UNIVERSITÄT
BERN

The Lego Playground

Providing an IDE for live programming Lego Mindstorm robots

Bachelor Thesis

Stefan Borer
from
Kleinlützel SO, Switzerland

Faculty of Science
University of Bern

February 16, 2016

Prof. Dr. Oscar Nierstrasz
Dr. Mircea Lungu, Dr. Jan Kurš
Software Composition Group

Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

The Lego Mindstorms robotics kit with its visual programming language is often used in schools and universities teaching programming and mathematics. Meanwhile Live Programming is gaining traction in the field of robotics, offering the programmer more feedback and control over the robot than traditional methods. In his work on the back end of this project, Theodor Truffer implements a new way to program Lego Mindstorms robots in a Live Programming way using the Polite programming language. This thesis provides an Integrated Development Environment for the back end including state machine visualization, inspection and manipulation of state machine objects, creating a Live Programming experience.

Contents

1 Introduction

Lego Mindstorms EV3 is an easy solution to enter the fields of robotics that is often used to teach programming and mathematics at a beginners level from schools to university. The principle of Live Programming has emerged more and more in the field of robotics, providing better control and interaction to the programmer than traditional approaches. Live Robot Programming (LRP) [1] is a notable project following this principle. It makes use of the Pharo Smalltalk environment in order to provide interaction with robots in real-time, both by monitoring and changing its behavior.

The Polite Smalltalk programming language written by Mircea Lungu and Jan Kurš [2] addresses a problem most traditional languages ignore: inconvenient identifiers. Defining meaningful identifiers is often solved by gluing together words by using either camel case or underscores. This convention reduces readability of programming languages compared to natural language and may feel intimidating for beginners and inexperienced programmers. Polite solves this problem by introducing space separated identifiers, resulting in a syntax that is closer to natural language. Being still a language of the Smalltalk family, Polite inherits all its object-oriented principles and the perks of a Smalltalk environment.

Theodor Truffer's work [3] pairs the Live Programming principle with the Polite language for a solution to interact with EV3 robots. It lets the programmer create state machine driven programs for the EV3 robot and interact with it in real-time by changing transitions and other properties on the fly. Truffer's project makes use of JetStorm to interface with the robot and adds an abstraction layer to further simplify interaction.

Using Polite and a user-friendly API, this project acts as the back-end for an alternative to visual programming languages for beginners interested in robots but it lacks a suitable front-end.

In this thesis we introduce the Lego Playground which aims to provide an Integrated Development Environment (IDE) for live programming of EV3 robots based on the mentioned back-end. Following the Live Programming approach, the IDE comes with features like real-time modification of program properties, visualization of its logical state machine including highlighting of the current state and more.

After a short Chapter on related work, we will learn about the most important concepts implemented in the back end. The third Chapter collects requirements for the IDE, followed by a description on how the Lego Playground is implemented based on these requirements. At last the proposed solution is evaluated through usability testing and made accessible in a tutorial.

2
Related Work

This Chapter provides an overview of related products and projects that influence the topic discussed in this thesis.

2.1 Lego Mindstorms

Lego Mindstorms[1] is a kit containing software and hardware to build and program robots. The core element of this kit is the programmable brick, which is used to connect with electromotors and sensors. Starting from 1994 a whole series was developed with the most recent generation called Lego Mindstorms EV3. It features software based on LabVIEW[2] which offers a visual programming language to program the brick.

The programming language is based on graphical elements of different kinds, identified by color. The elements can be dragged onto the workspace and interconnected to design a program flow. For example green boxes represent motors and offer handles to set different properties such as their speed percentage or when to stop. Differently colored elements feature ways to control program flow through loops and timers, read out sensor values, compare them through logical operators and more.

[1] https://www.lego.com/en-us/mindstorms
[2] http://www.ni.com/labview

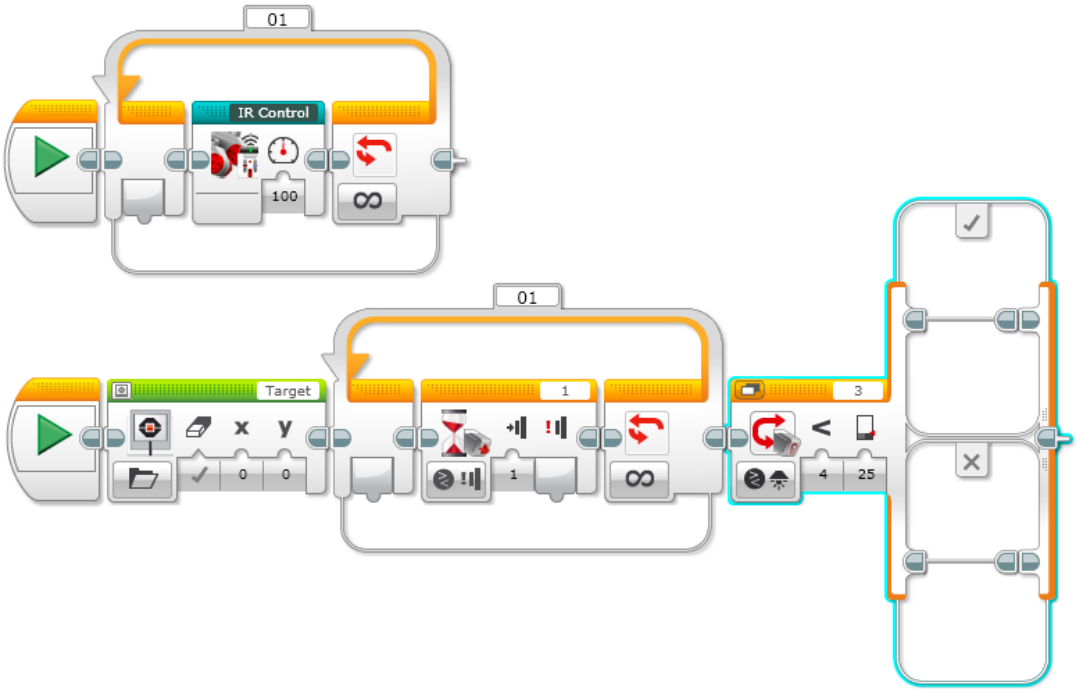

Figure 2.1: A sample program in LabVIEW.[3]

While this graphical approach aims for beginners, there are also alternative ways to control the EV3 brick for advanced users. Numerous projects exist to use different programming languages, either running directly on the brick like *e.g.,* Java using the leJOS firmware[4] or interface with the brick through its remote control interface such as ROBOTC, a C-based programming language[5].

2.2 Live Programming

Live Programming is a concept introduced by Tonimoto in VIVA [4]. Initially limited to visual programming languages it defines different levels of liveness for programs. The highest level of liveness is reached - and thus regarded as Live Programming - when programs are continually active and their behaviors are modified immediately when changed by the programmer [5].

[3]Taken from http://arstechnica.com/gadgets/2013/08/review-lego-mindstorms-ev3-means-giant-robots-powerful-computers

[4]http://www.lejos.org/

[5]http://www.robotc.net

2.2.1 LRP

Live Robot Programming (LRP) is a programming language for nested state machines designed around the principles of Live Programming [1]. In addition to the language it features an Integrated Development Environment programmed and running in Pharo[6] that comes with state machine visualization (see Figure 2.2). One of its targets is the EV3 brick which - equipped with a Wifi key - can be interfaced using JetStormForPhratch[7].

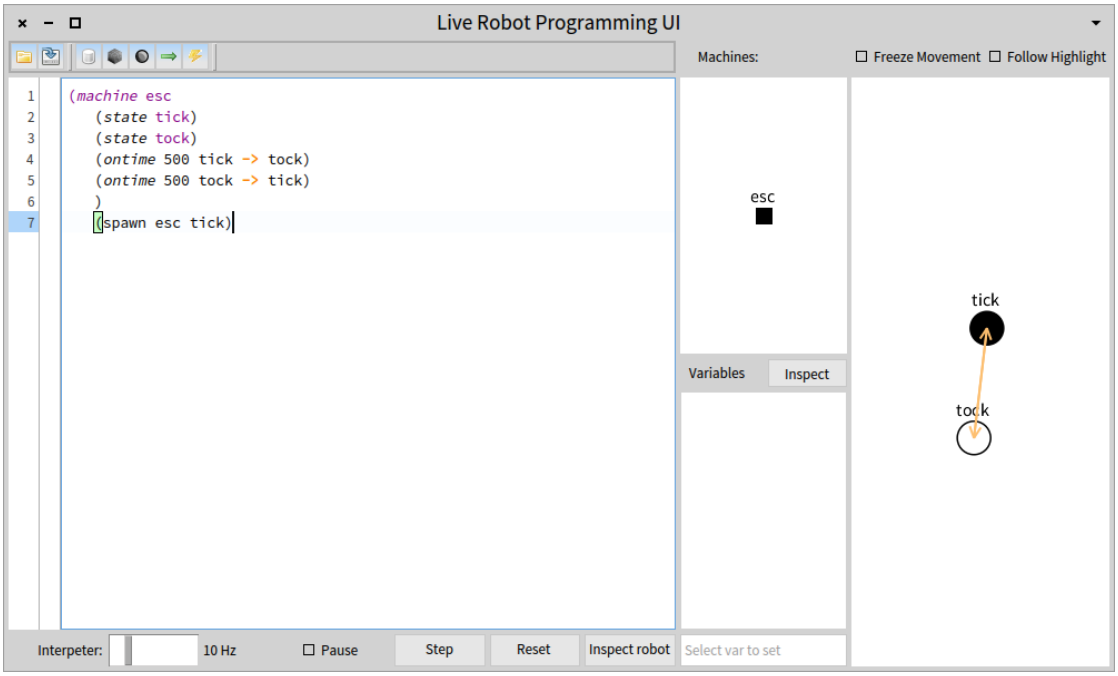

Figure 2.2: The Live Robot Programming IDE

2.3 Polite

Polite is a programming language introduced by Mircea Lungu and Jan Kurš in 2013 [2][6][7]. It's derived from Pharo Smalltalk but allows a different grammar for identifiers. It features so called *sentence case* or *phrase case identifiers* which are replacements of *camelCaseIdentifier* or *underscore_identifier*. As a result program entities such as classes or methods can be declared using phrases as names.

The main motivation for Polite is that naming of program entities is critical for readability. With good class and method names, code can be made nearly self-explanatory.

[6]http://pharo.org/
[7]http://www.phratch.com/jetstorm

As its syntax favors readability, Polite seems to be a good idea for this project because its code looks friendlier to both newcomers in Smalltalk and programming beginners in general.

3 Back end

In his work, Theodor Truffer describes the computational model that allows the use of Polite with state machines, how it interacts with the LEGO Mindstorms hardware and how we can take control over the robot using this API[3]. The following sections provide a short overview of this architecture, specifically of the underlying state machine model, as it serves as the back end to this project.

3.1 Architecture

The back end is designed around the `PoliteVehicle` class. This class represents a robot, which can be seen as its physical entities and the means to control it in a programmable way. As such, `PoliteVehicle` uses the JetStorm library to connect to the LEGO robot and provides a wrapper for specific functionalities such as driving forward or reading a sensor.

Once a `PoliteVehicle` is initalized, a state machine can be run on it through the `PLProcessor` instance, which implements the main execution loop. It accepts a `PLStartState` as the beginning of a state machine and makes use of further information stored in a `PLContext`. We will now go into more details.

3.1.1 State Machine Model

To control the behavior of a robot, finite state machines are popular choice. State machines can be used as a representation of a robot's physical (*e.g.,* driving speed,

distance from an object) and logical (*e.g.,* what to do if condition X is encountered) state, while the robot is always in exactly one state at a given time, called the current state. From a given state, there is a finite set of transitions to successor states available. One of the transitions is used once its condition becomes true, resulting in a change of state.

Because the program of a robot can be modeled using state machines, they offer a natural way to control it and a convenient way to visualize and keeping track of a robot's program. However, by default finite state machines do only passively keep track of a system without possiblity to interact with it.

Mealy Machines A *Mealy machine* is a kind of finite state machine. It is a set of states connected by transitions which themselves are evaluated under a certain condition and, as an extension to basic finite state machines, result in an action.

In the back end, states and transitions are represented by instances of the `PLState` and `PLTransition` classes. A `PLStartState` is a kind of `PLState` which indicates the start of a state machine, while any state lacking an outgoing transition implicitly is an end state.

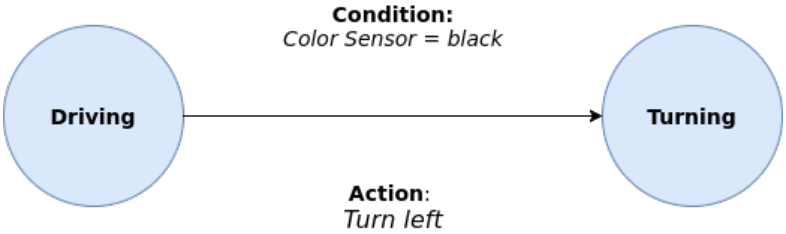

Figure 3.1: A transition of a Mealy Machine, which changes from the state *Driving* to *Turning* if the condition (*Color Sensor = black*) is met, resulting in an action (*Turn left*).

The action or output function of the Mealy Machine is needed to change the robot's behavior while running a program. Instead of just being a passive object as in a normal finite state machine, the Mealy Machine makes the robot an active subject in its environment. However, in order to get more flexibility out of this state machine model, we need to introduce some additions to the basic state and transition concept.

3.1.2 Additions to State Machine Model

In order to allow more flexible programming of robots, the paragraphs introduce extensions to the traditional state machine model.

Wildcards Wildcards are a kind of transitions, consisting of conditions and actions but without a starting state. Instead, a `PLWildcard` can be triggered at any time of execution

(that is, no matter what the current state is). This allows regular checking for certain conditions (*e.g.,* if the distance is less than 10 cm), without the need of defining such a transition for every state there is.

Variables The use of variables gives us more computational possibilities. For example, variables can be incremented in a `PLTransitions` action block every time a black tile is detected through the color sensor. In turn a `PLTransitions` condition can be dependent on the value of a variable, allowing more versatile logic.

Timers Timers are much like special variables containing a timer object which keeps track of the time passed since its creation. In the program flow, timers offer another possibility to control the robot's behavior.

These elements are all stored in the `PLContext` upon execution time, allowing modification during execution.

3.1.3 Nested State Machines

A nested state machine is an ordinary state machine. As such, it consists of a `PLStartState`, transitions and a set of states, at least one of which is an implicit end state. Once such a state machine is saved to a script (more about this later), it can be referenced by another state machine through its script name.

As a result, a transition can take a whole state machine definition as an action block, execute this nested state machine and continue to the succeeding state once the nested state machine is terminated.

Nesting of state machines allows to simplify scripting of robots, because a complex task can be divided into multiple simpler tasks. It also promotes reuse of existing code, since a saved script can be interpreted as a state machine on its own and therefore referenced in another script, treating it as a nested state machine.

4 Requirements

This thesis aims to describe a full-fledged Integrated Development Environment (IDE) to control LEGO Mindstorms robots using the back end of the previous Chapter 3, combined with the Polite Smalltalk language. We will now take a look at what such an IDE should offer to the programmer.

4.1 Graphical User Interface

4.1.1 Code Editor

The most important part of an IDE is obviously the code editor. As such, it needs to take keyboard input and print it to a text field. To support the programmer, highlighting key elements of the used syntax is an effective solution: The programmer finds variable names based on visual preattentive processing or gets signaled of syntax errors when the code turns red.

4.1.2 State Machine Visualization

It is helpful for the robot programmer to have an ongoing visualization of this representation, as it enables him to verify the correctness of the code in an easy way. Also, debugging of misbehavior is simplified, because the logical state of the robot can be tracked in a real time manner.

Because the state machine model in this project is extended by additional elements, it must draw the state machine not only with its states and transitions, but also take into account the special elements that were added as well. Nesting of state machines as well as wildcards should each get a fitting representation, while not making the main visualization confusing for the programmer.

Highlighting Having a representation of the state machine is one thing, but how can we verify in which state it is at the moment? This is where highlighting comes into play. It means that the current state in the current state machine should be distinguishable from other states/state machines and that this highlighting must be updated upon each change of state.

4.1.3 Inspection and Manipulation

Next to visualizing the underlying state machine, live manipulation of elements adds up to a good programming experience using an IDE, encouraging the user to interact with the robot in as natural a fashion as possible. To provide this kind of immersion, key elements such as variables and timers should expose a way to interact with them, changing values and inherently the characteristics of the program.

4.1.4 Saving and Loading

As the programmer writes more and more complex scripts, the possibility to save his work and load it again is another requirement. This standard task is of special importance in our project, because state machines can be nested by a reference to a script containing a state machine definition.

4.1.5 Connection handling

Because the IDE needs to interact with a LEGO Mindstorms brick, there needs to be a way to connect to and disconnect from it. This should be as easy as possible, only needing the IP as information.

5 Implementation of Lego Playground

To fulfil the requirements mentioned in the previous Chapter, there are a number of decisions to make. Some of which are of a technical nature, such as the decision about the toolkit used for drawing the GUI, some are about the interaction of the UI with the underlying subsystem and some are design related. As this project takes place in the Pharo Smalltalk environment, the implementation resides in this ecosystem.

5.1 Toolkit

To start the implementation of the user interface, a decision about the tools used for drawing its elements is necessary. There are a number of available tools to draw graphical elements in Pharo which we will discuss here.

Morphic Morphic is Pharo's graphical user interface and everything drawn inside Pharo is a `Morph`. It is very low-level as every element of a UI is a subclass of `Morph`, the most basic object to be rendered in a Pharo VM.

Glamour Glamour is a dedicated framework to describe the navigation flow of browsers. Thanks to its declarative language, Glamour allows one to quickly define new browsers for their data.

Spec Spec is a widget based framework which promotes reuse. It takes a model and layout description to create a widget, which can be reused and combined with others to

create a UI. This allows for flexible and complex interfaces.

Decision Morphic is a rather exhaustive tool to implement complex user interfaces. It leaves the programmer a lot of freedom in the design but also requires a big amount of work to implement.

Glamour is good higher level framework for displaying and interacting with data. Because the Lego Playground is not browser for data, it somehow misses the point in this use case.

Therefore decision on the toolkit was in favor for the Spec framework, because it has good online documentation, a robust API and parts of the UI can be reused from the LRP project. Mainly the state machine visualization pane could be integrated, which does exactly what is need in the Lego Playground as well.

5.2 Graphical User Interface

According to Chapter 4 there are a number of characteristics a suitable GUI for live programming robots should offer. The `PLPlayground` class is the implementation of this GUI, containing all the functionality discussed in Chapter 4.[1] It interacts with the back-end to control the robot and also to get feedback from the robot and its logical state.

[1] To install the software, the reader is referred to Theodor Truffer [3, p. 38], which contains a tutorial.

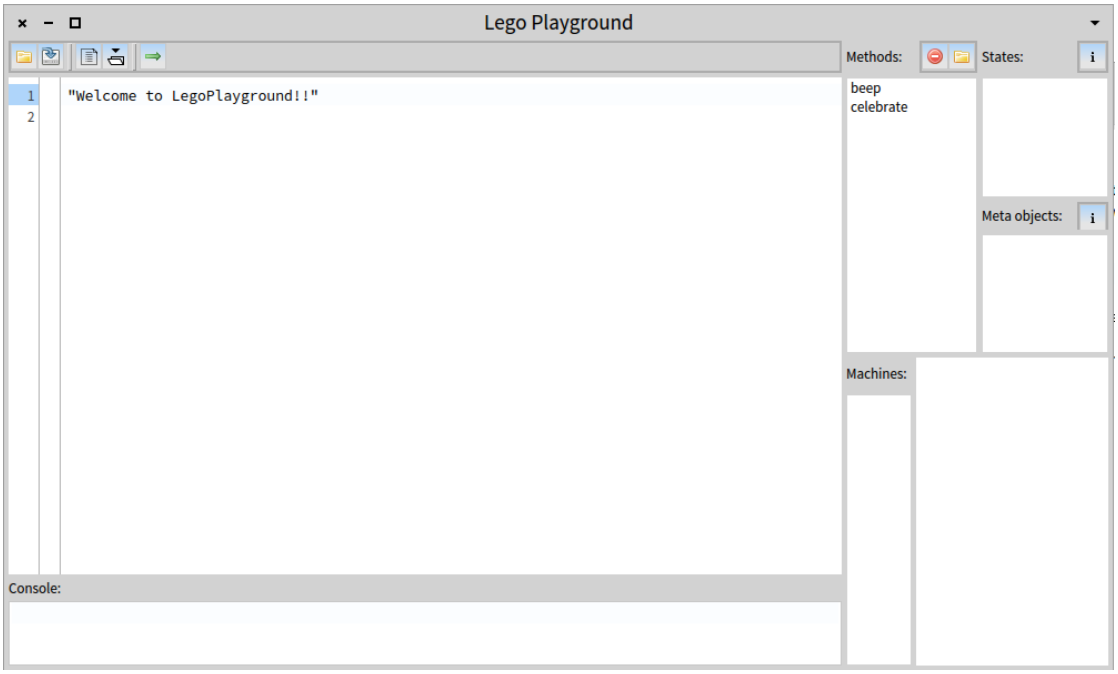

Figure 5.1: The Lego Playground

5.2.1 Layout

The Lego Playground (implemented by the `PLPlayground` class) consists of two columns, the left one containing the PLEditorPane and the right one the PLRightPane.

5.2.1.1 PLEditorPane

The PLEditorPane holds three rows: the top menu, the code editor and the console output. The menu contains buttons for saving and loading from a file, connecting to the EV3 brick, storing to a method and running the script. The code editor plugs in the code highlighter and the Polite parser.

5.2.1.2 PLRightPane

This is where the run time action is going on. In the upper part, the method list presents state machines already stored and offers to open them in the code editor or delete them. There are lists for states and meta objects, each with an inspection button allowing to inspect and manipulate a selected object in the list.

On the bottom, the state machine visualization is taking place. A big pane provides space for sophisticated state machine representations, while the left-hand list acts like a stack, showing the nesting depth of the current state machine running.

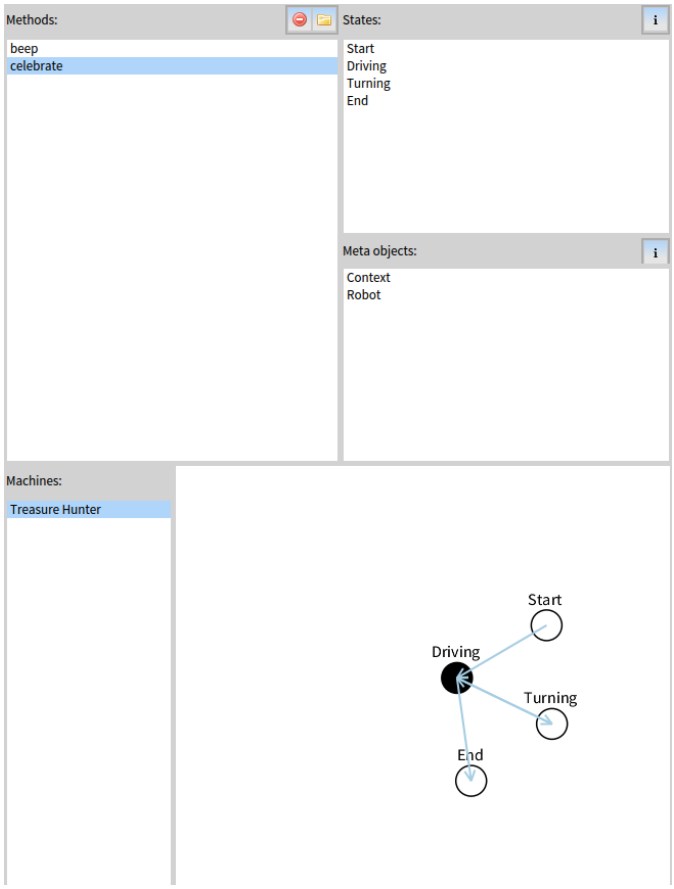

Figure 5.2: The PLRightPane in action, showing (1) available methods, (2) States of the current SM, (3) Context and Robot objects to check wildcards and various properties, (4) SM visualization and (5) the current machine nesting.

5.3 Implementation

5.3.1 Code Editor

According to our requirements, a code editor needs to take text input, print it to the screen, allow edits of the text and highlight the syntax of the language in use. Thanks to the widget driven design of Spec, we can reuse an existing implementation. In this case a `RubScrolledTextMorph` does the job, providing a code pane with line numbers display. The code highlighting is done by the `PoliteTextStyler`.[2] Every time the code is changed, the `PoliteTextStyler` parses it and returns a colorized version. If the programmer hits the GO button, the text is passed to `PoliteSmalltalk>>execute`.

[2]Please note that this project relies on version *PoliteSmalltalk-TheodorTruffer.62* of [8].

5.3.2 From Code To Visualization

Once the code is executed *i.e.,* the state machine is built, `PLProcessor>>execute:` takes over and runs the state machine. But how does the IDE know what the state machine looks like and what the current state is? The `PLProcessor` notifies the IDE everytime something changed.

On launch of `PLProcessor>>execute:` it passes the `PLStartState` to `PLRightPane>>pushMachine`.

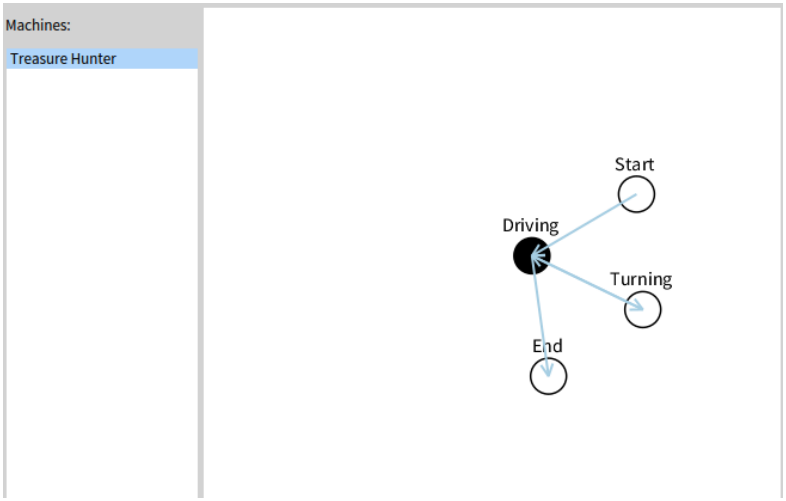

Figure 5.3: A running program with current state *Driving* and current machine *Treasure Hunter*.

As the terminology suggests, state machines are treated in a stack-like fashion. The method pushes the new state machine to the machine list, making it the top stack element. It also parses the state machine to get all states and populates the states list as well as draws the visualization.

From now on, the `PLProcessor` notifies the UI every time the state changes. In turn, the UI updates the visualization pane which highlights the new current state. For the highlighting, a color-based approached is used, where the current state is colored black and the nesting is highlighted in blue. This approach is straight forward to implement and sufficiently effective in hinting the current system state.

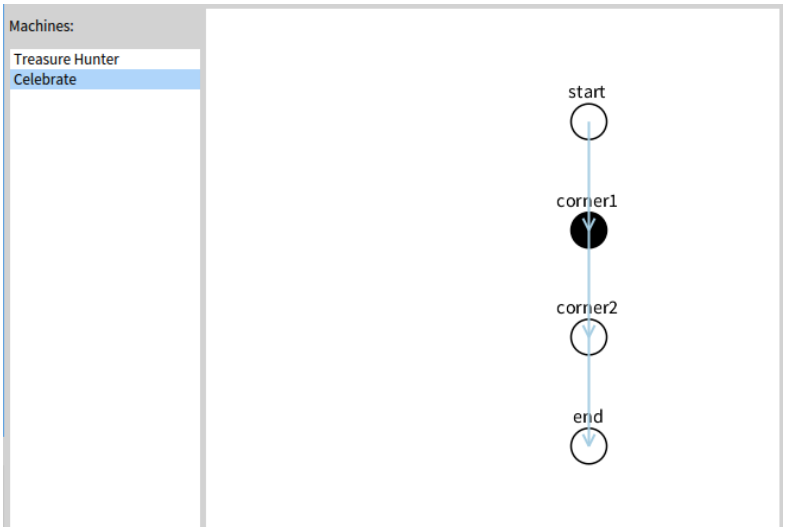

Figure 5.4: After the program hit an transition resulting in nesting, the current machine is now *Celebrate* and its current state is *corner1*.

5.3.3 Inspect and Manipulate

In the top right part of the UI are the States and Meta objects lists. The former holds the states objects of the state machine, while the latter consists of just two items: Context and Robot. Both lists come with an Inspect button, which launches a Pharo-native inspect window of the selected item. This allows viewing the details of each object and even manipulating its properties.

For example, inspection of a state object enables the programmer to interact with its transitions and change their conditions, actions or next state.

The Context item offers a similar functionality: interaction with all the variables and timers stored in `PLContext`.

The robot item does the same and offers the programmer means to change `PoliteVehicle` related values, such as the default speed. Most interestingly, we can read out sensor values such as color or distance from an object.

5.3.4 Nested state machines

Push When nesting of state machines occurs, a couple of changes happen to the UI which are handled by `PLRightPane>>pushMachine:`. It pushes a new item to the machine list and highlights it, collects the new states, repopulates the states lists and lastly redraws the visualization with the new machine.

Pop Similar actions are taken once a nested machine terminates. This time, `PLRightPane >>pushMachine:` takes care of the process utilizing a backed up copy of the underlying

state machine.

6 Usability Testing

A popular way to test a user interface for its usability is the heuristic evaluation proposed by Nielsen and Molich [9]. Authors propose a list of fundamental usability principles (called heuristics) which can be used to compare the individual UI elements with.

It's an easy, fast and cheap usability engineering method, relying only on a small set of evaluators: The sweet spot in most cases lies between three to five evaluators. A single evaluator was able to find 35% of usability problems averaged over 6 projects, while there was a decreasing benefit from each additional expert [10].

6.1 Method

The most important question to answer before the conduct of an heuristic evaluation is the number of experts needed. A couple of parameters play into the decision.

One of these is the complexity of the user interface. As this project never aimed to be a feature rich and extendable IDE (such as *e.g.,* Eclipse[1]), the resulting user interface is of relatively low complexity.

A big reason for the low complexity is its domain specific use case: The sole purpose of the Lego Playground is to program one exact model of Lego Mindstorms robots using state machine definitions in Polite Smalltalk language over a wireless network connection.

The previous points give rise to the last: cost. Introducing an expert into a domain specific software is cost intensive. At the same time the user interface is of limited

[1] http://www.eclipse.org/ide/

complexity, suggesting only a small number of experts.

As a result of these considerations, the evaluation was conducted by one expert.

Given the highly domain specific use case of this project (including the Polite Smalltalk programming language), leaving the expert unguided during the session seems unreasonable. In such a case the authors suggest to provide a usage scenario by listing the required steps for the evaluator to fulfill.

6.1.1 Usage Scenario

A suitable scenario for our use case contains all important dialogue elements but shouldn't make the evaluator learn the Polite Smalltalk language beforehand. It should however let the expert interact with the state machine handles.

As a compromise, the following list of steps was aggregated:

1. Connect to the EV3 robot
2. Load a script from file given by path and name
3. Run the script
4. Inspect the state *Driving*
5. In the first transition of *Driving*, change the minimal distance to 150
6. Observe the visualization, wait for the robot to finish
7. Make change from step 5 to the script and save it to file
8. Disconnect the robot

6.2 Results

In the conducted Heuristic Evaluation a number of usability problems were found. Each list entry is denoted by the violated heuristics as found in Table 6.1.

- There is no text-based menu bar to connect the EV3 robot. One has to guess which icon identifies which functionality, *e.g.,* the *Connect* button, only the tooltips may help. This violates a de-facto standard present in other applications. However it is consistent inside the Pharo environment, as applications therein lack traditional menu bars. Still the button could be text driven as *e.g.,* in the Monticello browser (see Figure 6.1 for a comparison). (2, 3)

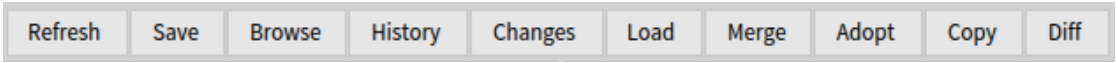

Figure 6.1: The Monticello menu bar using labels instead of icons.

- After hitting *Connect*, there is no way to tell if the connection was successful or not. The status of the connection is invisible in the main window. A second click on the button reveals a new dialogue to disconnect the robot, so one has to guess the connection was successful. (1)

- When inspecting the state *Driving*, there is no way to tell which transition is in question for modification. In the worst case, one has to click each entry in order to find the correct transition (see Figure 6.2). (6)

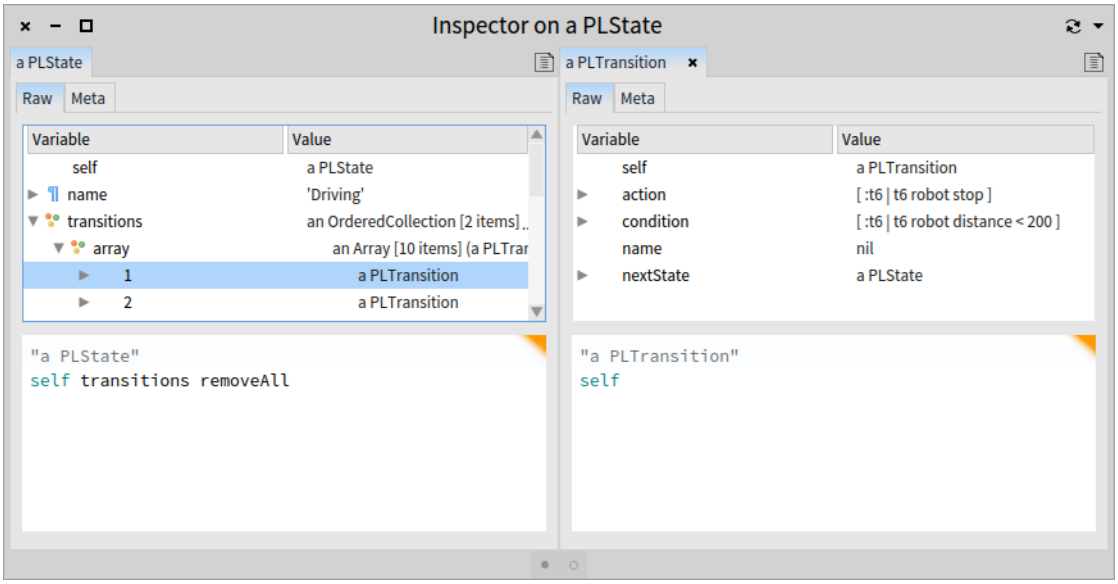

Figure 6.2: The transition of interest is found in array element 1, but no hint points to that entry.

- Individual panes cannot be minimized to get a better look at the visualization. This restricts the user's flexibility interacting with the UI. Resizing the whole window is similarly problematic, the user cannot decide how big each pane is. (7)

1. *Visibility of system status:* The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

2. *Match between system an the real world:* The system should speak the users' language, with words, phrases, and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

3. *User control and freedom:* Users often choose system function by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

4. *Consistency and standards:* Users should not have to wonder whether different words, situations, or action mean the same thing. Follow platform conventions.

5. *Error prevention:* Even better than good error messages is a careful design which prevents a problem from occurring in the first place.

6. *Recognition rather than recall:* Make objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

7. *Flexibility and efficiency of use:* Accelerators – unseen by the novice user – may often speed up the interaction for the expert user to such an extent that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

8. *Aesthetic and minimalist design:* Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

9. *Help users recognize, diagnose, and recover from errors:* Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

10. *Help and documentation:* Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Table 6.1: Revised set of usability heuristics by Nielsen [10]

7
Conclusion and Future Work

As we found in the Heuristic Evaluation, the Lego Playground has its usability issues. Nevertheless the Lego Playground can be a suitable tool for live programming Lego Mindstorms robots and as such fulfills the requirements of Chapter 4.

Software - especially graphical software - is never finished. This is indeed true for integrated development environments where software progression is very well noticeable. There are many approaches to assist programmers in their tasks, from project-wide code refactoring to git integration, static code analysis and debugging tools. With these enrichments happening in popular IDEs, the bar has risen for code editors to satisfy the modern programmer. While these were not the requirements for this project, there are many possible ways to extend the proposed IDE in the future without adding specific examples.

Leaving aside generic IDE tools, there are also possible enhancements specifically targeting our problem domain. One improvement could be migrating the back end to the newer Polite Smalltalk version as proposed by Thomas Steinmann [8]. This most recent version is able to replace standard Smalltalk with the exception of class methods, thus it would be possible to implement the large parts of the back end using Polite itself. As an integration in the Lego Playground, the programmer could extend the basic `PoliteVehicle` with different robot builds thanks to support for classes and inheritance.

8 Anleitung zu wissenschaftlichen Arbeiten

8.1 Tutorial

This section is a tutorial on how to use the Lego Playground. It contains information on how to connect and disconnect an EV3 robot and how to interact with it based on a given example. The main focus lies on the usage of the IDE. For instructions how to prepare the robot and how to install the software please refer to Theodor Truffer's thesis' section Anleitung zu wissenschaftlichen Arbeiten.

8.1.1 Prerequisites

We assume that the EV3 robot is connected to the same wireless network as the computer in use and we know the EV3 brick's IP address. Also, there needs to be an appropriate installation of Pharo present as well as a prepared image with the PoliteSmalltalk project installed. If that's the case, we can start the Lego Playground (which internally is called `PLPlayground`) in Pharo by executing the following command in a workspace using 'Super + D'):

```
PLPlayground open.
```

8.1.2 Connecting via IP

At this point we are presented with the Lego Playground, which we first use to connect to our EV3 robot. This is done via the green arrow in the top bar. We enter the EV3's IP

address and confirm the input. If no error message shows up in the Console field, we are good to go.

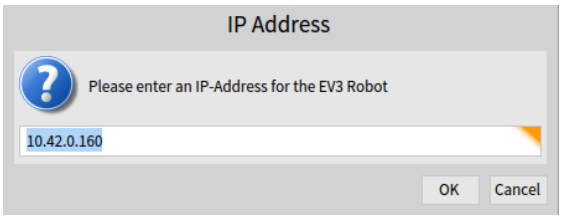

Figure 8.1: Connection dialog asking for an IP address.

8.1.3 An example program

Now it's time to run a program. The example may seem a bit overwhelming, but we will go through it step by step.

```
|start, end, driving, turning, speedup |
start := PLStartState, new Called: 'Start'
   machine Called: 'Treasure Hunter'.
driving := PLState, new Called: 'Driving'.
turning := PLState, new Called: 'Turning'.
end := PLState, new Called: 'End'.

start,
   when: [ :rt | true ]
   do: [ :rt | rt, robot, drive forward.
      rt, context, add Timer: #MachineTimer;
         when: [ :runtime |
            (runtime, context, get Time Of: #MachineTimer) > 100 ]
         do: [ :runtime | runtime, robot, stop ]
         goTo: end;
         when: [ :runtime | (runtime, robot, color = #black) ]
         do: [ :runtime | MyScripts, celebrate.
            runtime, robot, drive Forward. ]
         goTo: driving. ]
   goTo: driving.

driving,
   when: [ :rt | (rt, robot, distance < 200) ]
   do: [ :rt | rt, robot, stop ]
   goTo: turning;
   when: [ :rt | (rt, robot, color = #red) ]
   do: [ :rt | rt, robot, stop. ]
   goTo: end.

turning,
```

```
   do: [ :rt | (rt, robot, turn Random; drive Forward) ]
   goTo: driving.

PLRunTime, processor, start: start.
```

A script called *celebrate* is already saved in the project, so we can simply call it using `MyScripts, celebrate`. In doing so, the Lego Playground will spawn a nested state machine *celebrate* which holds its own states and transitions and return to the *Treasure Hunter* machine once it terminates.

We declare a state machine called *Treasure Hunter* which consists of 4 states:

- start
- driving
- turning
- end

With the states declared, we can start adding transitions to them as well as wildcards.

start The state *start* is a `PLStartState` and thus accepts a machine name, in this case *Treasure Hunter*. *start* is a special state not just because it's used to declare a machine name, it is also the place where we should enter wildcards.

```
start,
   when: [ :rt | true ]
   do: [ :rt | rt, robot, drive forward.
      rt, context, add Timer: #MachineTimer;
         when: [ :runtime |
            (runtime, context, get Time Of: #MachineTimer) > 100 ]
         do: [ :runtime | runtime, robot, stop ]
         goTo: end;
         when: [ :runtime | (runtime, robot, color = #black) ]
         do: [ :runtime | MyScripts, celebrate.
            runtime, robot, drive Forward. ]
         goTo: driving. ]
   goTo: driving.
```

We add a transition to *start* whose condition is `true` and points to the state *driving* as its successor. The main reason for this transition is its action block which we use to define a number of items the *context* takes care of. Namely, we add

- Timer *MachineTimer*
- Wildcards
    1. Wildcard:
        - When: *MachineTimer* is greater than 100 seconds

   - Do: Stop the robot
   - Go to: *end*
2. Wildcard:
   - When: Color is black
   - Do: Run nested state machine called *celebrate* and drive forward afterwards
   - Go to: *driving*

driving The state *driving* is active whenever the robot is just driving forward. It checks if there's an obstacle ahead or if it finds a red spot on the floor.

1. Transition:
   - When: Distance is less than 200 millimeter
   - Do: Stop the robot
   - Go to: *turning*
2. Transition:
   - When: Color is red
   - Do: Stop the robot
   - Go to: *end*

turning All this state does is turning the robot for a random amount of degrees to the left or right. Afterwards it goes back to *driving*.

1. Transition:
   - Do: Turn around randomly
   - Go to: *driving*

8.1.4 Running the example

Having cleared out all the details of this example, we can run it using the green arrow in the top row. The robot should now start moving and the Lego Playground gets updated with information about this state machine: The visualization shows up, the states list gets populated as well as the machines list.

8.1.4.1 Inspection and manipulation

Let's now go ahead and inspect the states. This is done by selecting an item - let's say *Driving* - in the states list and hitting the *Inspect* button. In the opening window, we see all the state's properties including transitions. Navigating to *transitions* $\rightarrow$ *array* $\rightarrow$ *1* $\rightarrow$ *condition*, we can edit the block statement which defines this transition's condition.

We can for example change the distance to 150 millimeters, letting the robot drive closer to obstacles than originally defined. These changes apply immediately, making the manipulation a truly live experience. Of course there are endless possibilities for similar manipulations. Some more examples are:

- Changing Timers in the *Context*
- Adding and / or changing action blocks of transitions and wildcards
- Changing color in condition blocks of transitions and wildcards

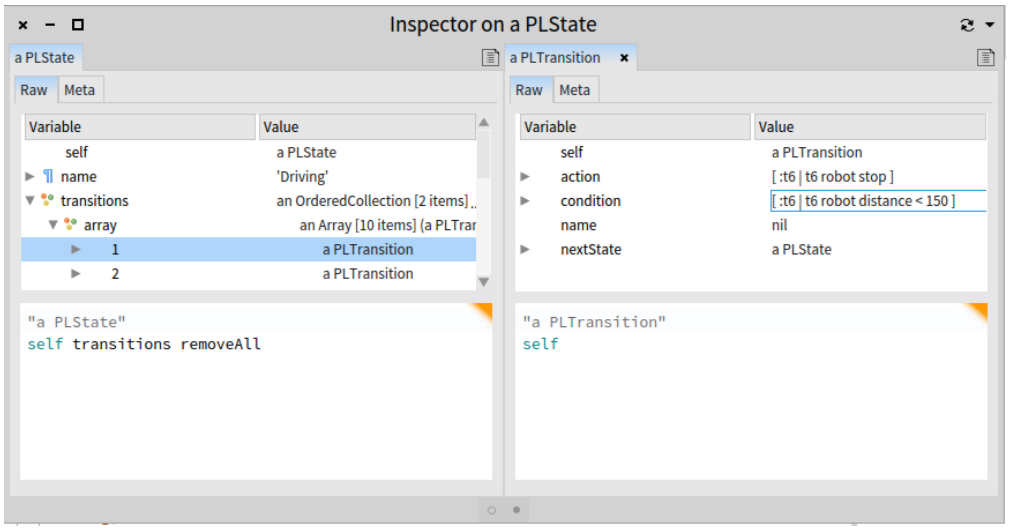

Figure 8.2: Inspector on *driving* used to change its transitions.

8.1.4.2 Nesting

When the robot finds a black spot on the floor, the second wildcard is triggered and thus the nested machine *celebrate* is executed. As a result, the state list consists solely of the new machine's states, the machine list features an new top element and the visualization pane has a new drawing to it.

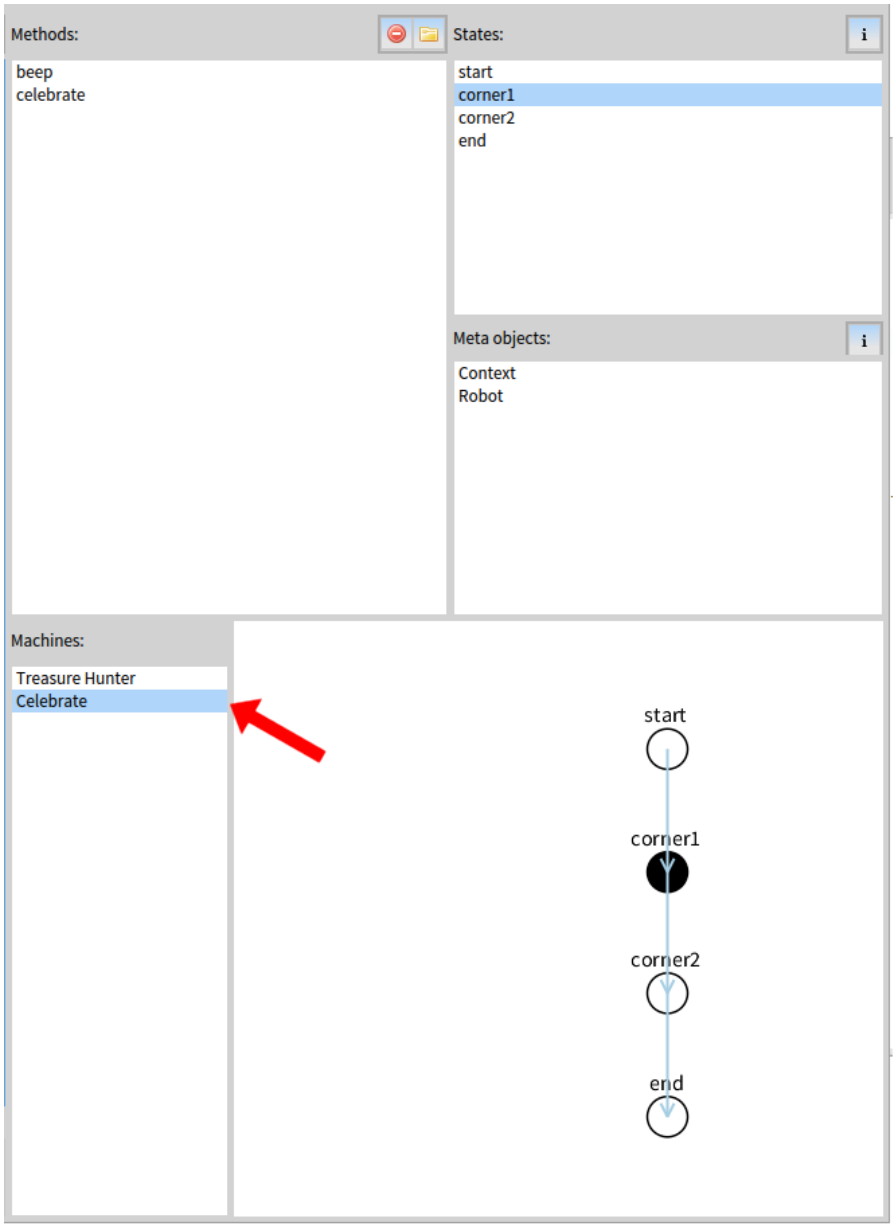

Figure 8.3: The updated view in `PLRightPane`, featuring states, an additional machine list item and fresh visualization of the nested machine *Celebrate*.

8.1.4.3 Reaching *end* and disconnecting

As we defined in the program, the state *end* is reached if the timer reaches 100 seconds or a red spot is found under the color sensor, assuming these transitions have not been altered by the programmer during execution. Once it's reached, the Lego Playground is in a clean state leaving only the visualization.

We can now hit the *Connection* icon and agree to disconnect the robot.

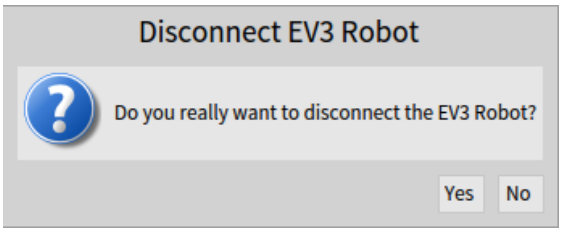

Figure 8.4: Disconnecting the EV3 robot makes it available to other parties.

Bibliography

[1] Johan Fabry and Miguel Campusano. Live robot programming. In Ana Bazzan and Karim Pichara, editors, *Advances in Artificial Intelligence – IBERAMIA 2014*, number 8864 in Lecture Notes in Computer Science, pages 445–456. Springer-Verlag, 2014.

[2] Mircea Lungu and Jan Kurš. On planning an evaluation of the impact of identifier names on the readability and maintainability of programs. *USER*, 13:13–15.

[3] Theodor Truffer. A Polite solution to interact with EV3 robots. Bachelor's thesis, University of Bern, September 2016.

[4] Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127 – 139, 1990.

[5] Miguel Campusano and Johan Fabry. Live robot programming: The language, its implementation, and robot API independence. *Science of Computer Programming*, 133:1–19, 2017.

[6] Mircea Lungu and Jans Kurs. Polite Programmers, Use Spaces in Identifiers If Needed, October 2016.

[7] Jan Kur, Mircea Lungu, Oscar Nierstrasz, and Thomas Steinmann. Polite Smalltalk - An Implementation, September 2016.

[8] Thomas Steinmann. Adding class support and global methods to Polite Smalltalk. Bachelor's thesis, University of Bern, May 2016.

[9] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 249–256. ACM, 1990.

[10] Jakob Nielsen. Heuristic evaluation. *Usability inspection methods*, 17(1):25–62, 1994.