Robotica: (2021) volume 39, pp. 1116–1139. © The Author(s), 2020. Published by Cambridge University Press.
doi:10.1017/S0263574720000934

A Visual Programming Approach for Co-designed Robots

Andrés S. Vázquez†*, Tomás Calvo‡, Raúl Fernández† and Francisco Ramos†

†*School of Industrial Engineering, University of Castilla-La Mancha, Avda. Camilo José Cela, s/n, 13071 Ciudad Real, Spain.*
E-mails: andress.vazquez@uclm.es, raul.fernandez@uclm.es, francisco.ramos@uclm.es
‡*Mundo Reader S.L. (BQ), Calle Dublín 1, 28232 Las Rozas, Madrid, Spain.*
E-mail: tomas.calvo@bq.com

(Accepted August 15, 2020. First published online: September 23, 2020)

SUMMARY
This paper proposes an approach for the high-level programming of co-designed robots that reduces programming complexity. Particularly, the work presented focuses on the programming framework of an intelligent system, based on the IEEE Standard Ontologies for Robotics and Automation, which allows users the automatic design of robots and the automatic implementation of controllers in the Robot Operating System (ROS). In our approach, the co-designed robot functionalities are automatically translated into visual programming blocks allowing non-expert users an easy robot programming by means of a visual programming language. Several robot configurations and three case studies are provided as a proof of concept. The validation, in terms of usability, of the framework has been carried out with inexperienced users showing promising results.

KEYWORDS: Co-design; Visual programming; Robotic ontologies; Modular robots; Software usability.

1. Introduction

Robot programming is evolving to adapt to what is being called the *robot revolution* of the 21st century, which is very different from the first assembly like robot languages. In the industrial sector, for instance, robot programming began with a limited first generation of languages (e.g., SIGLA, ROL, etc.), after which there was a long period of stagnation in the second generation (e.g., VAL2, VAL3, RAPID, etc). It has, however, now moved to new visual programming suites, such as Robotstudio (ABB), and these new programming frameworks incorporate aspects such as the integration of computer-aided design (CAD) environments and *off-line* 3D teaching and programming. However, the greatest evolution in industrial robot programming is owing to the change in the applications in which manipulators are used. Since their invention, industrial robots have been used only for a small set of tasks related to high-volume production, as required by industry (e.g., welding, handling material and dispensing). They are now, however, starting to be used for lower volume, mixed-part production. This change is forcing industrial robots to be programmed using agile techniques. In this respect, the Robot Operating System (ROS) is making the tools developed in research available to industrial robots, thus allowing programmers to use high-level planners (e.g., for grasping or path planning, etc.), and, therefore, simplifying and speeding up the programming of complex robots. The growth of ROS-Industrial[1] could be identified as a consequence of this new robot paradigm.

* Corresponding author. E-mail: andress.vazquez@uclm.es

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

This *robot revolution* is also affecting education. Although educational robots have been used for the last three decades, mainly in extracurricular activities,[2] governments are now realizing that it is necessary to train society in this *revolution*. In this context, the programming of educational robots has proven to be a key tool for the teaching of Science, Technology, Engineering and Mathematics subjects. This programming has evolved from the first educational languages, such as Lego Logo[3] in the early 1990s, to more friendly and easier visual programming languages (VPL), such as Lego EV3 or Scratch.

VPLs can indeed be used to simplify robot programming. Despite being principally used for educational robots, some powerful industrial and research visual tools, such as LabVIEW or Matlab/Simulink, have recently integrated robot programming packages. In this paper, we summarize the state of the art of those VPLs that are applied to industrial, educational and research robots (see Section 2), denoting that the majority of VPLs allow only high-level programming (e.g., use of motion planners) for specific robot designs and not for new user designs. This absence of high-level planners for customized robots restrains the simplicity of VPLs: the fact that a robot platform has an easy programming language (e.g., Blockly or Scratch) does not imply that any robot created with this platform will be easy to program. This makes the programming of planners for complex robots (e.g., modular humanoids) beyond the abilities of novice users.[4]

In Section 3, we present our automatic co-design framework which is the basis employed for our approach for the VPL explained in Section 4. In fact, our high-level visual programming cannot be understood outside this co-design framework: we use the information generated during the co-design process to interface robot functionalities in visual programming blocks. This paper specifically presents a new VPL environment called Botbloq and focuses on robot programming issues, while an in-depth presentation of the intelligent co-designer is provided in previous works.[5,6] This work is a step forward as regards solving a complex problem: the automatic creation of robot controllers for a new robot, which is still far from resolved, and remains an open issue in robotics research. In fact, our work is limited to new robots co-designed from an existing set of dimensionless configurations, which are parameterized according to user requirements. In Section 5, we present some conceptual case studies of our approach in order to show how it could be used in education, research or industrial applications. Finally, in Section 6, we present an evaluation of the usability of our framework when employed by inexperienced users.

2. Visual Programming of Robots

VPLs represent a different paradigm from classic text-based languages in the methodology of programming. Rather than writing lines of code in a certain language whose syntax must be strictly observed, they provide intuitive visual programming tools (such as blocks and arrows). The learning curve is, therefore, drastically reduced, and the focus is on the description of *what do I want to do* by simplifying *how do I write it*. In the following, we present the most relevant VPLs classified by their scope of application (see Table I for a comparison of the existing VPLs with the framework *Botbloq* presented in this paper).

2.1. VPLs for industrial robotics

VPLs have been relentlessly introduced into the industrial robotics field owing to the need for agile techniques during the reconfiguration and reprogramming of robots by the end-user, not directly by the developer.

1. **RoboGlyph** was an early attempt to create a VPL that uses the programming by demonstration technique with a visual interface to command industrial manipulators.[7] Its objective is to effectively communicate the programmer orders to the robot by means of a sequence of primitives (e.g., robot positioning, contour following or peg in hole insertion) represented by descriptive icons. It was tested on a Puma-260 robot in order to discover its ability to perform manipulation tasks, such as opening a refrigerator door.
2. **Microsoft Visual Programming Language** is a companion for Microsoft Robotics Developer Studio (RDS),[8] which was created as an alternative to text-based programming with C#. It is designed on the basis of a graphical dataflow-based programming model consisting of activities

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

Table I. Comparison of visual programming languages (VPLs) OS: open source; Prop: proprietary; Educ: educational; Ind: industrial; Res: research; BI: high-level instructions for a fixed set of basic robot configurations only (e.g., differential robot); CI: high-level instructions for a fixed set of complex robot configurations (e.g., hexapod robot, six wheels configurations and humanoid robot); CC: able to program user-defined configurations.

Language	License	Type	High level	Low level	ROS	Robots
ArduBlocks	OS	Educ	BI	Yes	No	Arduino robots
Aseba Studio VPL	OS	Educ	BI	No	No	Thymio
Bloxter	Prop	Educ	BI	Yes	No	GoPiGo
BotBloq	OS	Educ/Res	BI,CC CI	Yes	Yes	Arduino robots, Makeblock robots, BQ robots
Choregraphe	Prop	Res	CI	No	No	NAO
Code Lab	Prop	Educ	BI	No	No	Cozmo robot
CodeZ	Prop	Educ	BI	Yes	No	Lego EV3 VEX IQ robots
Cubelets Block.	Prop	Educ	BI,CC	Yes	No	Cubelets
EZBuilder RoboScratch	Prop	Educ	CI,CC	Yes	No	EZ robot config. Roomba Sphero AR drone
KUKA-prc	Prop	Ind	CI	No	No	KUKA robots
Intera	Prop	Ind	CI	Yes	Yes	Baxter Sawyer
LabVIEW (Robotics)	Prop	Res	CI,CC	Yes	Yes	Arduino robots Industrial robots Clearpath robots iRobot creator Lego NXT,EV3
MATLAB Simulink	Prop	Res	CI	Yes	Yes	Lego NXT,EV3 Arduino robots
ModKit	Prop	Educ	BI,CC	Yes	No	VEX IQ robots Arduino robots
MOSS Scratch	Prop	Educ	BI,CC	Yes	No	MOSS
MSVPL	Prop	Ind	BI	Yes	No	Industrial robots Research robots
NEPO	OS	Educ	BI,CC	Yes	No	Lego NXT,EV3 NAO Calliope robot
Robot-Blockly	OS	Res	CI	No	Yes	Erle robots
RobotC	Prop	Educ	BI	Yes	No	Lego NXT,EV3 VEX IQ robots
Root	Prop	Educ	BI	No	No	Root robot
RoboFlow	OS	Ind	CI	Yes	No	PR-2
RoboGlyph	OS	Ind	CI	Yes	No	Puma-260
RobotStudio	Prop	Ind	CI	Yes	No	ABB robots
Robot Task Commander	OS	Res	CI	No	Yes	Robonaut-2 Valkyrie
Software EV3	Prop	Educ	BI	Yes	No	Lego NXT,EV3
Stäubli Robotics Suite	Prop	Ind	CI	Yes	No	Stäubli robots

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

(blocks) and connections (lines) to other activities. It can be used to program industrial or service robots within the RDS framework.

3. **RoboFlow** is a flow-based VPL[9] that resembles data flow diagrams, in which several different-shaped boxes are connected by lines that represent the flow of data among boxes, inputs and outputs. The language is kept simple and high level, thus providing intuitiveness, while it is sufficiently expressive to capture useful tasks. The task domain comprehends most usual organizing tasks, such as pick up, put away or tidy up, for objects contained within a known environment. It has been tested on *PR2* robots so as to ascertain its capacity to carry out manipulation tasks, such as stacking cups or taking objects from drawers.
4. **Intera Software**[10] is a commercial visual programming environment that includes a VPL designed for Baxter and Sawyer collaborative robots from Rethink robotics. Intera allows the graphical programming of the robot actions, including advanced features such as force control, computer vision and real-time data monitoring, the simulation and fine-tuning of the robot tasks with Intera Studio, and the definition of custom tasks or movement patterns with its train-by-demonstration interface.
5. **KUKA-prc (Parametric Robot Control)**[11] is built upon the visual programming system Grasshopper, which is a part of the CAD Rhinoceros 3D software.[12] It is a commercial product designed to make KUKA industrial robots accessible to the creative industry by using visual programming blocks.
6. **RobotStudio**[13] ABB's simulation and offline programming software allows robot programming without shutting down production. The programming is visual and is performed in the simulation environment in a similar way to that which occurs with a CAD program.
7. **Stäubli Robotic Suite**[14] performs realistic simulations and allows the offline programming of Stäubli robots. The robots are programmed in the simulation environment by entering the instructions in a similar way to a CAD program.

2.2. VPLs for robotic research

In the last few years, some of the best-known and most frequently used research platforms have also included libraries for the graphical programming of robots:

1. **LabVIEW**[15] is the VPL that is most widely used in the industry. It is mainly designed for SCADA systems but also includes a fairly complete and advanced package for robotics. At the low level, it can virtually program any robot, as any communication protocol can, if necessary, be programmed with NI hardware, but there are already drivers for many hardware platforms such as Arduino. At the high level, it provides blocks for complex maneuvers for manipulators or path planning for mobile robots. A complete description of the robot is required in order to use these functionalities. Moreover, third-party packages have been developed by LEGO, Kuka, Fanuc, Stäubli, Schunk, etc., to enable the easy use of their robots. Furthermore, Tufts University has created an API for LabVIEW for ROS clients.
2. **MATLAB/Simulink**[16] is one of the most widespread research platforms in the world. MATLAB is a text-based programming framework but includes the Simulink Toolbox as a *blocks_and_arrows* VPL used for data acquisition and control systems among a long list of applications. Simulink has functionalities that allow it to communicate with external hardware and control it and, therefore, provides low-level programming for any robot as long as the communication protocol is known. For instance, it provides libraries with which to connect with Arduino or Raspberry Pi hardware platforms. On the high-level programming side, MATLAB allows the use of ROS nodes, once the robot configuration has been determined. In fact, ROS packages can be created from a Simulink model. On the downside, these nodes are quite limited in Simulink programming and it is usually necessary to add some text programming in order to create advanced functionalities. LEGO and Turtlebot are also supported by Simulink.
3. **Choregraphe Suite**[17] Choregraphe is a multi-platform desktop application for NAO robots that makes it possible to create animations and behaviors, test a simulated robot or monitor and control NAO. Choregraphe allows the creation of very complex behaviors without writing a single line of code.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

2.3. VPLs for educational robotics

In addition to industrial or research applications, VPLs are an excellent choice for education, and not only for programming,[18] as has been proved with the proliferation of VPLs based on Scratch,[19] but also for robotics. Many platforms and VPLS have been developed for educational purposes within the last 10 years. Some of those that provide a perspective as regards the VPL presented in this paper are listed below:

1. **LEGO NXT/EV3**.[20] The pioneer VPL in educational robotics is probably LEGO, with its NXT-G language based on LabVIEW, which has evolved to EV3[21] in the last few years. It is a *blocks_and_arrows* type of language that allows the low-level programming of each active component (i.e., actuators and sensors) and a few high-level commands such as direction movement of differential drive robots. It is proprietary software of LEGO but is free of charge.
2. **Aseba Studio VPL**[22] is a *shapes_and_colors* language that provides a set of high-level functionalities for the Thymio robot. It abstracts the Aseba text language in order to provide an intuitive interface. The VPL has no low-level commands for individual actuators, but they can be accessed using the text language.
3. **EZ Builder**[23] is the integrated development environment (IDE) employed to program the EZ robot platform and other robots like Roomba or Sphero. Internally, it uses both Blockly and RoboScratch VPLs and allows the programming of a variety of robot configurations. The preexisting robots can be programmed either at a low level, by accessing sensor data and commanding motors, or at a high level with existing complex tasks such as 'walk', 'turn' or 'follow object'. An important feature of EZ Builder is that its allows users to design and program new robots in a visual application called *robot design*. This software includes an open on-line library of active and passive elements that developers can use to build (and share) new configurations. Complex actions in new robots are programmed by the concatenation of *frames* that users must defined in the application.
4. **RobotC**[24] is a C-based robot programming language created at CMU. It has a graphical feature that allows the drag and drop of commands into the programming area but has eventually been used as a text-based language with a friendly user interface. LEGO and VEX platforms can be programmed with RobotC at a low level and also have some simple behaviors for differential drive robots such as 'forward', 'backward' and 'turn'.
5. **ModKit**[25] is a Blockly-based VPL designed for VEX platform, although it can aditionally be used with a set of microcontrollers such as Arduino. It provides blocks for low-level programming, but the robot configuration can also be defined to make use of certain high-level movements.
6. **CoderZ**[26] is an online learning environment for the programming of virtual robots using a Blockly-based VPL or, for experienced programmers, with the direct use of JAVA. It is not intended for the programming of real robots but rather for learning in the 3D simulator it includes. The robots provided with the simulator cover a number of platforms from SnapDragon to LEGO.
7. **NEPO**[27] (or Open Roberta Lab) is a Blockly-based VPL with which to program several different platforms such as LEGO or NAO. In the case of LEGO robots, it provides a robot configuration page on which the user can parameterize a differential drive robot (wheel diameter, track width, etc.). This allows users to use parametrizable high level actions, such as moving a distance at a certain speed (which obviously depends on the wheel diameter). As with CoderZ, it runs on a web browser.
8. **Bloxter**[28] Bloxter is a new graphical programming language based on the Blockly that is employed to learn coding with robots. It was specially designed for GoPiGo robot over DexterOS.
9. **Cubelets Blockly**[29] Cubelets Blockly is a free coding environment for the Cubelets robot blocks, based on the Blockly VPL originally developed by Google.
10. **MOSS Scratch**[30] MOSS Scratch is an extension for Scratch that enables programming support for the MOSS robots. Basically, Moss has added 5 blocks to Scratch that allows programmers to send or receive values (a number from 0 to 100) from the real robot blocks.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

11. **Root**[31] is a robotic platform with a programming IDE that provides three different levels, depending on the user's knowledge: a VPL with blocks for basic functionalities; Blockly-based VPL, in order to provide some advanced flow control, and full-text programming in either Python, JavaScript or Swift. It was developed at Harvard and funded with Kickstarter.

2.4. VPLs for ROS

The expansion of VPLs in the robotics domain is already reaching ROS, one of the most widespread frameworks in the research domain. Several efforts have been made to provide intuitive IDEs that simplify the use of ROS packages, either for the automatic creation of launch files[32] or for programming purposes, such as the discontinued ROS from Scratch.[33] Some of these initiatives are:

1. **Robot_Blockly** is a ROS package created by Erle robotics that provides web-based visualization and block programming tools for robots and drones. It is based on Blockly and provides high-level blocks with which to perform basic features, such as the direction of movement of a rover, and advanced features, such as SLAM. Predefined blocks exist for the different Erle robots, but it also allows the definition of new blocks using ROS features, which is a very powerful tool.
2. **Robot Task Commander**[34] is a framework for the definition, development and deployment of robot applications for use in different run-time contexts using a VPL that defines the state machine of the application and the participant ROS nodes. It has been applied to advanced robots such as Robonaut-2 and Valkyrie.

2.5. Discussion

VPLs are well established in robot programming, from industrial to research applications, educational robotics being the field where there are more variants. Many of the VPLs help the programming of commercial robots by making available high-level preprogrammed actions (e.g., walk behavior of the NAO robot in Choreographe). Some VPLs help with the programming of few parameterizable robots by including parameterizable kinematic functions (e.g., kinematics of manipulator robots in Labview or differential LEGO robots in NEPO). Some of the VLPs also help with the design of new robot configurations (e.g., EZ Builder), but the programming of the behaviors of these robots must be done by the user at low level. In this work, we propose an approach (BOTBLOQ) that is based on blocks like many of the existing VPLs (e.g., Scratch, EV3 and Blockly). Our approach, like other VPLs, allows users to create new robots. However, in our approach, users do not design robot configurations, users just specify the list of behaviors and robots and controllers are designed automatically by a co-designer. Also, as a difference to the existing VPLs, Botbloq makes available high level behavior functions for complex parameterizable robots like humanoids, snakes and hexapods.

3. Botbloq Co-design System

In order to explain our approach for the high-level visual programming of co-designed robots, it is necessary to introduce Botbloq. Botbloq is a framework developed with the aim of simplifying the design, manufacturing and programming of modular robots for both educational and research purposes. Our approach is comparable to the *robot_blocky* approach as regards the way in which we use block programming with ROS. However, we do not limit the high level use of planners to a set of specific robots but rather to a set of parameterized robot configurations. In this respect, our approach is more similar to other platforms such as *NEPO* or *LabVIEW*.

Figure 1 shows the robot creation scheme followed in Botbloq, which is summarized in the following items:

1. A software interface allows the user to select the skills that the robot should have.
2. A knowledge system extracts the set of structural robot parts needed to perform user's requirements (e.g., legs for walking or hands for grasping) from a robotic ontology.
3. The ontology is searched to find the dimensionless robot configuration that includes the set of structural parts found previously.
4. A parameterization process selects the active modules (actuators) and designs the passive modules (structures) according to kinematic and dynamic considerations specific for each configuration. The user can now build the robot modules using a 3D printer.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

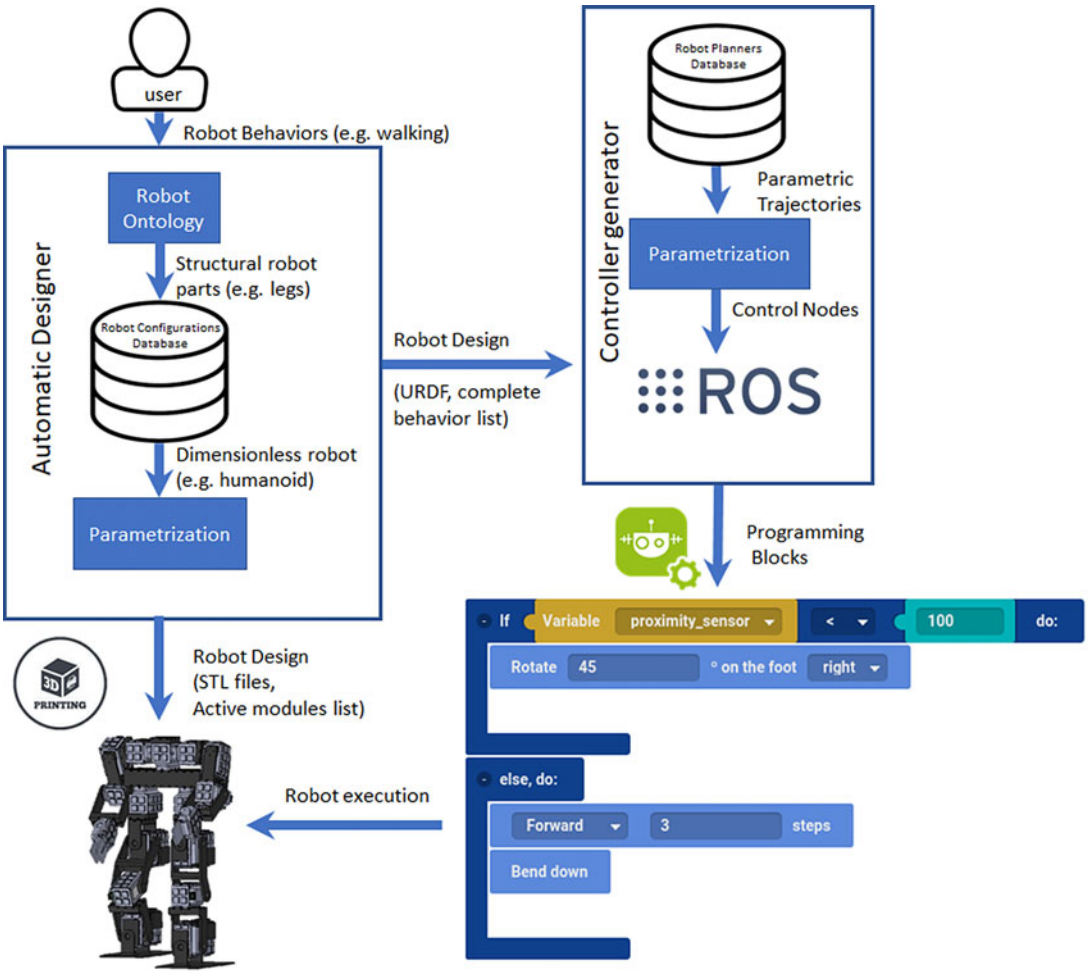


Fig. 1. Botbloq global scheme.

5. The formal description of the robots is represented in ROS using Unified Robot Description Format (URDF) models.[35]
6. The robot controller is obtained by carrying out a parameterization process for the dimensionless controllers stored in a database. This is performed by adding the data of the URDF to the controller.
7. An ROS service is created with each planner. An experienced user could use the ROS API to the robot.
8. Visual high-level blocks (i.e., planner functions) are generated for each ROS service planner, thus allowing users to robots with a VPL (Bitbloq).

Note that there are three main aspects in Botbloq: first, the robot generation, which is based on the use of a robot ontology and some user-provided parameters; second, the controller parameterization, which is obtained from the URDF model; and finally, the visual programming blocks, which are visual representations of the ROS controllers. The first two aspects have been presented and detailed in previous works,[5,6,36] but they will be briefly introduced in the following subsections for completion purposes. The visual programming blocks, however, will be thoroughly described in Section 4.

3.1. Ontology-based robot generation

One of the most important characteristics of our system is that it allows users to create new robots by using a knowledge agent. This agent is based on Automatic Design of Robots Ontology (ADROn),[5] which is built upon the IEEE Standard Ontologies for Robotics and Automation.[37] According to

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

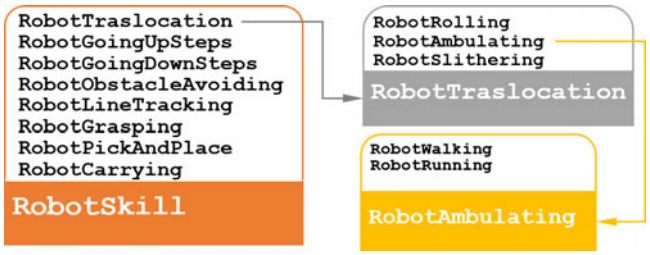


Fig. 2. List of robot skills addressed by actual state of ADROn.

ADROn, a robot consists of one or more `StructuralRobotParts`, and each of them includes a set of `Modules`.

The conceptual generation of a robot is a three-step process:

1. The robot instance generator receives a set of `RobotSkills`, selected from those of Figure 2, that the robot is required to perform (skills that the robot possesses) and the inference engine uses semantic queries in ADROn to determine the `StructuralRequirements` implied by the set of skills.[36]
2. The instance generator matches these requirements with the hardware available in the database of dimensionless robot configurations. If several matches are found, the generator searches (downwards first, upwards if necessary) in the taxonomy tree of the ontology for structural differences or different skills between these matches. Then, it infers some questions for the user in order to disambiguate the solution (e.g., if `RobotWalking` is a desired skill, both `HumanoidRobot` and `HexapodRobot` would meet that requirement, then the question would be if the user prefers a robot with two or six legs). If there were no differences, the generator would directly request the user to choose the preferred configuration.
3. The system creates the conceptual (instance) design of a robot that is able to perform those `RobotSkills` and passes it to the structure generator, which requests some physical parameters from the user and employs dynamic analysis in an iterative process to determine the robot actuators and the geometries of the links (i.e., the mechanical design). For example, in the case of a `RobotManipulator`, the payload parameter allows us to calculate the motor torque required and select adequate actuators, while the workspace provides a limitation for the link lengths.

Both actuators and links are physically implemented using our Parametric Modular Robotic Architecture (ParMoR). In fact, our designer system generates the robot design in ParMoR, which means that blueprints for each part of the robot (i.e., electronics, structure and actuators) are provided to the user, thus allowing the robot to be manufactured. Complete information about the robot design and its implementation with ParMoR can be found in previous works.[5,6]

Please note that ParMoR is not intended to be used in industrial environments, as the performance of the robots generated with ParMoR would not satisfy industrial requirements (e.g., durability, precision and repeatability). However, we consider that our approach could be of use in industry if we employed other modular robotic architectures, such as the modular robotic architecture for industrial manipulators developed by Schunk.[38]

3.1.1. Controller selection. In this work, the relations between skills and the controllers that are required to perform them have been included in ADROn in order to leverage the co-design problem, as we choose not only the abstract robotic structure but also their abstract controllers. The specific design for each controller is carried out after this selection process (Section 3.2). Two examples are provided below:

- a `WheeledRobot` with two wheels and the ability of moving (skill of `RobotTranslocation`): ADROn states the need for a `RobotDiffDriveController` that manages the actuators.
- an industrial robot (`RobotManipulator`) with the ability of kitting (skill of `RobotPickandPlace`): ADROn states the need for a `RobotPoseController` and a `GripperController` with which to perform the pick and place action.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

The complexity of the controllers grows rapidly if we add constraints or more advanced skills to the robots. For example, we could request the `WheeledRobot` to have the ability to move among different places on a `Map` (namely `RobotTravel`), which would imply the need for a `PathPlanner` and an `ObstacleAvoidingController` to navigate through the map. We could alternatively add constraints to the maneuvers of the `RobotManipulator`, such as the existence of obstacles in the path of the maneuver trajectory. The `PathPlanner` and the `RobotManipulator` constrained motion, for instance, lead to complex problems that are not supported in this work yet. The latter has been addressed in separate works with similar (ontology-based) approaches.[39,40]

3.2. Controller generation

The robot URDF file and the abstract controllers are then used by the system to create the specific structure and controllers for the robot designed. The controllers are obtained from a database developed according to the set of available robot configurations (i.e., rovers, manipulators, snakes, hexapods or humanoids). A summary of the parametric controllers developed for Botbloq is introduced below:

- Parametric controllers for wheeled robots. Two configurations of wheeled robots are included in our configuration database: differential and skid-steer kinematics. In both configurations, the radius of the wheels and the distance between them are the customized parameters. We use the Xacro language[41] for the automatic generation of URDF files from the `StructuralRobotParts` (i.e., wheels) that are repeated in the robot instance created with ADROn. This URDF is used to automatically generate the controller with the `diff_drive_controller` ROS package.[42]
- Parametric controllers for robot manipulators. The `StructuralRobotParts` of these robots are the joints and the links. In our framework, we deal with manipulators whose parameterization depends on the number of links (i.e., 3 for position or 6 for position+orientation control) and their lengths. We again use Xacro to generate the URDF. Once the URDF has been defined, we generate the controller with the **MoveIt!**[43] ROS tool. As a result, the controllers generated can drive any scaled version of a manipulator.
- Parametric controllers for snake robots. The number of modules, M, of the snake (from $M = 3$ to $M = 14$) is the main customized parameter of this type of robot. We have chosen Xacro, as it is the most efficient option for the automatic definition of the URDFs. A parametric controller for any M modules snake robot has, therefore, been programmed and added to the controller database. These parametric controllers consist of phase-shifted sinusoidal references. Several types of movements (depending on the size of M) can be performed by changing the parameters of the sinusoidal references[44] (e.g., $M = 3$ snakes can perform only rolling movements while $M = 14$ snakes can carry out any kind of movement).
- Parametric controllers for humanoid robots. These robots are defined by their various *Structural parts* and, in this case, the length of their passive modules (e.g., femur and tibia) can be parameterized. The kinematic chain of these robots is more complex than in a manipulator and **MoveIt!** cannot be used to generate the controller. In this case, we use KDL[45] to generate the joint references from the desired movement path.
- Parametric controllers for hexapod robots. Our automatic designer uses a parametric rectangular hexapod as a base configuration. Leg lengths and actuator torques are parameterized in order to ensure that the hexapod satisfies user requirements (i.e., a hexapod that can avoid obstacles of height h). Body links also change in order to adjust the area that can be reached by each leg (thus avoiding self-collisions). We have developed a motion controller in ROS that generates tripod, ripple and wave gaits for any hexapod obtained from the base configuration. Basically, the kinematic model for each leg is modeled with Xacro, converted to URDF and used with the KDL ROS package in order to obtain, via inverse kinematics, the joint trajectories for each gait.

4. Visual Programming of Co-Designed ROS Controllers

Our visual programming system relies on an existing visual programming platform denominated as Bitbloq,[46] which has been adapted to the parametric ROS controllers presented in the previous section.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

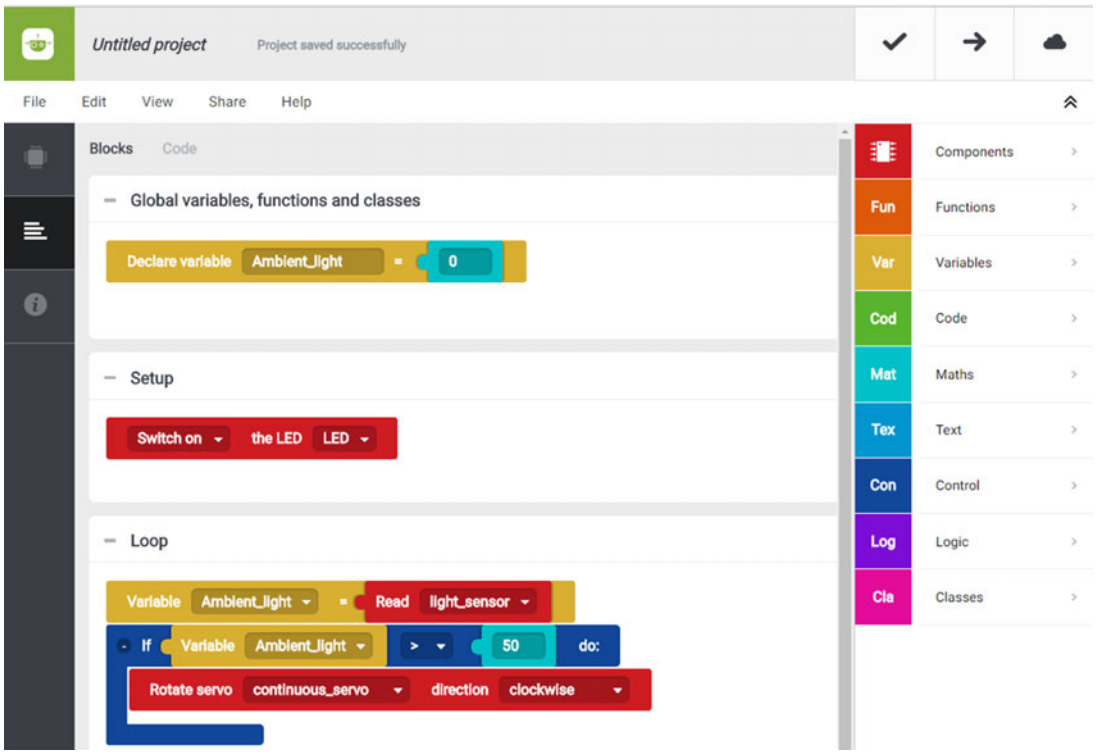


Fig. 3. Image of Bitbloq IDE.

4.1. Introduction to Bitbloq

Bitbloq is an open source tool for the online visual programming of arduino-based electronic projects (see Fig. 3). Bitbloq was developed by the BQ company to be used principally in education. Bitbloq is similar to other block-based programming languages such as Scratch, with the difference that Bitbloq programs are executed directly in the controllers. Bitbloq generates blocks by using its own open software library (https://github.com/bq/bloqs), which is based on HTML5, CSS3 and JavaScript. The main attributes that define blocks in Bitbloq are *type*, *connectors* and *contents*. These attributes have, as is shown in the following section, been adapted to be used with the co-designed ROS planners.

Following the same philosophy of Scratch and Lego, Bitbloq reduces the syntax errors as blocks only fit in ways that make sense. The command set of the basic Bitbloq includes 71 blocks. These basic blocks are the visual representation of processing instructions. In our modified Bitbloq, we have included 83 basic blocks, that represent the main Python instructions, and 22 actions blocks that are presented in the next section. In particular, our modified Bitbloq has the following features :

- Loops and control structures inherited from Python (i.e., if-else, while, for-in).
- Dynamically typed. As with Python, Bitbloq is not necessary to declare variable types, supporting the same types (e.g., numeric types, strings, lists, etc.).
- Object oriented. Our Bitbloq version, unlike the original Bitbloq, contains the necessary blocks for creating classes and instance objects.
- User-defined functions. As with Python, Bitbloq allows users to define their own functions. These functions may take arguments and may return a value. However, in our actual Bitbloq version, lambda expressions or higher-order functions are not allowed.
- Concurrency. The actual version of Bitbloq does not support concurrent programming, although the robot controllers are executed in independent threads using the ROS middleware.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

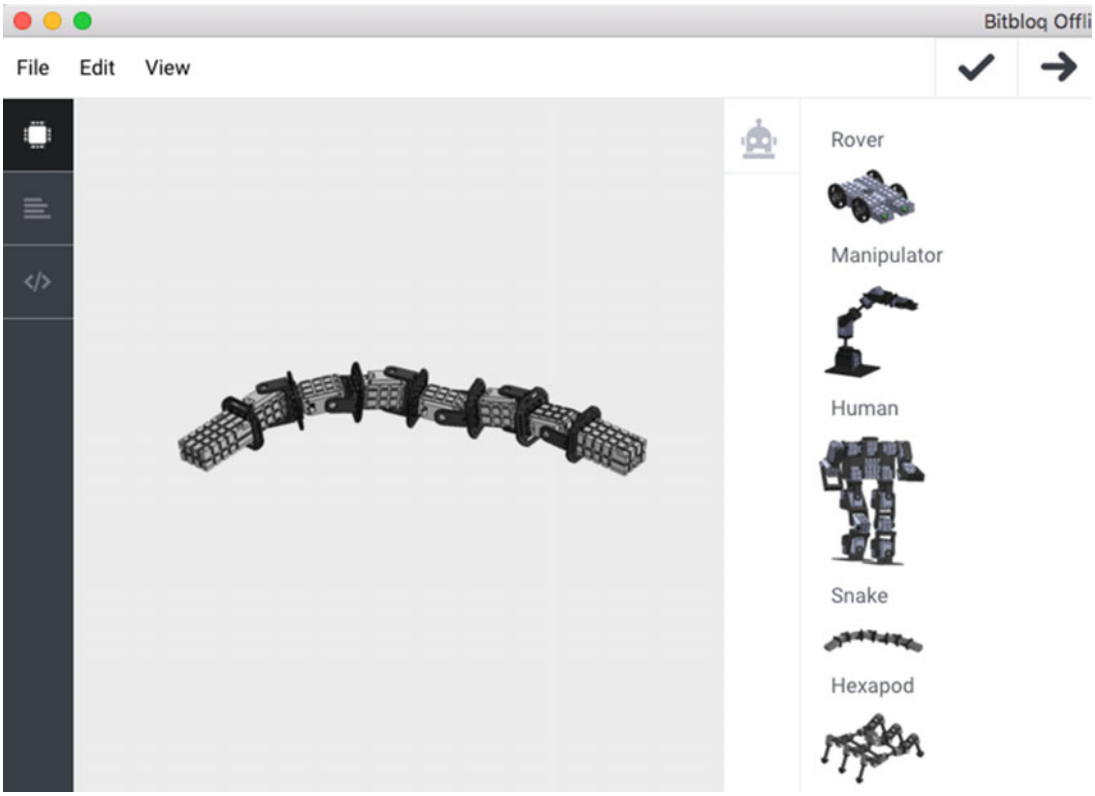


Fig. 4. Botbloq robots that can be programmed in Bitbloq.

4.2. Generation of visual blocks for robot actions

The process of generating Bitbloq blocks with which to interface the co-designed ROS controllers presented in Section 3.2 has two main stages:

1. Development of a visual IDE.
   Bitbloq IDE has been improved by adding the five types of parametric configurations that can be co-designed to date (see Fig. 4). One important characteristic is that the user needs merely to select one of the robots obtained after the co-design process, regardless of its real physical dimensions.
2. Generation of high-level action blocks for robots
   A visual block has been created for each robot skill defined in ADROn. These blocks include the Python code required to subscribe to the ROS service at the co-designed controller running in the robot. This implies that the code generated from the visual blocks is compact and independent of both the robot configuration and the implementation of the controller. The visual blocks created to date are presented below:
   - Rover action blocks. This is composed of two blocks as is shown in Fig. 5 :
     (a) Stop action.
     (b) Move action (forward, backward, left and right).

     Figure 6 shows the Python code generated with the block `move`. As can be seen, the method `move` from the `BotbloqVehicle` class is called. This method includes the subscription to the ROS service `move` that runs in the robot controller (e.g., the parameterized controller for the wheeled robot mentioned in Section 3.2).
   - Manipulator action blocks. This is composed of six blocks (see Fig. 7):
     (a) Cartesian movement. This allows the user to move the robot in a Cartesian position. The internal controller, using **MoveIt!** (and as explained in Section 3.2) will compute

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

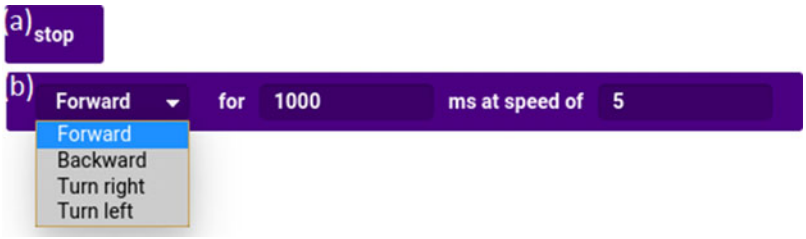


Fig. 5. Rover action blocks.



Fig. 6. Rover *move* action in python.

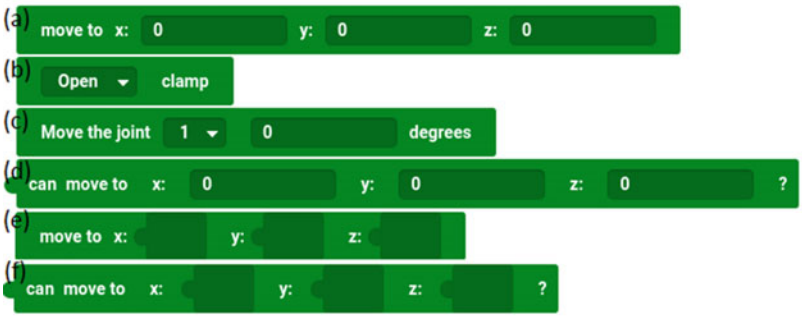


Fig. 7. Manipulator action blocks.

the inverse kinematics of the robot in order to obtain the joint values for this Cartesian position.

(b) Gripper open and close. This closes/opens the gripper of the robot.

(c) Joint movement. This allows the user to move each joint of the robot .

(d) Inverse kinematic solution. It tells the user whether there is an inverse kinematic solution for a Cartesian position.

(e) Advanced Cartesian movement. This allows the user to move the robot into a Cartesian position defined by `statement-input` blocks.

(f) Advanced inverse kinematic solution. It tells the user whether there is an inverse kinematic solution for a Cartesian position defined by `statement-input` blocks.

The aforementioned action blocks are methods from the `BotbloqManipulator` class that subscribe to the controller services in ROS. An object of this class is created when a user drags and drops a manipulator into the *robot window* in the Bitbloq IDE.

- Snake action blocks. We have implemented blocks for the different types of gaits that the parametric snake controller (see Section 3.2) can perform. However, these blocks will produce motion only if the snake configuration allows the movement to be statically stable[47] (i.e., there are sufficient modules in the snake). The blocks (see Fig. 8) and their suitable snake configurations are presented as follows:

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press



Fig. 8. Snake action blocks.

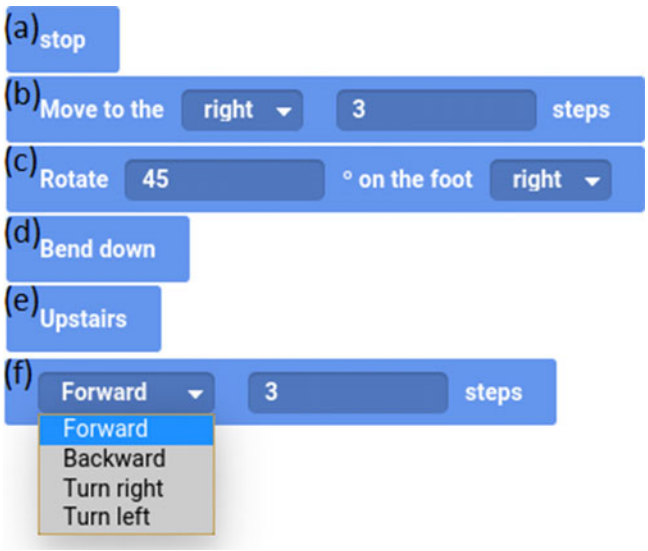


Fig. 9. Humanoid action blocks.

(a) Snake basic movement: Forward and backwards. This block produces linear displacement with snakes with $M >= 5$ modules.
(b) Stop. This block stops any movement.
(c) Lateral movement. This block produces side winding maneuvers for snakes with $M >= 6$ modules.
(d) Roll movement. Rolling movements for snakes with $M >= 3$ modules.
(e) Rotation movement for snakes with $M >= 5$ modules.

When a snake robot is selected in the IDE of Bitbloq, the code library creates an object in Python from the `BotbloqSnake` class. Each block is associated with a method in that class. Internally, these methods subscribe to the ROS services of the gait motion planners.

- Humanoid action blocks. Humanoid robots are the most complex of all robots and can, therefore, carry out a number of different actions. We have implemented the following (Fig. 9):
  (a) Stop any movement.
  (b) Lateral movement to the left and right.
  (c) Rotation on one foot to the left and right.
  (d) Bend down.
  (e) Climb stairs.
  (f) Basic movements: forward, reverse, right and left turn.
- Hexapod action blocks. As stated in previous section, our hexapod can perform three different gaits: tripod, ripple and wave. This means that, for example, a rectilinear movement can be achieved with three different motion planners (e.g., the robot controller will use wave locomotion when a high payload is demanded, as five legs are always resting on the floor with this

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

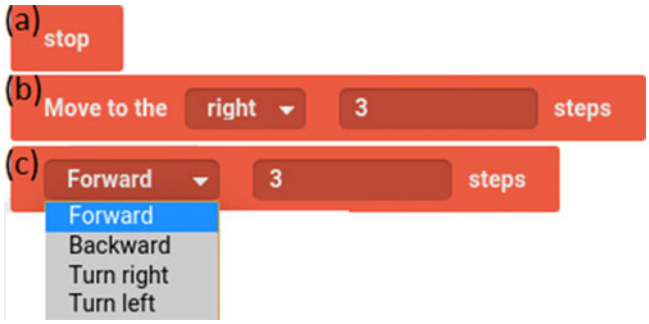


Fig. 10. Hexapod action blocks.

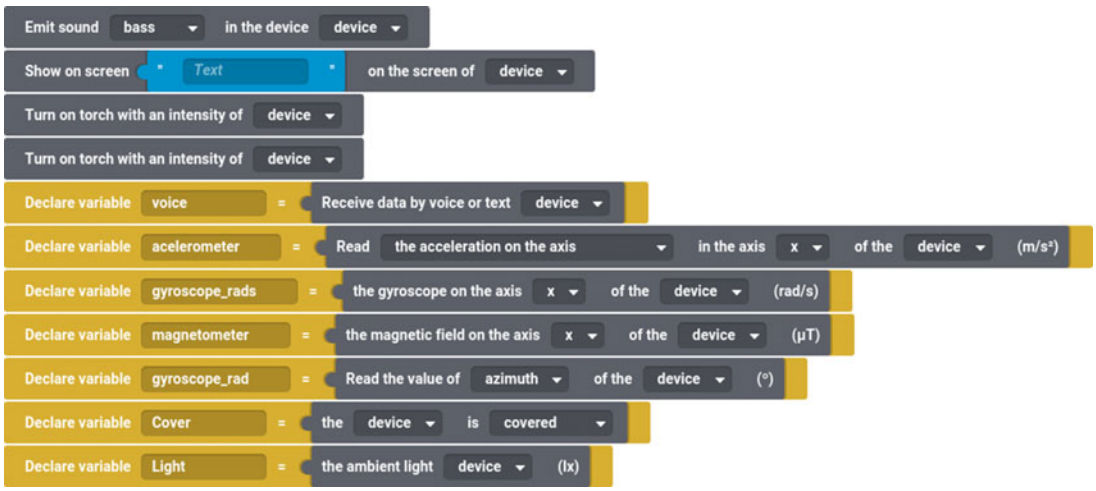


Fig. 11. Bluetooth blocks.

kind of motion). However, our framework allows the hexapod action blocks to abstract from the motion planner (i.e., the ROS services to which they subscribe share the same interface). The blocks included in Bitbloq to date (Fig. 10) are the following:

(a) Stop any motion.
(b) Side walk movement (left or right).
(c) Basic movements: Move forward and backwards and turn right or left.

As occurs with the other robots, once a hexapod robot has been selected in the Bitbloq IDE, an object from the `BotbloqHexapod` class is created. Each hexapod action block will correspond to a call to the related class method.

4.3. Android devices integration

One interesting characteristic of Bitbloq is that it allows Arduino controllers to be used together with Android devices. We have modified Bitbloq in such a way that the same Android devices can now be used together with the robot ROS controllers.

As Botbloq ROS controllers run on Linux machines, we have used the official Linux Bluetooth stack (*BlueZ*) to implement a Python class that communicates with Android devices. This is achieved by emulating a serial communication using the wireless Bluetooth serial profile. The Android devices must have the APP *Bitbloq Connect* installed, which provides functionalities such as data-like voice commands and gyroscope or accelerometer measures. This enables the use of Android blocks to program the robot (see Fig. 11).

4.4. Discussion

One of the best features of a block-based visual language, like Bitbloq, is that syntax errors are reduced. However, this does not mean that errors are completely eliminated. Users must still program

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

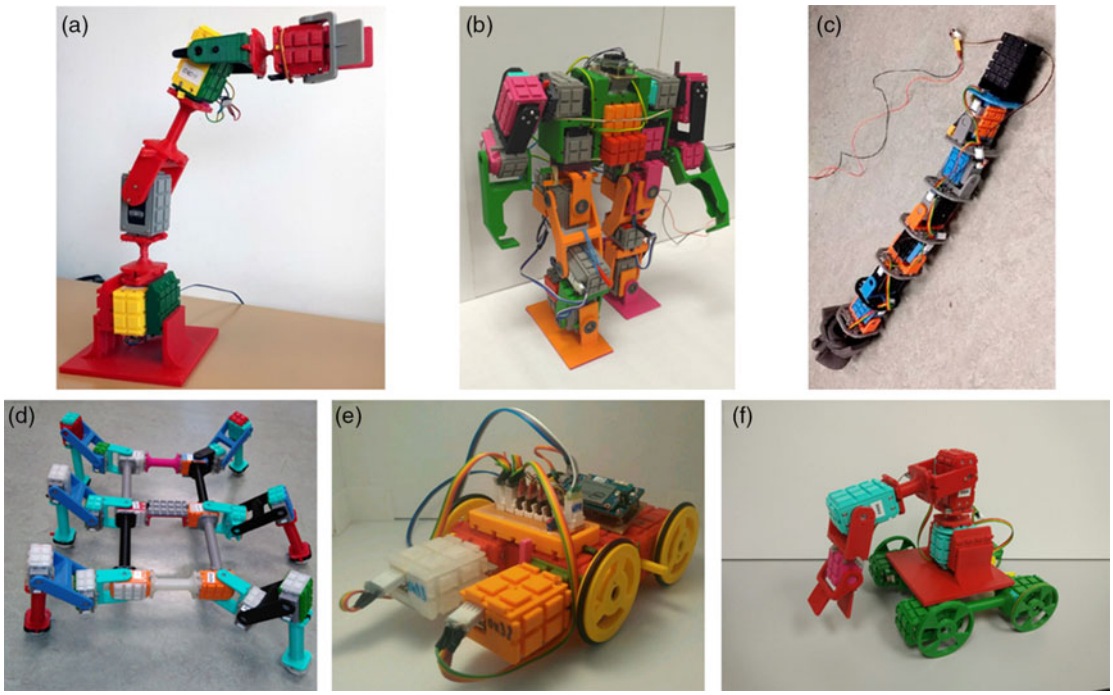


Fig. 12. Examples of robots that can be programmed with Botbloq: (a) manipulator; (b) humanoid; (c) snake; (d) hexapod; (e) rover.

carefully to avoid some other errors. For example, as there is not type declaration, users can use wrong variables when *writing* programs, getting mismatch errors in block types. In fact, this is an *ambiguity* problem analyzed in Section 6.1. Another feature of any visual language, like Bitbloq, is the command set suitability. A big command set would need a large command palette, consuming screen space and making it harder for the user to find a given command on the screen. On the contrary, a small command set could make it difficult for the user to find a combination of instructions that solve a problem. Therefore, in Section 6.1, we have analyzed the suitability of the command set of Bitbloq by analyzing how easy it is to find a giving command and the facility to find a solution of a problem with the available blocks. Concurrency is one of the main missing features of the actual development of Botbloq. Although ROS controllers are executed in parallel, users cannot program concurrent skills as it is possible to do in other visual languages like Scratch. This is a main problem to solve in future versions.

5. Case Studies

In the following, we present some case studies with the programming of various robots co-designed from the Botbloq framework (see Fig. 12). These robots have been built using the ParMoR modular robot architecture, whose structural parts are 3D printed and whose modules can be attached to one another using dovetail pins (see Fig. 13). The electronic parts are based on a main controller (Intel Edison or Raspberry Pi 3), which runs Botbloq ROS controllers, and several active modules (i.e., actuators and sensors) with AVR controllers (i.e., ATtiny85s) which communicate with the main controller via I2C. Figure 14 shows the scheme of the resulting architecture. Three examples of these robots are presented below.

5.1. Manipulator

In this first example, we present the programming of the manipulator robot depicted in Fig. 12(a). This robot was co-designed in Botbloq by following these steps (and as explained in Section 3.1): First, the user must employ Botbloq GUI (see Fig. 15) to describe the function to be performed by the robot. The program then infers the ontology and proposes the matching robot configurations. The user must subsequently select one of the possible configurations. The program will request the desired specifications from the user (e.g., robot speed, payload and workspace). The information

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

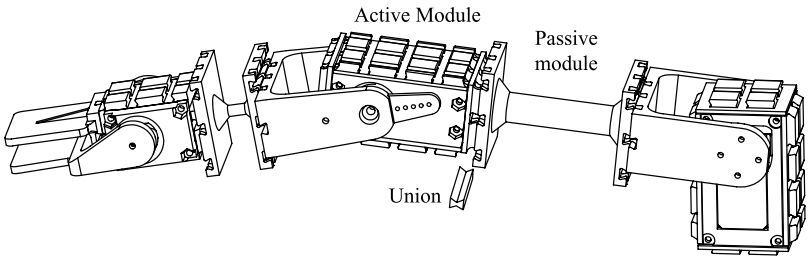


Fig. 13. ParMoR structure scheme.

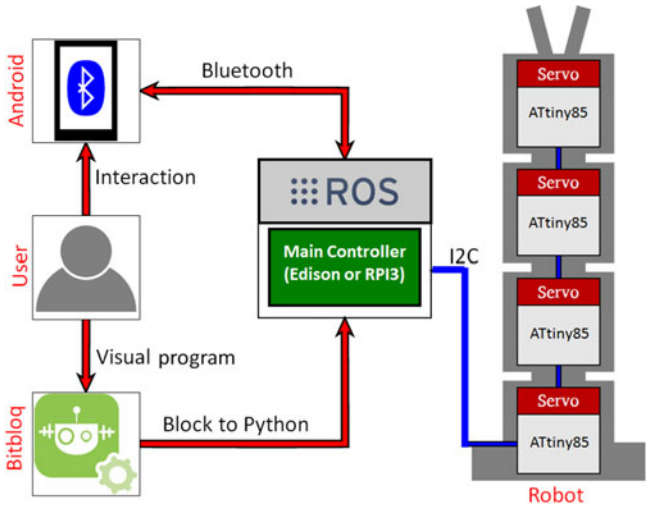


Fig. 14. ParMoR hardware architecture.

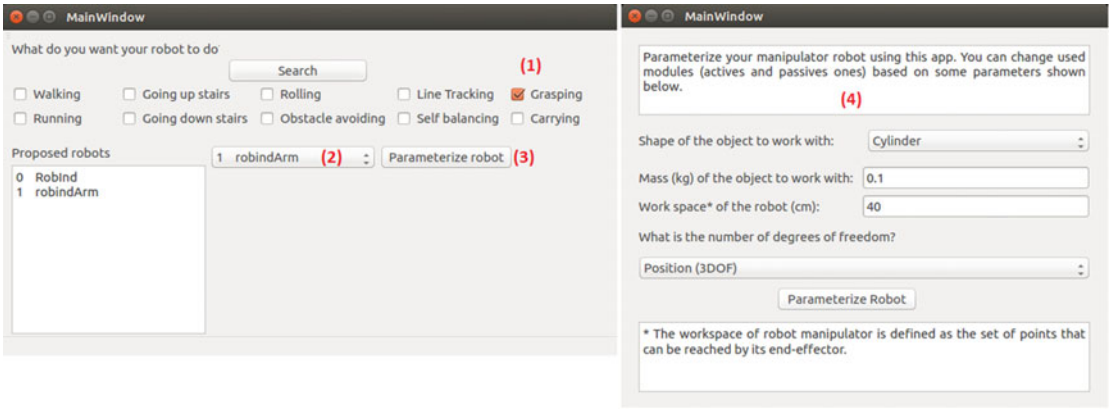


Fig. 15. GUI for the automatic design of robots using ADRON.

collected is then passed to the automatic designer which selects the necessary active and passive modules according to kinematic and dynamic considerations.

The designer will also generate the ROS controllers which will be interfaced with the visual blocks in Bitbloq. Note that the controller is generated by the parameterization of existing dimensionless controllers with user requirements, as explained in Section 3.2.

Figure 16 shows a demonstrating program implemented with the visual blocks generated from the manipulator actions. This program allows users to employ Android's voice recognition, to command the robot to pick up an object in a known position. As can be deduced, this program uses the inverse kinematics and trajectory planners of the co-designed ROS controller running in the robot. Once the robot has picked up the object, a text message is shown on the screen of the Android device.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press



Fig. 16. Example of Bitbloq program for a manipulator.

5.2. Rover

The second example corresponds to the four wheel mobile robot shown in Fig. 12(e). As occurs with the previous robot, this robot is obtained by means of the co-design process proposed in this work.

We have programmed the classic line follower as a demonstrator of the visual blocks for this configuration of robot. In a typical low-level programming (e.g., Arduino), the programmer must define each wheel movement (i.e., clockwise or anticlockwise) according to the desired movement. However, as can be seen in the block code (Fig. 17), with bitbloq programmers it is necessary only to select one of the available movements provided by the robot ROS controller.

5.3. Hexapod

The third example is the block programming for the hexapod robot shown in Fig. 12(d). In this example, together with the voice control of the robot, we show how Android device sensors (e.g., accelerometers) can be used to measure robot movements. The mobile or tablet device must be placed on top of the robot for this purpose. The code presented in Fig. 18 shows how the user can employ a voice command to order the robot to turn right or left. Once the robot has rotated, it transmits the increase in the azimuth measurement to the android device. As explained in the previous section, motion blocks use the ROS controllers running in the robot.

6. Validation

As stated previously in this work, one of our main motivations is to reduce the complexity of robot programming. Therefore, and owing to the nature of our framework (i.e., a block-based VPL), our system is particularly suitable for education in the same way that other similar visual programming platforms are (e.g., Scratch). In fact, the Botbloq framework has been satisfactorily used by the Educational Department at BQ in several *extracurricular* workshops with students between the ages of 10 and 15. Although the pedagogical validation of Botbloq is the subject of some ongoing research publications, it is possible to find more information about its validation with undergraduate students on the webpage of the project (https://botbloq.bq.com/) and in its github repository (https://github.com/bq/botbloq).

Nevertheless, we believe that our approach could be used for purposes other than education, such as in the agile design and programming of service and industrial robots. In order to prove this concept, we have evaluated the usability of Botbloq with potential users other than non-university students.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

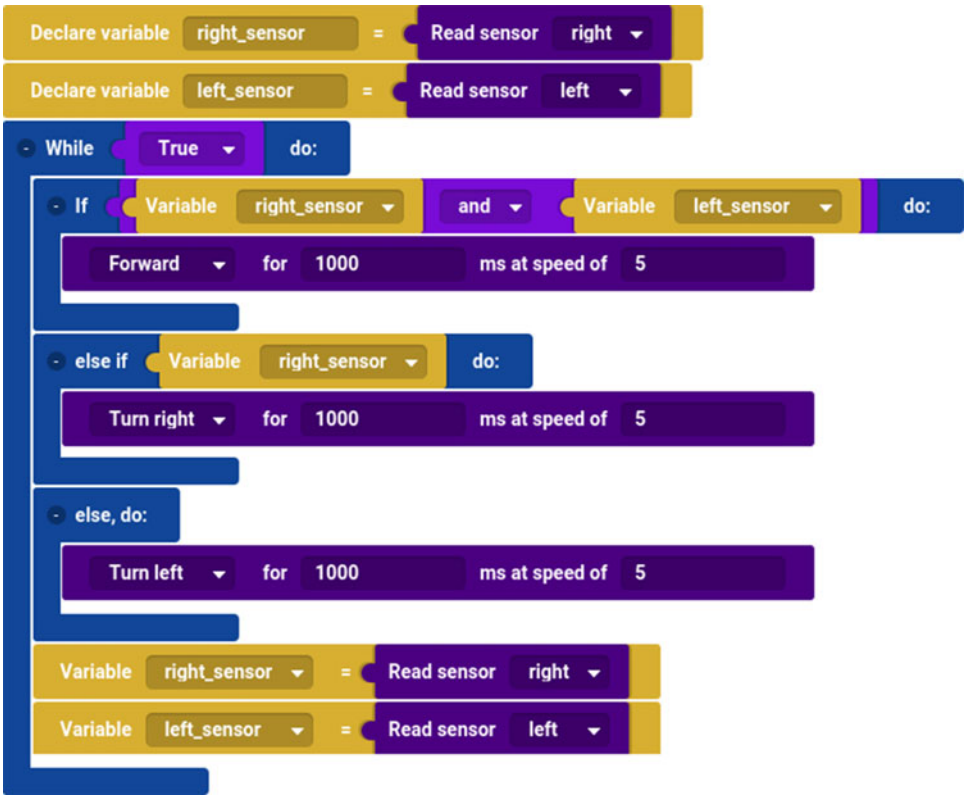


Fig. 17. Example of Bitbloq program for a rover.



Fig. 18. Example of Bitbloq program for a hexapod.

In particular, we have carried out a statistical analysis with 24 mechanical and electrical engineers (i.e., who have recently graduated or are in the last year of their degree). In this analysis, the usability was measured, as defined by the ISO9241 standard, by looking at three main characteristics: efficiency, effectiveness and satisfaction. The efficiency and effectiveness were then calculated by analyzing the programs generated by the users in two classic robot programming problems.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

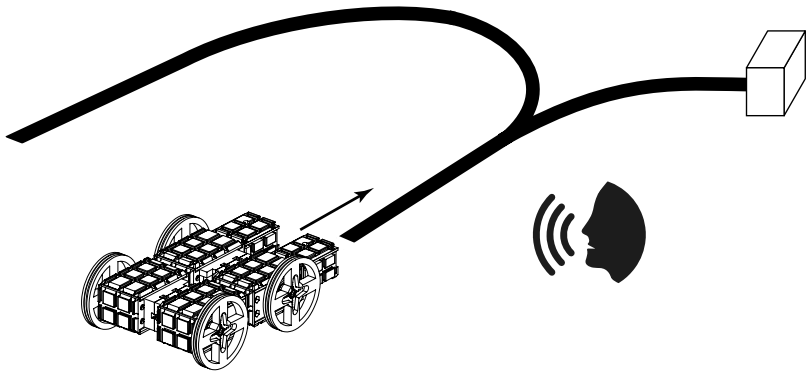


Fig. 19. Test 1. Line follower robot with voice control.

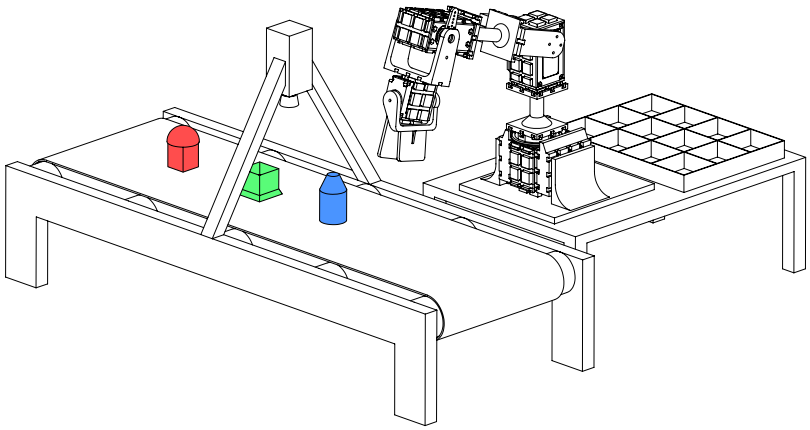


Fig. 20. Test 2. Kitting robot.

Satisfaction was calculated by employing a survey form that was completed by the users after their experience with Botbloq.

The users did not participate in an explanatory course prior to their use of Botbloq, signifying that they worked with the Botbloq framework for the first time in these tests.

6.1. Efficiency and effectiveness analysis

As mentioned above, 24 users were asked to solve the following problems with the Botbloq framework:

- Problem 1: *Line follower robot with voice control* (see Fig. 19). The users were asked to design and program a robot with the skill of line tracking at a maximum speed of 0.2 m/s (i.e., user specifications for the co-designer system). In order to make the programming more complicated, the robot had to handle voice orders (i.e., voice control through the use of a smartphone Bluetooth interface) and it had to be able to follow the left or right side of the line in order to allow it to make decisions at junctions.
- Problem 2: *Kitting robot* (see Fig. 20). The users were asked to design and program a robot with the skill of picking up and placing parts of a maximum weight of 0.1 kg in a workspace of 1 m of diameter (i.e., user specifications for the co-designer system). In particular, the users were told that an external vision system was able to detect parts on a conveyor and the information regarding the part detected (i.e., availability and type) could be read in two Botbloq variables: `part-available` and `part-type`. These variables simulated the memory mapping in the robot controller of the corresponding signals originating from the vision system. Depending on the type, the part had to be placed, regardless of the orientation, on a specific cell in the kitting box. Parts and cell locations were static and known beforehand.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

Table II. Data collected from the experimental validation, where D_i represents the following indicators.

	Mean	STD	Median	% Success
D1	–	–	–	100%
D2	140 s	92.3 s	100 s	–
D3	46.1 s	30.2 s	30 s	–
D4	–	–	–	83.33%
D5	7.2 min	1.8 min	6.5 min	–
D6	2	2.09	2.18	–
D7	–	–	–	83.33%
D8	17.5 min	3.3 min	17.5 min	
D9	–	–	–	66.77%
D10	33.3 min	7.9 min	33 min	–
D11	9.8 lines	2.7 lines	8.5 lines	–
D12	21.7 blocks	4.7 blocks	23 blocks	–
D13	34.6 lines	7.7 lines	33 lines	–
D14	73.9 blocks	11,76 blocks	74.5 blocks	–
D15	2.2 b/l	0.2 b/l	2.13 b/l	–

D1: Success as regards generating the right robot.
D2: Time required to find the right robot if correctly generated.
D3: Average time required to find the desired block for the first time.
D4: Success as regards understanding the use of control structures.
D5: Time required to successfully understand the use of a control structure.
D6: Number of mismatch errors in block types.
D7: Success of problem 1 solution after solving block type mismatch errors.
D8: Time taken to solve problem 1, if successful.
D9: Success of problem 2 solution after solving block type mismatch errors.
D10: Time taken to solve problem 2, if successful.
D11: Number of lines in problem 1.
D12: Number of blocks in problem 1.
D13: Number of lines in problem 2.
D14: Number of blocks in problem 2.
D15: Number of blocks per line in both programs.

The users' solutions to both problems were monitored to collect the data presented in Table II. The indicators shown in the table were selected in order to validate the efficiency and effectiveness of our framework with the following criteria:

- **Effectiveness**. This was evaluated by means of the following aspects:

  1. Robot generation success. D1 is an indicator of the effectiveness of our co-designer system (100% of success). In particular, this indicator denotes how effective our framework is as regards providing the right interface that will enable users to describe the skills to be performed by the robots.
  2. Program success. As the users that participated in this validation knew the theoretical solution to both problems (i.e., the users had sufficient training in the basics of robotics), the success rate of their solutions (indicators D7 and D9) proves how effective our framework is as regards solving problems. The effectiveness for the service robot problem was, accordingly, 83% while for the industrial robot problem it was 66%.

- **Efficiency**. This was evaluated by means of the following aspects:

  1. **Time cost**. Indicators D8 and D10 show the average time required to obtain the solutions to problems 1 and 2 (17.5 and 33.3 min, respectively). Considering that the users had no experience with Botbloq, we believe that this is a good indicator of how agile it is at generating solutions.
  2. **Compactness**. We evaluated how concise the programs generated with our framework can be and employed the following indicators for this purpose:

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press



Fig. 21. Mistake in a block type.

(a) Program length. Indicators D11 and D13 show the average length for problems 1 and 2, respectively (9.8 and 34.6 lines). As will be noted, this length is considerably smaller than its equivalent with other high-level programming languages such as C or python.
(b) Program size. Indicators D12 and D14 show the size of the programs in terms of the number of blocks (an average of 21.7 blocks for problem 1 and 73.9 for problem 2).
(c) Ratio size/length. Indicator D15 shows how verbose the programs generated in our framework are, with the average number of blocks per line being 2.2, signifying that the programmed structures are very concise.

Finally, we have analyzed another two aspects (i.e., simplicity and ambiguity) which, although they do not show how effective or efficient our framework is, are usually considered to be important aspects as regards boosting the usability of an environment:

- **Simplicity**, which was evaluated by studying the following aspects:

1. Robot generation. Indicators D1 and D2 show how quick and easy it is to find a robot configuration that fulfills the user's requirements (100% of achievement in less than 3 min for the proposed problems).
2. Programming environment. Indicator D3 shows how quick and easy it is to use the visual interface of Bitbloq (less than 45 s to find and use a desired block for the first time), which allows users to navigate around blocks and drag and drop them into the programming area.
3. Language structure. The intention of indicators D4 and D5 is to show how simple the control structures are to learn and use for new users (more than 80% of success in less than 8 min of learning time).

- **Ambiguity**. Indicator D6 shows how clear the block building process is (i.e., writing a program), since the ambiguity in block types can be an important drawback. A common mistake is shown in Fig. 21, in which the type of the `voice` block has been mismatched as a Boolean type while it should be text (i.e., text recognized from the users voice). However, indicator D6 shows an average of less than two mistakes, which represents less than 5% of the program code.

6.2. Satisfaction

One of the most standardized tests employed to measure user satisfaction in the usability of a system is the System Usability Scale (SUS) questionnaire.[48,49] This consists of a set of 10 assertions that are evaluated from *strongly disagree* (1) to *strongly agree* (5), thus providing an overall picture of the user experience. The results are summarized in Table III.

The average score obtained in the SUS questionnaire is 77.8, which is equivalent to a C+ grade or a 'Good' usability experience.[50] As this version corresponds to a beta release of the BOTBLOQ environment, we can conclude that the results are very positive.

6.3. Discussion

The problems proposed in this validation are the proof of concept of how our approach could be used in different robot environments, but they are not intended to evaluate the performance of our system as an *end product*. For example, the kitting robot problem would entail several constraints and requirements in the real world that are not considered in problem 2 (e.g., grasp configuration, synchronization with other machines or robots). As mentioned in Section 3.1.1, our system has been designed to be scalable in a way that a complex robot system could be designed and programmed only if its skills are represented in ADROn and the parameterized robot and controller also exist in the co-designer database. In fact, having the parameterized controller of any kind of robot configuration

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

Table III. SUS questionnaire results.

	Strongly disagree 1	**2**	**3**	**4**	**Strongly agree 5**
A1	-	3	11	10	-
A2	15	9	-	-	-
A3	-	-	2	13	9
A4	11	12	1	-	-
A5	-	-	8	14	2
A6	-	11	13	-	-
A7	-	-	1	8	15
A8	8	14	2	-	-
A9	-	-	1	23	-
A10	15	6	3	-	-

A1: I think that I would like to use this system frequently.
A2: I found the system unnecessarily complex.
A3: I thought the system was easy to use.
A4: I think that I would need the support of a technical person to be able to use this system.
A5: I found that the various functions in this system were well integrated.
A6: I thought there was too much inconsistency in this system.
A7: I would imagine that most people would learn to use this system very quickly.
A8: I found the system very cumbersome to use
A9: I felt very confident when using the system.
A10: I needed to learn a lot of things before I could get going with this system.

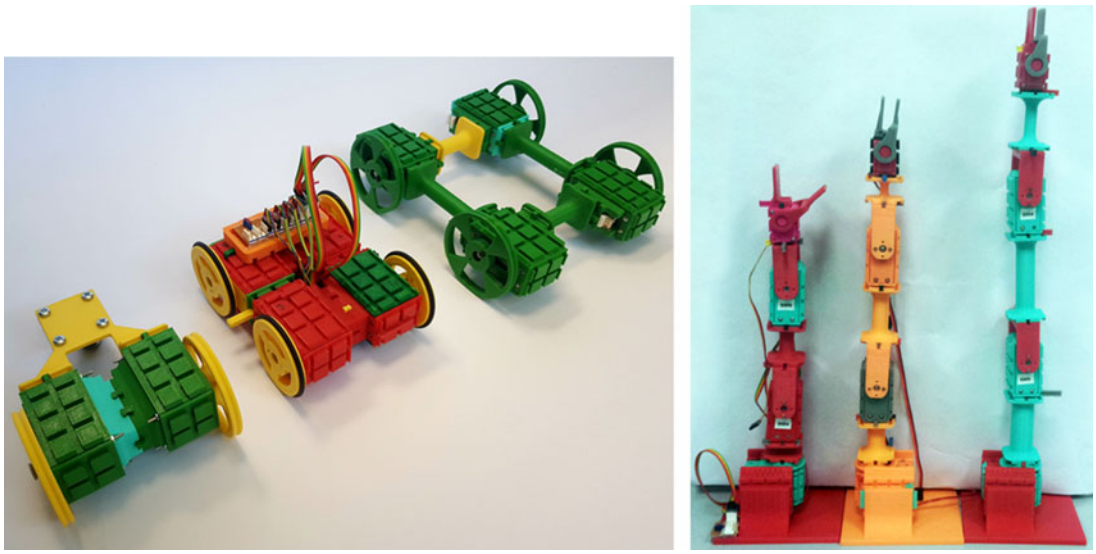


Fig. 22. Examples of different parameterizations of robot according to user requirements.

is an unapproachable problem nowadays, so it is fair to say that robot generation indicators (D1 and D2) show how easy it is to use our framework but not how complete it is.

Efficiency could also be evaluated as the performance to be translated into machine code or the performance when executing programs. As the problems evaluated in this section were proofs of concept, it was difficult to study the efficiency with these meanings. However, our approach is based on the ROS framework, which ensures efficiency with these aspects.

Abstraction is another common aspect that is used to evaluate programming languages. We have not included this aspect, but our approach, as presented in this work, is basically a system that abstracts the complex details of robot programming to users.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

7. Conclusion

As presented in the Introduction section, our approach is a contribution to the solution of a complex problem that involves the co-design of robots and the way in which these new robots can be easily programmed. The work presented in this paper is a proof of concept and is limited to robots co-designed from the parameterization of dimensionless configurations (see Fig. 22 for examples of real robots generated with different parameterizations). However, we have shown that, by following a similar approach, our visual programming framework could be used for any new co-designed robot, making it interesting for future applications.

It is also important to indicate that the controllers and the related blocks in Bitbloq are demonstrators of what can be done, but there is still much room for improvement. For example, manipulators, hexapods and humanoid controllers fulfill aspects such as inverse kinematics and path planning but other important specifications in service and industrial robots, such as accuracy, dynamic response and reactivity to the environment have not been dealt with in this work.

Acknowledgments

This work has been partly supported by the Centre for the Development of Industrial Technologies (CDTI) and the Mundo Reader S.L.(BQ) company under the project *Botbloq-Ecosistema Integral para el diseño, fabricación y programación de robots DIY*.

References

1. S. Edwards and C. Lewis, "Ros-Industrial: Applying the Robot Operating System (ROS) to Industrial Applications," **In:** *IEEE International Conference on Robotics and Automation, ECHORD Workshop* (2012).
2. F. B. V. Benitti and N. Spolaôr, "How Have Robots Supported STEM Teaching?" **In:** *Robotics in STEM Education* (Springer International Publishing, 2017) pp. 103–129.
3. F. Martin and M. Resnick, "Lego/Logo and Electronic Bricks: Creating a Scienceland for Children," In *Advanced Educational Technologies for Mathematics and Science* (Springer, 1993) pp. 61–89.
4. E. Schweikardt and M. D. Gross, "Learning About Complexity with Modular Robots," **In:** *Proceedings of the 2008 Second IEEE International Conference on Digital Game and Intelligent Toy Enhanced Learning, DIGITEL 2008* (IEEE Computer Society, Washington, DC, USA, 2008) pp. 116–123.
5. F. Ramos, A. Olivares-Alarcos, A. S. Vázquez and R. Fernández, "What Can Ontologies Do for Robot Design?" **In:** *ROBOT 2017: Third Iberian Robotics Conference* (Springer International Publishing, 2018) pp. 465–476.
6. F. Ramos, A. S. Vázquez, R. Fernández and A. Olivares-Alarcos, "Ontology based design, control and programming of modular robots," *Integr. Comput.-Aid. Eng.* **25**(2), 173–192 (2018).
7. D. S. Lees and L. J. Leifer, "A Graphical Programming Language for Robots Operating in Lightly Structured Environments," *Proceedings of 1993 IEEE International Conference on Robotics and Automation* (IEEE, 1993) pp. 648–653.
8. Microsoft Robotics Developer Studio, https://www.microsoft.com/en-us/download/details.aspx?id=29081. Accessed July 17, 2018.
9. S. Alexandrova, Z. Tatlock and M. Cakmak, "Roboflow: A Flow-Based Visual Programming Language for Mobile Manipulation Tasks," **In:** *IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015*, (2015) pp. 5537–5544.
10. Intera Software, https://www.rethinkrobotics.com/intera. Accessed July 17, 2018.
11. KUKAprc, http://www.robotsinarchitecture.org/kuka-prc. Accessed July 17, 2018.
12. S. Sharif, V. Agrawal and L. Sweet, "Adaptive Industrial Robot Control for Designers," **In:** *Proceedings of the 35th International Conference on Education and Research in Computer Aided Architectural Design in Europe* (September 2017) pp. 151–158.
13. Robotstudio, https://new.abb.com/products/robotics/robotstudio. Accessed November 11, 2018.
14. Staubli Robotic Suite, https://www.staubli.com/en-us/robotics/product-range/robot-software/pc-robot-programming-srs/. Accessed November 11, 2018.
15. Labview Robotics Toolbox. http://www.ni.com/es-es/shop/labview.html. Accessed July 17, 2018.
16. MATLAB Simulink, https://es.mathworks.com/solutions/robotics.html. Accessed July 17, 2018.
17. Choregraphe Suite, http://doc.aldebaran.com/1-14/software/choregraphe/choregraphe_overview.html. Accessed February 02, 2019.
18. J. Moreno-León and G. Robles, "Code to Learn with Scratch? A Systematic Literature Review," **In:** *2016 IEEE Global Engineering Education Conference, EDUCON 2016, Abu Dhabi, United Arab Emirates, April 10–13, 2016* (2016) pp. 150–156.
19. Scratch, Get Started with Coding, https://scratch.mit.edu/. Accessed July 17, 2018.
20. S. H. Kim and J. W. Jeon, "Programming LEGO Mindstorms NXT with Visual Programming," **In:** *2007 IEEE International Conference on Control, Automation and Systems, ICCAS 2007*, (2007) pp. 2468–2472.

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press

21. Software ev3, https://www.lego.com/es-es/mindstorms/downloads/download-software. Accessed July 17, 2018.
22. Aseba Studio VPL, https://www.thymio.org/. Accessed July 17, 2018.
23. EZ Builder Roboscratch, https://www.ez-robot.com/EZ-Builder/. Acessed July 17, 2018.
24. RobotC http://www.robotc.net/graphical/. Accessed July 17, 2018.
25. ModKit, http://www.modkit.com/. Accessed July 17, 2018.
26. CodeZ, http://gocoderz.com/. Accessed July 17, 2018.
27. NEPO (Open Roberta), https://lab.open-roberta.org/. Accessed July 17, 2018.
28. Bloxter, https://www.dexterindustries.com/introducing-bloxter-the-graphical-programming-language-for-learning-coding-with-robots/. Accessed February 02, 2019.
29. Cubelets Blockly, https://www.modrobotics.com/cubelets/apps/cubelets-blockly/. Accessed February 02, 2019.
30. Moss Scratch, https://www.modrobotics.com/moss/apps/moss-scratch/. Accessed February 02, 2019.
31. Root Robotics, https://rootrobotics.com/. Accessed February 02, 2019.
32. A. Narayanamoorthy, R. Li and Z. Huang, "Creating ROS Launch Files Using a Visual Programming Interface," **In:** *2015 IEEE 7th International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)* (IEEE, 2015) pp. 142–146.
33. Interfacing ROS with Scratch, http://wiki.ros.org/scratch. Accessed July 17, 2018.
34. S. Hart, P. Dinh, J. D. Yamokoski, B. Wightman and N. A. Radford, "Robot Task Commander: A Framework and IDE for Robot Application Development," **In:** *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, IL, USA, September 14–18, 2014* (2014) pp. 1547–1554.
35. Robot Operating System, http://www.ros.org/. Accessed July 17, 2018.
36. F. Ramos, C. O. Scrob, A. S. Vázquez, R. Fernández and A. Olivares-Alarcos, Skill-Oriented Designer of Conceptual Robotic Structures," **In:** *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (IEEE, October 2018), pp. 5679–5684.
37. IEEE Standard Ontologies for Robotics and Automation, IEEE SA 1872-2015, 2015. https://standards.ieee.org/standard/1872-2015.html. Accessed July 17, 2018
38. Schunk Modular Robotic System, https://de.schunk.com/de_en/gripping-systems/category/gripping-systems/handling/modular-and-mobile-gripping-systems/schunk-lightweight-arms/. Accessed July 17, 2018.
39. Muhayyuddin, A. Akbari and J. Rosell, "Ontological Physics-Based Motion Planning for Manipulation," **In:** *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, (September 2015) pp. 1–7.
40. Muhayyuddin, A. Akbari and J. Rosell, "Knowledge-Oriented Physics-Based Motion Planning for Grasping Under Uncertainty," **In:** *ROBOT 2017: Third Iberian Robotics Conference* (Springer International Publishing, 2018) pp. 502–515.
41. XML Macros ROS Package, http://wiki.ros.org/xacro/. Accessed July 17, 2018.
42. diff_drive_controller ROS Package, http://wiki.ros.org/diff_drive_controller. Accessed: July 17, 2018.
43. Moveit for ROS,. http://moveit.ros.org/. Accessed July 17, 2018.
44. J. G. Gómez, Modular Robotics and Locomotion: Application to Limbless Robot. *Ph.D Thesis* (Universidad Autonoma de Madrid, 2008).
45. KDL ROS Package, http://wiki.ros.org/kdl. Accessed September 1, 2016.
46. Bitbloq, http://bitbloq.bq.com/#/. Accessed February 28, 2017.
47. J. Gonzalez-Gomez, H. Zhang and E. Boemo, "Locomotion Principles of 1D Topology Pitch and Pitch-Yaw-Connecting Modular Robots," **In:** *Bioinspiration and Robotics Walking and Climbing Robots* (InTech, 2007).
48. J. Brooke, " SUS: A "Quick and Dirty" Usability Scale," **In:** *Usability Evaluation in Industry* (Taylor & Francis, 1996) pp. 189–194.
49. J. Brooke, "SUS: A retrospective," *J. Usability Stud.* **8**(2), 29–40 (2013).
50. A. Bangor, P. Kortum and J. Miller, "Determining what individual SUS scores mean: Adding an adjective rating scale," *J. Usability Stud.* **4**(3), 114–123 (2009).

https://doi.org/10.1017/S0263574720000934 Published online by Cambridge University Press