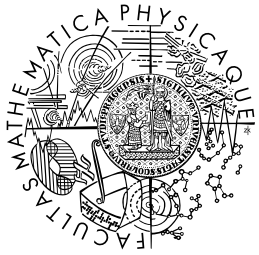

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Ondřej Staněk

Visual Programming Backend for a Mobile Robot

Department of Software Engineering

Supervisor of the master thesis: RNDr. David Obdržálek, Ph.D.
Study programme: Informatics
Study branch: Software Systems

Prague 2017

I would like to thank my supervisor, RNDr. David Obdržálek, Ph.D., for supporting me over the years in the field of robotics. His advice and comments helped to shape this thesis and I am grateful for his feedback.

I would like to dedicate this work to my beloved grandfather Standa, who led my first steps in electronics and programming.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb.., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague, date ....

Název práce: Podpora vizuálního programování mobilního robota

Autor: Bc. Ondřej Staněk

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Obdržálek, Ph.D.

e-mail vedoucího: David.Obdrzalek@mff.cuni.cz

Abstrakt: V této práci se autor zabývá návrhem a implementací řešení pro programování malých mobilních robotů pomocí vizuálních programovacích prostředků. Součástí práce je výběr vhodného front-endu pro vizuální programování i vytvoření back-end vrstvy umožňující běh programu v mobilním robotovi. Pro vykonávání kódu je vytvořen virtuální stroj, který běží v rámci původního firmware robota na 8-bitovém mikrokontroleru s omezenými prostředky. Vrstva generátoru kódu překládá vizuální reprezentaci programu do sekvence instrukcí bajtkódu, jež je následně interpretována v mobilním robotovi. Řešení podporuje typické rysy procedurálních programovacích jazyků, zejména: proměnné, výrazy, podmíněné příkazy, cykly, statická pole, funkční volání a rekurzi. Důraz je kladen na robustnost implementace. K ověření a udržení kvality kódu jsou použity metody automatického testování.

Klíčová slova: vizuální programovací jazyk, virtuální stroj, mobilní robot, Blockly

Title: Visual Programming Backend for a Mobile Robot

Author: Bc. Ondřej Staněk

Department: The Department of Software Engineering

Supervisor: RNDr. David Obdržálek, Ph.D.

Supervisor's e-mail address: David.Obdrzalek@mff.cuni.cz

Abstract: In this work, the author designs and implements a solution for programming small mobile robots using a visual programming language. A suitable visual programming front-end is selected and back-end layers are created that allow execution of the program in a mobile robot. The author designs and implements a virtual machine that runs alongside the original robot firmware on an 8-bit microcontroller with limited resources. A code generator layer compiles the visual representation of the program into a sequence of bytecode instructions that is interpreted on board of the mobile robot. The solution supports typical features of procedural programming languages, in particular: variables, expressions, conditional statements, loops, static arrays, function calls and recursion. The emphasis is put on robustness of the implementation. To verify and maintain code quality, methods of automated software testing are used.

Keywords: visual programming language, virtual machine, mobile robot, Blockly

Contents

1 Introduction **4**
1.1 Motivation 5
1.2 Organization of the Thesis 5

I Analysis **6**

2 Existing Solutions **7**
2.1 Wonder Workshop 8
2.2 CodeBug 9
2.3 Event-driven Programming 10

3 Visual Programming Languages **13**
3.1 Google Blockly 13
3.2 MIT Scratch 15

4 Target Robotic Platform **16**
4.1 Feature Overview 16
4.2 Resources Analysis 19
4.2.1 Program Space 19
4.2.2 Operation Memory 19
4.2.3 Performance 20
4.3 Functional Specification 20
4.3.1 Free Movement 20
4.3.2 Line Following 21
4.3.3 Signalization LEDs 22
4.3.4 Button 22

5 Virtual Machines and Interpreters **23**
5.1 Implementations for Embedded systems 24
5.1.1 EmbedVM 24
5.1.2 NanoVM 24
5.1.3 AmForth 25
5.2 Stack Computers 25

5.2.1 Reverse Polish Notation . . . 26
5.2.2 Forth Programming Language . . . 26

6 Test Automation **28**
6.1 Testing Frameworks . . . 29
6.1.1 CppUTest . . . 30
6.1.2 Check . . . 30

II Design **31**

7 Visual Programming Language **33**
7.1 Front-end . . . 34
7.2 Data Types . . . 34
7.2.1 Universal Data Type . . . 35
7.2.2 Collections . . . 40
7.3 Concurrency and Synchronization . . . 41

8 Testing **42**
8.1 Unit Testing . . . 42
8.1.1 Executing Tests on Target Architecture . . . 44
8.2 Integration Tests . . . 44

III Implementation **47**

9 Virtual Machine (VM) **48**
9.1 Scheduling and Preemption . . . 48
9.2 Virtual Memory Map . . . 51
9.2.1 Program Space . . . 51
9.2.2 Stacks . . . 52
9.2.3 RAM . . . 52
9.3 Input/Output . . . 52
9.3.1 Shared Memory (Register File) . . . 52
9.3.2 System Calls . . . 54
9.3.3 Synchronization on Events . . . 55
9.3.4 Movement Commands . . . 56
9.4 Instruction Set . . . 56
9.4.1 Push Immediate . . . 56
9.4.2 Operators . . . 57
9.4.3 Jumps . . . 58
9.4.4 Conditional Branches . . . 59
9.4.5 Memory Addressing . . . 60
9.4.6 Data Stack Manipulation . . . 60

9.4.7 Subroutines 61

10 Code Generation **63**
10.1 Intermediate Assembly Language 63
10.2 Bytecode 64
10.3 Library Functions 64
10.4 Generating Code 64
10.4.1 Expressions 65
10.4.2 Variables 66
10.4.3 Conditional Statements 67
10.4.4 Loops 68
10.4.5 Arrays 71
10.4.6 List 72
10.4.7 Functions and Procedures 74

11 Integration Test Suite **77**

Conclusion **78**
Text Summary 78
Future Works 79

Bibliography **81**

List of Abbreviations **85**

Chapter 1

Introduction

We are surrounded with technology. Almost every electronic device around us embeds a computer program - a wrist watch, a cellphone, a washing machine, a thermostat, a credit card... The untouchable yet crucial components of all these devices are computer programs that govern their operation. With the emerging era of the Internet of Things (IoT), programmable electronic devices become even more inevitably part of our lives.

The world is changing and this process could be also reflected in the curricula for schoolchildren. Children are introduced to science subjects to help them understand what the word around *is* and how it *works*. Nowadays, when programmed devices become part of a modern world, basic coding skills could be added to the curriculum to teach children also about this new aspect of the modern world. The unveiling of programming concepts to children could help them realize that all the technology they see around is based on simple concepts that they can grasp and practice.

Coding is a skill that becomes relevant in more and more professions that rely on IT in some way. Coding, scripting, programming - whatever we call it - is a tool to achieve automation of some routine tasks, that would be performed manually otherwise. A secretary, an accountant, a researcher, a banker or a businessman could benefit from little scripts that he or she writes for their own use to achieve their goals faster and in more efficient manner. There is a merit in making coding accessible outside the realm of professional software developers. There is much wider audience that may benefit from the software tools that drive todays world. We see this happening, there are projects around that open coding to general public [15, 3], and this effort is likely to continue in future.

The purpose of this work is to make the coding accessible and fun. We propose a new tool that could be used in the educational system.

1.1 Motivation

Learning coding might be challenging for novices. At their first attempts, all they might see is just a bunch of errors reported by the computer. Often, this is discouraging. If we look carefully, the errors are often of syntax nature. The textual representation of a computer program imposes quite high formal syntax requirements on the novice programmer and takes his mind away from the real problem solving. However, it is the problem solving that should be taught, not the nuances of a particular programming language. Visual programming languages address this issue nicely - they are constructed in a way that no syntax error can ever occur, which makes the learning process significantly easier. Specially, visually appealing interface might help children to start putting together their first programs prior they are sound in typing (or even reading).

The question is, what the program will do? Well, this depends on the platform where we are programming. It might control a virtual character on a computer screen, but as well, the program could drive a toy in the real world!

In this thesis, we build a solution that enables visual programming of a mobile robot, so that the programs and algorithms can be tested in real environment.

1.2 Organization of the Thesis

This thesis is divided into three parts.

Part I is an analysis of existing solutions and approaches. Chapter 2 describes existing robotic toys enabled with a visual programming interface and the event-driven programming paradigm is discussed. Visual programming editors are analyzed in Chapter 3. Chapter 4 defines the target robotic platform for which a specific visual programming language will be developed. In Chapter 5, the existing implementations of virtual machines are analyzed, with a focus on the general concept of stack machines. The importance of test automation is explained in chapter Chapter 6 and some testing frameworks are suggested.

Part II "Design" builds on the background research done in the Analysis part and presents the design of the solution. Chapter 7 proposes a visual programming language for the defined robotic platform and the features and properties of the visual programming language are specified. Chapter 8 suggests a method of automated testing and provides guidelines for implementation of the automated test suite.

Part III contains detailed description of the proposed Virtual Machine (Chapter 9) and Code Generator (Chapter 10) implementation, including numerous code examples. Finally, the implementation details about the automated test suite are mentioned in Chapter 11.

Part I

Analysis

Chapter 2

Existing Solutions

On the market, there are several solutions that enable visual programming of toy robots. Lego Mindstorms originated in 90s and introduced a programmable block that could execute programs designed visually on a PC. With the boom of smartphones, tablets and connected toys, there are many new competitors in the STEM[1] educational field. A reliable wireless link between a robot and a smartphone/tablet allowed offloading the computation from a robot to a smartphone or tablet. We can classify the programmable toys into two categories; *standalone* and *connected*. Standalone programmable toys execute the user program on the board of the robot - they are independent. Connected programmable toys always need a supplementary device that executes the user program.

Both approaches have their pros and cons. Tablets may have many times bigger computational performance than the robot hardware itself, so there are almost no limitations for the feature set of the visual language. Debugging of a visual program that executes in a tablet will be generally a simpler problem than debugging a program running in the real robot. The downside is that the user is not really programming a robot, but a tablet that controls a robot. From the pure educational standpoint, there is a value in showing that robot can execute the program independently. Such option opens new possibilities and is more versatile. The child can create a program for the robot, and once the program is loaded to the robot, the tablet is not needed anymore. The robot can be transported and used outdoors. There can be multiple robots programmed with the same or different programs, each performing its program autonomously.

We do an analysis of two existing solutions, one being a *connected* toy, the second a *standalone* programmable platform. There are many more solutions, such as Lego Mindstorms [17] or Fishertechnik [9], but we will focus only on products that use an open-source visual programming front-end.

[1]STEM: science, technology, engineering and mathematics

2.1 Wonder Workshop

Dash & Dot are toy robots from the Wonder Workshop company [22]. The mission of the Dash and Dot robots is to make learning to code fun. Dash is a mobile robot that can travel around, while his companion Dot is a stationary robot. Both robots are equipped with a speaker, microphones, infra-red sensors and attachement points for accessories. Dash & Dot are connected toys; they can be controlled from a tablet or smartphone wirelessly, over BLE[2]. Wonder Workshop supplements their robots with several apps for controlling and programming the robots. The apps are designed to teach childern the concepts of coding.

The *Xylo* is a music app where children can compose songs. The interface of the app is very visual and intuitive; the child touches the xylophone tone bars that he or she wants to play at the selected time instance. This way, the child programs a sequence of tones. A concept of loops is also introduced - user can configure any part of the song to be repeated n-times. After the song is composed, the Dash robot will re-play the song on a small xylophone, which comes as an accessory.

The *Go* application controls the movement of the robot. The child can plan a trajectory for a robot by finger-drawing. The designed movement is then performed by the Dash robot. There are also controls for real-time driving available.

There is an application called *Wonder* that introduces children to state machine programming. The user can design custom finite state automaton that controls the robot. The automaton is represented as graph consisting of states (vertices) and transitions (edges). The states are actions that the robot will perform, such as "set motor speeds", "stop motors", "move forward 10cm", "rotate 90 degrees", "play sound" etc.. The edges signalize the allowed transitions between the states, which could be "obstacle detected", "button pressed", "clap heard", etc.. There is a visualization during the state machine execution - the state that is currently being performed is highlighted.

The Wonder Workshop *Blockly* application teaches procedural programming. The user puts together a program in a visual editor based on Google Blockly (see Section 3.1). The Wonder Workshop visual programming language is event-driven. The programmer can assign event-handling code to each event, and the code is executed when the event arrives. An event can be a press of a button on the Dash or Dot robot, detection of an obstacle or acoustic stimulus such as voice or clap. The most usual entry point of a program is the "Start" event, which triggers when the "Play" button in the Blockly application is pressed.

The programing paradigm is event-driven and the concurrent execution of event handlers is not allowed.

Citation from Wonder Workshop FAQ: "Right now, Blockly does not allow concurrent block execution." [23]

This means that only one event handler can execute at a time. This restriction

[2]Bluetooth Low Energy

Figure 2.1: Dash & Dot - Blockly editor (screenshot)

is common for event-driven programming, and the same restriction applies to JavaScript, for instance.

The Wonder Workshop Blockly programming language has limited support for variables. The child can use up to 5 global variables, conveniently distinguished by pictures of fruit. Each variable can hold an integer number ranging from -9999 to 9999.

There is a basic support for user defined procedures. So far, there is no support for procedure arguments. User defined functions[3] are not supported.

The user program is executed in the tablet, not in the robot itself. [24] *"Programs do not need to be compiled and downloaded on to the robot; they are meant to run in real-time, allowing for easier learning and debugging."* [25]

2.2 CodeBug

The CodeBug [6] is a programmable wearable device. It features a 5x5 LED[4] array to display simple icons, animations or scrolling text. The programs for CodeBug are created in an online visual editor based on the Blockly library. The user programs are compiled to a binary executable. The compiled user program is to be loaded to the CodeBug via USB cable. The CodeBug MCU[5] is preprogrammed

[3] a callable unit that returns a value
[4] Light-Emitting Diode
[5] microcontroller unit

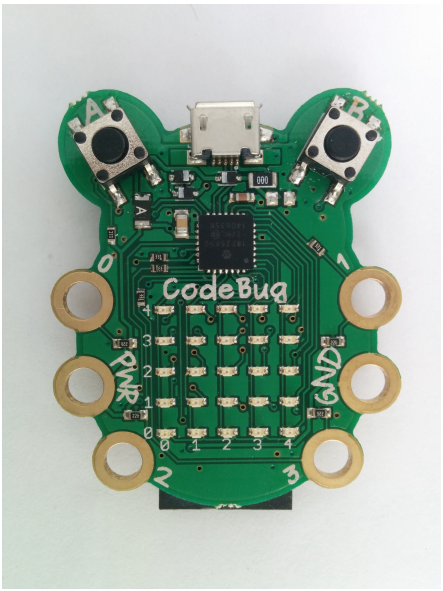

Figure 2.2: CodeBug (source: `https://www.codebug.org.uk/media/codebug_front.jpg`)

with a bootloader. The bootloader can make the CodeBug act as a USB drive when button is held during connection to a PC. The user program is then copied on the drive and executed. Once loaded, the user program can run independently of the PC, which is unique to other solutions which usually interpret the user program in tablet, smartphone or PC.

The CodeBug online editor has a simulation engine to run the user programs in the web browser. This allows convenient debugging of the code that is executed in the simulator. The simulator can also simulate hardware accessories that extend the features of CodeBug in terms of number of LEDs.

The visual programming language supports variables, expressions, conditional statements, loops and input/output functions. To date, user defined functions and collections are not supported.

2.3 Event-driven Programming

Event-driven programming is a paradigm in which the program is composed of event handling routines. When an event occurs, the appropriate handler routine is executed. This action-reaction principle is well suited for describing behavior of a mobile robot that reacts to inputs from its environment. The event driven approach is also often used in GUI applications, where events are generated by user's interaction with the graphic control elements in the application. An example of event driven language is JavaScript, which is widely used in web-based interactive applications.

The principle of asynchronous processing of events might suggest that the

event handlers are executed in parallel, however, this is not the case for many event-driven architectures. The aforementioned JavaScript implements event handling in a single thread, which guarantees that event handlers are always executed in series, and as such, there is no need to introduce synchronization. JavaScript is special in a way - because the API is purely asynchronous, no function can ever block. If a programmer misuses some event handler to perform intensive computation (or worse, an infinite loop), the model of event dispatching fails, as the event loop is blocked and no more events can be served by the system.

On Android platform, the GUI model also relies on event handlers, which are processed in a single thread (called UI thread). It is also required that the GUI elements are only manipulated from that single thread. This design ensures that all routines that interact with GUI are always serialized and executed from the single UI thread, and therefore, no synchronization or locking is required. Needless to say, the programmer must ensure that routines running on the UI thread do not block, otherwise, the whole GUI becomes unresponsive. This example shows that the single event loop architecture is used even on systems that have full support for threading and synchronization (Android is programmed in Java).

Citation: "While it is generally possible to apply multithreading, most of the aforementioned frameworks and platforms rely on a single-threaded execution model. In this case, there is a single event loop and a single event queue. Single-threaded execution makes concurrency reasoning very easy, as there is no concurrent access on states and thus no need for locks." [29]

As pointed out, the single-threaded execution model is widely used on many platforms because of its simplicity. Unfortunately, such model is not well suited for programming mobile robots. Let's consider an example of a simple program that makes a robot to go in a square. Assume that the API has a blocking *MoveForward(distance_cm)* and *Rotate(angle_deg)* routines that block the program execution until the robot completes the movement.

```
var count = 4;
while(count > 0) {
        MoveForward(10);
        Rotate(90);
        count = count - 1;
}
```

Such code is simple and should be easy to explain to a child. Now let's imagine how such behavior would be programmed if the API for controlling movement of the robot was non-blocking (asynchronous). In Javascript, the asynchronous versions *MoveForward_async(distance_cm, callback)* and *Rotate_async(angle_deg, callback)* take as argument a callback function that is performed after the movement command is finished. The JavaScript non-blocking implementation could look like this:

```
var count;
```

```
var callback1;
var callback2;

callback1 = function() {
        if(count > 0) {
                count = count - 1;
                MoveForward_async(10, callback2);
        }
}

callback2 = function() {
        Rotate_async(90, callback1);
}

count = 4;
callback1(); // makes the robot to go in a square
```

It is obvious that the callback API is much more complex to comprehend and use. We believe that API for robot control should use the blocking approach. However, this means that the event handling cannot be implemented on a single thread anymore (as JavaScript event loop or Android UI thread do). Because we introduced a blocking API, the event handlers don't finish immediately any longer, and there can be more event handlers in a process of evaluation at a time. Spawning separate threads for each handler would solve the problem, but at the same time it would add the complexity of a multi-threaded application, with all the inherent synchronization concerns. Alternatively, an asynchronous pattern such as async/await [31] could be used to ease the troubles of asynchronous programming.

Another alternative would be to use a completely different modeling tool which is better suited for describing concurrency. Grafcet [50, 30], for instance, is an industrial solution derived from a theoretical concept of Petri nets. If we could refrain from the procedural programming paradigm, it would be a viable alternative for visual programming of mobile robots.

To conclude, the event-driven programming paradigm can be achieved on a single-threaded system as long as the API doesn't block. Specially, the API must not contain the *pause()* method that waits for a specified amount of time. If there is a function that blocks the execution, the event-driven programming paradigm can be still used, but requires concurrent execution of the event handlers, which can be achieved by threads and context-switching.

Chapter 3

Visual Programming Languages

Visual programming is a broad term that covers various approaches. Some visual programming languages use diagrams to model a dataflow (such as MATLAB Simulink or LabVIEW), other depict graphically a state machine or a Petri net (Grafcet [50, 30]). In this thesis, we will focus on educational visual programming languages, which introduce children to a concept of procedural programming dressed in a graphical interface.

3.1 Google Blockly

Blockly [12] is a web-based library for building visual programming editors. It is an open-source project licensed under the Apache 2.0 License [2]. The library is programmed in JavaScript and runs client-side. It supports major web browsers including Chrome, Firefox, Safari, Opera and Internet Explorer, as well as mobile touchscreen devices.

The visual program is made of basic blocks that interconnect together in similar manner as puzzle pieces. The interconnection mechanism enforces semantical correctness of the program. Blocks can be only connected in a way that makes semantic sense. This way, the user can never make a syntax error and he can focus solely on the logic of the program. The program can be then exported to JavaScript, Python, PHP or Dart. Blockly generates well formated, syntactically correct code. Basically, the library translates the visual program (blocks) into a syntactically correct code (text).

The library offers set of basic blocks and it can be extended with a new custom blocks. There is a Block Factory [13] tool for designing new blocks. Interestingly, the Blockly Factory itself is based on the Blockly visual programming language.

Blockly is designed for dynamically typed languages. As such, Blockly doesn't enforce static type checking for variables. There is a strong parallel with JavaScript, and Blockly perform very well in exporting visual program to JavaScript code or similar scripting languages. Variables in Blockly can hold any data type same as JavaScript variables would.

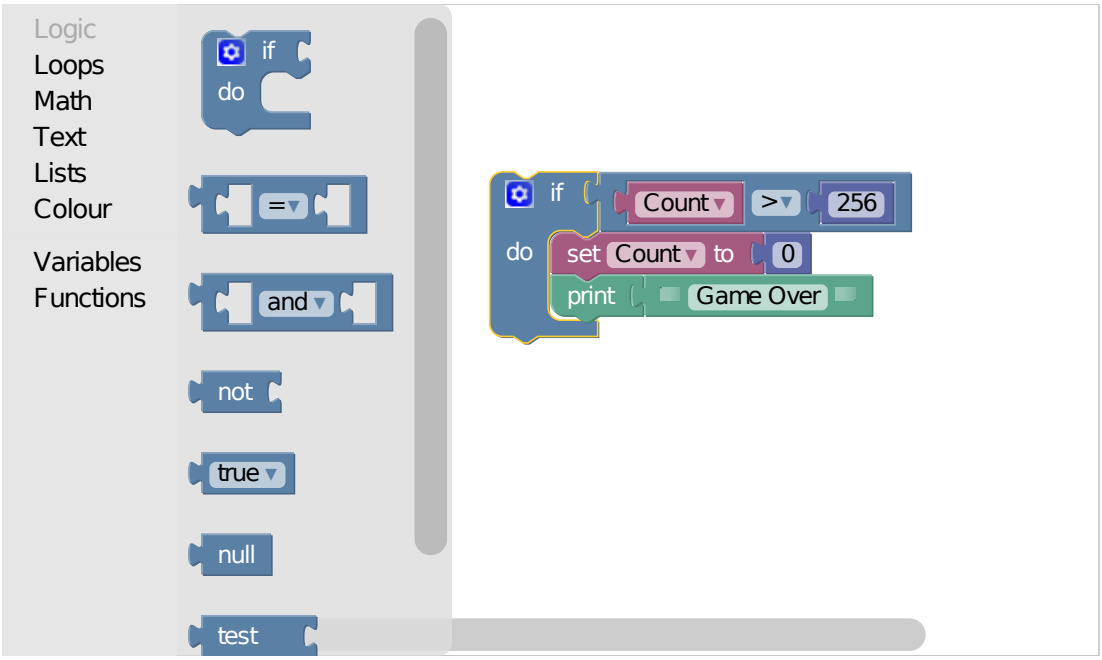

Figure 3.1: Blockly Library [12] (screenshot)

However, Blockly does have a support for minimal type checking, which can be extended further. The Open Roberta [20] project extended the Blockly type checking to support statically typed languages. User declares variables and their types at the beginning of the program and the editor then enforces all the type checks statically, at design time. In the Open Roberta editor, the types are visualized by colors. For instance, the editor doesn't allow to connect `number` (blue) block to an input where a `string` (green) is expected.

Blockly library supports functions and procedures. User can specify arguments that the function or procedure will accept. The type of arguments or return value is not specified and Blockly doesn't put any restriction on data types. This is reasonable, as the target class of scripting languages doesn't require static type checking neither. In many ways, Blockly reflects the features of its target scripting languages. Unfortunately, there is one shortcoming that relates to scope of variables. Blockly currently supports global variables only. Local variables in functions cannot be declared. This is a significant drawback especially when it comes to recursion. The absence of local variables significantly limits the possibilities of recursive function calls. There are various ways how Blockly could be extended to support variable scoping, which was discussed thoroughly in the Blockly forum [4]. For example, the Open Roberta approach of declaring variables could be used. To date, however, the Open Roberta editor doesn't have support for functions or procedures.

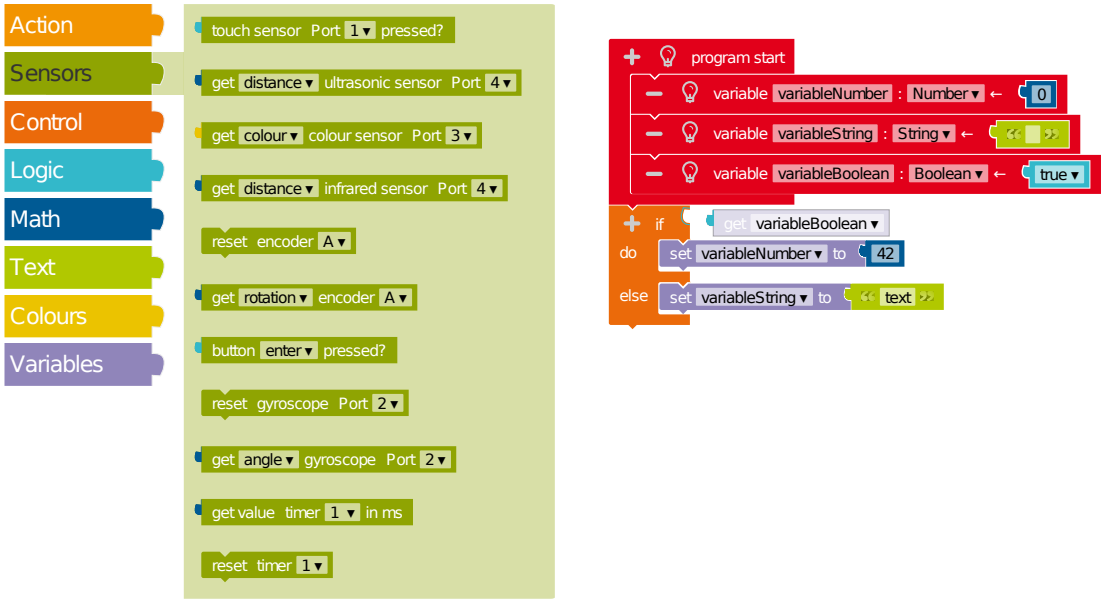

Figure 3.2: Open Roberta Lab [20] - Blockly based editor with static type checking (screenshot)

3.2 MIT Scratch

MIT Scratch [18] is an online visual programming environment. It is a platform built around an event-based visual programming language. Apart from the visual program editor, it offers set of tools for creating and editing multimedia content. The user can design entities called "sprites" that perform in a virtual environment ("scene"). Each sprite on the scene can be scripted in the Scratch visual programming language. The user can execute the scripts in a simulated environment within a web browser.

The Scratch visual programming language is event-based. Behavior of a sprite is driven by one or more event handling routines, which can trigger f.e. when the simulation is started, when key is pressed, when there is a mouse interaction etc... Specially, the sprite entities can communicate with each other by message passing. Receiving a message is an event that can be handled by the sprite's script. This opens possibilities of agent-based modeling.

Chapter 4

Target Robotic Platform

Mobile robots are widely used for educational purposes. The smaller the robot is, the less space is required for its operation. In classroom settings, small robots that can operate on student desks are often welcomed.

The size is also an important factor when it comes to swarm robotics exercised at universities. Smaller robots are easier to manipulate, they occupy significantly less space if used in bigger quantities, require less power for their operation and generally they are safer. In case of improper handling or user program failure, the damage caused by a mobile robot is likely to be proportional to the energy it can dissipate.

There are many competitions organized that motivate students and hobbyists to build mobile robots. Especially line-following contests are very popular among novices in robotics. Line-following robots are usually based on low-performance 8-bit MCUs, which are fully sufficient for the task.

The line following robots use simple sensors and the computational requirements are low, so the robots can be compact and inexpensive. There are minimalistic line-following robots [27, 38, 41, 43, 39] that can fit into a match-box[1] and yet they are programmable, some of them even feature obstacle detection and wireless communication interface [38, 41, 43, 39].

4.1 Feature Overview

In this section, we will summarize the features typical for small line-following robots.

Robots have two individually propelled wheels, providing a differential steering capability. They are equipped with a sensor module that detects the position of the line. There is a button [27, 43, 39] for controlling the robot operation. Optionally, the robot can detect the color of the surface [39]. There might be a uni-directional [38, 41, 43] or a bi-directional [39] wireless communication interface present.

[1] inner dimension 48 × 32 × 12 mm

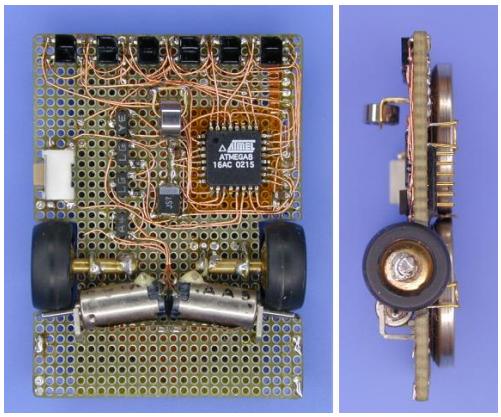

Figure 4.1: ChaN's Desktop Line Following Robot, 2004, [27]
(source: `http://elm-chan.org/works/ltc/rp/ltc02.jpeg`)

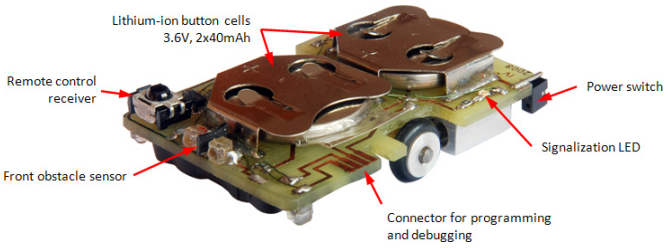

Figure 4.2: OStan's PocketBot, 2008, [38]
(source: `http://ostan.cz/pocketBot/images/top_view_descr.jpg`)

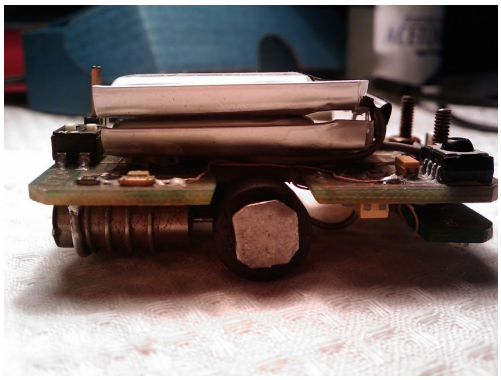

Figure 4.3: Twinsen's PocketBot, 2009, [41]
(source: `http://twinsen.info/img/portfolio/2/2.jpg`)

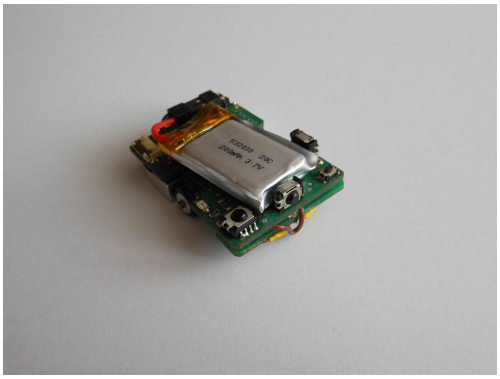

Figure 4.4: Bodie's Fuzee, 2011, [43]
(source: `http://bodie.xf.cz/img/fuzee/2.jpg`)

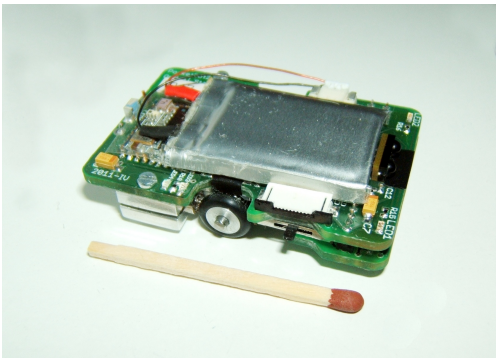

Figure 4.5: OStan's PocketBot2, 2011, [39]

4.2 Resources Analysis

This section is of a big importance, as it describes the hardware limitations that must be taken into account in design phase of the Virtual Machine. The Virtual Machine implementation should be able to run on an 8-bit microcontroller (MCU) comparable to the Atmel ATmega8, as used in projects [27, 43, 38, 41]. The resources of such MCU are typically limited to few kilobytes of program memory (FLASH), single kilobytes or even less amount of runtime data memory (typically SRAM) and clock frequency of typically 8 to 20 MHz. The embedded system has only a single thread of execution. No real-time operating system is expected due to very limited resources of the MCU. It is expected that all the tasks performed by the firmware are scheduled statically.

This section analyses the typical robot firmware in terms of functionality, interfaces and resource requirements. We will start with evaluation of the free resources available to the Virtual Machine. The considered 8-bit MCUs have Harward architecture, where the storage for program instructions and data are physically separated.

4.2.1 Program Space

The compiled firmware is stored in non-volatile FLASH memory. The available free Program Space on a given robotic platform can be found at compile time. On the reference platform, there is 6kB of free FLASH memory available. It is expected that the Virtual Machine implementation should not exceed 5kB limit for the FLASH space. At least 1kB of flash should be left free for debugging routines or future firmware changes. This footprint limitation will impose some trade-offs on the Virtual Machine design.

4.2.2 Operation Memory

The Virtual Machine should be able to run on MCUs that offer as little as 1kB of SRAM. It has to run alongside with all the functionality implemented in the original robot's firmware, which needs some SRAM for its own operation. We can assume that the system doesn't use dynamic allocation of memory. It is quite usual that heap memory is not used at all on small embedded systems. As such, the only memory that is allocated at runtime is on the stack. The worst-case stack size of the reference robot firmware was analyzed to be 80 bytes. The size of the data segment is known at compile time. It is expected that there will be always at least 500 bytes of SRAM available for the Virtual Machine. However, the Virtual Machine should be able to use more memory if there is free RAM space available on the target platform.

4.2.3 Performance

The third important measure are the available computational resources. The MCU has an 8-bit architecture and it is clocked at frequency as low as 8MHz. There is no hardware support for floating point numbers (FPU[2]), therefore, any operations on floats are costly. The worst case CPU utilization of the reference firmware was measured to be 76%. The tasks carried out by the firmware are periodic. That means the load to the core occurs in periodic chunks, which are interleaved with quiet periods when the MCU is in idle-mode. The tasks are scheduled statically, one strictly after another within one period of the main loop. At every iteration of the main loop, the MCU core processes first all pending tasks and then turns into idle mode until next period arrives. The Virtual Machine should take use of these idle periods at the end of each main loop cycle to execute the user program. This way, the Virtual Machine operation will not interfere with firmware tasks that are already scheduled in the firmware. For the sake of completeness, we have to mention there are several interrupt handlers that trigger asynchronously, but the overall utilization due to interrupts is neglectable.

4.3 Functional Specification

In this section, we point out the main aspects of the line-following robot that the Virtual Machine (and user program) will interact with. The previous section listed the hardware limitations imposed on the Virtual Machine, while this section lays out feature requirements on the Virtual Machine driven by a functional capabilities of the robots. These feature requirements are derived from fundamental functions of the robot, such as line-following, handling of external events, free movement and others. The Virtual Machine (and the user program executing within) should have full control over the robot. When executing, the user program will override the robot's implicit behavior. Still, the user program will take advantage of the drivers and modules already implemented in the firmware of the robot.

We list the functional requirements on the Virtual Machine in this section.

4.3.1 Free Movement

Mobile robots can move in their environment. A simple interface for controlling a differential driven robot is to set directly the speed of left and right wheel. Another option is to issue basic movement commands parametrized with speed and distance or angle (move forward, rotate in place).

[2]floating-point unit

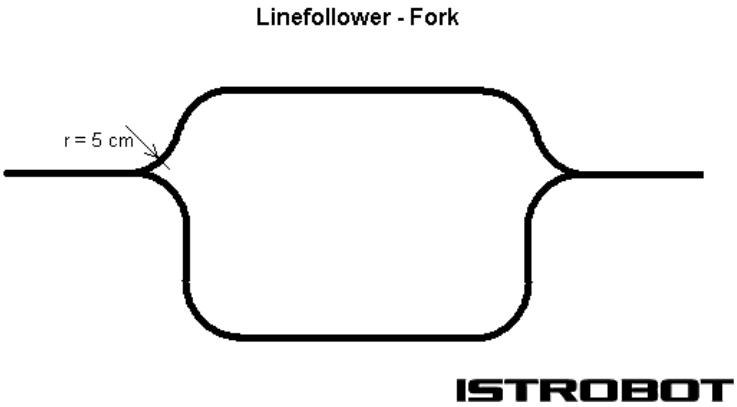

Figure 4.6: Istrobot Line-Following competition allows line forks on the race track. One path can be shorter than the other. [16]
(source: `http://www.robotika.sk/contest/2016/images/LinefollowerFork.png`)

4.3.2 Line Following

Line-following robots can move automatically, guided by a path marked on a surface. There are many competitions in the line-following discipline. The objective is to finish the race in shortest time possible. Some competitions make the discipline more challenging by placing obstacles on the track, introducing line forks, changes in line color etc... If the rules of the particular competition allow it, it is beneficial to optimize the race strategy ad hoc, for each individual race track. The speed of line-following is a major factor that can be optimized for individual parts of the track. Various landmarks on the tracks can hint where the line-following speed should be adjusted for best performance of the racing robot. The need for quick and easy modification of the robot's control program is another motivation for introducing the visual programming language.

It is expected that the robot's firmware can control the line-following speed and handle line forks. Optionally, the firmware might have functions implemented to recognize significant track characteristics, such as line color, curvature change marks [14] or other checkpoints placed on the race track. Then, the user program can adjust behavior of the robot accordingly:

Citation: "Difficult segments of the track can be marked in red, so the robot knows it had better to slow down. In the same manner, the straight segments can be marked with a blue tape, indicating that the speed can be increased safely." [39]

When robot is following a line, the Virtual Machine (and user program) can access and alter these:

- line-following speed
- detect line forks
- choose way on line forks

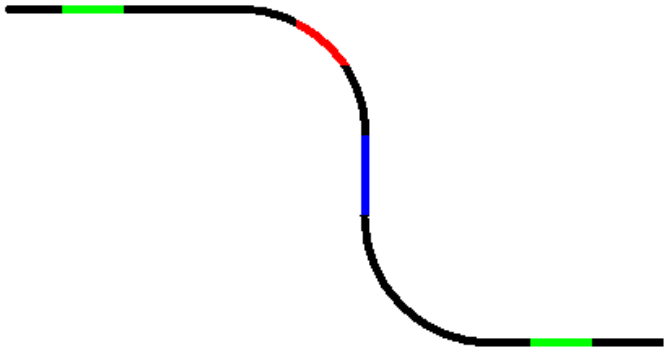

Figure 4.7: The color of the line can change to red, green or blue. Istrobot [16] (source: `http://www.robotika.sk/contest/2016/images/LinefollowerColor.png`)

- recognize color of the line
- recognize checkpoints on the track
- recognize line position and width
- read line sensors

4.3.3 Signalization LEDs

Robots are usually equipped with one or more LEDs for visual signalization. It is expected that the LED brightness can be controlled from the firmware. The user program should be able to control the signalization LEDs of the robot.

4.3.4 Button

In case there is button installed in the robot, the user program should be able to detect button presses. Typically, the button will be used to start / stop the function of the user program, or to pass some information to the running user program, such as selected operation mode, race strategy, etc.. Button press is a simple external event that could be used as a synchronization point of the user program with real environment - as a simple program breakpoint, for example.

Chapter 5

Virtual Machines and Interpreters

Virtual machine (VM) is a concept of an abstract processor or computer system that executes some form of computer program, usually referred to as *bytecode*[1]. The *bytecode* accepted by the VM is defined by its instruction set. The particular *virtual machine* can be implemented on various platforms where it provides standardized and isolated execution environment for its *bytecode*. The concept of virtual execution environment allows to restrict memory access and manage CPU time and other resources that will be used by the user program running within the VM. In many implementations, the virtualization often comes at a cost of reduced performance, as the bytecode has to be interpreted on the target platform.

Virtual machines often incorporate the *stack machine* model. The instructions of such machine operate on the stack and unlike *register machines*, there is no need to specify operands for an instruction. Therefore, the footprint of a *stack machine* code tends to be smaller compared to code for a *register machine*. Code generators for *stack machines* are more straightforward to implement, because *stack machine* can evaluate expressions in postfix notation right away, and no register allocation is necessary.

Interpreter, as opposed to a *virtual machine*, usually processes human-readable program statements and the interpreted languages provide much higher abstraction. Interpreters process the input in text format (ASCII) and therefore they include a parser. Compared to VMs, *interpreters* have often bigger resource requirements, which is also designated by the expressiveness of a language they can interpret.

Both *virtual machines* and *interpreters* can provide isolated execution environment and can manage the resources assigned to the user program. In this chapter, will focus on some implementations relevant to embedded systems.

[1] resembles machine code, but it is universal and portable

5.1 Implementations for Embedded systems

There are several implementations of virtual machines and interpreters that can run on embedded systems; a Python interpreter [21], complete implementation of Java Virtual Machine [28], a BASIC interpreter [44] or a JavaScript interpreter [8] just to list a few. Many occupy tens of kilobytes of program footprint and have relatively high requirements on available RAM. We will focus only on compact implementations that are known to work on low-performance systems comparable to the target platform identified in Section 4.2.

5.1.1 EmbedVM

Embedded Virtual Machine (EmbedVM) [49] is a small virtual machine designed for 8-bit microcontrollers. It emulates a 16-bit stack machine. Integer data types of 8 and 16 bit width are supported. The VM is supplied with a compiler that compiles from a custom C-like programming language to bytecode, that is then interpreted by the VM. The compiler is based on the Flex[2] [10] and GNU Bison parser [11] which makes its own source code very lightweight. The custom programming language supports arrays of the base data types. The instruction set of the EmbedVM is documented, but to date, there is no documentation for the custom programming language that is accepted by the compiler.

There is a set of automated tests to verify the function of the compiler and the VM. The test framework is programmed in *bash* and works on a principle of comparison the expected and actual output of a program under test.

The EmbedVM was ported to the Atmel AVR platform and requires as low as 3kB of program memory. The implementation does not rely on any dynamic memory allocation. All memory accesses of the VM (and the user program running within) are performed through two callback functions, which allows easy sand-boxing of the user code and also gives high flexibility. For example, the virtual memory can be mapped to an external chip. Also, this gives the possibility to assign a memory mapped hardware, which can be operated by the user program.

The user program can perform I/O operations[3], either through memory mapped devices, or by calling a universal user function that accepts variable number of arguments.

5.1.2 NanoVM

NanoVM [19] is a minimalistic implementation of the Java Virtual Machine (JVM) for Atmel AVR microcontrollers. It occupies 8kB of program space and requires 1kB RAM space and 512B of non-volatile EEPROM memory to store the

[2]The Fast Lexical Analyzer
[3]input/output operations

bytecode (user program). It uses the available resources of the Atmel ATmega8 MCU to the fullest extent. NanoVM implements its own garbage collector, supports float numbers, inheritance and includes several native classes for I/O. It runs a subset of the JVM command set. The bytecode for the NanoVM is produced by a standard Java compiler. However, some post-processing of the class files generated by the Java Compiler is necessary in order to run in the NanoVM.

The NanoVMTool is used to load the class files into the NanoVM hosted on an embedded system. When the application is to be loaded to the host, the NanoVMTool searches classpath for all required classfiles. Then, it processes the bytecode from classes to reduce the JVM command set. Finally, the bytecode is loaded to the embedded system where is to be executed by the NanoVM. The execution speed is about 20k Java opcodes per second on an 8MHz MCU. [19]

5.1.3 AmForth

Unlike the EmbedVM and NanoVM, AmForth [1] is an interactive interpreter. It interprets the Forth programming language. The Forth statements (called *words* in the Forth terminology) can be processed in interactive mode, as they are received by the embedded system. Interactive approach is interesting on its own, as it allows easy and effective development and debugging. In compilation mode, AmForth translates the Forth *words* into bytecode, which improves performance of the Forth program.

The AmForth interpreter works on the Atmel AVR Atmega controllers and requires 8 to 12kB of FLASH memory and 200 bytes of RAM. The downside is that the interpreter is designed to run on bare metal and it cannot be integrated into an existing project:

Citation: "Embedding amforth into other programs (e.g. written in C) is almost impossible. Amforth is designed to run stand-alone and does not follow any conventions that may be used on other systems." [1]

However, the Forth language itself has properties that make it well suited for low-performance embedded systems. The most pronounce characteristics of the language is its post-fix notation[4] which greatly simplifies evaluation of expressions.

5.2 Stack Computers

Historically, stack computers played an important role in the early era of digital computers, dating to early 1940's. The Forth programming language designed by Charles "Chuck" Moore initiated an era of second-generation stack computers [34]. The second-generation stack computers had hardware support for the stack-based Forth programming language and were favored in embedded systems due

[4] also known as the Reverse Polish Notation

to their simplicity, fast interrupt response and high code density.

Citation: "Many of the designs for these stack computers have their roots in the Forth programming language. This is because Forth forms both a high level and assembly language for a stack machine that has two hardware stacks: one for expression evaluation/parameter passing, and one for return addresses. In a sense, the Forth language actually defines a stack based computer architecture which is emulated by the host processor while executing Forth programs. The similarities between this language and the hardware designs is not an accident. Members of the current generation of stack machines have without exception been designed and promoted by people with Forth programming backgrounds." [40]

Although the stack computers were superseded in the early 1980's by the RISC[5] architecture, the concept of a stack computer is still used in so called stack-based virtual machines, whose instructions operates on a pushdown stack. Java Virtual Machine (JVM) being one of them.

5.2.1 Reverse Polish Notation

Reverse Polish Notation (RPN) is a parenthesis-free mathematical notation for expressions. The operators are written after their arguments, therefore, the RPN is also known as *postfix* notation. Evaluation of postfix expressions is straightforward to implement on a *pushdown automaton*. The automaton parses input that consists of numbers and operators. When a number is encountered, it is pushed to the stack. Operators pop their arguments from the stack, perform the computation and the result is pushed back on the stack. Unlike infix notation, the postfix notation is unambiguous, and therefore, no parentheses and operator-precedence grammar is required.

5.2.2 Forth Programming Language

Forth programming language is based on the concept of postfix expression evaluation. *A word*[6] in Forth is a generalized concept of a postfix operator that processes arguments passed on stack and pushes the result back to the stack.

Forth is a *concatenative programming language*; every expression in Forth can be seen as a function, and concatenation of expressions is equivalent to function composition. The concatenative property of the language opens possibilities of algebraic manipulation of Forth programs. For example, straightforward code size optimization can be performed. The program is searched for repeated occurrences of an identical code fragment. The repeated code fragment is factored to a new Forth *word*, and all occurrences are replaced with that single new *word*. Concatenative property guaranties that such manipulation can be done mechanically, without any understanding of the context or semantic of the language.

[5] Reduced Instruction Set Computer

[6] analogous to a procedure or function in other programming languages

The Forth uses two pushdown stacks, one *data stack* that is manipulated by the programmer directly, and a *return stack* that is used internally for storing return address during *word* invocation (similar to function calls). Although theoretically, one stack would be enough, having separated *data* and *return* stack simplifies the programming as the return addresses and computations are not mixed together.

Chapter 6

Test Automation

Software testing investigates whether a software meets requirements laid out by its specification. The challenge for the software tester is to fully comprehend the specification and assess if the implementation doesn't diverge from the specification. Specially, the tester needs to think about various use-case scenarios and corner-cases that could lead to a failure of the software implementation. The software tester is responsible for designing test scenarios and executing them. Out of these two activities, only the later one could be automated. The test automation releases the tester from performing tedious and routine work and allows him to focus more on the test scenario design, which is more creative.

Automation of the tests execution is especially important when testing becomes part of the development cycle, which is usual for some software engineering approaches, such as *continuous delivery* or *agile development*. When tests are expected to be executed repeatedly, test automation should be considered.

Citation: "Test Automation is an investment. The initial investment may be substantial, but the return on the investment is also substantial. After the number of automation test runs exceeds 15, the testing form that point forward is essentially free." [32]

Automated testing, however, introduces some new challenges. The responsibilities of a tester and a developer become linked to an extent where boundaries between these two activities disappear and the tester becomes a developer and/or maintainer of the automated test suite.

Citation: "What most of us probably do not understand is that a suite of automated tests is really a software system itself with the same problems faced by the system it is designed to test. It is prone to errors and extremely sensitive to changes. So, any time a client-server system is tested with automated test scripts, you are really dealing with two systems that have to be maintained. This doubles the maintenance problem." [36, pg. 215-216]

When a tester discovers a bug, he can design a new automated test that demonstrates the problem. The developer can then use such test for localizing and fixing the bug in the software. It is a welcomed bonus that the automated test will prevent the same issue to appear later due to regressions.

Citation: "The purpose of software testing is to identify new errors, while the purpose of debugging is to locate and remove known errors. Fat test cases (test cases that cover many test conditions) are used to identify new errors. Lean test cases (test cases covering only a single test condition) are used to locate and remove known errors. So, test cases that are not independent are better and more economical for finding errors because of fewer are required, but additional test cases that are independent are frequently required to find and remove the errors." [36, pg. 80]

Automated testing helps to assure and maintain code quality. It is especially useful for detecting regression issues that can occur due to code refactoring or branch merges.

There are two approaches to automated testing that are well suited for a software system suggested in this thesis; *unit testing* and *integration testing*. The purpose of unit tests is to verify function of individual code fragments and to help locating and fixing bugs. On the other side, *integration tests* can exercise the whole software system and discover any problems that can arise on the interface level or any interoperability issues between the parts of the system. End-to-end integration tests treat the system as a black box, and are only tied to the input and output interface of the system under test. As such, they are very universal and independent on the system architecture and its inner structure. Such integration tests are usually valid and useful even when the whole software system goes through significant architectonic changes. The end-to-end tests are of a big value, as they can assess whether significant changes in the system design didn't break a function that was working before. To give an example, sufficiently big end-to-end integration test suite could help to assess if adding a code optimization to a compiler was successful or not. More interestingly, the statistics performed on the test programs in the end-to-end test suite could yield some interesting data about improvements in efficiency of the compiled code.

6.1 Testing Frameworks

Testing frameworks suggest guidelines for writing test cases and offer tools for effective implementation of these guidelines. Usually, the testing framework defines how to organize test cases, introduces methods for expressing and verifying expectations. Next, the framework should have some mocking capability that allows to hook into the application under test. Finally, the framework should execute the test cases and report test results.

In this chapter, we will focus on frameworks that can be used for testing C code, as this is the implementation language for the Virtual Machine.

6.1.1 CppUTest

CppUTest [7] is a unit testing framework for C and C++ projects. The testing framework itself is written in C++, but uses only a subset of the C++ features, which makes it lightweight. For this reason, it is frequently used in embedded systems. It comes together with CppUMock, a framework for building test mocks. Concept of a mock functions and mock objects is important in unit testing in general. The code under test often has some dependencies to other objects or functions. For the purpose of unit testing, these dependencies should be replaced with a mock implementation that is bound to the testing framework. This way, the testing framework can verify if the code under test exercises its dependencies correctly, i.e. if it passes the right parameters to output functions, etc..

CppUTest has a mechanism for memory leak detection, supports test groups with custom setup and teardown methods. It can be also configured to generate test coverage report and has support for Eclipse IDE.

6.1.2 Check

Check [5] is a unit testing framework implemented in C. It is inspired by JUnit for Java and has similar features as the CppUTest framework described above. There is one big difference thought - the Check framework runs each test in a separate address space, so that the (potentially buggy) code under test cannot corrupt the testing framework itself. This property is especially important for testing C code, which is known to be prone to memory problems. Each test has a timeout that assures the test will not hang indefinitely. When the timeout of a test expires, it is reported as an error and Check continues with other tests. The Check testing framework relies on Autotools and it is more complex than CppUTest.

Part II

Design

After the analysis of various visual programming languages, existing robotic applications, features and computational capabilities of selected line-following robots, we have solid ground to continue with the design phase.

In this section, we will commit to important design decisions and reason about them. First, we will develop a specification for the Visual Programming language, which will in turn lay out requirements for the Virtual Machine.

Chapter 7

Visual Programming Language

The proposed Visual Programming Language will be shaped by the specific features of line-following robots on one hand, and by their resource limitations on the other. We will design a domain-specific language that will reflect the needs of mobile robot programming. However, many of the language features are general so that they can be applied to other applications as well.

To some extent, we can divide the specs of the language to two categories; *core language specification* and *domain-specific language extensions*. The core language specs are mainly derived from the computational capabilities of the hardware, while the *domain-specific language extensions* reflect the particular features and functions of the target robotic platform. In a way, the *core language specification* could be applied to any visual-programming-enabled robot or device of comparable or greater computational power.

In Section 4.2, we described the limitations of the target hardware. It has shown the 8-bit embedded system is short of memory and performance. Certainly there will be some trade offs due to these limitations, but our goal is to develop a language that supports the fundamental concepts of procedural programming. The purpose of the proposed Visual Programming Language is education in the first place. We believe that these concepts are especially important to learn programming:

- variables and expressions
- conditional statements
- cycles
- arrays
- procedures and functions
- recursion

This is an outline of the *core language specification*. To make the programming experience fun, of course the user program should have control over the robot's movement and sensor data. For this we need to develop an input/output mechanism between the user program and the real environment.

For example, basic recursive algorithms such as depth-first search (DFS) can be demonstrated when line-following robot searches a maze made of lines and intersections. That way, the line-following feature can help to demonstrate theoretical concepts. Interface for line following control is an example of *a domain-specific language extension.*

7.1 Front-end

The front-end part is a key piece of every visual programming language. The graphic user interface (GUI) is what lays out feature boundaries of the language itself. The front-end capabilities define the important aspects of the overall language design. From a certain point of view, it is the GUI what defines the visual programming language. That is so because the GUI can enforce the semantics of the program, given that the block components of the program are designed to interconnect only when it is semantically correct.

We chose Google Blockly library as the visual programming editor front-end. The Blockly library supports most of the desired language features listed in the beginning of this chapter. There are already implementations on the market that demonstrate Blockly library is highly customizable. Another good reason for favoring Blockly is good support of different platforms and touch screen support. The project is well maintained and has live contributors community around. These facts together make it a solid base for the proposed Visual Programming Language.

7.2 Data Types

Very important decision is to select the data types supported by the proposed Visual Programming Language. This decision will have a big impact on the Virtual Machine performance, user program code size and thus code transfer time.

The hardware doesn't have native support for floating point arithmetics. Emulating floating point arithmetics in software is inefficient and requires linking additional library routines that blow up the firmware footprint. Because of resource limitations, a decision was made that only integer numbers will be supported.

Programming languages often have `string` data type, but for simple line-following robots, there is no way how the robot could read or output a `string` variable. As there would be no use for it, the `string` data type doesn't need to be supported.

7.2.1 Universal Data Type

Programming languages usually support basic data types such as *integers*, *booleans*, *characters* or *enumerations*. These data types have usually different bit width. For example, integer might be 32 bit long while character data type is only 8 bit long.

Because the different bit width and different operations that can be performed on the basic data types, programming languages usually feature a type system. The type system prevents execution errors that could happen when an operation is applied to unsupported data types. Common languages have either static or dynamic type system, however, some languages don't have type system at all. We call them untyped languages:

Citation: "Languages that do not restrict the range of variables are called untyped languages: they do not have types or, equivalently, have a single universal type that contains all values. In these languages, operations may be applied to inappropriate arguments: the result may be a fixed arbitrary value, a fault, an exception, or an unspecified effect. The pure λ-calculus is an extreme case of an untyped language where no fault ever occurs: the only operation is function application and, since all values are functions, that operation never fails." [35]

As explained earlier, the Blockly library doesn't enforce static type checking on variables at design time, as it relies on the dynamic type checking of the target language (JavaScript, Python..) at runtime. The language for the target Virtual Machine, however, will be much simpler than any script language. In fact, the Virtual Machine will accept a bytecode similar to assembler, which is an untyped language by its nature. Enforcing type checking at the level of Virtual Machine would be costly and wouldn't make much sense, as there is no way how to signal where in the code the exception occurred at runtime.

Other option is to enforce static type checking at the level of Blockly editor, but this adds extra complexity for the user, requiring him to specify types explicitly for every variable. Therefore, we decided that there will be no type checking whatsoever, and the language will be effectively untyped. Instead of various data types, there will be a single universal data type of 8 bits in width. This greatly simplifies the design of the Virtual Machine and the Compiler. The value of a variable will be interpreted based on context where the variable is used. Because there are only few contexts in which the variable can be used, the risk of confusion or failure is limited. Principally, the beginner user will use the universal data type in Number Context only, which is very easy to grasp. In this use case, user will never come across other contexts. In the following text, we will explain the relations and representation of other data type contexts. These are used internally and are wrapped into nice graphic interface that limits any confusion for the user. Eventually, the power-user may use this advanced concept when he requires full control. Finally, we will use the minimal type checking feature of Blockly for constants, that can be verified statically at design time.

Number Context

If the universal data type is supplied to arithmetic operations, it will be treated as an 8-bit signed integer. The values range from -128 to 127, which is equivalent with the `int8_t` data type range in the C language. Based on our estimation and analysis of some use case scenarios, we believe that this range is sufficient for most practical calculations that user will perform and it is a well balanced trade-off between the usability on one hand and code footprint and performance on the other.

Because the range of integer data type is always limited, it is necessary to define behavior of arithmetic operations when the result would fall beyond the accepted range. Common practice is to use modular arithmetics and allow the calculations to overflow. We believe that for children, the concept of integer overflow/underflow is too difficult to grasp and would cause a lot of confusion. Therefore, we decide to use *saturation arithmetics*[1]*:*

Citation: "If the result of an operation is greater than the maximum, it is set ("clamped") to the maximum; if it is below the minimum, it is clamped to the minimum. The name comes from how the value becomes "saturated" once it reaches the extreme values; further additions to a maximum or subtractions from a minimum will not change the result." [46]

Saturation arithmetics will be used implicitly, but the power-user will still have an option to switch the virtual machine into *modular arithmetics* mode, which certainly has other advantages.

For the division and modulo operations, not all argument values are allowed. The result of such operations is not defined if the second argument is equal to `0`. In this case, the Virtual Machine should signal an exception and terminate the user program.

Boolean Context

Boolean data type will be handled in similar manner as in C.[2] The value 0 will be evaluated as `false`, while any non-zero value is evaluated as `true`.

Color Context

Line-following robots might have a color sensor installed. The color sensor outputs raw color data sampled on the three channels (red, green, blue). The user program could process the raw color sensor data and perform color classification on its own. Alternatively, the color classification can be done in the firmware. No matter which approach the implementation will take, it would be beneficial to define an enumerated data type context that describes the basic classified colors. For the enumerated colors, we will use the base *red*, *green*, *blue* colors and

[1] The saturation arithmetics is commonly used in Digital Signal Processing (DSP).

[2] C language doesn't have any built in boolean type either

their combinations; *cyan*, *magenta*, *yellow*, *black* and *white*. These colors are represented in the universal data type as numbers, see table 7.1

color	code (dec)	code (binary)	comment
blacK	0	000	no color component
Red	1	001	= 2^0 (basic color)
Green	2	010	= 2^1 (basic color)
Blue	4	100	= 2^2 (basic color)
Cyan	6	110	= Green + Blue = 2 + 4
Magenta	5	101	= Red + Blue = 1 + 4
Yellow	3	011	= Red + Green = 1 + 2
White	7	111	= Red + Green + Blue = 1 + 2 + 4

Table 7.1: Color codes

The table 7.1 suggests that some arithmetics operations still make sense when they are performed in the color data type context. See the figure 7.1 for additive color mixing (left) and subtractive color mixing (right). In particular, for additive color mixing the operator of choice is bitwise OR, while for subtractive color mixing the operator is bitwise AND.

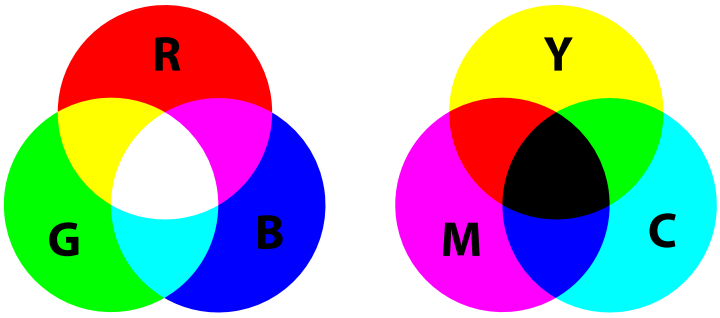

Figure 7.1: Additive (on left) and subtractive (on right) color mixing (By SharkD at English Wikipedia Later versions were uploaded by Jacobolus at en.wikipedia. - Transferred from en.wikipedia to Commons., Public Domain, https://commons.wikimedia.org/w/index.php?curid=2529435)

When the universal data type is interpreted in color context, only the three least significant bits (0 to 2) are taken into account. Bits 3 to 7 are ignored (masked out).

Intersection and Direction Context

When a robot performs line following, it can encounter intersections. Intersection is a point on line where there are two or more possibilities in which direction the robot could continue. Anytime the robot reaches an intersection, it needs to

decide what to do next - which way to continue. We also consider a degenerated case when robot has no way to continue (i.e. when it reaches line end) to be a special case of an intersection.[3]

Both the intersection types and decisions are encoded into numbers, see table 7.2. The logic is similar to the binary representation of colors in previous section. The four least significant bits (0 to 3) denote the direction or intersection type, bits 4 to 7 are ignored (masked out).

intersection/direction	code (decimal)	code (binary)
DIRECTION_ERROR	0	0000
DIRECTION_STRAIGHT	1	0001
DIRECTION_LEFT	2	0010
DIRECTION_RIGHT	4	0100
DIRECTION_BACK	8	1000
INTERSECTION_LINE_END	8	1000
INTERSECTION_T_END	14	1110
INTERSECTION_PLUS	15	1111
INTERSECTION_T_JOINT_LEFT	11	1011
INTERSECTION_T_JOINT_RIGHT	13	1101

Table 7.2: Intersection and direction codes

When robot arrives at intersection, the user program can retrieve the intersection type encoded to a 4-bit code, which specifies one of five possible intersection types:

Plus "+" intersection - crossing of two perpendicular lines, resembles the "plus" sign. Robot has four options to go; left, right, straight or back.

"T"-end intersection - a situation when robot is moving towards "T" on the bottom line, so that at the intersection there are three options to go: left, right or back.

"T"-joint left intersection - robot follows a straight line and it sees there is a perpendicular line coming from the left. It has three options to go: strainght, left or back.

"T"-joint right intersection - analogical to previous case

[3] in the case of special "line-end" intersection, the robot has only one direction to choose - go back

Line end - a degenerated case of an intersection. The line doesn't continue, so the robot has only one possible direction to go: backwards

These five intersection types are encoded in binary form. The user program can tell right away which directions are possible based on the binary code of the intersection. For example, if the robot encounters "T"-end intersection, the binary code of this intersection is 1110_b, which means that the robot can go either left (0010_b), right (0100_b) or back (1000_b).

Bitwise operations can be used to detect if the intersection type allows to continue in certain direction. For example, the expression `(intersection &` 0001_b`)` is `true` if the `intersection` variable is `INTERSECTION_PLUS(`1111_b`)`, `INTERSECTION_T_JOINT_LEFT (`1011_b`)` or `INTERSECTION_T_JOINT_RIGHT` (1101_b) and `false` otherwise.

The decision at the intersection is expressed in a similar manner. The user program will signal the direction to the firmware as a 4-bit decision mask. The firmware takes the decision mask and performs bitwise AND with the intersection type. The result states which direction the robot should continue. If the result is `DIRECTION_ERROR` (0000_b), it means the user program chose a direction that is not suitable for the current intersection. In that situation, the firmware signals a failure and the user program is terminated. If the result is exactly one direction (left, right, straight or back), the robot choses just that direction.

Interestingly, the result can allow more than one possible direction. For example, if the robot reaches "T-end" intersection (1110_b) and the decision mask is set to `(DIRECTION_STRAIGHT | DIRECTION_LEFT | DIRECTION_BACK) =` (1011_b), the result of the operation is (1110_b `&` 1011_b`) =` 1010_b, which means the robot can decide left or back (but not right). In this case, the robot could choose between the available directions randomly.

If the user program doesn't specify any decision mask, a default decision mask (0111_b) can be used. This means the robot, by default, can select left, right or straight randomly, but will never turn back. Specially, the robot will stop at line end if the default decision mask is applied (it forbids the robot to go back).

This section described the general concept of intersection handling at the level of Virtual Machine and the Bytecode. In the Visual Editor, this concept will be simplified graphically, in fact the user will never come across these low-level details unless he is willing to program his robot at the very low-level, as a power-user.

Other Enumeration Contexts

In previous two subsections, we explained how the universal data type can encode enumerated data types. The bit representation can be chosen such that some binary or arithmetic operations still make sense, even though they are applied to an enumerated type. This concept might be also applicable for any other platform

specific enumerations, such as states of other sensors, detection of checkpoints on the race track, etc...

7.2.2 Collections

In previous section we introduced the single universal data type and described how it will be interpreted in different contexts. The proposed Visual Programming Language should support collections of this single data type. Two basic collections will be supported: arrays and lists.

Array

The proposed Visual Programming Language will support static arrays. Size of the array has to be specified when it is declared (it has to be known at compile time). Due to performance and memory limitations, we do not allow dynamic array allocation. The user can declare any number of static arrays, the only limit is the size of the available virtual memory.

Arrays are indexed from zero. The runtime environment will take care of bounds checking; anytime the user will attempt to read or write array element that is out of bounds, an exception is signalized and the user program is terminated.

List

List is an abstract data type that is more intuitive than an array, because the user doesn't need to specify length when he wants to use it. The length of a list can grow dynamically at runtime. Initially, the list is empty and has zero length. User can intuitively add elements to the list and the list grows as required. This dynamic property adds some complexity to the implementation of the dynamic list data type.

Because of the hardware limitations, we decided that the proposed Visual Programming Language will support only one instance of a list data structure. There will be only one singleton list available to the user. This limitation is imposed due to the limited resources and absence of dynamic memory allocation.

The Singleton List will support these operations:

- append an element to the end of the list (list length is increased by 1)
- retrieve element at specified position
- delete element at specified position (list length is decreased by 1)
- write element at specified position
- get list length

- search in list (get index of a specific element)

Any access to beyond the end of the list triggers the "out of bounds" error and the user program is terminated.

7.3 Concurrency and Synchronization

Generally, single-threaded programs are easier to design, maintain and understand. Therefore, they are better suited for education and for first steps in robotics. The proposed Visual Programming Language will have only a single thread of execution. Still, the user thread can synchronize on external events that occur asynchronously. The user program itself, however, cannot create concurrent asynchronous tasks.

Line following is a built-in asynchronous service that runs concurrently to the user program. The user program can start or stop this service at any time. While the line following service is running, the robot travels on a line with a given speed, and signals various events to the user thread. The user program keeps executing in parallel and can synchronize on the events and process them. These events are, for example: intersection, line color change, track checkpoint detected etc...

There are other events that the user program can synchronize to, such as button press or internal real-time clock.

The proposed Visual Programming Language introduces a universal `wait` block. See Figure 7.2. The `wait` block is configurable and user can select which events the program flow should wait for. When any of the selected events occurs, the appropriate code is executed and the program flow continues to a block immediately following the `wait` block. When the wait block is nested inside a loop, the user can handle the incoming events repeatedly.

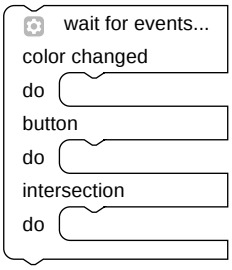

Figure 7.2: Universal Wait Block

Chapter 8

Testing

Testing is an important part of the development cycle. The testing process will be automated so that it can be executed efficiently and repeatedly. The automated test suite will be developed simultaneously with the implementation of the Virtual Machine and the Compiler. The automated tests will ensure that the Virtual Machine and the Compiler meet their specification and handle exceptional cases correctly, as defined in the specification.

Automated tests will also help to discover any regression issues that could occur when new features are added, or when some part of the source code is refactored. Specially, *integration tests* will be crucial for verifying merges with upstream Blockly repository.

Some automated tests will be derived directly from the specification. In the early phase, we will use the Test Driven Development (TDD) model. First, a test case is created from the specification and then the actual implementation is produced. This process is repeated iteratively until the specification is covered by test cases and fully implemented.

Some automated tests are created to verify specific use-case scenarios. These tests are complex, they can combine various language features in a single program and then verify that there are no unexpected side-effects when the program executes. Other automated tests are created ad-hoc, based on results of manual testing, when a bug is found, or after a code review.

When there is an erratic behavior observed in a particular user program, an automated test is created to demonstrate the issue. This test has two functions; first, the failing test can be executed in debugging environment to locate precisely the bug in the source code. Second, the test will ensure that the same bug will not reappear in future (regression).

8.1 Unit Testing

There will be a unit test suite for the Virtual Machine implementation. The VM has a clearly defined interface which allows to test it independently of the

firmware. We will use the CppUTest framework for automated testing the VM. The CppUTest framework is written in C++, but it is well suited also for testing C code. The C source files of the VM are compiled by a GCC C compiler and then linked to the test suite, which uses the CppUTest library. The test suite mocks the functions that the VM implementation uses for interfacing the firmware.

We will demonstrate the mocking mechanism on an example. When the user program invokes instruction to move forward, the VM calls a firmware function `system_move()`. The firmware then makes the robot move by a specified distance. In the test suite environment, the `system_move()` function is reimplemented to notify the CppUTest framework that the VM actually called a `system_move()` and passes the arguments.

```
void system_move(int8_t distance_mm,
                 int8_t speed_mmps) {
  mock_c()->actualCall("system_move")
    ->withIntParameters("distance_mm", distance_mm)
    ->withIntParameters("speed_mmps", speed_mmps);
}
```

The particular unit test can then verify if the `system_move()` function was called at the correct time with correct arguments. A minimalistic example for such unit test is shown in Program Listing 8.1.

```
TEST(VM_unitTests, MoveInstuction)
{
  instructionSet_t code[] = {
      INSTRUCTION_CONSTANT(10),
      INSTRUCTION_CONSTANT(25),
      INSTRUCTION_MOVE,
      INSTRUCTION_BREAKPOINT
  };
  VM_loadProgram(code, sizeof(code), 100);

  mock().expectOneCall("VM_event_programStarted");
  VM_startProgram();
  mock().checkExpectations();

  mock().expectOneCall("system_move")
      .withIntParameter("distance_mm", 10)
      .withIntParameter("speed_mmps", 25);
  VM_RunUntilBreakpoint();
  mock().checkExpectations();
}
```

Listing 8.1: Simplified main loop implementation

Please note that we used a special `BREAKPOINT` instruction at the end of the test program. The `VM_RunUntilBreakpoint()` function is an interface of the testing framework that exercises the VM to process instructions until it reaches the `INSTRUCTION_BREAKPOINT`.

8.1.1 Executing Tests on Target Architecture

We explained that the VM source code is compiled and linked to the testing suite. Here, we need to make an important disclaimer. The unit tests are executed on a regular PC, so they are built for the `x86` architecture. The VM module has to be also compiled for the `x86` architecture so that it can be linked with the test suite. In production, however, the VM and the firmware are compiled for the Atmel `AVR` architecture. Because of the differences between compilers and architectures, there is a particular set of issues that the automated tests cannot discover. It is generally advised to run the automated tests in an environment that is the same as in production. Unfortunately, this is not feasible for the particular 8-bit MCU, so we have to rely on executing tests in the x86 environment.

Still, the set of problems that can arise due to architectural differences is very specific, and the VM is implemented to overcome any possible compatibility issues. Static code analysis tools could be used to verify this rigorously.

8.2 Integration Tests

The purpose of the integration tests is to verify that a complex system is working correctly as a whole. In our particular case, the integration tests will exercise the whole chain. The chain consists of Blockly Editor, Code Generator, Assembly Compiler, Program Encoder, Program Decoder and finally, the Virtual Machine. The whole chain is depicted in Figure 8.1. One part of the chain is implemented in JavaScript (on the PC side) and the second part is implemented in C (runs in robot). The integration tests will feed a visual program (stored in Blockly native XML format) into the chain. The chain will perform all compilation steps, from code generation to bytecode loading to the Virtual Machine. Finally, the test suite starts the compiled program in the Virtual Machine and observes its behavior through mocked functions, as shown in previous section. The visual program might be enhanced with special `breakpoint` blocks, which allows stepping of the program and evaluate the intermediate computations, in similar manner as for traditional debugging techniques.

The integration testing is very powerful, as it exercises every single component in the system and tests the correctness of the interconnection between the components. The test cases are often created from real user programs, so they are very helpful in testing the VM, and are generally more convenient to create than unit tests because of their high-level nature.

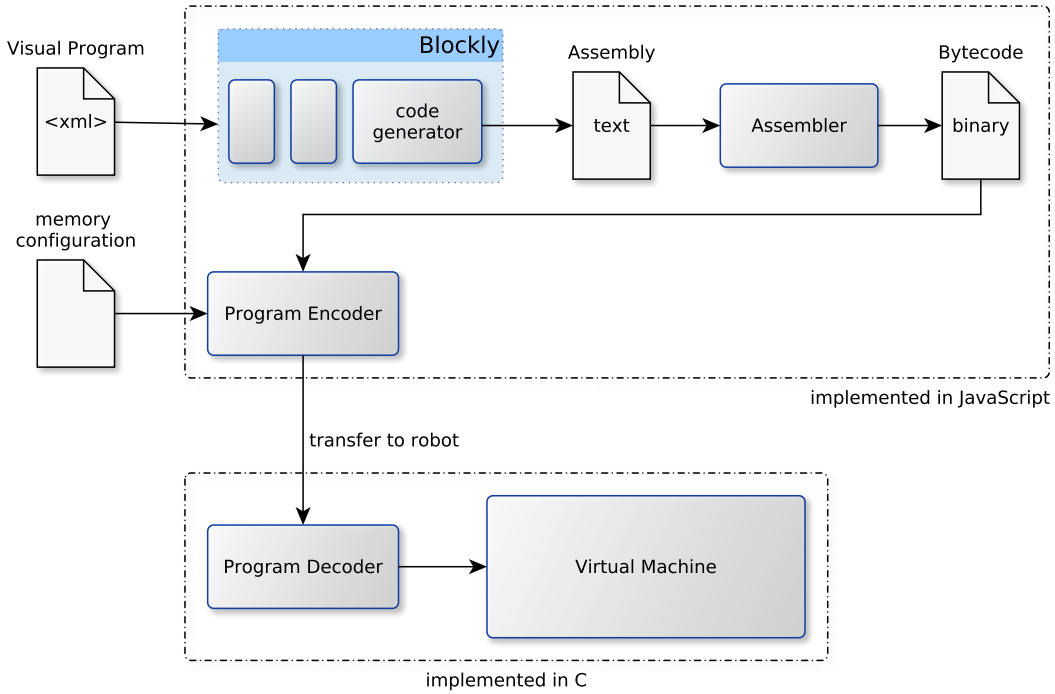

Figure 8.1: Integration test overview - the whole chain

We will show an example of a simple integration test. We have a user program implemented in Blockly editor, as shown in Figure 8.2. The program has special `breakpoint` blocks which allow easy stepping when the program is executed in the test suite.

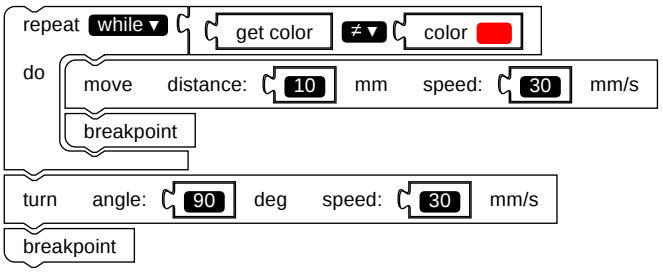

Figure 8.2: Simple Blockly program for Integration Test

The visual program from Figure 8.2 is saved in Blockly native XML format, and the XML source is then used by the integration test to exercise the complete chain. Please see Program Listing 8.2. The test first loads the XML to the test suite with the `Blockly_load_program_full_chain()` function. This function starts Blockly, compiles the XML program to bytecode and simulates loading the bytecode to the Virtual Machine. After all this is done, the compiled bytecode

program is ready to be executed in the Virtual Machine and the workflow is then the same as for unit tests, described in previous section.

```
TEST(IntegrationTests, MoveUntilRedIsFound)
{
  std::string xmlSource = "<xml>...</xml>";
  Blockly_load_program_full_chain(xmlSource);

  mock().expectOneCall("VM_event_programStarted");
  VM_startProgram();
  mock().checkExpectations();

  for(int i = 0; i < 10; i++) {
    mock().expectOneCall("system_move")
        .withIntParameter("distance_mm", 10)
        .withIntParameter("speed_mmps", 30);
    VM_RunUntilBreakpoint();
    mock().checkExpectations();
  }

  VM_signal_surfaceColor(COLOR_RED);

  mock().expectOneCall("system_rotate").
      withIntParameter("angle_deg", 90).
      withIntParameter("speed_mmps", 30);
  VM_RunUntilBreakpoint();
  mock().checkExpectations();
}
```

Listing 8.2: Integartion Test Example

This integration test simulates a situation that the robot makes 10 steps, of 1cm each, until it finally reaches the red color surface. Then, the while loop exits and the robot makes a 90 degree turn left. From this example, you can see that it is up to the integration test to simulate the inputs for the user program. In this case, the integration test simulated change of the surface color. This was done by the `VM_signal_surfaceColor()`, which is an interface function of the Virtual Machine. Normally, this function would be called by the firmware when it detects surface color change, but in this simulated case, it is triggered by the test suite.

There is about a hundred integration tests in the test suite, each of them verifies a specific use case. Often, the test cases are complex and they test advanced user programs that involve recursion, array manipulation, lists and other features of the language. The tests are derived from real use case scenarios and are a great source of data when it comes to bytecode instruction statistics, for instance.

Part III

Implementation

Chapter 9

Virtual Machine (VM)

The Virtual Machine is implemented in C as a module to the original robot's firmware. The interface between the VM and the firmware is clearly specified, which allows porting the VM to different platforms. The independence of the VM and the firmware is especially important for standalone automated testing of the Virtual Machine implementation.

With the limited resources of the target 8-bit MCU in mind, the features of the Virtual Machine have to be chosen deliberately. Many design decisions that will shape the VM were already made in the Chapter 7. The language features were chosen such that the implementation of the VM can be minimalistic. Only this way, the implementation can meet the resource limitations imposed in Section 4.2.

9.1 Scheduling and Preemption

It is expected that modules in the robot's firmware use co-operative static scheduling. The VM is implemented to obey this non-preemptive multitasking style imposed by the firmware design.

Citation: "Co-operative scheduling is where the currently running process voluntarily gives up executing to allow another process to run. The obvious disadvantage of this is that the process may decide to never give up execution, probably because of a bug causing some form of infinite loop, and consequently nothing else can ever run." [33]

The CPU resources are time-multiplexed between the firmware tasks and VM (user program). The VM can only run when the firmware already finished all its tasks and it is waiting for next iteration of the main loop. When the CPU would turn idle otherwise, the control is passed to the VM module. The VM has to co-operate and give back the CPU resources before the next main loop iteration comes. This way, all the tasks that are already scheduled in the original firmware are not touched and the VM uses CPU only when it would be idle anyway, so the hard real-time requirements that the original firmware has to meet are not

affected in any way.

The VM module is implemented to co-operate with the firmware when it comes to CPU resources. There is a non-preemptive scheduling contract between the VM module and the firmware. In contrast, the user program (bytecode) running within the VM module is scheduled preemptively. The VM is in full control of how many bytecode instructions will be executed within one main loop iteration. When the time is up, the VM module preempts the user program execution and gives control back to the firmware.

We conduced a test to measure the Worst-Case Execution Time (WCET) of any bytecode instruction. We use this WCET estimation to reason about the correctness of the user program preemption mechanism.

In Figure 9.1 we demonstrate the time multiplexing scheme between the VM module and the firmware. The time for which the VM executes changes dynamically depending on the firmware load. The VM module has to accommodate flexibly to fill the idle slot. There is a WCET estimation that the firmware will utilize the CPU at 76% at most in single main loop iteration. That means the slot length for the VM is always at least 24% of each main loop period.

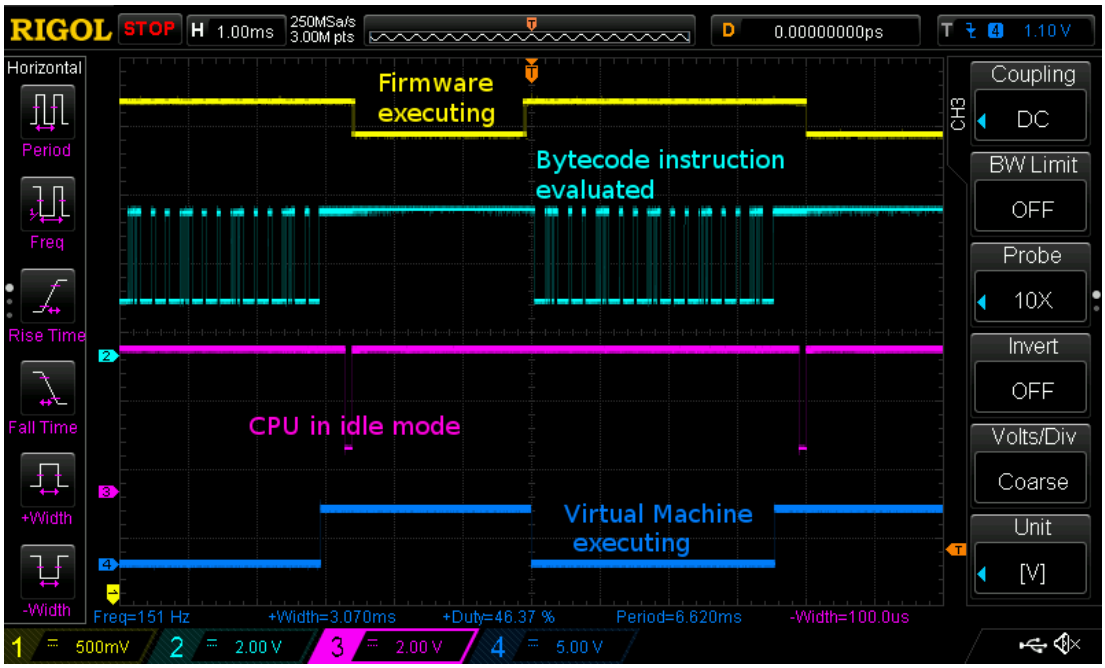

Figure 9.1: Scheduling of the Virtual Machine and firmware within one main loop iteration (digital oscilloscope screenshot)

The entry point of the VM module is the `VirtualMachine_task()` function. The function takes one argument - a pointer to a callback function. The callback function will be called before a bytecode instruction should be evaluated. The callback function estimates if there is enough time left to perform next bytecode instruction. If not, the VM must postpone evaluation of the instruction to next

main loop cycle and return control back to the firmware. See the Program Listing 9.1 for a simplified code snippet that demonstrates the static scheduling of the tasks.

```
do {
  /**
   * The main loop of the firmware carries out almost
   * all tasks. Only few tasks are implemented in
   * asynchronous manner, using interrupts.
   * Those are:
   * - UART data transmission
   * - PWM signal generation for motors and LEDs
   * - main loop timing
   */

  /**
   * The main loop need to be performed at regular
   * time intervals. At the start of the loop,
   * we clear the time counter:
   */
  MainLoopTiming_reset();

  ProcessSensors_task();
  LineFollowing_task();
  /*
  .. other statically scheduled tasks of the firmware
   */

  char preemption_check_callback(void) {
    char preempted = 0;
    /*
     * ...
     * checks if there is enough time to
     * evaluate next bytecode instruction
     */
    return preempted;
  }

  // evaluates bytecode instructions until the
     preemption_check_callback returns 1
  VirtualMachine_task(preemption_check_callback);

  /**
   * Blocks the execution until it is time for
```

```
    * the next loop iteration.
    * During waiting, the CPU is in idle mode.
    */
   MainLoopTiming_wait();

} while (!ShutDownRequested()); // if shut down was
    requested, we leave the loop and turn robot off
```

Listing 9.1: Simplified main loop implementation

9.2 Virtual Memory Map

User program (bytecode) runs in a sand-boxed environment governed by the Virtual Machine module. Previous section showed the CPU resources available to the user program are strictly controlled. Restricting memory access is of the same importance. The execution of (potentially malicious) user code must not under any circumstances compromise the consistency of the firmware. The user program can only access virtual memory that was assigned to it. Any attempt to reach outside the virtual memory area terminates the user program immediately.

Following sections describe the virtual memory layout. The boundaries of the memory segments are configurable and can be set to fit the needs of arbitrary user program[1]. Before user program is loaded, the memory arrangement can be reconfigured.

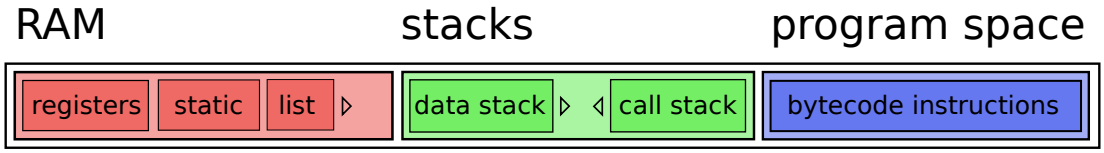

Figure 9.2: Virtual Memory Map (not to scale)

9.2.1 Program Space

The bytecode instructions (user program) are loaded in the Program Space segment. The size of the program space segment is usually configured so that the user program fills the segment completely and no memory is wasted.

[1] programs generally can have varying memory requirements. Some programs have bigger footprint but doesn't require much RAM space, some use recursion which is demanding on size of the stack, some might use dynamic list or big arrays stored in RAM..

9.2.2 Stacks

The virtual machine uses two stacks for the bytecode execution. The Data Stack is used for expression evaluation, passing function parameters and storing local variables. The Call Stack stores the subroutine return addresses and pointers to subroutine activation frames.

The two stacks share a common stack memory segment. The Data Stack grows from the lower addresses up, while the Call Stack grows from the upper addresses down. This way, the stack memory is used efficiently. When user program runs out of stack memory, stack overflow error is signalized and user program is immediately terminated. The size of the stack space is configurable.

9.2.3 RAM

Virtual RAM segment contains a register file, statically allocated data (variables and arrays) and a dynamic list data structure. Register file is always mapped to a fixed address both in virtual memory and in host RAM. Register file allows input/output operations between the sand-boxed user code, Virtual Machine and other modules in the firmware. The register file is basically a shared memory between the user program and the firmware. Function of every register is well defined and values are handled carefully when used by internal modules of the firmware.

9.3 Input/Output

The user program retrieves signals or data from the environment (Input) and controls the robot movement and its LEDs (Output).

Virtual Machine communicates with the host firmware through well defined interface. Two main approaches for input/output operations are supported: Shared Memory and System Calls.

9.3.1 Shared Memory (Register File)

The Register File contains 8-bit registers which can have input function, output function or both.

Input register is a register that is updated by the firmware and represents some state or configuration of the execution environment or outer world. The user program reads input registers and uses the data for calculations and program flow control.

Output register is written by the user program and processed by the VM (or eventually, by firmware). For instance, the user program can control line-following or LEDs intensities by writing the output registers.

Some registers have both input and output function. There is a register that stores the current line following speed, for instance. This register can be written by the user program (when the user program wants to override current line-following speed), but it can be as well written by the firmware (when the speed is updated externally, with a remote control, for instance). The user program can read the register to verify what is the actual line following speed.

Some registers are used to configure the properties of the VM, and some have a special function for indirect memory addressing, which is used for operations on arrays and lists.

Persistent Registers

Persistent registers (`REGISTER_PERSISTENT_n`) are used for storing data that should persist through program resets. User program can read and write these registers. The persistent registers could be also updated externally, by a remote control, for instance. The motivation for persistent registers arose from the need of parametrized user programs. User program can be designed to load its configuration from persistent registers. Anytime such user program needs to be reconfigured, only the persistent registers are updated. This is a fast process and it is more convenient than re-compilation and loading the whole user program again. Persistent registers retain their values throughout power-cycle and bytecode reprogramming.

Line Following Registers

REGISTER_LINE_FOLLOWING_SPEED (input/output register)
This register sets the current line-following speed of the robot. It can be written and read both by the firmware and the user program.

REGISTER_INTERSECTION_TYPE (input register)
Anytime the firmware detects an intersection, this register is updated with the detected intersection type (see 7.2.1 on page 37). Note that an event is generated anytime the register is written, no matter if the value was changed or not. This way, the `wait` block can still detect the intersection event, even if the robot repeatedly visits, let's say, "+" intersections.

REGISTER_INTERSECTION_DECISION (output register)
User program can select direction at next intersection by writing this register.

REGISTER_LAST_INTERSECTION_DECISION (input register)
Anytime the firmware makes a decision at intersection, this register is updated with a direction that was selected. The user program can record the decisions that the robot made at intersections, and use them later, for example in a maze search program.

Real Time Clock Register

Real Time Clock register (`REGISTER_REAL_TIME`) is increased every second by 1. By default, the current value of the Real Time Clock register shows how many seconds passed since the user program execution started. User program can write the register anytime to override its value. The program flow can be synchronized to changes of the register by the `WAIT` instruction, see Subsection 9.3.3 for more details.

Button register

When a button is pressed, the register `REGISTER_BUTTON_PRESS_COUNT` is incremented by 1. The user program can synchronize on changes of the register, as shown in 9.3.3 on the next page.

Other I/O registers

Other functions of the firmware can be mapped to the Register File. There can be registers dedicated for accessing the data measured by a color sensor, obstacle sensor etc.. External events (such as an command received from a remote control) can write a dedicated input register and the user program can synchronize on such events using the `WAIT` instruction, as shown in 9.3.3 on the following page.

9.3.2 System Calls

Other way how the user program can interact with the environment is to perform system calls. Frequently used I/O operations are more efficient to invoke through a dedicated bytecode instructions. Other reason to prefer system calls to input/output registers is when atomicity of operation is required. For instance, setting the color on an RGB LED requires 3 bytes to be written into a register file. If the operation is not made atomically[2], the first two bytes can be written in one main loop cycle, while the last byte would be written in the next main loop cycle, effectively creating a delay of several milliseconds between the register updates. This would cause unwanted artifact in the color reproduction. That is why for this case, it is preferred to use a dedicated system call that takes 3 bytes

[2]the bytecode has a way how to enforce atomicity of multiple registers read/write operations, however, this method introduces some performance overhead

as arguments and performs the write always at once. Setting motor speeds is also done through a system call (dedicated instruction) for exactly the same reason.

Because the instruction set is limited, not all functions can have a dedicated system call. That is why we decided to combine these two I/O approaches.

9.3.3 Synchronization on Events

The register file can be updated asynchronously by the firmware. An update of the register file can happen when the button is pressed, when surface color changes, or when robot arrives at intersection, for instance. All of these are events that are potentially interesting for the user program and should be processed as soon as they occur. It is inefficient for the user program to poll the register file for changes. Many MCU architectures have interrupts that trigger when an external event occurs, so that it can be immediately served by an appropriate interrupt routine. However, the interrupt subsystem is often complicated, the program needs to install interrupt routines and manage interrupt vector tables. We assume that the typical use case is to synchronize the program flow with an external event (i.e. block the program execution until the right event comes in). It is not expected that there is any performance-intense computation done while the user program is waiting for an external event. Implementing interrupts in the Virtual Machine would create unnecessary complexity for the user.

We introduce a simplified way of synchronization on external events that corresponds with the specification laid out in the Section 7.3. There is a special `WAIT` instruction in the instruction set. The `WAIT` instruction suspends the execution of the user program until any register in the register file is updated. When this event occurs, the VM pushes the address of the updated register on the data stack and the user program is resumed. Then, it is up to the user program how it handles the new information about the updated register.

We will demonstrate the concept on an example. Let's say the user program is waiting for a button press. Such code fragment would be implemented as a loop that waits for any register update, and when any update occurs, it verifies if the address of the register of interest (`REGISTER_BUTTON_PRESS_COUNT`) was returned by the `WAIT` instruction. If so, the program exits the loop and handles the situation. Otherwise, it starts new iteration of the loop, waiting for next event. In pseudocode, we would write this synchronization fragment as:

```
while(wait() != addressOf(REGISTER_BUTTON_PRESS_COUNT));
```

The `wait()` statement blocks the execution of the user program until any register is updated. When this happens, the `wait()` returns the address of the updated register.

This concept can be extended so that the user program can wait for any subset of events and handle each separately. In the rare case that two or more events

fire at the same time, the events are ordered by priority[3] and served one after another. Anytime the `WAIT` instruction serves an event, the dirty flag on that register is cleared. The user program may clear the dirty registers flags explicitly if it needs to.

9.3.4 Movement Commands

The firmware has an interface for two basic movement commands; *move straight* and *rotate*. These two commands are invoked via the `INSTRUCTION_MOVE` and `INSTRUCTION_ROTATE` instructions. These are blocking commands - they block the program flow until the movement is finished.

INSTRUCTION_MOVE takes two arguments, passed on the data stack. First argument is the speed in millimeters per second (mmps). Second argument is the distance to travel, in millimeters. If either the first or the second argument is negative, robot moves backwards. If both arguments are negative, robot moves forward.

INSTRUCTION_ROTATE takes two arguments, passed on the data stack. First argument is the speed in millimeters per second[4]. Second argument is the angle in degrees. If either the first or the second argument is negative, robot rotates clockwise. If either both arguments are positive or both arguments are negative, robot rotates counter-clockwise.

9.4 Instruction Set

The Virtual Machine interprets bytecode instructions. All instructions are exactly 8-bit long. Instructions don't need to carry any additional information about operands, as the operands are implicit. Instructions read their operands from the top of the Data Stack (DS). If there is any result to be stored, it pushed back to the Data Stack.

9.4.1 Push Immediate

Pushing a constant to the Data Stack is a very frequent operation. A significant part of a any program will be commands for loading the Data Stack with a constant data. Note that every instruction takes its arguments from the top of the Data Stack, therefore, loading a constant to the Data Stack is heavily used and worth optimization, especially in terms of code footprint.

[3] the priority of events corresponds to their register addresses, lower address has higher priority

[4] the speed in millimeters per second refers to the peripheral speed at circumference of robot's wheel. We use this unit for the reasons of consistency among the movement commands.

The first half of instruction space (instruction codes from 0 to 127) is interpreted as constants. Anytime the VM decodes an instruction with MSB[5] equal to 0, it interprets the instruction code as an unsigned integer and pushes that number to the top of the Data Stack immediately. This can be seen also from a different perspective: There is a special `PUSH_IMMEDIATE` instruction that carries some embedded data within. The instruction is of format `0xxxxxx`, where the 7 least significant bits carry the immediate value that will be pushed on the top of the Data Stack.

Apparently, the `PUSH_IMMEDIATE` instruction can only load 7-bit numbers, which happens to be only the positive range (0 to 127) out of the (-128 to 127) universal data type range. The `PUSH_IMMEDIATE` instruction can be combined with the `INSTRUCTION_NEG_ONES_COMPLEMENT`, which does a bitwise NOT operation to the top value on the Data Stack.

For example, pushing integer constant `4` is done by instruction code `00000100`, while pushing integer `-4` is done by instruction code `00000011` (pushes number `3` on the stack) followed by `INSTRUCTION_NEG_ONES_COMPLEMENT`, which performs a bitwise NOT on the top element on the stack. Finally, we have `11111100` on the stack, which is integer `-4` in two's complement form [47].

9.4.2 Operators

Instruction performing arithmetics or binary operations always pop their operands from the top of the Data Stack and push the result back to the Data Stack.

Arithmetics

All the arithmetics operations are done in saturation arithmetics [46]. The division and the modulo operation terminates user program if the operation is not defined on given arguments (the divisor is 0).

instruction	stack operation	comment
`INSTRUCTION_ADD`	a, b → (a + b)	sum of two integers
`INSTRUCTION_SUB`	a, b → (a - b)	subtracts the top integer
`INSTRUCTION_MUL`	a, b → (a * b)	multiplies two integers,
`INSTRUCTION_DIV`	a, b → (a / b)	divides with the top integer
`INSTRUCTION_MOD`	a, b → (a % b)	reminder of division
`INSTRUCTION_ABS`	a → abs(a)	calculates absolute value

Table 9.1: Arithmetic Operators

[5] most significant bit

Binary

Binary operators perform bitwise operations on their arguments. From other perspective, these can be also seen as *set operations*, where AND stands for *set intersection*, OR stands for *set union* and XOR stands for *symmetric difference*. This set interpretation mates well with the Color Context and Intersection and Direction Context in 7.2.1 and 7.2.1 respectively.

The `INSTRUCTION_NEG_TWOS_COMPLEMENT` calculates two's complement of the argument on the top of the stack. Saturation arithmetics is in effect, so argument -128 turns to 127 after the operation. Normally, it should be 128 but this doesn't fit the 8bit data range, therefore, truncation must occur.

instruction	stack operation	comment
`INSTRUCTION_AND`	a, b $\rightarrow$ (a & b)	bitwise AND
`INSTRUCTION_OR`	a, b $\rightarrow$ (a | b)	bitwise OR
`INSTRUCTION_XOR`	a, b $\rightarrow$ (a ^ b)	bitwise XOR
`INSTRUCTION_NEG_ONES_COMPLEMENT`	a $\rightarrow$ (~a)	bitwise NOT
`INSTRUCTION_NEG_TWOS_COMPLEMENT`	a $\rightarrow$ (-a)	two's complement

Table 9.2: Binary Operators

Logic

Logic operations pop operands from stack, evaluate them and then push the `true (1)` or `false (0)` constant on the Data Stack. Logic operators are listed in Table 9.3.

instruction	stack operation	comment
`INSTRUCTION_LOGIC_AND`	a, b $\rightarrow$ (a && b)	true if both arguments are true
`INSTRUCTION_LOGIC_OR`	a, b $\rightarrow$ (a || b)	true if at least one argument is true
`INSTRUCTION_EQ`	a, b $\rightarrow$ (a == b)	true if arguments are equal
`INSTRUCTION_COMPEQ`	a, b $\rightarrow$ (a $\geq$ b)	true if a greater or equal than b
`INSTRUCTION_COMP`	a, b $\rightarrow$ (a $>$ b)	true if a is greater than b

Table 9.3: Logic Operators

9.4.3 Jumps

Instructions that change program flow unconditionally are called *jumps*. The jump instructions need an argument that specifies what address the program execution should jump to. The address can be either *absolute* or *relative*.

Technically, the jump is performed by updating the current address in Program Counter (PC). In case of relative jumps, the PC is incremented by a constant, in case of absolute jumps, the value of PC is overwritten with a completely new (absolute) address.

INSTRUCTION_RJUMP is always accompanied with a relative 8-bit address. The address is not passed on stack, but it is stored in the program space immediately after the `INSTRUCTION_JUMP` opcode. The address is interpreted as signed integer in two's complement form. Therefore, the relative jump can be performed either forward or backward, depending on the sign of the argument.

INSTRUCTION_JUMP is always accompanied with an absolute 16-bit address[6]. The address is interpreted as unsigned integer and loaded to the Program Counter. This performs a jump in program flow and next instruction is fetched from the target address.

9.4.4 Conditional Branches

Conditional branches are similar to *jumps*, but the actual jump will occur only when a certain condition is met, otherwise, the program flow continues uninterrupted.

INSTRUCTION_BRANCH_SHORT the instruction is accompanied with a relative 8-bit address, in the same fashion as the `INSTRUCTION_RJUMP`. However, the jump is only performed when the top of the stack is `0` (`false`). Otherwise, the jump is not performed and program flow continues with the following instruction. The top element of the stack is always popped and discarded when this branch instructions are evaluated.

INSTRUCTION_BRANCH_ABSOLUTE analogical to the `INSTRUCTION_JUMP`, but the jump only happens when the value popped from the stack is `0` (`false`).

INSTRUCTION_INDIRECT_BRANCH same as the `INSTRUCTION_BRANCH_SHORT`, to an exception that it doesn't pop data from stack, but it branches if value of `REGISTER_Z` is equal to value of `REGISTER_F`. Otherwise, the branching doesn't occur. This instruction was introduced to increase the efficiency of array and list operations.

[6]The universal data type cannot represent pointer to every instruction in program space as it is an 8-bit integer. We also considered an option that instruction addressing in the program space would be aligned to 4 byte blocks. Then the jump addresses could be truly 8-bit, but all the jumps destination would need to be aligned to 4 byte boundary and the code footprint would grow in vain.

9.4.5 Memory Addressing

The RAM can be addressed *directly* or *indirectly*. For *direct addressing*, we have `INSTRUCTION_LOAD` and `INSTRUCTION_STORE`. These instructions take an address as an argument on stack. The address is always interpreted as 8-bit unsigned number. The width of the address data type implies that only the first 256 bytes of RAM can be addressed directly. The register file is mapped to the beginning of RAM, therefore, it can be manipulated using *direct addressing*.

INSTRUCTION_LOAD pops an 8-bit address argument from stack, then pushes on the stack the value stored at the specified RAM address.

INSTRUCTION_STORE pops an 8-bit address argument and a byte data from the stack (in this order), then writes the data byte to the specified RAM address.

An indirect addressing scheme is used to address RAM beyond the 256 byte boundary. Indirect addressing uses the `B` and `Z` registers. Specially, indirect addressing can be used also for addressing data within the 256 byte boundary, which might be convenient and efficient for array manipulation.

REGISTER_B Base address offset register, usually set to the address of the first element in array or list.

The value of the register is interpreted as 8-bit unsigned number multiplied by a factor of 2.

REGISTER_Z Relative address register. Its value is interpreted as 8-bit unsigned integer. Together with the Base Address Register, it is used for indirect addressing of the RAM. The RAM address is calculated by the formula: `address = (REGISTER_B * 2) + REGISTER_Z`
Note that the maximum addressable space is `255 * 2 + 255 = 765 bytes`. Because memory resources are limited on the target platform, this is not an issue.

INSTRUCTION_INDIRECT_LOAD Pushes on stack a value located at address calculated from the B and Z registers.

INSTRUCTION_INDIRECT_STORE Pops a value from stack and stores it to RAM address calculated from the B and Z registers.

9.4.6 Data Stack Manipulation

There are several instructions that manipulate the Data Stack. Operations on the Data Stack are necessary for passing and accessing arguments, locals and

return values of subroutines. Data stack manipulation can be also leveraged in code optimization.

INSTRUCTION_DELETE Removes the top element from the Data Stack.

INSTRUCTION_CLONE Duplicates the top element of the Data Stack.

INSTRUCTION_SWAP Interchanges the two elements on the top of the Data Stack.

INSTRUCTION_STACK_RESIZE Allocates or deallocates space on the Data Stack, depending on the argument. The argument is passed on stack as an 8-bit signed integer. If argument is negative, the stack height gets increased by that value. If argument is positive, the stack height gets decreased. Note that this instruction manipulates the Data Stack Top pointer, but doesn't overwrite any data on the Data Stack itself.

INSTRUCTION_PICK Picks a buried element on the Data Stack, addressed relatively from the top of the stack, and copies it to the top of the stack. The address is passed as an argument on stack.

9.4.7 Subroutines

Support for subroutines (functions and procedure calls) is integrated in the architecture of the Virtual Machine on many levels; there is a set of dedicated instructions, a dedicated *Call Stack* (CS) that enables subroutine recursion and a system for handling *activation frames* of subroutines.

INSTRUCTION_CALL Performs a subroutine call. The program flow jumps to the address of the subroutine, specified as a 16-bit unsigned number stored in the 2 bytes that immediately follow the INSTRUCTION_CALL opcode. First, the return address (calculated as PC + 3) and the current Frame Pointer (FP) is pushed to the Call Stack. Then, the Program Counter (PC) is updated with the subroutine address and the Frame Pointer (FP) is updated with the current Data Stack top pointer.

INSTRUCTION_RET Subroutine finishes with the return instruction (`RET`). This instruction pops the return address and the previous Frame Pointer from the Call Stack (CS) and updates the Program Counter and Frame Pointer register.

INSTRUCTION_POINTER_CALL Function pointers are supported by the VM. The Pointer Call instruction is similar to the `INSTRUCTION_CALL`, to an exception that it pops the subroutine address from Data Stack as an 8-bit unsigned integer. This address is multiplied[7] by factor of 2 before it is copied to the Program Counter.

INSTRUCTION_FRAME_LOAD Loads an element buried in the Data Stack, addressed relatively to the current Frame Pointer. The instruction pops its relative address parameter from the Data Stack.

This instruction is necessary for accessing arguments and locals of subroutines that are stored in the Data Stack. It is different from the `INSTRUCTION_PICK`, because the element address is relative to the Frame Pointer, which is always constant inside the scope of the subroutine.

INSTRUCTION_FRAME_STORE Stores an element to a location in the Data Stack, addressed relatively to the current Frame Pointer. It is analogous to `the INSTRUCTION_FRAME_LOAD`. This instruction is used for updating subroutine locals and for storing the return value.
The use of these instructions will be shown in detail the following chapter, 10.4.7 on page 74.

[7] This implies that when pointers to functions are used, compiler must ensure that functions entry points are 2-byte aligned. Function pointer support is an advanced feature that currently doesn't have any use in the proposed Visual Programming language, but might come in handy in hand-optimized bytecode.

Chapter 10

Code Generation

The Visual Program will be eventually complied to a bytecode that is interpreted by the Virtual Machine. The process of visual program compilation is carried in two steps, see Figure 10.1.

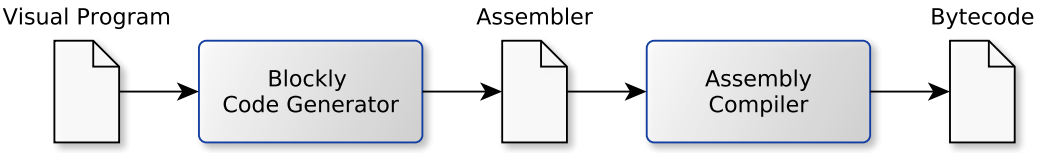

Figure 10.1: Compilation process

The Visual Program in Blockly Editor is internally represented as a tree data structure, very similar to Abstract Syntax Tree (AST) known from compiler theory [26, pg 69-70]. The Blockly library has modules called *code generators* that traverse the program tree and generate JavaScript, Python, Dart or PHP.

We extend the Blockly library with a new code generator that generates the intermediate assembly code, which will be later processed to a bytecode.

10.1 Intermediate Assembly Language

The proposed Intermediate Assembly Language is a symbolic machine code. Structurally, it is very close to the final bytecode interpreted by the VM. Most of the assembler instructions have one-to-one correspondence to the VM bytecode instructions. However, these instructions take symbolic labels as arguments, in contrast to bytecode where all arguments are numeric.

The proposed Intermediate Assembly Language is a human readable code that can be manually created and edited. But, this was not the reason why this assembly language was initially introduced. The reason is more pragmatic; Blockly code generators are better suited for generation of high-level languages

than a machine code / bytecode. It is easier to generate a language that uses symbolic references and later translate this language to a bytecode in a separate pass.

Citation: "The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an assembler *that produces relocatable machine code as its output."* [26]

10.2 Bytecode

The proposed Bytecode is directly interpreted by the Virtual Machine. It is a sequence of bytes that constitutes the compiled user program. It is straightforward to translate the bytes back to instruction names, so sometimes the bytecode is represented as a sequence of instructions rather than sequence of bytes. All arguments and constants are, however, numeric, therefore, it is very impractical to modify the bytecode once it was generated.

10.3 Library Functions

The proposed Visual Programming Language has a library of predefined functions that the user program can invoke. At code generation phase, the dependencies to library functions are resolved and the library function bodies are linked to the user program assembly.

Internally, the library functions are stored in the form of *position independent code* [45], which in fact is a compiled bytecode that meet these requirements:

- uses only relative jumps or branches
- constitutes a *callable unit*, compatible with the calling conventions of the proposed Visual Programming Language

Some library functions are programmed in bytecode directly and hand-optimized, some were designed in the Visual Programming Language itself and compiled to a position independent bytecode.

10.4 Generating Code

In this section, we will describe how the high-level language constructs are compiled to the Assembly Language.

10.4.1 Expressions

In the Blockly visual programming language, expressions are formed easily by nesting puzzle-like blocks. This graphical notation is interesting because it implicitly captures the semantics and order of evaluation. In the Figure 10.2, there is the expression $2 \cdot (3 + 4)$ depicted in visual programming language. In textual source code, the operators are infix and therefore, we had to add a parenthesis to keep the semantics of the expression. In Blockly, however, the nesting of the blocks already carries the information about order of evaluation of the arithmetics operations.

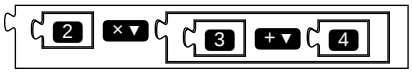

Figure 10.2: Expression in Blockly

Internally, such Blockly code fragment is represented as a tree with the numeral constants blocks as leafs and operator blocks as internal nodes. See Figure 10.3.

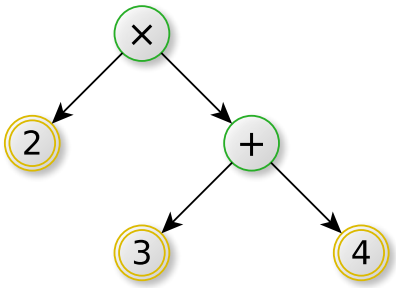

Figure 10.3: Abstract representation of an expression in Blockly

Evaluation of expressions in the Virtual Machine is trivial. The VM uses the Data Stack for expression evaluation. As it is, the VM can right away evaluate expressions in postfix notation (also known as Reverse Polish notation).

The Assembly code generator performs a post-order traversal on the expression tree. For this particular example, the post-order traversal would produce a sequence $2, 3, 4, +, *$. When we map the constants and operators to the Assembler instructions, we receive a code that can be executed by the VM right away:

```
PUSH_CONST(2)
PUSH_CONST(3)
PUSH_CONST(4)
INSTRUCTION_ADD
INSTRUCTION_MUL
```

The Virtual Machine processes the instruction one after another. First, it fills the Data Stack with integers: $\begin{matrix}4\\3\\2\end{matrix}$. Then it performs the `ADD` operation on the two topmost elements and stores the result to the stack: $\begin{matrix}7\\2\end{matrix}$. Finally, it performs the `MUL` operation, and the stack contains single element, 14, which is result of the calculation.

This simple concept is applied also for expressions that contain variable references or function calls. The postfix evaluation is the key concept here.

10.4.2 Variables

In Blockly, all variables are global. Blockly editor doesn't have any means how to limit scope of variables. As a consequence, there is no point in declaring a variable on some explicit place in the program. This allows to omit variable declarations completely.

Because of this fact, and because there is only one universal data type (as explained in Subsection 7.2.1), the Assembly code generator is very minimalistic.

Figure 10.4: Set Variable and Get Variable blocks

In the proposed Assembly Language, the programmer can use labeled memory cells. Textual labels are automatically mapped to free (unused) cells in the RAM memory. Anytime the assembly programmer wants to introduce a new labeled memory cell, he just uses its name with a special prefix. For instance, `addr$var` is a valid Assembly statement that means "address of RAM memory cell labeled `var`". The Assembly compiler maintains a table of allocated labels. When it detects the `addr$var` for the first time, the `var` symbol is not yet in the table, therefore, it is added and an address of next free memory cell is assigned to it. Further references to `addr$var` will be always translated to that specific address in RAM.

The `Get Variable` block is then translated to two Assembly instructions:

```
addr$var
INSTRUCTION_LOAD
```

This is enough for the Assembly Language. After this code is compiled to bytecode, the `addr$var` placeholder instruction will be replaced by a real address of a memory cell in RAM. Then, this bytecode fragment will load data from a particular memory address to the top of the Data Stack.

The `Set Variable` block is translated in similar manner:

```
PUSH_CONST(6)
addr$var
INSTRUCTION_STORE
```

The `STORE` instruction takes the address of variable `var` and stores there whatever was calculated on the Data Stack. In this case, it is the constant `6`, but in general, it could be result of an expression or a value returned from a function call.

Here, we need to make an important disclaimer. The lack of local variables would be in some situations very limiting, for example when recursive algorithms are considered. The proposed Visual Programming Language doesn't have true local variables, as Blockly editor doesn't support such constructs. However, the function parameters are handled in a way that allow to overcome these limitations and emulate local variables in functions. This will be explained further in Section 10.4.7. However, the proposed Virtual Machine and bytecode have full support for local variables, that are allocated on the Data Stack.

10.4.3 Conditional Statements

Translation of conditional `if` and `if/else` blocks uses the *branch* instructions. The proposed Assembly Language provides an abstraction for the branch instructions. There is a `branch(label)` meta instruction that performs a conditional jump to a specified label if the value popped from the Data Stack is equal to `0` (`false`). The Assembly compiler later resolves the symbolic label to a real address in the Program Space and replaces the `branch` meta instruction with either relative branch (`INSTRUCTION_BRANCH_SHORT`) or absolute branch (`INSTRUCTION_BRANCH_ABSOLUTE`).

We demonstrate translation of the if block on a simple code fragment, see Figure 10.5.

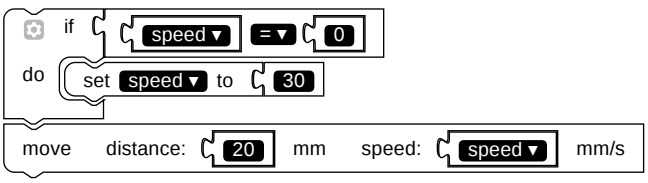

Figure 10.5: If block - code fragment

Such code fragment is translated to Assembler as shown in Table 10.1.

Conditional statements `if/else-if/else` are translated in similar manner, but these use more labels and every code branch is trailed with an unconditional jump instruction to the end label.

Assembler:		Bytecode:
addr$speed	0	PUSH_CONST(37)
INSTRUCTION_LOAD	1	INSTRUCTION_LOAD
PUSH_CONST(0)	2	PUSH_CONST(0)
INSTRUCTION_EQ	3	INSTRUCTION_EQ
branch(label1)	4	INSTRUCTION_BRANCH_SHORT
	5	DATA(5)
PUSH_CONST(30)	6	PUSH_CONST(30)
addr$speed	7	PUSH_CONST(37)
INSTRUCTION_STORE	8	INSTRUCTION_STORE
label1:		
PUSH_CONST(20)	9	PUSH_CONST(20)
addr$speed	10	PUSH_CONST(37)
INSTRUCTION_LOAD	11	INSTRUCTION_LOAD
INSTRUCTION_MOVE	12	INSTRUCTION_MOVE

Table 10.1: If block fragment translated to Assembler and Bytecode

10.4.4 Loops

Blockly editor supports *while* loops and *for* loops. We will show on examples how these blocks are translated to Assembler and Bytecode.

While Loop Block

Figure 10.6 and Table 10.2 shows how the `while` block is translated.

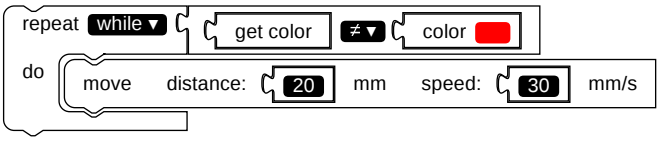

Figure 10.6: While loop - code fragment

Repeat n-times Block

The `repeat n-times` block example is a special case of the *for* loop. Because this case is simple, we will start with this first. Please refer to Figure 10.7 and Table 10.3. The translated Bytecode initializes the cycle counter variable on the stack and then decreases it in every iteration, until it is not equal to 0. Finally, the loop exits and program flow continues beyond the *repeat* block. It is very important to keep the stack consistent, so we have to delete the cycle counter variable from the stack when the loop finishes.

Assembler:		Bytecode:
label1:		
REGISTER_SURFACE_COLOR	0	PUSH_CONST(14)
INSTRUCTION_LOAD	1	INSTRUCTION_LOAD
COLOR(red)	2	PUSH_CONST(1)
INSTRUCTION_EQ	3	INSTRUCTION_EQ
INSTRUCTION_NOT	4	INSTRUCTION_NOT
branch(label2)	5	INSTRUCTION_BRANCH_SHORT
	6	DATA(7)
PUSH_CONST(20)	7	PUSH_CONST(20)
PUSH_CONST(30)	8	PUSH_CONST(30)
INSTRUCTION_MOVE	9	INSTRUCTION_MOVE
jump(label1)	10	INSTRUCTION_JUMP_RELATIVE
	11	INSTRUCTION_DATA(-10)
label2:	12	...

Table 10.2: While loop block fragment translated to Assembler and Bytecode

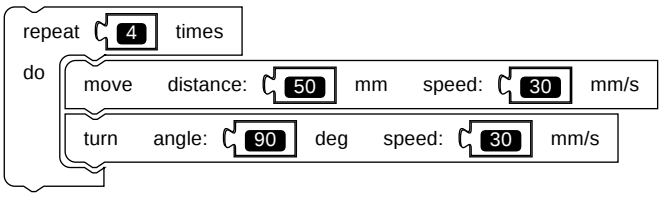

Figure 10.7: Repeat n-times - code fragment

For Loop Block

In Blockly, the for loop is implemented using the `count with` block, which in essence corresponds to usual for loop known from other programming languages. However, there are some subtle differences that on one hand simplify the use, but make the compilation more tricky.

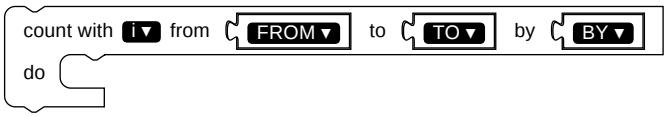

Figure 10.8: The Blockly `count with` block has a function of typical *for loop*

From the Blockly documentation: "The count with block (called a for loop in most programming languages) advances a variable from the first value to the second value by the increment amount (third value), running the body once for each value." [37]

The `count with` block accepts three numeric inputs; `FROM`, `TO` and `BY`. If the `FROM` number is lower than the `TO` number, the control variable is incremented by

Assembler:		Bytecode:
PUSH_CONST(4)	0	PUSH_CONST(4)
label1:		
INSTRUCTION_CLONE	1	INSTRUCTION_CLONE
PUSH_CONST(0)	2	PUSH_CONST(0)
INSTRUCTION_COMP	3	INSTRUCTION_COMP
branch(label2)	4	INSTRUCTION_BRANCH_SHORT
	5	DATA(11)
PUSH_CONST(50)	6	PUSH_CONST(50)
PUSH_CONST(30)	7	PUSH_CONST(30)
INSTRUCTION_MOVE	8	INSTRUCTION_MOVE
PUSH_CONST(90)	9	PUSH_CONST(90)
PUSH_CONST(30)	9	PUSH_CONST(30)
INSTRUCTION_ROTATE	10	INSTRUCTION_ROTATE
PUSH_CONST(1)	11	PUSH_CONST(1)
INSTRUCTION_SUB	12	INSTRUCTION_SUB
jump(label1)	13	INSTRUCTION_JUMP_RELATIVE
	14	DATA(-12)
label2:		
INSTRUCTION_DELETE	15	INSTRUCTION_DELETE

Table 10.3: Repeat n-times block fragment translated to Assembler and Bytecode

the absolute value of the `BY` number. The loop can also count down, if the `FROM` is bigger than `TO`, then the control variable is decremented by the absolute value of the `BY` number.

The JavaScript generator for the "count with" block produces quite a lot of code in the most general case to handle corner cases:

```
var i;
var FROM;
var TO;
var BY;

var i_inc = Math.abs(BY);
if (FROM > TO) {
  i_inc = -i_inc;
}
for (i = FROM;
     i_inc >= 0 ? i <= TO : i >= TO;
     i += i_inc) {
}
```

In the most general case, the generated code has to decide at runtime which direction to count and what is the condition to exit the loop.

The `count with` block should behave the same when compiled to bytecode and executed in the target robot. The JavaScript code generator could be used as reference when implementing the Assembly generator. However, it turns out that such approach would not be optimal in code footprint. Moreover, the JavaScript generator for the `count with` block doesn't take into account that the variable actually might have limited range and saturation of the control variable might occur. This would cause infinite loops for certain settings of input parameters, for example when `FROM` and `TO` are set to boundaries of the integer data type, `-128` and `127` in case of the proposed Virtual Machine.

To prevent faulty infinite loops, we need to update the inner implementation of the `count with` block, but not change the specification that users are already used to.

One observation is that the loop doesn't care about the sign of the `BY` argument. As a side effect, the code in the for loop is always performed at least once, no matter how the `FROM`, `TO` and `BY` values are set. We use this observation to construct Assembly generator that generates more efficient bytecode.

The tricky part is the restricted range for the integer data type in the VM. This has shown to be quite complicated problem for the correct `for` loop implementation. To resolve this, we need to test variable truncation at runtime. Anytime the control variable would be truncated when the increment is added, we exit the loop.

The Assembly generator for the `count with` block generates code that fulfills the same specification as the original JavaScript generator, but it is tailored for the VM architecture. There are many automated tests that prove the implementation is correct.

Break and Continue

Blockly allows *break* and *continue* control flow statements within loops. This language feature is supported in the Assembly generator as well. Internally, the control flow statements are implemented as *jumps* to specific labels inside the loop control code. For the correctness, it was crucial to prove that such jumps always keep the state of the data stack consistent. This was later verified by number of automated tests.

10.4.5 Arrays

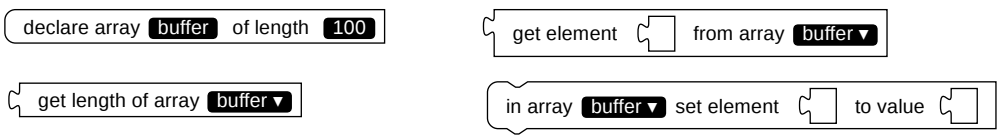

Figure 10.9: Array support in Blockly

Array is a new feature added to Blockly. Arrays in Blockly are declared statically and their size and memory location is known at compile time. Arrays are indexed from zero. Bounds checking is always performed prior read or write operation. If the index is out of bounds, the user program is terminated and robot signalizes the "index out of bounds" error. User can declare any number of arrays, only the RAM size of the VM is the limit.

Declaration

Array is declared with a the `declare array` block. User specifies a name of the array and its length. The declaration block doesn't have any connections and can be located anywhere in the workspace. Once the array is declared in the workspace, it can be accessed from anywhere in the user program - it has a global scope.

Array occupies a fixed space in the *static array pool*, which is a continuous segment in RAM. Within this memory segment, start of each array is aligned to the 2-byte boundary, which is a requirement for the indirect addressing via the `REGISTER_B`.

Get Array Size

Length of array can be requested using the "get size of array" block. The block has a drop-down menu that contains list of all arrays that were declared in the workspace. Since the size of all arrays is known at compile time, the generator for this block always generates constants and its implementation is trivial.

Get Element

The `get item` block retrieves an element from a particular array, which is selected from the drop-down menu. The index of the desired element is supplied as input.

Set Element

The `set item` block implementation is similar to the previous one. We present the Assembly code generated for the `set element`, as the `get element` is very similar. Please refer to Figure 10.10 and Table 10.4. The generated bytecode does index bound checking. The bound checking is performed by four additional instructions. The `INSTRUCTION_CONDITIONAL_FAIL` pops a number from stack and if it is other than 0, it terminates the user program and signalizes an error.

10.4.6 List

List is a data structure supported by the Blockly library. The default JavaScript Blockly code generator implements the list functionality through dynamic array. The proposed Visual Programming Language doesn't support dynamic allocation

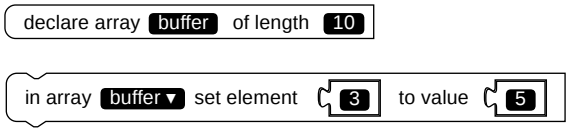

Figure 10.10: Writing array element in Blockly

Assembler:	comment
`arrayPool#0`	is resolved to a 2-byte aligned address of the array in RAM
`REGISTER_B`	
`INSTRUCTION_STORE`	loads the address of the array's first element to REGISTER_B
`PUSH_CONST(5)`	the new value
`PUSH_CONST(3)`	stored to array element 3
`INSTRUCTION_CLONE`	bounds checking
`PUSH_CONST(10)`	
`INSTRUCTION_UINT8_COMPEQ`	compares unsigned array indices
`INSTRUCTION_CONDITIONAL_FAIL`	fails if result of the comparison is true
`INSTRUCTION_RELATIVE_STORE`	write to an array

Table 10.4: `set element` block fragment translated to Assembly

due to resource limitations. However, we believe that the list data structure is important for educational purposes, therefore, we added at least a limited support for the list data structure. There is only one singleton List available to the user. The List is initially empty and grows when user appends elements to the List. The List is allocated in the RAM at compile time, it starts at the first free addressable memory location and grows up. Unlike arrays, the length of the List can change at runtime. The List can accommodate at most 127 elements, given that there is enough free memory available in RAM.

The code generation procedure is analogous to the Array implementation (including the dynamic bounds checking), except that the length of the list is not constant, so it is maintained at runtime in a dedicated variable.

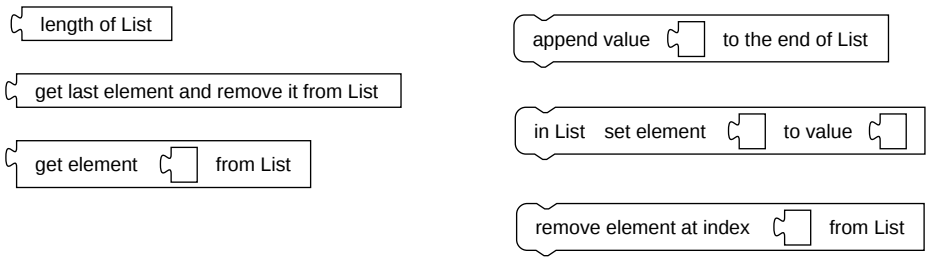

Figure 10.11: Singleton List support

10.4.7 Functions and Procedures

In Section 10.4.2 we claimed that Blockly has only global variables. There is one exception though; variables in functions are handled specially. First, arguments of functions and procedures are passed by value. Second, variable shadowing [48] is applied to arguments of functions and procedures. In practice, this means when function has a named argument `x`, and there is a global variable `x`, these two variables have different storage, thus they do not refer to the same object. In the scope of the function, all references to the `x` symbol actually refer to an argument of the function, not to the global variable. We say that the global variable `x` is shadowed by the inner variable. Because all arguments are passed by value, the storage space for arguments is always allocated on stack. Every invocation of a function has its own stack frame where its arguments are allocated.

It is very important to understand how function and procedures work in Blockly, so that the compiled bytecode executes in the same fashion as Blockly visual program would. The semantics of function calls cannot be learned from the original Blockly visual programming language itself. The original Blockly visual language is never interpreted directly, it is always the generated code (JavaScript, Python..) that can be executed, not the program in its visual form. For this reason, we will investigate how the Blockly visual program translates to JavaScript, and how the generated JavaScript code would execute. In Figure 10.12 there is a visual program in Blockly that demonstrates a simple function call to a function that returns double of its argument. The question we shall ask is, what values are applied to the `set wheel speed` block?

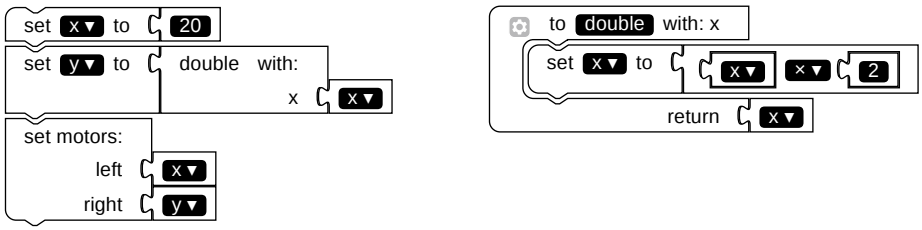

Figure 10.12: Example of a simple function call

To answer this question, we need to evaluate the generated JavaScript code:

```
var x;
var y;

function double(x) {
  x = x * 2;
  return x;
}

x = 20;
```

```
y = double(x);
system_setWheelSpeeds(x, y);
```

The argument `x` of function `double` shadows the global `x` variable. When the function executes, it can never modify the global `x`, it writes to a local `x` whose value is then returned and written to global `y`. Eventually, the `system_setWheelSpeeds(20, 40)` is called.

We see that the proposed Assembly code generator should:

- pass function arguments by value
- ensure variable shadowing

The arguments are stored in a subroutine activation frame on Data Stack. For functions, the activation frame also stores the return value. The caller is responsible for allocation and deallocation of the activation frame on the Data Stack. On the Call Stack, the activation frame containing return address and previous frame pointer are created automatically when the `INSTRUCTION_CALL` is executed.

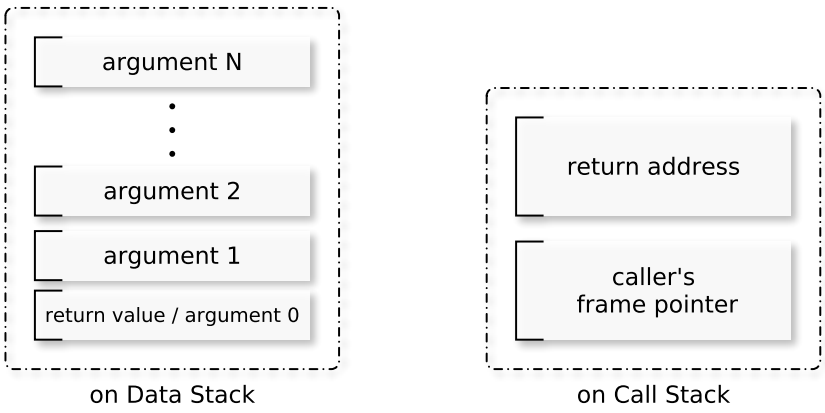

Figure 10.13: Activation frame is created both on Data Stack and Call Stack

The callee can then address the arguments via the `FRAME_STORE` and `FRAME_LOAD` instructions. These instructions take an address relative to the activation frame start, which is retrieved from the current Frame Pointer. For functions, the return value is always stored to the address 0 in the activation frame. Note that this location is also used for passing argument 0. The compiler must ensure the argument 0 is never used again by the callee after the return value has been written.

In Table 10.5, we present how the example from Figure 10.12 is complied to Assembly.

Although the original Blockly visual programming editor doesn't support local variables in functions and procedures, the locals could be easily supported by the

Assembler:	comment
`.....`	initialization of the x variable
`addr$x`	
`INSTRUCTION_LOAD`	caller loads value of x to the activation frame
`call(double)`	calls the function
`addr$y`	when function executed, the return value is on stack
`INSTRUCTION_STORE`	and the caller stores the return value to variable y
`.....`	setting the motors
`double:`	label of the function
`PUSH_CONST(0)`	
`INSTRUCTION_FRAME_LOAD`	callee loads the argument 0
`PUSH_CONST(2)`	
`INSTRUCTION_MUL`	multiplies the value by two
`PUSH_CONST(0)`	
`INSTRUCTION_FRAME_STORE`	and stores the result back to the activation frame
`INSTRUCTION_RET`	returns control back, pops record from the Call Stack

Table 10.5: Program with a function call translated to Assembly code

proposed VM and compiler. There is, however, a workaround how to use locals even in Blockly. The idea is to declare locals as extra arguments of the function or procedure. This effectively allocates extra variables in the activation frame, which later the callee can use as local variables allocated on stack.

Chapter 11

Integration Test Suite

In Chapter 8, we described the integration test suite design and typical test cases. Here, we will point out some interesting implementation details about the integration test suite. As was shown in Figure 8.1, some parts of the chain are implemented in JavaScript, while others are implemented in C. This makes the implementation of a test suite challenging. Fortunately, the interface between the JavaScript part and the C part is simple. The Blockly editor compiles the visual program to Assembler, which is in turn translated to Bytecode and encoded into a format that can be transmitted to the robot. All this is done in JavaScript.

On the side of the robot, the received data is fed to the Program Decoder module, written in C. The Program Decoder module verifies data checksum to combat any transmission errors and then loads the program into the VM. At this point, the VM is ready to execute the Bytecode, and further program testing is conduced same way as for the unit tests (see Section 8.1).

The test suite implements function `Blockly_load_program_full_chain()`, which takes an XML source as an argument and feeds that into the whole chain. Internally, this function communicates with a separate process that caries out interpretation of JavaScript part of the chain. We use the *rhino* JavaScript engine to interpret all the JavaScript code. The *rhino* process loads the JavaScript part of the chain and uses it to translate the XML visual program source into a packed bytecode format, that can be received by the robot. The data is then output to the C++ part of the test suite and used to exercise the Program Decoder, which loads the VM with compiled bytecode.

Because the loading of Blockly into *rhino* JavaScript engine is costly, it is done only once when the *rhino* process is started, and later calls to the `Blockly_load_program_full_chain()` use the rhino process that is already initialized and running. This greatly reduces the overhead, but creates a little dependency between the individual test cases.

To date, there is about 143 automated tests that together perform more than 60.000 checks. The whole test suite executes under 10 seconds on a modern PC.

Conclusion

The author designed and implemented a solution that enables visual programming of small mobile robots. He chose a suitable visual programming front-end and adapted it for programming line-following robots. He extended the Blockly visual programming language with support for arrays and synchronization to external events. Next, he added several domain-specific features inherent to line-following robots. A code generation layers were implemented, which compile the visual program representation into a bytecode that is executed on board of a mobile robot.

The Author designed and implemented a compact Virtual Machine that can run on an 8-bit microcontroller with minimum system requirements. Yet, the solution has full support for variables, expressions, conditional statements, loops, static arrays, functions calls and recursion. The implementation occupies 5kB of FLASH footprint and 30 bytes of RAM when compiled for the AVR platform. The recommended additional RAM space for the user program virtual memory is 500B at least.

The solution is stable and robust, which was confirmed by a comprehensive automated test suite that supplements the implementation.

Text Summary

We analyzed existing visual programming languages for programming mobile robots. We explained the challenges implied by event-driven programming paradigm and pointed out why this approach might not be suitable, both in terms of performance requirements and complexity imposed on the user. We chose procedural programming paradigm as a viable alternative.

We analyzed two open-source visual programming front-ends for building visual programming editors. The analysis showed that the Blockly library is highly customizable and although it primarily targets dynamically typed languages, it can be further extended with type checking. For these reasons, we selected it as a front-end of the proposed visual programming language.

Then, we characterized the category of small line-following robots and depicted their common characteristics. We continued with detailed analysis of available computational resources on the target robotic platform. The analy-

sis showed the resources are very limited, which imposes some restrictions and challenges that must be addressed during the design phase.

We analyzed existing implementations of virtual machines and interpreters that could potentially run on such performance-limited system. Finally, we explained the importance of test automation and suggested two testing frameworks.

In the Design part, we laid out the specification of the Visual Programming Language and defined its features. We explained the concept of a universal data type and the interpretation contexts in which it can be used. Two basic collections of this data type were specified. Next, an important synchronization feature of the language was presented, which allows to synchronize the program flow to external events.

We proposed the design of the automated test suite and showed how mocking of dependencies is achieved in the CppUTest framework. We demonstrated the unit testing and integration testing on code examples.

In the Implementation part, the proposed Virtual Machine is described thoroughly. The mechanism of scheduling and preemption is explained. Next, the arrangement of the virtual memory is described and the input/output approaches are explained in detail. Then, the instruction set of the Virtual Machine is documented.

Finally, the Code Generator is described. We reason about the need for an intermediate assembly language representation and explain the code generation process. We demonstrated how the individual language features are compiled to a bytecode on numerous code examples and specified the subroutine calling convention. The support for library functions is explained. After all, the integration test suite implementation is described.

Future Works

While this thesis described a complete solution that enables visual programming of a mobile robot, there are opportunities how the work could be extended further, for example:

Backward compatibility of bytecode might be a desired feature if it is expected that the instruction set of the VM will be extended in future. For this reason, it would be beneficial to organize the instruction space into distinct categories, based on the data stack size difference before and after applying the particular instruction.[1] This mechanism could be designed such that legacy implementations of the VM can silently process unknown instructions without corrupting the data stack integrity. (Of course, there would be no other action

[1] For example, the `CLONE` instruction will be in category "+1", as it pushes one element on the stack. On the other side, the `ADD` instruction pops two elements and pushes one, therefore, it would be in the "-1" category.

for such unknown instruction, but in certain situations, this might be a desired behavior.)

Bytecode compression could be implemented to reduce the program transmission time. It was calculated that the instruction entropy of the programs in the test suite is 6.078 bits. Currently, each instruction is encoded in 8 bits. The test suite is reasonably big and contains representative user programs, therefore, it can be expected that in real use-cases, programs will have similar instruction frequency histogram. This would allow to reduce the transmission time by approximately 25% (best case).

Bytecode optimization layer could be implemented to achieve smaller program footprint and better runtime performance. Global Stack Allocation [42] method or similar could be implemented. The integration test suite can verify if the optimization layer works and some statistics about the efficiency of the optimization can be generated.

Bibliography

[1] AmForth. http://amforth.sourceforge.net/. Accessed: 2017-01-02.

[2] Apache license. http://www.apache.org/licenses/LICENSE-2.0. Accessed: 2017-01-02.

[3] Arduino. https://www.arduino.cc/. Accessed: 2017-01-02.

[4] Blockly discussion group. https://groups.google.com/forum/?fromgroups=#!topic/blockly/rGMz3NnFVRI. Accessed: 2017-01-02.

[5] Check - unit testing framework for C. https://libcheck.github.io/check/. Accessed: 2017-01-02.

[6] CodeBug. https://www.codebug.org.uk. Accessed: 2017-01-02.

[7] CppUTest. http://cpputest.github.io/. Accessed: 2017-01-02.

[8] Espruino. http://www.espruino.com/. Accessed: 2017-01-02.

[9] Fishertechnik. http://www.fischertechnik.de. Accessed: 2017-01-02.

[10] flex, the fast lexical analyzer generator. http://flex.sourceforge.net/. Accessed: 2017-01-02.

[11] GNU Bison. https://www.gnu.org/software/bison/. Accessed: 2017-01-02.

[12] Google Blockly. https://developers.google.com/blockly/. Accessed: 2017-01-02.

[13] Google Blockly Factory. https://blockly-demo.appspot.com/static/demos/blockfactory/index.html. Accessed: 2017-01-02.

[14] High-speed line-follower rules. http://www.robotgames.com/wp-content/uploads/2012/01/WCRG-2012-High-Speed-Line-Follower-Rev0.pdf. Accessed: 2017-01-02.

[15] Hour of code. https://hourofcode.com. Accessed: 2017-01-02.

[16] Istrobot. http://www.robotika.sk/contest/2016/index.php?page=rules&type=follower. Accessed: 2017-01-02.

[17] Lego Mindstorms. https://www.lego.com/en-au/mindstorms. Accessed: 2017-01-02.

[18] MIT Scratch. https://scratch.mit.edu/. Accessed: 2017-01-02.

[19] NanoVM - Java for the AVR. http://www.harbaum.org/till/nanovm/index.shtml. Accessed: 2017-01-02.

[20] Open Roberta Lab. http://lab.open-roberta.org/. Accessed: 2017-01-02.

[21] python-on-a-chip. https://code.google.com/archive/p/python-on-a-chip/. Accessed: 2017-01-02.

[22] Wonder Workshop. https://www.makewonder.com/. Accessed: 2017-01-02.

[23] Wonder Workshop FAQ. https://help.makewonder.com/customer/portal/questions/12615136-concurrent. Accessed: 2015-07-23.

[24] Wonder Workshop FAQ. https://help.makewonder.com/customer/portal/questions/11310719-any-possibility-of-\downloading-behavior-to-the-robots-directly-. Accessed: 2015-07-23.

[25] Wonder Workshop FAQ. https://help.makewonder.com/customer/portal/articles/1394965-what-are-the-technical-specifications-of-the-robots-. Accessed: 2015-07-23.

[26] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, 2006.

[27] ChaN. Desktop line following robot. http://elm-chan.org/works/ltc/report.html. Accessed: 2017-01-02.

[28] Grinberg Dmitry. uJ - a Java VM for microcontrollers. http://dmitry.gr/index.php?r=05.Projects&proj=12.%20uJ%20-%20a%20micro%20JVM. Accessed: 2017-01-02.

[29] Benjamin Erb. Concurrent programming for scalable web architectures. Diploma thesis, Institute of Distributed Systems, Ulm University, April 2012. http://www.benjamin-erb.de/thesis Accessed: 2017-01-02.

[30] Nathalie Gaertner and Bernard Thirion. Grafcet: an analysis pattern for event driven real-time systems. In *PLoP 1999 conference*. Université de Haute-Alsace, 1999. http://hillside.net/plop/plop99/proceedings/gaertner/gaertner.pdf Accessed: 2017-01-02.

[31] Mariana Goranova, Elena Kalcheva-Yovkova, and Stanimir Penkov. Task-based asynchronous pattern with async and await. In *International Scientific Conference Computer Science'2015*, pages 150–155. Technical University of Sofia, 2015. http://e-university.tu-sofia.bg/e-publ/files/2245_TAP.pdf Accessed: 2017-01-02.

[32] James Hancock. *When to Automate Testing A Cost-Benefit Analysis.* Testers' Network, 1998.

[33] Wienand Ian. Computer science from the bottom up. http://www.bottomupcs.com/scheduling.html. Accessed: 2017-01-02.

[34] Charles Eric LaForest. Second-generation stack computer architecture. Bachelor Thesis, University of Waterloo, Canada, April 2007. http://fpgacpu.ca/stack/Second-Generation_Stack_Computer_Architecture.pdf Accessed: 2017-01-02.

[35] Cardelli Luca. Type systems. http://www.lucacardelli.name/Papers/TypeSystems.pdf. Accessed: 2017-01-02.

[36] Daniel J. Mosley and Bruce A. Posey. *Just Enough Software Test Automation.* Pearson Education, Inc., 2002.

[37] Fraser Neil. Loops - google/blockly wiki. https://github.com/google/blockly/wiki/Loops. Accessed: 2017-01-02.

[38] Staněk Ondřej. PocketBot. http://www.ostan.cz/pocketBot/. Accessed: 2017-01-02.

[39] Staněk Ondřej. PocketBot2. http://www.ostan.cz/PocketBot2/. Accessed: 2017-01-02.

[40] Jr. Philip J. Koopman. *Stack Computers: the new wave.* Ellis Horwood, 1989. http://users.ece.cmu.edu/~koopman/stack_computers/index.html Accessed: 2017-01-02.

[41] Asaftei Robert. http://twinsen.info/project-details.php?id=2. Accessed: 2017-01-02.

[42] Mark Shannon and Chris Bailey. Register allocation for stack machines. In *22nd EuroForth Conference*, pages 13–20, 2006. http://www.complang.tuwien.ac.at/anton/euroforth2006/papers/shannon.pdf Accessed 2017-01-02.

[43] Dočekal Tomáš. Fuzee linefollower. http://www.bodie.xf.cz/files/fuzee.htm. Accessed: 2017-01-02.

[44] Renxin Wang. MY-BASIC. https://github.com/paladin-t/my_basic. Accessed: 2017-01-02.

[45] Wikipedia. Position-independent code. https://en.wikipedia.org/wiki/Position-independent_code. Accessed: 2017-01-02.

[46] Wikipedia. Saturation arithmetics. https://en.wikipedia.org/wiki/Saturation_arithmetic. Accessed: 2017-01-02.

[47] Wikipedia. Two's complement. https://en.wikipedia.org/wiki/Two%27s_complement. Accessed: 2017-01-02.

[48] Wikipedia. Variable shadowing. https://en.wikipedia.org/wiki/Variable_shadowing. Accessed: 2017-01-02.

[49] Clifford Wolf. Embedvm: a small embeddable virtual machine for microcontrollers. http://www.clifford.at/embedvm/. Accessed: 2017-01-02.

[50] Hanzálek Zdeněk. Petriho sítě a GRAFCET. http://labe.felk.cvut.cz/~tkrajnik/sdu/data/K333/Hanz01.PN.automatizace.pdf Accessed: 2017-01-02.

List of Abbreviations

ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
CPU	Central Processing Unit
DS	Data Stack
DSP	Digital Signal Processing
EEPROM	Electrically Erasable Programmable Read-Only Memory
FP	Frame Pointer
FPU	Floating Point Unit
GCC	GNU Compiler Collection
IDE	Integrated Development Environment
Input/Output	Virtual Machine
JVM	Java Virtual Machine
LED	Light-Emitting Diode
MCU	Microcontroller Unit
PC	Program Counter
RISC	Reduced Instruction Set Computing
RPN	Reverse Polish Notation
SRAM	Static Random Access Memory
TDD	Test-driven Development
VM	Virtual Machine
WCET	Worst-case Execution Time
XML	Extensible Markup Language