Direction Invariant Path Rover

Bernhard Mühlbacher Hannes Payer

Abstract

The Rover is an autonomous direction invariant pathfinder, based on the Lego Mindstorm System and is implemented in Java and Giotto. The challenge is to follow an arbitrary line and adapt the speed of the Rover to the color of the line. Additionally, the Rover should communicate with its environment using an infrared device.
Interpreting and executing Giotto code on the Lego Mindstorm RCX implies developing a customized lightweight Giotto virtual machine and an adapted Giotto compiler. These implementations can be re-used by other Giotto implementations on devices which offer for performance reasons just a small Java API.

1 Introduction

The goal of the Rover project was to develop a direction invariant pathfinder using the Lego Mindstorm System Hardware[1], the Java programming language and Giotto, a high-level programming language for control applications. Additionally, the color[1] of the path should determine the speed of the Rover and it should use the infrared port of the Lego Mindstorm RCX to communicate with its environment. A smart controller is needed to get direction invariance. There exist high demands for hardware and software.
In the following sections, the design and the implementation of the project are presented. In section 2, the Lego Mindstorm Hardware is introduced and the hardware design of the Rover is explained. The developed software is discussed in section 4 and the last two sections give a resume of the project.

2 Hardware

The Rover is fully composed of the Lego Mindstorm System Hardware. The sensors are:

- 2 light sensors
- 1 rotation sensor

and the actuators are

- 2 motors.

The two light sensors are needed to recognize the direction of the curve, and the two motors are used to react. To steer, the motors run with different speed, depending on the curve direction. To go right, the left motor has to run faster and vice versa. Therefore, no additional steering is necessary. In the front of the Rover is a small steering support unit

[1]Differentiation between two colors: black means fast speed, green means slow speed

which is able to rotate. This construction has one disadvantage: the speed of the motors is unbalanced and therefore the Rover begins to drift in one direction in the "drive straight forward" mode. To handle this, a differential gear and a rotation sensor is used to correct motor speed differences.

The construction plan is taken from [2] and modified by adding a second light sensor.

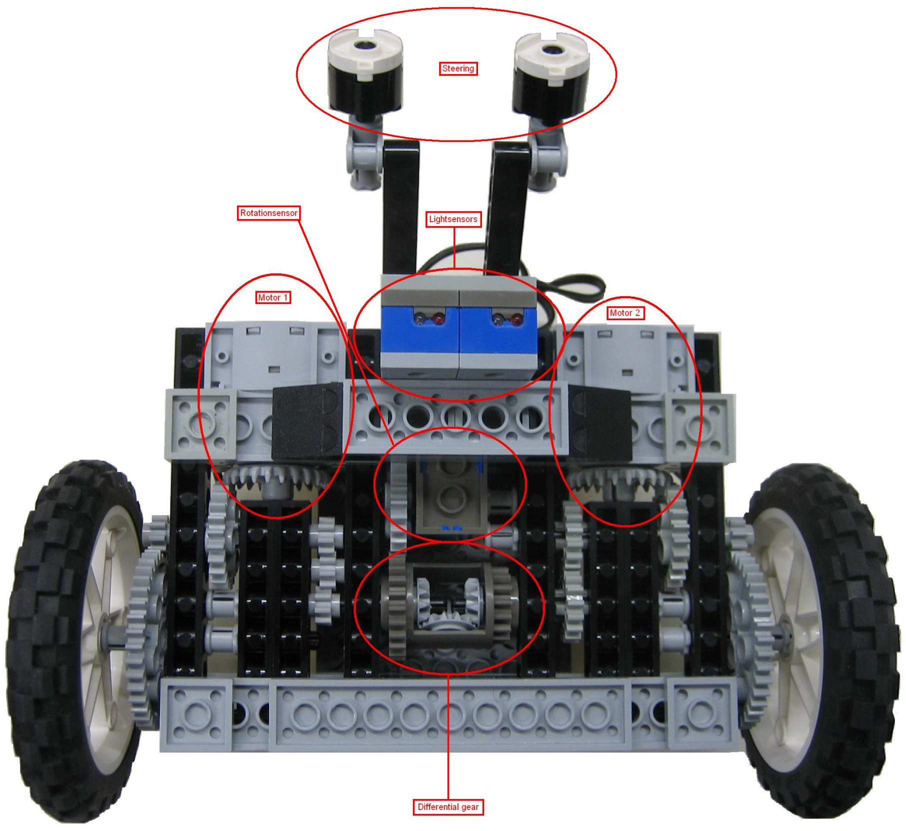

Figure 1. Rover bottom view

3 Implementations

The implementations are based on the [3] replacement firmware for the Lego Mindstorms RCX brick. It offers a Java virtual machine that fits within the 32KB on the RCX and comes with a slim Java API. The major drawback of the API is the lack of a file I/O subsystem. There exists no possibility to read some content from an external file.
The following section describes two different implementations of the Rover. The first one is a *straightforward* implementation of a simple controller, used to explore the functionality of the hardware and to achieve the project goal of an invariant path finder. The second one is a Giotto Compiler extension and a Giotto Virtual Machine implementation to consider the logical execution time[4] aspects.

3.1 leJOS

The leJOS API offers calls to the hardware units:

- motor: The motor speed is configurable with integers between $[0, 7]$. This implies that there is not much latitude for different speeds.
- light sensor: The light sensor returns luminance values. Black is a value between $[36, 43]$ and green represents a value between $[43, 48]$. The light sensor is very sensitive and depends on the room light. A 100% correct color classification is not possible.
- rotation sensor: The rotation sensor returns the angle of the path deviation.

3.2 Tasks

Three tasks are defined, which force the Rover to follow the path:

- Straightforward Task:
  - precondition: Both light sensors have to be on the path.
  - Both motors run with the same speed and the rotation sensor is used to correct motor-speed differences.
- Curve Task:
  - precondition: Only one light sensor is on the path.
  - One motor runs with normal speed, the other one with slower speed, with respect to the curve direction.
- Security Task:
  - precondition: Both light sensors are not on the path.
  - Circular Buffers[2] are used to analyze the light sensor history. A history size of 20 entries satisfies the Rover.

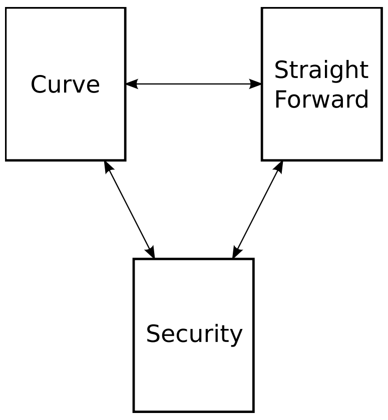

Figure 2. controller states

[2]Whenever one of the light sensor returns a value to the controller, this value is stored in the circular buffer of the light sensor.

3.3 Simple Controller

The *Simple Controller* implementation is very elementary. The Controller checks the input ports and executes one of the predefined tasks. The goal was to test the vision of a direction invariant path finder and to build a fundamental solution for the Giotto implementation.

3.4 Giotto

There was no need to implement a Giotto program, because in the "simple Controller" implementation are only three tasks and three modes. However, implementing Giotto on leJOS was a big goal and challenge to see, how this high-level programming language works for control applications.

The Rover Giotto program[3] in short:

- 3 modes
- 3 tasks
- 2 actuators (right wheel or motor 1, left wheel or motor 2)
- 3 sensors (light left, light right, rotation)
- 3 outputs to communicate between the tasks
- 3 mode switch drivers to switch between the modes.

The hovercraft example implementation in Giotto, which can be found in the Giotto tutorial [5], was useful to understand the basics of the programming language and what happens in the Giotto VM internals. All the hovercraft operation classes, which are needed for the simulation, are implemented and reused on the RCX for the Rover again. The strict memory boundary of the RCX and the missing file I/O subsystem leads to build a special *Giotto on leJOS* implementation. The output files of the Giotto compiler needs some modifications and a lightweight Giotto VM, which uses no file I/O, is needed too.

3.4.1 Compiler

The target system comes without a file I/O subsystem, therefore a complete image of the executable code for the RCX has to be created at compile time. The compiler has to create dynamically program specific handler classes. To create these classes, a menu "Write Handler Class" was added to the Giotto compiler tool. The "Show Ecode" output of the Giotto-Tool was the guideline for the handler classes design.

ECode Ecode is code, which is executed by the Embedded Machine and supervises the timing constraints. Further information can be found at [6].

Handler Classes There are 3 Handler classes. The first one is the *ECode.java* which holds a vector in which all instructions are stored in the right order. This means that the instructions are in the order of their execution. The second one is the *Porthandler.java* which holds all ports in a hash map and the keys of the ports are the port names. How

[3]For details please look at the rover05.giotto file on the Rover project webpage.

the ports get their names, why there are so many ports, and how the information flows is explained in figure 3.

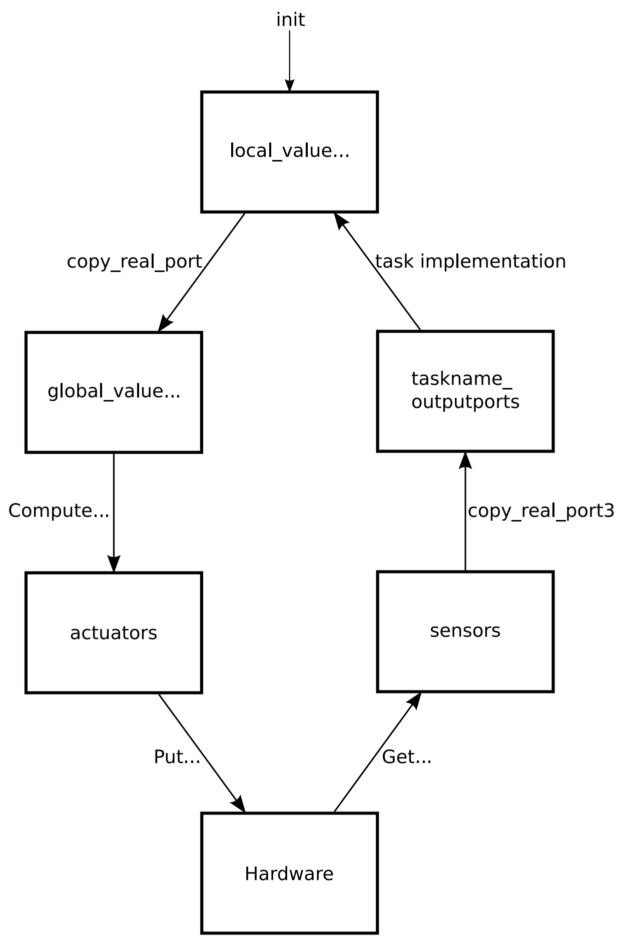

Figure 3. port information flow

The third one is the *OperationHandler.java*. As expected the OperationHandler holds all operations in a hash map and the keys are again their names. All the operations have to implement the abstract class *Operation* with the abstract method *int op(int pc, Vector params)*. The problem specific operation code has to be inserted in the method body. This basis offers the possibility, to implement a smart VM which is described in detail later in this section.

The “Apache Velocity Project” [7] is used to generate the handler classes dynamically. The dynamic classes are defined in specific template files. The following example shows the template file for the ECode class:

```
private static void initECode(){
     Vector tmp;
     #foreach( $value in $ecode )
          tmp = new Vector();
          #foreach( $elem in $value)
               tmp.addElement("$elem");
          #end
     code.addElement(tmp);
     #end
}
```

The result after compiling the *rover05.giotto* is the following for the ECode:

```
private static void initECode(){
     Vector tmp;
     tmp = new Vector();
     tmp.addElement("2");
     tmp.addElement("vm.functionality." +
                    "operation.copy_real_port");
     tmp.addElement("valueRightLight");
     code.addElement(tmp);

     tmp = new Vector();
     tmp.addElement("2");
     tmp.addElement("vm.functionality." +
                    "operation.ComputeLeftMotorPower");
     tmp.addElement("leftWheel");
     code.addElement(tmp);
     ...
}
```

The other templates are *OperationHandler.template* and *PortHandler.template.* The resulting java files are *OperationHandler.java* and *PortHandler.java.*

3.4.2 Virtual Machine

The interpreter uses the generated ECode class to execute the defined operations. The OperationHandler loads registered operation objects and the PortHandler is used to load registered ports. The operations and the tasks communicate via ports and the hardware specific calls are executed within the different operations.

The design of the interpreter is pretty easy and the dynamic is still kept. Just a few case differentiations are needed to execute the ecode. The *nop* command is just an increment of the program counter. All the *operation* commands are combined to a single case. Every operation is responsible for a correct program counter update. The *jump* instruction represents a program counter update and the *return* statement is not used.

```
Vector instruction;
while(pc < limit){
    instruction = eCode.getInstruction(pc);
    op = VMUtils.stringToInt((String)instruction.elementAt(0));

    if(op==0){ //nop
       pc++;
    }
    else if(op==1 || op==2 || op==3 || op==4){//operations
       operation = (Operation)oHandler.
                    getOperation((String)instruction.elementAt(1)
                       );
       pc = operation.op(pc, instruction);
    }
    else if(op==5){//jump
       pc = VMUtils.strintToInt((String)instruction.elementAt(1))
          ;
    }
    else if (op==6){//return
    }
}
```

4 Effort

The man hours of the project:

Hardware Design	10h
Simple Controller	52h
Giotto	16h
Giotto-Compiler	24h
velocity package	7h
Giotto-VM	82h
Total	191h

5 Conclusion

The experiment to build a direction invariant path Rover was successfully completed. Several tests on a testplattform acknowledged our project goal.
The communication with the Rovers environment is used for debugging concerns. The Rover sends messages about his task status to a second receiver RCX.

References

[1] Lego Mindstorm. *http://mindstorms.lego.com*.

[2] Stephan Höhrmann. Rover. *http://www.informatik.uni-kiel.de/inf/von-Hanxleden/mindstorms/Bauplan/Rover/Dateien/rover.pdf*, 2002.

[3] leJOS Software. *http://lejos.sourceforge.net*.

[4] C.M. Kirsch A. Ghosal, T.A. Henzinger and M.A.A. Sanvido. Event-driven programming with logical execution times. *Proc. International Workshop on Hybrid Systems: Computation and Control (HSCC), volume 2993 of LNCS,*, pages 357–371, 2004.

[5] M.A.A. Sanvido and Aaron Walburg. GiottoTutorial. 2004.

[6] Thomas A. Henzinger and Christoph M. Kirsch. The Embedded Machine: Predictable, Portable Real-Time Code. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326, 2002.

[7] The Apache Velocity Project. *http://velocity.apache.org*.