# Visual Programming Environment to Generate Arduino Code for Wearable Toolkit

HYEON KYEONG KIM, University of Twente, The Netherlands

*Abstract.* Wearable technology has been flourishing for the past few decades, facilitating and making our lives more comfortable. Thanks to its ubiquity, the development of wearables does not limit to physical computing or IoT (Internet of Things) fields: it is applied in areas such as healthcare, fashion design, entertainment, and more. Due to this it is essential to meticulously integrate the expertise of these varying disciplines with physical computing in order to develop wearables that meet the needs of the users. This does, however, decelerate the process creating quick prototypes as the researchers from different fields face the challenges of not only learning the hardware, but as well as programming of physical computing.

This research offers insights about the current challenges and state-of-the-art of development for wearable technology, as well as design choices made for the prototype development of a code-generating visual environment platform for wearable toolkits.

Additional Key Words and Phrases: visual programming, no-code programming, wearable technology, compiler construction, automatic code generation

## 1 INTRODUCTION

Wearable technology refers to relatively small devices or computers that can be worn, or even embedded in human bodies, usually to enhance our daily lives [12]. Complex applications such as smart watches allow us to monitor our body's health situations and take appropriate actions according to the information. The versatile nature of wearable technology allows to open up to a broad range of development possibilities for different fields such as healthcare [7], service, or fashion and arts [2]. Due to this, the prospect of growth in wearable technology does not seem to cease: in 2019, Jayathilaka et al. [8] expected that the market for wearables will grow to US$160 billion by 2026 as the technology develops to more complex embedded systems such as textile-embedded sensors, actuators, and therapeutic solutions.

Despite its tremendously rapid development, Ferreira et al. [5] expresses the concern that the research in wearable technology still seems fragmented. Because it is so multidisciplinary, "there is a lack of integration between this research field's micro and macro perspectives" [5]. This requires thorough communication between the tech team and experts in different fields to simply create a prototype. This leads to inefficiency in the process of developing innovative tools. Moreover, the current wearable technology is more focused towards the wearable aspect of the product, while the building of a such prototype still remains complicated due to the barrier of programming. The developers then find it hard to quickly develop a prototype if they are not coming from an IT background and have no knowledge of programming. This hurdle could, in the end, result in a large obstacle to the development of such a growing market.

To address this issue, Mader et al. [10] assembled a toolkit (more detail in Section 3) for wearables that mitigate the hardware knowledge issue when building a prototype. This toolkit contains a few basic sensors and actuators and allows users to explore with ease with different wearable components. By following a plug-and-play paradigm, the users can quickly create novel prototypes without needing proficient knowledge of the hardware. To create these prototypes, however, the users are still required to understand the dynamics of programming to produce Arduino code. This creates an unbridgeable technical knowledge gap between people of IT backgrounds and those who are not.

Taking these aforementioned problems into consideration, this research aims to find and implement an appropriate design paradigm for a visual programming platform for the wearable toolkit. It will focus on composing a structure that (1) does not follow the existing traditional programmatic way of thinking and (2) is extensible.

### 1.1 Research Outline

We will first present the current challenges and research questions that will be answered throughout this paper in Section 2. Afterwards, Section 4 will discuss some state-of-the-art developments concerning visual programming and how they do not suffice as solutions for the discussed research questions. Later, Section 6 and Section 7 will detail the methods and concepts applied to the development of the visual programming platform. The results of the prototype usability test will be described in Section 8, followed by the conclusions and discussions in Sections 9 and 10.

## 2 PROBLEM STATEMENT

The current wearable toolkit does not have a software for visual programming, and some of the workshops carried out with this toolkit evidenced that programming could be a bottleneck for participants from non-programming backgrounds. To broaden the range of users of different disciplines, Mader et al. [10] suggest the implementation of a visual platform to mitigate from the discussed problem. As the objective of the paper focuses on the design of such a platform, the research will mostly follow the methodologies of a design-based research inspired by Wieringa [14].

To this end, this paper aims to answer the following general Research Question:

*How to design a visual programming platform for wearable toolkit that generates Arduino code so that stakeholders without IT background can also create prototypes without the barrier of programming?*

This technical research question is followed by the next set of sub-questions formulated to clarify the research objective:

(1) What are the set of programs that should be possible to describe with the graphical input and what would be a suitable abstract representation of this set?

*TScIT 37, July 8, 2022, Enschede, The Netherlands*

(2) How to implement a graphical interface to define a program? What are the requirements of this graphical interface?
(3) How to translate the abstract representation to Arduino code?

(4) How to design the platform in such a way that it is modular and extensible?

## 3 HARDWARE – WEARABLE TOOLKIT

The wearable toolkit in concern consists of several sensors, actuators, and a single minimalistic microcontroller component called 'Arduino Beetle' (similar to Arduino Micro) with 3 analog input pins and 3 digital output pins [10]. Additionally, it comes with informational cards that contain the names and drawings of the respective materials and some starter project ideas that the user can easily get started with. The full list of components in the wearable toolkit is listed in Figure 1.

As mentioned previously, there are currently no visual programming platforms that are suitable for novice programmers for this toolkit. Although there are platforms such as the ones mentioned in Section 4, these are more focused on educating people on how to program, rather than keeping it out of concern in the first place. They still enforce users to follow the programming way of thinking, which is something that should not be required from people of different disciplines.

## 4 RELATED WORK

Prior to the research, gathering related literature about similar work has been carried out with IEEE, ScienceDirect, and Google Scholar. Some key terms such as "wearable technology", "end-user programming", "visual programming" and "Arduino" showed relevant studies and works done in the respective areas.

### 4.1 Block-based Coding

The following works are oriented towards block-based coding, allowing the users to interlock puzzle-resembling blocks of code to simulate a piece of actual code. The visual programming environment for Arduino currently developed in the market mostly consist of fork projects of Google Blockly[1], which uses block-based coding as its main mechanism.

Ardublockly[2] is a programming platform specifically for Arduino products, adapted from Google Blockly [3]. It generates Arduino codes constructed from drag-and-drop blocks and loads the program to the connected Arduino board, and it works on various platforms. Although Ardublockly gave a great initiative in providing an easier environment for developers to prototype their projects, it missed out on supporting various educational contexts. Additionally, there are also projects such as senseBox Blockly[3] that provide a coding environment for their custom toolkits with sensors and actuators. On a similar context, ArViz [3] uses the same coding scheme to coach novice programmers on the Internet of Things (IoT) development concepts [3]. Finally, there are platforms such as mBlock[4] that are more focused on the educational experience of programming with Arduino. The list of block-based coding platforms continues, but for the purpose of this proposal I have only listed out the most relevant ones to this research.

All the systems mentioned above translate conceptual-level instructions to a lower-level programming language in their own ways, targeting a multidisciplinary set of users. These tools provide a "'low [threshold]' (easy to get started) and a 'high ceiling' (opportunities to create increasingly complex projects over time)" [13]. However, most of these platforms still use the fundamental sequence and logic of programming, or they are not suitable for the wearable toolkit. They merely simulate conceptual programming, using the same logic operators and paradigms simply made more visually appealing. Moreover, most of the platforms listed above – especially mBlock – are specifically targeted toward novice programmers on how to code, leaving other people from non-IT backgrounds out of the scope.

### 4.2 Flow Chart based programming IDE

In the 2010 Eighth International Conference on Creating, Connecting and Collaborating through Computing, Kato presents Splish, a visual programming software that lowers the barrier for physical computing [9]. Unlike the platforms discussed above, Splish is not based on Google Blockly but its main interaction is done with icons representing objects such as LED. The objects are mapped out on the interface resembling more to a diagram than a sequence of code instructions. This way the user is allowed to elude from the traditional way of thinking of programming, which is more welcoming for non-technical users. However, this program lacks compatibility with other sensors or actuators and is very outdated. Some other similar flow-chart/block-diagram based IDEs for physical computing include Simulink [1] and DrawCode[11]. Inspired by these applications, the graphical user interface of the visual programming platform will follow a similar design pattern to engage the users in configuring their program, rather than programming them with code.

## 5 REQUIREMENTS

The primary objective of this research is to develop a visual programming platform so that the different experts from disciplines other than IT so that they can quickly develop prototype projects for the aforementioned wearable toolkit. The prototype must relieve the users from programmatical thinking by minimizing programming syntax. Therefore, this research emphasizes on the following set of requirements:

(1) Functional Requirements:
    - User must be able to configure a control flow statement.
    - User must be able to configure the delay time after each execution.
    - User must be able to configure the type of sensors and actuators.
    - After configuration, the user must be able to produce Arduino code that they can open in their Arduino IDE.
(2) Non-functional Requiremnts:
    - The environment must not resemble the target code whilst keeping the logic accurate.

[1]https://developers.google.com/blockly
[2]https://ardublockly.embeddedlog.com/
[3]https://blockly.sensebox.de/
[4]https://mblock.makeblock.com/en-us/

| Sensors | Actuators | Control & Connectivity Components | (Not included) Tools and Supplies |
|---|---|---|---|
| Bending Sensor | Buzzer | Microcontroller (Arduino Beetle) | Glue Gun |
| Touch Sensor | Miniature RC Servo | 3-pin Male-Female Jumper wires | Cardboard |
| GSR Sensor | Light strip (Neopixel) | 5V Power bank | Staples |
| HR (Heart Rate) Sensor | Vibration motor | | Tape |
| Accelerometer | | | Tie-wrap |
| Sound sensor | | | Velcro tape |
| Distance sensor | | | Sewing machine |

Table 1. List of components of the wearable toolkit

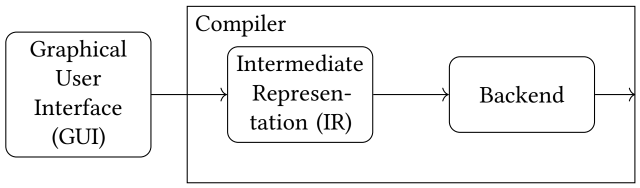
Compiler
Graphical User Interface (GUI)
Intermediate Represen-tation (IR)
Backend

Fig. 1. Conceptual structure of the compiler for the visual programming platform for wearable toolkit

- Extensbility: The system must be designed in a way that the following are supported with minimum effort :
  - Adding / modifying number of ports
  - Adding / modifying sensor and actuator types
  - Support for logical operators such as `AND`/`OR`/`NOT`.

## 6 SYSTEM DESIGN

The visual programming platform is largely divided in two parts – the graphical user interface (GUI) where user interaction is handled, and the compiler that takes the user input, translates to an Intermediate Representation (IR) and finally generates code via the backend. This conceptual structure of the prototype is shown in Figure 1. For these different parts, there are several design considerations that must be taken into account. This section details all the design choices made throughout the development as well as the reasoning behind each choice.

### 6.1 Visualization

The design of the user interface was mostly inspired by Splish from Kato [9] and DrawCode from Mahesh and Sivraj [11], while satisfying all the criteria detailed in Section 5. Most importantly, it must be emphasized on designing a platform where each graphical component does not directly represent programming code like Blockly applications.

*6.1.1 Configuration.* First, the user is presented with several input and output nodes at each end of the program window with expression nodes in between. As the platform is not connected to the microcontroller, it will not have any information about the peripherals attached to the microcontroller. Therefore, with the help of the informational cards included in the toolkit, the user must configure and specify to which sensor the analog input node is connected, and the same applies to an actuator with the digital output node. Subsequently, the user can then interconnect these nodes, allowing them to further configure `if... else` control flow statement in the expression block. The basic example of one-to-one statement configuration is shown in Figure 2.

The following is the list of expressions that must be considered valid in the program:

(1) 1 Input - 1 Output connection with control flow statement: one sensor value affects one actuator
(2) 1 Input - `n` Output connections: one sensor input can affect the values of `n` actuators
(3) `n` inputs-1 output with input expressions connected with logical operators. For example: `A0 > 50 AND A1 < 60 : D9 = 20`)
(4) Input - Output direct connection without expression block: while there are not many use cases for this, there are some edge cases that can be covered with this connection. For example, a servo does not need control flow statements. Instead, it can take the analog input value and directly output a mapping from 0 to 180 degrees.

The following expressions are not considered valid expressions:

- Input - Input : this should not happen anyway, as there is no use case of connecting two analog inputs together.
- Output - Output : similar to the input connections, one actuator value should not affect the other actuator.
- Expression - Expression : also an invalid argument, there is no use case where the expressions are affecting each other while taking no input values and producing no output values.
- `n` Input - 1 Output configured in different expression blocks : if the digital output value depends on two different control flow expressions (for example: `A0 > 50 : D9 = 20 ; A1 < 60 : D9 = 100`) without being connected with logical operators (such as `and` or `or`), the program will not know which value to accept. Therefore, this expression should not be accepted. This leads to the conclusion that digital output statements should only be connected to a single expression block.

*6.1.2 Expression blocks.* The basic program flow should follow that a sensor input value affects an actuator output value through some control flow statement. Therefore, it was concluded that `if . . . then . . . else . . .` statements are the most appropriate choice as it has the most use case when building a wearable prototype. Also, it is the most expressive out of the other control flow

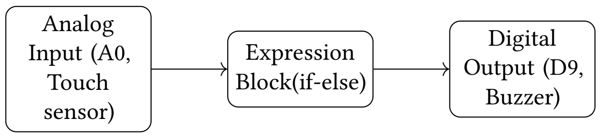
Analog Input (A0, Touch sensor)
Expression Block(if-else)
Digital Output (D9, Buzzer)

Fig. 2. Basic one-to-one expression flow in GUI

statements. More complex statements such as `while`, `switch`, `for`, or `do` are not expressed as they have little to no use cases.

Expression blocks must also come with a single `delay` variable each, which can also be configured by the user. If the user does not wish to have a delay value, they can leave it blank (and the delay value will be interpreted as 0). While this could be expressed with a simple boolean component if the user wants to add a delay, it has been designed in a way that the user leaves it blank if they do not wish to configure a delay time, as this does not affect the expressiveness of the delay function itself.

## 6.2 Backend

Once the user has completed all the necessary configurations, they must be able to finalize the program by clicking on a button to finally generate the code according to their specifications. The IDE will compile the interface, eventually generating a `.INO` file (Arduino sketch file) that the user can directly open from the Arduino IDE. For this, we will be following the general process of constructing a compiler inspired from Cooper and Torczon [4] using Java. However, while a traditional compiler requires a frontend that analyzes a source program, it is not needed in this research as it does not have to interpret another source code. As a frontend will be unnecessary, lexical analysis and syntax analysis can be discarded from part of the development process and generate Intermediate Representation (IR) immediately from the GUI (more will be discussed in Section 6.2.2). From there, the program will finally be able to generate Arduino source code according to the user's configurations.

*6.2.1 Abstraction and Intermediate Expression.* First, the IR that will represent the expressions and statements made in the GUI must be determined. Generally, the flow of the program looks like the following: the digital output value is dependent on the value received after calculating the control flow statements (expressions), and those statements are dependent on the value of the analog input value/s. Therefore, the program must traverse the parse tree in that order due to its dependence hierarchy, making a nested structure starting with the digital output statement. This flow resembles the diagram shown in Figure 2, read from right to left. Therefore, our parse tree starts from the `DigitalOutputStatements` which are the children of the `Program` node, and the `AnalogInputExpressions` are the leaves of the tree.

An example of such IR is shown in Figure 3. It starts with the program node, and immediately afterward an `updateFrequency` is defined as well as a `DigitalOutputStatement`. Here we can see how the rest of the nodes are defined and nested until the program gets to the `AnalogInputExpression`.

*6.2.2 Code generation.* The target Arduino code is divided into 5 parts: (1) import statements, (2) global variable statements, (3) setup function, and the loop function which is subdivided to (4) prologue and (5) epilogue. By tracking and managing these 5 different fragments of the target code, the code generation can be easily kept modular and extensible. In the case that the stakeholders of the wearable toolkit wish to expand on the types of sensors/actuators and they need to generate new lines of code accordingly, they can just add the statements depending on which part it belongs to. To generate each of these code fragments, the visitor must traverse through the abstract syntax tree (AST). For example, a `DigitalOutputStatement` with Neopixel as the `ActuatorType` will generate an import statement for the Adafruit Neopixel library. Throughout the traversal, the generated statements are stored as global variables of the `ProgramVisitor` node. When the `ProgramVisitor` is done with its traversal, it concatenates all the generated statements and finally returns the target source code.

There were considerations of generating the source code on the fly, without the user having to click the button to indicate completion of configuration. However, this method can be inefficient as whenever the user adds more configurations, the program must go through the entire source code file and generate the code fragment in the correct line. Instead, it is easier to generate the code in one go when the user has completed their configuration.

# 7 IMPLEMENTATION

## 7.1 Graphical User Interface

To build the interface, we are using Processing [5], an open-source and flexible graphical library for Java. The simplicity and accessibility of Processing allows to build a quick interface and create an IR for the backend. Also, with the use of controlP5 [6] library, common GUI components such as buttons and number fields could be easily implemented. There are also other softwares such as Qt [7] or JavaFX [8] for building a GUI, however, Processing is more accessible and more suitable for quick prototyping. Additionally, Processing is simply an extension of Java, so it makes a cross-platform program which means that the IDE can be built in any operating system (OS).

The IDE in which the user can configure can simply be executed by running the main program. Then, the user will be presented with an interface such as the one shown in Figure 4. In this figure, we can find one-to-one mapping of two sensors and two actuators. For the configuration, user can simply drag their mouse from one node to another to make a new connection. When a connection is made, necessary expressions corresponding to the connected node/s will appear on the expression block for the user to complete the statement. Afterwards, the user can configure the statement to fit their use case by altering the values of `if. . . then. . . else` statements as well as `delay` values. They can either fill in the number field with keyboard inputs or mouse drags. Furthermore, to let the program know which ports the sensors/actuators are connected to, the user must also specify those according to the physical connection in the microcontroller. They can do so by clicking on the input / output box, which pops up a menu like shown in Figure 5. Here

[5]https://processing.org/
[6]https://www.sojamo.de/libraries/controlP5/
[7]https://www.qt.io/
[8]https://openjfx.io/

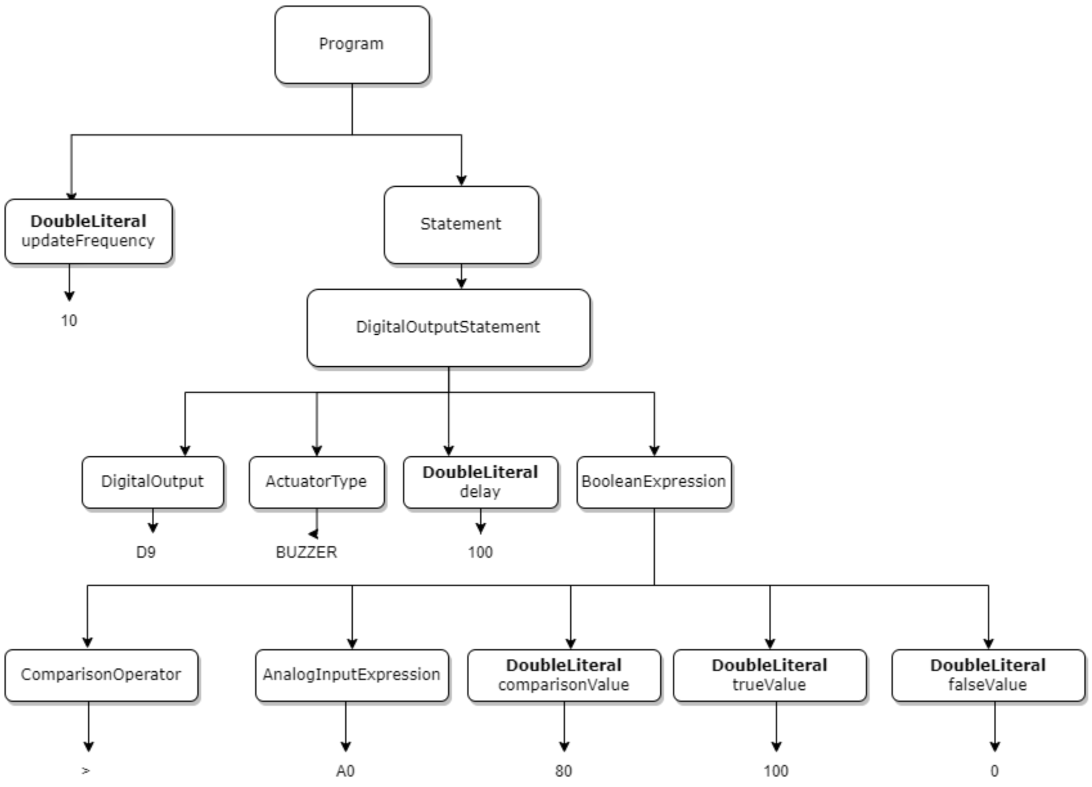

Fig. 3. Intermediate Representation (IR) of the no-code programming platform for wearable tookit

the user has a complete list of the sensors/actuators from Table 1, and they can choose the peripheral that they have connected to the microcontroller. Finally, the user can click the green button located in the top right corner of the window (Figure 4) to complete the configuration, which creates a new `.INO` file that can be opened with the Arduino IDE. The user can then simply upload the generated code to the microcontroller using the IDE.

*7.1.1 Extensibility.* The wearable toolkit can be easily expanded in terms of sensors and actuators, and so should this visual programming platform as well. In the case that more input/output ports are available on the microcontroller, then additional pins can simply be instantiated with a unique pin number (such as A0 or D9) in the `Background` class of the GUI. The GUI will then render and display the added pin/s accordingly. The expression blocks can also be added in the same way in case the user needs more of them. Adding the pins must be done in the source code as the pins and the types of sensors and actuators should already be defined before the users are able to explore around with the wearable toolkit.

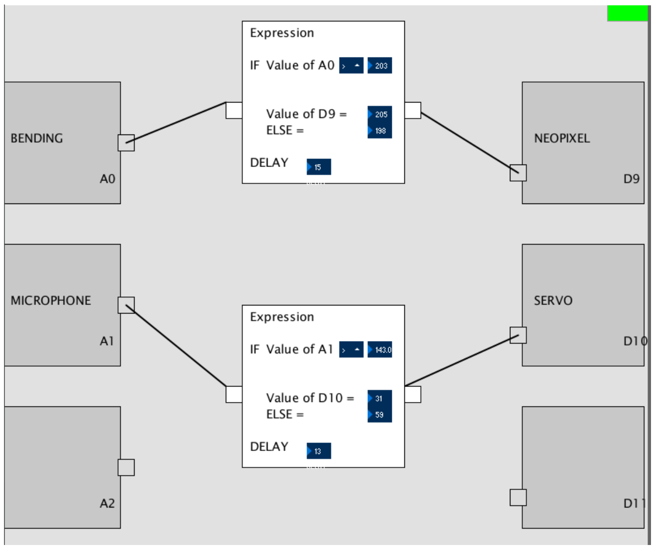

Fig. 4. Example of 2 input - 2 output one-to-one mapping in GUI

## 7.2 Program Visitor

To generate the code, the program must traverse the IR generated after the user configuration. For this, a simple `ProgramVisitor` inspired by Gamma et al. [6] was implemented. Visitor pattern was chosen over a listener pattern as visitor pattern is more flexible and has full control of the traversal of the AST. Also, since visitor methods return the accepted value, the `ProgramVisitor` does not have to store all the accepted values, which distributes the memory burden from the `Program` node. The `ProgramVisitor` can be used

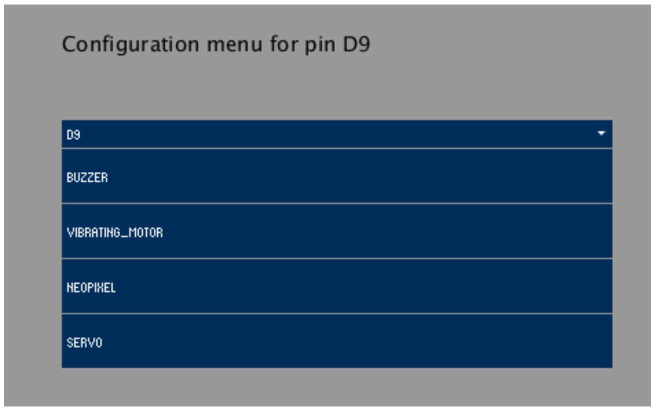

Fig. 5. Menu for configuration of Digital Output Pin (D9)

essentially to define the operations for type-checking, analyze control flow statements, and check for variables from the children node with assigned values before used by the parent node. And most importantly, we use this `ProgramVisitor` to generate structured code like mentioned in 6.2.2.

```
public void visit(DoubleComparisonExpression expression)
     {
    var rightExpr = exprStack.pop();
    var elseVal = exprStack.pop();
    var thenVal = exprStack.pop();
    var leftExpr = exprStack.pop();
    var operator = switch (expression.operator) {
        case EQUALS -> "==";
        case NOT_EQUALS -> "!=";
        case GREATER_THAN -> ">";
        case GREATER_THAN_EQUALS -> ">=";
        case SMALLER_THAN -> "<";
        case SMALLER_THAN_EQUALS -> "<=";
    };
    var expr = String.format("(%s %s %s) ? %s : %s",
     leftExpr, operator, rightExpr, thenVal, elseVal);
    exprStack.push(expr);
}
```

Listing 1. DoubleComparisonExpression visitor

*7.2.2 Extensibility.* Thanks to the visitor pattern, as long as the grammar that the compiler accepts does not change, it is easy to add new functionality by defining a new concrete NodeVisitor that implements the `ProgramVisitor`, like shown in Listing 1. This can be done by modifying the source code by adding a new `record` class that implements the `ProgramNode` and specify the parameters (grammar) that it needs, like shown in Listing 2. Here we can see that the DoubleComparisonExpression needs some `operator`, left and right `expressions`, and finally `if` and `else` values. Later, the appropriate code fragment that it must produce (or the values that must be passed) can be implmemented like shown in Listing 1. Meanwhile, adding sensor and actuator types is done quite easily: they can simply be added as an `enum` in `SensorType` or `ActuatorType`.

```
public record DoubleComparisonExpression(
     ComparisonOperator operator,
    DoubleExpression leftExpr,
    DoubleExpression rightExpr,
    DoubleExpression ifValue,
    DoubleExpression elseValue) implements BoolExpression
      {
    @Override
    public void accept(ProgramVisitor visitor) {
        leftExpr.accept(visitor);
        ifValue.accept(visitor);
        elseValue.accept(visitor);
        rightExpr.accept(visitor);
        visitor.visit(this);
    }
}
```

Listing 2. Creating a new node for `DoubleComparisonExpression`

*7.2.3 Example of Generated Code.* Listing 3 shows the final generated code by traversing the abstract syntax tree in Figure 3. Notice how the code fragments are separated in different parts. In this example, there were no import statements generated as none of the sensors or actuators required additional library to load. The global variable statements and the setup are generated according to the `DigitalOutputStatements` and `AnalogInputExpressions` found while traversing the tree. If the actuator were to be Servo, for example, then it would have had an additional `#include <Servo.h>` code as import statement and `servo.attach(PORT_D9)` in the setup function.

```
// Import statements
// Global variable statements
static const uint8_t PORT_A0 = A0;
static const uint8_t PORT_D9 = 9;

void setup(){
    // Setup
    pinMode(PORT_A0, INPUT);
    pinMode(PORT_D9, OUTPUT);
}

void loop() {
    // Prologue
    double DIGITAL_OUT_D9 , DIGITAL_OUT_D10 ,
     DIGITAL_OUT_D11;

    double ANALOG_IN_A0 = analogRead(PORT_A0);
    DIGITAL_OUT_D9 = (ANALOG_IN_A0 > 80.0) ? 100.0 : 0.0;

    // Epilogue
    analogWrite(PORT_D10,DIGITAL_OUT_D10);
}
```

Listing 3. Code generated from a simple if. . . else program

## 8 USABILITY TESTING

### 8.1 Fulfilled Requirements

To evaluate the basic requirements of the prototype, a moderated usability test was conducted with one of the main stakeholders of this prototype, Dr. Ir. Edwin Dertien. We consider him as the appropriate test user as he has also contributed in the development of the wearable toolkit and therefore is interested in the prototype of the software [10]. As a minimum viable product, the prototype must have fulfilled the requirements listed in Section 5. In this section, we will list out the requirements satisfied as well as the feedbacks given by the stakeholder.

### 8.2 Functional Requirements

(1) *User must be able to configure a control flow statement.* The user was able to configure each of the values of the `if... then ... else ...` statements with keyboard input (for numerical values) and mouse clicks (for dropdown menus).
(2) *User must be able to configure the delay time after each execution.* The user was able to configure the delay value on the expression block with keyboard input to specify the numerical value.
(3) *User must be able to configure the type of sensors and actuators.* By clicking on each of the input/output boxes, they were able to specify the type of sensor/actuator by mouse click on the dropdown menu. The user was also able to see their choice of sensors/actuators of the corresponding pins in the main screen.
(4) *After configuration, the user must be able to produce Arduino code that they can open in their Arduino IDE.* The user was able to click on the green button on the top-right corner of the screen after they have completed their configuration and finally generate the Arduino code. They could double click on the generated `.INO` file and a new sketch would be made in the Arduino IDE.

### 8.3 Feedback and Suggestions

After the usability testing, there were some feedbacks from the stakeholder about the paradigm of the design on the graphical user interface. The current program takes one sensor input value and directly connect it to the actuator output value, either with or without an expression. However, this could limit the expressiveness of the user on certain programs. For example, in the case of Neopixel, several configurations can be done in terms of the color of the LEDs, the intensity, patterns, and possibly more. However, some of these configurations would have to be set as a default static value as only one variable can be configured by the user. Therefore, Dr. Dertien has given the suggestion to modify the current paradigm in a way that the program can accept multiple variables depending on the different sensor/actuator. For example, a buzzer would only need a `tone` variable while a Neopixel could have `color`, `intensity`, and `pattern`). Also, by making these configurations directly in the input/output blocks rather than the expression blocks, it is possible to distribute the burden of information that the expression block needs to hold.

## 9 CONCLUSION AND DISCUSSION

In this paper we have seen the design decisions and the implementation procedure of the prototype for the visual programming platform for wearable toolkit. Some of the challenges that were faced by experts from different disciplines in programming were discussed and how that is a bottleneck to creating new prototypes for wearable technology. Additionally, some of the work related to a visual programming environment for physical computing were presented. In the following sections we will discuss how each of the research questions formulated in Section 2 were addressed and answered throughout this research.

### 9.1 Answering RQ 1

The set of expressions that should be described by the graphical interface and a possible abstract representation were presented in Sections 6.1 and 6.2. The intermediate representation was designed based on the dependency hierarchy of the program and the variables that the program must store in order to produce the end source code.

### 9.2 Answering RQ 2

As described in Section 7.1, we have chosen to work with Processing in building the graphical user interface, according to the requirements described in Section 5. For each design choice, the program was made as modular as possible.

### 9.3 Answering RQ 3

To translate the abstract representation (IR) to Arduino code, a `ProgramVisitor` was implemented as introduced in Section 7.2. Using the Visitor pattern, the program traverses through each of the nodes in the abstract syntax tree such as the one shown in Figure 3. By implementing each visit methods for each element in the tree like Listing 1, we were able to generate the necessary fragments of code to complete the program.

### 9.4 Answering RQ 4

Throughout the trajectory of this research, extensibility was constantly emphasized. Modularity and extensibility is something that was considered both in the GUI and the compiler construction, as described in Sections 7.1.1 and 7.2.2 respectively. For GUI, adding components such as pins, expression blocks should be done effortlessly. Similarly for backend, adding sensors and actuators should also be as easy as adding a new `enum` value in `SensorType` and `ActuatorType`. Finally, of course, the developer should make sure that the frontend (GUI) communication is handled correspondingly in the backend (code generation).

We hope that, by the development of this prototype, we have alleviated the burden of programming knowledge in physical computing and that this prototype would be used as a stepping stone for quick prototyping for the aforementioned wearable toolkit. It must be mentioned, however, that the prototype is not yet polished and does not cover all the use cases.

## 10 FUTURE WORK

The prototype produced in this research covers the basic use cases for the minimum functionality of the visual programming platform. Luckily, the structure of the program is extensible enough that allows to expand on the features without requiring much effort. However, there are some edge cases that must be covered by adding more implementation. In this section we list out some of the areas of improvement for the prototype.

Firstly, the control flow statements in the expression blocks always produce one output value for each `if. . . then. . . else. . .` expression. However, actuators such as Neopixel should be able to have more configuration options than other simple actuators, because it has additional variables such as color and intensity, as mentioned during the usability testing in Section 8.3. For this, there should be an option for a more tailored configuration for actuators that should be able to receive multiple variables. Similarly, the servo currently only maps from 0 to 180 degrees, depending on the analog input value. However, if we want to keep this more flexible, the user must also be able to configure this to fit their use case as well. This can be done by adding and rendering a new connection node in the input/output box, and labeling what each node represents (e.g. color, intensity). Then, this expansion will allow each of the variables to be handled by different peripherals. For example, the color of the Neopixel could be handled by an input value of sensor pin A0, while the intensity could be dependent on pin A1, and so on. This could give a lot more expressiveness of the program to the users without limiting them to the simplest configurations.

Additionally, the program must be able to accept more complex conditions with logical operators such as `if... and... if...`. Currently the backend fully supports such grammar, however, the GUI does not. This could be done by adding another dropdown list of logical operators between two `if` statements when two or more analog inputs are connected to the same expression block. Currently, a single `AND` operator is supported from the GUI.

Finally, the UI needs more tweaking in error handling and friendlier with modifications. Currently, the user cannot remove a connection once they have created one, so they must restart the program and start the whole configuration process again. To prevent this, there could be a delete button implemented on each connection and when the user clicks on them, they can be deleted. Also, the GUI will throw an error and stop the program if the user has forgotten to specify the sensor and actuator types. This is because the type-checking feature of the `ProgramVisitor` does not allow empty `SensorType` or `ActuatorTypes`, as they are essential for the generation of code. However, instead of throwing an error and stopping the whole program, there must be a smoother error handling methodology (e.g. reminding the user to specify the sensor type and actuator type when it's blank). Furthermore, the GUI does not handle well in connecting digital output pins to the expression blocks before connecting analog input pins. The user must be able to connect the pins in whichever order, so this should also be handled in a different manner.

## SOURCE CODE

The source code for the prototype of the visual programming platform can be found at https://github.com/rlagusrud61/wearable_nocode. Also, some example programs that the prototype can generate are listed on this link.

## ACKNOWLEDGEMENTS

I would first like to thank Dr. Angelika Mader, the supervisor of this research, and Dr. Edwin Dertien for providing helpful suggestions for the prototype implementation throughout the research project. I would also like to extend my gratitude towards the track chair Dr. Mariët Theune for her guidance and constructive feedback in every track meeting. Finally, I give my thanks to the reviewers for their thoughtful comments and suggestions.

## REFERENCES

[1] 2022. Generate C Code from Simulink Model - MATLAB & Simulink. https://www.mathworks.com/help/dsp/ug/generate-c-code-from-simulink-model.html [Online; accessed 26. Jun. 2022].
[2] J. Birringer and M. Danjoux. 2009. Wearable technology for the performing arts. , 388-419 pages. https://doi.org/10.1533/9781845695668.4.388
[3] Kitsiri Chochiang, Kullawat Chaowanawatee, Kittasil Silanon, and Thitinan Kliangsuwan. 2019. Arduino Visual Programming. *2019 23rd International Computer Science and Engineering Conference (ICSEC)*, 82–86. https://doi.org/10.1109/ICSEC47112.2019.8974710
[4] Keith D. Cooper and Linda Torczon. 2011. *Engineering a Compiler.* Morgan Kaufmann. https://doi.org/10.1016/C2009-0-27982-7
[5] João J. Ferreira, Cristina I. Fernandes, Hussain G. Rammal, and Pedro M. Veiga. 2021. Wearable technology and consumer interaction: A systematic review and research agenda. *Computers in Human Behavior* 118 (5 2021), 106710. https://doi.org/10.1016/j.chb.2021.106710
[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc. https://doi.org/10.5555/186897
[7] A. Godfrey, V. Hetherington, H. Shum, P. Bonato, N.H. Lovell, and S. Stuart. 2018. From A to Z: Wearable technology explained. *Maturitas* 113 (7 2018), 40–47. https://doi.org/10.1016/j.maturitas.2018.04.012
[8] Wanasinghe Arachchige Dumith Madush Jayathilaka, Kun Qi, Yanli Qin, Amutha Chinnappan, William Serrano-García, Chinnappan Baskar, Hongbo Wang, Jianxin He, Shizhong Cui, Sylvia W. Thomas, and Seeram Ramakrishna. 2019. Significance of Nanomaterials in Wearables: A Review on Wearable Actuators and Sensors. *Advanced Materials* 31 (2 2019), 1805921. Issue 7. https://doi.org/10.1002/adma.201805921
[9] Yoshiharu Kato. 2010. Splish: A Visual Programming Environment for Arduino to Accelerate Physical Computing Experiences. *2010 Eighth International Conference on Creating, Connecting and Collaborating through Computing*, 3–10. https://doi.org/10.1109/C5.2010.20
[10] Angelika Mader, Edwin Dertien, Judith Weda, and Jan Van Erp. 2022. Tinkering With Social Touch Technology. (2022). under publication.
[11] M. Mahesh and P. Sivraj. 2017. DrawCode: Visual tool for programming microcontrollers. In *2017 3rd International Conference on Advances in Computing, Communication & Automation (ICACCA) (Fall)*. IEEE. https://doi.org/10.1109/icaccaf.2017.8344708
[12] Aleksandr Ometov, Viktoriia Shubina, Lucie Klus, Justyna Skibińska, Salwa Saafi, Pavel Pascacio, Laura Flueratoru, Darwin Quezada Gaibor, Nadezhda Chukhno, Olga Chukhno, Asad Ali, Asma Channa, Ekaterina Svertoka, Waleed Bin Qaim, Raúl Casanova-Marqués, Sylvia Holcer, Joaquín Torres-Sospedra, Sven Casteleyn, Giuseppe Ruggeri, Giuseppe Araniti, Radim Burget, Jiri Hosek, and Elena Simona Lohan. 2021. A Survey on Wearable Technology: History, State-of-the-Art and Current Challenges. *Computer Networks* 193 (7 2021), 108074. https://doi.org/10.1016/j.comnet.2021.108074
[13] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch. *Commun. ACM* 52 (11 2009), 60–67. Issue 11. https://doi.org/10.1145/1592761.1592779
[14] Roel J. Wieringa. 2014. *Design Science Methodology for Information Systems and Software Engineering.* Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-43839-8