CommonLang: a DSL for defining robot tasks

Adrian Rutle[1], Jonas Backer[1], Kolbein Foldøy[1], and Robin T. Bye[2]

[1] Western Norway University of Applied Sciences, Bergen, Norway
{aru,149906,149909}@hvl.no
[2] Cyber-Physical Systems Laboratory,
NTNU—Norwegian University of Science and Technology, Ålesund, Norway
robin.t.bye@ntnu.no

Abstract. Robots are becoming more and more complex and heterogeneous; their abilities and domains of usage are increasing exponentially. Programming these robots requires special skills and usually does not follow standard software engineering methodologies. Adhering to model-driven software engineering principles, definition of robot behaviour is abstracted and represented in models while robot-specific code is generated from these models using code generation. With a robot modelling framework, we can work on a higher abstraction level making the task of programming complex heterogeneous robots more efficient. In this paper, we present such a modelling framework and evaluate its flexibility by extending it with wireless communication functionalities.

Keywords: Robot programming · Robot modelling framework · Model-driven software engineering · Robot communication protocol.

1 Introduction

Robots come in a variety of sizes and shapes and with different purposes and abilities. In today's society robots have revolutionized the efficiency and productivity in many fields. The use of robots in the field of manufacturing cars has increased the capacity and quality of the cars, in addition to protecting workers from performing extremely dangerous tasks. In agriculture, drones are being used to analyze soil and plant seeds and choosing the best moment to harvest. Whether robots are used for bomb disposal or vacuum cleaning, their operation depends on both their hardware and the software they are running [10].

Current robot programming frameworks require both a steep learning curve and specific hardware and software environments. A large number of robotic software exist but interoperation across robots is difficult, with dependencies on specific hardware or software platforms that are hard-wired into the robot code [6,7]. Addressing recent challenges in robotics [21] has led to various conference and workshop series focusing on robotics software and design [19]. Specifically, using model-driven software engineering (MDSE) methodologies and technologies to raise the abstraction level of robotic software development, enable simulation and verification, and tackle complexity challenges, are common research goals at events such as MORSE, SIMPAR, ICRA, ROSE, etc.

In this paper, we present a prototype framework *CommonLang* (CoL) which uses MDSE-techniques [2,3] to abstract away from underlying technologies and create executable code for various different robot platforms using code generation [9]. The framework comes with a text-based domain-specific language (DSL) that enables users to write scripts through abstracted *metamethods*; i.e., function abstractions providing a higher-level language.

Given a number of heterogeneous robots, each with its own programming language, the main feature of CoL is its ability to parse a common set of robot instructions into multiple tailor-made scripts to yield identical behaviour for the robots. The layout of the output scripts is dependent on XML configuration files that contain the implementation of the metamethods and specify what language they should be parsed to (e.g., C, Python, Java, etc.). In this paper, we demonstrate behavioural programming of two robots that use C/C++ and Python, respectively, as well as providing an evaluation of CoL's ability to avoid increased complexity—keeping usability and ease of access at an acceptable level—when adding WiFi communication functionality by utilizing metamethods.

2 CommonLang

CommonLang (CoL)—which was first described in [9]—is a DSL created specifically for writing code for a set of various robots using different programming languages. It is an imperative and structured language, and in many ways similar to Java, which is one of the languages CoL can be parsed to. CoL includes various control flow constructs found in most other programming languages, e.g., `for`, `if`, and `while` statements.

The aim of CoL is to enable reusable behavior for different robots with similar abilities. To accomplish this goal, a textual DSL was defined using Xtext in Eclipse [20]. The solution builds on the concept of metamethods that declare the common abilities of target systems for a behavioral script. Fig. 1 illustrates how the functionality of three different robot platforms can be used to create a

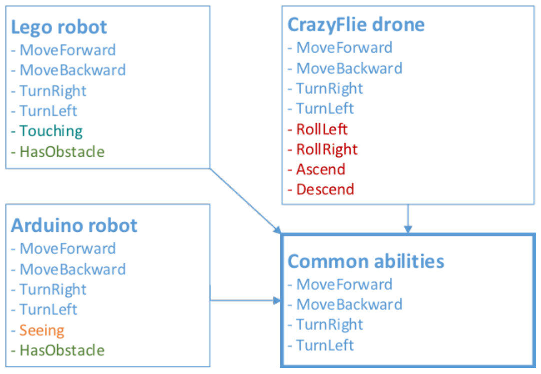


Fig. 1. Example of common abilities in different robot platforms.

common set of abilities, consisting of the metamethods that can be used in scripts targeted to these platforms. These commonalities are not enforced, meaning that robots which do not naturally exhibit common functionalities will not be sharing behavior. However, in the CoL script it is possible to use metamethods that are not present in all target platforms. The invocation of these will be ignored on unsupported platforms. In this way, a script can define tasks for a heterogeneous environment of interacting robots.

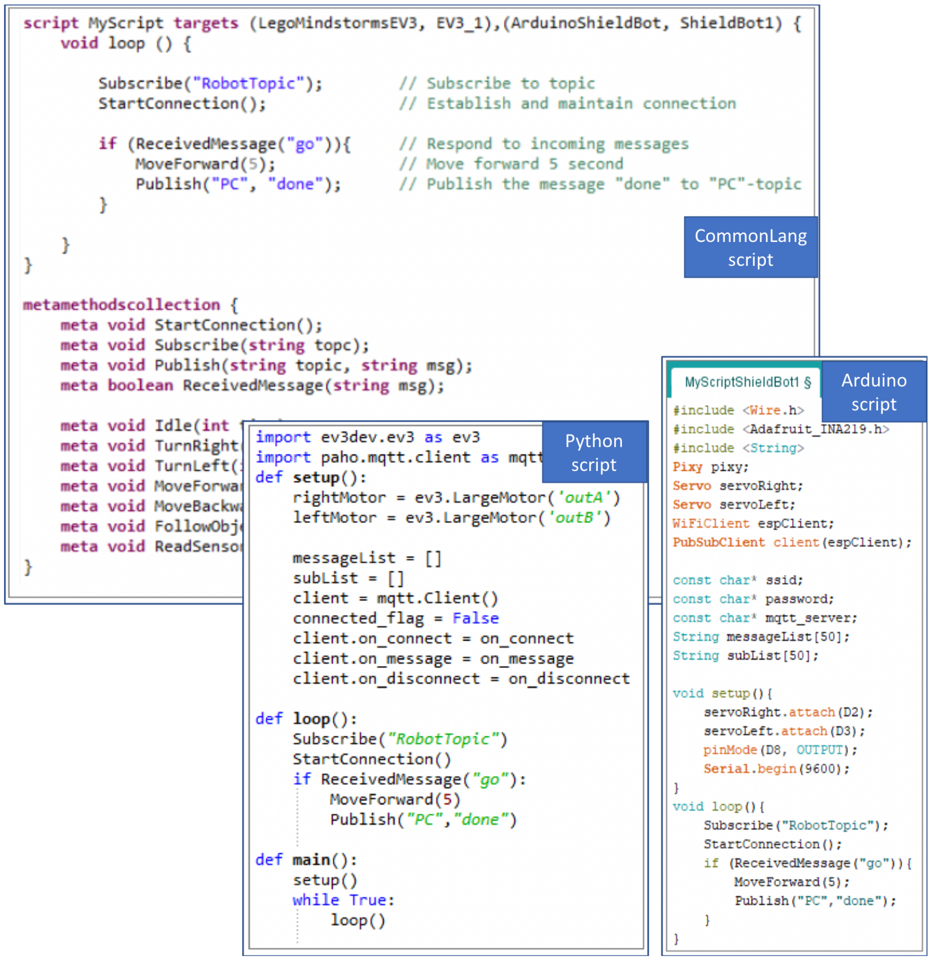
2.1 Defining behavior with CoL scripts

Defining behavior in CoL consists of two steps: writing a CoL script and listing the collection of metamethods. The script is declared by the script keyword, the script name, then the target keyword, and finally the platforms to be targeted (see Fig. 2). The script is written in a file with the extension name `.commonlang` and consists of two main blocks. The first part contains the `script` definition, name of the script (`MyScript`), and a set of what robots the script will be generated for (`LegoMindstormsEV3` and `ArduinoShieldBot`), in the form of configuration file names for each robot. Target platforms must include a specific configuration for each platform as many robot platforms allow the hardware to be configured differently as needed. Scripts may consist of variables, metamethod invocations, and some common control statements such as `if`, `else`, and `while`. The program code will be written inside a method called `loop` contained in the script block, and consists of user methods and metamethods. The output code, here parsed to Arduino (C/C++) and Python, will continuously iterate this method, hence it does not allow for storing states, like integers and booleans.

The second block is called `metamethodcollection`, which is a set of predefined methods that every robot with a different programming language is able to run and has its own implementation of. Each metamethod declaration consists of the reserved keyword `meta`, followed by a return type, name and its corresponding parameters. For example, the metamethod `MoveForward` may for some robots make two of the wheels spin forward, whereas for other robots may make four wheels spin at different speeds. Some robots may even be flying, meaning they will have rotors spinning in different directions to be able to fly forward.

For CoL to generate code for a robot's native language, we need to declare two XML files (see Fig. 3). The first XML file is generic for all robots of its kind and contains various types of data, such as the file format for the language files (e.g., `.py` or `.c` indicating Python or C files, respectively), global variables, declarations and method calls for the setup method of the final script. It also contains which metamethods the robot can execute. In this XML file, the metamethods have a simple code segment, either calling its corresponding method from the other XML file where it is implemented, or returning a value from global variables or forwarding the return value from the method it is calling [9].

The other XML file is specific for the robot configuration and contains the implementation of the methods in the robots' own language, for instance, setting a motor to run for the duration specified in the parameters or reading the value of a digital pin and returning it. These methods with their implementations are



```
script MyScript targets (LegoMindstormsEV3, EV3_1),(ArduinoShieldBot, ShieldBot1) {
    void loop () {

        Subscribe("RobotTopic");          // Subscribe to topic
        StartConnection();                // Establish and maintain connection

        if (ReceivedMessage("go")){       // Respond to incoming messages
            MoveForward(5);               // Move forward 5 second
            Publish("PC", "done");        // Publish the message "done" to "PC"-topic
        }

    }
}

metamethodscollection {
    meta void StartConnection();
    meta void Subscribe(string topc);
    meta void Publish(string topic, string msg);
    meta boolean ReceivedMessage(string msg);

    meta void Idle(int t
    meta void TurnRight
    meta void TurnLeft(
    meta void MoveForwa
    meta void MoveBackw
    meta void FollowObj
    meta void ReadSenso
}
```

```
import ev3dev.ev3 as ev3
import paho.mqtt.client as mqtt
def setup():
    rightMotor = ev3.LargeMotor('outA')
    leftMotor = ev3.LargeMotor('outB')

    messageList = []
    subList = []
    client = mqtt.Client()
    connected_flag = False
    client.on_connect = on_connect
    client.on_message = on_message
    client.on_disconnect = on_disconnect

def loop():
    Subscribe("RobotTopic")
    StartConnection()
    if ReceivedMessage("go"):
        MoveForward(5)
        Publish("PC","done")

def main():
    setup()
    while True:
        loop()
```

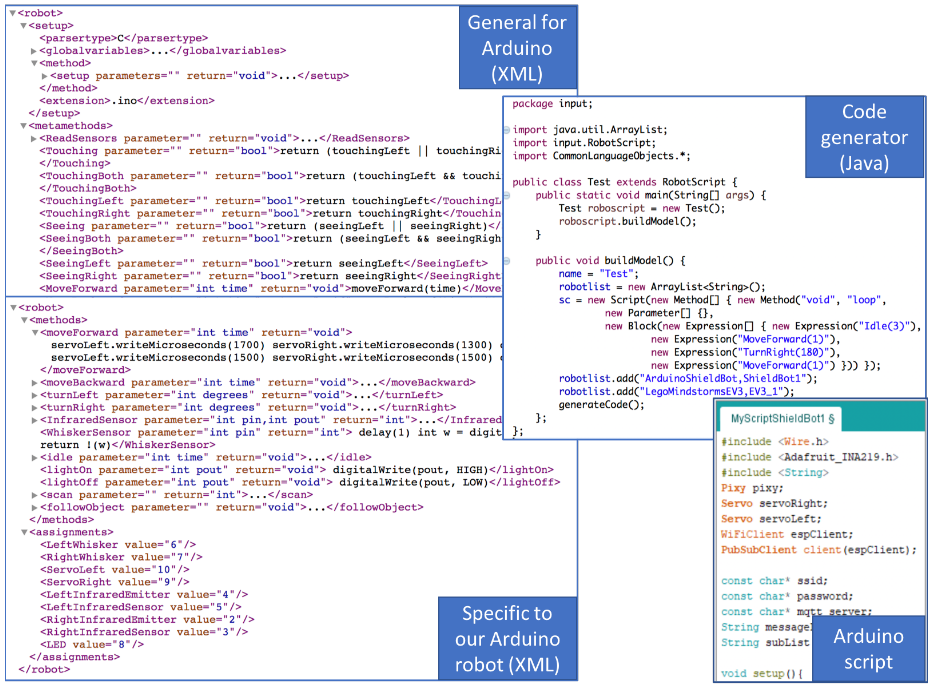
```
#include <Wire.h>
#include <Adafruit_INA219.h>
#include <String>
Pixy pixy;
Servo servoRight;
Servo servoLeft;
WiFiClient espClient;
PubSubClient client(espClient);

const char* ssid;
const char* password;
const char* mqtt_server;
String messageList[50];
String subList[50];

void setup(){
    servoRight.attach(D2);
    servoLeft.attach(D3);
    pinMode(D8, OUTPUT);
    Serial.begin(9600);
}
void loop(){
    Subscribe("RobotTopic");
    StartConnection();
    if (ReceivedMessage("go")){
        MoveForward(5);
        Publish("PC","done");
    }
}
```

Fig. 2. Sample CommonLang program with code-generation to Python and C

then automatically added without changes to the final output script which is already in the target language. Additionally, this XML file contains a tag called `assignments` for setting up robot sensors to the right pin values, since this configuration is robot-dependent.

2.2 Code generation from CoL scripts

As mentioned, the CoL grammar is defined in Xtext [20]. The details of Xtext and how it works is out of the scope of this paper, but in short, Xtext facilitates the definition of DSLs using a powerful grammar language and as a result we can get a full DSL-infrastructure, including parser, linker, typechecker, compiler as well as a language editor with syntax-highlighting in Eclipse. CoL relies on code-generation functionalities provided by Xtext. These functionalities are defined as templates in the language Xtend—a flexible and expressive dialect of Java


General for Arduino (XML)
Code generator (Java)
Specific to our Arduino robot (XML)
Arduino script
MyScriptShieldBot1 §

```xml
<robot>
  <setup>
    <parsertype>C</parsertype>
    <globalvariables>...</globalvariables>
    <method>
      <setup parameters="" return="void">...</setup>
    </method>
    <extension>.ino</extension>
  </setup>
  <metamethods>
    <ReadSensors parameter="" return="void">...</ReadSensors>
    <Touching parameter="" return="bool">return (touchingLeft || touchingRi
    </Touching>
    <TouchingBoth parameter="" return="bool">return (touchingLeft && touchi
    </TouchingBoth>
    <TouchingLeft parameter="" return="bool">return touchingLeft</TouchingL
    <TouchingRight parameter="" return="bool">return touchingRight</Touchin
    <Seeing parameter="" return="bool">return (seeingLeft || seeingRight)</
    <SeeingBoth parameter="" return="bool">return (seeingLeft && seeingRigh
    </SeeingBoth>
    <SeeingLeft parameter="" return="bool">return seeingLeft</SeeingLeft>
    <SeeingRight parameter="" return="bool">return seeingRight</SeeingRight
    <MoveForward parameter="int time" return="void">moveForward(time)</Move
```

```xml
<robot>
  <methods>
    <moveForward parameter="int time" return="void">
      servoLeft.writeMicroseconds(1700) servoRight.writeMicroseconds(1300) 
      servoLeft.writeMicroseconds(1500) servoRight.writeMicroseconds(1500) 
    </moveForward>
    <moveBackward parameter="int time" return="void">...</moveBackward>
    <turnLeft parameter="int degrees" return="void">...</turnLeft>
    <turnRight parameter="int degrees" return="void">...</turnRight>
    <InfraredSensor parameter="int pin,int pout" return="int">...</Infrared
    <WhiskerSensor parameter="int pin" return="int"> delay(1) int w = digit
    return !(w)</WhiskerSensor>
    <idle parameter="int time" return="void">...</idle>
    <lightOn parameter="int pout" return="void"> digitalWrite(pout, HIGH)</lightOn>
    <lightOff parameter="int pout" return="void"> digitalWrite(pout, LOW)</lightOff>
    <scan parameter="" return="int">...</scan>
    <followObject parameter="" return="void">...</followObject>
  </methods>
  <assignments>
    <LeftWhisker value="6"/>
    <RightWhisker value="7"/>
    <ServoLeft value="10"/>
    <ServoRight value="9"/>
    <LeftInfraredEmitter value="4"/>
    <LeftInfraredSensor value="5"/>
    <RightInfraredEmitter value="2"/>
    <RightInfraredSensor value="3"/>
    <LED value="8"/>
  </assignments>
</robot>
```

```java
package input;

import java.util.ArrayList;
import input.RobotScript;
import CommonLanguageObjects.*;

public class Test extends RobotScript {
    public static void main(String[] args) {
        Test roboscript = new Test();
        roboscript.buildModel();
    }

    public void buildModel() {
        name = "Test";
        robotlist = new ArrayList<String>();
        sc = new Script(new Method[] { new Method("void", "loop",
                new Parameter[] {},
                new Block(new Expression[] { new Expression("Idle(3)"),
                        new Expression("MoveForward(1)"),
                        new Expression("TurnRight(180)"),
                        new Expression("MoveForward(1)") })) });
        robotlist.add("ArduinoShieldBot,ShieldBot1");
        robotlist.add("LegoMindstormsEV3,EV3_1");
        generateCode();
    };
};
```

```c
#include <Wire.h>
#include <Adafruit_INA219.h>
#include <String>
Pixy pixy;
Servo servoRight;
Servo servoLeft;
WiFiClient espClient;
PubSubClient client(espClient);

const char* ssid;
const char* password;
const char* mqtt_server;
String message
String subList

void setup(){
```

Fig. 3. XML configuration files

which compiles into readable Java 8 compatible source code. To understand how this code-generation step works, we include an excerpt of the CoL grammar in Listing 1.1. Xtend creates a Java class for each language construct in the grammar, e.g. `Script.java`, `MetaMethods.java`, `Block.java`, etc. In addition to these classes, we have implemented a class `BotMethods.java` which parses the two XML files for each robot in the `targets` block. Following the Xtext methodology, we have also implemented a customized code-generator class for each target language; in this case one for `Python` and one for `C`.

Once a valid CoL script (according to the grammar) is saved, all these classes gets instantiated automatically by the Xtext language-infrastructure. In addition, a code generator (which is a simple Java program as shown in Fig. 3) will be generated automatically for each CoL script. Running this Java program will create the final robot scripts for the target platforms.

Listing 1.1. Grammar of CommonLang

```
grammar org.xtext.Commonlang with org.eclipse.xtext.common.Terminals

generate commonlang "http://www.xtext.org/Commonlang"

CLfile:
  scripts+=(Script)*
  mets=MetaMethods;
Script:
  'script' name=CAPITALFIRST 'targets' '(' robottypes+=(LOWERFIRST |
      CAPITALFIRST) ',' robotconfigs+=(LOWERFIRST |
    CAPITALFIRST) ')' (',' '(' robottypes+=(LOWERFIRST | CAPITALFIRST) ','
         robotconfigs+=(LOWERFIRST | CAPITALFIRST)
    ')')* '{'(methods+=UserMethod*)'}';
  MetaMethods:
    {MetaMethods} 'metamethodscollection' '{'(methods+=MetaMethod)*'}';
  Block:
    {Block} '{' ((exs+=SimpleExpression ';' | exs+=StructureExpression))*
         '}';
  SimpleExpression:
    Crement | Call | Assignment | Return;
  StructureExpression:
    Block | If | For | While;
  Expression:
    SimpleExpression | StructureExpression;
//...
  Method:
    (UserMethod | MetaMethod);
  MetaMethod:
    'meta' type=Methodtype name=CAPITALFIRST '(' parameters+=Declaration?
         (','+ parameters+=Declaration)* ')' ';';
  UserMethod:
    type=Methodtype name=LOWERFIRST '(' parameters+=Declaration? (','+
         parameters+=Declaration)* ')' bl=Block;
//...
```

Using the standard Xtext methodology, code-generation is divided into two steps. Firstly, an object model will be created for the parsed CoL script, then, target scripts in the native languages of the robots will be created using the customized code-generators. The complexity of these code generation steps depends on the complexity of the CoL script and the number of robot kinds which are defined in the `targets` block. These steps are hidden from the CoL script and metamethod developers.

Extending CoL with support for generating code to new target platforms require some knowledge of Java and Xtext. This consists of implementing a Java class for each new target language (however, C and Python covered all the robots which we have had in hand, e.g. Arduino robot, Lego Mindstorm, CrazyFlie mini-drones and Land-rovers with Raspberry PI 3). In the envisioned scenario, this activity is hidden from CoL script developers and delegated to DSL-infrastructure experts.

Configuring and developing metamethods are done in the XML files. Our optimal goal is to ship the DSL with a number of most-used metamethods for various commonly available robots. However, we see this activity as a community effort and envisage that the future of the DSL will be depending on the availability of this infrastructure. We have currently not defined any scheme or convention for the definition of these metamethods due to their simplicity, however, in future work we plan to define a guideline and a metamodel in form of an XML Schema Definition (XSD) for the configuration XML files.

3 Extension with Wireless Communication

In order to evaluate the DSL, we have extended its functionality by adding features for WiFi communication using Message Queuing Telemetry Transport (MQTT) [11] as a sample communication protocol. This functionality is not hard-coded and can be replaced by other protocols if needed.

MQTT is a lightweight messaging protocol that allows the robots to subscribe to topics. They can also publish messages to the topics, so that clients subscribed to a particular topic will receive the messages. A broker is required for this setup to work. It keeps track of available topics and clients connected to the server, forwarding incoming messages to all the clients subscribing the respective topics. For a client to subscribe to a topic it needs only to know the IP address of the broker. We have used the Eclipse Mosquitto [16] implementation of MQTT, which grants the ability to set up a broker and enter commands such as subscribing and publishing messages to desired topics into the command prompt window.

An important decision had to be made regarding whether the communication should be integrated as part of the CoL grammar or as additional metamethods through the XML files. Editing the grammar of the language results in fewer unnecessary lines of code in the XML files. However, modifying the CoL grammar is more complex than making metamethods for it: the DSL is designed for handling metamethods specified in the robots' XML files. It creates an environment for general purpose robot programming which could be compromised by adding grammar specifically for MQTT communication since with further development of the language other communication protocols may be desirable. By refraining from doing grammatical adjustments, it becomes quite manageable to add or edit already existing metamethods for communication.

A crucial part of integrating communication through metamethods is making the code intuitive and user-friendly, and at the same time not limiting what the end user is able to do. Various approaches to how the syntax and methods should look like have been considered. One of the approaches was to let the end user create a method surrounding a selection of metamethods. This user method would then be connected to a message string through a metamethod called `OnMessage`. This is how it would look like:

Listing 1.2. OnMessage metamethod

```
OnMessage("go", forwardAndLightsOn);

void forwardAndLightsOn() {
 MoveForward();
 LightsOn();
}
```

This approach would result in more code to write. It would also require a change in the language's grammar, allowing it to take a method as a parameter. After considering variations of similar syntax, we settled on utilizing the simplicity of metamethods combined with `if` statements, which are already part of the Commonlang grammar. The metamethod is called `ReceiveMessage` and takes a single string parameter, returning a boolean value. Here is an example:

Listing 1.3. ReceiveMessage metamethod

```
if (ReceivedMessage("go")) {
 MoveForward();
 LightsOn();
}
```

The method is checking whether the robot has received a specific message. When the robots receive messages, the method triggers a callback called `on_message` that will put the messages in a list. This is because the callback method is hardcoded in the XML files and cannot be dynamically changed. Instead, this `ReceivedMessage` method was created to scan through the message list to check for the string from its parameter. If it finds it, a single instance of the message will be removed from the list and it will return true; otherwise false.

4 Related Work

Robots can be programmed in different ways: from writing robot-specific code directly to using frameworks with varying functionality. Some of these frameworks, such as Papyrus-RT [17] and RobotML [6], utilize MDSE techniques [2,3] to deal with software complexity. Others, such as the Robot Operating System (ROS) [18] and RoboDK [13], use programming frameworks which are tailored for specific kinds of robots. Below we shortly introduce these frameworks, keeping in mind that the main distinctive characteristic of CoL is its support for writing behavior for a heterogeneous environment of various kinds of robots.

ROS is an open source project consisting of a collection of tools, libraries and conventions that aims to simplify programming robust and complex behaviour of robots of all shapes and sizes across a wide variety of platforms, whereas Papyrus-RT is another framework for real-time programming of robots which generates C++ code from models defined in the UML-RT modeling language. RobotML is a DSL that aims at solving several central problems with robot programming today [6] such as costs and difficulty of development and reusability issues because of low level details being hard-wired into the code at early stages in the development. Interaction modelling [5], Deep modeling [1], and Multi-robot systems [12] also use DSLs for modeling robotic software but they do not have the same maturity as RobotML and Papyrus-RT. The robot instructions written in these languages can be simulated and visualized in 3D as well as exported to the native languages of each of the robots. Unfortunately, RoboDK currently only supports industrial robots.

5 Conclusion and Future Work

We have presented CoL, a DSL for robot task specification which enables compilation of tasks to heterogeneous robot platforms. With CoL, one can do the time-consuming work of writing functions and setup once, then use these functions to define tasks for robot teams. The functions are developed as metamethods which extend what the users can express with CoL scripts. Since one could make any kind of metamethods, the functionality can vary greatly from a simpler move-forward function to highly abstract functions such as "water field" or "defuse bomb." The defined functionality is contained within XML configuration files that are included in CoL scripts. This means one could make these robot-specific files beforehand and make documentation for them so that the CoL scripts can be written by less technical domain-experts to define heterogeneous robot behavior in various domains. These domains could be anything as long as the tasks in the CoL script can be achieved by robots for which the appropriate metamethods are implemented. The usability and extendability of CoL was demonstrated by adding support for wireless communication. The work done to add communications functionality to CoL was implemented through configuration files and metamethods; i.e., the DSL itself is not modified. This proved the potentials to add functionality to CoL with a library-like approach.

Future work would includes (i) *expanding the available library of metamethods* to offer greater flexibility in how the user would like to use the DSL, possibly dividing the metamethods into different collections with solid documentation. This includes also guidelines and conventions for developing and documenting metamethods. Testing and calibrating the robots was time consuming, connecting via USB or SSH; transferring the script and then running the script when the robot was in a semi-suitable location was not ideal. (ii) *Implementing a simulator* could cut down on testing time, as well as being a solid tool for evaluation. Moreover, (iii) *implementing a modeling and reasoning functionality* on top of CoL to describe intended behavior of a team of robots—in the direction of [15], which uses hierarchies of finite state machines to structure the behavior of the team—would make CoL scripts more robust and allow for formal reasoning about the collective behavior of the robot team. Features for dynamically assigning roles to robots (based on their capabilities) and robots to roles (based on the situation) would be included in such an extension which should also consider planning and verification of achievement of the desired goals of a mission. Finally, retrieving a concrete execution plan from a general and abstract goal or mission description and verifying that the robots would act accordingly to reach the goal would be a valuable addition to CoL. As also described in [7], (iv) *implementing self-adaptation and dynamic recalculation of plans* for missions which are defined as CoL scripts would increase its usefulness and robustness to unforeseen situations at runtime. To make CoL more accessible for laypersons, (v) a *graphical syntax* will be defined for the language using guidelines from [4] and Sirius [14]. This step should be possible due to the compatibility between Xtext and Eclipse Modeling Framework (EMF) [8].

References

1. Atkinson, C., Gerbig, R., Markert, K., Zrianina, M., Egurnov, A., Kajzar, F.: Towards a deep, domain specific modeling framework for robot applications. In: Assmann, U., Wagner, G. (eds.) MORSE. pp. 1–12. No. 1319 in CEUR Workshop Proc. (2014), http://ceur-ws.org/Vol-1319/#morse14_paper_01
2. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, Second Edition. SLSE, Morgan & Claypool Publishers (2017)
3. Brugali, D.: Model-driven software engineering in robotics. IEEE Robotics & Automation Magazine **22**(3), 155–166 (2015)
4. Cho, H., Gray, J., Syriani, E.: Syntax map: A modeling language for capturing requirements of graphical dsml. In: 2012 19th Asia-Pacific Software Engineering Conference. vol. 1, pp. 705–708 (Dec 2012). https://doi.org/10.1109/APSEC.2012.20

5. Cornelius, G., Hochgeschwender, N., Voos, H.: Model-driven interaction design for social robots. In: Seidl, M., Zschaler, S. (eds.) Software Technologies: Applications and Foundations - STAF. LNCS, vol. 10748, pp. 219–224. Springer (2017)
6. Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., Ziane, M.: Robotml, a domain-specific language to design, simulate and deploy robotic applications. In: Noda, I., Ando, N., Brugali, D., Kuffner, J.J. (eds.) SIMPAR. pp. 149–160. Springer Berlin Heidelberg (2012)
7. Dragule, S., Meyers, B., Pelliccione, P.: A generated property specification language for resilient multirobot missions. In: Romanovsky, A., Troubitsyna, E.A. (eds.) Software Engineering for Resilient Systems. pp. 45–61. Springer International Publishing, Cham (2017)
8. Eclipse Modeling Framework (EMF): Web site. http://www.eclipse.org/modeling/emf (2018), accessed on 2018-08-20
9. Gya, M., Solhaug, T.: Common programming interface for multiple types of robots (2017), bachelor thesis, Western Norway University of Applied Sciences (available at http://ict.hvl.no/commonlang-a-dsml-for-robot-task-definition/)
10. Mazur, M., Wiśniewski, A., McMillan, J.: Clarity from above: PwC global report on the commercial applications of drone technology (May 2016)
11. MQTT.org: Machine queuing telemetry transport protocol. http://mqtt.org (2018), accessed on 2018-08-20
12. Ruscio, D.D., Malavolta, I., Pelliccione, P.: A family of domain-specific languages for specifying civilian missions of multi-robot systems. In: Assmann, U., Wagner, G. (eds.) MORSE. pp. 13–26. No. 1319 in CEUR Workshop Proc. (2014), http://ceur-ws.org/Vol-1319/#morse14_paper_02
13. Simulation and OLP for Robots: RoboDK. https://robodk.com/ (2018), accessed on 2018-07-13
14. Sirius: Web site. http://www.eclipse.org/sirius/ (2018), accessed on 2018-08-20
15. Skubch, H.: Modelling and controlling of behaviour for autonomous mobile robots. Ph.D. thesis, University of Kassel (2013), http://d-nb.info/1026180120
16. The Eclipse Foundation: Eclipse mosquitto. https://projects.eclipse.org/projects/technology.mosquitto (2018), accessed on 2018-08-20
17. The Eclipse Foundation: PapyrusRT. https://www.eclipse.org/papyrus-rt/ (2018), accessed on 2018-07-13
18. The Robot Operating System (ROS): Web site. http://www.ros.org (2018), accessed on 2018-08-20
19. The Robotics Summit & Expo: Design and development track. https://www.roboticssummit.com/tracks/#design (2018), accessed on 2018-07-13
20. Xtext: Web site. https://www.eclipse.org/Xtext/ (2018), accessed on 2018-08-20
21. Yang, G.Z., Bellingham, J., E. Dupont, P., Fischer, P., Floridi, L., Full, R., Jacobstein, N., Kumar, V., McNutt, M., Merrifield, R., Nelson, B., Scassellati, B., Taddeo, M., Taylor, R., Veloso, M., Lin Wang, Z., Wood, R.: The grand challenges of science robotics. Science Robotics **3**, eaar7650 (2018)