IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 13, 2020, accepted February 2, 2020, date of publication February 7, 2020, date of current version February 18, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2972410

LearnBlock: A Robot-Agnostic Educational Programming Tool

PILAR BACHILLER-BURGOS[1], IVÁN BARBECHO[2], LUIS V. CALDERITA[1], PABLO BUSTOS[1], AND LUIS J. MANSO[3]

[1]Robotics and Artificial Vision Laboratory (RoboLab), Department of Computer and Communication Technology, University of Extremadura, 10003 Cáceres, Spain
[2]Research and Development Department, Mobbeel Solutions, SL., 10195 Cáceres, Spain
[3]School of Engineering and Applied Science, Aston University, Birmingham B4 7ET, U.K.

Corresponding author: Pilar Bachiller-Burgos (pilarb@unex.es)

This work was supported in part by the Spanish Government under Grant RTI2018-099522-B-C42, in part by the Government of Extremadura under Grant IB16090 and Grant GR18133, in part by the Institute of Coding, and in part by the U.K. National Initiative supported by the government via a £20-million grant from the Office for Students.

ABSTRACT Education is evolving to prepare students for the current sociotechnical changes. An increasing effort to introduce programming and other STEM-related subjects into the core curriculum of primary and secondary education is taking place around the world. The use of robots stands out among STEM initiatives, since robots are proving to be an engaging tool for learning programming and other STEM-related contents. Block-based programming is the option chosen for most educational robotic platforms. However, many robotics kits include their own software tools, as well as their own set of programming blocks. LearnBlock, a new educational programming tool, is proposed here. Its major novelty is its loosely coupled software architecture which makes it, to the best of our knowledge, the first robot-agnostic educational tool. Robot-agnosticism is provided not only in block code, but also in generated code, unifying the translation from blocks to the final programming language. The set of blocks can be easily extended implementing additional Python functions, without modifying the core code of the tool. Moreover, LearnBlock provides an integrated educational programming environment that facilitates a progressive transition from a visual to a general-purpose programming language. To evaluate LearnBlock and demonstrate that it is platform-agnostic, several tests were conducted. Each of them consists of a program implementing a robot behaviour. The block code of each test can run on several educational robots without changes.

INDEX TERMS Educational tool, learning programming, robot-agnostic, software architecture.

I. INTRODUCTION

In our highly technological world, Computational Thinking (CT) is becoming a valuable skill. The term was coined in [1] and popularised by [2] where it was defined as: "*the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer -human or machine- can effectively carry out*". However, there is still a lack of consensus on the definition and dimension of the term [3]. In fact, the authors propose a working definition of Computational Thinking: "*The conceptual foundation required to solve problems effectively and efficiently (i.e., algorithmically, with or without the assistance of computers) with solutions that are reusable in different contexts. This definition highlights that CT is primarily a way of thinking and acting*" [3].

Knowing how to communicate with different devices through own and specific languages (programming languages), is becoming a crucial aspect for full personal development. Consequently, the current importance of encouraging CT could be equated with the learning of reading, writing or arithmetic in the last century [4]. According to [5]: "*Computational Thinking is essential to the development of computer applications, but it can also be used to support problem-solving across all disciplines, including math, science, and the humanities. Students who learn Computational Thinking across the syllabus can begin to see a relationship between subjects as well as between school and life outside the classroom.*"

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Luca Bernardi.

Given the importance of these ideas, efforts to introduce programming into the core curriculum have markedly increased. Nowadays, many European countries are including contents related to programming in curricula at pre-university stages. Clear examples of this trend can be found in [6]. Institutional support in this direction has also been strengthened. An example of this is the EU-funded TACCLE 3 project [7]. Its objective is to support the inclusion of learning programming in basic education, specifically in the age range from 4 to 14 years. For this purpose, TACCLE 3 provides practical ideas, materials and resources necessary to introduce computing or coding in the classroom [8]. Non-profit organisations, such as *Code.org* [9] or *Khan Academy* [10], have also emerged to expand computer science learning, as well as to encourage students of basic education to initiate in computer programming. *Code.org* also expects that computer programming will become a part of the core curriculum in education in the near future [5].

In addition to CT, programming electronic devices and robots will also be useful skills in the future. According to recent studies [11], [12], it is estimated that in 25 years a significant percentage of the current works could be performed by robots. More specifically, it is expected that in a decade, robots and/or automatisms will do the work of about 800 million employees [13]. Consequently, the incorporation of robotics into pre-college education system seems necessary and crucial. This is especially important considering that the 2030 labour market will mostly require profiles related to electronic device programming, system automation and robotics [14].

Fortunately, efforts in this direction have already begun. The incorporation of robotics in education to introduce STEM-related subjects is noticeably increasing [15], [16]. Likewise, using robots as teaching instruments is becoming more and more frequent [17]. In addition, a positive transverse effect is achieved on the student's performance in the rest of the subjects. This improvement is the result of applying the reasoning, logic and abstraction learned from programming to other subjects [18].

In this context, this paper presents an educational tool, called LearnBlock, for learning programming and CT-related contents through robotics. The main property of LearnBlock is that its design is robot-agnostic, but, additionally, it incorporates further important features not provided by other educational programming tools.

The rest of the paper is organised as follows. Section II discuses the most significant related work and describes the main motivation of our project. Section III presents LearnBlock, providing an overview of the different features of the tool. Section IV shows the software architecture of LearnBlock including details on all the modules composing it. In section V different tests probing the robot-agnostic property of LearnBlock are presented. Section VI summarises the main conclusions and future directions of our work.

II. RELATED WORK AND MOTIVATION

One of the most promising ways to learn programming is ''block-based programming''. Widely used educational tools, such as Alice [19] or Scratch [20], provide graphical environments for creating programs using blocks. Moreover, in the last few years, Blockly [21] has emerged as a tool that allows developers to create their own ''block-based'' languages. All these educational tools provide interesting features for learning coding: they alleviate the syntax-related difficulties associated with traditional programming languages, facilitate learning and coding and increase satisfaction [22].

Block-based approaches are also a common option for learning to program using robots. The most widespread and well-known programming environments are based on Scratch, such as *Mblock* [23] for Makeblock robotic kits [24], or Blockly, as *Makecode* [25] from Microsoft for the Micro:Bit device [26]. Other kits have their own software, such as *Lego Mindstorms EV3* [27] for Lego Mindstorms EV3 robots [28]. However, the trend of Lego is to integrate their robots with Scratch 3.0 [29]. This integration is achieved by offering a set of blocks that are added as extensions to Scratch 3.0.

Among the different educational tools for learning to code using robots, Open Roberta's approach with its NEPO meta language [30] is one of the most promising. NEPO uses Blockly for the design of their blocks but integrates a considerable number of robotic educational kits (*e.g.*, WeDo [31], EV3 series [28], Micro:bit [26], Nao [32], BOB3 [33], mBot (beta) [34], Calliope mini [35]). To achieve such a wide platform support, its design is built on top of a Hardware Abstraction Layer (HAL) [36]. However, for each robot, NEPO offers a specific set of non-reusable blocks in another robot.

Despite the variety of educational programming tools, none of them can be considered robot-agnostic. Thus, in many cases, the software tool was specifically designed for a specific robot. If there is a family of robots, an extension is included for each robot in the form of a set of new blocks. These extensions mainly provide access to the robot's sensors and actuators. In the case of NEPO (Open Roberta), the meta-language allows you to choose between several robots. Nevertheless, the same code, however simple, cannot be used in two different robots. If the selected robot is changed, the code has to be rewritten limiting code re-usability. In general, generated code from the blocks is different and specific to a particular educational robot.

From the perspective of hardware-abstraction, although PyRobot [37] is not an educational tool, its software design is worth considering. PyRobot is described as ''*a python-based robotics framework that isolates the ROS system [38] from the user-end and supports the same API across different robots*''. Despite its hardware-agnostic design, one of the main limitations of PyRobot is that only one hardware device of the same type can be configured for a certain robot. In addition, there exist some dependencies between the hardware and the

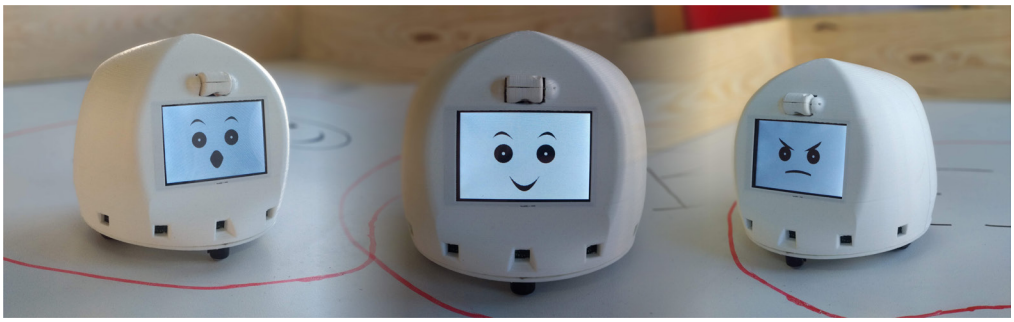


FIGURE 1. The robot EBO.

high-level functionality that could complicate its adaptation to new robots.

Considering other features, educational software tools generally allow users to visualise the code generated from the blocks. This property is inherent to all those based on Blockly and only depends on design decisions. However, it is not possible to directly modify this code from the tool and run it on the robot. In our opinion, this makes the transition to learning programming in a standard language more difficult. Furthermore, it is not common to find solutions that offer an intermediate code or text-based code that facilitates the jump from visual to textual programming.

In addition, the scope of most software tools for educational robots is limited, to a certain extent, to the elementary educational levels. Mainly, this is because complex behaviours that depend on certain type of hardware, such as cameras, are difficult to implement. In fact, the reviewed software tools do not have the ability to include external libraries, such as OpenCV [39] for image processing and analysis. In our experience in robotics, the implementation of robot behaviours with certain levels of complexity requires the use of third-party software that simplifies the development of the specific code. The authors of Pibot [40] agree with this argument by proposing a low-cost robot for STEM equipped with an RGB camera. This type of sensor is not usually included in educational robotics kits, however it is often found in real-life robotics applications. Image analysis offers great versatility in the type of challenges posed to students. In addition, it can help to raise the level of difficulty of programming to the level expected at university.

Another limitation of the tools studied is that it is not possible to create new blocks from code written directly in the high-level programming language. Although projects like Blockly provide the possibility to extend the set of blocks from code by writing code generators, the user cannot directly write the code in the target programming language. One of the advantages of this property is that writing small pieces of functional code in a formal language could provide a progressive way to introduce textual programming to a learner. Thus, the user could create a very limited functional block that could be easily tested. Even students could try to write the generated code from existing blocks, allowing them to self-assess their solution. But in addition, in our opinion, this property has the additional advantage for developers and advanced students of allowing them to develop complex functions (behaviours) directly in code.

LearnBlock, the proposed educational programming tool, emerges to integrate all the elements previously discussed and overcome the limitations of the existing tools. Next section provides an overview of its main features.

III. OVERVIEW OF OUR PROJECT

LearnBlock arose within the research project **Emorobotic** (supported by grant IB16090 from the Government of Extremadura), which aims to create a set of tools that facilitate the teaching-learning processes of programming in primary and secondary education, with a special focus on how programming serves as a means for developing abilities of emotional management. To this end, not only a programming tool has been built, but also a robot called EBO (Fig. 1), where students can emulate motor and emotional behaviours. LearnBlock provides a block-based language through which children can intuitively program robot behaviours and work with different robotic platforms. Both, EBO and LearnBlock, are open developments.

A. GENERAL FEATURES OF LEARNBLOCK

LearnBlock is available in the Python Package Index (PyPI) [41]. It can be installed using pip (https://pypi.org/project/learnblock) or from GitHub source (https://github.com/robocomp/LearnBlock).

LearnBlock provides a graphical user interface with the options to create a program using graphical elements (blocks) (Fig. 2). Thus, the user can build a program by selecting and connecting different blocks related to program control, actions and sensory information among others. In addition, the set of blocks to be used can be configured from the tool itself to let the user select the more appropriate types of visible blocks for a particular problem. Blocks can also be created from other blocks to encapsulate and reuse certain pieces of code.

All the above features are available in most of the existing tools, but LearnBlock also supports the creation of individual blocks from code. As previously mentioned, projects like

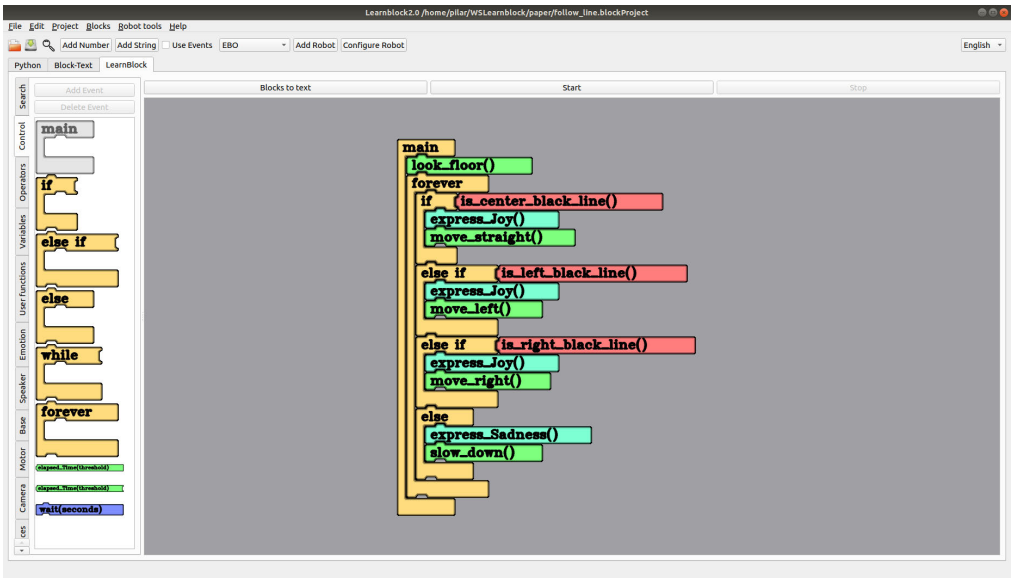


FIGURE 2. GUI of LearnBlock. Available blocks lists (left pane). Code editor (central pane).

Blockly provide the possibility to extend the set of blocks by writing code generators. In LearnBlock the user can directly write the code in the target language, specifying the statements that must be executed when the program flow reaches that point.

Regarding code generation, LearnBlock generates Python code from the textual representation of a visual program (block-based program). The generated code can be viewed and modified. Thus, the user can choose whether to create a program from blocks, from the textual representation of blocks or directly coding in Python. From the educational perspective, LearnBlock provides an integrated environment for learning programming where more complex concepts can be increasingly introduced, moving progressively towards a professional programming language.

LearnBlock is robot-agnostic, i.e., the same code can be used to program different robots. The novelty in this sense is that the robot-agnostic property is guaranteed not only for the visual code, but also for the generated one. This means that the user can create the visual or textual code with a high level of abstraction from the robot where the code will be finally executed. Currently, LearnBlock is compatible with different physical robots, such as Cozmo [42], Thymio [43], EBO and Lego EV3 [28], and simulated ones under RCIS [44] and V-REP [45]. New robots can be added without modifying the core code of LearnBlock or redefining blocks. More specifically, to use a new robotic platform, the only requirement is to provide a Python class implementing the access to the different hardware devices of the robot as well as providing connection and disconnection methods to the platform. Details of this process are explained in section IV.

B. PROGRAMMING MODELS IN LEARNBLOCK

To facilitate the programming of robot behaviours with various levels of complexity, two different models of programming can be used in LearnBlock: sequential programming and event-driven programming.

In sequential programming, the user includes the statement blocks inside a main block as they must be executed. Function blocks including other sequences of blocks can also be created. The execution of such functions is carried out in the order they are called.

In the event-driven programming model, statement blocks are included in blocks associated with events, named ***when*** blocks. A ***when*** block is activated whenever the associated event occurs, which implies the execution of the sequence of blocks it contains. ***when*** blocks can also be defined without an associated external event, and be triggered or stopped from other points of the code using specific statements to activate or deactivate them. These blocks allow for the creation of robot states that ease the implementation of not purely reactive behaviours [46]. In addition, each ***when*** block has two associated statements providing information about the current state of the block (active or inactive) and its active time. This information can be used to program changes of state in the robot according to the activation of other states or the time elapsed since a given situation has occurred.

IV. SOFTWARE ARCHITECTURE OF LEARNBLOCK

The development of a programming tool that can be used for programming a wide variety of robotic platforms requires a thoughtful design. To make LearnBlock an effective robot-agnostic tool, the platform-dependent code (mainly hardware-access code) has to be separated from the rest of

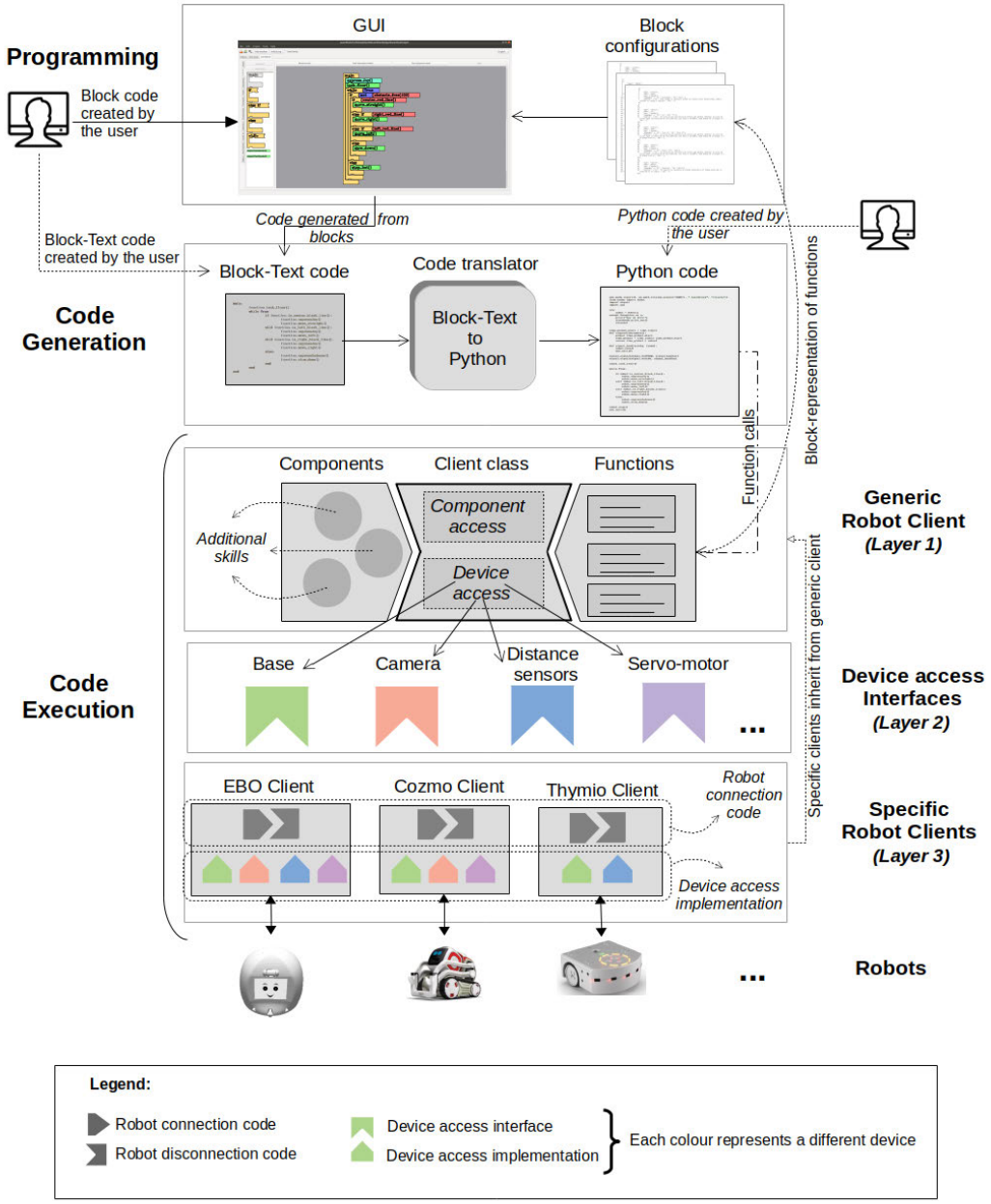


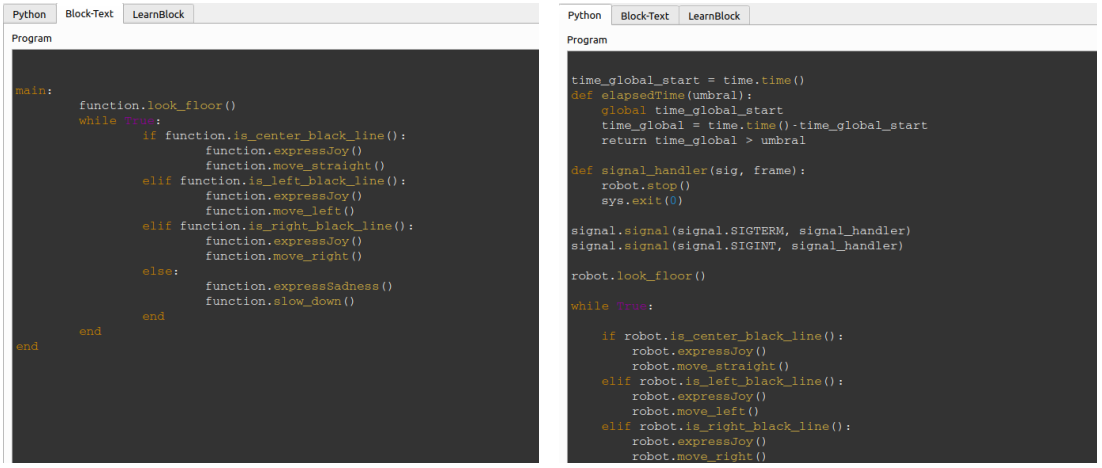
FIGURE 3. General architecture of LearnBlock.

the implementation. In addition, the set of functional blocks used to create a program using the visual language could be different for distinct robotic platforms or even could be extended over time. Thus, code generation should guarantee a high degree of adaptability to new robot skills while maintaining a weak dependency on the visual language. Taking all these considerations into account, LearnBlock design is based on a model of weakly coupled software modules that separates visual programming from code generation and program execution.

Fig. 3 depicts the software architecture of LearnBlock. The software elements are distributed into different levels related to programming, code generation and code execution. The code execution elements are in turn distributed in three different groups that separate the required generic code for program execution from the code related to each specific robotic platform.

A. BLOCK PROGRAMMING

Block programming is carried out through a graphical user interface that includes the necessary tools to create, modify and execute the code (Fig. 2). Blocks are organised in several tabs associated with different categories that represent



(a) Block-Text code view of the LearnBlock GUI.

(b) Python code view (partial) of the LearnBlock GUI.

FIGURE 4. Block-Text and Python code generated from the visual code of Fig. 2.

the main functionality of the blocks they contain. Four of these categories are inherent to the visual language: control, operators, variables and user functions (definition and calls). These four categories are often understood as compatible between different robots. It seems clear to all designers of block-based languages, that for example, the statement *If* is the same for any device, however, they do not raise that level of abstraction to common primitives in robotics, such as *move forward* or *rotate* a certain number of degrees. LearnBlock fills this gap by proposing an underlying software architecture that allows it.

Thus, in LearnBlock, all the remaining block categories not included in the above four ones are related to blocks providing platform-independent robot skills. Each of these blocks is associated with a Python function that implements that functionality. These Python functions (section IV-C) are independent modules that are included in the GUI through block-configuration files. A block-configuration file is a JSON-format file describing the block attributes used to visually represent each function. Listing 1 shows an example of the block configuration of a function named *turn* in charge of turning the base of the robot a certain angle.

```
"type": "motor",
"category": "Base",
"name": "turn",
"variables":
[
    {
    "type": "float",
    "name": "angle",
    "default": "0",
    "translate": {"ES": "angulo",
                  "EN": "angle"}
    }
],
"shape": ["blockVertical"],
"languages": {"ES": "girar",
              "EN": "turn"},
"tooltip": {"ES": "Gira la base del robot un
    angulo determinado","EN": "Make the robot
     base turn a certain angle"}
```

Listing 1. Block configuration example.

B. CODE GENERATION

Each block of the visual language has an equivalent statement in a domain-specific language called *Block-Text*. Block-Text is a simplified programming language that provides a textual representation of the visual language. Fig. 4(a) shows the Block-Text code of the visual program of Fig. 2.

Block-Text code is translated to Python (Fig. 4(b)) for code execution. The translator module is invoked whenever the user runs a program. Additionally, LearnBlock includes options to generate Block-Text and Python code without execution. Once the code has been generated, the user can access the Block-Text and Python versions through the corresponding tabs of the GUI. Fig. 4 shows examples of these two views.

Both views are editable, i.e., the user can modify Block-Text and Python codes or even create those codes from scratch. The purpose of these views is to progressively introduce the learner to a professional programming language i.e., Python. Thus, the user can load and save code in Block-Text and Python to directly program in any of these two languages besides using visual programming.

The final language for program execution is Python. The auto-generated Python program includes the creation of an instance of a robot client class that is in charge of the communication with the robotic platform. Python functions implementing robot skills are included as member functions of the client object during the creation of a new instance (section IV-C). Thus, each block associated with these functions is translated in the final Python code as a call to a method of the robot client. The generated program also includes control code for a clean interruption of the execution (section V), ensuring proper stop and disconnection from the robot.

As previously mentioned, LearnBlock is robot-agnostic not only at the level of the block language, but also at the level of the generated code. This means that the generated Python program is independent of the robot running the code. To achieve this level of abstraction, the robot instance is created from a specific client class that contains common methods to access the hardware of the robot. The specific client module is imported including the corresponding Python statement at the beginning of the generated code. This is the only line that differs in the final code when different robots are used for running the same implementation of a given behaviour.

C. CODE EXECUTION

The elements of the architecture related to code execution abstract the underlying hardware by means of an organisation composed of three layers: ''generic robot client'', ''device access interfaces'' and ''specific robot clients''.

The first layer (generic robot client) includes a generic client class characterised by the following features:

- Robot-abstraction layer: implements a base class for specific robot client classes.
- Generic communication with devices: provides generic access to common devices of robots.
- Extensible and adaptable functionality: adds external functions as class members during instance creation.
- Access to additional robot skills: starts and communicates with software components that implements additional perceptual abilities for robots.

Every client class communicating to a specific robot (third layer) inherits from this generic class (first layer). Thus, the communication with sensors and actuators, of any robot can be achieved using methods of the generic class, abstracting this way the tasks of programming and code generation from the specific robotic platform. Nevertheless, inheritance by itself could complicate the implementation of specific clients. Thus, according to the robot sensors and actuators the methods that should be re-implemented will differ among the different platforms making the creation of a new client potentially error-prone. In order to avoid this problem, specific clients adapt the behaviour of the generic client for specific robots through device access interfaces (second layer).

Device access interfaces are classes that include generic methods to communicate with common devices in robots. These methods bridge the generic access and the real communication with a particular device and are implemented as a call to a function that is passed as a parameter in the initialisation of a ''device object''. Such functions implement the access to the specific device.

Listing 2 shows the definition of a device access interface for gyroscopes. When a client class creates a device of this type, it must pass two functions as arguments, one for reading the current rotation and another one to reset the device. For example, the specific client class of Listing 3, extracted from the client class of the robot EV3, creates a device of type

```
1  class Gyroscope():
2      rot = None
3
4      def __init__(self, _readFunction,
           _resetFunction):
5          self.__readDevice = _readFunction
6          self.__resetDevice = _resetFunction
7
8      def set(self, _rot):
9          self.rot = _rot
10
11     def get(self):
12         if self.rot is None:
13             self.read()
14         return self.rot
15
16     def read(self):
17         _rot = self.__readDevice()
18         if self.rot is not None:
19             self.set(_rot)
20
21     def reset(self):
22         self.__resetDevice()
```

Listing 2. Device access interface of gyroscopes (second layer).

gyroscope (line 11), associating the read and reset functions with two methods implementing the required functionality, *deviceReadGyroscope* (lines 33–35) and *deviceResetGyroscope* (lines 37–39).

To manage and access the devices of a robot, the generic client class (first layer) includes a dictionary for each type of device as well as methods to add new devices to the associated dictionary. When a new device is defined in a specific client (see line 11 of Listing 3), it must be included in the set of accessible devices by calling the corresponding method in charge of adding the device to one of the dictionaries (*addGyroscope* in the case of a device of type gyroscope). In such a method, the device is associated with a unique key that identifies it from the rest of the devices of the same type. For example, in the case of a gyroscope, the key could be the rotation axis (''Z_AXIS'' in Listing 3). If no key is passed as a parameter, the method assigns the default key ''ROBOT''. The generic client class accepts multiple devices of any type. Thus, a specific client can define several cameras, gyroscopes, motors, etc. or even several robot bases to provide control for multiple robots.

In addition to the definition of devices and the implementation of the particular device access methods, specific clients include code for connecting and disconnecting the robot. The connection to the robot (method *connectToRobot* of Listing 3) is invoked in the client constructor to ensure it is properly established before calling any device method. On the other hand, the code for robot disconnection is provided as a re-implementation of the method *disconnect()* of the parent class (lines 28 and 29 of Listing 3). This method is called whenever a robot program is finished or interrupted.

The generic client class contains methods to access all the types of available devices by calling the appropriate methods of the device access interfaces. These methods provide the communication with the different devices to any external

```
from learnbot_dsl.Clients.Client import *
from learnbot_dsl.Clients.Devices import *
import rpyc, os
[...]

class Robot(Client):

  def __init__(self):
    Client.__init__(self, _miliseconds=100)
    self.connectToRobot()
    self.addGyroscope( Gyroscope (
        _readFunction=self.
        deviceReadGyroscope, _resetFunction=
        self.deviceResetGyroscope),"Z_AXIS" )
    [...]

  def connectToRobot(self):
    configRobot = {}
    with open(os.path.join(os.path.dirname(os
        .path.realpath(__file__)), "EV3.cfg")
        , "rb") as f:
        configRobot = json.loads(f.read())
    robotIP = configRobot["RobotIP"]
    self. conn = rpyc.classic.connect(robotIP
        )
    self.ev3Motor = self.conn.modules[
        'ev3dev2.motor']
    [...]
    self.ev3Base = self.ev3Motor.MoveTank(
        LEFT_MOTOR, RIGHT_MOTOR)
    self.ev3Sensors = self.conn.modules[
        'ev3dev2.sensor.lego']
    [...]
    self.gyrosensor = self.ev3Sensors.
        GyroSensor()
    self.gyrosensor.mode = 'GYRO-ANG'

  def disconnect(self):
    self.ev3Base.stop()

 [...]

  def deviceReadGyroscope(self):
      rz =self.gyrosensor.value()
      return rz

  def deviceResetGyroscope(self):
      self.gyrosensor.mode = 'GYRO-CAL'
      self.gyrosensor.mode = 'GYRO-ANG'
```

Listing 3. Specific Robot client example (client class of EV3).

code that uses a robot client. In addition, the generic client is in charge of periodically reading from all the defined sensors to avoid undesirable waits when sensory data are required.

The basic functionality for hardware access provided by a client class is extended in two ways. The first one is by dynamically adding external Python functions associated with certain types of blocks (blocks corresponding to robot skills). These Python functions take an instance of a robot client as a parameter through which they access the robot devices to implement a given skill. The set of existing functions is provided by a package of the LearnBlock project called *functions*. This package generates a list of the available functions searching for the modules defined from certain paths. Specifically, two initial paths are considered: the default function path of LearnBlock and an additional path where functions created by the users are included. In the creation of a new instance, the generic client class inspects this list of functions and adds them as class members. The final set of included functions can be limited by passing as argument a list with the names of the used functions. This is automatically done by the code generator.

The second way the functionality of the generic client is extended is through the connection to software components (RoboComp components [47]) implementing additional robot skills, mainly related to perception. Specifically, the current version connects to two components providing additional visual skills: detection and identification of AprilTags [48] and recognition of emotions in people. These new skills are accessible by means of a set of methods of the generic class that communicates with these components to obtain the required perceptual information. Thus, a code using a robot client instance, e.g. an added external Python function, can call this set of methods to implement interactive skills in any robot equipped with a camera.

Our approach shares some design features with PyRobot, although there are important differences. One of the principal ones is that in PyRobot the number of devices of each type that a robot can include is limited to one, restricting the support of new robots. In addition, platform interfacing in PyRobot is solved by the creation of device classes that inherit from certain classes of the core API. The methods that should be re-implemented are in some cases not only related to access to devices, but to more general functionalities, as occurs in the class *Base* which, for example, includes methods to go to an absolute or relative position. In other cases, new methods not defined in the parent class should be created to provide a complete functionality. For instance, this happens in the implementation of the class *Camera* of the wrapper on LoCoBot [49] to give access to its motorised camera. All these issues limit code re-usability and hardware abstraction.

V. ROBOT-AGNOSTICISM TESTS

This section presents some programming examples created with LearnBlock using various robots. Specifically, three different programs tested in different pairs of robots are shown. Results of the execution of the three tests are available in https://youtu.be/P6LK0w9KCDM. Next, they are briefly described:

- **Square trajectory**: robots executing this program follow a square trajectory by an iterative process of 4 iterations. In each iteration, the robot moves straight during a certain number of seconds and then changes its current orientation by 90 degrees. This program has been tested on Cozmo and EV3 (Fig. 5).
- **Reaction to tags**: in this program, the robot has to search for tags (AprilTag) and approach them. The approaching action stops once the robot is situated so near the tag that it cannot be detected. At that moment, the robot expresses a certain emotion depending on the detected

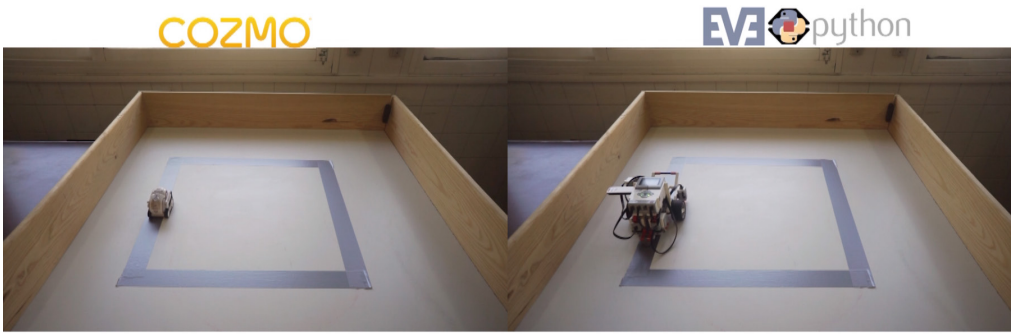


FIGURE 5. Square trajectory test.

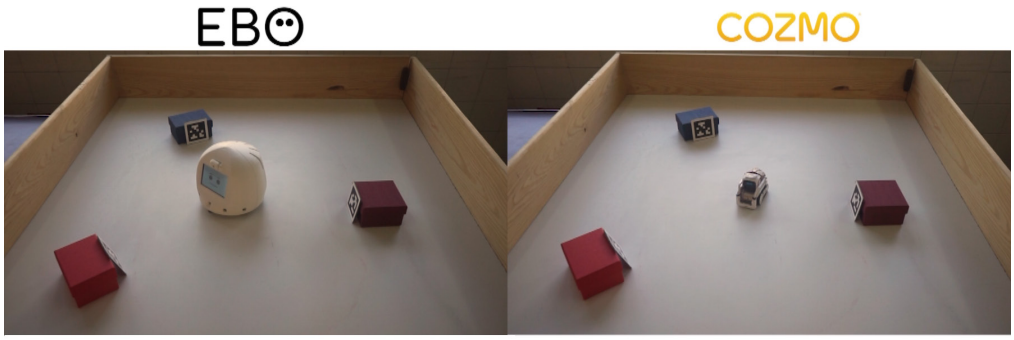


FIGURE 6. Reactions to tags test.

```
times = None
angle = None
main:
  times = 0
  angle = 90
  function.reset_orientation()
  while times < 4:
    function.move_straight()
    function.sleep(5)
    function.stop_bot()
    function.set_orientation(angle)
    times += 1
    angle += 90
  end
end
```

Listing 4. Block-Text code of the *Square trajectory* test.

tag. After several seconds, the robot turns around again to search for a new tag. The program was tested on the robots EBO and Cozmo (Fig. 6).

- **Line follower**: the robot has to detect and follow a black line on the floor, showing different emotions depending on whether the line is detected or not. The robots used for testing this program are EBO and Thymio (Fig. 7).

The Block-Text code of the first test, *Square trajectory*, is shown in Listing 4 and the corresponding generated Python code in Listings 5 and 6. The Python code in Listing 5 is common to every generated code from a Block-Text source in LearnBlock. It includes the creation of a robot instance and the definition of two functions. The first function, *elapsed-*

```
from __future__ import print_function,
    absolute_import
import sys, os, time, traceback
sys.path.insert(0, os.path.join(os.getenv('
    HOME'), ".learnblock", "clients"))
 from XXX import Robot
import signal
import sys

try:
    robot = Robot()
except Exception as e:
    print("Problems creating a robot instance
        ")
    traceback.print_exc()
    raise(e)

time_global_start = time.time()
def elapsedTime(umbral):
    global time_global_start
    time_global = time.time()-
        time_global_start
    return time_global > umbral

def signal_handler(sig, frame):
    robot.stop()
    sys.exit(0)

signal.signal(signal.SIGTERM, signal_handler)
signal.signal(signal.SIGINT, signal_handler)
```

Listing 5. Common Python code for all the examples.

Time, is used as a language primitive to check if a certain time has elapsed from the beginning of the program execution. The second function, *signal_handler*, is a signal

FIGURE 7. Line follower test.

```
times = None
angle = None
times = 0
angle = 90
robot.reset_orientation()

while times < 4:
    robot.move_straight()
    robot.sleep(5)
    robot.stop_bot()
    robot.set_orientation(angle)
    times += 1
    angle += 90

robot.stop()
sys.exit(0)
```

Listing 6. Specific Python code of the *Square trajectory* test.

handler that is associated with termination signals to properly stop and disconnect the robot when the program is interrupted. This common Python code also includes the only platform-dependent line of code (highlighted in the Listing 5), in charge of importing a specific robot client class.

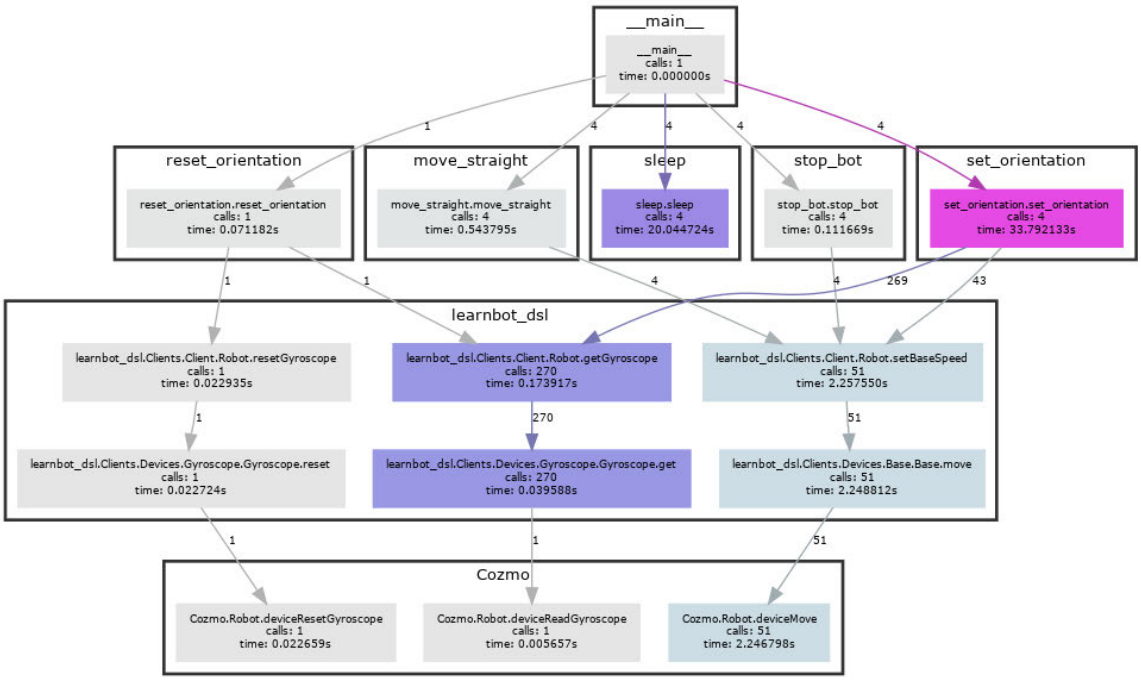
As can be seen in Listing 6, the specific Python code of the first example has many similarities with the Block-Text code, which favours the transition between the two languages during the learning process. Thus, the main change is related to *function* statements, which are replaced in the generated code with calls to methods of the *robot* object.

Results of the execution of this test can be seen in the first part of the video available in the link provided above. Besides these results, Fig. 8 shows two call graphs generated during the execution of the test on Cozmo and EV3. Each call graph depicts the calling relationships between the different modules composing the program. The main program (root node) makes calls to the different behavioral functions taking part in the square trajectory problem. These functions invoke methods of the generic client class related to the communication with the gyroscope and the base of the robot (top nodes of the central group). The access to these specific devices is carried out by calling methods of the corresponding device interfaces (lower nodes of the central group). These calling flows are independent of the robot executing the program.

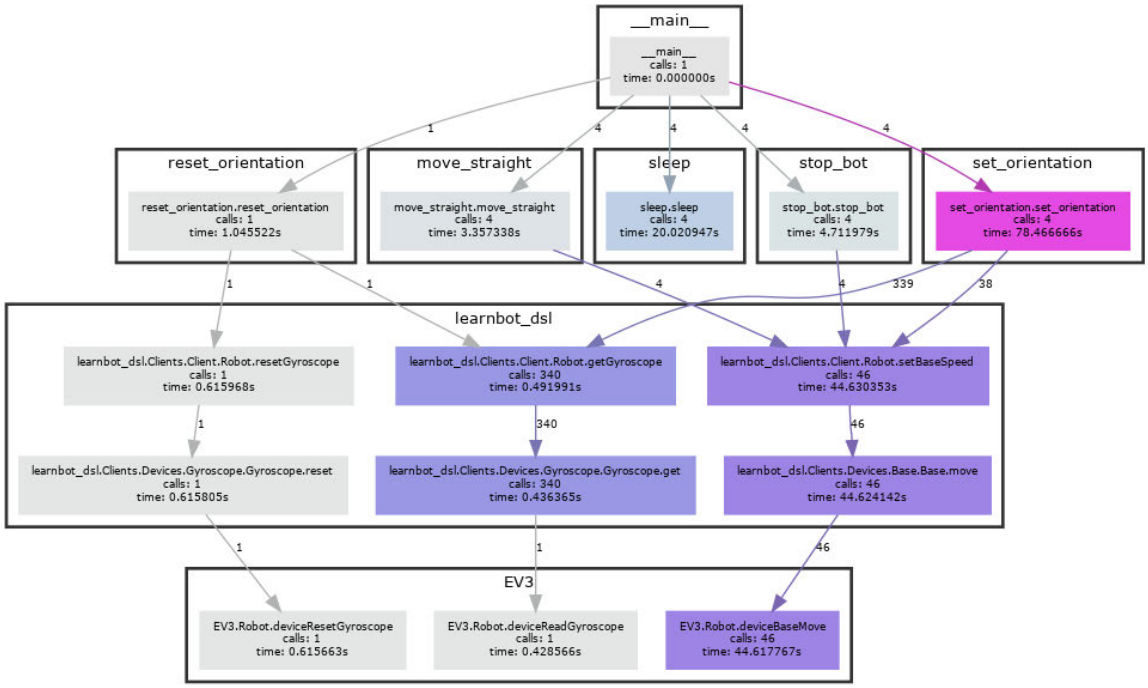
```
when start:
  function.expressNeutral()
  function.look_front()
  activate maincontrol
end

when maincontrol:
  if state_approach_tag:
    if function.is_tag(1):
      activate showJoy
    elif function.is_tag(2):
      activate showSurprise
    else:
      activate showFear
    end
  end
end

when no_tag = not function.is_any_tag():
  function.look_front()
  function.turn_right()
end

when approach_tag = function.is_any_tag():
  if function.tag_on_the_right():
    function.move_right()
  elif function.tag_on_the_left():
    function.move_left()
  else:
    function.move_straight()
  end
end
```

Listing 7. Block-Text code of the *Reaction to tags* test (part 1).

The only differences between the execution flows on the two robots take place in the final communication with the hardware devices, which has to be solved by the invocation of methods of the specific robot clients (lowest level nodes). Although this invocation is explicit in execution time, from the programmer point of view, it is implicitly solved by the definition of devices in the specific client code, as was described in section IV-C. Differences in the required execution time for completing the program can also be observed in the graphs. These differences, which can also be appreciated in the execution shown in the video, are depicted in the graphs by the distinct colours of each node (see caption of Fig. 8). They are mainly due to the particular features of the



(a) Call graph generated during the execution of the first test on Cozmo.



(b) Call graph generated during the execution of the first test on EV3.

FIGURE 8. Call graphs generated during the execution of the first test (*Square trajectory*). The different colours of each node are related to the execution time of each function or method. Gray and blue colours are assigned to functions consuming low to moderate percentages of the total execution time, while purple and pink colours are associated to those functions requiring higher time percentages.

sensors used by the two robots and to specific delays in the communication with both of them. Despite these differences, both robots behave according to the program specifications and successfully complete the execution of the test.

The second test, *Reaction to tags*, shows an example of event-driven programming using ***when*** statements. Block-Text code of this test is shown in Listings 7 and 8. The different ***when*** statements deals with the distinct behaviours

```
1  main:
2    function.look_floor()
3    while True:
4      if function.is_center_black_line() :
5        function.expressJoy()
6        function.move_straight()
7      elif function.is_left_black_line() :
8        function.expressJoy()
9        function.move_left()
10     elif function.is_right_black_line() :
11       function.expressJoy()
12       function.move_right()
13     else:
14       function.expressSadness()
15       function.slow_down()
16     end
17   end
18 end
19
```

```
1  main:
2    function.look_floor()
3    while True:
4      if function.is_center_ground_line() :
5        function.expressJoy()
6        function.move_straight()
7      elif function.is_left_ground_line() :
8        function.expressJoy()
9        function.turn_left()
10     elif function.is_right_ground_line() :
11       function.expressJoy()
12       function.turn_right()
13     else:
14       function.expressSadness()
15       function.slow_down()
16     end
17   end
18 end
19
```

FIGURE 9. Block-Text code of the *Line follower* test for EBO (left) and Thymio (right).

the robot has to exhibit when each considered situation takes place. Thus, when the program starts, the robot looks in front of it and activates the state *maincontrol*, which in turn activates emotional states (*showJoy*, *showSurprise* or *showFear*) related to the detected tag, if any. Two other ***when*** statements control the motor behaviour of the robot when it finds a visible tag and when no tag is perceived. These two states (*no_tag* and *approach_tag*) are associated with external events related to the presence or absence of a visible tag. Finally, the three ***when*** statements of Listing 8 control the emotional behaviour of the robot, making it express an emotion once the *approach_tag* state is deactivated. As a consequence, the corresponding emotion is shown once the robot is stopped in front of the detected tag, after approaching it. The second part of the video shows the results of the execution of this second test. As can be observed, both robots exhibit equivalent behaviours.

The first test, *Square trajectory* (Listing 4), and the second test, *Reaction to tags* (Listings 7 and 8), show two programs that have been executed in two different robots endowed with the same required types of hardware devices. The third example showcases code platform-independence, and is used to describe how LearnBlock's design overcomes some of those limitations, offering equivalent behaviours for different hardware devices when possible. This is illustrated through two implementations of the *Line follower* problem for EBO and Thymio robots. Results of the execution of this test can be seen in the last part of the video.

The two programs differ because EBO detects the line with a camera while Thymio uses ground infrared sensors (see Fig. 9). The differences in both Block-Text codes have been highlighted in blue in the listings. As can be expected, the line detection functions are different (lines 4,7,10) due to the use of different sensors in both robots.

Despite both robots including a differential base, the functions *move_left* and *move_right* of the program for EBO have been replaced with the functions *turn_left* and *turn_right* in the code for Thymio (lines 9 and 12). This change is not related to the base of both robots, but to the different

```
1  when showJoy:
2    if not state_approach_tag:
3      function.stop_bot()
4      function.expressJoy()
5      function.sleep(2)
6      deactivate showJoy
7    end
8  end
9
10 when showSurprise:
11   if not state_approach_tag:
12     function.stop_bot()
13     function.expressSurprise()
14     function.sleep(2)
15     deactivate showSurprise
16   end
17 end
18
19 when showFear:
20   if not state_approach_tag:
21     function.stop_bot()
22     function.expressFear()
23     function.sleep(2)
24     deactivate showFear
25   end
26 end
```

Listing 8. Block-Text code of the *Reaction to tags* test (part 2).

sensors used to detect the line. Thus, the limited range of the infrared sensors of Thymio makes it necessary to carry out turns, without translations, to prevent the robot from moving out of the line. Any additional translation movement when the robot is turning would cause the infrared sensors to stop detecting the line. It must be noted that the functions *move_straight* and *slow_down* are maintained in both codes since they are not affected by the different ranges of the sensors.

Even though the robot Thymio does not have a display to show emotional expressions, the client class of Thymio implements the *Display* device to show emotions as colours using its RGB LEDs. This is carried out in the Thymio class by adding a device of type *Display* and associating one of its interface methods to a method of the client class in charge of setting the LEDs of the robot to a certain colour according

to the emotion received as parameter. For this reason, just like EBO does, when Thymio perceives the line, it shows *joy* (Thymio's RGB LEDs light up in green) and, when the line is not detected, it shows *sadness* (Thymio's RGB LEDs light up in blue).

The call to the function *look_floor* (highlighted in red) has been maintained in the code for Thymio to show that its removal is not mandatory. Thus, if a function requires a device for its execution and that device is not available, it simply has no effect.

VI. CONCLUSION

There exists a wide consensus about the importance of learning programming during the early stages of education. In recent years many tools have emerged to introduce programming concepts to children, making use mostly of block-based programming languages. The use of robots in learning coding is increasing. Robots may improve the motivation of children in the learning process and, in addition, can be very useful for introducing other STEM-related contents and CT.

Despite the variety of educational programming tools and robots, there are no integrative solutions that provide a common environment for learning programming using robots. LearnBlock fills that gap by proposing a new software architecture that gives rise to a flexible programming environment for the integration of different robotic platforms, making it the first robot-agnostic educational programming tool. The extensibility of LearnBlock is related not only to the inclusion of new robots, but also to the possibility of extending the functional blocks from code without modifying the core code of the tool. This allows for the creation of complex blocks providing a wide variety of functionality for programming different types of robots. In addition, LearnBlock proposes a progressive learning process that leads to the use of a general-purpose programming language. This is achieved by extending the robot-agnostic property to the generated code.

Most popular educational tools for learning coding are web-based. This undoubtedly eases the usage of the tools avoiding any installation process. The main limitation of a web-based solution takes place in the code execution phase, since, for different robots, the communication mechanisms vary and, in some cases, a local connection is required between the computer and the robot (for instance, using Bluetooth or NFC). In addition, local installation of specific software could be required for using some robots. This is the main reason why LearnBlock was developed as a desktop application. Since LearnBlock is entirely written in Python, it is cross-platform. Moreover, it is available as a PyPi package to facilitate its installation and automatically resolve its dependencies. However, we are aware of the benefits of web-based software for educational purposes. Thus, part of our future work aims to provide a web-based version of LearnBlock that includes most of the features of the current version.

Hardware abstraction is relevant not only for educational purposes, but also for facilitating the development of high-level algorithms. LearnBlock provides the required software infrastructure for the implementation and testing of algorithms in the field of robotics, abstracting the development from low-level issues. This infrastructure is open to any kind of robotic devices, such as robotic arms, as well as to any type of robot, either educational or not. To make easier adapting new robot platforms to LearnBlock, we are developing a tool that generates templates for specific client classes based on high-level platform descriptions. In addition, software support for new kinds of sensors and actuators is being added in order to extend the variety of robots that can be programmed with LearnBlock.

REFERENCES

[1] S. Papert. (1981). *Mindstorms: Children, Computers, Powerful Ideas*. [Online]. Available: http://www.amazon.fr/exec/obidos/ASIN/0465046274/citeulike04-21
[2] J. Cuny, L. Snyder, and J. M. Wing. (2010). *Demystifying Computational Thinking for Non-Computer Scientists*. [Online]. Available: http://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf
[3] V. J. Shute, C. Sun, and J. Asbell-Clarke, "Demystifying computational thinking," *Educ. Res. Rev.*, vol. 22, pp. 142–158, Nov. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1747938X17300350
[4] J. Wing, "Computational thinking," *Commun. ACM*, vol. 49, pp. 33–35, Mar. 2006.
[5] J. Figueiredo and F. J. García-Peñalvo, "Improving computational thinking using follow and give instructions," in *Proc. 5th Int. Conf. Technol. Ecosyst. Enhancing Multiculturality*, Oct. 2017, pp. 1–7.
[6] A. Balanskat and K. Engelhardt, "Computing our future: Computer programming and coding—Priorities," School Curricula Initiatives Across Eur., European Schoolnet, Brussels, Belgium, Tech. Rep., Oct. 2015.
[7] F. J. García-Peñalvo, "A brief introduction to tacle 3—Coding European project," in *Proc. Int. Symp. Comput. Educ. (SIIE)*, Sep. 2016, pp. 1–4.
[8] F. J. García-Peñalvo, D. Reimann, M. Tuul, and A. Rees, "An overview of the most relevant literature on coding and computational thinking with emphasis on the relevant issues for teachers," TACCLE3 Consortium, Brussels, Belgium, Tech. Rep., 2016, doi: 10.5281/zenodo.165123.
[9] (2019). *Learn Computer Science. Change the World*. [Online]. Available: https://code.org/
[10] KhanAcademy. (2019). *Khan Academy*. [Online]. Available: https://www.khanacademy.org
[11] C. B. Frey and M. A. Osborne, "The future of employment: How susceptible are jobs to computerisation?" *Technol. Forecasting Social Change*, vol. 114, pp. 254–280, Jan. 2017.
[12] Deloitte Firm, "From brawn to brains: The impact of technology on jobs in the U.K.," Deloitte, London, U.K., Tech. Rep., 2015. [Online]. Available: https://www2.deloitte.com/uk/en/pages/growth/articles/from-brawn-to-brains–the-impact-of-technology-on-jobs-in-the-u.html
[13] J. Manyika, S. Lund, M. Chui, J. Bughin, J. Woetzel, P. Batra, R. Ko, and S. Sanghvi, *Jobs Lost, Jobs Gained: Workforce Transitions in a Time of Automation*, vol. 150. San Francisco, CA, USA: McKinsey Global Institute, Dec. 2017.
[14] M. A. K. Bahrin, M. F. Othman, N. N. Azli, and M. F. Talib, "Industry 4.0: A review on industrial automation and robotic," *J. Teknologi*, vol. 78, nos. 6–13, pp. 137–143, 2016.
[15] F. B. V. Benitti, "Exploring the educational potential of robotics in schools: A systematic review," *Comput. Educ.*, vol. 58, no. 3, pp. 978–988, 2012.
[16] L. Lammer, W. Lepuschitz, C. Kynigos, A. Giuliano, and C. Girvan, "ER4STEM educational robotics for science, technology, engineering and mathematics," in *Proc. Robot. Edu.* Cham, Switzerland: Springer, 2017, pp. 95–101.
[17] F. Rubinacci, M. Ponticorvo, R. Passariello, and O. Miglino, "Robotics for soft skills training," *Res. Educ. Media*, vol. 9, no. 2, pp. 20–25, Dec. 2017.
[18] G. Chen, J. Shen, L. Barth-Cohen, S. Jiang, X. Huang, and M. Eltoukhy, "Assessing elementary students' computational thinking in everyday reasoning and robotics programming," *Comput. Educ.*, vol. 109, pp. 162–175, Jun. 2017.
[19] S. Cooper, W. Dann, and R. Pausch, "Alice: A 3-D tool for introductory programming concepts," *J. Comput. Sci. Colleges*, vol. 15, no. 5, pp. 107–116, 2000.

[20] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. S. Silver, and B. Silverman, "Scratch: Programming for all," *Commun. ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[21] E. Pasternak, R. Fenichel, and A. N. Marshall, "Tips for creating a block language with blockly," in *Proc. IEEE Blocks Beyond Workshop (BB)*, Oct. 2017, pp. 21–24.

[22] M. Kölling, N. C. Brown, and A. Altadmri, "Frame-based editing," *J. Vis. Lang. Sentient Syst.*, vol. 3, no. 1, p. 1, 2017.

[23] Makeblock. (2019). *MBlock: Block-Based Coding Platform for Teaching and Learning Coding*. [Online]. Available: https://www.mblock.cc/en-us/

[24] Makeblock. (2019). *MakeBlock: A Global Steam Education Solution Provider*. [Online]. Available: https://www.makeblock.com/

[25] Microsoft. (2019). *Makecode for Micro: Bit*. [Online]. Available: https://makecode.microbit.org

[26] Micro:Bit Educational Foundation. (2019). *Micro:Bit*. [Online]. Available: https://microbit.org/

[27] Lego. (2019). *EV3 Mindstorms Software*. [Online]. Available: https://education.lego.com/en-us/downloads/mindstorms-ev3/software

[28] Lego. (2019). *EV3 Lego Mindstorms*. [Online]. Available: https://www.lego.com/en-es/themes/mindstorms

[29] Scratch. (2019). *Scratch—Lego Mindstorms EV3*. [Online]. Available: https://scratch.mit.edu/ev3

[30] M. Ketterl, B. Jost, T. Leimbach, and R. Budde, "Open roberta—A Web based approach to visually program real educational robots," *Tidsskriftet Læring og Medier (LOM)*, vol. 8, no. 14, p. 1, 2015.

[31] Lego. (2019). *Wedo 2.0: Primary School. Lego Education*. [Online]. Available: https://education.lego.com/en-us/elementary/intro/wedo2

[32] Softbankrobotics. (2019). *Nao: The Humanoid Robot*. [Online]. Available: https://www.softbankrobotics.com/emea/en/nao

[33] Bob3. (2019). *BOB3: A Little Robot Friend Who Learns Programming With You*. [Online]. Available: https://www.bob3.org/en/

[34] Makeblock. (2019). *MBOT: A Steam Education Robot for Beginners*. [Online]. Available: https://www.makeblock.com/steam-kits/mbot

[35] Calliope. (2019). *Calliope Mini: A Tiny Computer Designed to Show You the Fun in Programming*. [Online]. Available: https://calliope.cc/

[36] B. Jost, M. Ketterl, R. Budde, and T. Leimbach, "Graphical programming environments for educational robots: Open roberta-yet another one?" in *Proc. IEEE Int. Symp. Multimedia*, Jun. 2014, pp. 381–386.

[37] A. Murali, T. Chen, K. V. Alwala, D. Gandhi, L. Pinto, S. Gupta, and A. Gupta, "Pyrobot: An open-source robotics framework for research and benchmarking," CoRR, Cornell Univ., New York, NY, USA, Tech. Rep. 1906.08236, 2019.

[38] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, Kobe, Japan, vol. 3, 2009, p. 5.

[39] G. Bradski and A. Kaehler, *Learning OpenCV: Computer Vision With the OpenCV Library*. Newton, MA, USA: O'Reilly Media, 2008.

[40] J. Vega and J. Cañas, "PiBot: An open low-cost robotic platform with camera for STEM education," *Electronics*, vol. 7, no. 12, p. 430, Dec. 2018.

[41] PyPI. (2019). *The Python Package Index (PYPI): A Repository of Software for the Python Programming Language*. [Online]. Available: https://pypi.org

[42] Anki. (2019). *Cozmo*. [Online]. Available: https://anki.com/en-us/cozmo.html

[43] Thymio. (2019). *Thymio*. [Online]. Available: https://www.thymio.org/

[44] M. A. Gutiérrez, A. Romero-Garcés, P. Bustos, and J. Martínez, "Progress in RoboComp," in *Proc. Workshop Phys. Agents*, Santiago de Compostela, Spain, 2012, p. 1.

[45] E. Rohmer, S. P. N. Singh, and M. Freese, "V-REP: A versatile and scalable robot simulation framework," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Nov. 2013, pp. 1321–1326.

[46] M. Ben-Ari and F. Mondada, *Elements of Robotics*. Cham, Switzerland: Springer, 2018.

[47] L. Manso, P. Bachiller, P. Bustos, P. Núnez, R. Cintas, and L. Calderita, "Robocomp: A tool-based robotics framework," in *Proc. Int. Conf. Simulation, Modeling, Program. Auto. Robots*. Cham, Switzerland: Springer, 2010, pp. 251–262.

[48] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *Proc. IEEE Int. Conf. Robot. Autom.*, Jun. 2011, pp. 3400–3407.

[49] Locobot. (2019). *Locobot: An Open Source Low Cost Robot*. [Online]. Available: http://www.locobot.org/

PILAR BACHILLER-BURGOS was born in Cáceres, Extremadura, Spain, in 1974. She received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the University of Extremadura (UEx), Spain, in 1997, 2000, and 2008, respectively. She has been a member of the Robotics and Artificial Vision Laboratory (RoboLab), University of Extremadura, since 2001. Within RoboLab, she has actively participated in the design and construction of several robots of different types and functionalities, including mobile platforms for research and teaching, robots for neurorehabilitation, or expressive heads for social robots. She has also participated in the development of the robotic open-source framework RoboComp which is currently used by several universities and research groups. She is currently an Associate Professor with the Department of Computer and Communication Technology, UEx. She has published more than 45 articles in robotics, computer vision, and neural networks. She has participated in 25 research projects and contracts in the fields of social, educational, and assistive robotics, active perception, and software engineering for robot programming. Her current research interests include robotics, computer vision, active perception, software engineering for robotics, and machine learning.

IVÁN BARBECHO was born in Monesterio, Extremadura, Spain, in 1994. He received the B.S. and M.S. degrees in computer science from the University of Extremadura, Spain, in 2017 and 2019, respectively.

From 2017 to 2019, he was a Research Assistant with the Robotics and Artificial Vision Laboratory (RoboLab), University of Extremadura. He is currently working with the Research and Development Department, Mobbeel Solutions S.L. He has been one of the core developers of the robot-agnostic educational programming tool LearnBlock since it was created. Also, he has been actively collaborating in the development of the open-source robotics framework RoboComp, since 2016. His research interests include deep-learning software and software engineering for robotics.

LUIS V. CALDERITA was born in Cáceres, Extremadura, Spain, in 1982. He received the B.S. degree in computer science from the University of Extremadura, Spain, in 2009, the M.S. degree in intelligent systems from the University of Salamanca, in 2010, and the Ph.D. degree in computer science from the University of Extremadura, in 2016.

From 2010 to 2012, he was a Research Assistant with the Robotics and Artificial Vision Laboratory (RoboLab), University of Extremadura. From 2012 to 2015, he was a Research Assistant with the Group of Integrated Systems Engineering, University of Málaga. In 2016, he founded the company PlayMyHit. From 2017 to 2018, he was an Assistant Professor with the Electronics Technology Department, University of Málaga. From 2018 to 2019, he was an Assistant Professor with the Department of Computer and Telematic Systems Engineering, University of Extremadura. He is currently a Research Assistant with RoboLab. His research interests include social robotics, human–robot interaction, and software engineering for robotics. He has been one of the core developers of the open-source robotics framework RoboComp, which has been selected six times by Google in its Google Summer of Code Program.

Dr. Calderita is a member of the Spanish Society for Research and Development in Robotics (SEIDROB). SEIDROB is a Sister Society of IEEE-RAS.

PABLO BUSTOS was born in Madrid, in 1966. He received the B.Sc. and Ph.D. degrees in computer science from the Polytechnic University of Madrid, in 1992 and 1998, respectively. In 2000, he joined the University of Extremadura as an Associate Professor in computer science. In 2001, he founded with other colleagues, the Robotics and Artificial Vision Laboratory (RoboLab), University of Extremadura, which has remained very active until now. During this stage, he has advised five Ph.D. theses in mobile robotics and social robotics. He has also participated in the design and construction of 20 autonomous robots. The scientific and technological areas in which he focuses his research currently include social robotics, artificial vision, and cyber-physical systems applied to intelligent spaces and precision agriculture. He has a total of 128 publications, of which 30 are international journals and 15 book chapters. He has coordinated 20 projects with regional, national, and international funding. He is one of the creators of RoboComp, a robotics development framework selected six times by Google in its Google Summer of Code program.

LUIS J. MANSO was born in Badajoz, Extremadura, in 1983. He received the B.S., M.S., and Ph.D. degrees in computer science from the University of Extremadura, in 2009, 2010, and 2013, respectively.

He has been one of the core developers of the open-source robotics framework RoboComp since it was created in 2005. From 2010 to 2013, he was a Research Assistant with the Robotics and Artificial Vision Laboratory (RoboLab), University of Extremadura. From 2013 to 2018, he was a Postdoctoral Researcher with RoboLab. Since 2018, he has been a Lecturer (Assistant Professor) of computer science with the School of Engineering & Applied Science, Aston University. He has authored more than 70 articles in international journals and conferences. His research interests include active perception, grammar-guided perception, human–robot interaction, and software engineering for robotics. In 2019, he became a Fellow of the Higher Education Academy (FHEA).