HeyTeddy: Conversational Test-Driven Development for Physical Computing

YOONJI KIM, KAIST, Republic of Korea
YOUNGKYUNG CHOI, KAIST, Republic of Korea
DAYE KANG, KAIST, Republic of Korea
MINKYEONG LEE, KAIST, Republic of Korea
TEK-JIN NAM, KAIST, Republic of Korea
ANDREA BIANCHI, KAIST, Republic of Korea

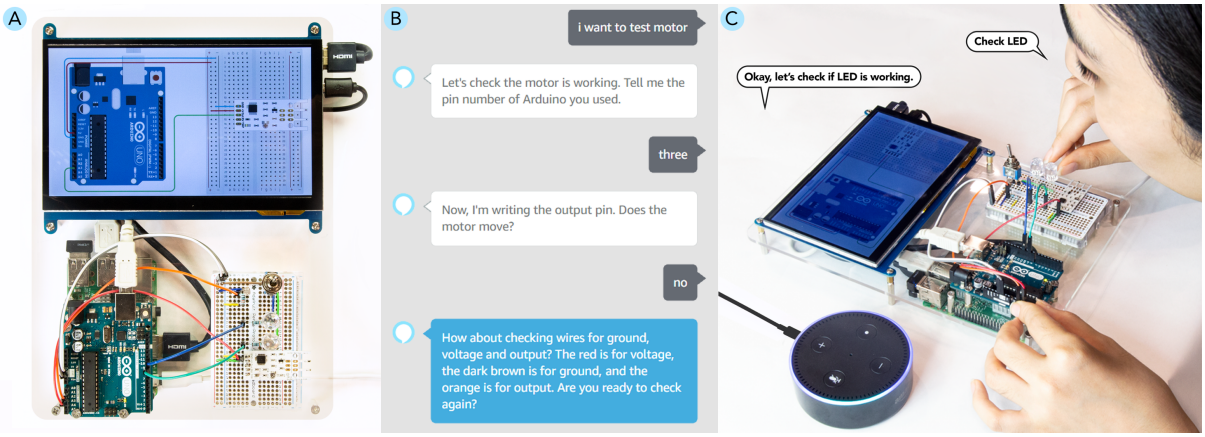

Fig. 1. (A) HeyTeddy overview and its core parts (a screen, Raspberry PI, Arduino, and a breadboard). (B) An example of conversation with HeyTeddy based on voice or text input (via Web chat). (C) A user's conversation with HeyTeddy via Amazon's Echo Dot about testing an LED.

Physical computing is a complex activity that consists of different but tightly coupled tasks: programming and assembling hardware for circuits. Prior work clearly shows that this coupling is the main source of mistakes that unfruitfully take a large portion of novices' debugging time. While past work presented systems that simplify prototyping or introduce novel debugging functionalities, these tools either limit what users can accomplish or are too complex for beginners. In this paper,

Authors' addresses: Yoonji Kim, Department of Industrial Design, KAIST, Daejeon, Republic of Korea, yoonji.kim@kaist.ac.kr; Youngkyung Choi, Department of Industrial Design, KAIST, Daejeon, Republic of Korea, youngkyung.choi@kaist.ac.kr; Daye Kang, Department of Industrial Design, KAIST, Daejeon, Republic of Korea, dayekang@kaist.ac.kr; Minkyeong Lee, Department of Industrial Design, KAIST, Daejeon, Republic of Korea, mkyeong.lee@kaist.ac.kr; Tek-Jin Nam, Department of Industrial Design, KAIST, Daejeon, Republic of Korea, tjnam@kaist.ac.kr; Andrea Bianchi, Department of Industrial Design, KAIST, Daejeon, Republic of Korea, andrea@kaist.ac.kr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
2474-9567/2019/12-ART139 $15.00
https://doi.org/10.1145/3369838

we propose a general-purpose prototyping tool based on conversation. *HeyTeddy* guides users during hardware assembly by providing additional information on requests or by interactively presenting the assembly steps to build a circuit. Furthermore, the user can program and execute code in real-time on their Arduino platform without having to write any code, but instead by using commands triggered by voice or text via chat. Finally, the system also presents a set of test capabilities for enhancing debugging with custom and proactive unit tests. We codesigned the system with 10 users over 6 months and tested it with realistic physical computing tasks. With the result of two user studies, we show that conversational programming is feasible and that voice is a suitable alternative for programming simple logic and encouraging exploration. We also demonstrate that conversational programming with unit tests is effective in reducing development time and overall debugging problems while increasing users' confidence. Finally, we highlight limitations and future avenues of research.

CCS Concepts: • **Human-centered computing → Interaction paradigms**; **Interactive systems and tools**.

Additional Key Words and Phrases: Physical computing, Test-driven development, End-user development, Conversational agent

ACM Reference Format:
Yoonji Kim, Youngkyung Choi, Daye Kang, Minkyeong Lee, Tek-Jin Nam, and Andrea Bianchi. 2019. HeyTeddy: Conversational Test-Driven Development for Physical Computing. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3, 4, Article 139 (December 2019), 21 pages. https://doi.org/10.1145/3369838

1 INTRODUCTION

Despite the growing popularity of physical computing in educational programs and in the maker community, prototyping interactive systems remains a challenging activity. Past research has demonstrated that the main difficulty novice users experience is the integration of two domain-specific tasks: assembling circuits by physically placing electronic components and wires (usually on a breadboard) and writing code for embedded systems such as the Arduino platform. The combination of these two activities makes the process challenging and often results in users being unable to debug problems [7]. Bugs as simple as identifying misplaced components, missing connections, and "crossed wires" [7] represent obstacles difficult to overcome for beginners because they cannot determine whether the source of the problem is software or hardware related.

Several hardware debugging tools have emerged in the form of modular electronic toolkits and testing devices that lower the barriers of circuit assembly and provide richer debugging information. LittleBits [4], Bloctopus [38], and Data Flow [8] simplify prototyping using standalone physical blocks that can be connected to program complex behaviors. However, while easy for novices to learn and use, these systems also limit what users can accomplish. Scanalog [41], CurrentViz [45], and Toastboard [10], on the other hand, provide users with detailed information about the electrical status of a live circuit, helping them to pinpoint the source of a bug. These debugging tools empower expert users but are not suitable for beginners.

This paper is, therefore, motivated by the need to create a general-purpose prototyping tool for software and hardware that does not inherently limit what users can accomplish but also simplifies prototyping and debugging for novices. Inspired by recent work on conversational agents for programming applied to agriculture [18], medicine [20], and statistics [12], we developed *HeyTeddy*, a conversational tool for novice users of physical computing. We hypothesize that conversational programming can reduce the distractions caused by dealing simultaneously with software and hardware [7] and that conversations can naturally lead to proactive debugging practices, such as test-driven development (TDD) [5] and *teddy bear debugging* [24]. Because these practices help programmers test individual units of code and explain their potentially flawed reasoning, we see an opportunity to integrate them in a physical computing prototyping task. Therefore, our system HeyTeddy (named for these practices) autonomously assists users with completing commands, suggests ways to test individual electronic components, and helps troubleshoot problems.

The contribution of this paper is twofold. We present HeyTeddy, the first general-purpose conversational programming environment for physical computing prototyping. HeyTeddy is the result of a participatory design session with ten users over six months. HeyTeddy is a system that supports several basic commands available in

Arduino as well as basic control-flow constructs. Our first user study highlights the strengths of conversational prototyping and compares voice and text input modalities. The second contribution is to show the advantages of conversational programming over traditional prototyping. With our second study, we demonstrate that conversational programming greatly increases the number of unit tests the users perform and consequently reduces obstacles, breakdowns, and bugs. Surprisingly, it also speeds up the overall development process by increasing information retrieving and users' confidence.

The rest of the paper is organized as such: After introducing related work, we present HeyTeddy, its capabilities, an example walk-through scenario and the technical implementation of the system. Two studies follow. The first study aims to compare the text and voice modalities, and the second study shows how TDD practices result in conversational and traditional prototyping performing differently. Finally, we present the limitations of this work and future avenues of research.

2 RELATED WORK

2.1 Tools for Beginners in Physical Computing

The following section is a summary of recent research achievements regarding prototyping toolkits for physical computing.

Several researchers have proposed systems that reduce or eliminate the need for programming. This is often achieved through a modular design, with physical objects encapsulating predefined behaviors that can be combined to produce programmable actions [4, 8, 38]. For example, LittleBits [4] are physical blocks with built-in input/output capabilities that can be snapped together to create complex prototypes. Data Flow [8] relies on wireless modules for sensors and actuators that enable event-driven and user-customized behaviors. Bloctopus [38] is another modular electronic prototyping toolkit and uses USB connectors and LEGO blocks as physical interfaces. All of these systems foster tinkering and support learning by allowing users to create programs by connecting modules instead of writing code.

Alternatively, several researchers have proposed systems for physical computing that reduce the complexity of writing programs (e.g., through visual programming) or have better real-time debugging features. The Learning-Arduino-With-Rules Introductory System (LAWRIS) [2] is a Web-based environment with Blockly[1] visual editor for Arduino programming. Splish [23] is a visual programming environment that translates connections between visual icons to code executed in real-time on the Arduino platform. The system Trigger-Action-Circuits [1] shares similar objectives and functionalities, but it also generates Fritzing[2] schematic diagrams and assembly instructions for building the circuit.

Other systems focus on improved real-time debugging features. Scanalog [41] visualizes the inner state of hardware components within a Field-Programmable Analog Array (FPAA) and translates the results into Fritzing schematics. VirtualComponent [25] allows for the interactive placement and modification of the value of electronic components in software, resulting in immediate changes to the physical components' values in the circuit. CurrentViz [45] uses a graphical interface to display the flow of current inside a circuit, and Toastboard [10] visualizes the voltage levels for every node of the circuit on the schematics and in the hardware.

The tools described in this section share with our system the common objective of simplifying programming and debugging by using real-time execution of codes through physical proxies. Our system borrows some of these features (e.g., simplified programming language, real-time execution of instructions, and visualization of the hardware's inner state) but primarily relies on conversation with an agent for programming and executing code.

[1] https://developers.google.com/blockly/
[2] http://fritzing.org

2.2 Conversational Interface for Data Exploration and Programming

Past researches on conversational interfaces for programming have highlighted the benefits of voice over typing for making data explorations and programming easier for beginners and more accessible to people with special needs [33, 42] or little programming experience [e.g., 18, 40]. Data exploration through voice interfaces can significantly lower the barrier for those users unfamiliar with programming and querying large datasets. For example, both FarmChat [18] and Jone et al.'s work [20] introduced conversational agents for farmers and biomedical researchers to interact with specific domain knowledge and massive databases. Other work targeted more general usage of conversational agents for data visualization [21, 40] by supporting custom and interactive queries. Finally, Iris [12] is an agent that performs open-ended data science explorations and, unlike other agents, combines commands using nested conversations.

Voice programming can also significantly lower programming barriers for nonexpert programmers [3, 34–37]. For example, Repenning et al. [34, 35] proposed a conversational agent system to assist novice programmers with proactive debugging capabilities that aim to reduce discrepancies between the intended and actual code. Voice has also been briefly introduced to physical computing for simple prototyping tasks. Jung et al. [22] presented a mockup prototype of a conversational agent that interacts with and guides a user in building a circuit and programming an Arduino. They do not focus on presenting a working system but rather demonstrate through a Wizard-of-Oz study how a voice companion fosters users' concentration and engagement when they are building a physical computing prototype.

Our work differs from the above by presenting the first working example of a voice-based programming system for physical computing, hence handling conversations to issue commands that are translated into code and executed in real-time.

2.3 Test-Driven Development for Software and Hardware

Test-Driven Development (TDD) is a software development process consisting of very short iterative cycles of code production followed by ad hoc tests to ensure code quality and that any new code addition executes without introducing new errors. Essentially, the programmer writes test cases meeting specific requirements, and any successive iteration of the software is validated by passing these tests [5]. TDD has been shown to improve overall software quality by reducing the complexity of the development and increasing the programmer's confidence, resulting in more reliable software and lower rework efforts [5, 11]. Specifically, Williams et al. [44] reported that software developed through TDD showed almost 40% fewer defects than code developed through the traditional development cycle, and Bhat et al. [6] presented similar findings through case studies that show significantly higher quality of code in projects developed using TDD.

Inspired by the advantages that test cases offer to software development, more recent efforts have been made to also apply TDD to hardware and physical computing prototypes. Specifically, we focus here on reviewing TDD and similar processes as debugging methods for traditional hardware design involving discreet physical components, as opposed to discussing programmable hardware test benches for Field-Programmable Gate Arrays (FPGAs) with hardware description languages [13, 28]. Recently, many researchers have been proposing methods to help users test circuits and physical computing prototypes. These attempts fall into two categories: systems that require users to write custom code to test hardware integrity and systems that adopt a tutorial-like approach to enforce users' self-assessment of the hardware.

The software d.tools [15, 16] assists users in designing, testing, and analyzing a physical prototype using both editable code and video logging. Users can add or edit Java code with the API (Application Programming Interface) system to test the state of a device with specific input and output. Scanalog [41] implements unit tests by replaying recorded input signals and allowing users to test them with assertions. Users can also implement custom checks by editing a skeleton JavaScript code from an example and interactively see reports of the results.

However, these systems require users to directly write the code for the test cases — a difficult task with high learning barriers for beginners. In fact, past research [7] demonstrated that, beyond the difficulty of learning to program in a specific programming language, even short code and simple circuits may contain interconnections with errors spanning software and hardware that are very difficult for beginners to identify. To reduce this complexity, Bifröst [29] supports tests with automatic code-checker and user-defined functions. In practice, the user can select an Arduino pin, an edge type (rise, fall, change), and a target line in the test code using a graphical interface. However, this solution is limited in scope (it only works for edge triggers) and still requires users to manually write code for the Arduino.

ElectroTutor [43] adopts a completely different strategy by introducing a tutorial with interactive tests. The user assembles a circuit and writes a program for Arduino following predefined steps. At each step, the user is prompted with an interactive test requiring manual verification and validation of the results before advancing to the next step. The merit of ElectroTutor is to simplify learning physical computing through the tutorial's steps, but it does not support free explorations and designs or custom test cases for different steps of the tutorial.

The work we present in this paper belongs to a third category, sharing similarities with the two approaches presented above. Similar to d.tools, Scanalog, and Bifröst, our system allows users to freely prototype beyond the constraints of a specific tutorial, but at the same time, as with ElectroTutor, it does not require users to write code in a specific programming language. Instead, we present a general-purpose conversational prototyping environment with a rich set of commands and customizable test cases.

3 SYSTEM

HeyTeddy is a conversational agent that allows users to program and execute code in real-time on an Arduino device without writing actual code but instead operating it through dialogue. This conversation can either be based on voice or text (through a Web chat). Commands spoken to HeyTeddy are parsed, interpreted, and executed in real-time, resulting in physical changes to the hardware. For example, the "write high" command configures an I/O pin to behave as a digital output with its internal state set to *high* (e.g., a 5V logic level), making driving an LED possible. Hence, the user does not need to write any code, compile it, deal with errors, and manually upload it on the hardware.

Furthermore, HeyTeddy supervises the user's choices, preventing incorrect logic (e.g., writing an analog value to a digital pin), guiding the user through each step needed to assemble the circuit, and providing an opportunity to test individual components through separate unit tests without interrupting the workflow (i.e., TDD functionalities). Finally, the user has the option of exporting the issued commands as a written code for Arduino (i.e., an Arduino sketch in C++, ready for upload).

The next subsections describe the codesign process utilized to develop the HeyTeddy prototype, a list of the system's capabilities, a walk-through example, and the technical implementation details.

3.1 Formative Study

HeyTeddy was informed by running a formative study over 6 months. We recruited from KAIST, Korea, 10 design students (7 male) aged 21–29 (M = 25.9, SD = 2.1) with basic physical computing knowledge gained through their coursework, such as some programming background (e.g., Arduino), prototyping with breadboards, and reading Fritzing pictorial schematics. Eight participants were graduate-level students, and two were undergraduate students. In terms of spoken language capabilities, two were native English speakers, and the others reported an average TOEIC score of 877.5 (SD: 62.5).

To facilitate the participatory design task, we developed an initial prototype of the conversational system and provided it to the users as a technological probe [17] for triggering reflections while completing a physical computing task [39]. The capabilities of this prototype were selected to offer the participants a realistic experience

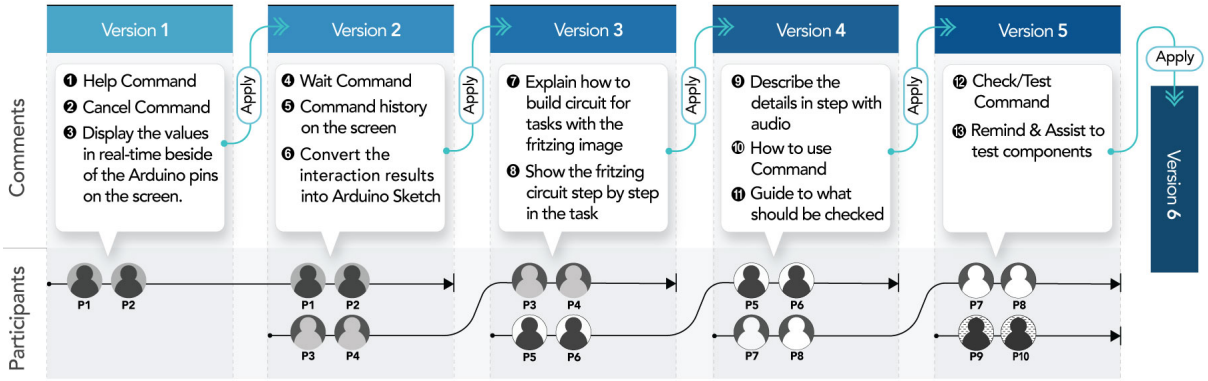

Fig. 2. Iterations of the codesign process with resulting features.

with a working device without limiting their ability to provide suggestions and comments [17]. With this initial system, each user was involved in two open-ended prototyping sessions in the lab, distributed over at least a week.

Each session lasted about two hours and consisted of an ice-breaking introduction with the moderator (10 minutes), a prototyping session with the system (50 minutes), and a post hoc interview (up to 1 hour). The exercises were based on three combined tutorials from the Arduino Projects Book included in the Starter Kit[3] (Spaceship Interface, Light Theremin, and Motorized Pinwheel). Participants were compensated with 50 USD in local currency per session. At the end of each session, we collected the participants' comments, extracted requirements and suggestions, and implemented them as features for the next version of the prototype.

This process resulted in six iterations (V1-V6), where V1 is the original design of the conversational system, and V2–V6 are the codesigned iterations. Specifically, for every iteration of the system (except the first), we collected the comments from four participants: two participants were new, and the other two had already experienced the previous version of the system (Figure 2). By having a mix of new and returning users for each iteration, we aimed to collect comments reflecting both a fresh perspective and a more critical account of the implemented changes from the previous version. In total, we had eighteen sessions.

From the formative study, we collected and analyzed 22h 52m of videos (13h 38m of interviews) and 9753 lines of log files from the system's commands history. We transcribed the interviews and analyzed them with an affinity diagram, and then further distilled the requirements and features to iteratively add to the final design (V6). Figure 2 shows in detail each feature and when it was added, while minor improvements and polishing are omitted.

3.2 System Capabilities and Language Details

The system capabilities were designed based on the users' feedback gathered through the participatory design process as explained.

Overall, HeyTeddy supports many Arduino commands and programming constructs. The user can set the electrical state of a specific pin (input/output) or reset its value. The user can *read* a value from analog and digital pins and *write* analog output (e.g., Pulse Width Modulation or PWM) and digital values on a pin following the Arduino pinout convention. The system allows users to *pulse* a pin at specific frequencies or for specific durations, or to virtually connect the output of a pin as input to another pin (*pass data* command). In terms

[3]https://store.arduino.cc/usa/arduino-starter-kit

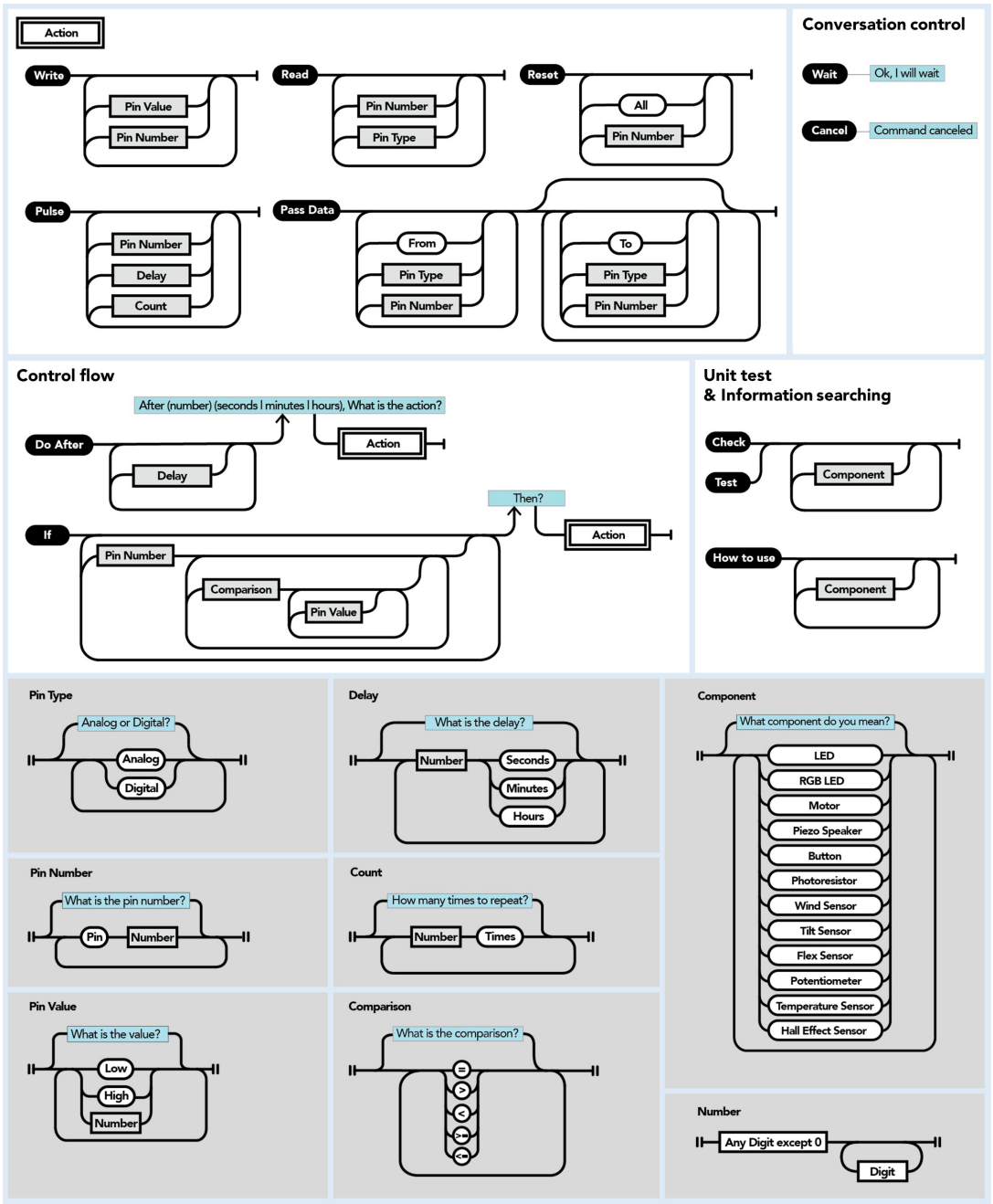

Fig. 3. The diagram describing the basic commands and language capabilities of HeyTeddy. Keywords (in black) are described as sequences of commands in white boxes and are grouped per type (actions, control flow, conversation control, unit test and information searching). Commands include tokens (in white circles) and references to simpler commands (defined in gray boxes).

of control flow, HeyTeddy supports multiple level branching using *if* statements that enable arbitrary (nested) branching instead of simple trigger-action mechanisms [1]. Finally, HeyTeddy supports scheduling an action after a specified interval of time with the *do-after* command.

The input commands are not required to follow a specific order as long as HeyTeddy can capture the user's intent (e.g., with keywords). The result is a rich set of phrases all mapped to execute the same commands, like a conversation with an agent, rather than a list of commands in a specific order. Users also do not need to remember all of the different options associated with a single command because HeyTeddy can ask for more details when presented with ambiguous or incomplete instructions. Moreover, the system is also capable of autonomously inferring from the context the correct options when no ambiguity arises.

For example, if a user issues the command "If pin 5 is *high*, then write pin 3," HeyTeddy will assume that pin 5 is a digital pin (because *high* is a digital state). However, the "write pin 3" part is ambiguous, and HeyTeddy will prompt "what value?" Depending on whether the user replies with a digital value (*high* or *low*) or a numeric value (a number between 0 and 255), HeyTeddy will perform a digital or analog (PWM) writing operation on pin 3. Following the same logic, the user who wants to read a value from the analog pin 1 can provide a completely unambiguous command ("read value from analog pin 1" or "read from analog pin 1") or a partial command ("read from pin 1" or "read"), and HeyTeddy will ask for clarifications ("analog or digital?" or "what is the pin number?"). A full description of the commands and the language associated with them is presented in Figure 3.

HeyTeddy also provides some visual feedback to help users remember the issued commands and navigate the interface. The graphical interface includes a history of the issued instructions, a help window with the possible voice commands, and a diagram of the Arduino pinout layout that shows with color-coding whether a pin is active or not, its electrical properties (input or output pin), and its state (low, high, or an analog value in the range 0–255) (Figure 4).

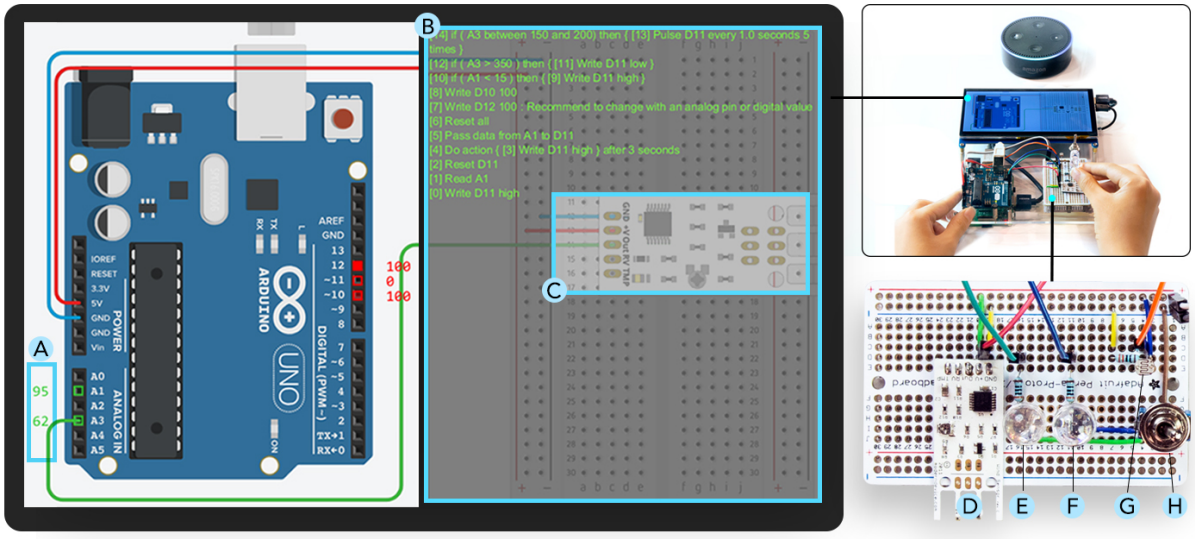

Fig. 4. HeyTeddy uses both conversation and a graphical interface to let the user know about the hardware inner state. (A) Numbers and overlays on the pins are used to indicate the pin mode (input/output) and state (high/low or numerical values). (B) A log console, and (C) a Fritzing schematic diagram are shown as well on the GUI, while the user can focus on assembling the hardware: anemometer (D), LEDs (E, F), photoresistor (G) and a switch (H).

HeyTeddy also includes features to support debugging and TDD. The user can test specific components without interrupting the programming task by issuing at any point the command *check* or *test*. HeyTeddy requires clarification of the component to be tested and its placement. It then attempts to read a digital or analog value in the case of input components (e.g., sensors) or to drive them with a digital or PWM signal for output components (e.g., LEDs or motors). After testing that the component works, the user can decide to move further or follow HeyTeddy's suggestions for troubleshooting.

Another feature is triggered by the *how to use* command, which allows users to ask for clarification about how to use specific components. HeyTeddy responds by showing a Fritzing diagram with an example of how to hook up the component. It also implicitly defines a context so that if the user tries to move on by asking how to use another component, HeyTeddy prompts a sequence of suggestions for testing the component in action. HeyTeddy currently supports the following 12 components of a different type (input/output, analog/digital), with different numbers of pins and interfaces: LED, RGB LED, hall effect sensor, tilt sensor, piezo speaker, flex sensor, anemometer, temperature sensor, servo motor, push-button, potentiometer, and photoresistor. This list can be customized by the user without the need for writing code but by using a graphical interface instead (Figure 5). With this interface, the user can create a new component by name, specify its behavior from checkboxes (input/output, digital/analog), customize dialogues, and upload an image with the wiring details.

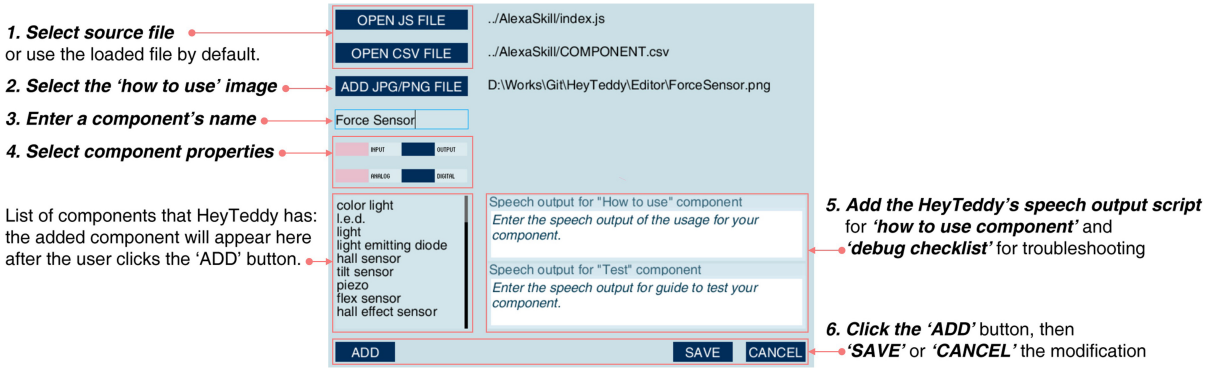

Fig. 5. The graphical software used to create and customize the behavior of additional components.

3.3 System Walk-through

We illustrate with an example how Alice, a novice user, converses with HeyTeddy to build the prototype of a lamp that turns on when it is dark and automatically adjusts the brightness depending on the ambient light. The lamp can be switched off by blowing it like a candle.

Conversational Programming: Real-Time Execution and Visualization. *Alice prepares the hardware components (an anemometer sensor, a toggle switch, and a few resistors, LEDs, and jumper wires) and starts up HeyTeddy with a vocal command. She wires a resistor and an LED to the solderless breadboard. After connecting the LED's anode to digital pin 11 of the Arduino board (D11), she says, "I want to* ***write high*** *to pin 11," and the LED turns on. She continues by wiring the photoresistor to analog pin 1 (A1) and instructs HeyTeddy to "Hmm... try to* ***read*** *from pin 1," to which HeyTeddy requests clarification of whether the pin is* ***analog*** *or* ***digital****. After specifying that the pin is analog, Alice uses her hand to partially cover the sensor and sees in real-time a numerical value on the screen next to the Arduino pin A1's graphical representation. Finally, Alice uses the readings from the sensor to adjust the brightness of the LED by simply saying, "****Pass data*** *from pin 1 to pin 11." Now Alice can control the LED's brightness*

by partially covering with her hand the photoresistor cell. Before moving on, Alice tests another LED with the ***pulse*** *command by saying, "****If*** *A1 is greater than 120, then pulse D10 every 200 milliseconds."*

The system supports twelve different commands (Figure 3), including reading and writing IO pins, passing data from an input to an output pin, pulsing, resetting, branching, and scheduling actions. All commands have an immediate effect, and the users can see the results in real-time, both directly on the hardware and on the graphical interface that shows the state of every pin. The voice recognition system allows users to speak naturally without having to remember precise sentences. HeyTeddy is capable of automatically inferring the meaning of a command even if the parameters are presented in the incorrect order and surrounded by emphatic interjections (e.g., "Hmm" or "Uh") or phrases ("Try to" or "I want"). In the case of an ambiguous or incomplete command, HeyTeddy asks the user to clarify it and to provide additional details.

Error Detection Instructions and Inferring from Context*. Later on, Alice wants to test analog writing (PWM) with an LED. After connecting the anode of the LED to pin 12 (which is a digital pin without analog output capabilities), Alice says, "I want to write the value 200." HeyTeddy prompts, "What is the pin number?" and she replies, "Pin 12." Understanding the mistake, HeyTeddy replies, "This is not possible. I wrote the* ***digital*** *value high on digital pin 12 instead." Alice also understands the error,* ***resets*** *the current state, and tries again with pin D10, which supports analog output.*

As with the regular Arduino board, a limited subset of pins is capable of analog input (A0 to A5) and analog output (D3, D5, D6, D9, D10, and D11, as indicated with a ~ in the GUI). While generally any pin can perform a digital read/write operation, only designated analog pins can perform analog output (PWM) or input operations. Specifically, output operations map a value between 0 and 255 to a 0–100% duty cycle, while input operations digitally convert a voltage in the range 0–5V to a number between 0 and 1023. HeyTeddy understands when the user accidentally attempts analog operations on digital pins and can alert or correct the user.

HeyTeddy prevents logical errors by notifying users of a possible misuse of pins such as using the numeric value on digital pins, the digital value on analog pins, or "pass data" between different types of pins. Moreover, the system is capable of autonomously inferring the correct options when no ambiguity arises. For example, if a user issues the command "If pin 5 is high, then write pin 10," HeyTeddy will assume that pin 5 is a digital pin (because *high* is a digital state). However, the "write pin 10" part is ambiguous, and HeyTeddy will request clarification. Depending on whether the user replies with a digital value (high or low) or a numeric value (a number between 0 and 255), HeyTeddy will perform a digital or an analog (PWM) writing operation on pin 10.

Assisting Users by Providing Information on Request & Test-Driven Development*. This is the first time that Alice uses an anemometer. Not knowing how to connect it, she asks,* ***"How do I use an anemometer?"*** *HeyTeddy displays on the screen a visual example of how to wire an anemometer using Fritzing schematics and then explains each step with voice.*

The last step of the process is to add a toggle switch for power. As in the previous step, Alice asks HeyTeddy how to use the switch. HeyTeddy infers that the task with the anemometer is complete and suggests she check whether it works: ***"Would you like to test the anemometer first?"*** *Alice replies "yes" and further tells HeyTeddy to which pin the anemometer was connected. Without adding any additional code, HeyTeddy allows Alice to blow in the anemometer and numerically visualize on the screen the sensed value. It also asks the user to verify whether the device is working or not. If it is not working, HeyTeddy suggests a list of possible troubleshooting actions. When Alice is satisfied with the result, she stops the test and moves to the next task.*

HeyTeddy allows users to gather information and test any components of the circuit. HeyTeddy contains a list of predefined parts and assembly instructions, but the user can define his or her own additional components

and specify assembly instructions, dialogues, modes of operation (input/output and analog/digital), and troubleshooting suggestions. These changes do not require users to write code but can be formed through a graphical interface that reads and updates the database of parts.

Tests can then be initiated directly by the user with the **test/check** commands or can be autonomously prompted by HeyTeddy based on the current context. For example, when a user asks for instructions on new components, HeyTeddy simply reminds the user to check her work so far. HeyTeddy then attempts to write or read a value on the pin of the component under test and asks the user to verify whether it works or not, without impacting the logic of the rest of the program.

Completing the Prototype. *The prototype is finally complete, and the programming logic is running on HeyTeddy. The lamp changes intensity depending on the ambient light, and Alice can switch it off by blowing into the anemometer. Now Alice can download the whole program running on HeyTeddy as a single Arduino code file (written in C/C++). After manually uploading it on the Arduino device, Alice can bring the lamp to her bedroom for usage.*

HeyTeddy provides the user with the option to convert the resulting interaction into code as an Arduino sketch file. The user can then either manually upload this code on any Arduino device or modify it by adding function blocks, copying and pasting parts of the code, and adding libraries if needed. This feature enables the user to reuse or refer to the auto-generated code as a template.

3.4 Technical Implementation

HeyTeddy consists of three independent modules running on different networked hardware, as illustrated in Figure 6. The first module deals with the voice input and conversations, and follows the rules shown in Figure 3. A second module maps the commands to the corresponding Arduino functionalities described in the Arduino reference page[4]. So, for example, *write* and *read* commands are implemented using the Arduino built-in *digitalWrite, digitalRead, analogWrite, analogRead* commands. The *pulse* action consists of repeated writing operations, while the *pass data* action concatenates a reading and a writing operation. Control flow commands and unit testing functionalities are scheduled in task queues, and executed as sequences of commands. A third module is used to link all the pieces together (i.e., it interfaces the voice to the hardware commands).

The voice input is handled by an Amazon Echo Dot device, which automatically converts the voice input to text and sends it to the cloud (i.e., Amazon Web Service's servers). This text provides the input for custom Amazon Alexa skill software that also runs on the cloud. The software was written using the Alexa Skill Kit[5] and consists of about 2500 lines of JavaScript code for parsing users' utterances and commands and for handling the flow of executing the instructions. In detail, the software is capable of handling 165 utterances (i.e., sentences) as a combination of different commands (53 slots) mapped to 13 different intents (the code that handles instructions for actions, conditions, replies from the system, unit testing capabilities, etc.). Finally, the Alexa Skill Software outputs a time-stamped field in JSON format that unambiguously describes the user's series of commands gathered with relative parameters during the conversation. In the case of a user preferring to type textual commands in a chat rather than using voice, this is also possible and does not require any change of code, as a Web chat client is integrated in Amazon's cloud.

The commands in JSON format are then sent in clear text format to an online HTTP server. A custom PHP script appends and saves each command to a JSON file on a public URL. This file is polled by the software running on HeyTeddy's hardware, and it is used to update the state of an Arduino board. Specifically, at every second, this software polls any new instruction from the server, compares the time stamp, and executes new entries by issuing a command to the user's Arduino board via serial using the Firmata library[6]. This software also takes care

[4]https://www.arduino.cc/reference/en/
[5]https://developer.amazon.com/alexa-skills-kit
[6]https://www.arduino.cc/en/Reference/Firmata

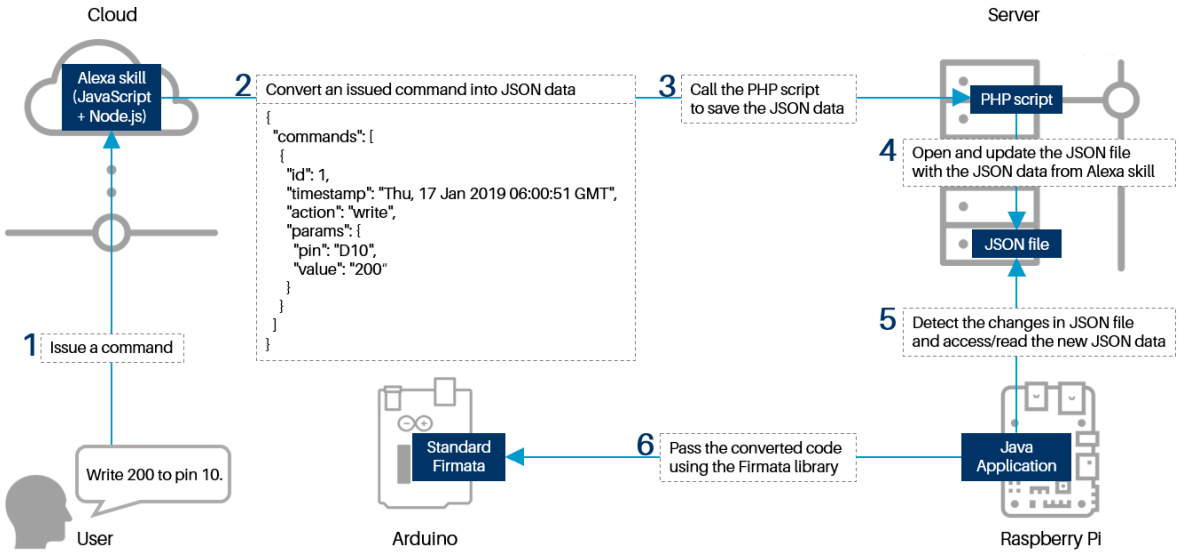

Fig. 6. Overview of the software process from the issued command to the execution of instructions on the Arduino.

of visualizing the state of the pins on the GUI as well as keeping track of the commands on a logging console. The software is written in Java and runs on a Raspberry PI 3 board. The Raspberry PI is interfaced to an Arduino UNO board. It is also connected to a 7-inch 1024x600 LCD touchscreen used for the GUI. These hardware components and a solderless breadboard are conveniently placed together on a supporting Plexiglas plate. The different software components of HeyTeddy can be found online at https://github.com/makinteractlab/heyteddy.

4 PRELIMINARY STUDY: INPUT MODALITY COMPARISON

We conducted an informal user study to compare users' modality preferences for conversations with HeyTeddy. The study compared voice and text inputs (via Web chat). The Web chat interface is included in the Amazon Developer Console as part of the Alexa Skills Kit and did not require additional coding, so HeyTeddy is guaranteed to run the same sets of commands and features across modalities. Commands issued through either voice or text result in an answer from HeyTeddy using both modalities (text-to-speech and written text).

Six students (2 graduate and 4 undergraduate), aged 21–29 (M = 23.5, SD = 3.3), with a variety of educational backgrounds (design, computer science, and mechanical engineering) and some self-reported prototyping experience with Arduino (M = 6.5, SD = 2.4) were recruited from KAIST, Korea. All students were non-native English speakers.

Users were introduced to HeyTeddy and could freely test the system with both input modalities (15 minutes). They then conducted a prototyping task with no time constraint and were subsequently interviewed about their experience using voice and text (30 minutes). The prototyping task consisted of assembling and programming one circuit randomly assigned from those in Figure 7. All exercises were designed to have similar complexity and were based on the tutorials from the Arduino Projects Book. The prototyping instructions were not delivered as a circuit diagram but similarly to how they are presented in Figure 7. The study lasted about 1.5 hours, and users were compensated with 20 USD in local currency.

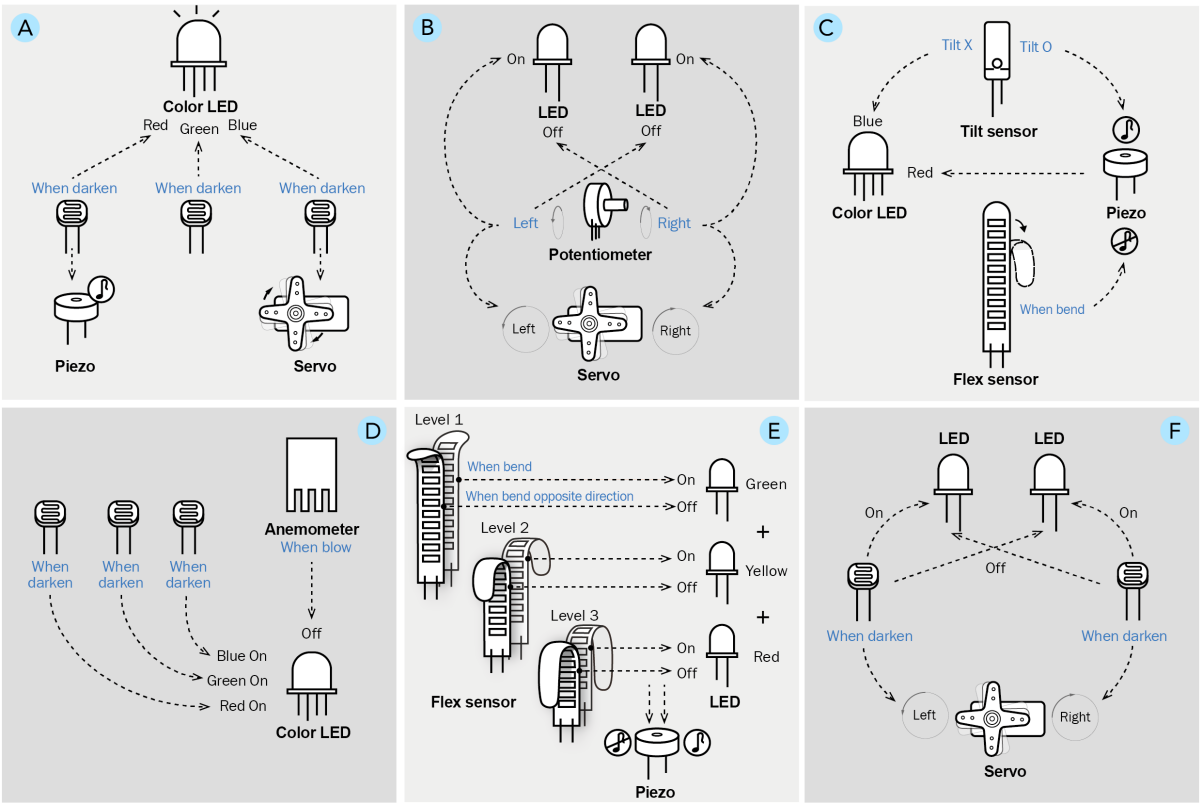

Fig. 7. Diagrams for the six prototypes used in the modality study. Users were given similar tasks consisting of controlling actuators (motors, LEDs, piezo) given some input from sensors (photoresistor, flex sensor, anemometer).

4.1 Results

All participants successfully completed the task between 40m 46s and 56m 57s (M = 49m 42s, SD = 4m 9s). All interviews resulted in 2h 31m of material, transcribed and analyzed with open and axial coding methods. The following themes emerged from the study.

Overall, participants felt positively engaged with conversational programming using both modalities. With the exception of P1 and P3, who exclusively used text and voice, respectively, all other participants seamlessly combined the two modalities. We observed several usage patterns. Most users started with voice but then switched to text when HeyTeddy could not parse what was said. However, they also switched back to voice input immediately after the command was registered by the system. For complex commands that required multiple expressions or specific parameters (e.g., the "if" statement), users preferred typing, but for simple commands (i.e., those followed by a single expression), voice was the preferred choice. Surprisingly, voice was also the preferred choice for the "test" command, which instead requires users to specify multiple options. P4 explains that *"If I say 'test', then it [HeyTeddy] keeps asking questions for completing the test of the component. It was nice to make a complex logic by exchanging conversation."* This suggests that the choice between voice or text does not depend merely on the complexity of the command, but also on how the expressions are logically structured and related to each other. For instance, users did not report any issue specifying one option at a time for the "test" command

because they felt guided in the debugging process. However, for the "if" command, users felt annoyed by having to specify related expressions in subsequent commands (the "if" block) instead of declaring them all at once in a single statement. For this latter case, text input felt more natural.

Although users reported that input could be more efficient (P1: *"I type faster than voice input"*) and less error-prone (especially for non-native speakers), most users indicated a preference for voice due to the benefits of having a conversation with the system. For example, both P3 and P4 reported how the conversation with its inherent pauses helped them organize their thoughts and elaborate on new ideas.

"By having a conversation with HeyTeddy, I felt that my thoughts became organized, and that complex ideas become words." (P3)
"HeyTeddy, let me do a fast and rough sketch for organizing the overall hardware behaviors using voice without coding."(P4)

Conversations were also a source of ideas (P3: *"I suddenly had a different idea when I talked to HeyTeddy, so I plugged in a tilt sensor instead of using a flex sensor"*) and a way to discover mistakes early in the process (*"HeyTeddy asked questions regarding things I did not recognize, mistakes, or missing information"* – P6). Furthermore, from a psychological point of view, the conversation provided reassurance about the task, as it felt like *"working with someone who knows a lot about the hardware"* (P3). Similarly, psychological reasons motivated P1 not to use voice, because *"[it] felt like I was giving orders to it [HeyTeddy] rather than having a conversation."* These motivations are interesting and demonstrate both the users' mental models and their perceptions of the system's agency.

Finally, some users suggested that voice conversation for programming has the advantage of freeing the user's hands, making prototyping hardware more efficient:

"I was able to keep my eyes focused on the circuit while listening to what HeyTeddy said." (P3)
"Using voice is great because you can make circuits without moving the hands to the keyboard or mouse."(P4)
"My eyes and hands are busy building the circuit, but I can still hear at the same time the information without [using a graphical] interference." (P2)

A complementary argument describes how voice can naturally overcome distances and hence be suitable for prototyping large or remotely controllable objects, like drones and RC cars (P6: "It would be very nice if I can say a command to program a drone in real-time and see the result while it is flying").

In conclusion, users felt comfortable using both modalities but tuned their choice depending on the command used. Voice was preferred overall as it more naturally led to conversations that helped users rationalize their processes, generating new ideas, and reflecting on the code structure. Finally, voice offered the opportunity to free users' hands from typing, hence allowing them to completely focus on the hardware. Nevertheless, text was appreciated for its efficiency and was always used as a backup option when voice input failed.

5 FEASIBILITY STUDY: CONVERSATIONAL VERSUS TRADITIONAL PROGRAMMING

The aim of this study was to understand the differences between traditional versus conversational physical computing prototyping. Both modalities were compared with users assembling a circuit in hardware, but while for traditional programming, users typed in software on a computer using the Arduino IDE and manually uploaded the code on the Arduino micro-controller, for conversational programming, they instead spoke to HeyTeddy.

As in previous work [7], we are interested in studying the practices and pitfalls that novice users experience with physical computing. Thus, we designed a study to reproduce the one by Booth et al. [7], aiming to compare differences between the two input modalities (HeyTeddy versus Arduino) for obstacles, breakdowns, and bugs during different tasks. For consistency, problems were defined as in previous work: **Obstacles** happen when "participants hit hurdles to overcome" and need to gather more information, **breakdowns** are "evidence of errors in action or thinking," and **bugs** are "evidence of faults" introduced by participants through their actions.

Therefore, in our study, events were coded as *problems* (obstacles, breakdowns, or bugs) [7] happening in either software or hardware, and *prototyping tasks* (programming the software, building the circuit, and searching for information). Unit tests conducted during the prototyping process were coded separately as hardware-based tests (e.g., the user tests a physical component) or software tests (e.g., the user tests the logic of the program or code functionalities).

Our study followed a between-subject design with two balanced conditions: HeyTeddy and Arduino IDE. We recruited 22 students: 12 male, aged 20–32 (M = 23.8, SD = 3.45), with some having experience with Arduino. They were then randomly assigned to one of the two modalities, forming two groups of 11 users. All participants received 10 USD in local currency as compensation.

The protocol of our study was the same as that in Booth et al.'s work. After a brief introduction, participants completed a demographics and a self-efficacy [9] form, in which participants evaluate their confidence with physical computing and Arduino (15 minutes). Users then did a prototyping session with one of the two modalities (up to 45 minutes). The goal of this section was to complete the "Love-O-Meter" tutorial from the Arduino Projects Book[7]. For practical reasons, the tutorial was modified to use a photoresistor instead of a temperature sensor, but the circuit and programming were the same. Participants could use any resource from the Internet (e.g., tutorials, component information, etc.) and were asked to think aloud during the session. We recorded videos of the participants from two different perspectives as well as the screen used for information gathering. We finally also collected users' perspectives in a post hoc interview (10-15 minutes).

5.1 Quantitative Results

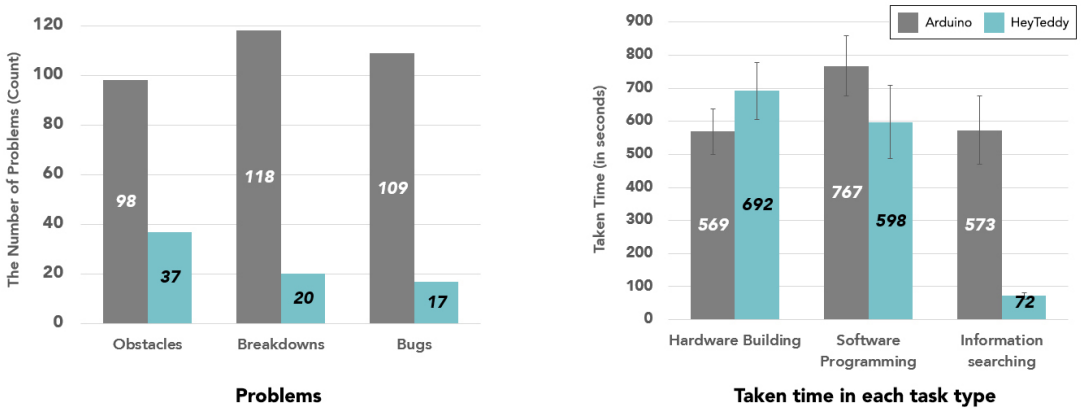

Fig. 8. Performance of HeyTeddy vs Arduino for number of problems (left), and task times (right).

Participants self-reported efficacy with physical computing is 67.72 (SD = 17.4) for the Arduino group and 62.8 (SD = 12.76) for HeyTeddy. An independent-samples t-test revealed no differences.

The videos of the recordings with the different views were combined into a single split-screen video and separately coded by two researchers following the protocol described above. We report an inter-rater agreement coefficient of K = 0.801. Raw results were then analyzed using independent-samples t-tests and three-way ANOVA tests followed by Bonferroni correction post hoc analysis with α=0.05.

[7]https://store.arduino.cc/usa/arduino-starter-kit

Overall, the success rate for completing circuits was 100% for HeyTeddy and 81% for Arduino. The analysis of variance revealed a statistical effect for problems (obstacles, breakdowns, bugs) across modalities (Arduino, HeyTeddy) ($F_{(1,60)} = 41.3, p < 0.01$), and pairwise post hoc comparisons showed that HeyTeddy outperformed Arduino with fewer problems in all three categories (p < 0.01). A histogram of problems (Figure 8) further corroborates these results.

HeyTeddy was faster overall than Arduino with an average completion time of 22m 42s versus 31m 49s ($t_{(17.6)} = 2.15, p < 0.05$). A deeper analysis also shows how users spent their time across the three tasks: programming, creating a circuit, and gathering information. We report statistical differences for time across modality ($F_{(1,60)} = 7.5, p < 0.01$), task type,($F_{(2,60)} = 11.5, p < 0.01$), and their intersection ($F_{(2,60)} = 7.4, p < 0.01$). A post hoc test further shows that information gathering was significantly faster when using HeyTeddy (p < 0.01).

HeyTeddy's users spent 16.5% of the overall prototyping time (on average 3m 45s, SD = 2m 6s) using unit tests (in software and hardware), but Arduino's users only spent 5.7% (M = 1m 49s, SD = 2m 11s). We found statistical differences for the duration of the tests across modalities (Arduino, HeyTeddy) ($F_{(1,60)} = 11.4, p < 0.01$), across task types (hardware building, software programming, information searching) ($F_{(2,60)} = 18.7, p < 0.05$), and their interaction ($F_{(2,60)} = 28.7, p < 0.01$). Post hoc analysis shows that in the case of Arduino, information searching took longer than the other tasks (p < 0.05). Furthermore, HeyTeddy's users spent more time on hardware tests than Arduino users (p < 0.05) but less time in information searching (p < 0.05). The number of tests, however, differed only for task type ($F_{(2,60)} = 4.3, p < 0.05$) and not modalities. Their interaction was, however, significant ($F_{(2,60)} = 8.5, p < 0.05$). Pairwise comparisons show that hardware tests were fewer than software tests (p < 0.05) — on average, 5.9 (SD: 6.1) tests on Arduino versus 12.1 (SD: 7.9).

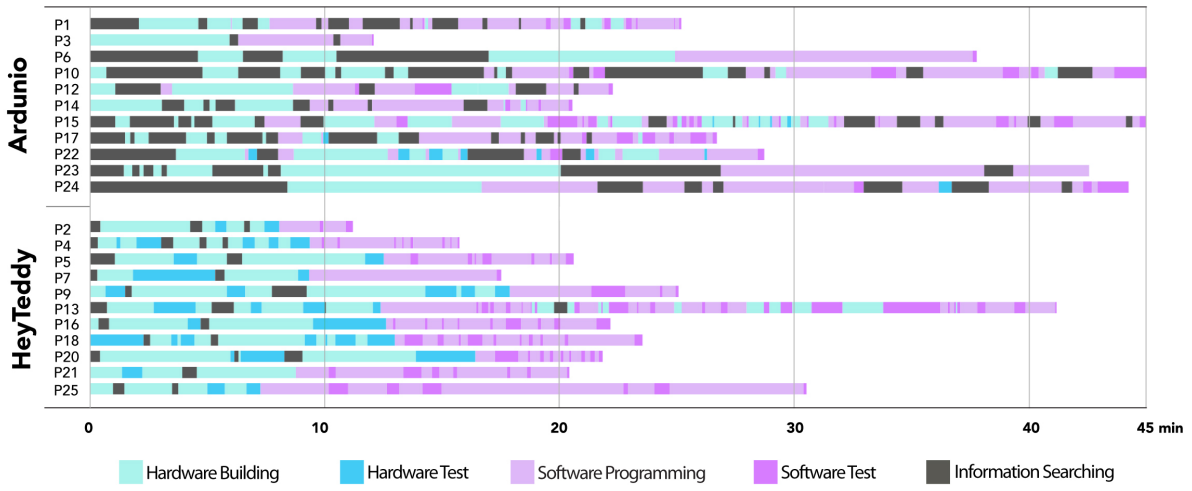

Fig. 9. Graphs showing how users spent their time during the prototyping process.

To sum up, our analysis shows that HeyTeddy was 26.1% faster than Arduino, but at the same time, that led to a higher completion rate and fewer obstacles, breakdowns, and bugs. Furthermore, despite users being faster with HeyTeddy and spending less time on information searching, they spent a considerably larger amount of time (+10.8%) testing their progress using the *test* command. However, the actual number of tests did not differ across modalities, only across the type of task (hardware, software, or information searching). These results are further corroborated by Figure 9, showing the time-stamped events for each participant in both conditions.

5.2 Qualitative Results

Overall, all participants who tried HeyTeddy expressed preference for conversational programming over traditional development with the Arduino and reported feeling generally more confident ("It was like having a friend or a teacher helping me" – P14). Users explained that HeyTeddy helped them quickly find information ("[with Arduino] I needed [...] to find exactly what I wanted, but with HeyTeddy, I can simply ask" – P17) and guided them through the process ("I had no confidence about how to build the circuit before. However, because of HeyTeddy, I didn't worry about wiring the circuit" – P7). Most users specifically discussed the testing of features and explained they were useful and easy to use because they did not require writing code.

"I used to worry about mistakes in wiring before, but I could confidently build a circuit with HeyTeddy because it alerted me whenever there was a possible mistake. It was easy to do tests such as 'check LED' without writing any software." (P5)

"I was concerned about writing code with Arduino IDE before, but I have easily finished the task with HeyTeddy because I don't need to write the code following a specific syntax." (P14)

These results are interesting when compared to what Arduino users reported. P9, one of the two participants who failed at completing the task, reported that s/he "could not find the reason for the problem" (P9). Even participants who eventually were successful reported similar experiences. For example, P20 explained, "I didn't test the circuit while building it, and it was not working after I uploaded the software to Arduino... so I was not sure what the problem was" (P20). Other participants further explained that the main sources of struggle were related to information gathering, coding syntax, and connecting components in the circuit.

"I struggled with finding the right circuit I needed on the Internet, and it took time for that." (P6)

"I hardly remember how those components should be connected, so I needed to search how to use them [...] even for the software, I was not sure how to combine [logical] conditions in C/C++ syntax." (P1)

"I searched for the code on the Internet and combined here and there, so I was not convinced the software would work." (P21)

6 DISCUSSION

The study results show that conversational programming with HeyTeddy led to faster prototyping and fewer obstacles, breakdowns, and bugs. Overall, participants appreciated having the possibility to ask HeyTeddy for guidance or for more information. They also regarded positively the interactive *test* command and the possibility to systematically troubleshoot problems using conversation. Overall, participants stated feeling more confident, and their performance with HeyTeddy reflected this self-assessment.

One of the difficulties that many end-users face in physical computing is learning a programming language [27]. Conversational programming offers the possibility of creating a custom program without writing code. Hence, users do not need to recall exact commands or the syntax of a specific programming language [12]. Our participants reported that a large portion of the time used information gathering was indeed spent searching for these details. Similarly, users can add unit tests for custom components [e.g., 29, 41]) without writing any software [e.g., 43]), simply through a graphical interface. In sum, HeyTeddy offers a simple yet rich set of commands from the Arduino platform, combined with unit tests for custom components. These two features allow for a general-purpose and open programming environment in which users are free to explore alternatives rather than being limited to tutorials [43] or limited components and behaviors [4, 8, 38].

Another important capability of HeyTeddy is that it handles ambiguous requests by either inferring information from the context or by asking the user to clarify the intent. In this way, HeyTeddy not only frees users from having to remember all of the necessary information to supply the program, but also prevents incongruities in the code. For example, writing or reading from a digital/analog pin often requires users to initially configure a specific I/O pin to specific states (*pinMode* operation). This situation can create unexpected errors that are difficult

to detect if the state of the pin is not compatible with the operation requested, because the compiler does not usually issue any warning in this situation. However, HeyTeddy alerts the user of a mismatch or even internally changes the state of a pin to be compatible with the issued command. By assisting users with completing the instructions through conversations, HeyTeddy effectively lowers learning barriers [26] for physical computing and prevents insidious bugs.

Our results also show that a conversational agent can engage users to test circuits before adding new components. Our participants, including those who reported that they did not test circuits until the whole hardware was assembled, performed interactive tests spontaneously and throughout the whole process. While many users in the Arduino condition built the whole hardware and then wrote code, HeyTeddy's users proceed with both aspects of the prototype in parallel. They are hence able to isolate and identify problems and errors while testing each component without carrying them over to the next step. While some of the Arduino users reported that they did not even know where the source of the error was, our data clearly show that all of HeyTeddy's users identified and solved their problems.

Finally, an interesting byproduct of using voice as the input modality was that users felt more engaged and reported their experience to be "fun." Interestingly, several users regarded the conversational agent as an intelligent and autonomous partner, and we believe that this aspect helped users better rationalize and plan their actions. It is, in fact, known that conversation activates all parts of the brain related to social interaction [30–32], regardless of whether the conversation is with another human being or with inanimate objects. Voice, however, was also the reason many users developed unrealistic expectations for the system, assuming that HeyTeddy had some semblance of artificial intelligence and autonomous decision-making capabilities. These aspects are interesting but beyond the scope of this work.

7 LIMITATIONS AND FUTURE WORK

This paper illustrates how the conversational agent HeyTeddy can simplify the cognitive effort of physical computing prototyping and provide assistance to debugging through unit tests. The limitations of HeyTeddy relates to the complexity of the prototypes that can be constructed using the system, the level of customization of the workflow, and the technical extendibility of the hardware and software features.

Issues related to scale are a primary concern. Conceptually, no ceiling exists regarding the number and type of components supported by HeyTeddy or the number of instructions that can be executed. The current system supports 12 components out-of-the-box, but this list is customizable by users. Future work will, however, focus on offering a greater spectrum of predefined components, updateable through an open API. There is also no upper limit to the complexity that can be achieved in the prototypes produced with the system. Through our studies, we report successful implementations of prototypes with 6 components, 20 wire connections, and 49 commands (exported by the system to 178 lines of C code). These numbers are compatible with the average project size presented in the Arduino Projects Book in terms of components (M = 6, SD = 3.64), wires (M = 9.3, SD = 4.7), and lines of code (M = 29.8, SD = 18.5). Although the current system is capable of supporting novice users' prototyping activities, we believe that the complexity of programs built through a conversation agent might be eventually bound by the cognitive load required to remember previously issued actions. Although complex circuits might also be harder to debug with HeyTeddy, we believe that conversation has the potential to help users decompose complex debugging tasks in simpler units that can be handled separately, as shown in our studies. Future work should present numerical evidence to characterize the effects of memory and cognitive load in conversational programming.

In terms of user customization and personalization, the system currently supports only one user at a time. We do not have an immediate plan for supporting multiple users, but it is worth noting that current research includes trends to make voice input possible from multiple users [19]. Furthermore, HeyTeddy currently provides

the same level of assistance regardless of the user's expertise. In the future, we are planning to introduce features that will allow users to specify their skill set or that will implicitly measure the user's level of expertise [14].

As for extensions of the system, future iterations will include more commands from the Arduino language as well as loop constructs and block-level scoping. To achieve this, exploring the structure of conversation and how it would impact the composition of multiple commands will be important [12]. Providing support for a wider range of micro-controller platforms beyond the Arduino UNO board is also desirable. Finally, the system currently lacks an "edit" feature that would allow users to modify previously issued commands. At this stage, the user has to undo what s/he did by assigning a new command to the same pin, which overrides older commands. Alternatively the user can reset the current program and restart. Future work will focus on making modifications possible.

8 CONCLUSION

We presented HeyTeddy, a general-purpose conversational programming system capable of assisting makers in building a physical computing prototype. The main advantage of this system compared to traditional prototyping is that it is based on real-time execution of vocal commands on the hardware, making prototyping simpler for novice users. Through two user studies, we collected evidence showing that voice input is preferred over text for programs with simple logic. Voice also encourages explorations and reduces the cognitive load needed to focus on two different tasks (programming on a computer and assembling hardware on a breadboard). The second study compared the usage of unit tests as a proactive debugging tool and showed that conversation implicitly fosters TDD practices, resulting in faster development time and lower errors. Future work will focus on improving the system by supporting more complex commands and programming constructs and will further demonstrate the practical benefits of using a conversational agent for physical computing prototyping.

ACKNOWLEDGMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2017R1D1A1B03035261).

REFERENCES

[1] 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 (UIST '17)*. ACM, New York, NY, USA, 331–342. https://doi.org/10.1145/3126594.3126637

[2] S Arakliotis, DG Nikolos, and E Kalligeros. 2016. LAWRIS: A rule-based arduino programming system for young students. In *2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 1–4. https://doi.org/10.1109/MOCAST.2016.7495150

[3] Barbara Rita Barricelli, Fabio Cassano, Daniela Fogli, and Antonio Piccinno. 2019. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *Journal of Systems and Software* 149 (2019), 101–137.

[4] Ayah Bdeir. 2009. Electronics As Material: LittleBits. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction (TEI '09)*. ACM, New York, NY, USA, 397–400. https://doi.org/10.1145/1517664.1517743

[5] Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.

[6] Thirumalesh Bhat and Nachiappan Nagappan. 2006. Evaluating the Efficacy of Test-driven Development: Industrial Case Studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)*. ACM, New York, NY, USA, 356–363. https://doi.org/10.1145/1159733.1159787

[7] 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 (CHI '16)*. ACM, New York, NY, USA, 3485–3497. https://doi.org/10.1145/2858036.2858533

[8] Alvaro Cassinelli and Daniel Saakes. 2017. Data Flow, Spatial Physical Computing. In *Proceedings of the Eleventh International Conference on Tangible, Embedded, and Embodied Interaction (TEI '17)*. ACM, New York, NY, USA, 253–259. https://doi.org/10.1145/3024969.3024978

[9] Deborah R Compeau and Christopher A Higgins. 1995. Computer self-efficacy: Development of a measure and initial test. *MIS quarterly* 19, 2 (1995), 189–211. http://www.jstor.org/stable/249688

[10] 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 (UIST '16)*. ACM, New York, NY, USA, 677–686. https://doi.org/10.1145/2984511.2984566

[11] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. 2005. On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering* 31, 3 (2005), 226–237.

[12] Ethan Fast, Binbin Chen, Julia Mendelsohn, Jonathan Bassen, and Michael S. Bernstein. 2018. Iris: A Conversational Agent for Complex Tasks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 473, 12 pages. https://doi.org/10.1145/3173574.3174047

[13] David Galloway. 1995. The transmogrifier C hardware description language and compiler for FPGAs. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE, 136–144.

[14] Jun Gong, Fraser Anderson, George Fitzmaurice, and Tovi Grossman. 2019. Instrumenting and Analyzing Fabrication Activities, Users, and Expertise. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 324, 14 pages. https://doi.org/10.1145/3290605.3300554

[15] Björn Hartmann, Scott R. Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. 2006. Reflective Physical Prototyping Through Integrated Design, Test, and Analysis. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*. ACM, New York, NY, USA, 299–308. https://doi.org/10.1145/1166253.1166300

[16] Björn Hartmann, Scott R Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee. 2006. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*. ACM, 299–308.

[17] Hilary Hutchinson, Wendy Mackay, Bo Westerlund, Benjamin B. Bederson, Allison Druin, Catherine Plaisant, Michel Beaudouin-Lafon, Stéphane Conversy, Helen Evans, Heiko Hansen, Nicolas Roussel, and Björn Eiderbäck. 2003. Technology Probes: Inspiring Design for and with Families. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '03)*. ACM, New York, NY, USA, 17–24. https://doi.org/10.1145/642611.642616

[18] Mohit Jain, Pratyush Kumar, Ishita Bhansali, Q. Vera Liao, Khai Truong, and Shwetak Patel. 2018. FarmChat: A Conversational Agent to Answer Farmer Queries. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2, 4, Article 170 (Dec. 2018), 22 pages. https://doi.org/10.1145/3287048

[19] William Jang, Adil Chhabra, and Aarathi Prasad. 2017. Enabling Multi-user Controls in Smart Home Devices. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy (IoTS&P '17)*. ACM, New York, NY, USA, 49–54. https://doi.org/10.1145/3139937.3139941

[20] Rogers Jeffrey Leo John, Jignesh M Patel, Andrew L Alexander, Vikas Singh, and Nagesh Adluru. 2018. A Natural Language Interface for Dissemination of Reproducible Biomedical Data Science. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 197–205.

[21] Rogers Jeffrey Leo John, Navneet Potti, and Jignesh M Patel. 2017. Ava: From Data to Insights Through Conversations. In *CIDR*.

[22] Malte F. Jung, Nik Martelaro, Halsey Hoster, and Clifford Nass. 2014. Participatory Materials: Having a Reflective Conversation with an Artifact in the Making. In *Proceedings of the 2014 Conference on Designing Interactive Systems (DIS '14)*. ACM, New York, NY, USA, 25–34. https://doi.org/10.1145/2598510.2598591

[23] Yoshiharu Kato. 2010. Splish: a visual programming environment for Arduino to accelerate physical computing experiences. In *Creating Connecting and Collaborating through Computing (C5), 2010 Eighth International Conference on*. IEEE, 3–10. https://doi.org/10.1109/C5.2010.20

[24] B.W. Kernighan, B.W.K.R. Pike, R. Pike, and B. Pike. 1999. *The Practice of Programming*. Addison-Wesley. https://books.google.co.kr/books?id=to6M9_dbjosC

[25] Yoonji Kim, Youngkyung Choi, Hyein Lee, Geehyuk Lee, and Andrea Bianchi. 2019. VirtualComponent: A Mixed-Reality Tool for Designing and Tuning Breadboarded Circuits. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Article 177, 13 pages. https://doi.org/10.1145/3290605.3300407

[26] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206. https://doi.org/10.1109/VLHCC.2004.47

[27] Michael J Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, et al. 2014. Principles of a debugging-first puzzle game for computing education. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, 57–64. https://doi.org/10.1109/VLHCC.2014.6883023

[28] Oscar Lucia, Isidro Urriza, Luis A Barragan, Denis Navarro, Oscar Jimenez, and José M Burdio. 2010. Real-time FPGA-based hardware-in-the-loop simulation test bench applied to multiple-output power converters. *IEEE Transactions on Industry Applications* 47, 2 (2010), 853–860.

[29] 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 (UIST '17)*. ACM, New York, NY, USA, 299–310. https://doi.org/10.1145/

3126594.3126658
[30] Clifford Nass and Youngme Moon. 2000. Machines and mindlessness: Social responses to computers. *Journal of social issues* 56, 1 (2000), 81–103.
[31] Clifford Nass, Jonathan Steuer, and Ellen R Tauber. 1994. Computers are social actors. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 72–78.
[32] Clifford Ivar Nass and Scott Brave. 2005. *Wired for speech: How voice activates and advances the human-computer relationship*. MIT press Cambridge, MA.
[33] Alisha Pradhan, Kanika Mehta, and Leah Findlater. 2018. "Accessibility Came by Accident": Use of Voice-Controlled Intelligent Personal Assistants by People with Disabilities. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 459, 13 pages. https://doi.org/10.1145/3173574.3174033
[34] Alexander Repenning. 2011. Making programming more conversational. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. IEEE, 191–194.
[35] Alexander Repenning. 2013. Conversational Programming: Exploring Interactive Program Analysis. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/2509578.2509591
[36] Alexander Repenning. 2017. Moving beyond syntax: Lessons from 20 years of blocks programming in AgentSheets. *Journal of Visual Languages and Sentient Systems* 3 (2017), 68–89.
[37] Alexander Repenning and Tamara Sumner. 1995. Agentsheets: A medium for creating domain-oriented visual languages. *Computer* 28, 3 (1995), 17–25.
[38] Joel Sadler, Kevin Durfee, Lauren Shluzas, and Paulo Blikstein. 2015. Bloctopus: A Novice Modular Sensor System for Playful Prototyping. In *Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '15)*. ACM, New York, NY, USA, 347–354. https://doi.org/10.1145/2677199.2680581
[39] Donald Schön. 2001. From Technical Rationality to reflection-in-action. In *Supporting lifelong learning*. Routledge, 50–71.
[40] Vidya Setlur, Sarah E. Battersby, Melanie Tory, Rich Gossweiler, and Angel X. Chang. 2016. Eviza: A Natural Language Interface for Visual Analysis. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 365–377. https://doi.org/10.1145/2984511.2984588
[41] 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 (UIST '17)*. ACM, New York, NY, USA, 321–330. https://doi.org/10.1145/3126594.3126618
[42] Amber Wagner, Ramaraju Rudraraju, Srinivasa Datla, Avishek Banerjee, Mandar Sudame, and Jeff Gray. 2012. Programming by Voice: A Hands-free Approach for Motorically Challenged Children. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems (CHI EA '12)*. ACM, New York, NY, USA, 2087–2092. https://doi.org/10.1145/2212776.2223757
[43] Jeremy Warner, Ben Lafreniere, George Fitzmaurice, and Tovi Grossman. 2018. ElectroTutor: Test-Driven Physical Computing Tutorials. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. ACM, New York, NY, USA, 435–446. https://doi.org/10.1145/3242587.3242591
[44] Laurie Williams, E Michael Maximilien, and Mladen Vouk. 2003. Test-driven development as a defect-reduction practice. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE, 34–45. https://doi.org/10.1109/ISSRE.2003.1251029
[45] 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 (UIST '17)*. ACM, New York, NY, USA, 343–349. https://doi.org/10.1145/3126594.3126646