Advances in Intelligent Systems and Computing 1316

Wilfried Lepuschitz · Munir Merdan ·
Gottfried Koppensteiner ·
Richard Balogh ·
David Obdržálek *Editors*

Robotics in Education

Methodologies and Technologies

Springer



Physical Bits: A Live Programming Environment for Educational Robotics

Ricardo Moran[1,2(✉)], Matías Teragni[1], and Gonzalo Zabala[1]

[1] Centro de Altos Estudios en Tecnología Informática, Facultad de Tecnología Informática, Universidad Abierta Interamericana, Av. Montes de Oca 745, Ciudad Autónoma de Buenos Aires, República Argentina
{Ricardo.Moran,Matias.Teragni,Gonzalo.Zabala}@uai.edu.ar

[2] Comisión de Investigaciones Científicas de La Provincia de Buenos Aires, Calle 526 e/ 10 y 11, La Plata, Buenos Aires, República Argentina

Abstract. The use of physical computing devices as teaching tools presents several challenges for educators and learners. Most introductory programming environments help to learn programming by removing the possibility of syntax errors, usually by using a visual programming language. However, understanding syntax is just one aspect of the learning process. One of the most challenging tasks for students is to build a correct mental model of the underlying machine model and its execution dynamics. Additionally, visual programming languages present issues when transitioning to text-based languages. In this paper we present Physical Bits, a web-based programming environment for educational robotics that attempts to solve these issues by providing a live programming experience using both visual and textual programming languages.

Keywords: Educational robotics · Live programming · Block-based · Introductory programming

1 Introduction

In the last couple of decades, the use of physical computing devices as teaching tools has increased all over the world. Experts agree on the importance of introducing children to computational thinking, programming, and technology. Using robots in the classroom encourages students into learning these concepts.

Teaching programming to young children is not a recent trend, there have been experiences designing and implementing educational programming environments since the 1970 [1, 2]. Most educational programming environments developed in recent years share the same visual programming style, usually in the form of blocks that snap together when they form valid constructs (inspired by Scratch [3]). Studies show that visual programming environments help to learn programming by removing the possibility of syntax errors and simplifying the programming language, which allows students to focus on understanding the underlying concepts [4, 5]. However, some studies reveal that learning the syntax is not the principal problem because the students only struggle

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2021
W. Lepuschitz et al. (Eds.): RiE 2020, AISC 1316, pp. 291–303, 2021.
https://doi.org/10.1007/978-3-030-67411-3_26

with it at an early stage. One of the most challenging tasks for students to assimilate is to correctly predict the impact of the source code they are writing on the actions performed when the program is executed. Teaching programming presents a dichotomy that is not easy to grasp for beginners: the source code is explicit and visible while the execution dynamics are implicit and harder to understand. Studies show that most student misconceptions are related to an imprecise perception of the execution model presented by the programming language [6].

In order to solve this problem, some studies propose to design educational programming environments in a way that makes the relationship between the source code and its effects more explicit. Most virtual introductory programming environments provide the impression of changing a program while it is running, a feature often described as liveness [7]. In a live programming environment, the users change the program and receive immediate feedback on the effect of the change, without requiring any manual compilation steps and minimizing the time wasted waiting for the code to begin executing. Live programming shortens the feedback loop and encourages experimentation and programming by "Trial & error" [4]. Some environments also support monitoring the internal state of the program by showing the value of the variables as well as highlighting the currently executing blocks. These features, however, are rarely seen in educational programming environment for physical devices [8] and the ones that support it usually do so in detriment of the autonomy of the robot, requiring a computer connected at all times in order to run the programs and send the commands to the robot as they are executed. Due to the latency of the communication, these environments are limited to projects that do not require precise timing and cannot be used in many robotics competitions.

Another common problem identified with teaching using visual programming environments is the eventual transition to text-based languages. Most block-based programming environments do not aid the learner in the transition to text-based languages, instead they are more concerned with simplifying the programming language and hiding the complicated syntax rules. This has led to a perceived "lack of authenticity": some students tend to think of block-based programming as not "real programming", potentially damaging their effectiveness in introductory courses [4]. Another problem found when using visual programming environments is referred to as "syntax overload" [9]. In a study performed by Powers et al. [10] a CS0 course started using the Alice visual programming environment and then moved to a text-based programming language (either Java or C++). The authors report the transition was unsuccessful. Students were overwhelmed and frustrated by the strict syntax requirements of the text-based languages. Furthermore, some students ended up acquiring poor programming habits that damaged their ability to work with text-based languages. A few different solutions have been proposed to alleviate this problem, such as: using blocks that look like text, automatic placement of syntax, and helpful syntax errors instead of removing the syntax altogether [9].

In this paper we present our attempt at solving these problems.

2 Proposed Solution

We have developed Physical Bits, a web-based programming environment for educational robotics. All the code for the system is open source[1] and, at the time of this writing, we have made five public releases with varying levels of functionality. We have decided to initially target the Arduino platform because of its low cost and popularity but we have designed the system with portability in mind and we plan to support other hardware platforms in the future.

Physical Bits was designed to solve the above-mentioned problems; thus, it supports a set of features that distinguish it from other similar tools. In this section we will briefly describe those features in relation to the problems identified above.

2.1 Live Programming and Autonomy

The programming environment is supported by a virtual machine responsible for executing the user programs. This virtual machine is installed on the robot, allowing Physical Bits to provide a live programming experience without sacrificing the autonomy of the robot, which is required for most projects. The programming tools connect to the robot through a serial port and every change made to the user program is automatically compiled, verified, and transmitted to the virtual machine, taking less than 100 ms, after which the robot will start to execute the new program. This allows to shorten the feedback loop, encouraging experimentation and programming by "trial and error". Furthermore, if the robot is connected to the computer, the environment displays all the values of the sensors as well as the global variables declared in the program. Making the data concrete and visible helps the user understand the underlying machine model [6]. Finally, the environment also supports classic debugging features such as breakpoints at arbitrary instructions and step-by-step execution. Apart from aiding the user in the process of fixing programming errors, this feature makes the underlying machine model visible, thus helping the students make the correct mental model.

2.2 Block-Based and Text-Based Programming

Recognizing the benefits of visual programming, Physical Bits includes a block-based language suitable for beginners. However, in order to avoid the above-mentioned issues Physical Bits also supports text-based programming using a custom programming language designed specifically for educational robotics. We decided to develop our own programming language instead of using an already established one for a simple reason: we can control its syntax and ensure it is minimal while also providing useful constructs for the robotics domain. This custom language is inspired by the C programming language but it is much more simple and limited, focused only on educational robotics. The carefully chosen syntax is designed to avoid the "syntax overload" problem experienced by beginners while at the same time being recognizable for experienced programmers. This language is not a goal in itself, instead we consider it as a tool to help the student learn more powerful and useful languages.

[1] https://github.com/GIRA/UziScript.

In order to provide a smooth transition from block-based and text-based programming the Physical Bits environment allows programming in either mode or both at the same time. The user can start programming by assembling blocks, not worrying about syntax, and the environment will automatically generate the corresponding textual code and display it alongside the blocks. Once the user has become so proficient using the visual programming language that he starts to feel its limitations, he can write textual code and the environment will update the blocks accordingly, allowing to compare the semantics of the written code with the equivalent blocks the student already knows. If, at any moment, the student starts to feel overwhelmed by the syntax he can go back to program with blocks without losing what he already wrote. The environment takes care of keeping both representations of the user program (visual and textual) automatically synchronized. This lets the student go back and forth between them as he pleases while learning about the relation between the language syntax and the blocks, helping smooth the transition without making the experience frustrating.

3 Related Work

We have reviewed some programming environments for educational robotics in order to compare their features with Physical Bits. This is not an exhaustive list, as there are many more programming environments available and including them all would exceed the scope of this paper. We have selected these environments based mainly in their relative popularity and our own familiarity with their implementations.

In order to compare functionality, we have selected a small number of features each environment can either support or not. The features we will be comparing are:

1. Liveness: The ability to change the program while it runs and see the changes immediately without manual user intervention (like clicking a button).
2. Monitoring: The ability of the environment to automatically inspect the state of the running program and let the user see the value of all variables as well as the state of all pins or sensors in the robot.
3. Debugging: The ability to pause the execution at any time in order to visualize the program execution step-by-step.
4. Autonomy: The ability to run the programs inside the robot without being connected to a computer.
5. Programming interface: Whether the environment supports block-based programming, text-based programming, or both.

The result of our analysis is presented in the table below (Table 1).

Although the list is not exhaustive, this selection offers a representative picture of the most commonly found strategies for developing programming environments for educational robotics. A majority of environments support visual programming in the form of blocks but only a few provide textual code generation.

The most notable programming environment we reviewed is microBlocks. What distinguishes it from all the alternatives is that it took the same approach as Physical Bits: using a virtual machine in order to run programs autonomously in the robot while

Table 1. Programming environments comparison

	Liveness	Monitoring	Debugging	Autonomy	Programming interface
Physical Bits	✔	✔	✔	✔	Blocks + Text
MicroBlocks [8]	✔	✔	✗	✔	Blocks only
Physical Etoys [11]	✔	✔	✔	✗	Blocks + Text
Lego Mindstorms [12]	✗	✔	✗	✔	Blocks only
Scratch4Arduino [13]	✔	✔	✗	✗	Blocks only
Snap4Arduino [14]	✔	✔	✗	✗	Blocks only
XOD [15]	✗	✔	✗	✔	Blocks only
Ardublock [16]	✗	✗	✗	✔	Blocks only
MakeCode [17]	✗	✗	✗	✔	Blocks + Text
Arduino IDE [18]	✗	✗	✗	✔	Text only

still providing an interactive and dynamic experience. However, microBlocks does not implement any debugging tools and it does not support text-based programming (the authors expressed it is one of their goals for the future). Apart from that, microBlocks target a different set of devices. The microBlocks virtual machine is designed for 32-bit microcontrollers while Physical Bits works on 8-bit microcontrollers. Moreover, microBlocks requires at least 12 Kb of RAM and 50 Kb of Flash memory while Physical Bits requires 2 Kb of RAM and 24 Kb of Flash memory.

Another interesting environment is XOD. In contrast to all other environments in this list, XOD is a data flow language. It uses blocks to represent nodes in a directed graph, in which each node can represent an input (typically a sensor), some computation, or an output (typically an actuator). The programs are built by linking nodes together, which are then used to generate the actual native program that the user can upload to the board.

MakeCode caught our interest mainly because it is a web-based programming environment that supports both blocks and text-based programming using Javascript, and it provides bidirectional code generation. Although the MakeCode environment does not provide a live programming experience, it compensates this limitation by providing a virtual simulator, making it possible to test programs without a physical device.

Finally, Physical Etoys is an extension to Etoys we developed in order to interact with a wide range of physical devices such as LEGO NXT, Arduino boards, innovative joysticks like Microsoft Kinect or Leap Motion, etc. It allows to program robots using either blocks or a text-based programming language (Smalltalk). However, once the user

has modified a script using the textual mode it does not support going back to the blocks without losing the changes. This problem, together with its lack of autonomy, led us to develop the Physical Bits environment.

4 Implementation

4.1 Architecture

The system is composed of three distinct components: the IDE, which is a web application containing all the programming tools; the middleware, which is a desktop application that handles the communication with the robot; and the firmware, which contains the virtual machine (Fig. 1).

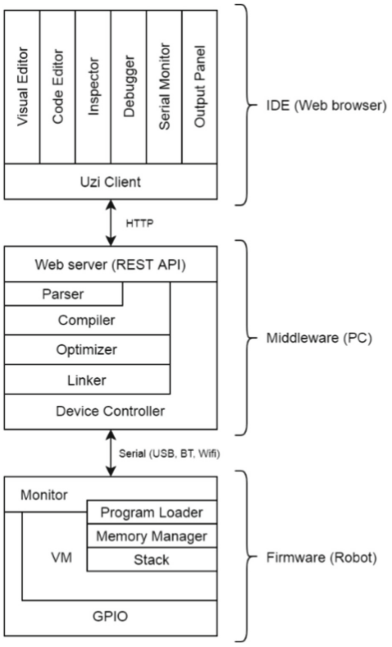


Fig. 1. Physical Bits architecture diagram

This architecture has several benefits. On the one hand it is flexible. The IDE, being a web app, could be used from any device with a web browser, such as a laptop or a mobile phone. It could be installed as a native app or accessed through a web browser. Both the middleware and the firmware are portable: although the only implementation of the latter currently supports Arduino boards, the code could be ported to other types of robots without changes to the middleware or IDE. It is fast: compiling, verifying, and uploading programs using the Physical Bits IDE takes a fraction of the time required to compile an Arduino sketch, mostly because of the small size of the programs. And finally, the communication to the robot can be done wirelessly either using bluetooth

or a network socket (although the current implementation has only been tested using a USB cable).

4.2 Firmware

The firmware is a regular Arduino sketch written in C++ that can be uploaded using the Arduino IDE. We have tested the firmware using several different boards, including: Arduino UNO, Micro, Nano, MEGA 2560, and Yun. We have also received reports of it working successfully on other compatible boards such as DuinoBot [19], Educabot [20], and TotemDUINO [21]. The firmware contains a stack-based high-level language virtual machine that executes user programs using a decode and dispatch bytecode interpreter [22]. This implementation was chosen mainly because of its simplicity. Apart from the virtual machine, the firmware also contains a monitor program that allows it to interact with the middleware in the host computer. Periodically, this monitor program will send the status of the Arduino and receive commands, allowing the middleware to fully control the virtual machine, including directly manipulating the variables and the pin state, debug the current user program, or download a new one. By having these two components running on the Arduino we can provide a live programming experience with a short feedback loop without sacrificing autonomy.

The firmware also contains a GPIO class that simplifies working with the Arduino pins. It takes care of setting the pin modes according to its usage; it stores the pin values in order to access them later; and it handles PWM, square wave generation, and servo control. This GPIO class is designed to be configurable using compile-time macros in order to support other pin layouts, enabling the use of different boards.

4.3 Middleware

The middleware contains a small set of tools that allow to compile, debug, and transmit the programs to the Arduino board through a serial connection. All these tools were originally developed using Squeak, an open source version of Smalltalk [23]. We decided to use Smalltalk to build the first prototype of the middleware mainly due to of our familiarity with the language. We later ported this code to the Clojure programming language for performance and ease of deployment [24].

In order for the IDE to interact with these tools the middleware exposes a REST API containing endpoints to: connect and disconnect from the robot; compile, run, and install programs; and to retrieve the state of the robot, which includes the sensor and global data. The compiler takes the source code of a program in our custom programming language and generates bytecodes in a format that the virtual machine can decode and execute. The language includes common syntactic constructs typically found in structured programming languages, such as: conditionals (if-then-else), loops (while, until, for, repeat, etc.), procedures and functions with positional arguments, variables (both local and global scope supported). Additionally, we added the “task” syntax for defining concurrent processes.

4.4 Ide

The Physical Bits IDE is a web-based programming environment that supports both visual programming (using the Blockly library [25]) as well as textual code using a custom programming language loosely inspired by C. Even though the middleware needs to run locally in order to access the serial port, the IDE, which is composed of HTML and Javascript files can either be hosted locally, on a web server, or even on the cloud (as long as the client browser has access to the middleware's API) (Fig. 2).

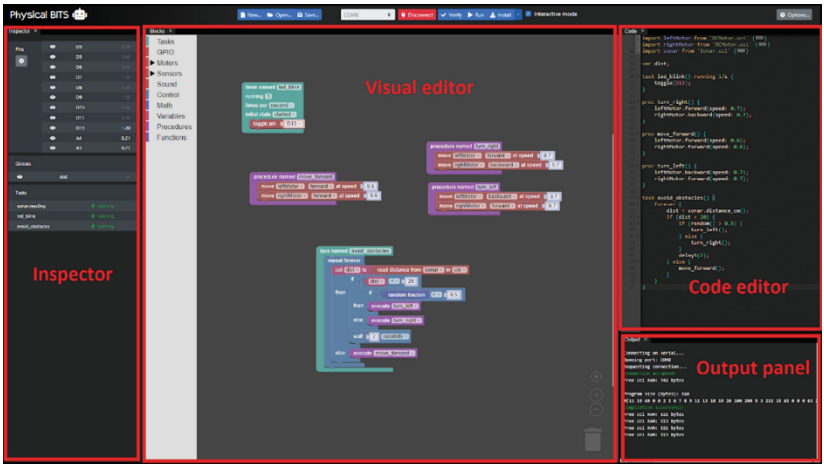


Fig. 2. Screenshot of the IDE and its panels

The environment is designed to provide a smooth transition from block-based programming to text-based programming. In order to do this the student can choose to work on his programs using either mode or both at the same time. The environment takes care of translating automatically from blocks to code and back, this helps with the transition by showing how each block is representing in code while the user edits the program.

Every time the user makes a change in the program (independent from the mode) the program is sent to the middleware, which compiles it, looks for syntax errors, and if the program is correct and the robot is connected it sends it to the robot to be executed immediately. Since the entire compilation process finishes within 35 ms on average, effectively every change in the program is automatically transmitted to the robot, which immediately starts executing it.

Apart from the instant compilation and execution, the IDE also supports inspection of the pins in the arduino and the variables in the program. The inspector panel is automatically configured to show the current values of each pin and variable referenced in the program, this allows users to see the values of any sensor immediately without requiring any extra code. This feature allows to make the process of programming a robot much more transparent and helps to understand the behavior of the program.

Finally, if a program doesn't behave as expected the user can stop the execution at any point (breakpoint) and execute it step-by-step, observing how the state of the program changes at each step.

5 Code Example

In order to demonstrate the programming language we made a small example program consisting of a robot car that autonomously wanders in a room while avoiding obstacles.

The robot is made of two DC motors that provide movement and an ultrasonic sensor in the front to detect obstacles. This type of robots is very common in robotics competitions such as the Roboliga, the Argentinian Robotics Olympics and the Robocup Junior, an international competition where students from around the world gather to share their experiences in educational robotics.

The expected behavior of the robot is simple: it should move forward until an obstacle is detected, in which case it should turn left for a second (Fig. 3).

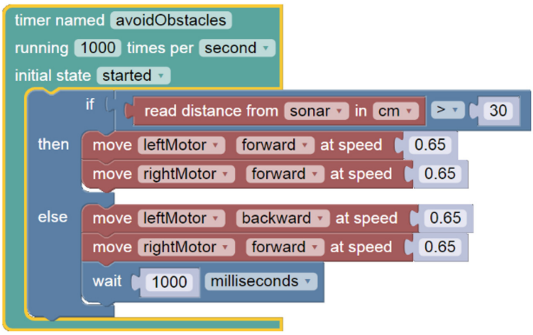


Fig. 3. Blocks-based version

The block-based program consists of a single "timer" block. This block allows to specify behavior that should be executed periodically. Inside the "timer" block we have an "if-then-else" block that represents a conditional statement. It reads the value of the ultrasonic sensor, called "sonar" in this example. If the reported value is greater than 30 cm, it will activate both motors forward at a constant speed. Otherwise it will move the "leftMotor" backward and the "rightMotor" forward (making the robot turn counter-clockwise) for 1000 ms.

The autogenerated textual code is a direct translation of the blocks above, with the only exception of the "import" statements at the top of the program. These statements allow to include external libraries in the user program. In the interest of brevity, we have omitted the source code for these imported libraries but they can be found in github[2].

[2] https://github.com/GIRA/UziScript/tree/master/uzi/libraries

```
import leftMotor from 'DCMotor.uzi' {
  enablePin = D5; forwardPin = D7; reversePin = D8;
}
import rightMotor from 'DCMotor.uzi' {
  enablePin = D6; forwardPin = D11; reversePin = D9;
}
import sonar from 'Sonar.uzi' {
  trigPin = A5; echoPin = A4; maxDistance = 100;
  start reading;
}

task avoidObstacles() running 1000/s {
  if (sonar.distance_cm() > 30) {
    leftMotor.forward(speed: 0.65);
    rightMotor.forward(speed: 0.65);
  } else {
    leftMotor.backward(speed: 0.65);
    rightMotor.forward(speed: 0.65);
    delayMs(1000);
  }
}
```

Just for reference we have included the same program written in the Arduino language. In this case, we also decided to omit the source code of the external libraries used but they can be found online[3].

As we can see, except for a few syntactic differences, the Arduino code is very similar to the code generated by Physical Bits. This is by design. We have chosen the language syntax to be very easy to understand for any experienced programmer and very similar to Arduino code. Our goal for this language is to be just a learning tool, an intermediate step before moving to more powerful languages.

```
#include <L293.h>
#include <NewPing.h>

L293 leftMotor(5, 7, 8);
L293 rightMotor(6, 11, 9);
NewPing sonar(A5, A4, 100);

void setup() {}

void loop() {
  delay(50); // Wait between pings to avoid echo issues
  int dist = sonar.ping_cm();
  if(dist == 0 || dist > 30) {
    leftMotor.forward(165);
    rightMotor.forward(165);
  } else {
    leftMotor.back(165);
    rightMotor.forward(165);
    delay(1000);
  }
}
```

One important difference, however, is the "task" construct, which just like the "timer" block allows to specify behavior that the robot should execute periodically. It can be seen

[3] L293: https://github.com/qub1750ul/Arduino_L293NewPing: https://bitbucket.org/teckel12/arduino-new-ping/wiki/Home

as the equivalent of the loop() procedure in Arduino but with the possibility of declaring multiple tasks instead of just one.

Since this capability does not exist in the Arduino language without the inclusion of a third-party library it can be a source of confusion for beginners. However, we feel that since any programming language for robotics needs to support concurrency in order to be effective [26], the benefits of moving away from the Arduino execution model outweigh its disadvantages.

6 Future Work

Although we have reached a point in development in which we feel confident enough to start using this tool with students, this project is still a work in progress.

Some of the features described in this paper, while working in previous releases, are not fully integrated in the latest version of the IDE yet. These include: the watchers, the debugger, and the text-to-blocks translation. The latter remains an open issue due to the lack of a 1:1 mapping between the textual language constructs and the blocks. We have discussed extending the block-based language by introducing more complex blocks but we fear we might overwhelm the users if we provide "too many" blocks to choose from. Even though the discussion is still open we have decided, for the moment, to translate as much code as we can using the existing high-level blocks and, for any statement that can't be translated, provide a special block (hidden from the user) that simply represents this untranslatable code.

Apart from that, we have plans for porting the Uzi VM to other hardware platforms like ESP8266 and ESP32. We also plan to improve the robustness of the communication protocol with the robot. We have observed some issues that can leave the robot in an inconsistent state and force the user to reset the connection, interrupting the live experience we are trying to reach. We expect to be able to optimize the virtual machine in order to provide faster execution.

Finally, some related projects we hope to develop are: a cloud repository for users to share their creations and discover other people's projects; and a designer tool to help teachers to plan activities for their classroom.

7 Conclusions

We have described the more common problems found when using visual programming languages for teaching robotics.

Our proposal consists of Physical Bits: a web-based programming environment for educational robotics that attempts to solve these issues by providing a live programming experience as well as bidirectional blocks to code generation. These features make Physical Bits a suitable introductory environment for teaching programming using robots. We have reviewed popular alternative programming environments and compared its characteristics with Physical Bits. We also described the implementation of the system as well as a small code example to demonstrate the language features.

Although this is still a work in progress, we believe this project is a step towards our vision of what educational robotics should be in the future.

References

1. Feurzeig, W., Papert, S., Bloom, M., Grant, R., Solomon, C.J.: Programming-languages as a conceptual framework for teaching mathematics. ACM SIGCUE Outlook **4**(2), 13–17 (1970)
2. Kay, A.C.: The early history of Smalltalk. In: History of Programming Languages---II, pp. 511–598. Association for Computing Machinery, New York (1996)
3. Meerbaum-Salant, O., Armoni, M., Ben-Ari, M.: Habits of programming in scratch. In: ITiCSE 2011 Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, pp. 168–172 (2011)
4. Weintrop, D., Wilensky, U.: To block or not to block, that is the question: students' perceptions of blocks-based programming. In: Proceedings of the 14th International Conference on Interaction Design and Children, pp. 199–208 (2015)
5. Kelleher, C., Pausch, R.: Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers. ACM Comput. Surv. (CSUR) **37**(2), 83–137 (2005)
6. Lodi, M., Malchiodi, D., Monga, M., Morpurgo, A., Spieler, B.: Constructionist attempts at supporting the learning of computer programming: a survey. Olympiads Inf. Int. J. Vilnius Univ. Int. Olympiad Inf., 19–121 (2019)
7. Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., Pape, T.: Exploratory and live, programming and coding: a literature study comparing perspectives on liveness. Art Sci. Eng. Program. **3**(1) (2019)
8. Cabrera, L., Maloney, J., Weintrop, D.: Programs in the palm of your hand: how live programming shapes children's interactions with physical computing devices. In: Proceedings of the 18th ACM International Conference on Interaction Design and Children, pp. 227–236 (2019)
9. Moors, L., Luxton-Reilly, A., Denny, P.: Transitioning from block-based to text-based programming languages. In: 2018 International Conference on Learning and Teaching in Computing and Engineering (LaTICE), pp. 57–64 (2018)
10. Powers, K., Ecott, S., Hirshfield, L.: Through the looking glass: teaching CS0 with Alice. ACM SIGCSE Bull. **39**(1), 213–217 (2007)
11. Grupo de Investigación en Robótica Autónoma del CAETI (GIRA), Physical Etoys (2010). https://tecnodacta.com.ar/gira/projects/physical-etoys/, Accessed 15 Junio 2017
12. Kim, S.H., Jeon, J.W.: Programming LEGO mindstorms NXT with visual programming. In: 2007 International Conference on Control, Automation and Systems, pp. 2468–2472 (2007)
13. Citilab, About S4A (2015). https://s4a.cat/, Accessed 15 Junio 2017
14. Pina, A., Iñaki, C.: Primary level young makers programming & making electronics with Snap4Arduino. In: Educational Robotics in the Makers Era, pp. 20–33 (2017)
15. XOD. https://xod.io/, Accessed 24 Jan 2020
16. Ardublock|A Graphical Programming Language for Arduino. https://blog.ardublock.com/, Accessed 15 Junio 2017
17. Microsoft. https://makecode.microbit.org/, https://makecode.microbit.org/, Accessed 24 Jan 2020
18. Arduino, Arduino Playground - Structure. https://playground.arduino.cc/ArduinoNotebookTraduccion/Structure, Accessed 23 Julio 2017
19. Rojas, A.: Reporte Robótica Educativa. Universidad Nacional de La Pampa (UNLPam) (2017)
20. Educabot. https://educabot.org/, Accessed 13 Dec 2019
21. Totem, TotemDUINO|Totemmaker.net, Totemmaker.net. https://totemmaker.net/product/totemduino-arduino/, Accessed 13 Dec 2019
22. Smith, J., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann Publishers Inc, San Francisco (2005)

23. Ingalls, D.: The evolution of smalltalk: from smalltalk-72 through squeak. In: Proceedings of the ACM on Programming Languages (PACMPL), vol. 4, no. HOPL (2020)
24. Hickey, R.: A history of clojure. In: Proceedings of the ACM on Programming Languages (PACMPL), vol. 4, no. HOPL (2020)
25. Fraser, N.: Ten things we've learned from blockly. In: Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), pp. 49–50 (2015)
26. Resnick, M.: MultiLogo: a study of children and concurrent programming. Interact. Learn. Environ. **1**(3), 153–170 (1990)