concurrency)*. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html (visited on 07/24/2018).

[20] Oracle. *System (Java Platform SE 7)*. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#currentTimeMillis() (visited on 07/24/2018).

[21] Python multiple inheritance. *9. Classes — Python 2.7.15 documentation*. URL: https://docs.python.org/2/tutorial/classes.html#multiple-inheritance (visited on 09/11/2018).

[22] ROS. *ROS.org | About ROS*. Aug. 30, 2018. URL: http://www.ros.org/about-ros/ (visited on 08/30/2018).

[23] R.G. Simmons. "Structured control for autonomous robots". In: *IEEE Transactions on Robotics and Automation* 10.1 (Feb. 1994), pp. 34–43. ISSN: 1042296X. DOI: 10.1109/70.285583. URL: http://ieeexplore.ieee.org/document/285583/ (visited on 09/12/2018).

[24] Software_testing_fundamentals. *Performance Testing*. Software Testing Fundamentals. July 15, 2013. URL: http://softwaretestingfundamentals.com/performance-testing/ (visited on 07/24/2018).

[25] *Solution 1: Control Loop*. URL: https://www.cs.cmu.edu/~ModProb/MRsol1.html (visited on 08/19/2018).

[26] *Solution 2: Layered Architecture*. URL: https://www.cs.cmu.edu/~ModProb/MRsol2.html (visited on 08/19/2018).

[27] *Solution 3: Implicit Invocation.* URL: https://www.cs.cmu.edu/~ModProb/MRsol3.html (visited on 08/19/2018).

[28] *Solution 4: Blackboard Architecture.* URL: https://www.cs.cmu.edu/~ModProb/MRsol4.html (visited on 08/19/2018).

[29] *Task Control Architecture.* URL: http://www.cs.cmu.edu/~TCA/tca.orig.html (visited on 08/24/2018).

[30] ej-technologies. *Java Profiler - JProfiler.* URL: https://www.ej-technologies.com/products/jprofiler/overview.html (visited on 07/24/2018).

[31] Techtarget. *What is object-oriented programming (OOP)? - Definition from WhatIs.com.* URL: https://searchmicroservices.techtarget.com/definition/object-oriented-programming-OOP (visited on 04/03/2018).

[32] Techtarget-Performance_Testing. *What is performance testing? - Definition from WhatIs.com.* SearchSoftwareQuality. URL: https://searchsoftwarequality.techtarget.com/definition/performance-testing (visited on 07/24/2018).

[33] Techterms. *OOP (Object-Oriented Programming) Definition.* URL: https://techterms.com/definition/oop (visited on 04/03/2018).

[34] Testing_Performance. *What is Performance Testing - definitions - Testing Performance.* URL: http://www.testingperformance.org/definitions/what-is-performance-testing (visited on 07/24/2018).

[35] Timer (Java Platform SE 7). *Timer (Java Platform SE 7).* 2018. URL: https://docs.oracle.com/javase/7/docs/api/java/util/Timer.html (visited on 07/30/2018).

[36] TimerTask (Java Platform SE 7). *TimerTask (Java Platform SE 7).* 2018. URL: https://docs.oracle.com/javase/7/docs/api/java/util/TimerTask.html (visited on 07/30/2018).

[37] tutorialspoint.com. *Design Patterns Observer Pattern.* www.tutorialspoint.com. 2018. URL: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm (visited on 07/29/2018).

[38] tutorialspoint.com. *OpenCV Reading Images.* www.tutorialspoint.com. Aug. 15, 2018. URL: https://www.tutorialspoint.com/opencv/opencv_reading_images.htm (visited on 08/15/2018).

[39] tutorialspoint.com. *Performance Testing.* www.tutorialspoint.com. URL: https://www.tutorialspoint.com/software_testing_dictionary/performance_testing.htm (visited on 07/24/2018).

[40] University of Sussex - AI Lecture: Braitenberg Vehicles. *AI Lecture: Braitenberg Vehicles.* URL: http://users.sussex.ac.uk/~christ/crs/kr-ist/lecx1a.html (visited on 09/06/2018).

[41] University Waterloo. *Mobile Robots: A case study on architectural styles.* Oct. 9, 2018. URL: https://cs.uwaterloo.ca/~gweddell/cs446/ArchCase.pdf.

[42] V-REP add-ons. *Add-ons.* Aug. 30, 2018. URL: http://www.coppeliarobotics.com/helpFiles/en/addOns.htm (visited on 08/30/2018).

[43] V-REP embedded scripts. *Embedded scripts.* Aug. 30, 2018. URL: http://www.coppeliarobotics.com/helpFiles/en/scripts.htm (visited on 08/30/2018).

[44] V-REP main client application. *The main client application.* Aug. 30, 2018. URL: http://www.coppeliarobotics.com/helpFiles/en/mainClientApplication.htm (visited on 08/30/2018).

[45] V-REP Plugins. *Plugins.* 2008. URL: http://www.coppeliarobotics.com/helpFiles/en/plugins.htm (visited on 08/30/2018).

[46] V-REP remote API modus operandi. *Remote API modus operandi.* URL: http://www.coppeliarobotics.com/helpFiles/en/remoteApiModusOperandi.htm (visited on 08/30/2018).

[47] V-REP_API_Frameworks. *V-REP API framework.* Aug. 16, 2018. URL: http://www.coppeliarobotics.com/helpFiles/en/apisOverview.htm (visited on 08/16/2018).

[48] V-Rep-GPU_usage. *V-REP's GPU usage - V-REP Forum.* URL: http://www.forum.coppeliarobotics.com/viewtopic.php?t=1113 (visited on 07/26/2018).

[49] V-Rep-licensing. *Coppelia Robotics V-REP: Create. Compose. Simulate. Any Robot: Educational Licensing.* URL: http://www.coppeliarobotics.com/educational-licensing.html (visited on 04/17/2018).

[50] V-Rep-main. *Coppelia Robotics V-REP: Create. Compose. Simulate. Any Robot.* URL: http://www.coppeliarobotics.com/ (visited on 04/17/2018).

[51] V-Rep-remote_API_overview. *Remote API.* URL: http://www.coppeliarobotics.com/helpFiles/en/remoteApiOverview.htm (visited on 04/17/2018).

[52] V-Rep-writing_code. *Writing code in and around V-REP.* URL: http://www.coppeliarobotics.com/helpFiles/en/writingCode.htm#sixMethods (visited on 04/17/2018).

[53] Webots-cpp_java_python. *Webots documentation: C++/Java/Python.* URL: https://www.cyberbotics.com/doc/guide/cpp-java-python (visited on 04/17/2018).

[54] Webots-main_page. *Webots: robot simulator.* URL: https://www.cyberbotics.com/#webots (visited on 04/17/2018).

[55] Webots-prices. *Webots: buy.* URL: https://www.cyberbotics.com/buy (visited on 04/17/2018).

[56] Webots_using_java. *Webots documentation: Using Java.* Nov. 9, 2018. URL: https://www.cyberbotics.com/doc/guide/using-java (visited on 09/11/2018).

Statuatory Declaration

I declare that I have developed and written the enclosed work completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. This Bachelor Thesis was not used in the same or in a similar version to achieve an academic degree nor has it been published elsewhere.

Dornbirn, at 12.10.2018 Daniel Thomas Groß

Appendices

A. Code snippets of section 4.1.1

```
/**
*Register as observer for the sensor values to get updates.
* @param observer
*/
public void registerSensorObserver(SensorObserver observer){
  _sensorObservers.add(observer);
}

/**
 * Unregister from the observer list.
 * @param observer
 */
 public void unregisterSensorObserver(SensorObserver observer){
    _sensorObservers.remove(observer);
 }

 /**
 * Notify all observers about a the change of sensor values by
 * giving them a list of all sensors that have changed.
 * @param sensors
 */
 protected void notifySensorObservers(LinkedList<Sensor> sensors){
   for (SensorObserver observer: _sensorObservers) {
        observer.sensorValuesChanged(sensors);
   }
 }
```

Code A.1.: Methods of the class DifferentialWheels to handle sensor observer

```java
/**
* Register as observer for camera image to get updates.
* @param observer
*/
public void registerCameraImageObserver(CameraImageObserver observer){
    _cameraImageObservers.add(observer);
}

/**
* Unregister from the observer list.
* @param observer
*/
public void unregisterCameraImageObserver(CameraImageObserver observer){
   _cameraImageObservers.remove(observer);
}

/**
* Notify all observers that the camera image has changed by
* sending them the current image.
* @param cameraImage
*/
protected void notifyCameraImageObservers(CameraImage cameraImage){
    for (CameraImageObserver observer: _cameraImageObservers) {
        observer.cameraImageChanged(cameraImage);
    }
}
```

Code A.2.: Methods of the class DifferentialWheels to handle image observers

B. Code snippets of section 4.1.3.1

```
public static void loadLibrary(){
 if(OSValidator.isWindows()){
   if(OSValidator.isWindows64Bit()) {
     loadLib("Windows"+File.separator+"64Bit"+File.separator
              +LIBREMOTEAPIJAVA+".dll","dll");
   } else {
     loadLib("Windows"+File.separator+"32Bit"+File.separator
              +LIBREMOTEAPIJAVA+".dll","dll");
   }
 } else if(OSValidator.isMac()){
     loadLib("Mac"+File.separator+LIBREMOTEAPIJAVA+".dylib","dylib");
 } else if(OSValidator.isUnix() || OSValidator.isSolaris()){
     if(OSValidator.isLinux64Bit()) {
       loadLib("Linux"+File.separator+"64Bit"+File.separator
                +LIBREMOTEAPIJAVA+".so","so");
     } else {
       loadLib("Linux"+File.separator+"32Bit"+File.separator
                +LIBREMOTEAPIJAVA+".so","so");
     }
   } else {
     System.out.println("Error: Unknown Operating System!!");
   }
}
```

Code B.1.: Method "loadLibrary()" of the class LibraryLoader

```java
public static boolean isWindows() {
        return (OS.indexOf("win") >= 0);
}

public static boolean isWindows64Bit(){
        if(windowsArch != null){
            System.out.println(windowsArch);
        }
        if(windowsWow64Arch != null){
            System.out.printf(windowsWow64Arch);
        }
        return (windowsArch != null && windowsArch.endsWith("64"))
            || (windowsWow64Arch != null
            && windowsWow64Arch.endsWith("64")) ? true : false;
}
```

Code B.2.: Methods of OSValidator to detect a Windows OS and the architecture type.

```java
public static boolean isUnix() {
        return (OS.indexOf("nix") >= 0 || OS.indexOf("nux") >= 0
             || OS.indexOf("aix") > 0 );
}

public static boolean isLinux64Bit(){
         return System.getProperty("os.arch").endsWith("64")? true : false;
}

public static boolean isSolaris() {
        return (OS.indexOf("sunos") >= 0);
}
```

Code B.3.: Methods of OSValidator to detect Linux systems.

```java
public static boolean isMac() {
        return (OS.indexOf("mac") >= 0);
}
```

Code B.4.: Method of OSValidator to detect MacOS.

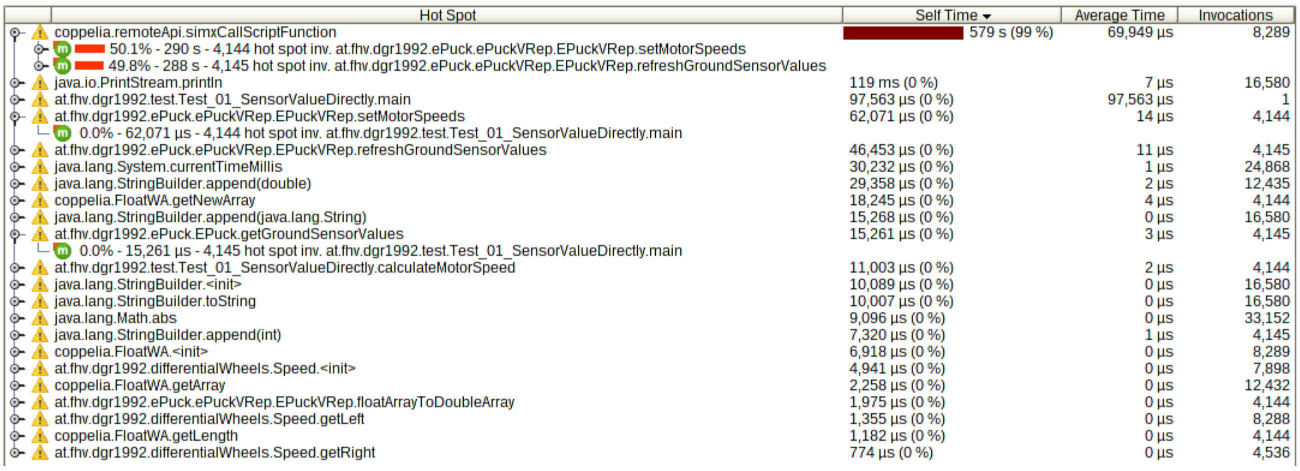
```java
public class remoteApi
{
    static{
        System.loadLibrary("remoteApiJava");
    }
        /*
        *
        *
        */
}
```

Code B.5.: Original version of the class remoteAPI.java

```java
public class remoteApi
{
    static{
        //Load the correct library for the current OS
        LibraryLoader.loadLibrary();
    }
        /*
        *
        *
        */
}
```

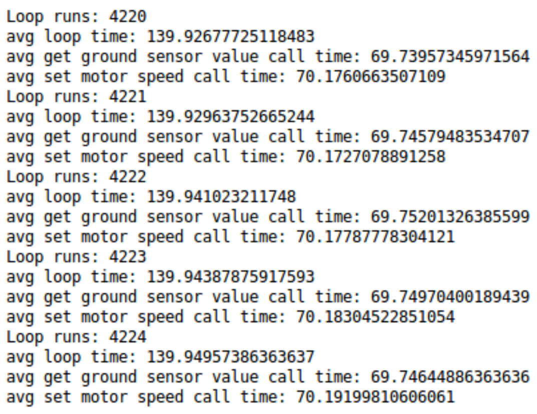
Code B.6.: Modified version of the class remoteAPI.java

C. Figures of the performance testing results



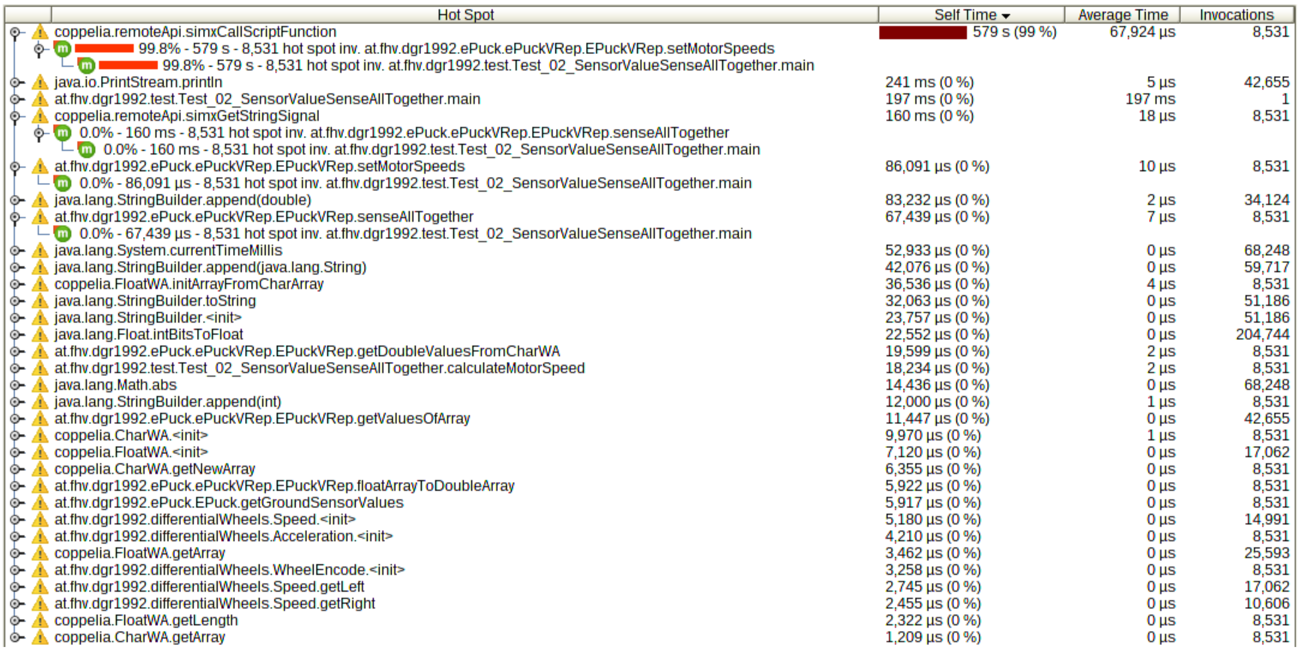
Hot Spot	Self Time	Average Time	Invocations
coppelia.remoteApi.simxCallScriptFunction	579 s (99 %)	69,949 µs	8,289
50.1% - 290 s - 4,144 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds			
49.8% - 288 s - 4,145 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshGroundSensorValues			
java.io.PrintStream.println	119 ms (0 %)	7 µs	16,580
at.fhv.dgr1992.test.Test_01_SensorValueDirectly.main	97,563 µs (0 %)	97,563 µs	1
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds	62,071 µs (0 %)	14 µs	4,144
0.0% - 62,071 µs - 4,144 hot spot inv. at.fhv.dgr1992.test.Test_01_SensorValueDirectly.main			
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshGroundSensorValues	46,453 µs (0 %)	11 µs	4,145
java.lang.System.currentTimeMillis	30,232 µs (0 %)	1 µs	24,868
java.lang.StringBuilder.append(double)	29,358 µs (0 %)	2 µs	12,435
coppelia.FloatWA.getNewArray	18,245 µs (0 %)	4 µs	4,144
java.lang.StringBuilder.append(java.lang.String)	15,268 µs (0 %)	0 µs	16,580
at.fhv.dgr1992.ePuck.EPuck.getGroundSensorValues	15,261 µs (0 %)	3 µs	4,145
0.0% - 15,261 µs - 4,145 hot spot inv. at.fhv.dgr1992.test.Test_01_SensorValueDirectly.main			
at.fhv.dgr1992.test.Test_01_SensorValueDirectly.calculateMotorSpeed	11,003 µs (0 %)	2 µs	4,144
java.lang.StringBuilder.<init>	10,089 µs (0 %)	0 µs	16,580
java.lang.StringBuilder.toString	10,007 µs (0 %)	0 µs	16,580
java.lang.Math.abs	9,096 µs (0 %)	0 µs	33,152
java.lang.StringBuilder.append(int)	7,320 µs (0 %)	1 µs	4,145
coppelia.FloatWA.<init>	6,918 µs (0 %)	0 µs	8,289
at.fhv.dgr1992.differentialWheels.Speed.<init>	4,941 µs (0 %)	0 µs	7,898
coppelia.FloatWA.getArray	2,258 µs (0 %)	0 µs	12,432
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.floatArrayToDoubleArray	1,975 µs (0 %)	0 µs	4,144
at.fhv.dgr1992.differentialWheels.Speed.getLeft	1,355 µs (0 %)	0 µs	8,288
coppelia.FloatWA.getLength	1,182 µs (0 %)	0 µs	4,144
at.fhv.dgr1992.differentialWheels.Speed.getRight	774 µs (0 %)	0 µs	4,536

Figure C.1.: JProfiler results for test 01.



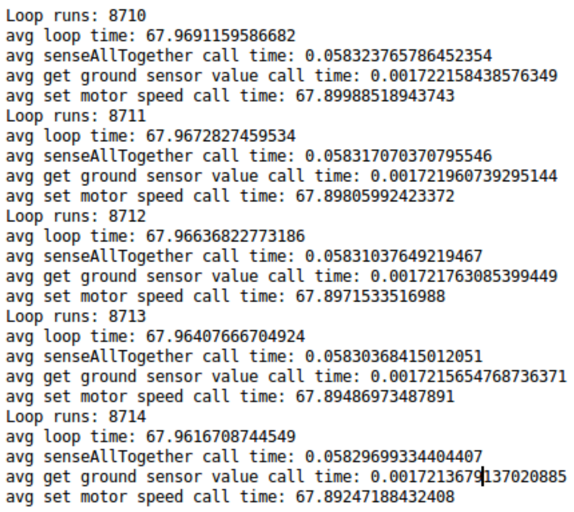
```
Loop runs: 4220
avg loop time: 139.92677725118483
avg get ground sensor value call time: 69.73957345971564
avg set motor speed call time: 70.1760663507109
Loop runs: 4221
avg loop time: 139.92963752665244
avg get ground sensor value call time: 69.74579483534707
avg set motor speed call time: 70.1727078891258
Loop runs: 4222
avg loop time: 139.941023211748
avg get ground sensor value call time: 69.75201326385599
avg set motor speed call time: 70.17787778304121
Loop runs: 4223
avg loop time: 139.94387875917593
avg get ground sensor value call time: 69.74970400189439
avg set motor speed call time: 70.18304522851054
Loop runs: 4224
avg loop time: 139.94957386363637
avg get ground sensor value call time: 69.74644886363636
avg set motor speed call time: 70.19199810606061
```

Figure C.2.: Results for test 01 using the Java system method.



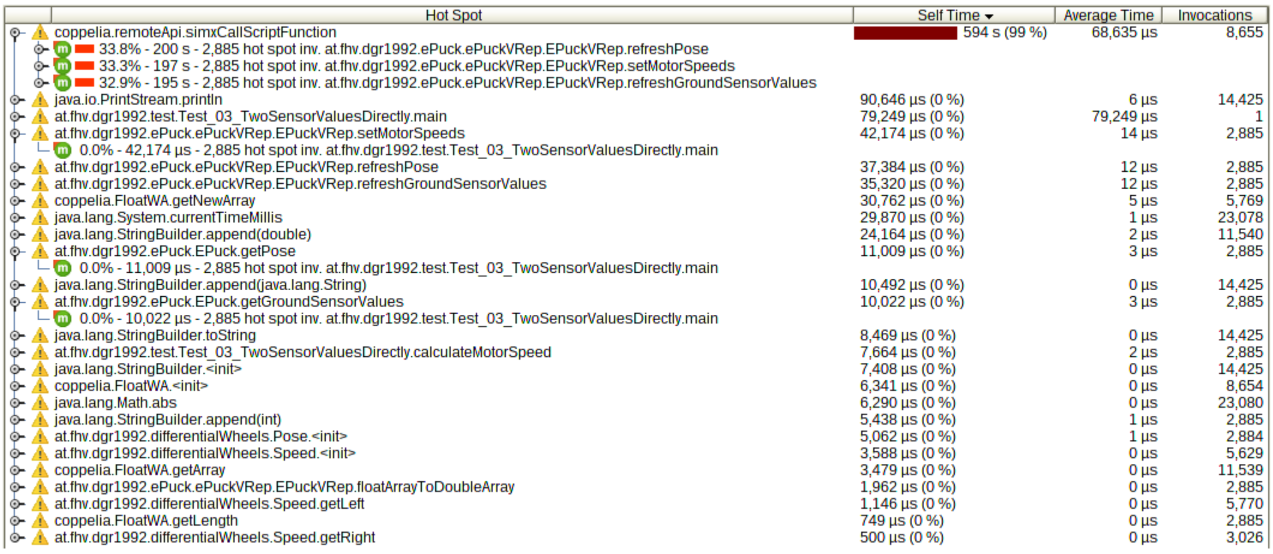
Hot Spot	Self Time	Average Time	Invocations
coppelia.remoteApi.simxCallScriptFunction	579 s (99 %)	67,924 µs	8,531
99.8% - 579 s - 8,531 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds			
99.8% - 579 s - 8,531 hot spot inv. at.fhv.dgr1992.test.Test_02_SensorValueSenseAllTogether.main			
java.io.PrintStream.println	241 ms (0 %)	5 µs	42,655
at.fhv.dgr1992.test.Test_02_SensorValueSenseAllTogether.main	197 ms (0 %)	197 ms	1
coppelia.remoteApi.simxGetStringSignal	160 ms (0 %)	18 µs	8,531
0.0% - 160 ms - 8,531 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.senseAllTogether			
0.0% - 160 ms - 8,531 hot spot inv. at.fhv.dgr1992.test.Test_02_SensorValueSenseAllTogether.main			
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds	86,091 µs (0 %)	10 µs	8,531
0.0% - 86,091 µs - 8,531 hot spot inv. at.fhv.dgr1992.test.Test_02_SensorValueSenseAllTogether.main			
java.lang.StringBuilder.append(double)	83,232 µs (0 %)	2 µs	34,124
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.senseAllTogether	67,439 µs (0 %)	7 µs	8,531
0.0% - 67,439 µs - 8,531 hot spot inv. at.fhv.dgr1992.test.Test_02_SensorValueSenseAllTogether.main			
java.lang.System.currentTimeMillis	52,933 µs (0 %)	0 µs	68,248
java.lang.StringBuilder.append(java.lang.String)	42,076 µs (0 %)	0 µs	59,717
coppelia.FloatWA.initArrayFromCharArray	36,536 µs (0 %)	4 µs	8,531
java.lang.StringBuilder.toString	32,063 µs (0 %)	0 µs	51,186
java.lang.StringBuilder.<init>	23,757 µs (0 %)	0 µs	51,186
java.lang.Float.intBitsToFloat	22,552 µs (0 %)	0 µs	204,744
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getDoubleValuesFromCharWA	19,599 µs (0 %)	2 µs	8,531
at.fhv.dgr1992.test.Test_02_SensorValueSenseAllTogether.calculateMotorSpeed	18,234 µs (0 %)	2 µs	8,531
java.lang.Math.abs	14,436 µs (0 %)	0 µs	68,248
java.lang.StringBuilder.append(int)	12,000 µs (0 %)	1 µs	8,531
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getValuesOfArray	11,447 µs (0 %)	0 µs	42,655
coppelia.CharWA.<init>	9,970 µs (0 %)	1 µs	8,531
coppelia.FloatWA.<init>	7,120 µs (0 %)	0 µs	17,062
coppelia.CharWA.getNewArray	6,355 µs (0 %)	0 µs	8,531
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.floatArrayToDoubleArray	5,922 µs (0 %)	0 µs	8,531
at.fhv.dgr1992.ePuck.EPuck.getGroundSensorValues	5,917 µs (0 %)	0 µs	8,531
at.fhv.dgr1992.differentialWheels.Speed.<init>	5,180 µs (0 %)	0 µs	14,991
at.fhv.dgr1992.differentialWheels.Acceleration.<init>	4,210 µs (0 %)	0 µs	8,531
coppelia.FloatWA.getArray	3,462 µs (0 %)	0 µs	25,593
at.fhv.dgr1992.differentialWheels.WheelEncode.<init>	3,258 µs (0 %)	0 µs	8,531
at.fhv.dgr1992.differentialWheels.Speed.getLeft	2,745 µs (0 %)	0 µs	17,062
at.fhv.dgr1992.differentialWheels.Speed.getRight	2,455 µs (0 %)	0 µs	10,606
coppelia.FloatWA.getLength	2,322 µs (0 %)	0 µs	8,531
coppelia.CharWA.getArray	1,209 µs (0 %)	0 µs	8,531

Figure C.3.: JProfiler results for test 02.



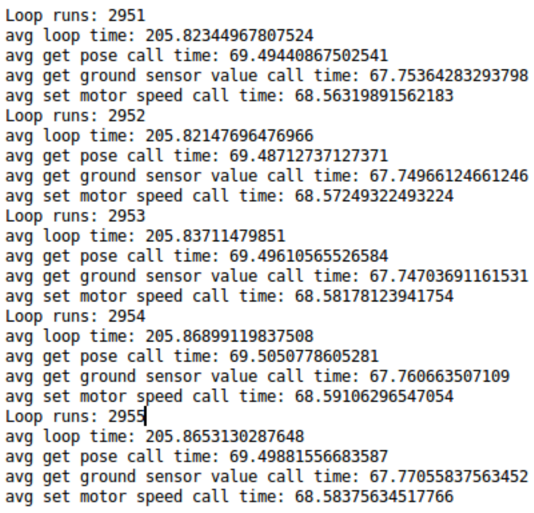
```
Loop runs: 8710
avg loop time: 67.9691159586682
avg senseAllTogether call time: 0.058323765786452354
avg get ground sensor value call time: 0.001722158438576349
avg set motor speed call time: 67.89988518943743
Loop runs: 8711
avg loop time: 67.9672827459534
avg senseAllTogether call time: 0.058317070370795546
avg get ground sensor value call time: 0.0017219607392951144
avg set motor speed call time: 67.89805992423372
Loop runs: 8712
avg loop time: 67.96636822773186
avg senseAllTogether call time: 0.05831037649219467
avg get ground sensor value call time: 0.0017217630853994449
avg set motor speed call time: 67.8971533516988
Loop runs: 8713
avg loop time: 67.96407666704924
avg senseAllTogether call time: 0.05830368415012051
avg get ground sensor value call time: 0.0017215654768736371
avg set motor speed call time: 67.89486973487891
Loop runs: 8714
avg loop time: 67.9616708744549
avg senseAllTogether call time: 0.05829699334404407
avg get ground sensor value call time: 0.0017213679137020885
avg set motor speed call time: 67.89247188432408
```

Figure C.4.: Results for test 02 using the Java system method.



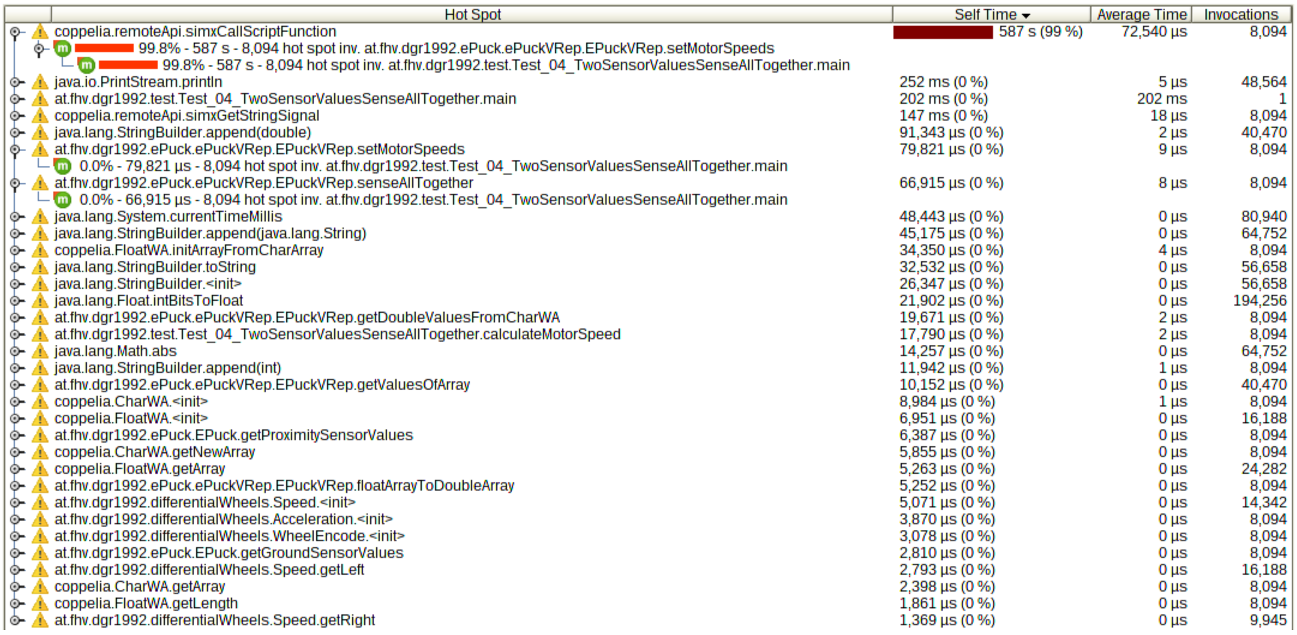
Hot Spot	Self Time	Average Time	Invocations
coppelia.remoteApi.simxCallScriptFunction	594 s (99 %)	68,635 µs	8,655
33.8% - 200 s - 2,885 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshPose			
33.3% - 197 s - 2,885 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds			
32.9% - 195 s - 2,885 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshGroundSensorValues			
java.io.PrintStream.println	90,646 µs (0 %)	6 µs	14,425
at.fhv.dgr1992.test.Test_03_TwoSensorValuesDirectly.main	79,249 µs (0 %)	79,249 µs	1
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds	42,174 µs (0 %)	14 µs	2,885
0.0% - 42,174 µs - 2,885 hot spot inv. at.fhv.dgr1992.test.Test_03_TwoSensorValuesDirectly.main			
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshPose	37,384 µs (0 %)	12 µs	2,885
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshGroundSensorValues	35,320 µs (0 %)	12 µs	2,885
coppelia.FloatWA.getNewArray	30,762 µs (0 %)	5 µs	5,769
java.lang.System.currentTimeMillis	29,870 µs (0 %)	1 µs	23,078
java.lang.StringBuilder.append(double)	24,164 µs (0 %)	2 µs	11,540
at.fhv.dgr1992.ePuck.EPuck.getPose	11,009 µs (0 %)	3 µs	2,885
0.0% - 11,009 µs - 2,885 hot spot inv. at.fhv.dgr1992.test.Test_03_TwoSensorValuesDirectly.main			
java.lang.StringBuilder.append(java.lang.String)	10,492 µs (0 %)	0 µs	14,425
at.fhv.dgr1992.ePuck.EPuck.getGroundSensorValues	10,022 µs (0 %)	3 µs	2,885
0.0% - 10,022 µs - 2,885 hot spot inv. at.fhv.dgr1992.test.Test_03_TwoSensorValuesDirectly.main			
java.lang.StringBuilder.toString	8,469 µs (0 %)	0 µs	14,425
at.fhv.dgr1992.test.Test_03_TwoSensorValuesDirectly.calculateMotorSpeed	7,664 µs (0 %)	2 µs	2,885
java.lang.StringBuilder.<init>	7,408 µs (0 %)	0 µs	14,425
coppelia.FloatWA.<init>	6,341 µs (0 %)	0 µs	8,654
java.lang.Math.abs	6,290 µs (0 %)	0 µs	23,080
java.lang.StringBuilder.append(int)	5,438 µs (0 %)	1 µs	2,885
at.fhv.dgr1992.differentialWheels.Pose.<init>	5,062 µs (0 %)	1 µs	2,884
at.fhv.dgr1992.differentialWheels.Speed.<init>	3,588 µs (0 %)	0 µs	5,629
coppelia.FloatWA.getArray	3,479 µs (0 %)	0 µs	11,539
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.floatArrayToDoubleArray	1,962 µs (0 %)	0 µs	2,885
at.fhv.dgr1992.differentialWheels.Speed.getLeft	1,146 µs (0 %)	0 µs	5,770
coppelia.FloatWA.getLength	749 µs (0 %)	0 µs	2,885
at.fhv.dgr1992.differentialWheels.Speed.getRight	500 µs (0 %)	0 µs	3,026

Figure C.5.: JProfiler results for test 03.



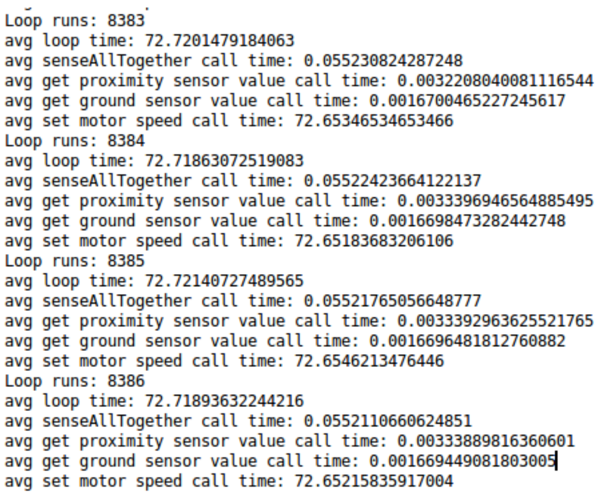
```
Loop runs: 2951
avg loop time: 205.82344967807524
avg get pose call time: 69.49440867502541
avg get ground sensor value call time: 67.75364283293798
avg set motor speed call time: 68.56319891562183
Loop runs: 2952
avg loop time: 205.82147696476966
avg get pose call time: 69.48712737127371
avg get ground sensor value call time: 67.74966124661246
avg set motor speed call time: 68.57249322493224
Loop runs: 2953
avg loop time: 205.83711479851
avg get pose call time: 69.49610565526584
avg get ground sensor value call time: 67.74703691161531
avg set motor speed call time: 68.58178123941754
Loop runs: 2954
avg loop time: 205.86899119837508
avg get pose call time: 69.5050778605281
avg get ground sensor value call time: 67.760663507109
avg set motor speed call time: 68.59106296547054
Loop runs: 2955
avg loop time: 205.8653130287648
avg get pose call time: 69.49881556683587
avg get ground sensor value call time: 67.77055837563452
avg set motor speed call time: 68.58375634517766
```

Figure C.6.: Results for test 03 using the Java system method.



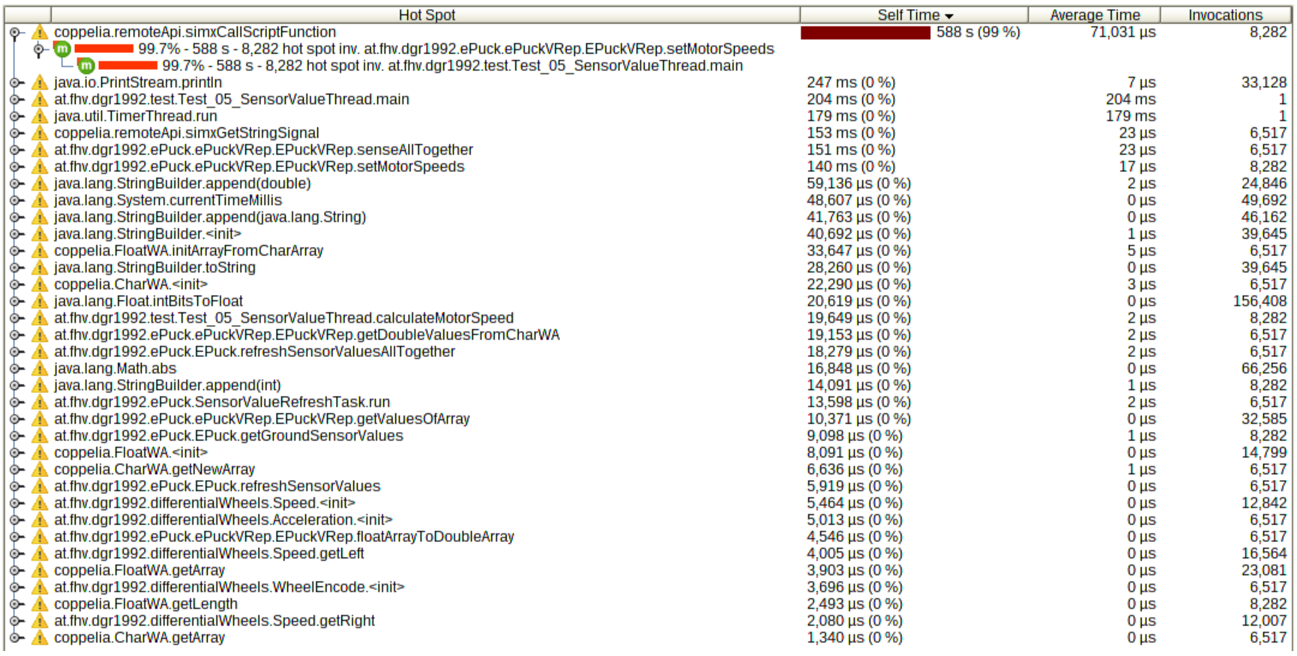
Hot Spot	Self Time	Average Time	Invocations
coppelia.remoteApi.simxCallScriptFunction	587 s (99 %)	72,540 µs	8,094
99.8% - 587 s - 8,094 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds			
99.8% - 587 s - 8,094 hot spot inv. at.fhv.dgr1992.test.Test_04_TwoSensorValuesSenseAllTogether.main			
java.io.PrintStream.println	252 ms (0 %)	5 µs	48,564
at.fhv.dgr1992.test.Test_04_TwoSensorValuesSenseAllTogether.main	202 ms (0 %)	202 ms	1
coppelia.remoteApi.simxGetStringSignal	147 ms (0 %)	18 µs	8,094
java.lang.StringBuilder.append(double)	91,343 µs (0 %)	2 µs	40,470
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds	79,821 µs (0 %)	9 µs	8,094
0.0% - 79,821 µs - 8,094 hot spot inv. at.fhv.dgr1992.test.Test_04_TwoSensorValuesSenseAllTogether.main			
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.senseAllTogether	66,915 µs (0 %)	8 µs	8,094
0.0% - 66,915 µs - 8,094 hot spot inv. at.fhv.dgr1992.test.Test_04_TwoSensorValuesSenseAllTogether.main			
java.lang.System.currentTimeMillis	48,443 µs (0 %)	0 µs	80,940
java.lang.StringBuilder.append(java.lang.String)	45,175 µs (0 %)	0 µs	64,752
coppelia.FloatWA.initArrayFromCharArray	34,350 µs (0 %)	4 µs	8,094
java.lang.StringBuilder.toString	32,532 µs (0 %)	0 µs	56,658
java.lang.StringBuilder.<init>	26,347 µs (0 %)	0 µs	56,658
java.lang.Float.intBitsToFloat	21,902 µs (0 %)	0 µs	194,256
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getDoubleValuesFromCharWA	19,671 µs (0 %)	2 µs	8,094
at.fhv.dgr1992.test.Test_04_TwoSensorValuesSenseAllTogether.calculateMotorSpeed	17,790 µs (0 %)	2 µs	8,094
java.lang.Math.abs	14,257 µs (0 %)	0 µs	64,752
java.lang.StringBuilder.append(int)	11,942 µs (0 %)	1 µs	8,094
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getValuesOfArray	10,152 µs (0 %)	0 µs	40,470
coppelia.CharWA.<init>	8,984 µs (0 %)	1 µs	8,094
coppelia.FloatWA.<init>	6,951 µs (0 %)	0 µs	16,188
at.fhv.dgr1992.ePuck.EPuck.getProximitySensorValues	6,387 µs (0 %)	0 µs	8,094
coppelia.CharWA.getNewArray	5,855 µs (0 %)	0 µs	8,094
coppelia.FloatWA.getArray	5,263 µs (0 %)	0 µs	24,282
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.floatArrayToDoubleArray	5,252 µs (0 %)	0 µs	8,094
at.fhv.dgr1992.differentialWheels.Speed.<init>	5,071 µs (0 %)	0 µs	14,342
at.fhv.dgr1992.differentialWheels.Acceleration.<init>	3,870 µs (0 %)	0 µs	8,094
at.fhv.dgr1992.differentialWheels.WheelEncode.<init>	3,078 µs (0 %)	0 µs	8,094
at.fhv.dgr1992.ePuck.EPuck.getGroundSensorValues	2,810 µs (0 %)	0 µs	8,094
at.fhv.dgr1992.differentialWheels.Speed.getLeft	2,793 µs (0 %)	0 µs	16,188
coppelia.CharWA.getArray	2,398 µs (0 %)	0 µs	8,094
coppelia.FloatWA.getLength	1,861 µs (0 %)	0 µs	8,094
at.fhv.dgr1992.differentialWheels.Speed.getRight	1,369 µs (0 %)	0 µs	9,945

Figure C.7.: JProfiler results for test 04.



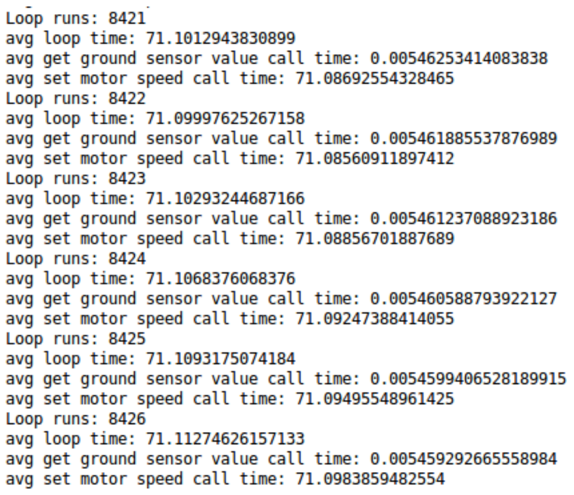
```
Loop runs: 8383
avg loop time: 72.7201479184063
avg senseAllTogether call time: 0.055230824287248
avg get proximity sensor value call time: 0.0032208040081116544
avg get ground sensor value call time: 0.0016700465227245617
avg set motor speed call time: 72.65346534653466
Loop runs: 8384
avg loop time: 72.71863072519083
avg senseAllTogether call time: 0.05522423664122137
avg get proximity sensor value call time: 0.0033396946564885495
avg get ground sensor value call time: 0.0016698473282442748
avg set motor speed call time: 72.65183683206106
Loop runs: 8385
avg loop time: 72.72140727489565
avg senseAllTogether call time: 0.05521765056648777
avg get proximity sensor value call time: 0.0033392963625521765
avg get ground sensor value call time: 0.0016696481812760882
avg set motor speed call time: 72.6546213476446
Loop runs: 8386
avg loop time: 72.71893632244216
avg senseAllTogether call time: 0.0552110660624851
avg get proximity sensor value call time: 0.003338898163606001
avg get ground sensor value call time: 0.0016694490818030005
avg set motor speed call time: 72.65215835917004
```

Figure C.8.: Results for test 04 using the Java system method.



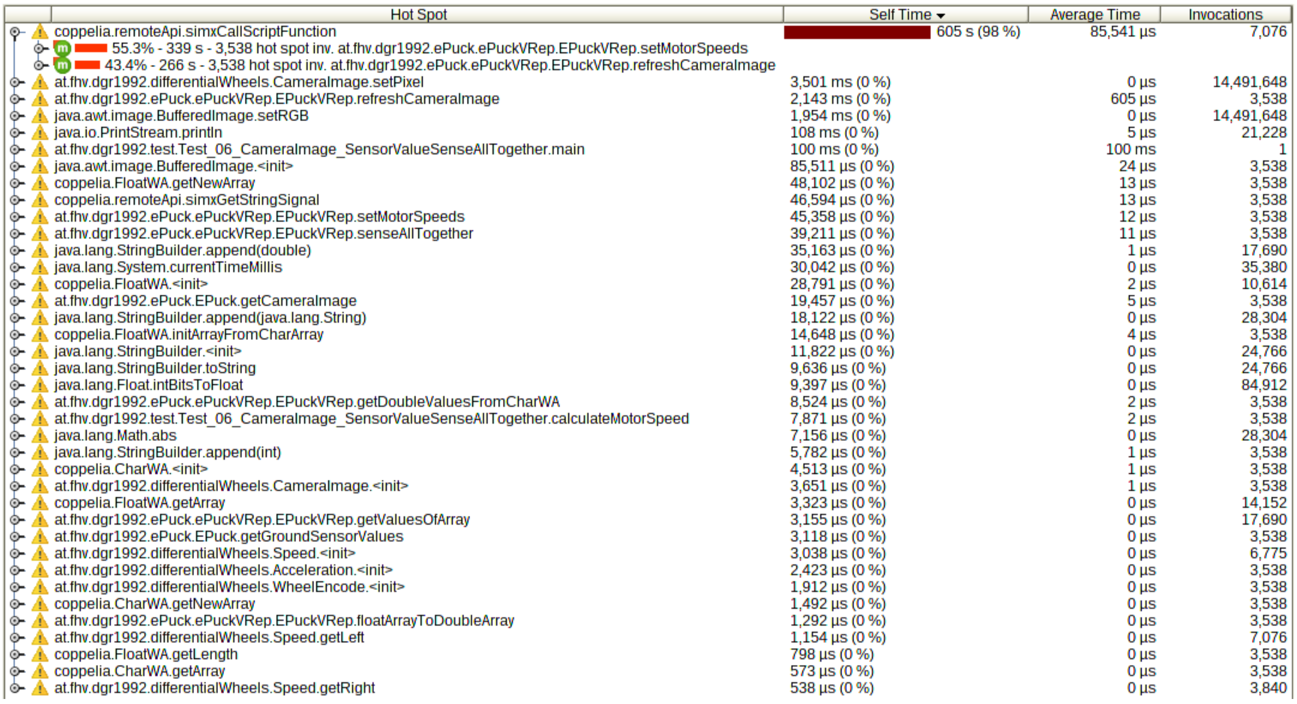
Hot Spot	Self Time	Average Time	Invocations
coppelia.remoteApi.simxCallScriptFunction	588 s (99 %)	71,031 µs	8,282
99.7% - 588 s - 8,282 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds			
99.7% - 588 s - 8,282 hot spot inv. at.fhv.dgr1992.test.Test_05_SensorValueThread.main			
java.io.PrintStream.println	247 ms (0 %)	7 µs	33,128
at.fhv.dgr1992.test.Test_05_SensorValueThread.main	204 ms (0 %)	204 ms	1
java.util.TimerThread.run	179 ms (0 %)	179 ms	1
coppelia.remoteApi.simxGetStringSignal	153 ms (0 %)	23 µs	6,517
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.senseAllTogether	151 ms (0 %)	23 µs	6,517
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds	140 ms (0 %)	17 µs	8,282
java.lang.StringBuilder.append(double)	59,136 µs (0 %)	2 µs	24,846
java.lang.System.currentTimeMillis	48,607 µs (0 %)	0 µs	49,692
java.lang.StringBuilder.append(java.lang.String)	41,763 µs (0 %)	0 µs	46,162
java.lang.StringBuilder.<init>	40,692 µs (0 %)	1 µs	39,645
coppelia.FloatWA.initArrayFromCharArray	33,647 µs (0 %)	5 µs	6,517
java.lang.StringBuilder.toString	28,260 µs (0 %)	0 µs	39,645
coppelia.CharWA.<init>	22,290 µs (0 %)	3 µs	6,517
java.lang.Float.intBitsToFloat	20,619 µs (0 %)	0 µs	156,408
at.fhv.dgr1992.test.Test_05_SensorValueThread.calculateMotorSpeed	19,649 µs (0 %)	2 µs	8,282
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getDoubleValuesFromCharWA	19,153 µs (0 %)	2 µs	6,517
at.fhv.dgr1992.ePuck.EPuck.refreshSensorValuesAllTogether	18,279 µs (0 %)	2 µs	6,517
java.lang.Math.abs	16,848 µs (0 %)	0 µs	66,256
java.lang.StringBuilder.append(int)	14,091 µs (0 %)	1 µs	8,282
at.fhv.dgr1992.ePuck.SensorValueRefreshTask.run	13,598 µs (0 %)	2 µs	6,517
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getValuesOfArray	10,371 µs (0 %)	0 µs	32,585
at.fhv.dgr1992.ePuck.EPuck.getGroundSensorValues	9,098 µs (0 %)	1 µs	8,282
coppelia.FloatWA.<init>	8,091 µs (0 %)	0 µs	14,799
coppelia.CharWA.getNewArray	6,636 µs (0 %)	1 µs	6,517
at.fhv.dgr1992.ePuck.EPuck.refreshSensorValues	5,919 µs (0 %)	0 µs	6,517
at.fhv.dgr1992.differentialWheels.Speed.<init>	5,464 µs (0 %)	0 µs	12,842
at.fhv.dgr1992.differentialWheels.Acceleration.<init>	5,013 µs (0 %)	0 µs	6,517
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.floatArrayToDoubleArray	4,546 µs (0 %)	0 µs	6,517
at.fhv.dgr1992.differentialWheels.Speed.getLeft	4,005 µs (0 %)	0 µs	16,564
coppelia.FloatWA.getArray	3,903 µs (0 %)	0 µs	23,081
at.fhv.dgr1992.differentialWheels.WheelEncode.<init>	3,696 µs (0 %)	0 µs	6,517
coppelia.FloatWA.getLength	2,493 µs (0 %)	0 µs	8,282
at.fhv.dgr1992.differentialWheels.Speed.getRight	2,080 µs (0 %)	0 µs	12,007
coppelia.CharWA.getArray	1,340 µs (0 %)	0 µs	6,517

Figure C.9.: JProfiler results for test 05.



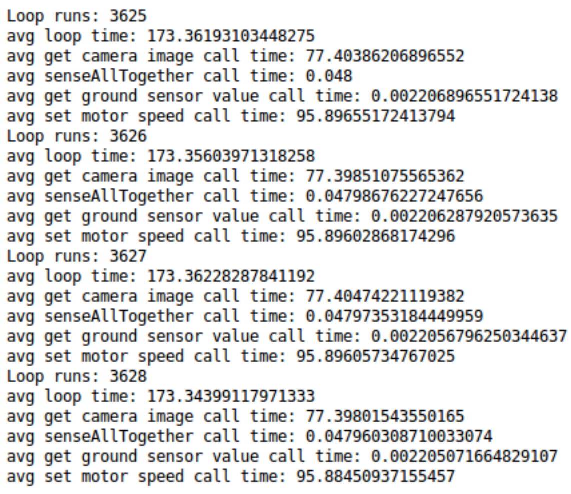
```
Loop runs: 8421
avg loop time: 71.1012943830899
avg get ground sensor value call time: 0.00546253414083838
avg set motor speed call time: 71.08692554328465
Loop runs: 8422
avg loop time: 71.09997625267158
avg get ground sensor value call time: 0.005461885537876989
avg set motor speed call time: 71.08560911897412
Loop runs: 8423
avg loop time: 71.10293244687166
avg get ground sensor value call time: 0.005461237088923186
avg set motor speed call time: 71.08856701887689
Loop runs: 8424
avg loop time: 71.1068376068376
avg get ground sensor value call time: 0.005460588793922127
avg set motor speed call time: 71.09247388414055
Loop runs: 8425
avg loop time: 71.1093175074184
avg get ground sensor value call time: 0.0054599406528189915
avg set motor speed call time: 71.09495548961425
Loop runs: 8426
avg loop time: 71.11274626157133
avg get ground sensor value call time: 0.005459292665558984
avg set motor speed call time: 71.0983859482554
```

Figure C.10.: Results for test 05 using the Java system method.



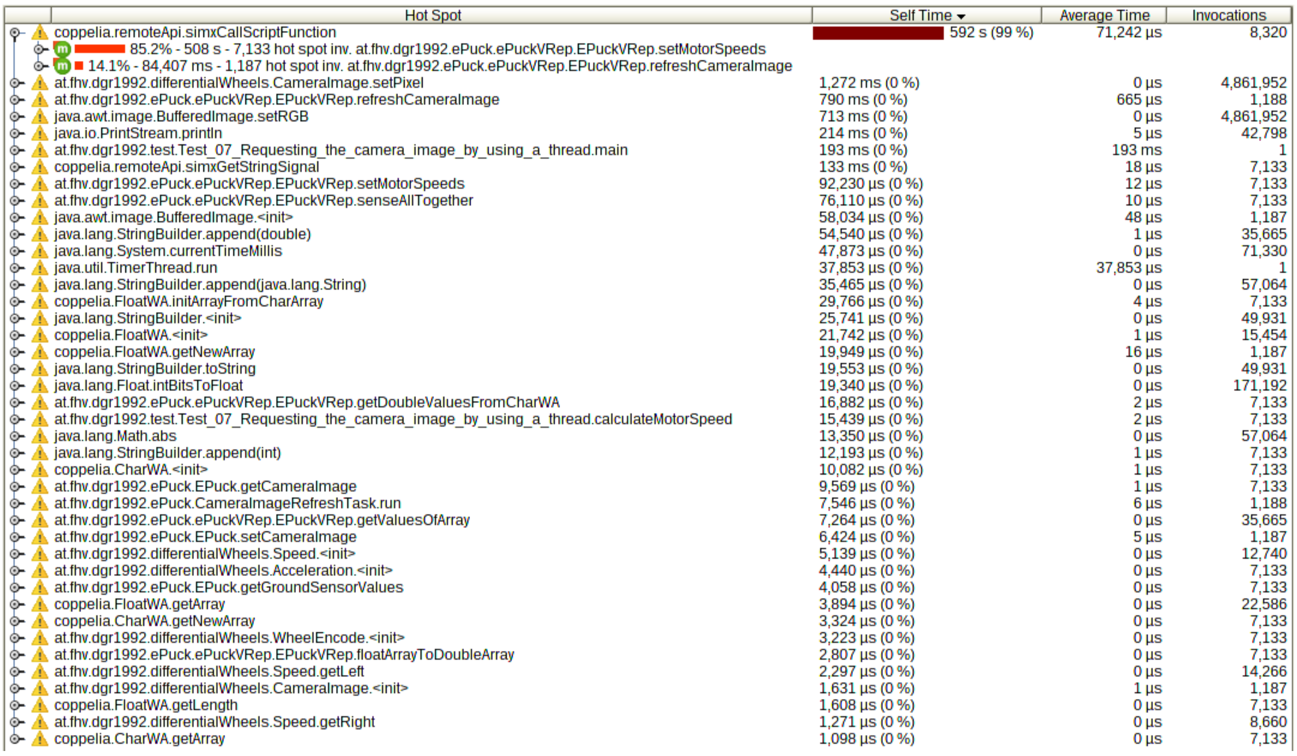
Hot Spot	Self Time	Average Time	Invocations
coppelia.remoteApi.simxCallScriptFunction	605 s (98 %)	85,541 µs	7,076
55.3% - 339 s - 3,538 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds			
43.4% - 266 s - 3,538 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshCameraImage			
at.fhv.dgr1992.differentialWheels.CameraImage.setPixel	3,501 ms (0 %)	0 µs	14,491,648
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshCameraImage	2,143 ms (0 %)	605 µs	3,538
java.awt.image.BufferedImage.setRGB	1,954 ms (0 %)	0 µs	14,491,648
java.io.PrintStream.println	108 ms (0 %)	5 µs	21,228
at.fhv.dgr1992.test.Test_06_CameraImage_SensorValueSenseAllTogether.main	100 ms (0 %)	100 ms	1
java.awt.image.BufferedImage.<init>	85,511 µs (0 %)	24 µs	3,538
coppelia.FloatWA.getNewArray	48,102 µs (0 %)	13 µs	3,538
coppelia.remoteApi.simxGetStringSignal	46,594 µs (0 %)	13 µs	3,538
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds	45,358 µs (0 %)	12 µs	3,538
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.senseAllTogether	39,211 µs (0 %)	11 µs	3,538
java.lang.StringBuilder.append(double)	35,163 µs (0 %)	1 µs	17,690
java.lang.System.currentTimeMillis	30,042 µs (0 %)	0 µs	35,380
coppelia.FloatWA.<init>	28,791 µs (0 %)	2 µs	10,614
at.fhv.dgr1992.ePuck.EPuck.getCameraImage	19,457 µs (0 %)	5 µs	3,538
java.lang.StringBuilder.append(java.lang.String)	18,122 µs (0 %)	0 µs	28,304
coppelia.FloatWA.initArrayFromCharArray	14,648 µs (0 %)	4 µs	3,538
java.lang.StringBuilder.<init>	11,822 µs (0 %)	0 µs	24,766
java.lang.StringBuilder.toString	9,636 µs (0 %)	0 µs	24,766
java.lang.Float.intBitsToFloat	9,397 µs (0 %)	0 µs	84,912
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getDoubleValuesFromCharWA	8,524 µs (0 %)	2 µs	3,538
at.fhv.dgr1992.test.Test_06_CameraImage_SensorValueSenseAllTogether.calculateMotorSpeed	7,871 µs (0 %)	2 µs	3,538
java.lang.Math.abs	7,156 µs (0 %)	0 µs	28,304
java.lang.StringBuilder.append(int)	5,782 µs (0 %)	1 µs	3,538
coppelia.CharWA.<init>	4,513 µs (0 %)	1 µs	3,538
at.fhv.dgr1992.differentialWheels.CameraImage.<init>	3,651 µs (0 %)	1 µs	3,538
coppelia.FloatWA.getArray	3,323 µs (0 %)	0 µs	14,152
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getValuesOfArray	3,155 µs (0 %)	0 µs	17,690
at.fhv.dgr1992.ePuck.EPuck.getGroundSensorValues	3,118 µs (0 %)	0 µs	3,538
at.fhv.dgr1992.differentialWheels.Speed.<init>	3,038 µs (0 %)	0 µs	6,775
at.fhv.dgr1992.differentialWheels.Acceleration.<init>	2,423 µs (0 %)	0 µs	3,538
at.fhv.dgr1992.differentialWheels.WheelEncode.<init>	1,912 µs (0 %)	0 µs	3,538
coppelia.CharWA.getNewArray	1,492 µs (0 %)	0 µs	3,538
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.floatArrayToDoubleArray	1,292 µs (0 %)	0 µs	3,538
at.fhv.dgr1992.differentialWheels.Speed.getLeft	1,154 µs (0 %)	0 µs	7,076
coppelia.FloatWA.getLength	798 µs (0 %)	0 µs	3,538
coppelia.CharWA.getArray	573 µs (0 %)	0 µs	3,538
at.fhv.dgr1992.differentialWheels.Speed.getRight	538 µs (0 %)	0 µs	3,840

Figure C.11.: JProfiler results for test 06.



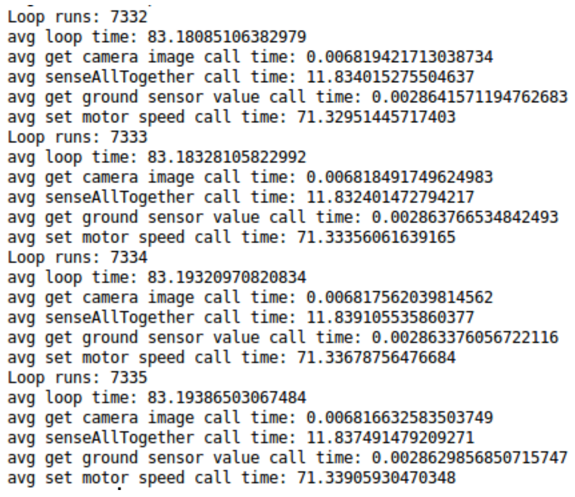
```
Loop runs: 3625
avg loop time: 173.36193103448275
avg get camera image call time: 77.40386206896552
avg senseAllTogether call time: 0.048
avg get ground sensor value call time: 0.0022068965517241 38
avg set motor speed call time: 95.89655172413794
Loop runs: 3626
avg loop time: 173.35603971318258
avg get camera image call time: 77.39851075565362
avg senseAllTogether call time: 0.04798676227247656
avg get ground sensor value call time: 0.002206287920573635
avg set motor speed call time: 95.89602868174296
Loop runs: 3627
avg loop time: 173.36228287841192
avg get camera image call time: 77.40474221119382
avg senseAllTogether call time: 0.04797353184449959
avg get ground sensor value call time: 0.0022056796250344637
avg set motor speed call time: 95.89605734767025
Loop runs: 3628
avg loop time: 173.34399117971333
avg get camera image call time: 77.39801543550165
avg senseAllTogether call time: 0.047960308710033074
avg get ground sensor value call time: 0.002205071664829107
avg set motor speed call time: 95.88450937155457
```

Figure C.12.: Results for test 06 using the Java system method.



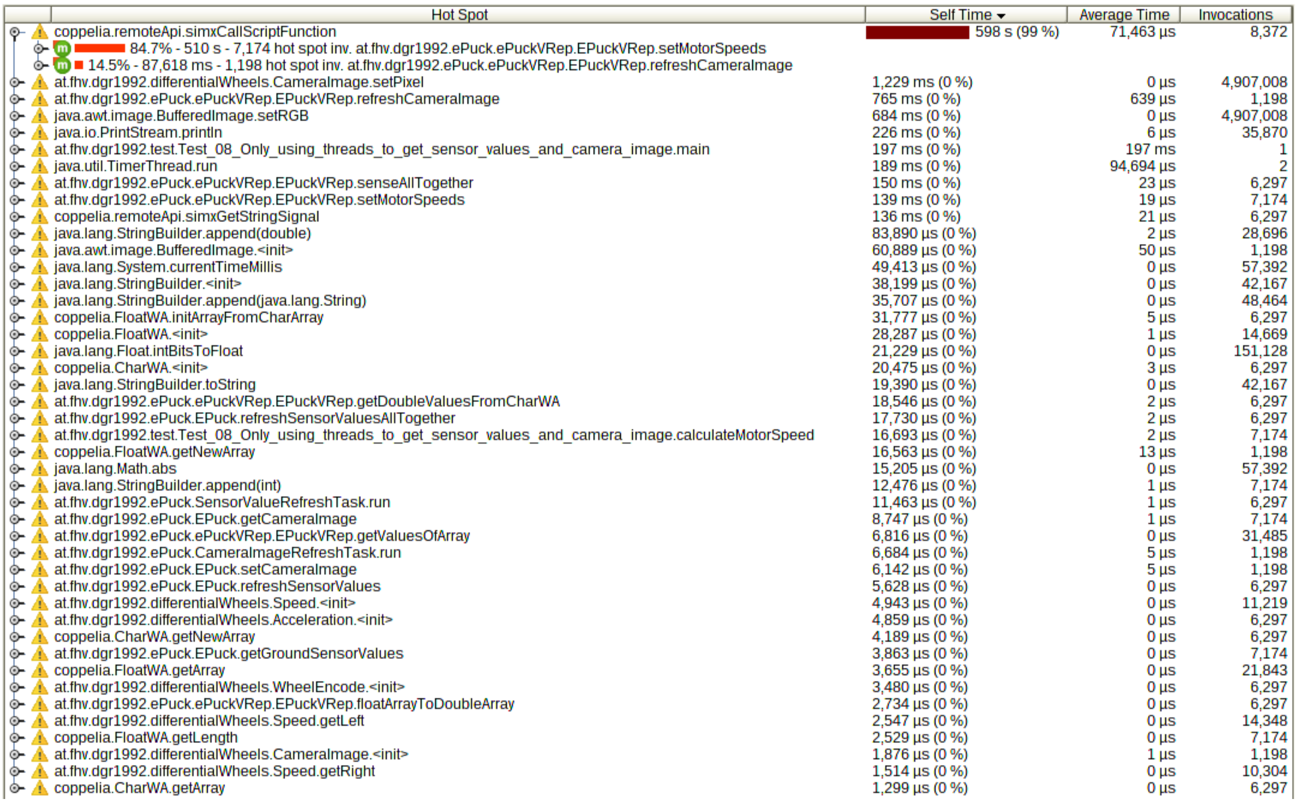
Hot Spot	Self Time	Average Time	Invocations
coppelia.remoteApi.simxCallScriptFunction	592 s (99 %)	71,242 µs	8,320
85.2% - 508 s - 7,133 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds			
14.1% - 84,407 ms - 1,187 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshCameraImage			
at.fhv.dgr1992.differentialWheels.CameraImage.setPixel	1,272 ms (0 %)	0 µs	4,861,952
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshCameraImage	790 ms (0 %)	665 µs	1,188
java.awt.image.BufferedImage.setRGB	713 ms (0 %)	0 µs	4,861,952
java.io.PrintStream.println	214 ms (0 %)	5 µs	42,798
at.fhv.dgr1992.test.Test_07_Requesting_the_camera_image_by_using_a_thread.main	193 ms (0 %)	193 ms	1
coppelia.remoteApi.simxGetStringSignal	133 ms (0 %)	18 µs	7,133
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds	92,230 µs (0 %)	12 µs	7,133
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.senseAllTogether	76,110 µs (0 %)	10 µs	7,133
java.awt.image.BufferedImage.<init>	58,034 µs (0 %)	48 µs	1,187
java.lang.StringBuilder.append(double)	54,540 µs (0 %)	1 µs	35,665
java.lang.System.currentTimeMillis	47,873 µs (0 %)	0 µs	71,330
java.util.TimerThread.run	37,853 µs (0 %)	37,853 µs	1
java.lang.StringBuilder.append(java.lang.String)	35,465 µs (0 %)	0 µs	57,064
coppelia.FloatWA.initArrayFromCharArray	29,766 µs (0 %)	4 µs	7,133
java.lang.StringBuilder.<init>	25,741 µs (0 %)	0 µs	49,931
coppelia.FloatWA.<init>	21,742 µs (0 %)	1 µs	15,454
coppelia.FloatWA.getNewArray	19,949 µs (0 %)	16 µs	1,187
java.lang.StringBuilder.toString	19,553 µs (0 %)	0 µs	49,931
java.lang.Float.intBitsToFloat	19,340 µs (0 %)	0 µs	171,192
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getDoubleValuesFromCharWA	16,882 µs (0 %)	2 µs	7,133
at.fhv.dgr1992.test.Test_07_Requesting_the_camera_image_by_using_a_thread.calculateMotorSpeed	15,439 µs (0 %)	2 µs	7,133
java.lang.Math.abs	13,350 µs (0 %)	0 µs	57,064
java.lang.StringBuilder.append(int)	12,193 µs (0 %)	1 µs	7,133
coppelia.CharWA.<init>	10,082 µs (0 %)	1 µs	7,133
at.fhv.dgr1992.ePuck.EPuck.getCameraImage	9,569 µs (0 %)	1 µs	7,133
at.fhv.dgr1992.ePuck.CameraImageRefreshTask.run	7,546 µs (0 %)	6 µs	1,188
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getValuesOfArray	7,264 µs (0 %)	0 µs	35,665
at.fhv.dgr1992.ePuck.EPuck.setCameraImage	6,424 µs (0 %)	5 µs	1,187
at.fhv.dgr1992.differentialWheels.Speed.<init>	5,139 µs (0 %)	0 µs	12,740
at.fhv.dgr1992.differentialWheels.Acceleration.<init>	4,440 µs (0 %)	0 µs	7,133
at.fhv.dgr1992.ePuck.EPuck.getGroundSensorValues	4,058 µs (0 %)	0 µs	7,133
coppelia.FloatWA.getArray	3,894 µs (0 %)	0 µs	22,586
coppelia.CharWA.getNewArray	3,324 µs (0 %)	0 µs	7,133
at.fhv.dgr1992.differentialWheels.WheelEncode.<init>	3,223 µs (0 %)	0 µs	7,133
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.floatArrayToDoubleArray	2,807 µs (0 %)	0 µs	7,133
at.fhv.dgr1992.differentialWheels.Speed.getLeft	2,297 µs (0 %)	0 µs	14,266
at.fhv.dgr1992.differentialWheels.CameraImage.<init>	1,631 µs (0 %)	1 µs	1,187
coppelia.FloatWA.getLength	1,608 µs (0 %)	0 µs	7,133
at.fhv.dgr1992.differentialWheels.Speed.getRight	1,271 µs (0 %)	0 µs	8,660
coppelia.CharWA.getArray	1,098 µs (0 %)	0 µs	7,133

Figure C.13.: JProfiler results for test 07.



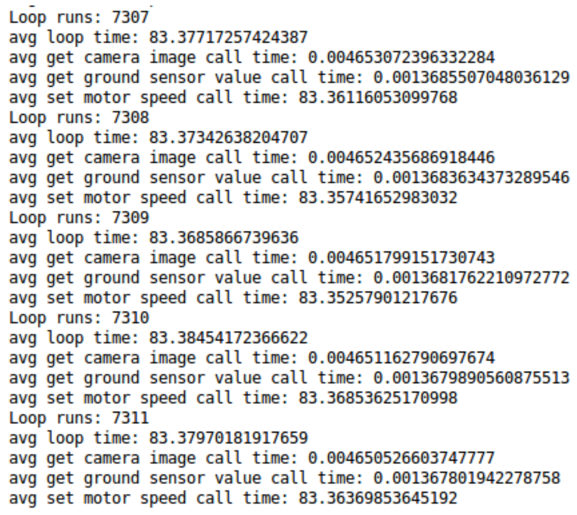
```
Loop runs: 7332
avg loop time: 83.18085106382979
avg get camera image call time: 0.006819421713038734
avg senseAllTogether call time: 11.834015275504637
avg get ground sensor value call time: 0.0028641571194762683
avg set motor speed call time: 71.32951445717403
Loop runs: 7333
avg loop time: 83.18328105822992
avg get camera image call time: 0.006818491749624983
avg senseAllTogether call time: 11.832401472794217
avg get ground sensor value call time: 0.002863766534842493
avg set motor speed call time: 71.33356061639165
Loop runs: 7334
avg loop time: 83.19320970820834
avg get camera image call time: 0.006817562039814562
avg senseAllTogether call time: 11.839105535860377
avg get ground sensor value call time: 0.002863376056722116
avg set motor speed call time: 71.33678756476684
Loop runs: 7335
avg loop time: 83.19386503067484
avg get camera image call time: 0.006816632583503749
avg senseAllTogether call time: 11.837491479209271
avg get ground sensor value call time: 0.0028629856850715747
avg set motor speed call time: 71.33905930470348
```

Figure C.14.: Results for test 07 using the Java system method.



Hot Spot	Self Time	Average Time	Invocations
coppelia.remoteApi.simxCallScriptFunction	598 s (99 %)	71,463 µs	8,372
84.7% - 510 s - 7,174 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds			
14.5% - 87,618 ms - 1,198 hot spot inv. at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshCameraImage			
at.fhv.dgr1992.differentialWheels.CameraImage.setPixel	1,229 ms (0 %)	0 µs	4,907,008
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.refreshCameraImage	765 ms (0 %)	639 µs	1,198
java.awt.image.BufferedImage.setRGB	684 ms (0 %)	0 µs	4,907,008
java.io.PrintStream.println	226 ms (0 %)	6 µs	35,870
at.fhv.dgr1992.test.Test_08_Only_using_threads_to_get_sensor_values_and_camera_image.main	197 ms (0 %)	197 ms	1
java.util.TimerThread.run	189 ms (0 %)	94,694 µs	2
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.senseAllTogether	150 ms (0 %)	23 µs	6,297
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.setMotorSpeeds	139 ms (0 %)	19 µs	7,174
coppelia.remoteApi.simxGetStringSignal	136 ms (0 %)	21 µs	6,297
java.lang.StringBuilder.append(double)	83,890 µs (0 %)	2 µs	28,696
java.awt.image.BufferedImage.<init>	60,889 µs (0 %)	50 µs	1,198
java.lang.System.currentTimeMillis	49,413 µs (0 %)	0 µs	57,392
java.lang.StringBuilder.<init>	38,199 µs (0 %)	0 µs	42,167
java.lang.StringBuilder.append(java.lang.String)	35,707 µs (0 %)	0 µs	48,464
coppelia.FloatWA.initArrayFromCharArray	31,777 µs (0 %)	5 µs	6,297
coppelia.FloatWA.<init>	28,287 µs (0 %)	1 µs	14,669
java.lang.Float.intBitsToFloat	21,229 µs (0 %)	0 µs	151,128
coppelia.CharWA.<init>	20,475 µs (0 %)	3 µs	6,297
java.lang.StringBuilder.toString	19,390 µs (0 %)	0 µs	42,167
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getDoubleValuesFromCharWA	18,546 µs (0 %)	2 µs	6,297
at.fhv.dgr1992.ePuck.EPuck.refreshSensorValuesAllTogether	17,730 µs (0 %)	2 µs	6,297
at.fhv.dgr1992.test.Test_08_Only_using_threads_to_get_sensor_values_and_camera_image.calculateMotorSpeed	16,693 µs (0 %)	2 µs	7,174
coppelia.FloatWA.getNewArray	16,563 µs (0 %)	13 µs	1,198
java.lang.Math.abs	15,205 µs (0 %)	0 µs	57,392
java.lang.StringBuilder.append(int)	12,476 µs (0 %)	1 µs	7,174
at.fhv.dgr1992.ePuck.SensorValueRefreshTask.run	11,463 µs (0 %)	1 µs	6,297
at.fhv.dgr1992.ePuck.EPuck.getCameraImage	8,747 µs (0 %)	1 µs	7,174
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.getValuesOfArray	6,816 µs (0 %)	0 µs	31,485
at.fhv.dgr1992.ePuck.CameraImageRefreshTask.run	6,684 µs (0 %)	5 µs	1,198
at.fhv.dgr1992.ePuck.EPuck.setCameraImage	6,142 µs (0 %)	5 µs	1,198
at.fhv.dgr1992.ePuck.EPuck.refreshSensorValues	5,628 µs (0 %)	0 µs	6,297
at.fhv.dgr1992.differentialWheels.Speed.<init>	4,943 µs (0 %)	0 µs	11,219
at.fhv.dgr1992.differentialWheels.Acceleration.<init>	4,859 µs (0 %)	0 µs	6,297
coppelia.CharWA.getNewArray	4,189 µs (0 %)	0 µs	6,297
at.fhv.dgr1992.ePuck.EPuck.getGroundSensorValues	3,863 µs (0 %)	0 µs	7,174
coppelia.FloatWA.getArray	3,655 µs (0 %)	0 µs	21,843
at.fhv.dgr1992.differentialWheels.WheelEncode.<init>	3,480 µs (0 %)	0 µs	6,297
at.fhv.dgr1992.ePuck.ePuckVRep.EPuckVRep.floatArrayToDoubleArray	2,734 µs (0 %)	0 µs	6,297
at.fhv.dgr1992.differentialWheels.Speed.getLeft	2,547 µs (0 %)	0 µs	14,348
coppelia.FloatWA.getLength	2,529 µs (0 %)	0 µs	7,174
at.fhv.dgr1992.differentialWheels.CameraImage.<init>	1,876 µs (0 %)	1 µs	1,198
at.fhv.dgr1992.differentialWheels.Speed.getRight	1,514 µs (0 %)	0 µs	10,304
coppelia.CharWA.getArray	1,299 µs (0 %)	0 µs	6,297

Figure C.15.: JProfiler results for test 08.



```
Loop runs: 7307
avg loop time: 83.37717257424387
avg get camera image call time: 0.004653072396332284
avg get ground sensor value call time: 0.0013685507048036129
avg set motor speed call time: 83.36116053099768
Loop runs: 7308
avg loop time: 83.37342638204707
avg get camera image call time: 0.004652435686918446
avg get ground sensor value call time: 0.0013683634373289546
avg set motor speed call time: 83.35741652983032
Loop runs: 7309
avg loop time: 83.3685866739636
avg get camera image call time: 0.004651799151730743
avg get ground sensor value call time: 0.0013681762210972772
avg set motor speed call time: 83.35257901217676
Loop runs: 7310
avg loop time: 83.38454172366622
avg get camera image call time: 0.004651162790697674
avg get ground sensor value call time: 0.0013679890560875513
avg set motor speed call time: 83.36853625170998
Loop runs: 7311
avg loop time: 83.37970181917659
avg get camera image call time: 0.004650526603747777
avg get ground sensor value call time: 0.001367801942278758
avg set motor speed call time: 83.36369853645192
```

Figure C.16.: Results for test 08 using the Java system method.