Flowboard: How Seamless, Live, Flow-Based Programming Impacts Learning to Code for Embedded Electronics

ANKE BROCKER, RWTH Aachen University, Germany
RENÉ SCHÄFER, RWTH Aachen University, Germany
CHRISTIAN REMY, Lancaster University, United Kingdom
SIMON VOELKER, RWTH Aachen University, Germany
JAN BORCHERS, RWTH Aachen University, Germany

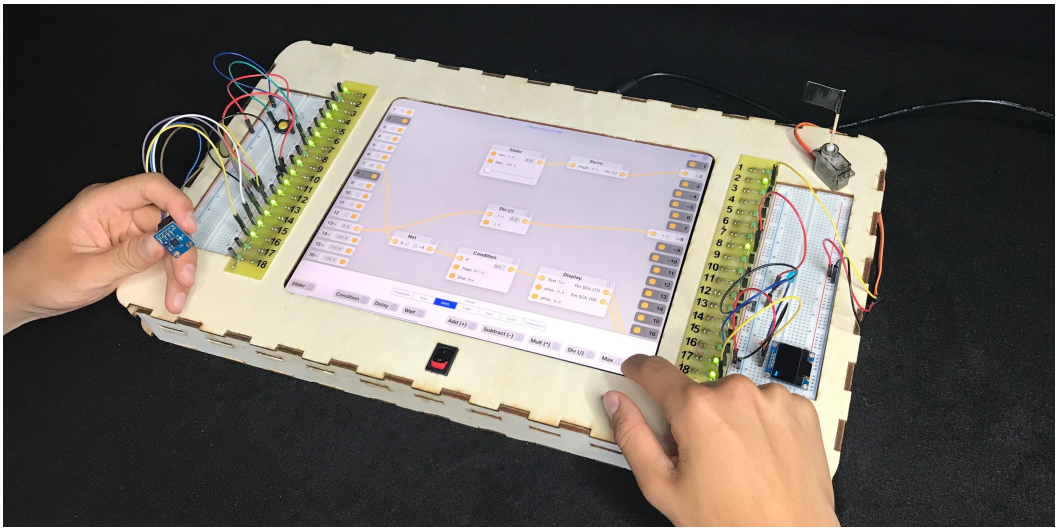

Fig. 1. Flowboard is a system to learn embedded coding based on the flow-based programming (FBP) paradigm. Its hardware combines a large iPad Pro with an Arduino and switchboard circuit beneath it and a breadboard on each side. The user develops her code using Flowboard's visual FBP editor on the iPad. Electronic components for sensing (input, left) and actuation (output, right) are plugged into the breadboards and linked seamlessly to processing nodes on the screen via two duplicate rows of I/O pins next to the iPad. Parallel processes are easy to code: Here, a physical button (top left) controls what is written to a serial OLED display (bot. right), a force sensor (left center) dims an LED (right center), an on-screen slider (top center) controls a servo, and a 3-axis accelerometer (bot. left) is connected just to see its sensor values live (bot. left).

Authors' addresses: Anke Brocker, brocker@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany, 52056; René Schäfer, rschaefer@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany, 52056; Christian Remy, remyc@lancaster.ac.uk, Lancaster University, Lancaster, United Kingdom, LA1 4WA; Simon Voelker, voelker@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany, 52056; Jan Borchers, borchers@cs.rwth-aachen.de, RWTH Aachen University, Aachen, Germany, 52056.

1073-0516/2022/1-ART1 $15.00
https://doi.org/10.1145/3533015

Toolkits like the Arduino system have brought embedded programming to STEM education. However, learning embedded programming is still hard, requiring an understanding of coding, electronics, and how both sides interact. To investigate the opportunities of using a different programming paradigm than the imperative approach to learning embedded coding, we developed *Flowboard*. Students code in a visual iPad editor using *flow-based programming*, which is conceptually closer to circuit diagrams than imperative code. Two breadboards with I/O pins mirrored on the iPad connect electronics and program graph more *seamlessly* than existing IDEs. Program changes take effect immediately. This *liveness* reflects circuit behavior better than edit-compile-run loops. A first study confirmed that students can solve basic embedded programming tasks with Flowboard while highlighting important differences to a typical imperative IDE, Ardublock. A second, in-depth study provided qualitative insights into Flowboard's impact on students' conceptual models of electronics and embedded programming and exploring those.

CCS Concepts: • **Human-centered computing** → **Human computer interaction (HCI)**; **User studies**; *Interactive systems and tools*; • **Applied computing** → *Interactive learning environments.*

Additional Key Words and Phrases: Embedded Development Environments, Visual Flow-Based Programming, Arduino, Electronics, Young Learners, Learning Tools

ACM Reference Format:
Anke Brocker, René Schäfer, Christian Remy, Simon Voelker, and Jan Borchers. 2022. Flowboard: How Seamless, Live, Flow-Based Programming Impacts Learning to Code for Embedded Electronics. *ACM Trans. Comput.-Hum. Interact.* 1, 1, Article 1 (January 2022), 37 pages. https://doi.org/10.1145/3533015

1 INTRODUCTION

Beginner-friendly embedded development environments like the Arduino IDE (Integrated Development Environment) have greatly lowered the threshold for makers to create interactive artifacts [14], to integrate electronics into the physical world, and thus realize the idea of Physical Computing [12]. They have become a main ingredient of Science, Technology, Engineering, and Mathematics (STEM) education [40]. However, learning embedded development remains inherently challenging, as it requires an understanding of (a) coding, (b) electronics, and (c) how code and electronics connect and influence each other [34, 42].

While HCI research has a long tradition of improving the first aspect of the coding user experience (see [38] for an overview), and several research projects [18, 53, 54] have tackled the second aspect of helping learners with the electronics design side of their work, we set out to address the third aspect: to help students better understand the interplay between the hardware and software sides of their work and develop a more integrated understanding of embedded programming.

To provide us with new room to improve this aspect of the learner experience in embedded development, we decided to use *flow-based programming (FBP)* instead of the traditional imperative programming as the underlying programming paradigm (Fig. 1). The initial reason was that a program graph of connected processing nodes in FBP resembles the concept of an electronic circuit in hardware more closely than sequences of commands in imperative programming.

However, we quickly realized that FBP possesses properties that also allowed us to improve learning embedded coding in other ways:

- It made it natural to create a *live* IDE. *Liveness* is defined as the property of an IDE that lets program changes take effect immediately. This mirrors the experience of changing an electronic circuit, and users of embedded programming IDEs expect them to be live [13].
- FBP also enabled a more *seamless* arrangement of electronics and code: We define *seamlessness* in embedded programming as reducing the gap, both physically and conceptually, between the software and hardware parts of a project, creating an integrated embedded coding experience. For example, the physical Arduino pin connectors in our Flowboard prototype (Fig. 1) are

right next to their virtual representations in the programming editor on the iPad screen (the actual Arduino board is hidden inside the Flowboard case).
- Finally, coding *parallel* independent processes is trivial in FBP, even together with another person, and again closely resembles building several parallel circuits in electronics (Fig. 1).

We created *Flowboard*, a flow-based embedded IDE using these concepts of liveness and seamlessness, to explore these opportunities. It is primarily aimed at children starting to learn embedded programming in informal learning scenarios. We evaluated it in two studies with high school students.

In the remainder of this article, we first review related work on simplifying embedded development for learners, before describing the concept, design, and implementation of *Flowboard*. We then report on two user studies. Study 1 evaluates basic Flowboard use and confirms that students can solve embedded programming tasks with Flowboard, while highlighting important differences to a typical imperative IDE for embedded programming, Ardublock. The second, in-depth study provides qualitative insights into how using Flowboard impacts students' conceptual models of electronics and embedded programming. Finally, we conclude with a discussion of the limitations of our work and suggestions for future research.

Our key contributions therefore are

- the *Flowboard* system concept and open-source functional hardware and software prototype that offers flow-based programming for Arduino embedded development with the qualities of *seamlessness* and *liveness*. We propose FBP as an alternative paradigm for learning embedded programming that has received little attention so far.
- the results of two qualitative user studies, with the first showing Flowboard's utility as a learning tool for embedded development, highlighting differences to an imperative IDE, and the second providing more in-depth insights into how Flowboard and the flow-based paradigm impacts students' perception and learning of electronics and embedded programming.

Parts of the technical description of Flowboard were published as a nonarchival Interactivity demonstration at the CHI 2019 conference [11]. All other content of the present article, including the two user studies, is new. To enable replication, further research, and classroom use, all parts of Flowboard are available as open-source[1] *(archive also provided as a supplement)*.

2 RELATED WORK

We structure our review of related work according to the three areas for improvement identified above: First, since Flowboard does not focus on supporting the hardware side of electronic circuit design specifically, we provide only a brief overview of projects in that area, most of which could be combined with Flowboard's approach. Second, we take a closer look at the software side and review projects employing *flow-based programming*, in particular in learning environments. We then discuss projects that, like Flowboard, aim primarily at reducing the conceptual gap between hardware and software in learning embedded programming. Finally, we look at research studying how learning systems in physical computing, embedded programming, and electronics affect students' perceptions. We distinguish between physical computing—the entire range of programming that involves a variety of sensors and actuators rather than just a computer with (touch)screen, keyboard, and mouse [12]—and embedded programming, in which code is always developed for and uploaded to a separate microcontroller.

[1] https://hci.rwth-aachen.de/flowboard

2.1 Software: From Blocks to Flow-Based Programming

Physical computing in general, and embedded programming in particular, come with additional challenges regarding programming the circuits built [10], which underlines the importance of developing usable and efficient tools for these development tasks. Consequently, on the software side, various research approaches to improve the coding experience, especially for beginners and children, have also been applied to embedded IDEs.

One of the first toolkits intended to lower the threshold to physical computing is Phidgets [20]. Its collection of sensor boards is connected to a (Windows) computer without the need for soldering or breadboards. The code, however, is developed and runs on the connected PC, e.g., in Visual Basic, which sets this project apart from embedded programming situations in which the resulting code is uploaded to and executed on a standalone microcontroller.

An approach popularized by projects such as Scratch[2] is to replace the text editor in an IDE with a graphical editor that lets the learner assemble statements visually as *blocks*. This aids learners by removing the need to know the vocabulary of programming language statements by heart and by removing the issue of syntax errors, such as a missing semicolon after a statement. The block-based approach has been carried over to embedded development by projects like the Scratch extension *Scratch for Arduino (S4A)*[3], the Java-based *Ardublock*[4], and the *ModKit*[5] IDE used in robotics education. LAWRIS [3], a rule-based web learning platform for Arduino, uses the widely adopted open-source visual block-based programming editor *Blockly*[6] and limits itself to four input and output devices to reduce complexity. Other Arduino-like platforms like the micro:bit[7] or Calliope[8] also use block-based editors and have been used to teach embedded programming in schools [9].

However, the projects above rely on the traditional imperative programming paradigm. Understanding the sequentiality of statements in imperative programming is a major challenge [48], especially for young learners. This has inspired environments that use the flow-based programming (FBP) paradigm [36] instead. Johnston et al. [23] provide a comprehensive overview of advances in FBP research. In FBP, programs are not sequences of commands but a network of processing nodes connected via their inputs and outputs. Data flows through this network, each node changing the data based on its type and parameters. This paradigm closely resembles many physical processes, from biological signaling in neural networks to signal flow in analog and digital electronic circuits. Unlike in imperative programming, expressing parallel processes within one program is straightforward [23]. In commercial IDEs, FBP has been used for decades across a broad range of domains, from scientific experimentation (LabVIEW [52]) to interactive music and multimedia installations (Max/MSP[9]) and mobile UI design (Facebook's Origami Studio[10]). FlowHub[11] uses FBP to create distributed data processing systems and internet-connected artworks, while XOD[12], Microflo[13], and iStuff [5] employ FBP for maker-friendly embedded development. However, these do not take advantage of the opportunities of seamlessness or liveness we identified.

[2]https://scratch.mit.edu
[3]http://s4a.cat
[4]http://blog.ardublock.com
[5]https://www.modkit.com
[6]https://developers.google.com/blockly/
[7]https://microbit.org
[8]https://calliope.cc/en
[9]https://cycling74.com/products/max
[10]https://origami.design
[11]https://flowhub.io/ide/
[12]https://xod.io
[13]http://microflo.org

Liveness in an IDE describes its ability to provide immediate feedback of the effects of code changes without the need to go through the edit-compile-run cycle explicitly, and has been shown to help developers [25]. It is an especially good match for embedded development because basic electronic components also process and react to incoming electric signals immediately. Hence, the hardware side of such projects is naturally "live". FBP is particularly suited to support liveness by continuously evaluating the current program graph and available input data and updating the output data of each node accordingly. With Quartz Composer[14], Apple introduced a live visual FBP editor to create graphics processing pipelines in its Xcode development environment. Facebook's Origami Studio, originally built on top of Quartz Composer, is a modern live flow-based IDE. However, these live IDEs do not support embedded development.

2.2 Hardware: Electronic Circuit Building Support

Numerous researchers have looked at supporting electronics design and learning. Avilés and Cruz [4] use a markerless Augmented Reality app on a smartphone to visualize otherwise invisible or hard-to-see information, such as the current flow through a circuit or the resistance of a component. Instead of adding an AR layer, Toastboard [18] instruments a breadboard to continuously sense voltage levels on each row. When the user duplicates her circuit in a graphical desktop editor, the system can detect errors and provide potential solutions, simplifying debugging the circuit. CircuitSense [54] automates this process further by electrically sensing and identifying a core collection of electronic components as they are plugged into a small breadboard. This lets the system derive a virtual representation of the populated breadboard for the popular Fritzing [24] circuit design software, which also helps share the circuit online.

TinkerCAD[15] is a web-based design tool that can also simulate, among other things, Arduino microcontroller boards. This enables makers to create and test circuits online, by programming the virtual Arduino using regular code or using a block-based language similar to *Ardublock*[16]. Another web-based simulator is Wokwi[17] which can simulate various microcontrollers and provides code for working examples. The editor provides suggestions when coding and is capable of including external libraries. AutoFritz [27] supports users building virtual breadboard circuits using an auto-completion approach to help avoid mistakes. Similarly, CurrentViz [53] measures and visualizes current flow in the user's circuit ubiquitously, and CircuitStack [50] replaces the cluttered and error-prone jumper wiring on a breadboard with a custom inkjet-printed conductive sheet, with traces derived from the schematic, to place beneath the breadboard. Lin et al. [26] used semi-structured interviews with printed circuit board (PCB) designers to gain better insight into their work practice and identify challenges and opportunities for future PCB design tools.

2.3 Bridging the Hardware-Software Gap

This third challenge of simplifying the understanding of code and electronics in embedded development is what Flowboard also aims at. One way to address it is abstraction: TAC [2], for example, lets the user specify the desired system response to particular inputs abstractly in a flow-based graphical editor, then presents a set of possible circuit designs that implement that behavior, complete with matching embedded textual source code, from which the user can choose based on criteria like component cost, etc. The user then just needs to build the chosen physical circuit. While convenient, this hides the actual electronics and embedded coding details from the user, making it unsuitable for learning those concepts. Similarly, Scanalog [47] uses programmable analog hardware and a

[14]https://developer.apple.com/documentation/quartz
[15]https://www.tinkercad.com
[16]http://blog.ardublock.com
[17]https://wokwi.com/

flow-based desktop UI to let users interactively design and tune analog signal processing circuits, supporting assertions alerting the user to conditions like exceeding a voltage limit. Here, the user no longer creates a circuit from discrete hardware components, again limiting its use in learning.

Bifröst [33] instead combines an instrumented breadboard with code instrumentation to provide a side-by-side view of electrical signal activity, Arduino source code lines, and variable values. While it does not provide an easier way to enter source code, it positively affects the embedded debugging process. Wifröst [34] successfully expands this approach also to integrate debugging support for network code. Finally, without leaving the imperative programming paradigm, ElectroTutor [51] provides support to follow a test-driven development approach across both software and hardware development, with positive impact on users' ability to complete introductory tutorials.

2.4 Studies of Learning Embedded Development

Improving the learning experience with a tool requires gathering knowledge of its impact through studies with learners. Below, we highlight relevant related work that illustrates how to extract such knowledge in the domain of learning to code, particularly for embedded development.

In the *'Art' of Programming* project, Moskal et al. [37] requested study participants (first-year students) to draw what they imagine when asked, "What does programming mean to you?" The

	Programming Paradigm	Conveyed Skills	Supports Live Coding	Seamless
Toastboard	Imperative	Electronics	No	No
CurrentViz	Imperative	Electronics	No	No
CircuitStack	Imperative	Electronics	No	No
CircuitSense	Imperative	Electronics	No	No
LAWRIS	Imperative	Programming	No	No
Bifröst	Imperative	Both	No	Some
Wifröst	Imperative	Both	No	Some
ElectroTutor	Imperative	Both	No	Some
Microflo	Flow-Based	Electronics	No	No
XOD	Flow-Based	Programming	No	No
TAC	Flow-Based	Both	No	No
Scanalog	Flow-Based	Electronics	No	No
Flowboard	Flow-Based	Programming	Yes	Yes

Fig. 2. A comparison of related projects that aim to help with embedded development. For each project, we list its underlying programming paradigm, where its main focus for support lies, and whether it supports liveness and seamlessness.

authors found that there was only one person in the image in most drawings, suggesting that social aspects were not essential and people mostly program on their own. The idea of pair programming seemed not to be present, although working in groups is known to promote discussion about the task. We adopted their drawing method for our second study.

With Physical Computing (building embedded systems with sensors and actuators that can sense and influence their physical environment) [39] having become a part of STEM education, Przybylla and Romeike [41] developed requirements for tools to explore this topic in schools. They found that while students enjoy the experimental approach of Physical Computing to get in touch with electronics, current practices may emphasize making and tinkering with electronics too much for a structured learning experience.

Dasgupta et al. [15] focus on *Remixing*, the process of reusing and reworking existing artifacts. They consider remixing important and powerful for learning, especially in social interactions. Their results from a study using Scratch indicate that remixing programming code helps learners to develop computational thinking concepts. Other research projects using Scratch [16, 31] also show the importance of online communities and sharing knowledge with others for a better learning curve. Based on Piaget's learning theory [49], Bergner et al. emphasize the need of working actively with a system to learn it properly [7], which supports the idea of integrating Physical Computing into education. Maloney et al. [29] found that a live environment without run/edit loops such as Scratch supports users exploring and tinkering while programming.

More recently, a study [13] investigated how programming physical computing devices is impacted by live programming. It compared programming the BBC micro:bit[18] device either with the MicroBlocks[19] live programming environment or the default MakeCode [20], The authors found that non-live environments can become used less as users expect liveness while performing embedded programming. Liveness impacts children who are programming embedded components, and the authors emphasize the importance of understanding the impact of liveness on learning physical computing in future studies.

In all, our review of related work indicates that the potential of flow-based programming to support embedding programming in a more integrated, seamless, live, and possibly collaborative manner has not been fully explored yet (Fig. 2). This motivated us to study this potential further by designing, implementing, and studying the Flowboard system described in the following sections.

3 FLOWBOARD DESIGN AND IMPLEMENTATION

Below, we describe the conceptual design of Flowboard, its system architecture, implementation, and a usage scenario. Full details for replication, such as source code and hardware design files, are available in the open-source documentation[21] *(archive also provided as supplement).*

3.1 Flowboard: Conceptual Design

Flowboard's hardware and software design is based on the concept of flow-based programming. The list for requirements defined that Flowboard should consist of simple and commercially available materials, integrate live coding, link electronics and programming seamlessly, and be easy to replicate. Our goal was to support learning in an explorative way, by enabling people to learn about embedded programming without predefined tasks and steps [44].

Flowboard offers a visual, flow-based multitouch editor on a large iPad for coding. Program graphs are created from a library of processing nodes that each process data depending on their

[18]https://microbit.org
[19]https://microblocks.fun/
[20]https://scratch.mit.edu
[21]https://hci.rwth-aachen.de/flowboard

type. Nodes are connected by drawing virtual wires between node outputs and inputs, along which data flows. Electronic components are plugged into two breadboards next to the iPad. The left breadboard is used for buttons, sensors, and other input devices. The right breadboard is used for LEDs and other actuators and output devices. This supports a unified left-to-right data flow from the input electronics through the program graph to the output electronics, corresponding to the established direction of signal flow in electronic circuit diagrams.

The program graph is evaluated constantly, and drives an Arduino Uno board hidden beneath the iPad. The user does not need direct access to the Arduino board since its I/O pins are brought out twice, once on each side of the iPad, so each pin can be used for input or output as needed to keep an overall left-to-right signal flow. These hardware I/O pins are placed directly next to their virtual counterparts in the visual editor, creating a more *seamless* view of the entire system, and removing a frequent source of errors in embedded development. Changes in the program graph take effect immediately, creating a *live* IDE to match the live behavior of the electronics hardware.

3.2 Flowboard: System Architecture

The Flowboard system consists of the editor running on an iPad, an Arduino board, two breadboards, and the *switchboard*, a custom printed circuit board that ensures that each Arduino pin is only connected to either the pin on the input or output row at any time using a series of solid-state switches controlled by another microcontroller (Fig. 3). The iPad editor controls the Arduino and the switchboard via Bluetooth.

The Arduino is running our modified version of the Firmata[22] protocol sketch. With Firmata, the iPad editor can set and read Arduino pins, and trigger more complex operations like controlling a servo or reading an I^2C device, through serial commands sent via Bluetooth. In addition, we extended the protocol to cover additional electronic components. Similarly, the iPad editor also controls the switchboard using Firmata. Using a real-time protocol like Firmata enables Flowboard to become a *live* environment. Unlike in other embedded FBP environments, such as MicroFlo or XOD, in Flowboard, there is no need to compile and upload the current program graph to the Arduino each time the code is changed. Changes to the program graph are live immediately because the iPad editor interprets the graph continuously, sending corresponding Firmata commands to the Arduino to achieve the appropriate behavior. The device then computes the function the corresponding node implements on the updated arguments. One or possibly more results of the computation are forwarded to the node's output in the process.

Of course, there are also downsides to this system design: The Arduino cannot be disconnected from the iPad to run the program graph in standalone mode, as is possible with standard embedded IDEs. There are also limits to the real-time performance that the Bluetooth connection supports. For our use cases of learning and beginning development, these issues were not critical. However, we discuss possible ways to also generate Arduino standalone code under *Future Work*. Flowboard aims to facilitate exploration of embedded programming for learners, which usually revolve around small and simple coding projects. Nevertheless, Flowboard's conceptual design allows integrating any other microcontroller that can provide pins to the user. The design and software (e.g., the Firmata protocol) would just need to be adapted regarding the number of pins and their characteristics (like analog or digital).

Flowboard is also not 'Turing-complete', but for its intended usage this was not our goal. Still, the embedded system can be extended to commands and calculations the iPad can offer. Therefore, the power is not limited by the Arduino but can be as complex as the iPad is capable of, using its processing and memory resources. Of course, the output of Flowboard is limited to what the

[22]https://github.com/firmata/protocol/blob/master/protocol.md

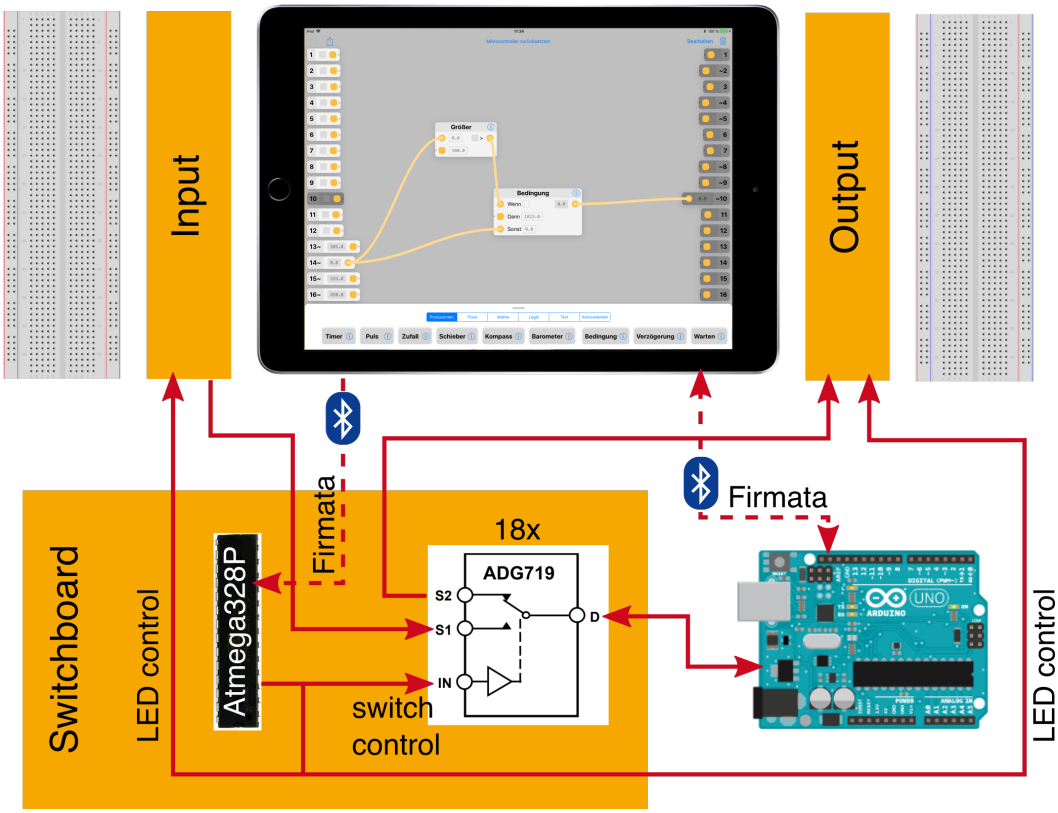

Fig. 3. Flowboard system architecture. The Arduino pins are brought out to the input and output side of the Flowboard via the switchboard, which controls the connection to either side. The iPad runs the visual flow-based editor, and communicates with both the Arduino and the switchboard via Bluetooth using the Firmata protocol [11].

Arduino can control, but a complex command running on the iPad could be black-boxed again to make the Arduino able to show the result, e.g., using a small screen.

Finally, Flowboard currently does not provide options for traditional troubleshooting of code, but these could be adopted from other visual FBP enviroments.

3.3 Flowboard: Hardware Implementation

We chose to run our editor on a large tablet instead of a desktop because the resulting system lies flat on the table when used. This allowed for a system design in which users place the electronics directly adjacent to their program graph, creating a more seamless development space. Touch-based interfaces have also been found to support more natural interactions that can support certain learning activities [19, 22], and multitouch supports the collaboration of collocated users more readily than a single-user desktop app [35].

3.4 Flowboard: Language, Visual Editor and Layout

Flowboard provides access to all digital and analog I/O pins on the Arduino except D0 and D1, which are required for serial Bluetooth communication. However, most Arduino projects avoid using

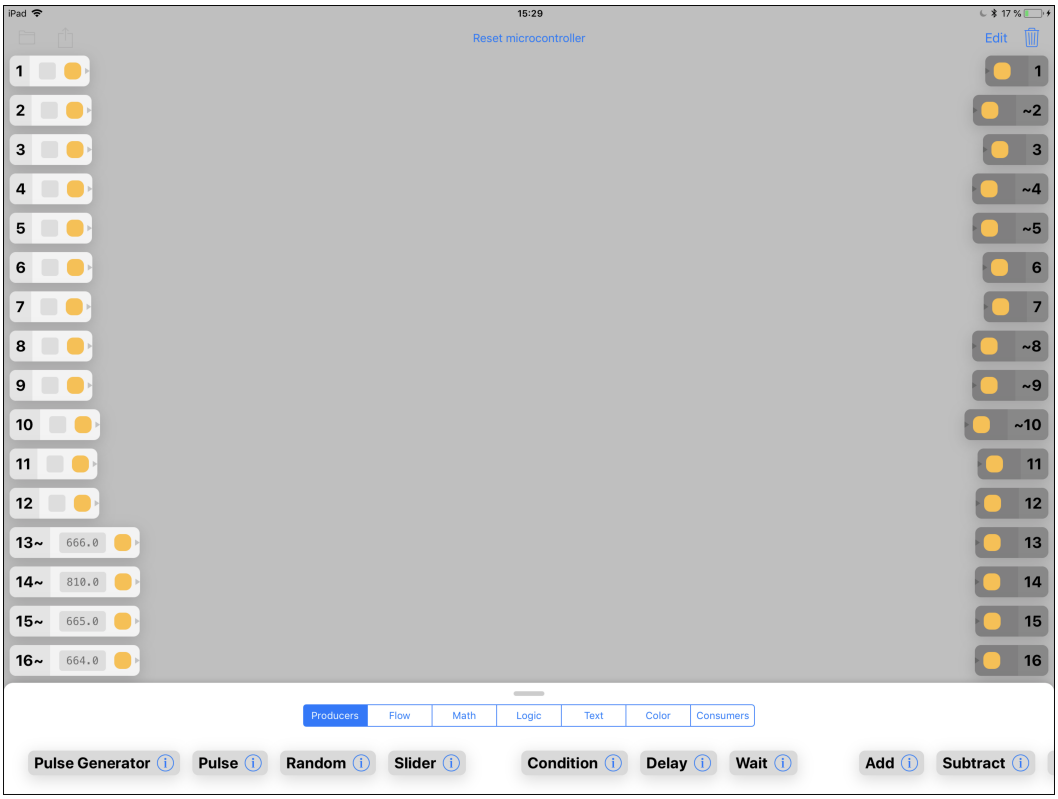

Fig. 4. Flowboard's editor after launching. Virtual I/O pins are visible on the left and right. The bottom menu is only visible while the user picks a node to add to his program [11].

these pins for precisely that reason. By default, the switchboard connects all I/O pins to the input side, matching the behavior of the Arduino and its ATmega microcontroller. If a user connects to a virtual output pin in the visual editor, the editor automatically instructs the switchboard via Firmata to toggle the corresponding electronic switch. Green LEDs on each pin indicate which direction is currently active. No explicit user action is required for this switching. The two breadboards have their power rails connected to the Arduino to power the user's circuits. For untethered use, especially in the classroom, the Arduino, switchboard, and breadboard circuits can be powered through a rechargeable power bank in the frame connected to the Arduino power jack. Alternatively, the system can be powered via the Arduino USB connector. Since the iPad turned out to last long enough for our sessions, it is not currently connected to the rest of the circuit electrically, although charging it through the power bank is possible.

Below the screen, a hardware toggle switch lets the learner disconnect power from the breadboards while inserting or removing components, reducing the risk of short circuits. The Arduino is not powered down by this switch to avoid disrupting the live processing of the Firmata sketch running on it (an even safer design is discussed in *Future Work*). The Flowboard case is lasercut from plywood and consists of three layers. Its bottom layer contains the custom switchboard circuit board, cabling, and battery. The middle layer supports the iPad and breadboards. The top layer has cavities for the breadboards, the external pin row connectors, a power switch, and the iPad itself.

The three layers are held in place by the walls of the case. The rear wall has a connector for USB power and a power switch to turn off the power bank after untethered use.

Our Flowboard programming language is most similar in scope to other existing embedded FBP IDEs such as XOD[23] and Microflo[24]. The UI layout differs from other tools but the basic concept behind the nodes is similar. Besides standard nodes for basic mathematical and logical processing functions, it includes nodes to work with more complex electronic components, such as RGB LED strips using WS2812 controllers, servo motors, and serial OLED displays using I^2C communication. Initially, the editor displays an empty canvas to place nodes and connections on (Fig. 4). Virtual representations of the I/O pins appear on both sides, aligned with the corresponding hardware pins next to the iPad. A virtual pin is greyed out if the physical pin is currently connected to the other side by the switchboard. Active pins are thus indicated both physically by their green LED and virtually by not being greyed out on screen. Behind the scenes, the iPad does not program the Arduino by uploading code as usual, but interacts with it by sending configuration commands and receiving results over a wireless serial interface. To maximize screen space for the program graph, the node menu on the bottom is visible only while adding a node. The menu contains shortcuts to categories for experienced users, but scrolling sideways shows all available node types. The visual editor was implemented in the Swift 4.2 programming language as an app for iPad tablets running iOS 12 or later. The user can interact with the editor via touch, providing an easy way to arrange nodes on screen and connect them with each other and the virtual pins. The editor space can be zoomed out to have more space for nodes and zoomed in again to work on single nodes more easily.

3.5 Programming Primitives

Resnick and Silverman [43] stated that the programming primitives—the 'black boxes' chosen—have to be designed carefully, as these determine what is visible to novices, and what they can explore and understand with them. Flowboard's primitives are block-shaped nodes consisting of predefined input and output ports. These inputs and outputs have a name and type (typical data types such as string, integer, etc., as used in traditional programming languages like Java), but the type is hidden from the user. Each node is designed to fulfill a function, e.g., an *if* condition (Fig. 7), and presents all elements (such as parameters) required to use the function. The user can configure these elements via touch input. On the bottom, Flowboard's editor provides a menu holding the library of nodes that can be dragged onto the editor's canvas (Fig. 4 and Fig. 5). The output values of a node are the results of its internal logic: a computation that uses the node's respective values at the input ports. Nodes are immediately live when appearing on the canvas and are initialized with default values.

The editor provides well-established programming primitives, such as an *if* statement and *greater than* comparison, but also includes nodes designed specifically for talking to electronic components such as a servo motor or buzzer (see Fig. 7). In total, Flowboard currently offers 25 node types in five categories for programming. Users can connect node ports and pins by drawing lines either between an input and output port, between an input pin and input port, or between an output port and output pin (see Fig. 5). For this, both drawing directions are possible. A connection between two components can be overwritten by drawing a new connection starting from an output port. When these connections are established, data flows from the output to connected inputs. An output can connect to multiple inputs, but inputs only accept one connection. Small triangles next to a node's port visualize the direction of the data flow. Flowboard provides a type-safe snapping mechanism, i.e., only valid connections can be established. Flowboard's editor provides explanations for node

[23] https://xod.io
[24] http://microflo.org

Fig. 5. Flowboard's editor for the programming task: make an LED shine weakly resp. brightly, depending on whether a force sensor exceeds a threshold. In comparison to figure 4, pin 10 is greyed out as input pin on the left as it is used as an output pin on the right [11].

functions via an 'information' button next to each node type in the menu. These buttons explain in detail what the node type is capable of, including usage examples with sample input and output values.

3.6 Flowboard: Usage Scenario

Michael and Anna, two 14-year old students, found an "electronic dice" circuit online that uses LEDs to indicate a random number rolled. For their favorite board game, they would like to build their own variant that can roll numbers from 1 to 10 instead of 1 to 6. They have had a basic programming class at school but found the complex language and many syntax rules intimidating. They have heard of Arduino boards and are interested in learning how to use one for their project.

To get acquainted with Flowboard, they first follow a "Getting Started" tutorial to create a circuit that lights up an LED when pressing a force sensor hard enough. They plug a force sensor and resistor in series into the left 'input' breadboard, creating a voltage divider, and use a jumper cable to connect the center of that divider to a physical input pin marked with a tilde (~) indicating that it can read analog voltages (Fig. 6 and Fig. 8). Since Flowboard is a live IDE, the virtual input pin on the screen edge immediately starts displaying the voltage applied to that physical pin as an integer.

Anna presses down on the sensor, watching the numbers on the virtual pin, and the friends determine a value of 500 as a suitable threshold to turn on the LED. Next, they add a Greater-Than node in the

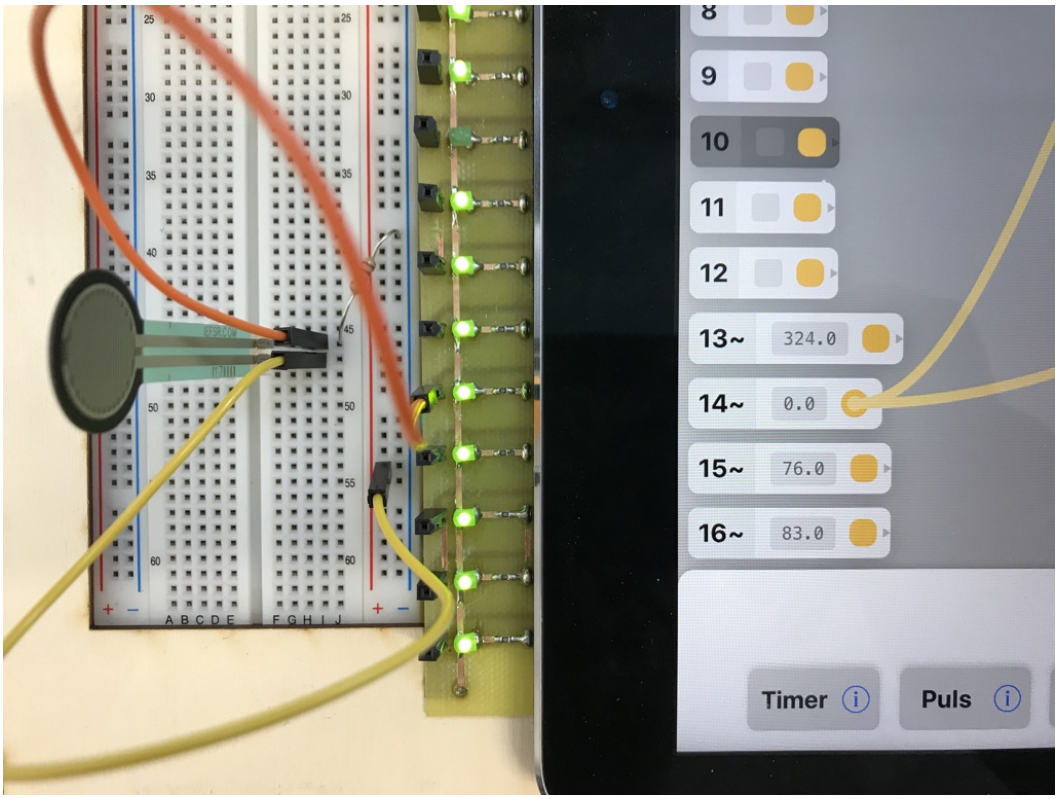

Fig. 6. Close-up of a force sensor connected as input to pin 14. Unlike the other analog pins (13, 15, 16), pin 14 is not floating anymore because it is connected to a defined voltage level. Pins 17 and 18 are currently hidden by the menu in the editor. When the menu is closed, they become accessible again. This example is from a programming task, and shows connections from pin 14 to two different nodes in the editor, as used in Fig. 5.

editor, drag a connection from the virtual input pin on the screen edge to the node's input, then tap on the node and set its threshold value to 500. Similarly, they connect the node's output to a virtual output pin on the right. Now, as soon as they connect an LED and resistor to that physical pin, their circuit is complete and working. This encourages Michael and Anna to continue experimenting with Flowboard to gather enough knowledge for their dice project.

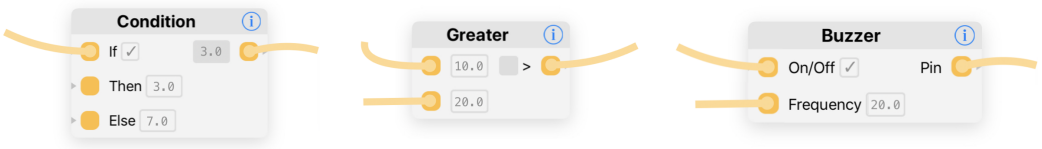

Fig. 7. Three example nodes included in the library of the Flowboard IDE. The *Condition* node (left) currently outputs 3.0 if the condition is true (represented by the checkmark) and 7.0 otherwise (then no checkmark would be visible). The *Greater* node (center) compares two values (Fig. 10 explains the *Greater* node in detail). The *Buzzer* node (right), when turned on (visualized by the checkmark), produces a square wave with a frequency of 20 Hz at its output, which users could connect to a piezo speaker via a Flowboard output pin.

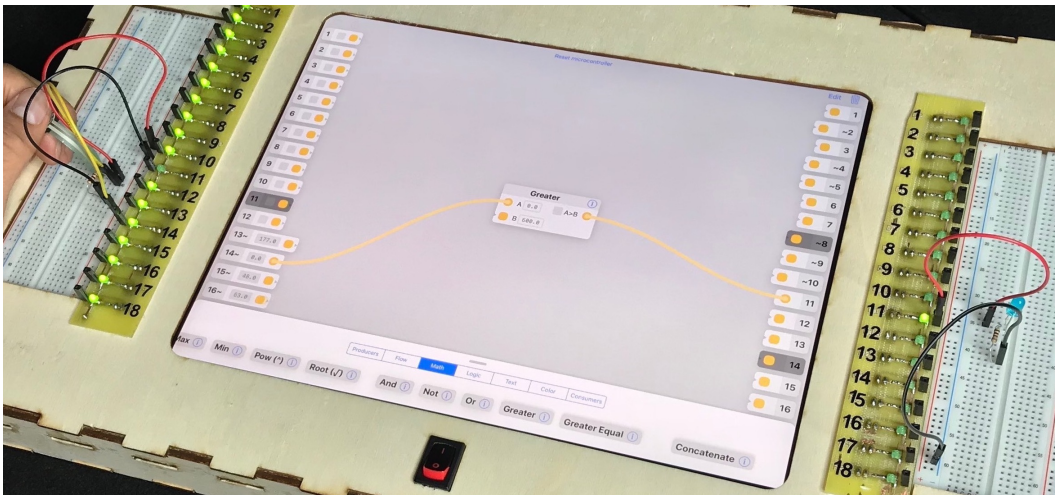

Fig. 8. Building the circuit from our scenario in Flowboard. The students have hooked up a force sensor and an LED, and created a program graph to turn the LED on whenever the input force exceeds a threshold.

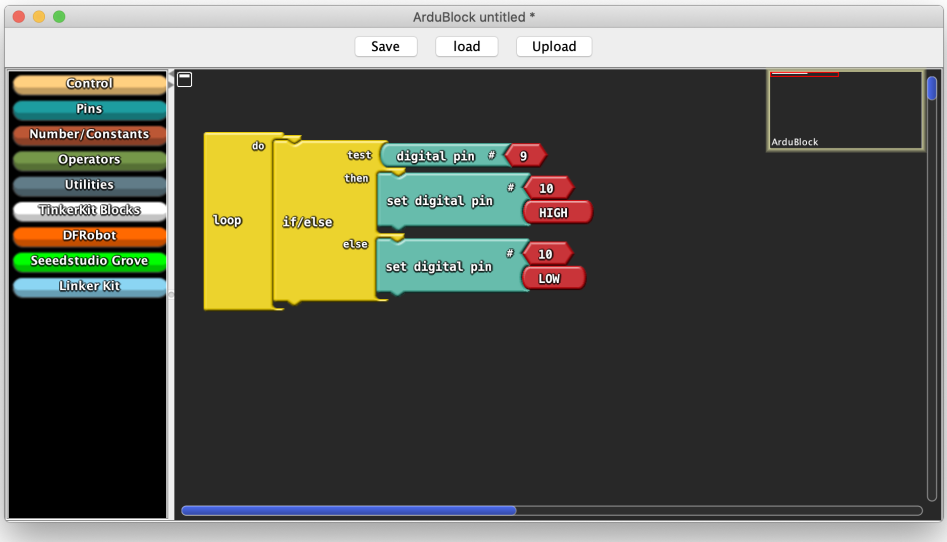

Fig. 9. The Ardublock IDE showing code for a pushbutton that controls an LED. In this program, the yellow blocks represent controls (here the *loop* method and an *if* statement), the green blocks are used to define the pins as input or output pins, and the red blocks serve to program the action of the pins. The blocks of the Ardublock editor can snap into each other to assemble a program for the Arduino. Block shapes provide a visual cue which other blocks they can be snapped into.

4 STUDY 1: EXPLORING EMBEDDED FBP

With our first user study, we wanted to see whether FBP could be a viable paradigm to learn basic embedded coding if an IDE made good use of its conceptual opportunities for seamlessness and liveness. We also wanted to see how learning with such a tool differed from using an IDE in the traditional imperative programming paradigm. For this reason, we used both Flowboard and Ardublock (see Fig. 9), an imperative IDE for Arduino used in STEM education and a close competitor in the field of learning embedded coding, in a between-groups user study. Ardublock uses a block-based visual code editor similar to Scratch, which brings it closer to Flowboard's visual FBP editor than a text-based IDE would have. However, the goal of including Ardublock was not to look for quantitative benchmark comparisons between both conditions but rather to have Ardublock serve as a conceptual lens to frame FlowBoard's usage.

4.1 First Study Design

To see if Flowboard was a viable option required avoiding learning effects, which led to a between-groups study design. Both groups conducted 1-on-1 workshops with one of the authors (two authors in total), one participant at a time, building and programming basic Arduino projects, such as making an LED blink. The two authors conducted both conditions to assure the results were not biased by only one author executing one condition. To ensure that our research followed established ethical standards, we planned both studies taking the ACM guidelines as well as specific lessons from the Interaction Design and Children (IDC) community into account, and consulted with teachers and experienced researchers whose feedback informed the design of our studies. Parents received a consent form beforehand, which they had to sign to let their child take part in

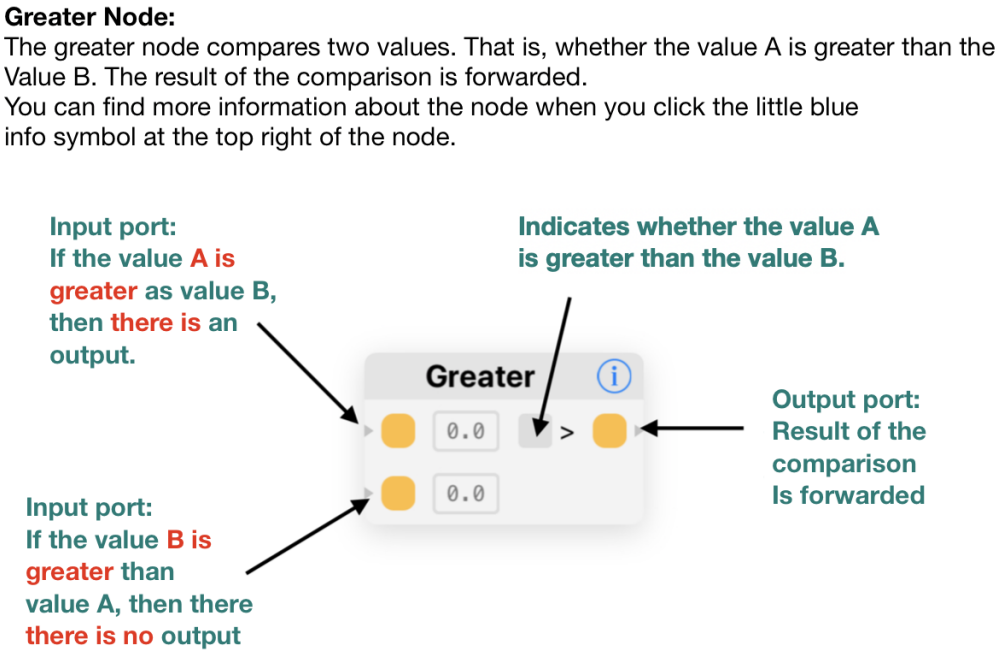

Fig. 10. Example of the explanation of nodes on the cheat-sheet from the Flowboard condition.

the user study. Children could take breaks at any time and were encouraged to tell us whenever there were any problems or they felt like stopping the user study. Total session time was limited to one hour. In the Flowboard setup, the complete Flowboard kit, cables, sensors, and other required components were laid out on a table in front of the participant (Fig. 12). In the Ardublock condition, the setup consisted of a standard tutorial kit with an Arduino Uno, a breadboard, and the same sensors, components, and cables (Fig. 13). Participants were allowed to ask clarifying questions at any time. We decided to let participants not only program but also build the electronic circuits. For this, they received an instruction sheet with the circuit schematic. This user study design was important since the goal of Flowboard is exploring the fusion of hardware design and programming. While we focused on understanding the impact of FBP on the programming task, building the circuits is an important part of the experience and was therefore considered essential to evaluate Flowboard for embedded programming.

We walked participants through the three guided tasks: (1) mapping a push button directly to an LED, (2) making an LED blink, and (3) making an LED blink while the pushbutton is held down. It was a conscious decision that the second task only involves one electronic component, while the first task involves two components. Our pre-studies showed that having no (hardware-based) input for a component confuses people with no experience when working in an FBP context. This is because, in such cases, the flow of action starts with a node in the editor and not on the breadboard as usual. Because of this, we let them connect a button to an LED first so that they gained a basic understanding of the workflow and got to know the editor and its node types first.

After the three guided tasks, they were asked to complete two assignment tasks on their own: (1) make an LED shine weakly resp. brightly, depending on whether a force sensor exceeds a threshold; and (2) buzzing a piezo speaker if the sum of all three outputs from an accelerometer exceeds a threshold. They received one task and the matching cheat sheet at a time. The cheat sheet included all nodes (Flowboard condition (see example in Fig. 10) or blocks (Ardublock condition (see example in Fig. 11) that the participant needed for the current task, with a description of each node or block. This way, participants knew which elements they needed to use, but not how many of each, or where to put them in the program.

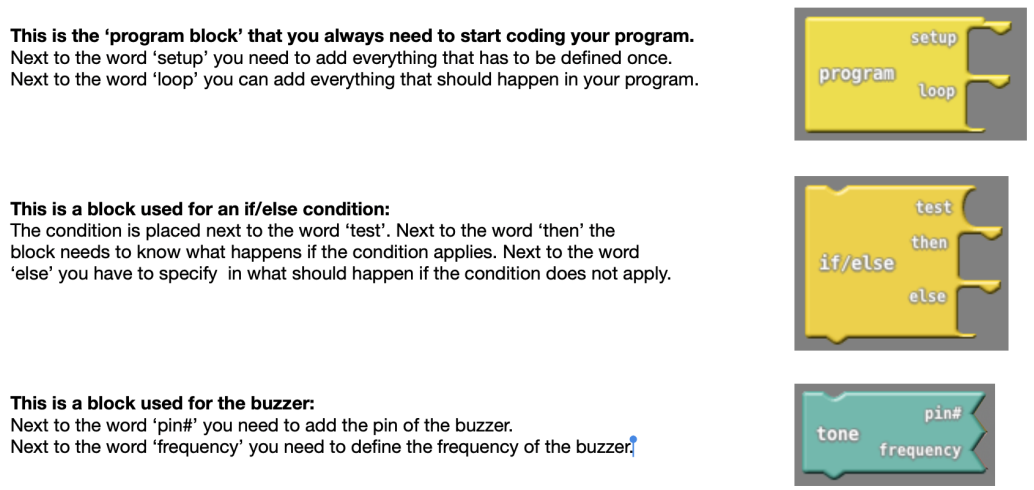

Fig. 11. Example of the explanation for the blocks on the cheat-sheet in the Ardublock condition.

Participant	Condition	Age	Gender	Experience with Tools
P1	FlowBoard	15	Female	Scratch
P2	Ardublock	13	Male	No Experience
P3	FlowBoard	10	Male	Lego Mindstorms
P4	Ardublock	11	Male	No Experience
P5	FlowBoard	12	Female	No Experience
P6	FlowBoard	12	Female	S4A, Scratch
P7	FlowBoard	14	Male	Scratch
P8	Ardublock	12	Female	No Experience
P9	Ardublock	14	Male	Cold Cadim, Scratch, Arduino IDE
P10	Ardublock	15	Male	Scratch

Table 1. Overview of participants of the first user study.

We debugged our study design and the Flowboard user interface in a **pilot study** with this setup: Three teenagers (two female, mean age 17, SD 1.0, two with Java or Python experience, one also with Arduino) went through our study design using Flowboard. After a general introduction to Flowboard and handing out a user manual with basic instructions and examples, we gave them one task after the other, with increasing complexity, each with the cheat sheet listing the nodes needed for that task. All three completed their session, including introduction, three guided tasks, and two assignment tasks, within the limit of one hour. Based on the observations in this pilot, we improved details of the Flowboard UI, such as highlighting matching inputs based on input type when connecting nodes, and letting participants replace existing connections between nodes by dragging a new connection, without having to delete the old one.

In the subsequent full study, ten participants (4 female, aged 10–15, mean 12.8, SD 1.68) took part, five per condition. Five had experience with Scratch, one of them additionally with embedded programming using the original Arduino IDE (see Table 1). Participants were assigned to the conditions alternately. We tried to balance the level of programming expertise in both conditions, but recruiting turned out to be rather challenging, and the resulting groups are not balanced perfectly.

4.2 Measures

We measured whether participants successfully completed each task and how much time they needed. We captured all sessions on video to later help us determine the cause of any problems participants encountered (e.g., whether time was lost working on the hardware or in the visual editor). After their session, we asked participants what they perceived as the main challenges.

4.3 Results and Implications

Complex cognitive tasks like the ones in this study tend to have high individual performance differences, and our number of participants was comparatively small, so we did not seek statistically significant numerical results, but were rather interested in initial trends and insights to help us understand if FBP could be a viable paradigm to further explore for learning basic embedded coding.

All students in the Flowboard condition were able to finish the embedded programming tasks. In the Ardublock condition, 2 students were not able to finish within the allotted hour. The adjusted Wald Method [1] can be used to calculate confidence intervals for binary success tasks and is proven to be a precise estimate even with very few participants [45]. For Flowboard, the 95% adjusted-Wald binomial confidence interval is [0.599, 1.00], and for Ardublock [0.229, 0.884] respectively. The

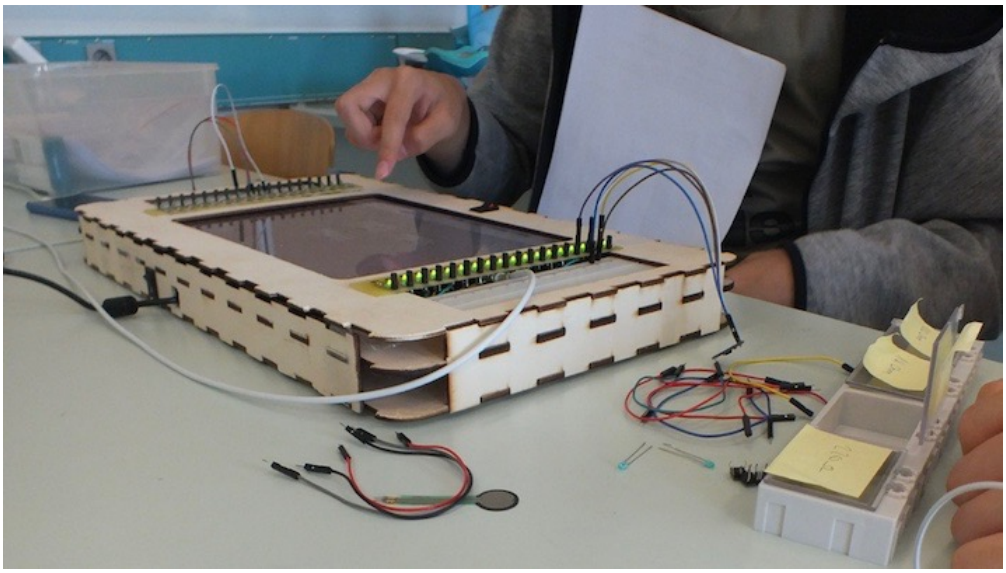

Fig. 12. User study in the Flowboard condition. The particiant is seen creating a program to control a piezo buzzer with an acceleration sensor in task 2.

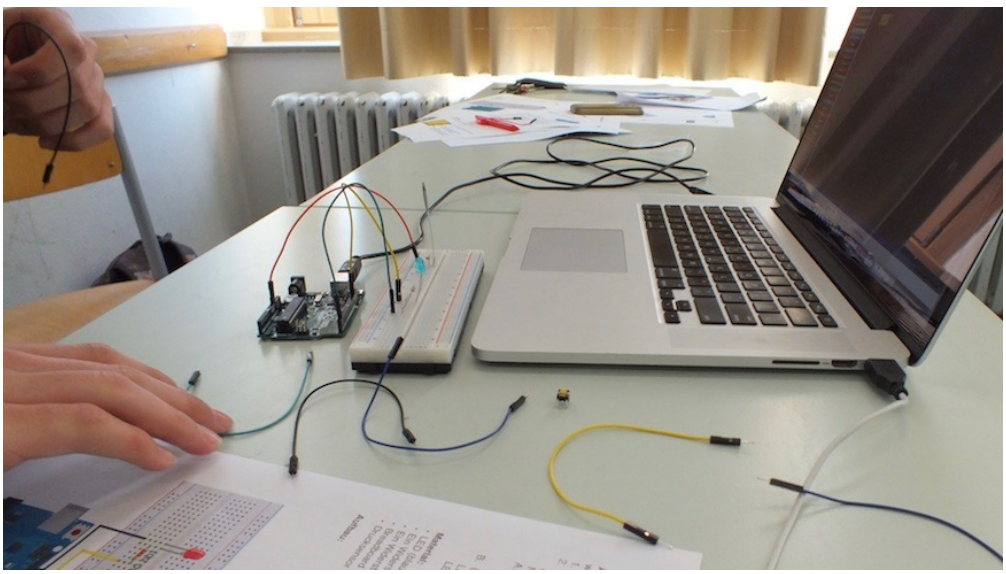

Fig. 13. User study in the Ardublock condition. The participant is currently connecting the force sensor and LED in assignment task 1.

fastest Flowboard participant finished in only 30 minutes, the fastest Ardublock participant in a little over 40 minutes. On average, Flowboard participants needed 6 minutes and 57 seconds for assignment task 1 (see Fig. 14). This was 2 minutes and 8 seconds faster than in the Ardublock condition. In assignment task 2, Flowboard participants were 18 seconds faster on average. In total,

Flowboard participants needed 14 minutes and 2 seconds on average to complete all tasks, which made them 2 minutes and 16 seconds faster than our Ardublock participants. Summarized, we found that Flowboard learners tended to complete their tasks in less time. This matches our observation that Flowboard participants were able to transfer what they learned in the guided tasks to the assignment tasks more easily. In both groups, we had participants (three in the Flowboard condition and two in the Ardublock condition) with previous experience using Scratch, an imperative, block-based programming environment extremely similar to Ardublock. Those participants were on average 3 minutes and 24 seconds faster with Flowboard than the experienced ones with Ardublock, while participants [P2, P4, P5, P8] without any programming experience in both groups took the same time for finishing their condition. The results do not produce statistical significance but encourage further research.

Additionally, this gave us confidence that visual flow-based programming could be a viable alternative to block-based imperative programming for learning embedded coding. In the Ardublock condition, we observed participants often snapping blocks together like puzzle pieces, based on their visual shape rather than their actual function, thus limiting learning transfer. P2 stated that he did not understand how the blocks of the Ardublock environment work, and that they should have better explanations (see UI in Fig. 9). Two other participants [P4, P8] mentioned difficulties finding blocks in the Ardublock menu. They tried to use the block color when searching, but the menu uses colors for its block categories that do not match the colors of the blocks in each category. In both groups, the participants struggled to build the electronic circuits correctly during the guided tasks. This was still challenging in the non-guided tasks, and mistakes mainly included wrong plugging of the cables and sensor legs into the breadboard. This is not an easy task when new to embedded programming and electronics. As building electronic circuits correctly was not the primary goal of the user study, the instructor helped the participants to build correct circuits.

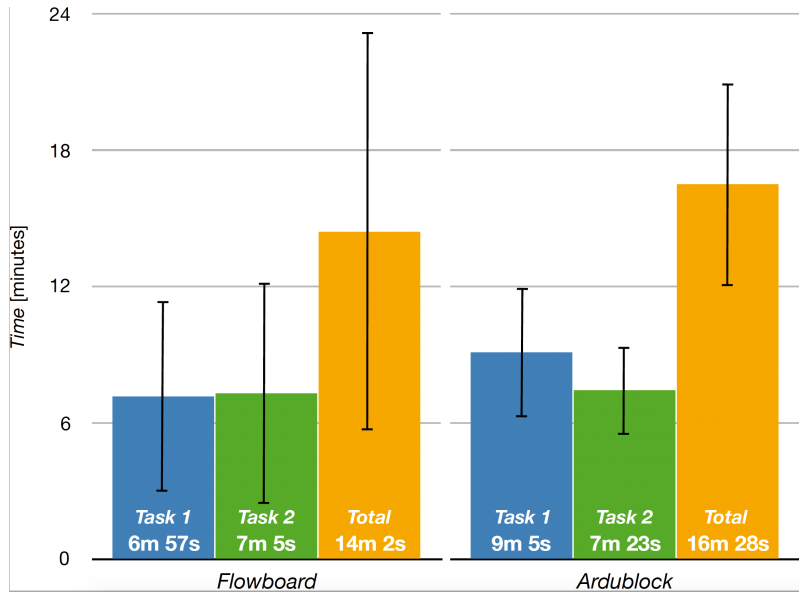

Fig. 14. Time to complete assignment tasks 1 and 2, and sum for both tasks, in the two conditions. In each case, using Flowboard tended to take less time than Ardublock, although individual differences between the users are too high to claim significance.

We observed that all participants in the Ardublock condition regularly forgot to press the *Upload* button in the IDE, each time waiting for a few moments until realizing it. P9 stated that "uploading is an annoying task". Flowboard's live visual feedback supported participants to execute faster iterations during exploration and coding. We learned that the direct link between physical and virtual I/O pins gave participants the feeling that they could connect input and output components with only one line. P7 mentioned that he "likes the direct connections". We observed that participants were making use of these direct connections very quickly. On the other hand, certain programming tasks, like creating a 'heartbeat' in FBP code to blink an LED, were challenging students more. We reflect on this in the discussion at the end. We also noticed that some students misinterpreted Flowboard's seamlessness, thinking that actual electricity would be flowing through the program graph from left to right. This suggested a potential downside of this seamless experience that a teacher or instruction materials could address, warranting further research. In our study, learners with prior programming knowledge, even outside embedded environment, tended to work faster with flow-based programming than using the block-based imperative programming environment. This suggests that transferring programming knowledge to a flow-based programming paradigm is relatively easy, and can help shorten the time needed for learners wishing to learn embedded programming. However, even for those with no prior coding experience, flow-based programming appears to be as suitable as the established imperative programming paradigm.

After this study, we were missing detailed qualitative insights regarding the influence of Flowboard. Therefore, we wanted to concentrate on qualitative aspects in the second user study. Overall, our first study showed the viability of FBP for learning embedded coding but also raised questions that we wanted to address in the follow-up study.

5 STUDY 2: STUDENT PERCEPTIONS USING FLOWBOARD

Our second qualitative study aimed at better understanding students' opinions and perceptions of Flowboard. Specifically, it was aimed at gathering insights about these questions:

(1) How do students picture electronics and embedded programming?
(2) How do the concepts of liveness and seamlessness help to facilitate the understanding of what is happening inside electronics in Flowboard?
(3) How does Flowboard help as a tool to introduce students to electronics sensor values and functionality of electronics?
(4) How does Flowboard encourage students to explore electronics and embedded programming on their own?

Our first user study had been conducted with one participant at a time. However, since the architecture and layout of Flowboard make it physically reachable from all sides, and since FBP makes parallel programs a natural choice, we decided to include the aspect of working in pairs. We assumed that working in pairs could support participants to be explorative when programming electronics with Flowboard. We followed established ethical standards as in our first study. E.g., parents had to sign a consent form before the study, and children could stop the study at any time.

Our study comprised of four steps: In step 1, we wanted to gather information about students' perceptions of the functional aspects of electronics and embedded programming. Therefore, we asked them to visualize their thoughts on these topics in a drawing task inspired by and adapted from previous research that investigated students' perceptions when learning to program [37, 46]. Step 2 was an embedded programming session with Flowboard, starting with a short introduction to the system, and finishing with letting participants code on their own to collect data about children using Flowboard, and its suitability and functionality. Step 3 repeated the drawing task from step 1 to elicit if Flowboard changed their perception of electronics and embedded coding. Step 4 was

a semi-structured interview for deeper insights into the students' thoughts. Those interviews included a reflection on their drawings, their general perception of embedded programming, and their experience using Flowboard.

5.1 Detailed Study Procedure

Our study lasted 50–60 minutes and was executed in groups of two participants. Only one and always the same author was leading all groups in the second user study. At first, participants were asked about their demographics, like age and experience with electronics and programming in general. Afterwards, participants watched a one-minute video about setting up the electronic circuit for an LED. An Arduino controlled the LED to blink, but the programming code was not shown in the video. This served to level the playing field somewhat with respect to prior knowledge of concepts such as an LED. Participants were invited to ask questions as needed. Then each of the two participants received a blank sheet of paper and was asked to create drawings individually about (a) electronics and (b) programming electronics, based on the following questions. They had roughly 10 minutes for this task.

(1) How do you picture what happens inside the electronics to let the LED blink?
(2) What do you imagine programming the electronics looks like?

The middle part of the study consisted of an introduction and exploring Flowboard (step 2). After the introduction to Flowboard's functions and UI, participants solved three tasks using Flowboard:

(1) **Button and LED:** Turn the LED on with the button.
(2) **Pressure sensor and piezo:** Whenever the value of the pressure sensor is more than 500, the buzzer should sound at a different frequency than below 500. The buzzer node provides an input to adjust its frequency.
(3) **Pressure sensor and LED stripe:** Create different colors for the first three LEDs of the stripe using the values the pressure sensor produces.

As Resnick and Silverman [43] already emphasized, it is crucial to follow principles such as 'Make it as Simple as Possible – and Maybe Even Simpler' when developing construction kits for children. Therefore, we want to make clear why we chose these three tasks to understand what electronic principles and programming principles Flowboard can make visible to the participants:

(1) **Button and LED:** *True* and *False* are a basic concept in most programming languages, used, e.g., to evaluate conditions. Since this is a relatively simple construct, we used it for the first task. We can recreate this concept in an electronics context, e.g., a button has two states, circuit closed and circuit open (*True* and *False*). The participant decides via programming in which state the LED is turned on.
(2) **Pressure sensor and piezo:** The concept behind this task builds on the previous task by adding more states than just *True* and *False*. Programmatically speaking, we introduce the concept of variables. In an electronics context, we can map this concept to analog sensors. In comparison to the button, a pressure sensor has more output values. The participant maps these values to a certain behaviour by programming. Flowboard encodes the pressure sensor values into the range of 0 to 1023. We did not describe this range to the participants as we wanted them to figure it out on their own. As Flowboard updates the pressure input value in real time as participants apply pressure to it, they can feel which applied force corresponds to which pressure values. E.g., participants can interpret the value 500 as a medium force that the sensor can sense. This relates to the fact that Resnick et al. reported: "Kids generally have little difficulty learning to use imperative (action-oriented) commands (like **forward** and **on**), [...]".

(3) **Pressure sensor and LED stripe:** The last task is about transferring the gained knowledge from the previous tasks to a new context. This is important both in programming and circuit building. Since RGB LEDs have three input channels, we decided to use them as the new context. To ensure all participants understand how to create RGB colors, we explained that they can choose a value between 0 and 255 for each of the three basic colors to create their custom colors. We left it to the participants to figure out how to achieve full red, yellow or blue. Unlike task 2, participants define three values instead of one in this task. Additionally, the output generated was ten LEDs, not one, giving more room for creative visuals.

Basically, the tasks get more complex while displaying basic concepts for programming. They offer more options one by one for the participant to explore electronics and embedded programming.

One researcher served as an instructor in our user study. Participants were able to build circuits and program them as Flowboard aims to fuse hardware and programming for a complete experience. For each task, participants received a schematic, and the instructor helped them to build the electronic circuit, as Flowboard's primary aim is not to facilitate that aspect but experiencing the circuit building process is important for the overall experience. Helping included plugging cables and electronic components into the breadboards and explaining the schematics to participants. The instructor solved the first two tasks together with the participants to convey a feeling of success and to explain how programming in the FBP paradigm works. Of course, helping the users with the tasks can also influence their learning and hence also the analysis of the qualitative results. Therefore, we made sure that the instructor explained programming, electronic components and tasks always the same way. This established the same starting point for all users as far as possible and minimized the instructor's influence on the results. For the last task, the instructor gave the participants 10 minutes to solve it on their own, such that they had a chance to explore Flowboard independently. If they were faster than 10 minutes, they were allowed to program any component they liked to.

After the programming session, participants were asked again to draw their perceptions regarding (a) electronics and (b) programming electronics. The researcher then put the drawings produced at the beginning of the study next to these, and asked both participants to elaborate on their drawings and the differences between the two versions, if applicable. These reflections were important to gather insights about students' perceptions of embedded programming and how, if at all, Flowboard changed those perceptions.

After the conclusion of the drawing exercise and the experiment with Flowboard, semi-structured interviews (step 4) followed. We asked participants about their experience with Flowboard:

- What did/didn't you like about Flowboard?
- What was particularly easy for you?
- What values did the sensors give you, what was difficult to find out, what was easier?
- What did you learn about electronics today that you didn't know before?
- Would you also use Flowboard with friends?
- Is there anything you would like to build with Flowboard and what is it?
- What values did the electronics show?
- How did you find out what values they have?
- Was there anything you found to be rather complicated?
- Is there anything you would like to change about Flowboard?

At the end of the interview, students were given the chance to ask any questions they had about Flowboard.

Participant	Group	Age	Gender	Experience with Tools
P1	1	14	Female	No Experience
P2	1	15	Male	Arduino, Java, Python, Scratch
P3	2	15	Male	Scratch, Python, Arduino
P4	2	15	Male	Blender, little Arduino
P5	3	15	Female	App Inventor
P6	3	15	Female	No Experience
P7	4	12	Female	Lego Mindstorms
P8	4	11	Female	Scratch
P9	5	14	Female	No Experience
P10	5	14	Female	No Experience
P11	6	13	Female	Scratch
P12	6	13	Female	Scratch, BOB 3
P13	7	10	Female	Roboter (Dash), Lego Mindstorms
P14	7	11	Female	Roboter (Dash), Lego Mindstorms, Calliope
P15	8	10	Male	Calliope
P16	8	11	Female	Scratch, Calliope, Lego Mindstorms

Table 2. Overview of participants of second user study.

5.2 Pilot Study

We conducted a pilot study to test our study procedure and ensure participants would not become fatigued or overwhelmed by the tasks requested. For the pilot, two male students, both aged 15, who had experience with Arduino, Python, C, and Java, completed the study in 34 minutes.

5.3 Participants

Sixteen participants (twelve female, aged 9–15, mean 13, SD 1.9) took part in groups of two (see Table 2). The pairs for the study were formed by chance, i.e., whenever we were able to find two more children to participate in the user study, they were paired, and the study was executed. This happened for three groups. Five of the eight study sessions were conducted in school classes during the lessons. Each time, two volunteers were chosen randomly to execute the study together. Of course, only those children could participate who had our consent form signed by their parents. One group consisted of two males, five groups of two females, and two groups were mixed. They had varying experience in programming: Four stated not to have done any programming before, while three were quite experienced and had used various programming tools, including the Arduino IDE. In group two, both participants were very experienced. Compared to all other participants, they had the most experience. All other participants stated to have used tools such as Scratch, Bob3, MIT App Inventor, or others before but did not describe themselves as experienced, especially not with electronics.

5.4 Measures & Data Analysis

Drawings can help understand underlying mental models, especially children's [6]. Data collected by drawings has its limitations as it is dependent on the children's drawings abilities. But research of the last 25 years shows that children have "extraordinary skills in making meaning through the affordances of drawing" [32]. Researchers have previously applied drawing as a technique for investigating children's thoughts and understanding of technology [46]. Therefore, we followed

the methodology Selwyn et al. [46] applied in their research. We used open coding to analyze the drawings and defined four categories for sorting the drawings:

(1) *Electronics:* electronic components such as sockets, LEDs, bulbs, cables.
(2) *Basic Electronics Principles*: units and electrical quantities such as power, force, resistance, and electronic circuits, including symbols for components and current [8].
(3) *Programming Artifacts:* objects needed for programming represented in the drawings, such as hardware that children picture to be needed to program, e.g., computers.
(4) *Programming Principles*: any drawing that depicts a programming command, such as letter sequences, blocks or numbers. These commands, of course, can be of any abstraction as none of the participants were experts in programming.

Some drawings were assigned to more than o/drawings
, P7 drew a circuit and a light bulb for the category *Electronics*. In addition to identifying their general perception of programming, we aimed for finding aspects that students changed after using Flowboard.

Interviews were recorded on video and fully transcribed afterwards for analysis. Since part of our data was qualitative feedback and interviews, we analyzed our results using the coding software MaxQDA[25]. After preparing the interview transcriptions, we defined a code system to flag participant statements with summarising keywords. The four iterative coding cycles were performed by one person, with feedback rounds from others. In total, 121 sections were marked as valuable and were sorted and assigned to suitable categories. In the first two coding cycles, one researcher combined the first 19 themes into 11 themes; e.g., 'electronics placing hard' and 'nodes hindering advanced circuits' were joined to 'building circuits'. Afterwards, two more iterations were conducted with two other researchers to concretize and shape the categories into result themes: e.g., problem statements regarding the breadboards were merged into 'problems building circuits', statements referring to the fast system feedback were put together as 'liveness of components', and so on. We report on the findings of this categorisation in the interview results section below by aligning them with the focus questions we set out at the beginning of this study.

5.5 Findings of the Drawing Analysis

More than half of the participants (nine) changed their drawing describing their perception of electronics after the experiment. For the drawing concerning programming of electronic components, all but four participants made changes. Below, we report on the insights gained from analysing the drawings according to the four categories detailed in 5.4; examples of the drawings are shown in Fig. 15.

5.5.1 Picturing Electronics. When asking participants to picture 'electronics', eight drew components such as LEDs, cables, or electronic devices they knew from everyday life, like light bulbs or sockets. After the instructor had demonstrated Flowboard, only three drawings still pictured only electronic components. In the other drawings, parts of circuit and programming principles could now be identified, e.g., connections between electronic components.

5.5.2 Basic Electronics Principles. Basic electronics principles, such as electricity, closed electric circuits, or positive and negative terminals, were drawn by 11/16 participants. The total number of drawings showing electronics principles did not change after using Flowboard.

5.5.3 Programming Artifacts. When asked to visualize the term 'Programming Electronics', 10/16 participants drew objects like laptops, screens, or desktop computers. After the Flowboard session,

[25]https://www.maxqda.com

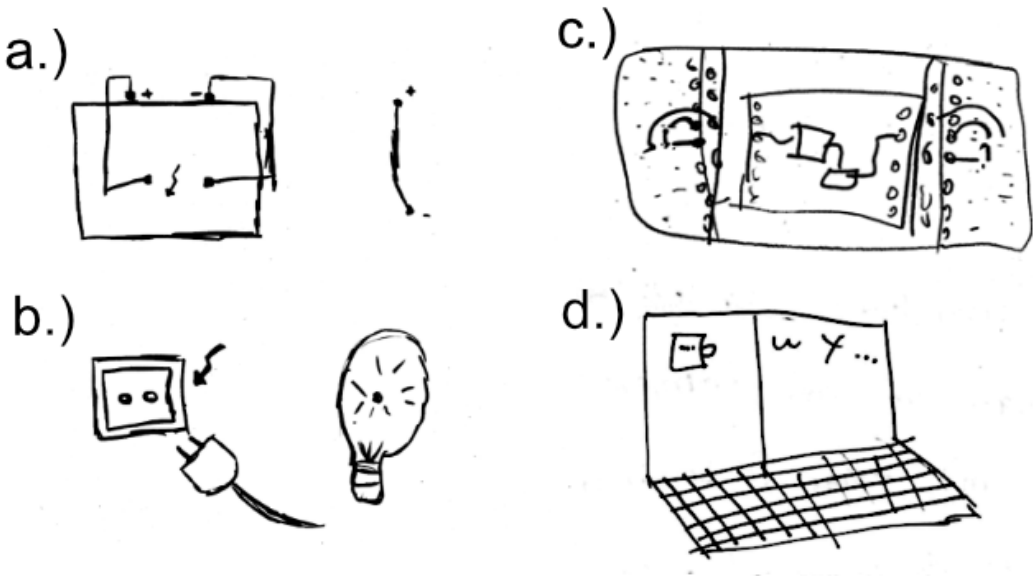

Fig. 15. Some drawings by participants [a.) and b.) were drawn for the question of how you picture electronics inside; c.) and d.) for how do you imagine programming the electronics]: a) A closed electronic circuit. b) Electronic components. c) Programming principle showing Flowboard. d) A laptop showing programming with letter sequences.

this number went down from ten to six participants. Four participants changed their drawings slightly, by focusing the drawing on the programming principle instead of the object to program the electronics with.

5.5.4 Programming Principles. Three quarters of all participants drew programming principles, e.g., programming commands or terms like 'numbers', 'letters' or any kind of letter sequences. This number did not change after the exploration session with Flowboard.

5.5.5 Summary of Drawing Results. Overall, the drawing task revealed that participants were unfamiliar with the flow-based programming concept prior to the study. For changing their drawings, participants named reasons that highlighted at least a fleeting understanding of flow-based programming:

"Before, I had no experience. Now I can imagine what happens inside, like, starting from the left the signal goes through the programming and on the right side the action happens." [P11]
"I also didn't really know what was going on in it beforehand. Now that I've done it, you can imagine what's going on there." [P12]

5.5.6 Interview Findings. The semi-structured interview took 10–15 minutes. We report on our findings below, categorized in themes according to the focus questions two to four of our study (see Table 3). Those are not questions we asked the participants verbatim, but research questions that helped guide our analysis.

Theme Index	Theme Name
1	Changes Pictures
2	Understandable
3	Liveness of Components
4	Beginner-friendly
5	Programming Alone
6	Programming Together
7	Support of Seamlessness
8	Improvements
9	Building Circuits
10	Connections of Nodes easy
11	Clear Arrangement

Table 3. Overview of the final themes.

5.5.7 How do the concepts of liveness and seamlessness help? One task in the study was to change the buzzer frequency when the force sensor value exceeded 500. When asking participants how they knew the value, fifteen commented along the lines of "you could see that" and explained that obviously, they could see that value at the virtual pin in the iPad editor. This points to a feature of Flowboard that can be attributed to its liveness and seamlessness:

"I liked that you can watch everything, also what is happening inside. And you can see changes when you modify the components on the iPad." [P3]
"You could see them [the sensor values] on the screen." [P5]
"Down here at pin 16, because the sensor is plugged in at 16, and when you press the sensor, you always see the value." [P14]

The participants P3 and P4, who had experience with the Arduino IDE, gave a reason why Flowboard is easy to use.

"With Arduino, you always need the serial monitor. Here I can see directly at the connection what is happening." [P3]
"To get started, you don't have to know the exact commands like with the Arduino IDE, and you can't make stupid mistakes like syntax errors." [P4]

5.5.8 How does Flowboard help as a tool? 10/16 participants stated in their interviews that Flowboard is especially suitable for beginners:

"Flowboard is practical for beginners because it is clear and easy [to understand without any programming experience]." [P9]
"Flowboard is great to achieve an overview of how everything works. [...] In a second step, you can build something with an Arduino Nano [because users can learn about the textual Arduino commands from the information dialogs of each programming node]." [P5]
"I think it is very simple. I myself have no prior knowledge, and if someone knows a little something, it might be a little easier. It's well-meant and as simple as possible." [P6]
"[I] liked that you had so many options on the iPad [...]. Making the connections was clear." [P5]
"Connecting [components] on the iPad with the yellow lines [was easy]." [P10]

Using an iPad made it easy to connect electronic components, and the use of the touchscreen was regarded favorably. Concerns were brought forward regarding Flowboard's case and usage within projects:

"The wooden case looks a little fragile. Maybe consider one out of plastic." [P8]
"Flowboard is too large to embed it somewhere." [P3]

The three features that were listed as most helpful were connecting nodes, using nodes, and the menu to find new nodes. The participants proposed improvements enabling students to use Flowboard on their own without an introductory session and creating shortcuts for advanced users:

"I would add a starting page, where everything is explained... this way you can work freely." [P14]
"For beginners, the nodes are handy, but for experienced users, it would waste time to look for each block in the menu." [P3]

Overall, 13/16 participants pointed out that the scenario of programming Flowboard together with another person is advantageous to explore and help each other:

"When a person sits next to you, you tell her when seeing a mistake. You have four eyes." [P5]
"[...] if the other [person] doesn't understand something, one can help." [P11]
"I like programming together as I am inexperienced and thus can ask the other person about things that are unclear." [P8]

Programming alone was preferred by only two participants, who found it easier to work alone, e.g., at home.

5.5.9 How does Flowboard encourage students to explore? Eleven participants stated that building something with Flowboard is possible, but most had no specific project idea in mind. Two more experienced participants (P2 and P5) remarked that Flowboard's size can be an obstacle when building integrated objects. However, Flowboard would be convenient for controlling electronics such as an LED strip to decorate your room:

"[...] when the temperature rises, the color of the LED strip changes." [P5]

Overall, participants found Flowboard suitable as a starting point to develop an idea, later switching to another environment that fit their needs, i.e., using Flowboard as a mock-up tool to test an early concept:

"Flowboard is great for getting an overview of how it all works exactly, and if you then want to go a step further and you then know all the commands, then you can read exactly how you have to write the commands and know where and how you use them to, like, in another step to build things with an Arduino Nano." [P2]

P1, P4, P12, and P13 mentioned they were unsure at first if they were able to solve any tasks at all, but found that Flowboard took away some of their concerns, as they found that the system provided clear arrangements and a good visual concept.

"Flowboard holds a clear arrangement [as opposed to Arduino]." [P4]

Most difficulties arose when building the electronic circuits on the breadboards left and right. Six participants reported that they either did not like this part or thought:

"The plugging in on the breadboard was hard." [P9]

We noticed that some students needed extra encouragement to read the instructions on each task sheet properly. With the low overall number of participants, this characteristic may not have been balanced across conditions, causing the difference. Additionally, we observed that floating analog input pin values confused participants as they did not understand whether the pin was broken and why there was a floating value.

To summarize, our key findings from the second study are:

- Seamlessness and liveness of the system provide cues for participants to understand electronic components.
- Perceiving sensor values on the iPad was easy and fast.
- Flowboard is suitable for beginners, and supports working in pairs, but is not considered ideal for complex projects.
- Participants suggested improvements, such as a built-in tutorial to support independent exploration and learning.

6 DISCUSSION & IMPLICATIONS

The above results indicate that Flowboard's concept provides helpful cues to facilitate learning about embedded programming. In the following sections, we will discuss the insights we gained from the studies in detail.

6.1 Effect of Liveness and Seamlessness

We were interested in whether the presence of *live* feedback had a particular impact on the learning experience. We learned that Flowboard's live visual feedback, e.g., when nodes or components like a pressure sensor were connected, supported much faster iterations during exploration and debugging, as no edit-compile-run cycles like in the Arduino IDE are needed [P3, P12]. Users found it easy to determine what data a sensor was delivering, and they immediately realized value changes at the virtual pins. This indicates that they were able to understand Flowboard's overall interface, and that perceiving changes was intuitive. Our FBP setup gave the users the feeling that the electronics were "alive", which is very suitable as a concept. It matches the behavior of basic electronic circuits, which also react to changes immediately. We consider this effect one reason to recommend FBP for embedded programming learning environments.

Some of our users even thought that the yellow connections in the editor were indeed cables carrying power, showing that the analogy was readily accepted, and could even be over-applied by users. They developed misconceptions about how the system worked because of Flowboard's design that hides the Arduino and puts the input and output sides of the circuit next to the iPad in a seamless manner. This effect, and ways to avoid such misconceptions, warrant further study.

Similarly, the live visualization of the floating voltage at an unconnected analog input pin confused participants, as they could not interpret what it meant. Following design principles as expressed by Resnick and Silverman [43], we suggest handling this as a "black box": Floating pins should be avoided in electronics, and they can be detected and flagged accordingly, as Toastboard

[18] demonstrates, which could remove the visual distraction and resolve this issue of "too much" liveness.

We also looked at the impact of Flowboard's seamless transition between hardware and software. We learned that the direct link between physical and virtual I/O pins gave users the feeling that they could connect input and output components with only one line [P14, P3]. Experienced users also mentioned that Flowboard could be an easier and faster option to learn about a sensor's value range than the Arduino serial monitor [P4]. These comments confirm our assumption that reducing the steps necessary to see live sensor data encourages exploration. Encouraging students to explore supports building up knowledge through "learning by doing".

Nevertheless, building the actual electronic circuits in hardware remained challenging. This illustrates the ongoing need for projects such as CurrentViz [53], which displays current flow in the user's circuit ubiquitously to let them detect problems with their circuit.

6.2 Mental Model

We also wanted to understand our participants' mental models of electronics and embedded programming and how using Flowboard would change these models. To this end, we used an established technique [21] of letting participants draw concepts of electronics and embedded programming, before and after using Flowboard.

Flowboard did not change the mental model of participants who already had experience with programming toolkits and whose model, not surprisingly, was quite specific already, e.g., understanding that LEDs can be turned on and off. Their perception of programming seems to have become manifested already through prior experience. However, it provided us with interesting opportunities to reflect on how their mental model differed from that of students with less or no experience.

We found that especially among those participants that had no experience, the typical picture of programming electronics was characterized by numbers and letter sequences. This indicates that the general programming model seems to be the traditional programming paradigm using textual commands, most likely informed by depictions of programming in the media. However, Flowboard still provoked wrong mental models. For example, some participants thought that the data for the output components came from the iPad and was not triggered by the connected input components.

After using Flowboard, participants added the nodes and lines that are used for programming in Flowboard to their drawings. Participants that had been introduced to programming in school previously drew block-based commands in addition to numbers and letters. This is likely due to schools using tools like Scratch, which seems to be the state of the art to introduce programming, based on our interviews and own experience in this area. Their reason for changing their pictures after using Flowboard mainly was that they had not seen flow-based programming before, and wanted to add it to their drawing for completeness.

Flowboard's system design of feeding sensor values into output components no matter whether they can be interpreted led participants to think about the meaning of those values. They wanted constantly working output components, and were keen to understand how to achieve this. Flowboard's system design made the values visible, helping them to understand that sensor and actuator value ranges can differ. It guided them to understand that values can be mapped to each other, and that sensors and actuators can have different sensitivity.

Five participants added programming principles to their drawing after using Flowboard, such as connections or cables between components or programming commands. This indicates that those participants' mental model consolidated electronics with programming, fusing both together. As embedded systems are permeating today's everyday environments, programming and electronics are indeed in a process of fusing. Whether this view is valuable in teaching embedded programming

needs further investigation. However, since physical computing is a part of STEM education, and since Flowboard seems to support bringing coding and hardware closer together—physically and conceptually—, we believe that any system teaching embedded programming should link the electronics very closely to their associated program code.

In some tasks, users had problems developing a correct mental model of the system. In study 1, we observed that tasks like using a 'heartbeat' node in the editor to let an LED blink without an input component were more challenging to complete than controlling an LED with a button. This is not surprising, as the FBP paradigm fits signal-processing tasks very well but is less suitable for modeling state and state changes internally, something that imperative programming supports more naturally through variables. Interestingly, some participants labeled Flowboard's input side as the 'beginning', the program as the 'middle' part, and the output side as the 'end'. However, defining a color for the LED strip without input in the third task was no problem, raising the question of whether the problem was the 'heartbeat' issue or not having an input component. Investigating this further, e.g., by conducting more tests with a wider variety of tasks, is one of our avenues for future research.

In conclusion, it should have become clear that our work has just begun to investigate the mental models of embedded programming that emerge while using a tool like Flowboard. We discuss some aspects of future work to unearth more of the potential of this avenue of research in section 7.

6.3 Impact on Learning Programming and Electronics

As mentioned in section 6.1, our findings indicate that Flowboard encourages exploration, a key element of approaching and learning a new subject matter [7, 28]. Our participants also deemed Flowboard suitable for beginners in embedded programming, and they consider Flowboard's layout less confusing than other similar tools [P4, P9]. Various comments also pointed out that the editor running on an iPad made it easy to connect electronic components, and the use of the touchscreen was regarded favorably [P14].

On the other hand, Flowboard uses a visual programming editor, which is further away from natural language than the blocks of text in textual imperative programs. However, we paid close attention to following the recommendations by Resnick and Silverman [43] to mitigate the adverse effects of a graphical representation. While we did not investigate the effects of visual vs. textual programming languages in our studies in detail, we did observe some effects of this representation, which we discussed above. For future work, we believe that a qualitative user study with that focus would be beneficial in developing additional design recommendations for embedded programming learning tools.

Some participants voiced concerns regarding Flowboard's case and the constructed circuits being fragile [P8] or not visually appealing [P12]. In contrast, the three features that were listed as most helpful were the way how nodes were connected by simply drawing lines between them [P9], using the nodes to program [P2, P3], and the menu to find new nodes [P5].

Compared to, e.g., an Arduino microcontroller, the Flowboard system can be too "bulky" to use for complex projects or permanent installations [P4]. However, our studies and related research projects [2, 33, 34, 47] indicate that it is the bar to enter embedded programming that needs to be lowered, and here, the default textual programming of, e.g., an Arduino IDE can be complicated for beginners. For example, Demetrescu et al. [17] combined procedural languages and dataflow, taking care that the coding process for the programmer is simple. Many tools use language blocks to address this challenge, but based on our findings, we recommend taking FBP into consideration when developing tools to learn embedded programming. A suite of Flowboard-based tools in various form factors for projects of different size and complexity could be a useful extension.

Our studies indicate that Flowboard provides a good starting point to learn embedded programming. However, the tool is currently tailored towards simple starter projects for learners. Slightly more advanced users could be accommodated better by, e.g., providing appropriate shortcuts for more efficient use [P3].

Our first study included FBP and imperative programming. The latter was not expressed in a text file but in blocks of imperative commands to make both conditions use a similar means of graphical manipulation instead of editing text. Our pilot tests confirmed that the same embedded programming task can be of varying difficulty depending on the programming paradigm used. In our case, FBP as used in Flowboard seemed to be easier for tasks with a physical electronic input, as users considered the program to follow the signal flow, starting at the input component and leading to an effect at the output component. Imperative programming seemed to be easier for tasks without an input component, such as a blinking LED that required providing a trigger such as a clock in software. We also observed that loops were a natural concept to code explicitly in imperative programming. The implicit continuous sense-react loop that drives an FBP runtime behind the scenes remains invisible to the user (as it probably should). Therefore, when developing environments using FBP to teach embedded coding, we recommend starting with tasks that include input and output components. This way, users can experience from the start what effects they can trigger with input components and their programming code, learning the useful embedded programming strategy of continuously reacting to changing input values early on. Our results suggest that the programming primitives that Flowboard provides were suitable for the tasks we set out. For example, the blackboxing of components helped students to fulfill the tasks (see Resnick and Silverman [43]). However, the selection and design of the nodes available in the language needs more careful review; nodes for more complex commands might lead to user mistakes. We observed that for a well-functioning FBP editor, the black boxes that imply and hide programming details need to be chosen even more carefully than in imperative programming (see also Resnick and Silverman [43]). We believe the reason for this difference is that the syntax in imperative programming is closer to natural language [30] than the graph structures in flow-based programming. We recommend investigating how flow-based nodes can be designed to be closer to natural language while keeping the language flow-based. While designing those, it is necessary to keep in mind how abstract the nodes should be, what the goal of the node is and what the user should learn from it. We believe it is more important for the user to be able to provoke action with their code than understanding what the code commands are provoking exactly on a compiler level.

As discussed in section 3, Flowboard was originally designed as a tool to help novices explore embedded programming, which could be considered an informal learning scenario. At the same time, according to the definition by Rogers [44], Flowboard is not a purely informal learning tool either, because it is not an everyday object that provides learning as a side effect of using it. We recommend integrating Flowboard into formal STEM education as a tool with which children can learn about the topic exploratively: The tasks children have to solve should be designed leaving room for their own ideas, encouraging creativity and exploration.

6.4 Collaboration

In our second user study, we observed participants beginning to collaborate when building their circuits. They helped each other with problems even before the instructor assisted them. During the first two guided tasks, participants still asked the instructor for help frequently. In the third task, six groups began to split the LEDs to create colors and to negotiate how to create different colors. The concept of RGB was not clear to 11 participants, yet they created colors by feeding sensor values into a color node in the Flowboard IDE. When participants wondered why the RGB stripe did not show a color change when they applied very high pressure on the pressure sensor,

they started to ask each other whether all connections in the program were correct. When they were still unsure, they turned to the instructor: "We cannot see any change. Why? Can you help us?". She then explained the mapping problem (RGB values ranging from 0 to 255, while sensor values range from 0 to 1023, leading to sensor values of 256 to 1023 being mapped to the same color). Participants then began to discuss how to change the program. Working together helped participants overcome many challenges compared to the first user study, which had only single participants. Participants mentioned that they felt more encouraged to first ask their partner before turning to the instructor. This resonates with the findings of Matias et al. [31] and Dasgupta and Hill [16], who found that sharing knowledge with others improves the learning curve. Since the first study did not take place in pairs, we cannot currently say whether spontaneous collaboration would also happen with ArduBlock. We hope to further explore this question in follow-up studies.

To conclude, we observed that the live, seamless, flow-based programming approach increased students' understanding of the interaction between embedded hardware and software components. Students used the liveness of the system to explore how different input values influence their design. The importance of liveness has been investigated in imperative programming systems such as by Krämer et al. [25] and Cabrera et al. [13], showing that liveness is essential for embedded programming. The drawings generated indicate that Flowboard adds another layer to students' mental model of what programming can look like, going beyond the traditional text-written programming metaphor. Especially for electronics, which display output signals as soon as power is turned on, developing a mental model of how single components work could be supported by a live, flow-based environment. Liveness helps users see the changes of their modifications directly. Flow-based programming supports users in developing a more clear mental model on how data signals flow through the program from input to output components. Flowboard supported pair programming in a satisfactory manner, indicating another potential advantage of FBP for learning embedded coding. Further investigations are needed to identify which factors, e.g., FBP or the physical design of Flowboard, are more advantageous for learning embedded programming compared to existing approaches. Since enabling students to learn and explore in hands-on sessions is of critical importance [7, 49], we believe that taking these characteristics into consideration when designing such tools can advance and enrich the learning process.

7 LIMITATIONS AND FUTURE WORK

Our current prototype faces a few technical challenges. As discussed earlier, using a serial protocol like Firmata on the Arduino limits embedded performance compared to native Arduino code, making some advanced Arduino applications impossible with our approach. A more significant limitation may be that Flowboard requires the iPad to remain connected to the Arduino board via Bluetooth to run programs. This is different from standard embedded programming practice, in which the embedded system, such as an Arduino board, can be disconnected from the computer after the code has been uploaded. Being able to upload an FBP code graph to the Arduino microcontroller and execute it there is possible, as MicroFlo[26] illustrates. Flowboard is designed as a learning tool and for simple projects. To help with transitioning from Flowboard to a standard environment like the Arduino IDE, we are exploring how to generate native Arduino code from a given program graph to reuse in the Arduino IDE. This would be a useful next step to allow Flowboard projects to run standalone and could also help to address Firmata's performance limitations.

As mentioned in section 6.1, the physical input pins currently "float" when not connected. While this is technically the correct behavior, the constantly changing value displayed next to some pins can become irritating. Detecting floating pins, similar to [18], could address this issue. Finally,

[26]http://microflo.org

the switch that currently disconnects only the power connections to both breadboards should interrupt all connections between the breadboards and the rest of the system to reduce the danger of damaging components while working on the electronics further. Participants also proposed that Flowboard's design can hinder advanced users due to size considerations for more complex tasks. We hope to integrate the user feedback and these technical improvements in our next hardware iterations.

We evaluated our approach only with young learners, not with adult beginners or more experienced users. Our user studies not only revealed further opportunities to improve the Flowboard user interface (such as preview animations explaining a node's function, our participants' biggest challenge), but we also noticed a fairly clear difference between age groups in understanding programming concepts, likely tied to their advancement in analytical thinking. We want to focus future studies on groups with a smaller age spread, e.g., one group for 10–13 and another for 14–15 years, to investigate the applicability of Flowboard for those different age groups. Larger studies with statistical power and more homogeneous user groups could help confirm and underline our quantitative observations, and help us understand the conceptual "ceiling" of the visual FBP approach in embedded development. Additionally, we would like to run studies to see if quantitative measurements can demonstrate significant advantages of Flowboard over, e.g., Ardublock regarding task performance and ease of use.

Our study 2 began investigating how collaboration helps to learn embedded programming using FBP. Comparing this to the effects of collaboration in imperative programming environments for embedded development would warrant further study. Furthermore, in the second user study, we did not evaluate any gender effects in the collaborative pair programming task. We hope to learn more about this through additional analysis of our existing study data, and in follow-up studies.

We initially set out to better understand the mental models of novices regarding embedded programming when using FBP, and discussed our results in section 6. Based on the tendencies we found in our second study, we hope to sharpen our understanding of how individual aspects of Flowboard and FBP in general impact the user's emerging mental model of embedded programming in a future study.

Transferring knowledge is a key concept when learning. In future studies, we would like to better understand what knowledge our Flowboard users are able to transfer over to, e.g., imperative programming, or designing electronic circuits. Knowledge transfer also depends heavily on the design of the programming language itself. At the moment, we are limited by the HCI research lens, but beyond we would like to look at didactic concepts such as those described by Resnick and Silverman [43] and see how they can be applied to FBP. Similarly, our recommendations for support tools in section 6.1 could be expanded by considering additional didactic aspects and methods in the design and study. We also did not test all the nodes of the Flowboard library in our studies. We plan to collect additional feedback to gather a more complete understanding of the value of different programming primitives in FBP for embedded coding. Finally, hybrid solutions that join FBP and imperative programming constructs may offer a way to bridge the gap between both paradigms that is worth exploring.

8 CONCLUSION

We have proposed switching from traditional imperative programming to the *flow-based programming paradigm* as an intriguing new way to tackle the steep learning curve of embedded programming. Our *Flowboard* prototype includes a touch-based visual FBP editor running on a large iPad, and a hardware design that integrates the iPad with a hidden Arduino microcontroller board and two breadboards with pin breakouts for input and output circuitry. Fitting the nature of FBP, Flowboard is a *live* environment, with both program and sensor value changes taking effect

immediately and a more *seamless* environment, linking physical I/O pins directly to their virtual counterparts in the IDE. The approach naturally supports coding independent parallel processes and pair programming.

Many tasks in early embedded programming use a signal-processing structure that continuously receives input from external hardware sensors and reacts to these, e.g., by driving hardware actuators, in a tight run-loop (Arduino's "loop()" program architecture reflects this). For such tasks, a live environment is much more beneficial, as liveness allows one to instantly see the effect of code changes without the need for the traditional edit-compile-upload loop of embedded programming. FBP can provide this liveness.

Our first study tested whether users can solve basic embedded programming tasks with Flowboard. To focus our analysis, we compared it to Ardublock, a Scratch-like, similarly visual, but imperative programming IDE. Our observations and user feedback showed that FBP is a promising approach to learn embedded coding.

Our second study explored different programming tasks and what impact Flowboard has on high school students learning embedded programming. We found that the seamlessness and liveness of Flowboard provide cues to develop a mental model of electronic components and how to program them. In addition, Flowboard encourages users to learn embedded programming through exploration and collaboration.

We hope that our findings inspire others to further explore the concepts of FBP, liveness, and seamlessness in order to further lower the bar for beginners and young learners to discover embedded programming.

ACKNOWLEDGMENTS

We would like to thank all our participants for taking part in the user study. Additionally, we would like to thank our external colleagues who helped us in setting up the experiment.

REFERENCES

[1] Alan Agresti and Brent A. Coull. 1998. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician* 52, 2 (1998), 119–126.

[2] Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2017. Trigger-Action-Circuits: Leveraging Generative Design to Enable Novices to Design and Build Circuitry. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. ACM, New York, NY, USA, 331–342. https://doi.org/10.1145/3126594.3126637

[3] Stelios Arakliotis, Dimitris G. Nikolos, and Emmanouil Kalligeros. 2016. LAWRIS: A rule-based Arduino programming system for young students. In *2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. 1–4. https://doi.org/10.1109/MOCAST.2016.7495150

[4] Fernando R. Avilés and Carlos A. Cruz. 2017. Mobile augmented reality on electric circuits. In *2017 Computing Conference (Computing Conference '17)*. 660–667. https://doi.org/10.1109/SAI.2017.8252166

[5] Rafael Ballagas, Faraz Memon, Rene Reiners, and Jan Borchers. 2007. iStuff Mobile: Rapidly Prototyping New Mobile Phone Interfaces for Ubiquitous Computing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. ACM, New York, NY, USA, 1107–1116. https://doi.org/10.1145/1240624.1240793

[6] Laura Barraza. 1999. Children's Drawings About the Environment. *Environmental Education Research - ENVIRON EDUC RES* 5 (02 1999), 49–66. https://doi.org/10.1080/1350462990050103

[7] Nadine Bergner and Ulrik Schroeder. 2015. Informatik Enlightened - Informatik (neu) beleuchtet dank Physical Computing mit Arduino. In *Informatik allgemeinbildend begreifen*, Jens Gallenbacher (Ed.). Gesellschaft für Informatik e.V., Bonn, Germany, 43–52.

[8] John Bird. 2017. *Electrical and electronic principles and technology*. Taylor & Francis.

[9] Tracey Booth and Simone Stumpf. 2013. End-User Experiences of Visual and Textual Programming Environments for Arduino. In *End-User Development*, Yvonne Dittrich, Margaret Burnett, Anders Mørch, and David Redmiles (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 25–39.

[10] Tracey Booth, Simone Stumpf, Jon Bird, and Sara Jones. 2016. Crossed Wires: Investigating the Problems of End-User Developers in a Physical Computing Task. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 3485–3497. https://doi.org/10.1145/2858036.2858533

[11] Anke Brocker, Simon Voelker, Tony Zelun Zhang, Mathis Müller, and Jan Borchers. 2019. Flowboard: A Visual Flow-Based Programming Environment for Embedded Coding. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI EA '19)*. Association for Computing Machinery, New York, NY, USA, 1–4. https://doi.org/10.1145/3290607.3313247

[12] Erik Brunvand. 2021. LED Paper: Physical Computing with Handmade Paper. In *ACM SIGGRAPH 2021 Educators Forum* (Virtual Event, USA) *(SIGGRAPH '21)*. Association for Computing Machinery, New York, NY, USA, Article 6, 2 pages. https://doi.org/10.1145/3450549.3464418

[13] Lautaro Cabrera, John H. Maloney, and David Weintrop. 2019. 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* (Boise, ID, USA) *(IDC '19)*. Association for Computing Machinery, New York, NY, USA, 227–236. https://doi.org/10.1145/3311927.3323138

[14] Konstantinos Chorianopoulos, Letizia Jaccheri, and Alexander S. Nossum. 2012. Creative and Open Software Engineering Practices and Tools in Maker Community Projects. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (Copenhagen, Denmark) *(EICS '12)*. ACM, New York, NY, USA, 333–334. https://doi.org/10.1145/2305484.2305545

[15] Sayamindu Dasgupta, William Hale, Andrés Monroy-Hernández, and Benjamin M. Hill. 2016. Remixing As a Pathway to Computational Thinking. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing* (San Francisco, California, USA) *(CSCW '16)*. ACM, New York, NY, USA, 1438–1449. https://doi.org/10.1145/2818048.2819984

[16] Sayamindu Dasgupta and Benjamin M. Hill. 2017. Scratch Community Blocks: Supporting Children As Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. ACM, New York, NY, USA, 3620–3631. https://doi.org/10.1145/3025453.3025847

[17] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. 2011. Reactive Imperative Programming with Dataflow Constraints. *SIGPLAN Not.* 46, 10 (oct 2011), 407–426. https://doi.org/10.1145/2076021.2048100

[18] Daniel Drew, Julie L. Newcomb, William McGrath, Filip Maksimovic, David Mellis, and Björn Hartmann. 2016. The Toastboard: Ubiquitous Instrumentation and Automated Checking of Breadboarded Circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. ACM, New York, NY, USA, 677–686. https://doi.org/10.1145/2984511.2984566

[19] Clifton Forlines, Daniel Wigdor, Chia Shen, and Ravin Balakrishnan. 2007. Direct-touch vs. Mouse Input for Tabletop Displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. ACM, New York, NY, USA, 647–656. https://doi.org/10.1145/1240624.1240726

[20] Saul Greenberg and Chester Fitchett. 2001. Phidgets: Easy Development of Physical Interfaces through Physical Widgets. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology* (Orlando, Florida) *(UIST '01)*. Association for Computing Machinery, New York, NY, USA, 209–218. https://doi.org/10.1145/502348.502388

[21] Linda J. Harrison, Leanne Clarke, and Judy A. Ungerer. 2007. Children's drawings provide a new perspective on teacher–child relationship quality and school adjustment. *Early Childhood Research Quarterly* 22, 1 (2007), 55–71. https://doi.org/10.1016/j.ecresq.2006.10.003

[22] Eva Hornecker, Paul Marshall, Nick S. Dalton, and Yvonne Rogers. 2008. Collaboration and Interference: Awareness with Mice or Touch Input. In *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work* (San Diego, CA, USA) *(CSCW '08)*. ACM, New York, NY, USA, 167–176. https://doi.org/10.1145/1460563.1460589

[23] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[24] André Knörig, Reto Wettach, and Jonathan Cohen. 2009. Fritzing: A Tool for Advancing Electronic Prototyping for Designers. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction* (Cambridge, United Kingdom) *(TEI '09)*. ACM, New York, NY, USA, 351–358. https://doi.org/10.1145/1517664.1517735

[25] Jan P. Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How live coding affects developers' coding behavior. In *VL/HCC '14: IEEE Symposium on Visual Languages and Human-Centric Computing*. 5–8. https://doi.org/10.1109/VLHCC.2014.6883013

[26] Richard Lin, Rohit Ramesh, Antonio Iannopollo, Alberto Sangiovanni Vincentelli, Prabal Dutta, Elad Alon, and Björn Hartmann. 2019. Beyond Schematic Capture: Meaningful Abstractions for Better Electronics Design Tools. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. ACM, New York, NY, USA, Article 283, 13 pages. https://doi.org/10.1145/3290605.3300513

[27] Jo-Yu Lo, Da-Yuan Huang, Tzu-Sheng Kuo, Chen-Kuo Sun, Jun Gong, Teddy Seyed, Xing-Dong Yang, and Bing-Yu Chen. 2019. AutoFritz: Autocomplete for Prototyping Virtual Breadboard Circuits. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. ACM, New York, NY, USA, Article 403, 13 pages. https://doi.org/10.1145/3290605.3300633

[28] Shareen Mahmud, Jessalyn Alvina, Parmit K. Chilana, Andrea Bunt, and Joanna McGrenere. 2020. Learning Through Exploration: How Children, Adults, and Older Adults Interact with a New Feature-Rich Application. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376414

[29] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (Nov. 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

[30] Dino Mandrioli and Matteo Pradella. 2015. Programming Languages Shouldn't Be "Too Natural". *SIGSOFT Softw. Eng. Notes* 40, 1 (feb 2015), 1–4. https://doi.org/10.1145/2693208.2693232

[31] J. Nathan Matias, Sayamindu Dasgupta, and Benjamin M. Hill. 2016. Skill Progression in Scratch Revisited. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. ACM, New York, NY, USA, 1486–1490. https://doi.org/10.1145/2858036.2858349

[32] Diane Mavers. 2003. Communicating Meanings through Image Composition, Spatial Arrangement and Links in Primary School Student Mind Maps.

[33] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. Bifröst: Visualizing and Checking Behavior of Embedded Systems Across Hardware and Software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. ACM, New York, NY, USA, 299–310. https://doi.org/10.1145/3126594.3126658

[34] Will McGrath, Jeremy Warner, Mitchell Karchemsky, Andrew Head, Daniel Drew, and Björn Hartmann. 2018. Wifröst: Bridging the Information Gap for Debugging of Networked Embedded Systems. In *Proceedings of the 31th Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. ACM, New York, NY, USA.

[35] Meredith R. Morris, Jarrod Lombardo, and Daniel Wigdor. 2010. WeSearch: Supporting Collaborative Search and Sensemaking on a Tabletop Display. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work* (Savannah, Georgia, USA) *(CSCW '10)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/1718918.1718987

[36] John Paul Morrison. 1994. *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, New York, NY, United States.

[37] Adon Moskal, Joy Gasson, and Dale Parsons. 2017. The 'Art' of Programming: Exploring Student Conceptions of Programming through the Use of Drawing Methodology. 39–46. https://doi.org/10.1145/3105726.3106170

[38] Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (July 2016), 44–52. https://doi.org/10.1109/MC.2016.200

[39] Dan O'Sullivan and Tom Igoe. 2004. *Physical Computing: Sensing and Controlling the Physical World with Computers*. Course Technology Press, Boston, MA, United States.

[40] Sofia Papavlasopoulou, Michail Giannakos, and Maria L. Jaccheri. 2017. Empirical studies on the Maker Movement, a promising approach to learning: A literature review. *Entertainment Computing* 18 (2017), 57 – 78. https://doi.org/10.1016/j.entcom.2016.09.002

[41] Mareen Przybylla and Ralf Romeike. 2013. Physical Computing im Informatikunterricht. In *INFOS 2013: Informatik erweitert Horizonte - 15. GI-Fachtagung Informatik und Schule*, Norbert Breier, Peer Stechert, and Thomas Wilke (Eds.). Gesellschaft für Informatik e.V., Bonn, Germany, 137–146.

[42] Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. 2017. Turning Coders into Makers: The Promise of Embedded Design Generation. In *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication* (Cambridge, Massachusetts) *(SCF '17)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3083157.3083159

[43] Mitchel Resnick and Brian Silverman. 2005. Some Reflections on Designing Construction Kits for Kids. In *Proceedings of the 2005 Conference on Interaction Design and Children* (Boulder, Colorado) *(IDC '05)*. Association for Computing Machinery, New York, NY, USA, 117–122. https://doi.org/10.1145/1109540.1109556

[44] Alan Rogers. 2014. The Classroom and the Everyday: The Importance of Informal Learning for Formal Learning 1. (01 2014).

[45] Jeff Sauro and James R. Lewis. 2005. Estimating completion rates from small samples using binomial confidence intervals: comparisons and recommendations. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 49. SAGE Publications Sage CA: Los Angeles, CA, 2100–2103.

[46] Neil Selwyn, Daniela Boraschi, and Suay Özkula. 2009. Drawing digital pictures: An investigation of primary pupils' representations of ICT and schools. *British Educational Research Journal - BR EDUC RES J* 35 (12 2009), 909–928.

https://doi.org/10.1080/01411920902834282

[47] Evan Strasnick, Maneesh Agrawala, and Sean Follmer. 2017. Scanalog: Interactive Design and Debugging of Analog Circuits with Programmable Hardware. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. ACM, New York, NY, USA, 321–330. https://doi.org/10.1145/3126594.3126618

[48] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming Misconceptions for School Students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. ACM, New York, NY, USA, 151–159. https://doi.org/10.1145/3230977.3230995

[49] Burkhard Vollmers. 1997. Learning by doing - Piagets konstruktivistische Lerntheorie und ihre Konsequenzen für die pädagogische Praxis. *International Review of Education* 43, 1 (1997), 73–85.

[50] Chiuan Wang, Hsuan-Ming Yeh, Bryan Wang, Te-Yen Wu, Hsin-Ruey Tsai, Rong-Hao Liang, Yi-Ping Hung, and Mike Y. Chen. 2016. CircuitStack: Supporting Rapid Prototyping and Evolution of Electronic Circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. ACM, New York, NY, USA, 687–695. https://doi.org/10.1145/2984511.2984527

[51] Jeremy Warner, Ben Lafreniere, George Fitzmaurice, and Tovi Grossman. 2018. ElectroTutor: Test-Driven Physical Computing Tutorials. In *Proceedings of the 31th Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. ACM, New York, NY, USA.

[52] Kirsten N. Whitley, Laura R. Novick, and Doug Fisher. 2006. Evidence in Favor of Visual Representation for the Dataflow Paradigm: An Experiment Testing LabVIEW's Comprehensibility. *Int. J. Hum.-Comput. Stud.* 64, 4 (April 2006), 281–303. https://doi.org/10.1016/j.ijhcs.2005.06.005

[53] Te-Yen Wu, Hao-Ping Shen, Yu-Chian Wu, Yu-An Chen, Pin-Sung Ku, Ming-Wei Hsu, Jun-You Liu, Yu-Chih Lin, and Mike Y. Chen. 2017. CurrentViz: Sensing and Visualizing Electric Current Flows of Breadboarded Circuits. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. ACM, New York, NY, USA, 343–349. https://doi.org/10.1145/3126594.3126646

[54] Te-Yen Wu, Bryan Wang, Jiun-Yu Lee, Hao-Ping Shen, Yu-Chian Wu, Yu-An Chen, Pin-Sung Ku, Ming-Wei Hsu, Yu-Chih Lin, and Mike Y. Chen. 2017. CircuitSense: Automatic Sensing of Physical Circuits and Generation of Virtual Circuits to Support Software Tools.. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. ACM, New York, NY, USA, 311–319. https://doi.org/10.1145/3126594.3126634