Western Norway University of Applied Sciences

Common programming interface for multiple types of robots

Felles programmeringsgrensesnitt for flere typer roboter

Bachelor in Computer Engineering

Department of Computing, Mathematics and Physics

Faculty of Engineering and Business Administration

Submission date: 06/06-2017

Number of words: 10 811

Tore Martin Solhaug

Magnar Gya

I confirm that the submitted work is independently produced and that all references and sources are clearly stated according to *Forskrift om studier og eksamen ved Høgskolen i Bergen, § **9-1.***

Western Norway University of Applied Sciences

Avdeling for ingeniør- og økonomifag

Institutt for data- og realfag

TITTELSIDE FOR HOVEDPROSJEKT

Rapportens tittel: Felles programmeringsgrensesnitt for flere typer roboter.	*Dato:* ***06.06.2017***
Forfattere: Magnar Gya, Tore Martin Solhaug	*Antall sider u/vedlegg:* ***29*** *Antall sider vedlegg:* ***8***
Studieretning: Bachelor i Ingeniørfag, Data	*Antall disketter/CD-er:* 0
Kontaktperson ved studieretning: Adrian Rutle	*Gradering:* Ingen
Merknader: -	

Oppdragsgiver: Høgskolen på Vestlandet	*Oppdragsgivers referanse:* -
Oppdragsgivers kontaktperson: Adrian Rutle	*Telefon:* 55 58 77 91

Sammendrag:

This project is a tool for programming multiple different robots using a single input language. It can be used to simplify development of programs for robots.

Stikkord:

Programming	Robot	Code generation

Høgskolen i Bergen, Avdeling for ingeniør- og økonomifag

Postadresse: Postboks 7030, 5020 BERGEN Besøksadresse: Inndalsveien 28, Bergen

Tlf. 55 58 75 00 Fax 55 58 77 90 E-post: post@hib.no Hjemmeside: http://www.hib.no

PREFACE

This report is a representation on the work done in creating a Common programming interface for multiple types of robots.

We are a group of two students who have worked on this project.

The goal for this project was to create an abstraction layer for coding multiple types of robots. To do this we've created a language which can be parsed to any robot that has a configuration made for it.

We would like to thank our HVL contact person Adrian Rutle for the help he has given us.

TABLE OF CONTENT

1 INTRODUCTION

Robots that can be coded to do small tasks have become many, and languages and coding practices have grown at a similar pace. We want to simplify the process of programming different robots.

1.1 Goal and motivation

The goal of the project was to create a programming tool that programs robot behaviour to output code for multiple different types of robots, in different languages and APIs, based on the same input. The purpose of this is to make it easier to make platform-agnostic programs for robot behaviour, to make robot programming easier. The project intended to serve as a proof of concept and demonstration of the possibility for such a system to work effectively. As such we also made a demonstration program that can be run on the targeted robots, it will demonstrate the tool's capability. Having a system that can create programs which can run on any robot can be helpful, as more and more robots are introduced into peoples lives.

1.2 Context

This assignment is a continuation of previous work done by students at the Department of computing, mathematics and physics, as research into modelling for robots. Our project is not a direct continuation of the previous work, but is similar in nature. We will focus on code generation for multiple robots, and not focus too much on any specific task for the robots.

1.3 Limitations

We wanted to create both a diagrammatic and a textual programming tool to be used to program the robots. But due to time constraints and a lack of experience with Eclipse Sirius and EMF (an environment and framework respectively for developing model-based software), we decided against implementing the visual programming tool. We instead used Xtext to create only a textual programming language.

1.4 Resources

As the project result is a tool which can program multiple robots we needed to have robots to run code on.

We have used 2 Lego Mindstorms EV3 robots and an Arduino Shield Parallax Robot with a Pixy camera. Additional equipment was acquired, but not utilised during development.

Lego Mindstorms EV3

We used Lego Mindstorms EV3 (LEGO Group, 2017) robots running the ev3dev operating system (ev3dev.org, 2017). Our unit is configured with the following sensors:

An ultrasonic sensor facing forward, a touch sensor facing forward, a colour sensor facing

downward.

In addition it has two motors connected to wheels, one on each side.

With these components, it is capable of driving forward, backward and turning, while also sensing obstacles either by using the ultrasonic sensor to detect distance, or when coming into contact with an obstacle using the touch sensor. The colour sensor underneath allows it to detect the colour of a surface beneath it, and can be used to detect boundaries and edges.

Through the ev3dev OS we can program it in Python to achieve these tasks.

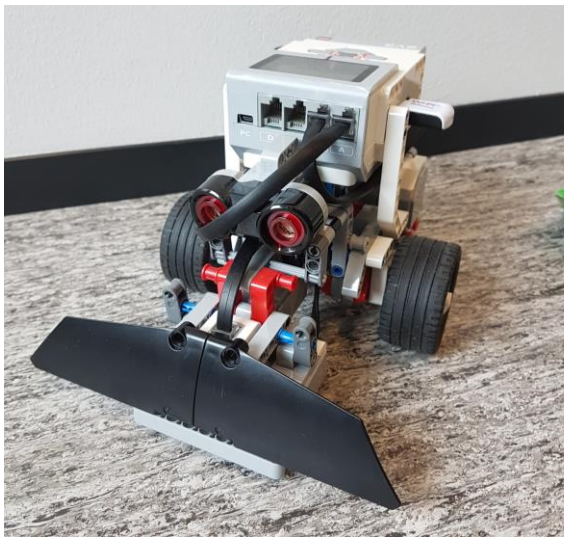


Fig. 1.1: Lego Mindstorms EV3 robot

Fig. 1.2: ev3dev logo

Arduino Shield Parallax Robot

We used an Arduino Shield robot which we could deploy C-code on through the Arduino IDE (Parallax, 2017). Our unit is configured with the following sensors:

Two infrared sensors facing forward, two touch sensors facing forward and a Pixy camera pointed forward.

In addition it has two motors connected to wheels, one on each side.

With these components, it is capable of driving forward, backward and turning, while also sensing obstacles either by using the infrared sensors to detect obstacles, or when coming into contact with an obstacle using the touch sensors. Unlike the EV3, with its current configuration the Arduino Shield can detect whether an obstacle is on its left, right or front. It also has the ability to identify the relative location of colours using the Pixy camera. This allows it to track objects with markers or distinct colours.

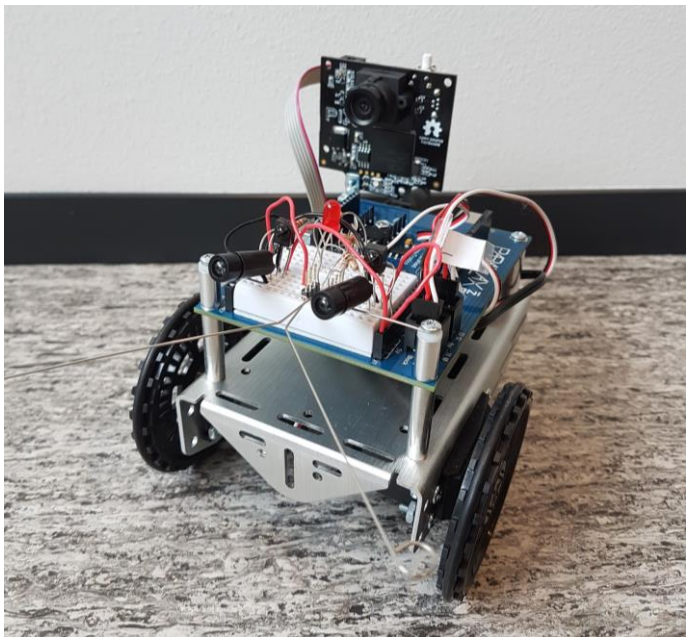


Fig. 1.3: Arduino Shield Parallax Robot

Fig 1.4: Arduino logo

Pixy camera

We mounted a Pixy camera on the Arduino Shield Robot to allow it to track objects and markers. The Pixy camera can be programmed to learn to identify a specific colour, and to provide information about where it detects that in its field of vision (Charmed Labs, 2017).

Fig. 1.5: Pixy logo

Software

In addition to the physical hardware, we required some software tools to enable us to develop our system. We used Eclipse Luna with Xtext to create a Eclipse plugin that allows a user to write code in the language we created.

We also used Eclipse to make a Java library portion of the system.

Fig 1.6: Git logo

Fig 1.7: GitHub logo

We used GitHub to manage our code, allowing us to easily share the source code with each other and collaborate.

2 PROJECT DESCRIPTION

2.1 Practical background

2.1.1 Project owner

Western Norway University of Applied Sciences, Institute of Computing, Mathematics and Physics is a university college institute located in Bergen. The institute has 42 employees (Bergen University College, no date) and is a part of the Western Norway University of Applied Sciences, Campus Kronstad.

The Western Norway University of Applied Sciences was made when Bergen University College, Sogn og Fjordane University College and Stord/Haugesund University College merged on the 1st of january 2017. The university has a total of 1 578 full-time equivalent, and has 16 637 students(Western Norway University of Applied Science, 2017).

2.1.2 Previous work

While there have been other projects at Bergen University College that worked with robot programming, and code generation for robots (Western Norway University of Applied Science, no date). In the previous work, They made a tool where you can draw models for robot behaviour. Through this you can make scenarios, such as a mapping scenario which maps out a room, which was the demonstration program they made.

Our project was more tangential to this work. This is because we were more concerned with finding ways to generalize the language, than having a goal for the robots in mind.

2.1.3 Initial requirements specification

The task we were given was to investigate how to create an abstraction level on top of multiple existing robot programming languages, in order to be able to provide instructions in a common format for multiple different types of robots. While normally, you would need to program each robot individually, with its own language and API, we would use this abstraction to allow for a single set of instructions to be translated into the code each robot requires to carry out the intended actions. We would then use this system to create a demonstration with multiple types of robots competing or cooperating.

2.1.4 Initial solution idea

The initial idea was to create a diagrammatical and/or textual language, that would then generate a programmatic representation of the desired program. This representation would then be run through one or more parsers that would interpret it and generate code for the desired target robots, based on configuration files that described the robots' capabilities and implementations of the general methods that it was instructed to carry out.

2.2 Literature background

The need for robot programming based on the actions it would perform, and not detail implementation, has been around for a long time (Lozano-Perez, 1983, 822). This type of robot programming was made as an earlier project on robot programming and modelling (Western Norway University of Applied Science, no date) at this University. The tool created in this project generates code to the Lego MindStorms EV3 based on a model. The model lets you place tasks, and the conditions for their execution, but they wished to expand the possibilities of the project. They wanted to be able to generate code for different robots. Our work is an attempt to implement such a system, where we create a tool for writing task-level code that is translated into runnable code for multiple types of robots.

3 PROJECT DESIGN

Our main project was to create a system for programming multiple different types of robots with identical input code/instructions. The conversion from this common instruction set/code/model into each robots' code should then be handled by our program. In other words, we were to make a programming language, and a system for translating that language into working code for multiple different robots, using different programming languages between them, and with different implementations of the same general instructions needed for each.

In our work, we needed to use our knowledge and experience in regards to programming, and programming languages.

3.1 Approach

We used Eclipse Luna as our main IDE. In it, we created a programming language and generation of intermediate code using the Xtext plugins, in addition to a Java library for transforming the intermediate code into the code to be run on the robots. The intermediate code is a small Java program that contains the code required to build and call the code-generation functions necessary to output the final code. One such small program is built per script written in our language.

3.1.1 Approach description

We started by familiarising ourselves with the technologies and systems we used, such as Xtext. We looked into the APIs used by the different robots, and their capabilities, in order to find what they have in common. This enabled us to abstract those capabilities into the common programming tool, allowing instructions in the script to be translated in a way that each type of robot can understand. Our initial goal was to use this knowledge to develop the ability to program an Arduino Shield Robot in addition to the EV3 Lego robot, using the same common programming tool.

Once we had enabled the programming tool to output effectively equivalent code to two robots, we then started working on a demonstration program. This program is a set of instructions that allows the robots to compete in a task based on code produced by the programming tool.

3.2 Specification

We created a system that takes input in a programming language we designed. This code input is then translated into a small Java program, which when it is run produces the final runnable code for the robots targeted in the original input.

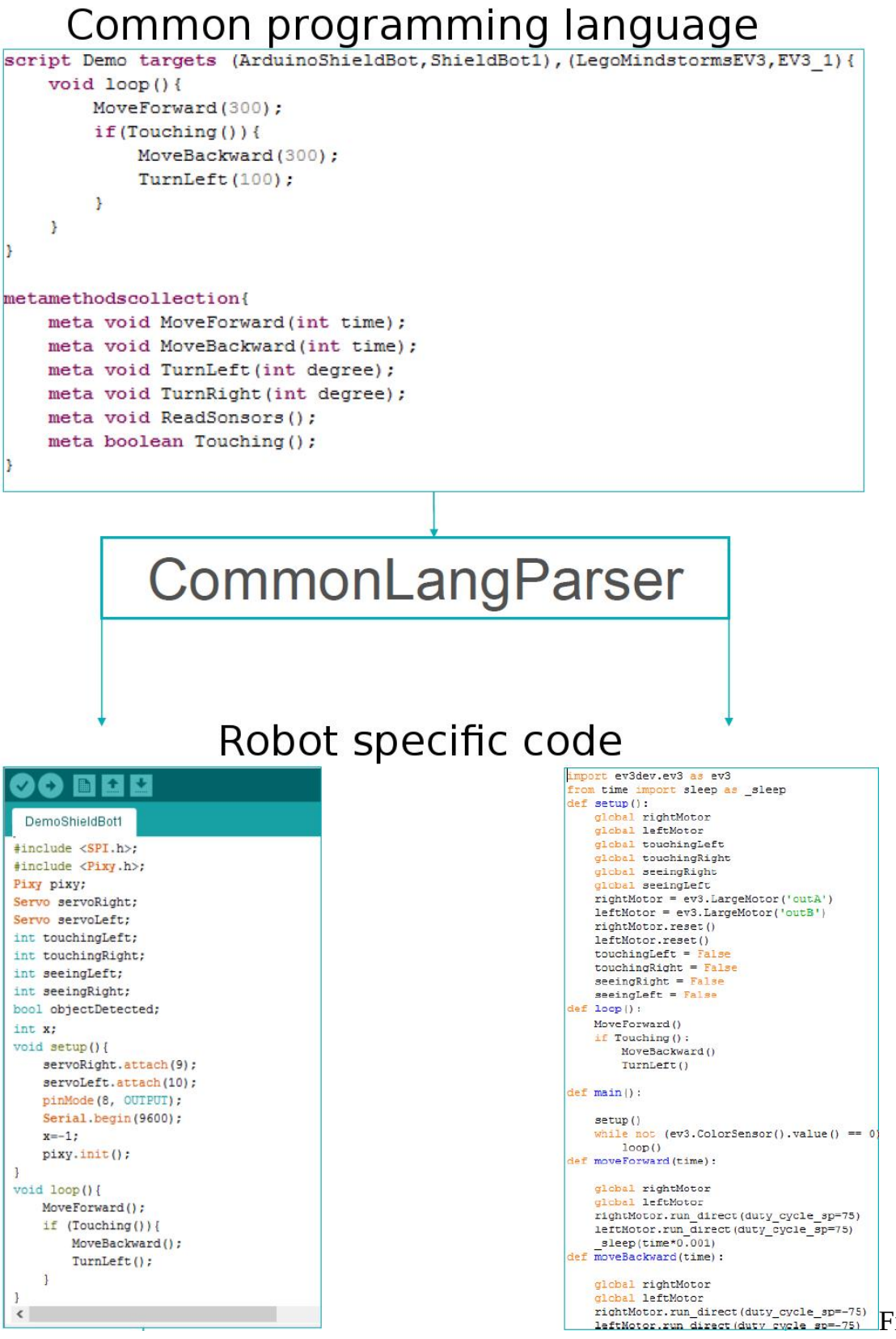


Fig. 3.1: How it works

Input language

For the input language, we created a language in Xtext, and the necessary validation for that language, in addition to Java code generation, which translates the input language into the Java code representation. This language contains the grammatical elements needed to write a description of desired behaviour in the target robot based on input. I.e., the language allows the user to specify general behaviour (like go forward, send some signal, activate some physical component) to be executed depending on input collected from sensors or other communications input. In order to read the sensor input, and perform tasks, the user calls something called “metamethods” in their code. These metamethods represent tasks a robot can perform, and their availability depends on the robot’s functionality. A user should know beforehand which metamethods are available for their robot if they are using an existing configuration, or determine which ones should be available if creating a new configuration themselves.

The system is flexible, with the ability to add functionality (metamethods) for robots, and adding new robots themselves without needing access to the source code of the project.

Java programs and Java library

The generated Java program contains the necessary code to create an object representation of the code written by the user. When run, these generated Java programs themselves generate the final code to be deployed on the robots.

In order to make these Java files runnable programs, we created a library of Java code that contains all the necessary components for the code generation programs. This library contains the definitions of the classes for all the language objects to be used to represent the program, and the code needed to generate the finished code files. It also contains the specification for creating a generator for a given language. The library initially only contains generators for Python and C (usable also with C-like languages), as those are the languages used by the robots we developed with, but a user may write their own generator for additional languages. Writing new generators, however, requires access to the source code.

Robot configuration

When it comes to the configuration of robots, there are three levels of specification: Language, platform and robot. A robot accepts code written in a specific language, and the general robot configuration file specifies which language that is. The library must contain the generator for that language in order to produce code for the robot. The general robot configuration typically represents the physical base unit that is to be programmed, but can be more specific according to need. It specifies which metamethods to be available, and a specification for how they are to be implemented, but contains only a minimal amount of robot code. The specific robot configuration file provides the implementations of the methods used by the metamethods in the general robot configuration file, in addition to any configurable values such as names, numbers or other identification data necessary to use physical components that may be configured in multiple ways.

3.3 Selection of tools and programming languages

We have used Eclipse Sirius as our IDE, and programming was done in C for the arduino and python for the lego robot. We used Xtext to develop a plugin that allows the use of an editor containing the grammar, validation and Java code generation, and Java to create the robot code generation for the last step of code generation. We also used xml to create the configuration files for the robots.

Xtext

Xtext is a plugin for Eclipse that allows for developing languages (The Eclipse Foundation, 2017). Xtext lets you describe the grammatical rules you want for your language, letting you create definitions for data types, grammatical structures and the relationship between these. Based on your rules, an editor is created for your language. When using the editor, the input is then checked against the rules, both to ensure that the input is valid according to the rules, and also to transform the input into a model representation of the code. Xtext generates classes that represent your grammatical structures, and when using the editor, these classes are instantiated into an object structure.

In addition to the grammar for the input, you can also specify custom validation rules. You can create a validator class that will run through the model. You can specify certain relationships and structures that are allowed by the grammar to be invalid, and provide error messages and warnings accordingly.

Finally, you can create a code generator that generates code output based on the model created by the editor. We chose to write a generator using Xtend, that generates runnable Java classes to generate the final code.

Fig. 3.1: Xtext logo

Java

We chose to use Java for the robot code generator, as it is a familiar language to us and that we could then easily keep everything in Eclipse, both for development of the tool and for deploying and using the tool once completed. Using a general-purpose language allows us to easily customise the code generation, and to easily combine multiple different types of data in a flexible way.

Fig. 3.2: java logo

XML

We have chosen xml for the robot configuration files. This is because xml is a common format for configuration files. It's intuitive, and easy to learn. We wished that the system should be easy to change to the users needs, and to make it easy to add more robots and functionality. For this we needed a format that is easy to read and easy to write.

Fig. 3.2: xml icon

3.4 Project development method

We chose to use agile development as our development method. We chose it because we only are 2 people on the development team, so we can easily communicate. We also had very adaptable goal for the project, so that we could together agree on how we wanted to develop further as we met hurdles. Our assignment giver was also easy to contact, so if we needed to check if changes we wanted to do were ok, we could easily contact him, and then resume. Agile focuses on creating functionality iteratively, where you try to first make the critical functionality, and then add the rest of the functionality after this is done. For our project that means that we could drop non-critical functionality when we saw that we didn't have enough time to develop them.

We used GitHub for version control. This helped us to easily share the code we had written. We created a repository which contains all components of the system. Internally, the project was divided into a Xtext project for the plugin development, and a Java project for developing the Java library for robot code generation. By doing this, we could both

easily have access to all the source code we needed.

This meant for us that we could develop with less interruptions, and also when we discovered things we wanted to change, we didn’t have to unnecessarily develop them, but could change the plan as we went along.

3.5 Evaluation method

The project contains different parts, which does different tasks. To test we will need to see that any user input will give valid output. In all the layers, but the first one we can expect correct input, as the first layer will validate the any user input, and only let correct input go further down. We use unit tests to test the input language and its generation, and functional testing one the rest.

4 Implementation

We have developed a system capable of translating instructions written in our textual language into multiple sets of code each runnable on a different type of robot. We have demonstrated the ability to have a common system for multiple robots. This will make it easier and quicker to develop general models of behaviour for robots, rather than needing to develop with a specific robot in mind. With the proof of concept, it also makes it easier to create new interfaces to allow for programming additional types of robots using the same system. This means you only need to adapt the interface once for each type of robot, and all models created with the system will be able to be translated to that type of robot.

The result will be helpful for anyone with some level of coding ability, especially researchers and developers, wanting to develop code and programs for robots, as it abstracts the programming into a more general and simplified approach, rather than having to directly write code for the robot on a lower level.

When using the program, the user will interact with an Eclipse Editor which uses a Xtext project for grammar validation. You will edit files which have the extension ".commonlang" in this editor. When you have valid code in this editor, and save the document a java-file will be made. When you run this java-file, the files for the robots chosen will be created. You then have to deploy these to the robot in the way the specific robots needs.

While this is system is easier to use than programming directly on the robots, it isn't easy enough that any person can pick it up and use it. For this it would need further development. The deployment of code is a several step process. A product for the layman should have this process integrated and automated. The coding is a C-like language. A layman might not know how to write the code. If the tool should be released as a product it should either have a diagrammatical tool, or the textual tool should help more in the writing of the code.

4.1 Structure

System structure

The program is divided into multiple components:

Code input

Java code generation

Robot code generation

Input code language

The input code language is a structured, imperative programming language, internally called commonlang. It does not expose any ways of storing state, however it may change states of the program through metamethods if a robot's configuration supports such

behaviour. Its syntax is similar to Java, but does not support object-oriented programming. Only methods and metamethods declared inside the script can be called from it, and only local variables may be referenced from inside a method.

We chose to make a structured, imperative language due to these traits being common between C and Python, the two output languages we have targeted during development. In addition, the sequential nature of the language prevents writing code meant to run in parallel for robots that may not support such behaviour. Having a language that is similar in its structure to the output languages also makes the process of transitioning from input to output easier, due to the similarities between them. We chose to not use an object oriented design, as we thought doing so would complicate the development process more than we thought necessary for the scope of our project. Our target was to create a language that could describe behaviour that would operate in response to the environment more than relying on complex internal data structures. However future development may expand the language's capabilities.

We created metamethods in the language as a way of abstracting the specification of the robots' behaviour from the input code. Since the goal of the project was to let multiple robots take the same input, the detailed implementation would have to be removed from the input code. By instead implementing the metamethods in configuration, and only referencing their definitions in the user code, we could create a clear separation between specifying the implementation of individual bits of behaviour and describing the overall behaviour of the robots.

Input code language structure

File

The user inputs code in the commonlang editor. A file has the following grammatical element as an entry point:

```
CLfile:
     scripts+=(Script)*
     mets=MetaMethods
;
```

The user can as such write as many scripts as they want in the same file, in addition to declaring a collection of metamethods at the end.

Script

A script has a name starting with a capital letter, and declares the specific configurations of robots it is intended for. A script contains a list of methods as written by the user. By convention, a configuration file for a robot will describe the name and parameters of the method to be used as an entry point for scripts. Multiple robots taking code from the same script as such need to have the same entry point, or have each of their entry points defined in the script.

```
Script:
     'script' name=CAPITALFIRST 'targets'
'('robottypes+=(LOWERFIRST|CAPITALFIRST) ','
robotconfigs+=(LOWERFIRST|CAPITALFIRST)')'(','')('robottypes+
```

```
=(LOWERFIRST|CAPITALFIRST) ','
robotconfigs+=(LOWERFIRST|CAPITALFIRST)')'* '{'
          (methods+=UserMethod*)
     '}'
;
```

Method (UserMethod)

Methods written inside the script by the user take the form of UserMethods. These methods have types, names starting with a lowercase letter, and a list of parameters. Inside the method's following block, the user can specify the behaviour of the method.

```
UserMethod:
     type=Methodtype name=LOWERFIRST '('
parameters+=Declaration? (','+ parameters+=Declaration)* ')'
bl=Block
;
```

Block

A block is a list of expressions contained between two curly brackets. `Method` and `StructureExpression` entities contain blocks, with the block serving as the defintion of that method or expression's core behaviour.

```
Block:
     {Block} '{' ( (exs+=SimpleExpression ';' |
exs+=StructureExpression))* '}'
;
```

Expression

```
Expression:
     SimpleExpression | StructureExpression
;
SimpleExpression:
      Crement | Call | Assignment | Return
;
StructureExpression:
     Block | If | For | While
;
```

Expressions form the core of the language. Expressions are the actions taken in the code, and either modify values, trigger other methods and actions, or alter the path of the code through `StructureExpression`.

`SimpleExpression` is the type of expressions that do not alter the flow of the block they are contained in, and typically only take up a single line.

`StructureExpression` is the type of expressions that themselves contain (or are) blocks of expressions. If expressions execute the content of their block if their tested expression evaluates to true, while for and while expressions run their blocks in a loop until their end condition is met.

Summary

There are more minute elements involved, described further in the appendix, while the described above are the main components. All these elements, however, represent equivalent elements in the targeted languages. Our decision to pick the elements that we chose is based on our subjective evaluation of their ubiquity, and their presence in the target languages.

Java code representation

The code generated from Xtext is a class containing a program that creates object instances of the classes from the package `CommonLanguageObjects`. We have 10 classes in this collection. We also have a class, `BotMethods`, to gather all the information from the two configuration files that will be used. The generated objects represent the grammatical structure of the input script, and are designed to contain all the necessary data to generate the final code in C and Python. If more output languages are added in the future, more classes may need to be added to this collection, to represent expressions and statements that are too significantly different in the new language to be easily translated from their current form. These additional classes would then remove some of the responsibilities of the Expression class, as it currently works as a catch-all class for expressions that currently do not require their own class to effectively represent.

The CommonLanguageObjects classes

`Expression`: This just contains a simple `String`, but is also the superclass for the rest of the classes. This class is used directly in cases where there is no specific class for the type of expression used. Assignments to variables, method calls, mathematical expressions and other simple types of expressions are stored directly as this class, with the `String` being a literal representation of the expression. The use of `Expression` as a catch-all class for certain expressions was chosen due to many of the simpler expressions being very similar between C and Python. While many of these expressions are not identical between the two languages, their differences are often minor enough that we considered string manipulation as a simpler approach to adjusting them between languages than adding additional classes for every type of expression in the input language. This decision was in part also made due to the fact that we were learning to design a language as we were doing it, as it would then limit our need to make adjustments on the Java side of the project for changes in the input grammar, structure and generation.

`Block`: This contains an array of Expressions, which can be a simple `Expression`, or any of the other classes in `CommonLanguageObjects`, which extends `Expression`. It is the equivalent of the `Block` type in the input language, and serves the same purpose.

`Declaration`: This is also a simple `String`. This is a separate class so that the parsers can write it correctly when writing the script. This is meant for global variable declarations from the config file.

`Else`: This contains an `Expression`. This will in most cases be a `Block`, or an `If`.

`If` : This contains an `Expression`, which is the condition that is checked, then the `Block` which is done if the expression is true, and an `Else`.

`For`: This has an `Expression` which is the action done before the loop, another `Expression` which is the statement to be tested, an `Expression` to be done at the end of each loop, and a `Block` of `Expressions` to be done in the loop.

`Parameter`: Two `Strings` make up this class, the data type and the content.

`Method`: `Methods` have two `Strings`, the method name and the return type. It also contains an array of `Parameters` and a `Block` of expressions to execute.

`While`: This contains one `Expression` to be evaluated, and one `Block` of commands to execute.

`Script`: `Script` is the outermost part, it contains an array of `Methods`. When generating the code, a script is the only instance of a `CommonLangObject` that is directly passed into the parser. The script itself works as the root of a tree structure that contains all the other objects needed to define its contents.

The class diagram for this looks like this.

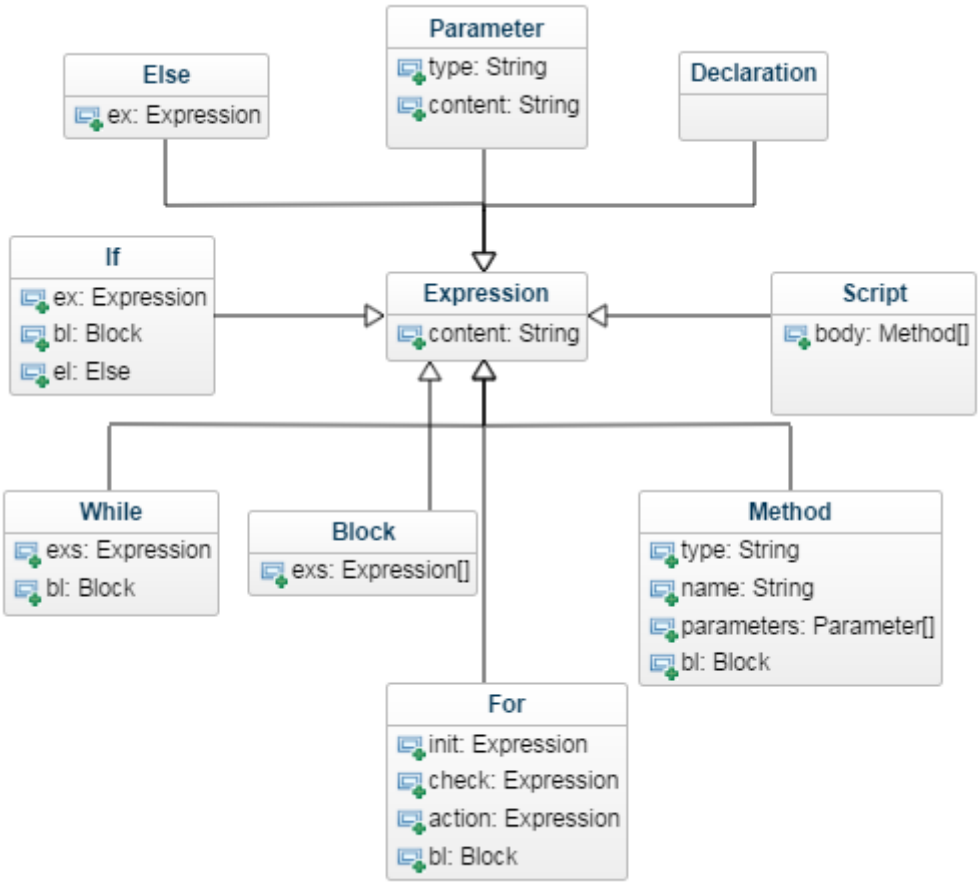


Fig. 4.1: Class diagram of associations

Gathering and validating the code.

`Robotscript` is the next part. It has a `String` which is the name, an `ArrayList` of `Strings` which represent the robots to write to, a `Script` which it gets from the user input and a `ArrayList` of `Methods`.

`Robotscript` then goes through the next operations for each robot in the ArrayList.

The `Robotscript` creates an object of `BotMethods`, which gather all the information from the two configuration files for each robot (The configuration files are explained in Appendix D). It then combines the script methods and the two lists of methods from the configuration files into a single list, and checks this list for method calls to methods not in the list, and removes these method calls.

Creating the file

To write the file you use the `CodeOutputWriter`. The `CodeOutputWriter` takes in a `BotMethods` that contains the information for the robot, a `Script` which is the code made by the user and a `String` that is the name for the file. First you initialize an object of the class with these values, then you call the method `writeToFile()`. This makes a `String` with the code to write to a file. First it writes the parts which are needed for the robot setup. Then it adds the user generated methods. Then it adds the methods from the configuration files, and at last it adds the metamethods from the configuration files.

When this `String` is written, it checks if the folder in "src-gen" exist, if it doesn't exist it creates it. Then it writes the file with the `String` as content, and the name as the file-name and the extension from the `BotMethods`.

4.2 Implementation

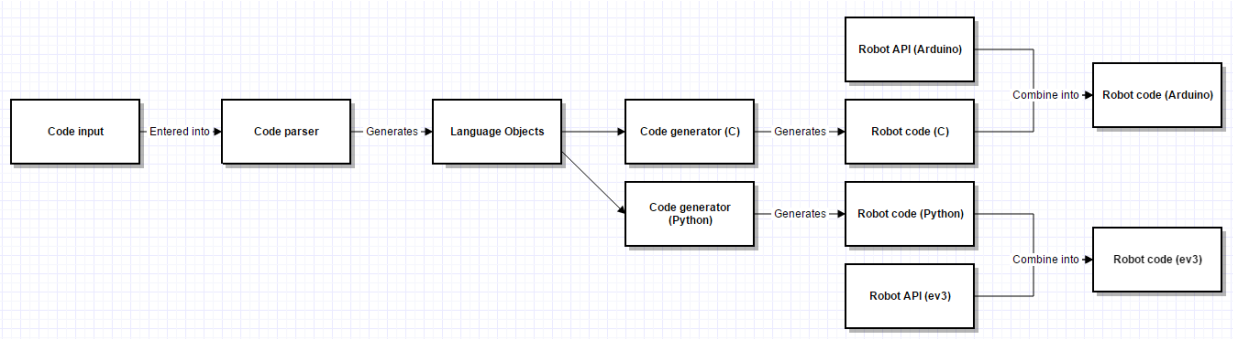


Fig. 4.2: Program flow

The user only interacts with two parts of the project directly. First, the user writes their code in the Xtext-created code editor in Eclipse. The editor validates the code written in it, making sure it conforms to the grammar and the additional validation rules. Error

messages and warnings appear in the editor where the validation rejects the user's code. Only input code that contains no errors will actually proceed to generate output code, which prevents the user from generating invalid code, assuming the validation process is working as intended. The only action required by the user in order to generate code, is for the code input they provide to be valid. The Java program for generating the final robot code is generated when the contents of the editor is updated, and the updated version contains no errors.

The plugin goes through each script in the user code, and generate a Java class. This class is a runnable program, that will generate the robot code when run by the user. This can be done by simply opening the file in Eclipse (it is generated inside the same project as the source files), and running it from there. The program gets most of its functionality by extending the `Robotscript` class. The generated class fills in fields inherited from the `Robotscript` class based on the values the user set in the code editor. The class generates code by instantiating classes from `CommonLangObjects` in a structure that represents the same code structure provided in the input code.

After instantiating the object structure representation of the code, a method is called to generate the final code. This method uses the provided data to find the appropriate configuration files for the target robot(s), and select the correct code generator, a class implementing the `CommonToLanguageParser` interface. It then fetches the data from the configuration files and starts building the contents of the final code. The script is entered into the `CommonToLanguageParser,` returning the code the user provided translated into the target language. This code is then combined with the predetermined code from the configuration file, and a string is produced containing the full code.

After that it sends the information into the `CodeOutputWriter`. This runnable code is then saved to a file with the correct extension. Deploying the code to the robot falls outside the scope of the tool, and is up to the user. For the Lego Mindstorms robot, we connected to it over bluetooth, transferring the code file and running it from the terminal. For the Arduino Shield robot, we connected to it via USB and used the Arduino IDE to deploy the code. A method for deploying the code to the target robot could potentially be provided in the documentation for a robot's configuration, but this is ultimately up to the developer of a robot configuration.

5 EVALUATION(S)

Since our project creates a highly configurable tool, the testing is limited to the configurations that we develop over the course of the project. We must ensure that the tool works as expected and intended for the robots we target during development.

5.1 Evaluation method

Our tool is divided into multiple parts, each performing a different step in the process. As such, each step has certain expectations as to input, and this input must then be supplied in the appropriate form and format.

When evaluating our tool, we must then look at each step and ensure it provides the expected output, then ensure that the next step accepts this as correct input and keeps going.

User code input

Testing the user code input means we need to ensure that all code that is considered grammatically correct is accepted, while code that is incorrect is rejected. It is important that any errors made in code input are caught, as their effects will otherwise propagate into the rest of the program.

To test this, all the different types of expressions and structures supported should be tested to verify that they are accepted. Also, wrongly configured expressions should be tested to ensure they are rejected. Of course, there are endless possibilities when it comes to wrong code, but due to the grammar essentially whitelisting correctly formed expressions, only expressions that would look correct to the grammar, but would be incorrect, need to be tested. This includes mismatched types in expressions, validation of the presence and validity of return statements in methods with a return type and other grammatical structures that would not be caught by the grammar itself.

Java code generation

With the Java code generation, we know exactly what we are targeting. It only generates a class that instantiates the objects as described by the user code, in addition to calling some methods and setting some variables. The implementation of these are not of interest to this step, only the correctness of the generated code itself.

Each piece of code dealing with object creation takes the form of a “new” statement, with the constructor being filled in with the details of fields in the object. As such, this step can be tested by writing a script in the user code that would produce instances of all the various classes used in describing a program, and making sure that all valid configurations are present. As long as the Java code does not contain errors, and no user-identifiable errors are present, this should clear the java code generation.

Robot code generation

Robot code generation must generate code that is runnable on the platform it targets. Otherwise, it is useless. For premade code (metamethods and their implementations, global variables etc.), this can be tested easily by functional testing while writing it.

For user generated code, however, what must be ensured is that each individual bit of code the user supplies is valid and runnable. If we have been provided with a valid RobotScript by the previous step, we know that any errors must lie in the robot code generation. If we have already tested the prewritten parts of the robot's code, we can clear that as well.

At this stage, we must make sure that all language objects are parsed correctly into valid code. This can be checked by opening the code in an editor for the relevant language to ensure no obvious errors are present. Once validated to this extent, functional testing can take place to see if the code works as expected.

5.2 Evaluation results

User code input

We wrote unit tests for the commonlang code input. One test takes in a string of valid code, representing the contents of a file with correct grammar used throughout, and using all types of expressions in the language. It's job is simply to assert that no errors are present, as the code should be valid. Should this test fail, we know that the editor is rejecting code intended to be valid.

We also wrote multiple tests to check for inputs that are known to be invalid. These tests focus mainly on expressions that are invalidated by our validator, rather than by not matching the grammar. Expressions that don't match the grammar are assumed to fail by default, as the editor would not be able to parse them.

Java code generation

The valid code test from the previous step also generates java code containing all the different types of language objects. As such, we will already have a java file to check for this step from that.

In order to verify that the java code generation is executing correctly, we need to check the generated java code and ensure that it is valid, and that it contains all code specified in the source code file correctly formatted.

Invalid Java code is easily spotted once generated. In addition, we have automated testing by comparing the generated code from the valid code input script to its intended output in a unit test.

Robot code generation

For robot code generation, we tested creating scripts and running them on the robots. Some scripts can be found under Appendix F. At this final stage, no scripts that were validated in the previous steps were found to be non-functional once deployed on a robot.

6 RESULTS

When we started working on the project, we made this visualization of components. The ones who are marked with a green check are components we completed.

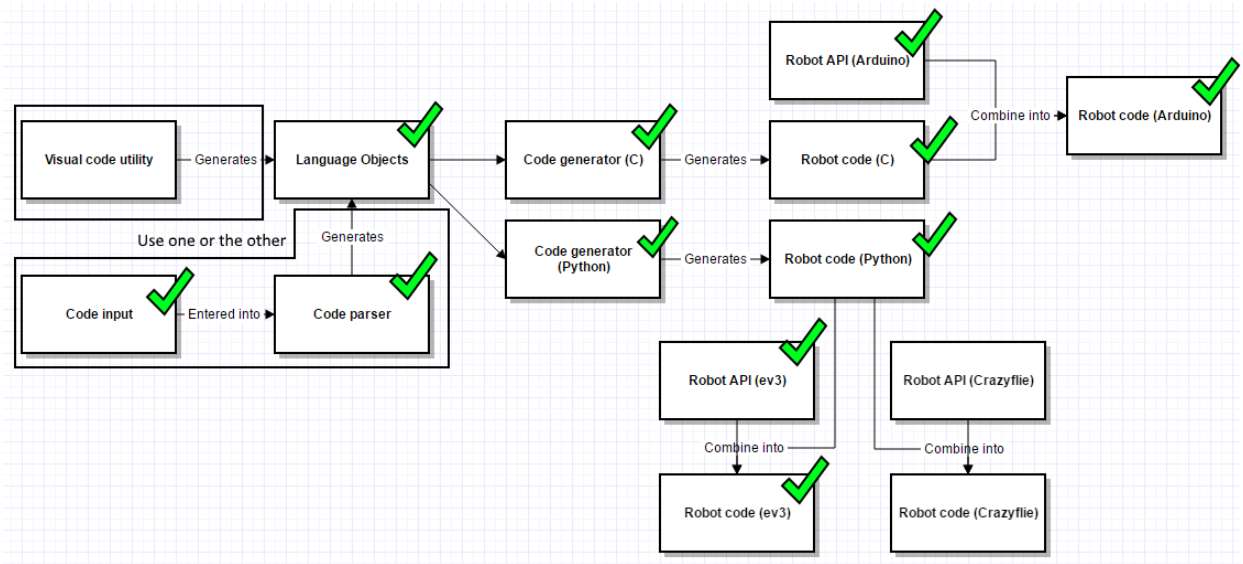


Fig. 6.1: Component completion diagram

We started with the goal of making a tool where you could make a visual model which could generate code for the Arduino Shield Robot, the LegoMindstormsEV3 and hopefully the Crazyflie 2.0. The requirements we agreed on to have a successful assignment were a code parser which could parse to at least 2 of these robots. As you can see from the illustration above, we've completed these requirements. Towards the end of the project we found out that we hadn't gotten familiar with the tools for making a modelling IDE, but expanding this project to also have a modelling aspect should be possible, and could be a project for students after us.

While we currently only have code output for 2 robots, this can easily be expanded, as explained in the Operation and Maintenance part of the Appendix D.

7 DISCUSSION

7.1 What we've achieved

We created a language and a code generation tool that allows a user to write code for multiple robots, each running on a different platform with a different language and configuration, without the user needing to write different code for each type of robot. This allows a user to write a program based on the desired behaviour of a robot without regards to the specific implementation. In order to allow an user to do so, a robot must have a set of configuration files written for it beforehand, however once this work has been done, new users can reuse these configuration files (possibly with minor changes) to allow them to program for the targeted robots.

7.2 Compared to the goal

The outcome of the project fulfills our goal as written. Our goal has changed slightly since the first iteration. The original version states that we are to make a modelling tool, while the more recent version states that we are to make a programming tool. The difference is minor, but reflects our decision to switch focus from making a more general modelling tool, that might be interacted with through either text or visual modelling, to creating only a text-based tool. Our tool then creates an internal model that is far more specific to our implementation than originally thought.

So while the goal has changed, the change to our goal can be seen as almost only a difference in semantics. It does however reflect a larger change in our approach.

7.3 Compared to the requirements

The requirements of our project from the first meeting with the assignment giver was:

- Make a language for at least 2 of the robots available to us.

As a further goal we should also try to hit these requirements

- Make a language that can be used to make different robots cooperate on a task.
- Make a language that can be used to make different robots engage in a battle scenario.

The tools we've created fulfills all of these requirements. Making the robots cooperate and engage in a battle scenario is a little hard, with the missing ability to communicate between the robots, but we have made a battle scenario which works even without communication.

The tools can also get more functionality added in at a later date, which allows for communication to be added. With communication between the robots, more cooperation and battle scenarios can be created.

7.4 The effects of changes made to goal and approach

In our initial stated approach, we talk about using an existing tool and modifying it to achieve our goal. Over time, we changed this, as we concluded that it would be easier and more realistic for us to develop our own tool from the ground up with our goal in mind. While this made the project more approachable to us, it also meant the resulting tool was not based on an existing modeling framework. This means that somebody wants to continue development on this tool in the future, and adapt it for implementation in an existing modeling framework, some of the work we have done will likely need to be repeated, or at least heavily modified.

7.5 Changes in plan if we were to redo the project

If we were to do this project again, knowing what we know now, we would have likely started with a more focused effort on learning about a modeling framework to allow us to build a more standardised solution. While this would likely have made the project more challenging for us, and made success less certain, it would have also resulted in a tool that would have been easier to further develop for others with more experience with modeling tools. Our initial wish to create a visual programming tool would likely have been much easier to achieve if we had chosen this approach, and as such would possibly have been part of the completed tool. We would also start by focusing more on the code generating from the start, and not focus so much on understanding the robots first. While taking the time to understand the robots helped us make the code generator output runnable code from the start, it also narrowed down the view of how the code should generate. A code generator which primarily outputs correct code, and secondly outputs robot code would make it more adaptable for future development.

7.6 Further work

We made a complete tool that allows a user to program robots, however there is still much work that may be done to increase its usefulness. Adding the ability to program more robots can be done with relative ease if the robots accept code in either Python or C-like languages. If other languages are desired, however, the source code must be accessed. Only minor changes need be made to existing code, but at least an additional class must be created, that can output code in the required language. It would have been desirable to make adding new languages possible without source code access, and that is something that could be addressed in further work.

In this project we have focused on the code generation. Further work could focus on ease-of-use feature, like deploying code in the same environment as the code generating is in. Other features is a diagrammatical development environment.

8 CONCLUSIONS

The goal of the project was to create a tool that allows a user to create programs for robots without needing to know the specifics of how a robot functions. We achieved this goal by creating a system where the user writes textual code in a common language we created, and can then generate runnable code for robots based on that code input. As long as the targeted robot accepts code in a language supported, and the required configuration files are present, the process is as simple as writing the code, and executing the code generation. However, when it comes to adding new robots, especially ones where the language required for the robot is not supported by our system, more in-depth knowledge is required. Our intention is that additional robot configurations and modifications to the system should be openly available, such that once the required coding and configuration has been performed once, it does not need to be done again.

The system should be reasonably simple to use for developers and others with some familiarity with programming and software development, but may be too technical in nature for the average user. It requires a user to use Eclipse, install plugins and set up a project in the correct manner, in addition to being able to write code. Further work could make the process easier for less technically knowledgeable users, for example by converting the system from a plugin and a library into a standalone tool, possibly with drag-and-drop diagrammatical programming rather than textual code input.

The system could be of interest for developers working with robots, who may want a easy and fast process to prototype and test programs for robots. It may also be of interest to hobbyists who enjoy tinkering with robots. The advantage of this system over simply developing code specifically for each robot, is that once the initial configuration has been done for a number of robots, the process for creating programs for these robots will be easy and consistent, and can be done in a single development environment.

9 LITERATURE/REFERENCES

- LEGO Group (2017) *31313 MINDSTORMS EV3 - Products Mindstorms LEGO.com* [Internet]. Available from: <https://www.lego.com/en-us/mindstorms/products/mindstorms-ev3-31313> [Read June 1st 2017]
- ev3dev.org (2017) *ev3dev Home* [Internet]. Available from: <http://www.ev3dev.org/support/> [Read June 1st 2017]
- Parallax Inc. (2017) *Robot Shield with Arduino | 32335 Parallax Inc* [Internet]. Available from: <https://www.parallax.com/product/32335> [Read June 1st 2017]
- Charmed Labs (2017) *Pixy (CMUcam5) - Charmed Labs* [Internet]. Available from: <http://charmedlabs.com/default/pixy-cmucam5/> [Read June 1st 2017]
- Bergen University College (no date) *Tilsette - Institutt for data- og realfag - Høgskolen i Bergen* [Internet]. Available from: <http://www.hib.no/om-hogskolen/avdeling-for-ingenior--og-okonomifag/institutt/institutt-for-data--og-realfag/ansattliste/?id=929> [Read June 2nd 2017]
- Western Norway University of Applied Science(2017) *Nøkkeltal for HVL - Høgskulen på Vestlandet* [Internet]. Available from: <https://www.hvl.no/om/nokkeltal/> [Read June 4th]
- Western Norway University of Applied Science (no date) *Master Thesis - Robot Programming and Modelling | ICT Engineering* [Internet]. Available from: <http://prosjekt.hib.no/ict/master-thesis-robot-programming-and-modelling/> [Read June 2nd 2017]
- Lozano-Pérez T., 1983, Robot Programming. *PROCEEDINGS OF THE IEEE* [journal] 71(7). Available through IEEE Xplore Digital Library <http://ieeexplore.ieee.org/abstract/document/1456949/> [Read June 3rd 2017]
- The Eclipse Foundation (2017) *Xtext - Language Engineering Made Easy!* [Internet]. Available from:<https://eclipse.org/Xtext/> [Read June 1st 2017]

APPENDICES

Appendix A: Dictionary

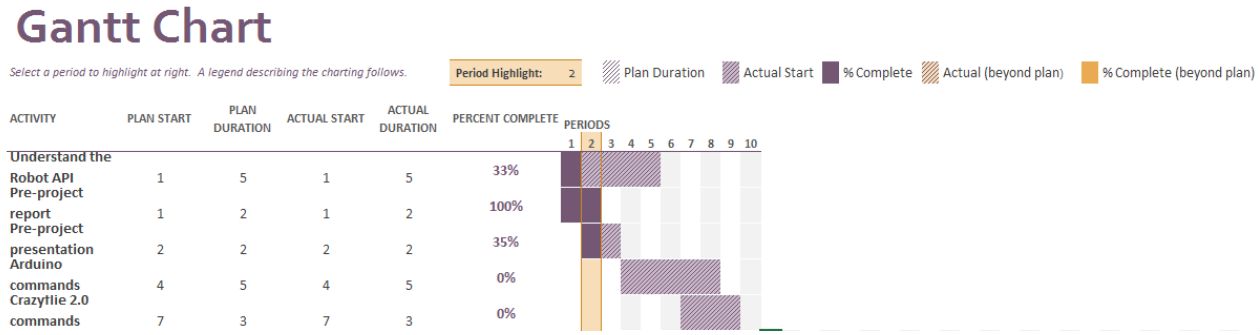
Specific robot configuration file - A xml file in a robot folder that is named differently from the folder. Contains the code which actually sends commands to the components of the robot and values which replaces codewords in the general robot configuration file.

General robot configuration file - A xml file in a robot folder that is named the same as the folder. Contains information for setting up the robot and metamethods. In the methods you can have codewords which can be replaced by the specific configuration file.

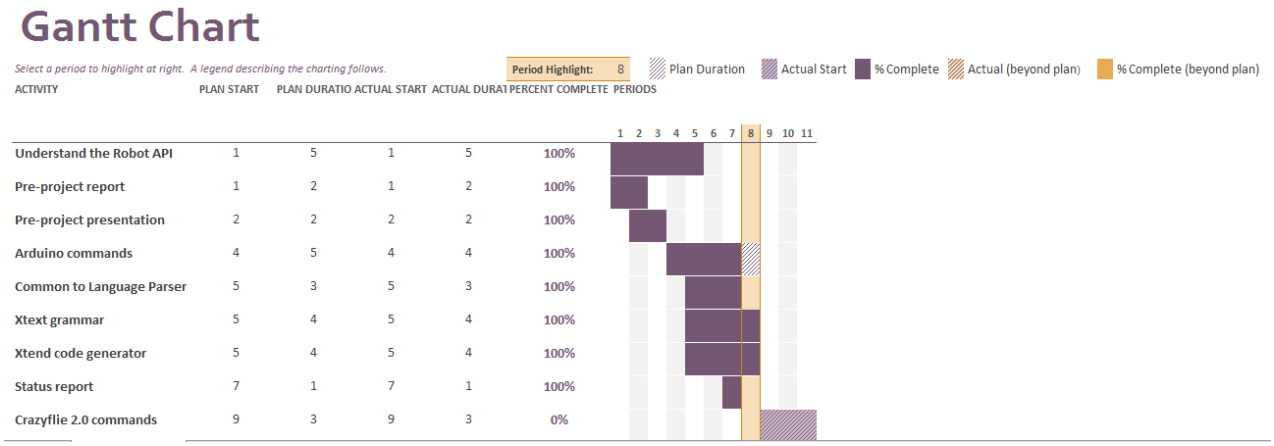
Metamethod - A method declared (but not implemented) in code input for our tool, and implemented in a general robot configuration file. A metamethod represent some task, simple or complex, to be performed by a robot.

Appendix B: Progress plan/Gantt diagram

This was the first version of our gantt diagram



The first gantt diagram was a rough draft. When we made it we weren't quite sure how we wanted to complete the task. After made a plan on what to make for the project, we created a more detailed Gantt Diagram, which had the plans for how to develop the tool, as well as when we wanted to work on the robots. We completed each of the activities, except for adding the Crazyflie 2.0 commands.



Appendix C: Risk list

What:	Likelihood	Impact	How we handle it
Robot destruction	Low	High	Be careful with the robots
Computer trouble	Low	Low-Medium	Using version control, so that any computer trouble doesn't impact the overall project.
Group member get sick	Low	Low	Be ahead of the necessary progress, so that if we get sick we don't fall behind what we need.
Bad communication in the group	Low	Medium	We work in the same location every day, so that it's easy to talk to eachother.
Bad communication with Assignment giver.	Low	Low-High	Regular meetings, and mail exchanging.

Appendix D: User manuals

User documentation (for software development projects)

This software provides an IDE which can be used to generate specific code for the

robots MindStorms EV3 and the arduino shield robot. This is done by writing in our language in the IDE, then running the generated code.

Any commonlang file starts by creating a file with the extension .commonlang. There will be generated a java file which you can run when your script is done.

The .commonlang file needs to have atleast 1 script and exactly 1 metamethodcollection.

The script is initialized by writing

```
script Scriptname targets (robot, robotname){
}
```

for one robot, and

```
script Scriptname targets (robot1,
robotname1),(robot2,robotname2)...{
}
```

for more than one robot. The words written in red are words you can change. Scriptname can be anything you want it to, but has to start with a capital letter. the robot is any folder in the OLTRTA project that contains a xml-file with the same name. This is general configurations which can have small differences in the specific robot.

The specific robot is chosen by the robotname, which should be the name of any other xml-files in the previous mentioned folder.

Before going into what the script can contain, we will go to the metamethodcollection. metamethodcollection is initialized by writing

```
metamethodcollection{

}
```

This part of the script is to define the methods which are on the robots already, which can be found in the “robot\robot.xml” file. Any method which isn’t on the robot will be trimmed away, and any dependent structure removed before the code-file is written. The methods are defined like this

```
meta returntype MethodName();
```

or

```
meta returntype MethodName(parametertype parametername);
```

or

```
meta returntype MethodName(parametertype1
parametername1, parametertype2 parametername2...);
```

Any of the red words with type means that you declare a datatype. for returntype this means the type the method vil give back, “void” if nothing. for parametertype it’s the type the parameter has to be.

Now into what can be in the script. between the curly-brackets on the script you can put in Methods. By convention the main part of a commonlang script is a loop which

is defined as

```
void loop(){ }
```

this is because the arduino shield robot requires this method to run. If you don't want to use the ArduinoShieldRobot you can change this by making a new robotname in the LegoMindStormsEV3 folder, or adding a whole new robot folder(read Operating and maintenance documentation for information on this). Any other method is declared like this

```
returntype methodname(){ }
```

or

```
returntype methodname(parametertype parametername){ }
```

or

```
returntype methodname(parametertype1 parametername1,
parametertype2 parametername2...){ }
```

This is pretty much the same as the metamethod declarations, but you don't put the word meta in front, and the curly brackets after will contain code. This code is a list of expressions. These expressions can be many things, and this is a general list

Simple expressions

Assignment	type varname;
	type varname = value;
	varname +|-|*|/=value;
		Assignments instantiate variables or change the values of existing ones

Call	methodname(parameter1,parameter2...);	Calls trigger methods/metamethods
Increment / decrement variables	varname++; varname--;	Incrementing or decrementing variables in steps of 1
Return	return value;	Returns a value from methods

Block expressions

If	if (booleanvalue) {blockcontent}	Conditional execution of block statements
Else	else expression	Adds else-clause to if-expressions

For	For (expression; booleanvalue; expression) {blockcontent}	Repeats until booleanvalue returns true. First expression called on start, last on each iteration
While	While (booleanvalue) {blockcontent}	Conditional looped execution of block statements

Operating and maintenance documentation

Setup

Firstly, you will have to get Eclipse Luna with Sirius at http://www.eclipse.org/sirius/download.html

When you've completed the installation process, you need to get Xtext and Xtend.

Go to Help>Eclipse MarketPlace and search for "xtext". Press install on Eclipse Xtend 2.11.0 and Eclipse Xtext 2.11.0.

When the installation is complete, you need to download the plugin, library and sample config files from this link: https://goo.gl/cnyLM6

Add place the contents of the plugin folder into the plugins folder of your Eclipse installation.

Launch Eclipse, and create a new Java project.

Add an additional source folder called src-gen.

Add the library to your project's build path.

Each general robot configuration shall have its own package in the src folder. Place the provided sample configurations xml files into packages with the same name as the folder they are provided in.

Now you can create a new commonlang file and start writing code.

For help writing code in the IDE look at the User documentation.

Correctly written code should generate a runnable Java class in the src-gen folder, under the input package. You may need to refresh the folder for the files to appear in your Eclipse package explorer.

Run the generated Java program to generate the robot code. The code will appear in the src-gen folder, in packages named after the general robot configuration.

Setup Arduino Shield Robot code deployment

First of all, you will have to have the arduino IDE. You can get it here: https://www.arduino.cc/en/Main/Software

Now, since we use the pixy cam for object detection, you will have to have the libraries that uses. Follow this guide. This also explains how to get the pixy cam to follow other objects than the one we set it up to use.

http://cmucam.org/projects/cmucam5/wiki/Hooking_up_Pixy_to_a_Microcontroller_(like_an_Arduino)

Now you can just open the code the program generates with the Arduino IDE and press the upload button.

Setup Lego Mindstorms code deployment

Download and install mobaXterm from: http://mobaxterm.mobatek.net/
Now connect the Lego Mindstorms EV3 to your computer through bluetooth, or connect it directly to the wifi.
When the robot is connected it will have an IP address on the top of the screen.

Open up mobaXterm, press the "Start local terminal" and type "ssh robot@" followed by the IP address from the robot. when it asks for a passord type "maker".

Now you can drag and drop the generated code into the filesystem. Use the terminal to find the directory you pushed the code to, and then write "python " and then the filename.

Adding a robot

To add a robot to the system you have to add 3 things. 1 folder, 1 xml with the folder name, and another xml with another name.

The xml file with the foldername is the general robot configuration file. The highest level xml element in this document is the `<robot>` element. this contains `<setup>` and `<metamethods>`.
The `<setup>` contains the information used in the setup of the robot.

`<parsertype>` is as of the end of this project either C or Python.

The `<extension>` is the extension you want on the file, and in our xml files it's either .ino or .py.

`<globalvariables>` has a list of elements which has two attributes: name, which is the name, and type, which is the datatype of the variable. It is important to note that the name attribute isn't the name of the element, but an attribute named name.

`<method>` has a single element called `<setup>`, this tag is defined as any other xml-method, as you can see in the metamethods part.

The `<metamethods>` contains any number of elements. Each element has the method name as its' name. then the parameter attribute is a "," separated string with "datatype parametername" structure. the return attribute contains the data type the method returns. The value of the method-element is lines of code. Don't add the ";" in the code, as it will be added in when printing the file. The code also have to be indented at the level the

method is.

Metamethods should be named the same as in other robots, if the functionality is similar.

Now for the xml file with a name that's different from the folder name. This is the specific robot configuration file. Several specific robot configuration files can use the same general robot configuration file if they are programmed in the same way, and have the same metamethods.

This file also has `<robot>` as the highest element. This element contains the `<methods>` and `<assignments>`.

the `<methods>` is basically the same as `<metamethods>` from the general robot configuration file. It's supposed to use the code to the physical robot parts, while metamethods is supposed to only interact with the `<methods>` methods.

`<assignments>` is a list of words you can use in all of the code, and configuration files, that will be changed into the value attribute's value. This is so that you can set values which will be entity-specific, like pins to sensors or pins to servos.

After you've made these files, all you have to do to generate code is put "`foldername,robotname`" in the targets part of the script.

Adding new functionality

Any functionality on a robot should have a metamethod in the general robot configuration file, and most will also have a method in the specific robot configuration file. As an example I will write how to add Flying functionality.

First you start by making a metamethod in the general robot configuration file, I'll name them `<FlyUp>` and `<FlyDown>`. the attribute "`parameter`" will have a "`int cm`" in both as this is a good variable to have for flying. the "return" attribute will be "`void`". the metamethod will then contain the code for calling the method we will make in the next part: "`flyUp(cm)`" and "`flyDown(cm)`". This might seem redundant, but this is so that if the code for controlling the actual parts is different, this part will look the same.

Now we go into the specific robot configuration file. for making the method, we do the same as with the metamethod, but name it `<flyUp>` and `<flyDown>` instead. Then in the method we put the code, or pseudocode in this example.

```
rotorFL.power(UP)

rotorFR.power(UP)

rotorBL.power(UP)

roterBR.power(UP)

//wait until it has climbed the distance

rotorFL.power(STABLE)

rotorFR.power(STABLE)
```

```
rotorBL.power(STABLE)
roterBR.power(STABLE)
```

And the same, just DOWN instead of UP for flyDown.

Appendix E: Source code

Available at https://github.com/MagnarGya/CommonLanguageForRobots.git

Appendix F: Other relevant material

Demo script

The following code was written for demonstration and testing purposes. It represents a scenario where the Arduino Shield robot attempts to follow and "catch" the Lego robot. Due to a lack of ability to communicate with eachother, however, the chase does not end when the Arduino catches the EV3, rather the Arduino's red LED lights up to signify having completed its assignment, then it restarts its efforts. The EV3 is never informed that it has been catched.

The EV3 also has a loop inside the loop function in this script, in order to make it move in a more interesting pattern (alternating left and right rather than sticking to a single turn direction). This means the default behaviour of stopping the program once the robot is picked up requires a little bit of help to work with this script.

```
script Police targets(ArduinoShieldBot,ShieldBot1){
    void loop(){
        ReadSensors();
        if(FoundObject()){
            if(Touching()){
                LightOn();
                Delay(2000);
                LightOff();
            }else{
                FollowObject();
            }
        }else if (Touching()) {
            if (TouchingLeft() == false) {
                MoveBackward(200);
```

```
                    TurnLeft(90);
               } else if (TouchingRight() == false) {
                    MoveBackward(200);
                    TurnRight(90);
               } else {
                    MoveBackward(500);
                    TurnLeft(90);
               }
          } else if (Seeing()) {
               if (SeeingLeft() == false) {
                    TurnLeft(45);
               } else if (SeeingRight() == false) {
                    TurnRight(45);
               } else {
                    TurnLeft(90);
               }
          } else {
               MoveForward(200);
          }
     }
}

script Robber targets (LegoMindstormsEV3,EV3_1) {
     void loop() {
          int stage = 0;
          while (stage < 2) {
               ReadSensors();
               if(Touching()) {
                    MoveBackward(500);
                    if (stage == 0) {
                         TurnRight(60);
```

```
                    stage++;
                } else {
                    TurnLeft(60);
                    stage++;
                }
            } else if (Seeing()) {
                if (stage == 0) {
                    TurnRight(60);
                    stage++;
                } else {
                    TurnLeft(60);
                    stage++;
                }
            } else {
                MoveForward(100);
            }
        }
    }
}

metamethodscollection {
    meta void ReadSensors();
    meta boolean Touching();
    meta boolean TouchingLeft();
    meta boolean TouchingRight();
    meta boolean Seeing();
    meta boolean SeeingLeft();
    meta boolean SeeingRight();
    meta void MoveForward(int time);
    meta void MoveBackward(int time);
    meta void TurnRight(int time);
```

```
    meta void TurnLeft(int time);
    meta void LightOn();
    meta void LightOff();
    meta boolean FoundObject();
    meta void FollowObject();
    meta void Delay(int time);
}
```