



^b
**UNIVERSITÄT
BERN**

A Polite Solution to Interact with EV3 Robots

Bachelor Thesis

Theodor Truffer
from
Kirchberg BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

September 6, 2016

Prof. Dr. Oscar Nierstrasz
Dr. Mircea Lungu, Jan Kurš
Software Composition Group
Institut für Informatik und angewandte Mathematik

University of Bern, Switzerland

Abstract

Lego Mindstorms is a combination of hardware and software to build and program a variety of different Lego robots. The Evolution 3 (short *EV3*) represents the third generation of these promising robots.

Besides Lego itself, plenty of other organisations, researchers and developers have designed software to interact with the EV3 robots, many of them with the target to teach Computational Thinking to amateur programmers, others to reach high functionality and open new possibilities.

But although there are already a lot of existing projects, there seems to be a gap between the simple and visual learning programs, and the richer programming environments.

Polite for EV3 closes this gap by combining the simple but nonetheless expressive object-oriented programming language *Polite* with powerful concepts like *State Machines* and *Real Time Programming*.

Contents

1	Introduction	4
2	Related Work	6
2.1	Lego Mindstorms	6
2.2	Phratch with JetStorm	7
2.3	Live Robot Programming	8
2.4	Others	8
3	The Gap	9
3.1	Visual Programming Environments	9
3.1.1	The Boundaries	10
3.2	JetStorm	10
3.2.1	The Limitations	10
3.3	Polite	11
3.3.1	Program Flow	12
3.4	Run Time Adaption	13
3.5	Summing Up	13
4	Closing the Gap	14
4.1	First Architectural Layer	14
4.2	The Interaction	15
4.2.1	JetStorm’s Architecture	15
4.2.2	Adapting JetStorm	16
4.2.3	Introducing PoliteVehicle	16
4.2.4	Initialization	17
4.2.5	Wrapping Methods	18
4.3	The Behavior	20
4.3.1	Finite State Machines	20
4.3.1.1	Using FSMs for Robotics	21
4.3.1.2	Mealy Machines	22
4.3.2	State Machine Architecture	22

4.3.2.1	States and Transitions	23
4.3.2.2	Special States	23
4.3.2.3	Epsilon Transitions	24
4.3.2.4	Wildcards	25
4.4	Abstraction	25
4.4.1	Nested State Machines	26
4.4.2	Implementation of Nested Machines	27
4.5	Computability	28
4.5.1	Context Matters	28
4.5.1.1	Global Access	29
4.5.1.2	Execution of Nested Machines	30
4.5.2	Turing Completeness	30
4.6	The Execution Loop	30
4.7	Run Time Programming	31
5	Validation	32
5.1	Turing Completeness	32
5.1.1	Turing Machines	32
5.1.2	Simulating Turing Machines	33
5.1.3	Simulation	33
5.2	Usability	35
6	Conclusion and Future Work	36
6.1	Asynchronous Processing	36
6.2	Extending PoliteVehicle	37
7	Anleitung zu wissenschaftlichen Arbeiten	38
7.1	Tutorial	38
7.1.1	Prerequisites	38
7.1.2	Installation	39
7.1.3	Start and Connect	39
7.1.4	First Steps	40
7.1.5	Robot Commands	41
7.1.6	Building a State Machine	42
7.1.7	Using the Context	43
7.1.8	Further Instructions	44

1

Introduction

Among the existing EV3 programming environments, two categories seem to become apparent: Learning Environments and Scientific Environments. The Learning Programs like Phratch or Lego Mindstorms' Software are user-friendly and easy to learn. Since their programming languages are purely visual, beginners understand them quickly. But at the same time, this simplicity brings a lot of constraints. With growing complexity, bigger programs quickly become confusing. Also, even if composing visual programming blocks is easy to learn, at some point the students will have to deal with writing code if they want to go deeper into programming.

The advantages and disadvantages of the Scientific Programs are the reverse of the Learning Programs. They offer great functionality with principles like nested state machines and live programming, or allow users and developers to write programs in different programming languages, but at the same time they are hard to learn for unexperienced programmers. *Polite for EV3* is a project which combines the benefits of both the approaches.

Mircea Lungu and Jan Kurš' paper *On Planning an Evaluation of the Impact of Identifier Names on the Readability and Maintainability of Programs* [1] discusses the new way of writing identifier names by allowing whitespaces (so called *sentenced case*), resulting in more intuitively readable programming languages. Out of this idea, they developed *Polite Smalltalk* [2] which extends the programming language *Smalltalk* by

introducing *sentence cased* writing of identifier names.

Polite for EV3 combines *Polite Smalltalk* with a strong back end based on concepts, best suitable for controlling robotics. Full computability and functionality should be reached, leaving the users with no constraints to implement their ideas, even with increasing complexity of the robots behavior.

This work focuses on the back end of *Polite for EV3*, whereas the graphical user interface and its interaction with the back end is discussed in Stefan Borer's work, *Lego Playground* [3].



Figure 1.1: The Robot Educator Vehicle

2

Related Work

There are already a number of projects which build interfaces between the user and the EV3 robots. This section will only focus on the most important ones, which are the ones that influenced this work in some noteworthy way.

2.1 Lego Mindstorms

Lego's EV3¹ aims to teach students subjects like computer science, scientific working, technology, engineering and mathematics. Therefore Lego's own developed software prioritizes a design which addresses a rather younger group of users. On their official website² it says:

Our software is optimised for classroom usage and follows the very latest developments in intuitive software design, which results in an extremely user-friendly interface.

The software consists of two main parts: a section for data logging and another for programming. In the former one, the user is able to observe all sensors of a connected EV3 robot in real time. Students can for example take a EV3 brick with a connected temperature sensor and then constantly watch and log the temperature of an object, interesting to their scientific work.

Inside the programming workspace users are able to develop programs using the Lego

¹<http://education.lego.com>

²<http://education.lego.com>

Mindstorms visual programming language. They can solve different exercises, create own projects and load the programs on the brick to execute them.

The Lego Mindstorms programming language is based on LabVIEW³ and uses a graphical programming syntax, so there's no actual code being written while creating programs. The user designs the desired program flow using boxes of different colors, each color representing a different group of elements: orange boxes for flow control, green boxes for motor control and so on. This kind of syntax allows even inexperienced programmers to implement simple tasks. Almost everything is done via drag and drop so there's no risk of creating syntax errors. Also, the user gets a feeling for how programming works, without having to learn a specific syntax.

The EV3 software is available for Windows and OS X and can be downloaded for free from Lego's official website⁴.

2.2 Phratch with JetStorm

Phratch⁵ is a project built on the visual programming environment Scratch⁶, where the user combines blocks via drag and drop to create a program. The important difference is, Phratch can be extended easily and new blocks can be added. Phratch is running on Pharo⁷ and thus written in the programming language Smalltalk⁸. A new block can be added by writing a Smalltalk method with a specific annotation⁹. The annotation notifies Phratch that this method should be a block.

Phratch alone doesn't provide the functionality to interact with an EV3 brick. But an extension called *JetStormForPhratch*[4] introduces new blocks to communicate with EV3 robots, by plugging in the JetStorm library. The JetStorm library is a Pharo framework which supports every interaction with EV3 robots. It offers Smalltalk methods for establishing connection, initializing and controlling motors, reading sensors, even drawing on the EV3 Brick's screen.

³<http://www.ni.com/labview>

⁴<http://www.lego.com/en-us/mindstorms/downloads>

⁵<http://www.phratch.com>

⁶<https://scratch.mit.edu/>

⁷<http://pharo.org>

⁸<http://pharo.org/>

⁹Annotations in Smalltalk are called *pragmas*

2.3 Live Robot Programming

Live Robot Programming[5] (LRP) is a programming language developed to control the behavior of robots by using the concept of nested state machines. In its GUI there's a canvas right next to the programming workspace. As the name implies, the programs can be edited live. So while defining the state machines inside the workspace, the states and transitions are being drawn inside the canvas, symbolised as vertices and edges of a directed graph. Instead of having to restart the state machines after a change, the states and transitions can be changed while the machine is running.

LRP can be used with different infrastructures of robots. There's also an AddOn to control Lego's EV3 Bricks via JetStorm.

2.4 Others

There are a number of other interesting projects. Libraries like LeJOS¹⁰ run a Java Virtual Machine on an EV3 Brick and thus allow developers to program the Brick in Java. Other projects do analogous things with Python or C#. Searching on GitHub¹¹ with the query string "EV3" will return projects related to this subject.

¹⁰<http://www.lejos.com>

¹¹<http://www.github.com>

3

The Gap

There are already many different projects building interfaces between the users and Lego's EV3 robots. Different programming languages and concepts have been used to implement. So at this point one might ask *why does there have to be another project?*

This chapter will illustrate the gap between visual, user-friendly programs like *Lego Mindstorm* and highly functional programs and APIs like *JetStorm*. In addition, it will evaluate problems of controlling robotics and set goals for a new approach.

3.1 Visual Programming Environments

A Visual programming language, or visual programming environment, allows developers to *build programs by using graphical elements*. Each graphical Element abstracts some programming code. E.g. there could be a graphical *if/else*-block which can be dragged into a workspace. This block can then be given a condition. More blocks can be dragged to the *if*-section or to the *else*-section of the *if/else*-block, which will be executed when the condition is true or false, respectively. Entire programs can be built and executed by composing these blocks.

Such programming languages are *user-friendly*. All possible functions are abstracted into blocks and users can at some place visually observe an overview of these blocks, so they receive an overview of all possible commands in this programming environment. This

way, they don't have to learn the syntax, like in a textual programming language. Also, through the abstraction of the code blocks *it is not possible to produce syntax errors*, which are often an obstacle when learning how to program.

3.1.1 The Boundaries

Programs like *Lego Mindstorm* and *Phratch* are built on such visual programming environment. They both aim to be as user-friendly as possible and *target a rather younger audience*. On their official website¹, Scratch says to be "*..designed especially for ages 8 to 16, ..*". On one hand this indicates good learnability but at the same time hints that these programs are not built for more complex, scientific use.

The simplicity of *visual programming brings constraints to the functionality* of programs like *Phratch* and *Lego Mindstorm*. Users are bound to the visual elements provided by the environment and thus not able to get use of the power and scalability of a common programming language. Therefore, to target a more professional and scientific audience, a project should be based on code to avoid restrictions and offer a greater range of possibilities.

3.2 JetStorm

In contrast to the projects based on visual programming stands JetStorm, an API built to interact with the EV3 bricks. JetStorm takes care of establishing a connection to the robots and provides functions for every possible command supported by Lego's EV3.

3.2.1 The Limitations

The question might appear, *why does JetStorm need to be adapted*, since every possible interaction with an EV3 robot seems to be covered by the API. The problem is its verbosity. To emphasize this, some examples of what steps are necessary to actually communicate with the robot are following.

The first example shows how to *read the connected color sensor*. To do this, we need to execute the following steps:

First, we have to initialize an `Ev3Vehicle` object and connect it to our EV3 brick. This is pretty straight forward.

```
ev3Vehicle := Ev3Vehicle newIp: '192.168.29.42' daisyChain: #EV3
```

¹<https://scratch.mit.edu/about>

Second, the vehicle needs to initialize its sensors via the following line:

```
ev3Vehicle detectSensors
```

Third, every sensor has multiple modes, so in this case if we want to get the actual color that the sensor is reading, we have to set the sensors mode to #Mode2. For this, we also have to know to which port (sensor1 - sensor4) the color sensor is connected.

```
ev3Vehicle sensor2 setMode: #Mode2
```

Finally, we want to read the sensor's value:

```
ev3Vehicle sensor2 read
```

This will return the code of the color which the color sensor is reading at this moment. So for example if the sensor is on a green surface, this line will return 3.

Of course, most of these lines of code only have to be executed once, at the beginning of a program. Still, without a detailed instruction, a beginner would never be able to figure this out. Also, an API where the code for reading the color is *sensor2 read* and the response is 3 is too complex for the requirements we set in simplicity and user-friendliness.

Another example is this line of code, required to make the robot *turn 90 degrees to its left*:

```
ev3Vehicle motorSync startSpeed: 20 turnRatio: -200 degrees: 180
```

This is very complicated for such a simple command and must be wrapped to provide a nice interface.

JetStorm is a powerful API, yet a *higher level of abstraction* must be reached for a simple usage.

3.3 Polite

In 2013, Jan Kurs and Mircea Lungu argued about the importance of identifier names [1]. Until then, there only seemed to be two absolute possibilities how to write identifiers: Either CamelCaseIdentifiers or underscore_identifiers. But they came up with a whole new idea. By allowing whitespaces in the identifier names, a new practice of writing code develops. *Sentence case* or *phrase case* is a way of writing identifiers that is much closer to our natural language and thus increases the readability and maintainability of source code.

Out of this idea, a new programming language came into existence. *Polite Smalltalk* [2]

is a synthesis of the object-oriented programming language Smalltalk with the newly invented sentence case writing. Since Smalltalk itself was developed as a language with *high readability*, Polite Smalltalk even goes one step further in this direction and allows and encourages programmers to write code in a very nice and readable way.

Furthermore, since Polite is built on Smalltalk it *inherits the functionality*. Therefore, Polite Smalltalk combines two important characteristics: simplicity and functionality.

Polite seems to be a good solution to build an interface to the Lego devices. Still, the usage of this programming language without additional concepts *will lead to problems* when building programs for the robots. The next section will take a closer look at these problems.

3.3.1 Program Flow

Controlling robotics can quickly reach a high complexity. With devices like the EV3 robots, the goal is often to teach the robots certain patterns of, sometimes humanoid, behavior. By using their sensors, the robots gain the *possibility to perceive their environment*. They are able to detect physical objects and even colors. Thus, users are able to make a robot perform different actions depending on its surroundings. The robots behavior can be constructed to *observe and react* to events triggered by environmental inputs.

It should be safe to say that programming such behavior is a common use case. Imagine building such a program with Polite.

As in every common object-oriented programming language, the program flow in Polite can be designed by using the *usual control flow statements*, such as *if-else-statements* or *for- and while-loops*. The execution of a program can take different turns and directions based on these statements.

To program the robot's behavior as explained above, some kind of *rules* have to be defined. E.g. a certain input of one of the robot's sensors leads to some action, like starting or stopping a motor. Furthermore, such defined reactions may depend on the current point of execution of the program. E.g. the robot might not have the same reaction each time it reaches an obstacle, but act differently depending on what happened before, or what kind of *state* it is currently in.

Defining such rules, states or reactions to inputs only by using the usual control flow statements will soon become painful when the robot's behavior is being extended. A programmer will easily get lost in constructing different loops, thereby running the *risk of losing the overview* of the program.

Without using additional patterns to control the EV3 devices, programs will quickly

tend to acquire difficult structures that are hard to understand and extend. The usual program flow does not seem to satisfy when it is used to control robotics. Although a programming language like Polite provides full computability, some way of abstraction is desired for a convenient interaction with the robots.

3.4 Run Time Adaption

As an additional function, we set the goal to allow users to adapt a program at run time. I.e., a program should be changeable during its execution. Thus, the robot's actions and behavior would be adjustable while a task is being executed.

Since it is not possible to interfere in a currently processing program in Polite, the realization of run time adaption also requires additional concepts or patterns on top of the programming language.

3.5 Summing Up

The evaluation of the problems and goals can be summarized as followed:

- *Visual programming languages* don't scale well for a more complex use with robotics. They are too limited to target a scientific and professional audience.
- *JetStorm's* level of abstraction is too low to be used directly and without an additional layer of abstraction. Many commands of the JetStorm API are hardly readable and require too much time to learn.
- *Polite* is a good candidate for a textual programming language. However, the traditional control flow approaches are too limited and are not suitable for programming complicated robot behaviors.

The conclusion is, although there are strong projects to work with, there is still *the need for a new approach*. A program providing a highly functional back end while at the same time remaining user-friendly. To manage the high complexity that controlling robotics implicate, an architecture built on strong concepts is required. The integration of features like the support of real time adaption helps to additionally improve the functionality of the system.

4

Closing the Gap

Now that the goals for this work have been set by analyzing the existing projects, the architecture can be designed to ensure the desired characteristics. Different concepts and patterns will be introduced and composed to build an environment able to *fill the evaluated gap and provide a strong back end*.

4.1 First Architectural Layer

The environment's highest level of abstraction, the *User Interface Layer*, uses Polite Smalltalk as programming language and consists of a GUI built on *Spec*, a UI library for Smalltalk. The UI is the focus of another thesis [3].

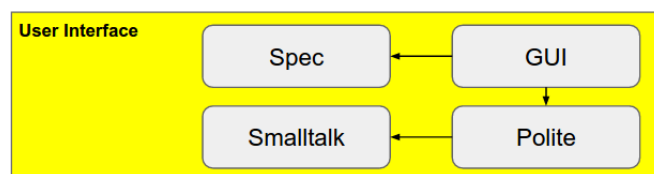


Figure 4.1: Interface Layer

4.2 The Interaction

For the interaction with the EV3 robots, a good API *written in Smalltalk* will be needed. In sections 2.2 and 2.3 we discussed the projects Phratch and Live Robot Programming, which are both written in Smalltalk and both are using JetStorm as the API.

The use of JetStorm is reasonable since it builds an interface and manages all communication to the EV3 Bricks in an object-oriented way.

4.2.1 JetStorm's Architecture

Lego's EV3 Brick is running on a Linux-based operating system. Besides USB and Bluetooth, the brick also supports communication over WiFi, which can be established via TCP. Commands and responses are then sent and received in form of bytestrings. JetStorm takes care of two main tasks:

1. Establishing and maintaining a connection to the EV3 brick
2. Translating commands into the correspondent bytestring and sending them to the brick, and accordingly translating received responses

To handle this, JetStorm is built on an object-oriented architecture, consisting five main classes (described in Jannik Laval's technical report [4]):

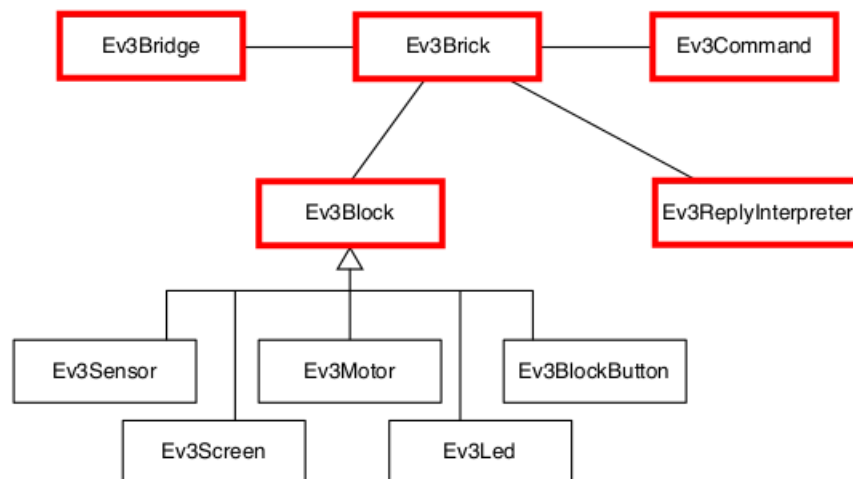


Figure 4.2: JetStorm architecture

The class *EV3Brick* builds the entry point. It controls the rest of the system. More important for this work will be its subclass *Ev3Vehicle*. In addition to a brick, a vehicle is able to synchronize two motors (e.g. left wheel, right wheel), so that no delay is generated when those two motors are addressed simultaneously.

The *EV3Bridge* is responsible for the connection to the EV3 brick. It contains the script to establish a TCP connection and is accessible via the EV3Brick-object.

The class *Ev3Block* handles the communication with the hardware. Each instance of a subclass of *Ev3Block* represents a piece of hardware on the connected robot. For example a subclass of *Ev3Block* would be *Ev3ColorSensor* or *Ev3MotorSync*. Each subclass provides methods for every possible command that can be sent to the corresponding piece of hardware.

The classes *Ev3Command* and *Ev3ReplyInterpreter* are helper classes which build the right byte array for a command and interpret the data received from the robot. Every possible command and every possible answer is supported.

4.2.2 Adapting JetStorm

For a more user-friendly interaction, an additional layer of abstraction must be implemented.

Lego Mindstorm provides a great range of different robots to construct. The following section will introduce a class to build an interface to one kind of these robots, the classical *Robot Educator Vehicle* from the *LEGO MINDSTORMS EV3 Education Core Set* (Figure 1.1). The introduced class will be *adjusted to the hardware of only this kind of robot*. To interact with other types, new classes need to be implemented (See also: *Extending PoliteVehicle* [6.2]).

The Robot Educator Vehicle is a robot with two wheels, therefore we will use *Ev3Brick*'s subclass *Ev3Vehicle*, so we can benefit from its ability to synchronize two motors. And since every action with JetStorm happens through one class, one class will also suffice to wrap the whole API. We'll call this class *PoliteVehicle*.

4.2.3 Introducing PoliteVehicle

The *PoliteVehicle* class is responsible for two things: It wraps the complicated commands of the JetStorm API into *better-named methods* and simplifies the *initialization* of a robot object.

To do so, the `PoliteVehicle` object contains an `Ev3Vehicle` as an instance variable, through which it is able to communicate with the robot. Furthermore, it has an instance variable for each sensor and motor, which are a direct reference to the sensors and motors inside the `Ev3Vehicle` object. But while in the `Ev3Vehicle` the sensors are called `sensor1 – sensor4` and the motors `motorA – motorD`, the `PoliteVehicle` *detects and identifies the sensors and motors* at initialization and stores them in the corresponding variables, called `colorSensor`, `rightWheel` and so on.

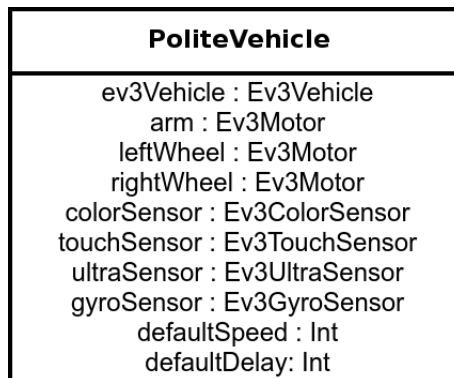


Figure 4.3: The `PoliteVehicle` object

`PoliteVehicle` contains two integer variables, *defaultSpeed* and *defaultDelay*, which are set to provide simple methods. For example, to get the robot start driving forward at a specific speed, the message *startAtSpeed:* has to be sent to the *motorSync* object inside the `Ev3Vehicle`. With the aid of *defaultSpeed*, `PoliteVehicle` wraps this message up as follows:

```
driveForward
  ev3Vehicle motorSync startAtSpeed: defaultSpeed.
```

4.2.4 Initialization

With the `Ev3Vehicle` of the `JetStorm` API one has to go through a number of steps to create a robot object. Therefore, `PoliteVehicle` aims to *automate* this long initialization process.

```
initialize
  self ev3Vehicle: (Ev3Vehicle newIp: (self class defaultIp)
    daisyChain: #EV3).
  self ev3Vehicle detectSensors.
  self ev3Vehicle syncMotorsLeft: (self ev3Vehicle motorB)
    right: (self ev3Vehicle motorC).
  self arm: self ev3Vehicle motorA.
```

```
self initializeSensors.
self defaultSpeed: 20.
self defaultDelay: 2.
```

Note that the *arm* is always *motorA*, the *leftWheel* always *motorB* and the *rightWheel* *motorC*. That is how the motors are connected to the ports if you build the robot according to the manual for the *Robot Educator Vehicle*. And unlike the sensors, the motors are not able to detect to what object they are connected to, so there is no way of automating the initialization of these variables without hardcoding them to these three motor ports. However it is possible to change the assignment of these variables manually after the initialization.

As mentioned above, the *sensors have the ability to identify themselves*, which makes it possible to assign all connected sensors correctly to the instance variables, without having to do anything manually.

```
initializeSensors
  self assignSensor: self ev3Vehicle sensor1.
  self assignSensor: self ev3Vehicle sensor2.
  self assignSensor: self ev3Vehicle sensor3.
  self assignSensor: self ev3Vehicle sensor4.

  self colorSensor setMode: #Mode2.
  self gyroSensor setMode: #Mode0.
  self ultraSensor setMode: #Mode0.
```

```
assignSensor: aSensor
  (aSensor getSensorType = 'Ultrasonic')
    ifTrue: [ self ultraSensor: aSensor ].
  (aSensor getSensorType = 'Color')
    ifTrue: [ self colorSensor: aSensor ].
  (aSensor getSensorType = 'Gyro')
    ifTrue: [ self gyroSensor: aSensor ].
  (aSensor getSensorType = 'Touch')
    ifTrue: [ self touchSensor: aSensor ].
```

4.2.5 Wrapping Methods

This section gives a quick overview of the *abstracted methods* provided by the *PoliteVehicle* object. The most important robot actions are implemented, yet there is a long list of possible commands which are not supported (see 6.2 *Extending PoliteVehicle*).

After actions like *turnLeft* there has to be a delay, because the control flow has to

be stalled until the execution of the action is over. This is what the instance variable *defaultDelay* is used for.

- *driveForward, driveBackward*. Starts driving forward/backwards at the default speed.
- *stop*. Stops the robots motors.
- *turnLeft, turnLeft:, turnRight, turnRight:.* Turns the robot to the left/right at the defaultSpeed and afterwards waits for defaultDelay-seconds. The amount of degrees to be turned can be passed as parameter, the default is 90 degrees.
- *turnRandom, turnRandomBetween:And:.* Turns randomly to the left or to the right. The range of degrees can be passed as parameter.
- *beep, beepLong*. Let the robot make a short or long beeping sound.
- *color*. Reads the color sensor and returns the name of the color as String. There are seven different possible colors: black, blue, green, yellow, red, white and brown.
- *distance*. Reads the ultrasonic sensor and returns its value. The returned number is the number of millimeters to an obstacle in front of the robot.
- *isTouched*. Returns the value of the touchsensor; true if the sensor is touching anything, false otherwise.
- *isConnected*. Asks if the object is currently connected to the Ev3Brick.

In addition, there are *getters* and *setters* for all instance variables which can be used to access all methods of the JetStorm API. However for the sake of convenience, for inexperienced programmers it should be sufficient to go with the methods provided by the PoliteVehicle object.

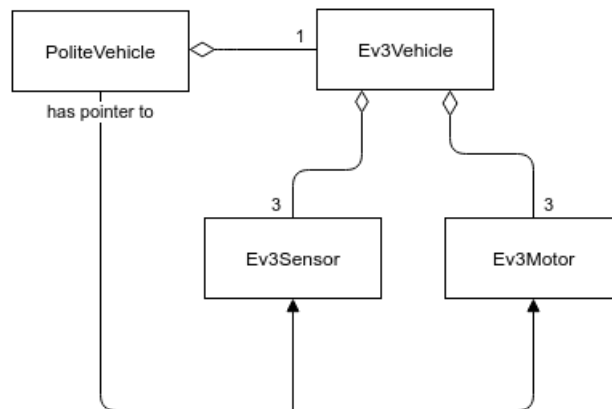


Figure 4.4: Architecture of adapted JetStorm API

4.3 The Behavior

Section [3.3.1] showed that there is the need for additional concepts or patterns to allow users to keep an overview of growing programs and robot's behaviors. The following sections will introduce such concepts and compose it to an architecture which fits the need to control robotics in a more natural way.

4.3.1 Finite State Machines

The pattern of Finite State Machines has often been used before to control robotics.

There are many ways of modeling the behavior of systems, and the use of state machines is one of the oldest and best known. State machines allow us to think about the state of a system at a particular point in time and characterize the behavior of the system based on that state.[6]

A finite state machine consists of a *finite set of states and transitions*. The machine is always in exactly one state, called the *current state*. The transitions (or *transition function*) are triggering events, which map current states and inputs to a successive state. Additionally, one of the state is to be defined as the *start-state*, in which the machine's execution begins.

FSMs are often visualized as in Figure 4.5, where the vertices represent the states and the edges represent the transitions of the state machine. Vertices with an incoming arrow without originating state symbolise start-states, while vertices drawn with a double-circle represent end-states.

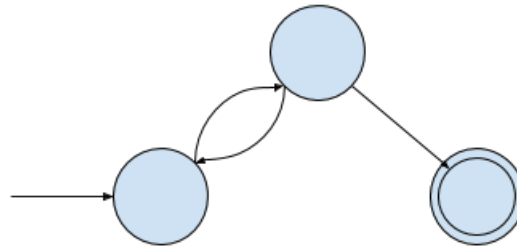
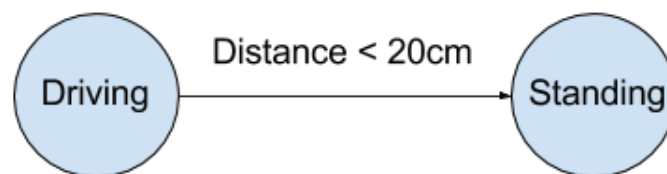


Figure 4.5: Visualization of a Finite State Machine

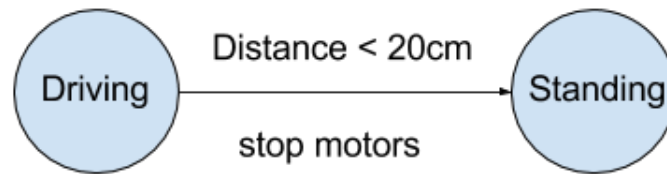
4.3.1.1 Using FSMs for Robotics

A reason to use this pattern for robotics is that programming based on states and transitions seems to be a natural way of controlling a robot's behavior: Depending on the current state, a robot can react to events and conditions.

E.g., a robot could be in the state *"Driving"*. To prevent the robot from driving into an obstacle, a transition with the condition *"Distance Sensor < 20cm"* leads to another state called *"Standing"*.



So when the robot is in the state *"Driving"* and it approaches an obstacle, the transition with the condition *"Distance Sensor < 20cm"* triggers and the robot changes its state to *"Standing"*. But with a (snippet of a) state machine like this, the robot will still be driving into the obstacle: The robot, or more precisely, the state machine has changed its state to *"Standing"* but since a *state is a purely logical element*, no actual stopping of the motors has been performed. Therefore, one important element is still missing: The *Action*. I.e., we would want to execute *"stop motors"* while switching from *"Driving"* to *"Standing"*.



4.3.1.2 Mealy Machines

The kind of a state machine where the transition not only triggers the changeover to another state, but also produces an output (in this case an *Action*) is called a *Mealy machine*.

The formal definition of a *Mealy machine*[6]:

A Mealy machine is a 6-tuple $(S, S_0, \Sigma, \Lambda, T, G)$ consisting of the following:

- *a finite set of states S*
- *a start state (also called initial state) S_0 which is an element of S*
- *a finite set called the input alphabet Σ*
- *a finite set called the output alphabet Λ*
- *a transition function $T : S \times \Sigma \rightarrow S$ mapping pairs of a state and an input symbol to the corresponding next state.*
- *an output function $G : S \times \Sigma \rightarrow \Lambda$ mapping pairs of a state and an input symbol to the corresponding output symbol.*

In some formulations, the transition and output functions are coalesced into a single function $T : S \times \Sigma \rightarrow S \times \Lambda$.

Since we will need an output function (whether it is integrated in the transition function or not), the model of a *Mealy machine* is appropriate for controlling the behavior of the EV3 robots.

4.3.2 State Machine Architecture

Concerning the concept of the finite state machines discussed in the previous section, the goal is to construct a clean architecture in an object-oriented way. We're beginning with the very basic objects of a FSM: States and transitions.

4.3.2.1 States and Transitions

To get an object-oriented architecture, there must clearly be a *State* object and a *Transition* object. A state doesn't really do anything but being a state, so for now the attribute *Name* (String) to identify a state suffices. A transition consists of a *condition* and an *action*. For the sake of convenience, those two attributes are executable Blocks. That way, the processor can simply execute the condition-block and if it returns *TRUE*, execute the action-block.

Now a transition leads from one state to another. There are several possible ways to create this connection. To find the best, we'll have a look at the execution of a state machine:

The machine is always in exactly one state, the current state. The possible triggering transitions are the ones originating from this state, all others can be ignored at this moment. So for the executing entity, the most simple way is to load these transitions directly from the current state. Therefore, it seems apparent that the *state object contains its outgoing transition objects*.

If a transition triggers, the transition's destination state becomes the current state. So again, for a clean control flow, we *store the successive state directly as an instance variable of a transition*.

Like this, there's no need for arrays or lists of states and transitions. Still, the whole state machine can be constructed by starting at the first state and going through all following states and transitions.

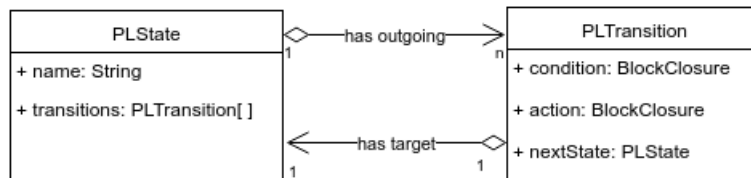


Figure 4.6: State Machine Architecture so far

4.3.2.2 Special States

A finite state machine not only consists of normal states but has two special kinds of states: A *start state* and (optionally) one or multiple *end states*. There must definitely be a *start state object* to be the entry point of a state machine. To keep a slim architecture, there will be *no kind of state machine object*. Therefore, a start state pretty much defines the whole state machine: Following the outgoing transitions and destination states, the

machine can be reconstructed. The *start state object* obviously inherits from the *state object*. It has two additional instance variables, a `String` called *machineName* and an `OrderedCollection` called *contextVars*. Further explanations to this will follow in the section *Context Matters [4.5.1]*.

For the end state(s), there could of course also be defined a new object, inheriting from the *state object*. But with a closer look, a simpler solution comes to mind: An end state has no outgoing transitions, which would make no sense, since the machine stops at an end state. Further, if some state has no outgoing transitions, the machine stops at this state, hence making it an end state. So to not make it unnecessarily complicated, we chose to have *no end state object*, but *define all states without outgoing transition to be end states*.

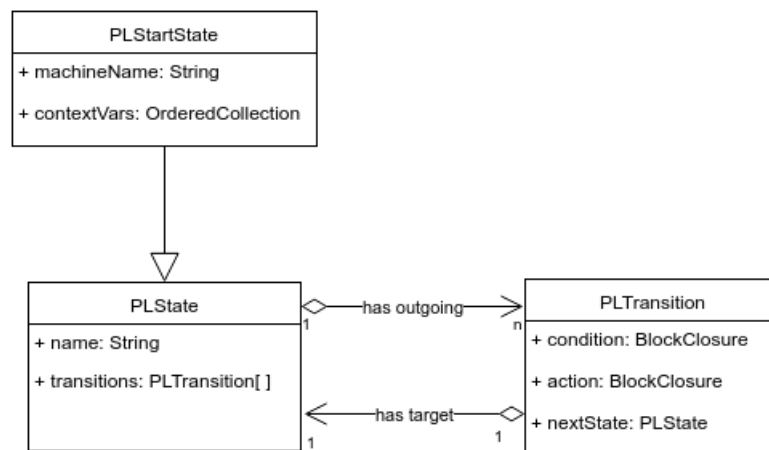


Figure 4.7: Introduced `PLStartState`

4.3.2.3 Epsilon Transitions

An epsilon transition is a kind of a transition where the condition is always true. So when entering a state with an epsilon transition, *this transition will immediately be triggered*. This makes sense for example, if the machine's execution begins and the robot should first of all start driving forward. So this transition should be triggered without any condition.

It may occur to some that the need for an implementation of an Epsilon Transition object is questionable, since it is just a transition with an always true condition.

A common use case for epsilon transitions is the initial execution of actions: often the robot should execute an action like starting a motor at the very beginning of the execution of a state machine. An epsilon transition originating from the start-state will be triggered immediately after starting the machine and can thereby be used to execute such initial

actions. Therefore it seems to make sense to integrate this entity into our system as well. Besides, Epsilon Transitions will be hidden in the backend, so they won't affect the simplicity of usage.

4.3.2.4 Wildcards

LRP [2.3] uses an additional feature, called *wildcards*. A wildcard is a transition with no origin state, meaning that this transition can be triggered regardless of which state the FSM is currently in. A wildcard is an *abstraction of multiple transitions* with the same condition, action and target state, but with every possible state as origin state. Therefore it does not change the model and behavior of the state machine, but simplifies the programming interface.

A use case would be the following: The programmer wants to stop a program by entering an end state as soon as a timer expires, no matter which state the FSM is in. This case would need one transition for each state in the FSM, but can be implemented with only one wildcard.

Therefore wildcards are a practical tool which will help to reach the desired level of abstraction.

The implementation is pretty straightforward, in fact: since a transition does not know its origin state, we can *use the transition class to implement wildcards*. We made the wildcard object inherit from the transition object, since from a logical point of view, a wildcard is kind of a transition. Also we modeled the situation where a wildcard can have an *empty next state*. If a wildcard with an empty next state is triggered, the state machine returns to the same state as it was before the execution of the wildcard.

The more important difference between wildcards and normal transitions is *where they are being stored*. Normal transitions are stored inside their origin state, whereas wildcards do not have an origin state, or rather, have every state of the state machine as origin state. Obviously, storing a transition inside every of these states is not a good solution. The problem is solved by storing the wildcards in a global variable. The details of this solution are in the section *Context Matters* [4.5.1].

4.4 Abstraction

As soon as a programmer goes beyond very simple and trivial robot behaviors, the state machines quickly grow bigger with a long list of states and transitions. To avoid losing the overview, the possibility to *encapsulate and combine state machines* needs to be offered.

To provide this possibility, we introduce the concept of *Nested State Machines*.

4.4.1 Nested State Machines

In a common programming language with functions, the program flow often looks like this: inside one function, another function is invoked. After the termination of this second function, the program returns to the upper function and continues executing the next lines of code.

With nested state machines it is the same: Inside a running state machine, a previously stored state machine can be started. As soon as this machine reaches an end-state, the upper state machine continues to execute.

Of course this can be carried on with nested machines inside nested machines, building a *hierarchy of machine executions*, equivalent to a *call stack* of functions.

Like this, even very complex state machines can be abstracted and stored, to then be called inside other state machines.

Usually nested machines are associated with states, meaning that a state machine can be abstracted as a state. The decision not to have a state machine object would make such an abstraction difficult, so we will introduce another approach: the execution of a nested machine will take place as the *action* of a *transition*.

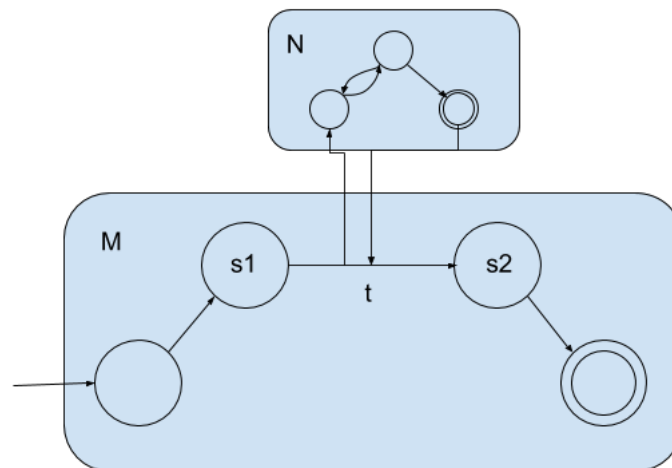


Figure 4.8: Execution of a Nested State Machine

For example, let there be a state *machine* M with a *transition* t which has *state* $s1$

as origin, *state s2* as its next state and the execution of a nested *machine N* as action (Figure 4.4.2). That way, when transition *t* is triggered, the execution of the machine *M* stalls between state *s1* and state *s2*, and the execution continues at the nested machine *N*'s start state. After an end-state of machine *N* is reached, the upper state machine *M* enters state *s2* and continues its execution.

4.4.2 Implementation of Nested Machines

A state machine can be executed by invoking `PLProcessor>>execute`:¹ and passing the `PLStartState` of the machine as a parameter.

To *nest* a state machine, we create a method on the class side of *MyScripts*, e.g. `MyScripts>>executeNestedMachine`². The created method contains the code to *create a machine* and after the creation, *executes the machine* by calling `PLProcessor>>execute`.

We then *set this method as a transition's action*. Since we created the method on the class side, we don't have to initialize an object first, but can simply call '*MyScripts, execute nested machine*' inside the Action-Block. When this transition triggers, the nested machine will be created and executed, and after its termination, the execution of the upper machine will continue.

```

executeNestedMachine
"this method executes a nested state machine"
| start |
start := PLStartState newCalled: 'start' machineCalled: 'Nested Machine'.

"create the states and transitions here ..."

PLProcessor execute: start.

```

108

Figure 4.9: Method to create and execute a nested machine

The graphical user interface of *Polite for EV3* provides an automated creation of methods for nested machines. Users are able to store methods into, and load methods from the *MyScripts* class via graphical elements.

¹The class `PLProcessor` will be introduced and explained in section 4.6.

²The class *MyScripts* is a class explicitly introduced to store methods which execute nested machines.

4.5 Computability

To have a powerful, fully functional architecture, it must be guaranteed that every desired program to control the robot can be developed with the used concepts. Or in other words, we must be able to *construct every computable function*.

The pattern of the Mealy machine together with nested machines does not yet fulfill this requirement. Consider this problem:

The Lego robot is equipped with a color sensor pointing to the floor. On the floor in front of the robot is an arbitrary long line of tiles. Each tile is either blue or yellow. A program should be able to let the robot drive over such a line and at the end of the line decide if the numbers of blue tiles and yellow tiles were equal or not. Based on this decision it will execute some further action.

This problem is not solvable with a Mealy machine. There is no kind of variable to store the number of detected blue or yellow tiles, the only way to remember the amount being to construct more states. But since there is a finite number of states and the number of tiles is arbitrary, there will always be a longer line which then is undecidable for the program.

4.5.1 Context Matters

Therefore, there has to be a way to store variables in the context of a state machine to guarantee full computability. Furthermore, we don't want to store a wildcard in every possible state which would create a great overhead. So a new object is needed, which we will call *PLContext*.

PLContext is a *global data class* which stores all information needed in the context of a state machine. This will be: *wildcards, variables and timers*. Besides these instance variables, the context supports methods to handle these variables, for example starting or resetting timers, storing and receiving variables and so on.

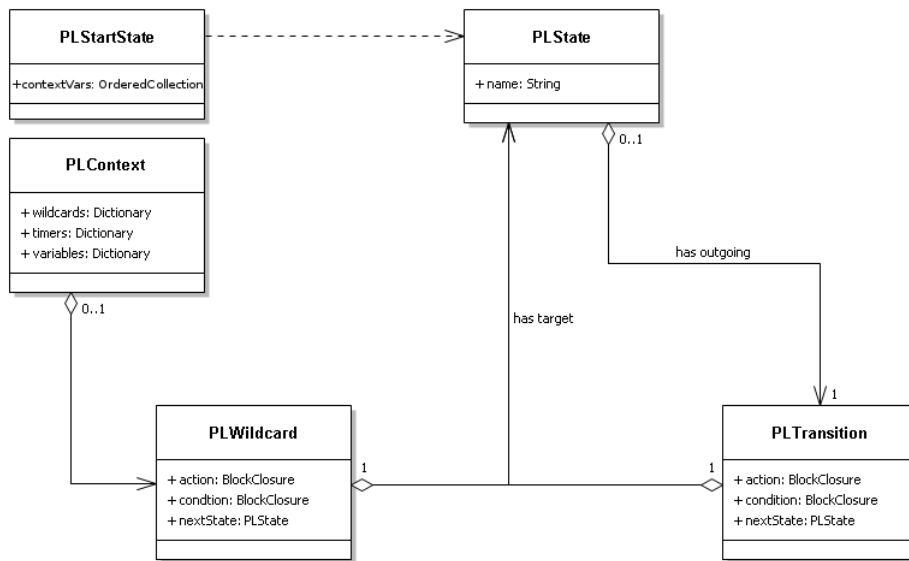


Figure 4.10: The State Machine Architecture with PLContext

4.5.1.1 Global Access

The context is defined globally to provide *access during the execution of nested machines*. E.g., in the use case of a machine which terminates at the end of a timer, the timers and the wildcards of this machine have to be checked at any time, even during the execution of a nested machine.

The start of a nested machine will not create a new context. The context data of the nested machine has to be *merged* with the context data of the upper machine. The global access of the context allows doing this simply by adding the nested machine's context variables, at the beginning of its execution, to the global context .

The decision for a global context is also the reason for PLStartState's instance variable *contextVars*. The method *PLStartState>>addToContext:* can be called with an executable *block* as parameter. The block contains methods for the *creation and storage of the machine's context variables*. The method will add this block to PLStartState's instance variable *contextVars*. When starting the machine, the processor will first of all execute all blocks stored in *contextVars*, thus adding the machine's context data to the global context.

Section 7.1.7 gives an example on how to use the context properly.

4.5.1.2 Execution of Nested Machines

For the correct execution of a nested state machine, we need to add a small but important detail. Since `PLContext` is a global variable, we have to *make a copy of this context* at the beginning of the execution of a machine and *restore this copy at the termination* of a machine. Only like this the context will have the exact same state right before and after the execution of a nested state machine.

So we copy the context at the very beginning of the execution:

```
contextCopy = PLContext copy
```

And restore it at the very end:

```
PLContext = contextCopy
```

4.5.2 Turing Completeness

Of course, solving one problem does not imply Turing completeness of the constructed model. Chapter 5 contains a proof to show that the constructed architecture at this point indeed is Turing complete.

4.6 The Execution Loop

Now that we have all the pieces of a state machine together, all kinds of states and transitions and a context to store wildcards and variables, let's have a look at the execution of such a machine. For the supervision of the whole workflow, we added the class `PLProcessor`. So here's an abstraction of how the method `PLProcessor>>execute` works:

1. Set the `StartState` as current state.

```
currentState = aPLStartState
```

2. Load all variables and wildcards into the context.

```
currentState contextVars loadIntoContext
```

3. Loop as long as the current state has transitions, which means as long as we haven't reached an end state yet.

```
WHILE: currentState hasTransitions
```

4. Loop through the wildcards, received from the global context. If the condition is true, execute the action and set its next state as the current state. Then jump to the next

execution of the WHILE-Loop, since we don't want to check the rest of the transitions if one has already triggered.

```
FOR EACH: context getWildcards AS w

  IF: w checkCondition
    w executeAction
    currentState = w getNextState
    continue WHILE-Loop
  END IF.

END FOR EACH.
```

5. Do the same as above with the transitions of the current state.

```
FOR EACH: currentState getTransitions AS t

  IF: t checkCondition
    t executeAction
    currentState = t getNextState
    continue WHILE-Loop
  END IF.

END FOR EACH.
```

6. When the WHILE-Loop is over, it means we reached an end state and the execution of the state machine has terminated.

```
END WHILE.
```

4.7 Run Time Programming

In chapter 3 the goal has been set to make the architecture support run time adaptation, i.e. that a programmed robot behavior can be altered at runtime. The constructed execution loop and the implemented state machine architecture are building a good base to support real time adaption: a robot's *behavior is defined by the states and transitions* of a state machine. Therefore, to alter the behavior it suffices to modify, add or remove states and transitions.

The runtime environment of Polite provides straightforward mechanisms for the run time modification of objects. By integrating them in the GUI, it is possible to access and alter the state machine's components while executing a program, thus allowing programming at run time.

Since this work focuses on the back end and doesn't go any further into the GUI layer, it suffices here to say that *the implemented back end supports run time adaption*.

5

Validation

To validate the functionality of the implemented architecture, the following section *proves full computability* of the system. The used method is based on the *Church-Turing Thesis*.

5.1 Turing Completeness

The *Church-Turing Thesis* is named after the mathematicians *Alonzo Church* and *Alan Turing*. The thesis states, among other things, that *the class of Turing computable functions coincide with the informal notion of an effectively computable function*¹.

Therefore, to show that the model is able to compute all effectively computable functions, it is sufficient to show that it is able to compute all Turing computable functions, or in other words, it is *Turing complete*. A way to prove *Turing completeness* is to *show that the system is able to simulate the behavior of any Turing machine*.

5.1.1 Turing Machines

The model of a Turing machine (TM) is quite similar to a finite state machine. It is also defined as a set of *states and transitions*. But further, a TM has a *read/write device* located on an *infinite stripe of tape*, which is divided into cells. Inside these cells are either symbols of a defined alphabet, or *Blanks*, whereas a Blank can be interpreted as

¹https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis

empty or *nothing*. The read/write device is always pointed at exactly one cell of the tape, reading one input.

With the *Mealy machine*, we had a transition consisting of a condition (or input), an action (or output), resulting in a successive state. Now with TMs, the transition's input is the symbol which the device is reading at this moment. The output consists of the symbol it will write on the current position of the device. After writing a symbol to this position, the device will move either one position to the left or one position to the right.

5.1.2 Simulating Turing Machines

So what is to be demonstrated is that the model of a Turing machine can be simulated with the constructed system.

Assuming there are *two variables*, one containing a *list of symbols* of a defined alphabet, the other one containing a *pointer* to one element of this list. The list simulates the infinite stripe of tape, while the pointer takes the place of the read/write device. The transitions' condition checks the symbol of the list, which the pointer is currently on, equivalent to the read/write devices input in a TM. The transitions' action writes a symbol into the current list item and decides whether the pointer moves to the previous item, the next item, or stays at the current item, equivalent to the moving of the read/write device. If the pointer is pointing at a empty list item, or if it moves out of the list, the input will return a *Blank*.

If we can build such a model with our architecture, the exact behavior of any Turing machine can be copied, thus the system would be *Turing complete* and therefore capable of calculating every effectively computable function.

5.1.3 Simulation

Now we want to show that we can do so with the implemented architecture.

In any case, we have to define a start state $q_0 \in Q$:

```
start := PLStartState new.
```

To simulate a Turing Machines endless tape of symbols, we *add a Dictionary* to the machine's context variables. A Dictionary allows us to add key-value associations, like $\{1 \rightarrow a\}$ to it. The key will be the symbol's position on the tape and the value will be the symbol.

The definition of a Turing Machine says that there is only a finite number of symbols (of a specified alphabet) on the tape, the rest are filled with blanks. Since we can't build an infinitely big Dictionary, we will only fill in the non-blanks of the input and then define that *null will be our definition of a blank*.

Furthermore, we *add a pointer*, simulating the read/write device. The pointer is simply storing a number which indicates what position the device is currently on. This would look something like this:

```
start addToContext: [ PLRunTime context addVariable: 'tape';
                    addVariable: 'pointer';
                    variables at: 'tape' put: (Dictionary new)]
```

When the machine is being started, this block will be executed, adding the two variables to the current context. We can then *add the non-blank symbols* of the input to the Dictionary. The Pointer is by default set to 0, the position where we want to start.

Once we have this set-up, we can build the rest of the Turing Machine. We need a *set of states*, a subset of which are the final states, one is the start state, and the *transition function*.

We have already built the start state. The rest of the set of states can easily be built by instatiating new PLState objects.

In our architecture, a state is said to be a final state if there is no transition originating from this state, so we don't have to explicitly define a set of final states. Now we can simulate the transition function, defined as followed:

$$\delta: (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

For an *arbitrary transition* $\delta(q_1, \gamma_1) \mapsto (q_2, \gamma_2, d)$ in the Turing Machine, we define an *equivalent transition* in our state machine:

```
q1
when: [((PLContext variables at: 'tape')
        at: (PLContext variables at: 'pointer')) ifAbsent: [^'blank']
        = \gamma_1]
do: [((PLContext variables at: 'tape')
        at: (PLContext variables at: 'pointer') put: \gamma_2).
      (PLContext variables at: 'pointer') :=
        (PLContext variables at: 'pointer') + d.]
goTo: q2.
```

We send the message *PLState>>when:do:goTo:* to state q_1 . This will add a transition with origin state q_1 to q_1 , whereas '*when:*' defines the condition, '*do:*' defines the action

and *'goTo:'* defines the next state.

In the condition-block we fetch the *tape*-variable and the *pointer*-variable from the Context, get the symbol of the tape at the pointers current position and compare it to the input symbol γ_1 . *'ifAbsent:'* defines what to do if there's nothing on this position, so we configure it to return a 'blank' in this case.

The action block writes γ_2 to the current position of the tape and increments or decrements the pointer. So d would be 1 in case of a *move-right* and -1 in case of a *move-left*. Finally we define q_2 to be the next state.

We are thus able to set up and simulate any Turing Machine. Therefore our architecture is said to be *Turing Complete* and, according to the Church-Turing thesis, *capable of calculating every computable function*.

5.2 Usability

Turing completeness shows that there are no limitations in functionality. Still it does not tell if the constructed system is suitable for the usage in robotics. In fact, the integrated pattern of finite state machines does not fit well for controlling *any* class of robots. A robot which is exclusively controlled remotely e.g. does not need any states and transitions, and would be easier to manage with another architecture. Or an industrial robot which executes the same moves over and over again and doesn't need to react on some kind of events.

The concept of state machines and especially Mealy machines, which is integrated in the architecture, is based on rules of behavior: a transition triggers depending on the current state and a condition, which can be the input of a sensor e.g.. The triggering of a transition can execute an action and leads to another state. The composition of these transitions (and states) builds a state machine, so a state machine can be seen as a composition of rules of behavior, resulting in an autonomous intelligence.

Therefore, the integrated concept of Mealy machines is suitable for a class of robots which act and complete tasks without the interaction of a human, and are able to perceive and react to their environment. This class can be titled as the class of *autonomous robots*. The other components of the implemented architecture, like Polite, run time adaption, wildcards or nested state machines additionally improve the usability and allow users to program autonomous robots in an easy and usable way. The abstraction of JetStorm through PoliteVehicle finally builds an easy-to-use interface between this back end and the autonomous Lego robots.

6

Conclusion and Future Work

The product of this work is a project which is capable of interacting with Lego Mindstorms' robots. It allows users to control the robot's behavior by building a *finite state machine*, whereas a better level of abstraction is supported through the implementation of *wilcards* and the possibility of executing *nested state machines*. By integrating the *use of variables* into our system, and with help of the *Church-Turing Thesis*, we proved that our construct is *Turing Complete* and therefore able to calculate every computable function.

We have met the previously set requirements and received a properly working system. But still, the product can be extended and improved in many ways. In this section, we will touch on some ideas for future work.

6.1 Asynchronous Processing

A new interesting feature would be to offer the possibility to run a nested state machine asynchronously. That could especially be helpful if you have a robot with multiple different robots. A user could e.g. let a state machine control the movement of the wheels a nested machine control the movement of the arms. It is like the robot would learn multitasking.

One can actually start a nested machine asynchronously already, by using *PLProcessor*>>*start*: instead of *PLProcessor*>>*execute*., but this wouldn't handle the Context correctly. The submachine would have to create a new context instead of copying and restoring the current context.

Another idea would be to even let the execution of multiple transitions be asynchronous or in other words let a transition have multiple next states and therefore splitting the processing into multiple threads when triggering this transition. This behavior may not seem to make a lot of sense with the Lego robots, but it does simulate *Non-Deterministic Finite Automata*, which can for example be used for the evaluation of regular expressions.

6.2 Extending PoliteVehicle

The adaption to the JetStorm library through PoliteVehicle provides a more natural and easy-to-read interface to interact with an Ev3Vehicle. Still there are a lot of possibilities to improve this layer. On one hand, the list of commands of PoliteVehicle could be extended, supporting more of the EV3s native commands and creating more abstractions of combinations of those.

Also, PoliteVehicle is adjusted to only one kind of the great number of possible EV3 robots. More could be covered by the implementation of additional classes, or the implementation of one adjustable class that could be configured to fit *any kind of EV3 robot*. In fact, PoliteVehicle is the only class which is associated to the Lego robot, so by exchanging this class, this back end could be used to program *any thinkable robot*.

7

Anleitung zu wissenschaftlichen Arbeiten

7.1 Tutorial

This section provides a short tutorial to help getting started with *Polite For EV3*. It contains Information on how to install and set up the project and a few examples to create simple finite state machines to interact with an EV3 robot. It focuses on the basic functions concerning the control of a connected robot, however the details of the Graphical User Interface won't be discussed since they are not part of this work. More information about the GUI can be found in Stefan Borer's work, *Lego Playground* [3].

7.1.1 Prerequisites

The EV3 Brick supports different ways for a connection with a device. Since this project is based on run time programming, a connection via USB or Bluetooth was no option, so we chose to *support only Wi-Fi connections*.

Unfortunately, the brick does not have the hardware to support Wi-Fi connections, therefore *a USB Wi-Fi adapter is necessary*. Moreover, not every USB adapter is compatible. On the official Lego Website¹ they say:

[...] The NETGEAR N150 Wireless Adapter (WNA1100) is the recommended Wi-Fi dongle for use with the EV3 Intelligent Brick.

¹<http://www.lego.com/en-us/mindstorms/support>

7.1.2 Installation

The project was developed for Pharo 4.0, precisely in a Moose 5.1 image. You can download the Moose 5.1 image or package at <http://moosetechnology.org/#install>. If you are new to Pharo, you might want to read the Documentation first at <http://pharo.org/documentation>.

Inside the Pharo VM, our Polite Lego project can be installed via the workspace, by entering and executing the following.

```
Gofer new smalltalkhubUser: 'JanKurs' project: 'PoliteSmalltalk';
  configurationOf: #PoliteSmalltalk; load.
(Smalltalk at: #ConfigurationOfPoliteSmalltalk)
  perform: #loadDevelopment.
```


The project can also be found at <http://smalltalkhub.com/#!/~JanKurs/PoliteSmalltalk>.

7.1.3 Start and Connect

The project's main GUI-object is called *PLPlayground*. So to start getting creative with the robots, simply execute:

```
PLPlayground open.
```

The appearing GUI is splitted up in two sections. In the left panel you'll find (from top to bottom) a toolbar with control buttons, a workspace to enter and execute code and a console in which the program will give the output of the executed code. The right panel will give out an overview and a visualization of the current state machine, and it allows the user to change the states and transition while the machine is being executed.

So let's start by connecting a computer with the EV3 Brick. It is important that the Brick and the computer are in the same Wi-Fi network. Also remember that the brick needs a Wi-Fi adapter. A connection to a Wi-Fi network can be established by navigating on the EV3 Bricks screen to *Settings > WiFi*. Once the Brick has managed to establish the connection, you can find out its IP address in the menu section *Brick Info*. To connect the Brick with the computer, click on the icon for connection  in the top toolbar and enter the Brick's IP address in the appearing window.

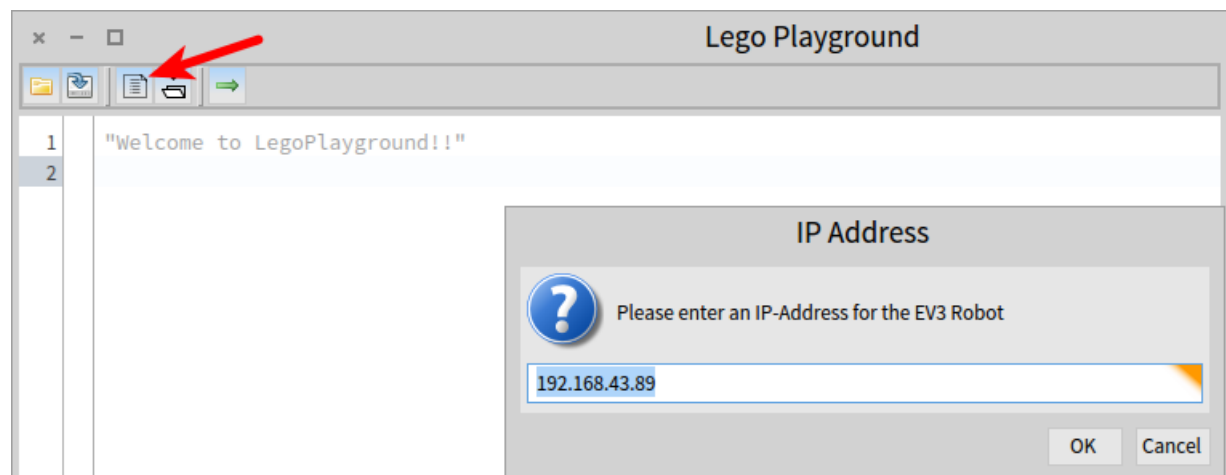
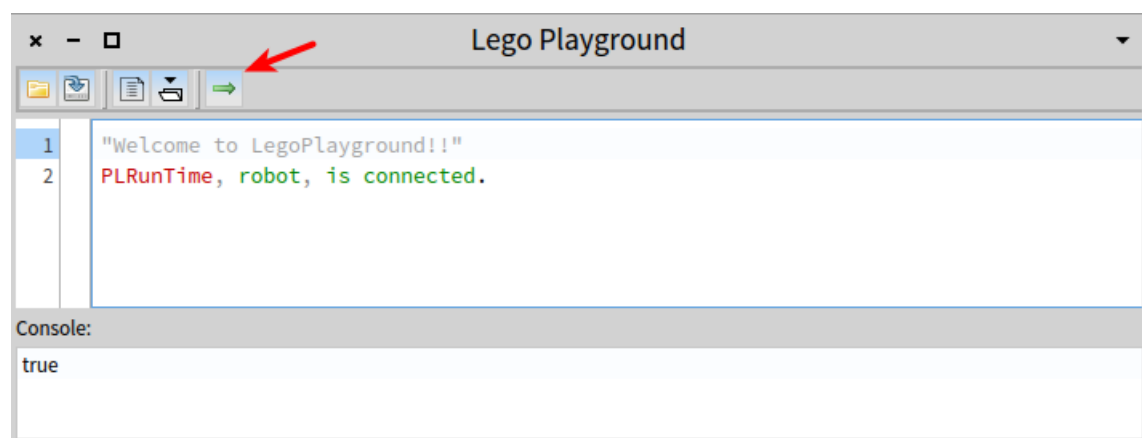


Figure 7.1: Establishing a Connection

7.1.4 First Steps

For starters we'll begin with some basic commands to introduce the most important controlling objects.



PLRunTime is a global variable which allows you to access all objects you will need. So in the example in Figure 7.1.4, by executing '*PLRunTime, robot, is connected*', we fetch the robot object and ask if it is connected to the bridge. The robot object is of the kind *PoliteVehicle* and was instantiated automatically while establishing the connection. The other objects stored inside *PLRunTime* are called *context*, *processor* and *log*. We will use them later in this tutorial.

By clicking the green arrow, all code written in the workspace will be executed and if the robot is connected, the console will return the value *true*.

Remember that the code is interpreted in *Polite Smalltalk*, which allows sentence-like writing. For example, the method *'isConnected'* is here written as *'is connected'*. Both camel-case and sentence-case are accepted by the interpreter, however the identifiers must be separated by a colon.

7.1.5 Robot Commands

Now that we know how to access the robot, we can try out some basic commands. Below is a list of the possible message to send to *PoliteVehicle*.

- *driveForward, driveBackward*. Starts driving forward/backwards at the default speed.
- *stop*. Stops the robots motors.
- *turnLeft, turnLeft:, turnRight, turnRight:.* Turns the robot to the left/right at the defaultSpeed and afterwards waits for defaultDelay-seconds. The amount of degrees to be turned can be passed as parameter, the default being 90 degrees.
- *turnRandom, turnRandomBetween:And:.* Turns randomly to the left or to the right. The range of degrees can be passed as parameter.
- *beep, beepLong*. Let the robot make a short or long beeping sound.
- *color*. Reads the color sensor and returns the name of the color as String. There are seven different possible colors: black, blue, green, yellow, red, white and brown.
- *distance*. Reads the ultrasonic sensor and returns its value. The returned number is the amount of millimeters to an obstacle in front of the robot.
- *isTouched*. Returns the value of the touchsensor; true if the sensor is touching anything, false otherwise.
- *isConnected*. Asks if the object is currently connected to the Ev3Brick.

Furthermore there are getters and setters for all instance variables of *PoliteVehicle*. So for example you can access the *Ev3Vehicle* from the JetStorm API, which supports a great number of methods for the Ev3Brick. However to use these, one has to have a closer look at this API, which can be interesting for experienced programmers but may be complicated to understand for inexperienced ones.

7.1.6 Building a State Machine

After learning the basics of how to control a robot, we can go a step further and create some more complex behaviors through the concept of state machines. The following example will create a state machine with three states: *Start*, *Driving* and *End*. Two transitions are leading from *Start* to *Driving* and from *Driving* to *End*. The first is triggered without a condition (and therefore called *Epsilon-Transition*) and will cause the robot to 'drive forward'. The second is triggered on the robot's *distance* sensor returning less than 200 (millimeters) and will cause the robot to stop.

```
| start, driving, end |
start := PLStartState, new called: 'Start'
      machine called: 'Watch your step'.
driving := PLState, new called: 'Driving'.
end := PLState, new called: 'End'.

start,
  do: [ :rt | rt, robot, drive forward ]
  go to: driving.

driving,
  when: [ :rt | (rt, robot, distance) < 200 ]
  do: [ :rt | rt, robot, stop ]
  go to: end.

PLRunTime, processor, start: start.
```

The creation of the states is pretty straightforward. The names you give them by calling 'new called:' and 'machine called:' are later represented in the visualization of the state machine.

The creation of the transitions may look a bit more complicated. Since a transition is stored in its origin state, a message with the necessary parameters must be sent to this state to create a transition. So by calling 'PLState>>when:do:goTo:' a transition is added to this state with the condition passed as parameter after 'when:', the action passed after 'do:' and the next state passed after 'goTo:'. If the 'when:'-Block is left out, the condition will always be true and therefore the transition will be triggered immediately after entering this state.

The condition and action are passed as executable *Block-Closures*. When executed, the processor will pass the global variable PLRunTime to the block. By beginning the block with [:rt — ...] we define the variable *rt* to be the PLRunTime variable so we can then act upon it, e.g. to fetch the robot object.

7.1.7 Using the Context

To define some more complex behavior we want to additionally use the context of a state machine. The context provides three handy elements: *Timers*, *variables* and *wildcards*. In the next example we will add a timer and a wildcard to solve the following use case:

The robot should drive around, meaning that it should avoid objects by turning around when approaching them. After doing this for thirty seconds, the robot should stop.

This piece of code will build and start the desired state machine:

```
|start, end, driving, turning|
start := PLStart State, new called: 'Start'
      machine called: 'Drive Around'.
driving := PLState, new called: 'Driving'.
turning := PLState, new called: 'Turning'.
end := PLState, new called: 'End'.

start, add to context: [ :c | c, add timer: #MachineTimer]
start, add to context: [ :c | c,
                      when: [ :runtime |
                          (runtime, context, get time of: #MachineTimer) > 30 ]
                      do: [ :runtime | runtime, robot, stop ]
                      go to: end

start, do: [ :rt | rt, robot, drive forward. ]
go to: driving.

driving,
  when: [ :rt | (rt, robot, distance) < 200 ]
  do: [ :rt | rt, robot, stop ]
  go to: turning.

turning,
  do: [ :rt | rt, robot, turn random; drive forward ]
  go to: driving.

PLRunTime, processor, start: start.
```

Like in the previous example we build a state *Driving* with a transition triggering on the distance sensor returning less than 200 millimeters. But instead of entering the end state, the transition leads to the state *Turning*. In this state a transition is being executed immediately (no condition) which causes the robot to turn around at a random amount of degrees, start driving forward again and returning to the state *Driving*.

To make the machine reach an end state and stop after 30 seconds, we add a timer

and a wildcard to the context. To do so, we use `PLStartState`'s method `addToContext`. The blocks passed to this method as parameter will be executed at the very beginning of the state machine's execution.

In the first block we add a timer called *MachineTimer*. The timer will instantly start.

Then we add a wildcard. This works just like adding a transition, but you add it to the context instead of a state. This way it will be checked and can be triggered regardless of the state the machine is currently in. So we define a comparison of the added timer as condition, the halting of the robot as action and the *End* state as the successive state.

7.1.8 Further Instructions

With this knowledge and a bit of imagination a lot of different programs can be constructed for EV3 robots. Further instructions on the usage of the GUI, including nested state machines, live-time interaction with the state machines and more can be found in Stefan Borers work, *Lego Playground* [3].

Bibliography

- [1] M. Lungu and J. Kurš, “On planning an evaluation of the impact of identifier names on the readability and maintainability of programs,” in *USER’13: Proceedings of the 2nd Workshop on User evaluations for Software Engineering Researchers*, 2013, pp. 13 – 15. [Online]. Available: <http://scg.unibe.ch/archive/papers/Lung13a-Planning.pdf>
- [2] J. Kur, M. Lungu, O. Nierstrasz, and T. Steinmann, “Polite smalltalk - an implementation,” Sep. 2016. [Online]. Available: <http://dx.doi.org/10.5281/zenodo.61578>
- [3] S. Borer, “Lego playground (to appear),” University of Bern, Bachelor’s thesis.
- [4] J. Laval, “Jetstorm - a communication protocol between pharo and lego mindstorms,” URIA – Ecole des Mines de Douai, Tech. Rep., 2014. [Online]. Available: www.jannik-laval.eu/assets/files/papers/Lava14a-JetStorm.pdf
- [5] J. Fabry and M. Campusano, “Live robot programming,” in *Ibero-American Conference on Artificial Intelligence*. Springer, 2014, pp. 445–456.
- [6] D. R. Wright, “Finite state machines,” 2005, cSC215 Class Notes. Prof. David R. Wright website, N. Carolina State Univ. Retrieved July 14, 2012.