Automated Design Tool for Arduino Circuits

A Major Qualifying Project

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Gabriel Buziba (ECE)

Yangyang Jin (ECE)

Andres Negron (CS)

Casey Wohlers (CS)

Date:

April 25th, 2024

Report Submitted to:

Professor David C. Brown (CS)

Professor Pradeep Radhakrishnan (MME)

I. Abstract

In engineering education, students often face challenges with microcontrollers, sensors, and actuators, hindering their progress and leading to time-consuming troubleshooting. Novice learners in Mechanical, Electrical, and Robotics Engineering struggle with wiring components, coding, and interpreting circuit results.

Our project, ADArC, seeks to alleviate this struggle by offering an intuitive simulator designed for circuit design and code generation. Users can choose components, have them automatically connected to the Arduino, generate and edit pre-written code, and run and modify simulations in real time. When showcasing our project in a class setting, students not only showed enthusiasm for this app's design and accessibility, but also praised its efficacy in enhancing their learning experiences.

In this report, we cover our project motivations, initial research, and our goals. We then proceed to examine our planning process, leading to our designs, and finally, our implementations. We conclude by evaluating the results of a survey we carried out as well as reflecting on what each group member gained as a result of the project.

II. Acknowledgments

We would like to thank Professor Radhakrishnan and Professor Brown for their support and guidance through our project. We would also like to thank Cam Tu Le for his work on his thesis paper, *An Automated Design Tool for Sensor and Actuator Integration in Mechatronic Systems*, for giving us the blueprints for what ADArC could become. Another group we want to thank is the BoGL Web team for their website framework, which showed us a structure around which to build ADArC. Furthermore, we want to thank the C-term 2024 ME/RBE 4322 class for their participation and feedback on our project.

III. Authorship

Sections	Author
1. Introduction	Casey Wohlers
2. Motivation & Problem Statement	Gabriel Buziba
3. Literature Review	All
3.1 The Arduino	Gabriel Buziba
3.2 Cam Tu Le's Thesis	Casey Wohlers
3.2.1 Graphs	Yangyang Jin
3.2.2 Code Libraries	Casey Wohlers
3.3 GraphSynth	Yangyang Jin
3.4 BoGL MQP Summary	Casey Wohlers
4. Goals & Requirements	All
5. Methodology	All
5.1 Solutions for Wiring and Representing Circuits	Casey Wohlers
5.2 Solutions for Circuit Simulation	Casey Wohlers
5.3 Solutions for Code Generation	Casey Wohlers
5.4 Solutions for Documentation	Casey Wohlers
5.5 Programming Languages	Casey Wohlers
5.6 Components List	Gabriel Buziba
6. ADArC Design	All
6.1 UI Design	Andres Negron
6.1.1 Prototype	Andres Negron

6.1.2 Component Palette	Andres Negron
6.1.3 Code Panel	Andres Negron
6.1.4 Toolbar	Andres Negron
6.1.5 Help Design	Gabriel Buziba & Andres Negron
6.2 Intended User Interaction Order	Andres Negron
6.3 GraphSynth Rules Edits	Yangyang Jin
6.3.1 Connection Lines and Color Coding	Yangyang Jin
6.3.2 Combining Components	Yangyang Jin
6.4 Visual Component Design	Gabriel Buziba
6.4.1 Wokwi Conversions	Gabriel Buziba
6.4.2 Creating New Components	Gabriel Buziba
7. ADArC Implementation	All
7.1 UI Design	Andres Negron
7.2 Component System	Casey Wohlers
7.3 Adding Components	Casey Wohlers
7.4 Connection Lines	Casey Wohlers
7.5 Code Generation	Yangyang Jin & Casey Wohlers
7.6 Component Removal	Yangyang Jin
7.7 Component Simulation	Casey Wohlers
7.7.1 Translation Functions	Yangyang Jin
7.8 Rules Loading	Yangyang Jin & Andres Negron
7.9 Environmental Settings	Andres Negron
7.10 Code Console	Casey Wohlers
7.11 Deployment	Casey Wohlers

IV. Table of Contents

V. List of Figures

VI. List of Tables

1. Introduction

The mission of the Arduino company is to make Arduino products available to everyone and break down the barrier between electronics, design, and digital technologies (Arduino, 2021). Their path to achieving this goal is through open-source microcontrollers. These are cheap, relatively easy to use, and can be used with breadboards, sensors, and actuators. They come with an intro-level programming language and an Integrated Development Environment (IDE). Arduino microcontrollers also have a resourceful community with thousands of example projects. These factors make Arduino microcontrollers a good choice to use for educational purposes. However, this does not mean that they are simple to use. For example, the Arduino Uno has over thirty points for wire connections, comprising a dizzying array of functionality.

For some students, the Arduino may be difficult to use for one reason or another. Finding the information and datasheets for certain components, as well as wiring everything to their specific pins, is a difficult task for beginners. With this in mind, creating an application that can solve these and other issues could be very helpful for struggling students.

In August 2021, Cam Tu Le released a report detailing a framework for generating circuit wiring, circuit simulation, and code generation along with a detailed help section (Le, 2021A). These modules formed the backbone of our project and guided our work over the duration of the project. His paper's suggested framework became one of our main goals; other goals will be described in Chapter 4.

This paper is organized into eleven chapters. Chapter 2 covers our motivation and problem statement. Chapter 3 describes our research and literature review prior to starting. Chapter 4 states our goals and requirements for having a successful project. Chapter 5 covers possible solutions to each of the main parts of ADArC. Chapter 6 goes over the design of ADArC's UI and Interaction Order. Chapter 7 describes how we implemented many of the solutions we discussed in Chapter 5. Chapter 8 explains the testing process for the GraphSynth Rules, Connection Lines, and Translation Functions. Chapter 9 covers data gathered from a User Feedback survey ran in 'ME/RBE 4322 Modeling and Analysis of Mechatronic System'. Chapter 10 evaluates the project based on how well we achieved the goals mentioned in Chapter 4. Chapter 11 provides a conclusion to our project as well as reflections from the team.

2. Motivation & Problem Statement

Students at WPI and other institutions that teach Mechanical Engineering, Electrical Engineering, and Robotics Engineering rely on using microcontrollers with sensors and actuators to complete projects for innovative and educational purposes. However, students without prior experience with circuit hardware may encounter challenges with wiring components, writing microcontroller code, and interpreting the results from the circuit. Troubleshooting processes can be not only time-consuming but also costly if the hardware is damaged due to incorrect connections. There are software programs and online services that can help, such as Wokwi (Wokwi, n.d.) and Fritzing (Fritzing, n.d.), but there is an absence of a tool that provides comprehensive solutions to all the problems mentioned above. That is where the Automated Design Tool for Arduino Circuits (ADArC) system fits in.

Previously, Cam Tu Le created the ADArC framework with automated circuit generation, simulation, code generation, and help modules (Le, 2021A). His thesis focused on the process of connecting components to an Arduino, leveraging a graph grammar tool called GraphSynth (DesignEngrLab, n.d.). Additionally, he wrote a dedicated support website (Le, 2021B) providing information on specific components. This thesis and some of its flaws are further discussed in Chapter 3.2. Despite the utility and educational value this framework could achieve, Le did not complete the user interface. Since the purpose of ADArC is to help students, we worked to create a user interface and incorporate all the proposed functionality of the existing framework.

The main goal of this project is to make an interactive tool that supports generating circuits based on the ADArC framework. Specifically, we are targeting users who are inexperienced with one or

more of these factors: coding, wiring, and Arduino circuits. We are paying special attention to make sure we serve the needs of students in the project-based engineering classes at WPI.

3. Literature Review

At the start of the project, we researched various topics to better understand how to achieve our project's vision and adequately address novice students' needs. In this chapter, we discuss Arduinos and their functionality, Cam Tu Le's MS Thesis that addresses how an application such as this would function, and also present details of GraphSynth rules from that thesis. Finally, we give a summary of the UI structure used by a similar project, Bond Graph Lab for Web, or BoGL Web for short (Vuolo, Misbach, and Earnest, 2023).

3.1 The Arduino

Arduino is a beginner-friendly microcontroller that is most coders and engineers' first encounter with the world of hands-on microcontroller projects. There are four main 'Families' of Arduino microcontrollers available for use: the Nano, MKR, Mega, and Classic Families (Arduino, 2022A). The Nano family microcontrollers are smaller and comparatively inexpensive ($13-$30, April 2024) that are built for home automation and wearable devices. Most of these devices have Bluetooth and Wi-Fi support as well as embedded sensors, such as for measuring temperature, humidity, and pressure (Arduino, 2022A). The MKR family is built for projects requiring complicated components such as radio, Wi-Fi, and Bluetooth (Arduino, 2022A). The Classic family contains the more popular boards, such as the Arduino Uno. These are usually the boards that classes select for projects. The Mega family is used for projects with more computing and power demands, which are satisfied with more General-Purpose Input/Output (GPIO) pins and a stronger microcontroller unit (MCU).

Because Arduino microcontrollers require code to run, they have a dedicated integrated development environment (IDE), normally running a C++ file known as a sketch, encoded as an '.ino' file. This file controls the way that the Arduino processes signals. The sketch file begins by having only two functions, *setup()* and *loop()*. *setup()* runs once on startup, while *loop()* runs continuously over the program's run time. This, combined with several preinstalled functions and features, is enough to run most programs for simple components like LEDs and buzzers. However, some advanced components, such as LCD screens and keypads, have additional code libraries to help process the input and output signals. These are additional C++ files that contain definitions of helpful objects and functions that can be installed using the IDE's library manager. They can then be used by including them in a file using an *#include* tag. Adding support for more components is not the only feature of libraries. Some, like the 'SD' library (Arduino, n.d.), provide additional functionality for user convenience, such as enabling reading and writing on SD cards. At the moment, Arduino keeps a record of over six thousand official and user-created libraries (Arduino, 2024).

3.2 Cam Tu Le's Thesis

Our work expands on the framework of modules developed by Cam Tu Le. His thesis (Le, 2021A) details the process of creating rules to describe how circuit components connect to an Arduino. Individually, the modules he created are suitable for inexperienced users, but they are difficult to use together. Despite this, the thesis's research into existing circuit wiring tools is invaluable. Le describes a system of four modules that comprise the basic functions of the application. The first is circuit generation or wiring assistance, where the program does most or all of the wiring work for the user. The

second module is circuit simulation, where the program emulates a real-world environment for the user. The third module is code generation, where the program stores suitable Arduino code snippets for components and displays this code when the component is used. The final module is user assistance or help, which serves as a built-in information repository to assist in searching for information about circuit components. In his report, Le also evaluates other circuit builders and simulators that attempt to serve the same purpose.

That report's evaluation of existing software is valuable information, but it covers only a narrow range of use cases, while we need different information. Hence, we also extensively evaluated existing software. Results of that investigation can be found in Chapter 6 of this paper. Le's report also describes the challenges that face a completed application, the target audience, and the circumstances motivating the need to develop such an app. However, these need some further consideration as we develop the user interface. Chapter 2.1 of Le's thesis explores a specific target class at WPI, ME3902, which includes the topic of Arduino circuit design. In Chapter 2.2, the report lists five main areas with which students struggled: integrating multiple components, programming, voltage regulation, sensor calibration, troubleshooting, and limitations in pins, memory, or processing.

There are a wide variety of circuit wiring assistants and simulators already developed. The thesis introduces and evaluates several simulators: CircuitO (CircuitO, n.d.), Wokwi (Wokwi, n.d.), Tinkercad (Autodesk, n.d.), PicSimLab (PicSimLab, n.d.), Proteus (Labcenter Electronics, n.d.), Fritzing (Fritzing, n.d.), and Virtual Breadboard (Virtual Breadboard, n.d.). From these, Le found that most of the existing applications are lacking in one or more ways. They either had little to no explanation about how to use most or all components, only allowed for connections and not simulations, or had

confusing coding functionality for the user (Le, 2021A). An exception to this rule is Wokwi, which not only allows custom wires but also custom code. While it does not do anything automatically, it is a trove of valuable resources, as the simulator is open source, allowing access to its component images, emulation software, and more.

Code generation is a very ambitious task for any application to have. The problem is twofold: understanding the user's intent and then assembling or creating correct code. Cam Tu Le's report barely details code generation at all, mentioning it only a few times, eventually stopping at providing a database of sample code as part of the Help Module. In general, code generation comes in two forms: AI-based and model-based. AI code generation, utilizing large language models like ChatGPT (OpenAI, 2021) or Google's Gemini (Google, 2023) can generate original source code based on a simple description of a problem. However, AI still has the problem of generating incorrect information despite how specific the prompt may be. Model-based code generation, in contrast, uses rules to precisely arrange code snippets together in a pre-arranged way. While this method is more certain, it is also more rigid and requires more rigorous descriptions of the desired program.

3.2.1 Graphs

The core of Cam Tu Le's thesis is a system designed to establish connections between components, such as sensors and actuators, and an Arduino. Sensors collect environmental data, such as temperature or light, and relay this information to the Arduino. Actuators, including motors and servos, then perform actions, often in response to the data received from the sensors. Le's connection system is largely based on the mathematical construct of graphs. In general, a graph is a series of nodes

connected by edges or arcs. In the report, Le uses graphs to represent circuit pieces and their pins. Each of the pins is represented by a node, and then they are all connected to a central object node using arcs. Using an extensive set of rules (83 rules in total by Cam Tu Le), the nodes representing components can be connected together correctly into one larger graph, again using arcs. The resulting graph represents a complete circuit and can be easily mapped into an image with components and wires.

Editing graphs is more difficult, especially when multiple nodes are destroyed and created at the same time. To do so, a replacement rule is used. A replacement rule details how to replace certain nodes, called the left-hand side, with other nodes, called the right-hand side. The sides may overlap, allowing the nodes in the middle to remain intact through the replacement.

The process of graph replacement is composed of three steps. First, find a replaceable section of the graph. Second, find the rule that is most fit to replace that part of the graph (if there are multiple rules). Third, execute the replacement and repeat. GraphSynth (DesignEngrLab, n.d.), used extensively by Cam Tu Le, is based on these constructs.

3.2.2 Code Libraries

Each of Le's modules (except code generation, which is not complete) is written in a different programming language, another complication to consider before moving forward with our project. GraphSynth, the backbone of the circuit wiring module, is written in C#, while GraphSynth's rules and rulesets are written in an XML variant. The circuit simulator is written in JavaScript and Typescript, using node.js and Electron (Le, 2021A) to create a downloadable executable. Finally, the help module is written in another XML variant, Darwin Information Typing Architecture (DITA) XML, this time

designed for web deployment. While these languages can be combined both by creating web apps with C# and by converting C# to Typescript, it is difficult and time-consuming to do and test.

Overall, this shows a need to choose a main language and support it, rather than spreading the application over so many unmaintainable segments. There are numerous libraries for JavaScript and Typescript. Of note are graph libraries, which provide functionality for creating and rendering graphs but do not support grammars. Libraries for graph drawing include Cytoscape.js (Franz et al., 2016), Graphology (Plique, 2023), and its extension, sigma.js (Jacomy et al., n.d.).

Another interesting library is AVR8js (Shaked, 2023), which allows Arduino Uno machine code to run in the browser. This can be extended to some other Arduino architectures.

Finally, other libraries, such as Wokwi Elements (Shaked, 2024), offer circuit element images, boasting an impressive collection of both premade and customizable SVG images. These could save a large amount of development time by avoiding the creation of many SVGs. Other more complicated libraries and frameworks such as node.js (OpenJS Foundation, n.d.a), Electron (OpenJS Foundation, n.d.b), React (Meta Platforms, n.d.), and ASP.NET (Microsoft, n.d.) could also be helpful in the development of the application. Node.js, Electron, and React all offer different ways to create and deploy JavaScript and TypeScript applications to the web and to standalone applications. ASP.NET is a framework that allows C# code to run in the browser.

Overall, the four modules of ADArC work together to provide a comprehensive tool for designing, simulating, and generating code for mechatronic systems. Cam Tu Le designed ADArC to be user-friendly and accessible to students and researchers with little or no experience in mechatronic systems design (Le, 2021A). This project shares this goal.

3.3 GraphSynth

Chapter 2.4 of Le's thesis specifically discusses GraphSynth, an open-source software tool developed by researchers at The University of Texas at Austin. (Le, 2021A; DesignEngrLab, n.d.) GraphSynth is a tool written in C# that allows users to store rules and graphs in a standard XML framework. It is designed to simplify the process of creating and testing graph transformation rules, which are a set of instructions that develop graphs. Graphs can be used to define how sensors and actuators can be connected to an Arduino microcontroller. In this section, we will introduce GraphSynth's "designGraph" structure and the three important graph elements: nodes, arcs, and labels. As it was introduced in section 3.2.1, we will further explain the rules and rule sets, as well as how they are used in the 'recognize and apply' process with an example.

The "designGraph" is a data structure that represents a graph. It stores information about nodes, connections between nodes (known as arcs), and any associated labels (as shown in Figure 3.3.1). This structure serves as the platform where transformations to the graph take place.

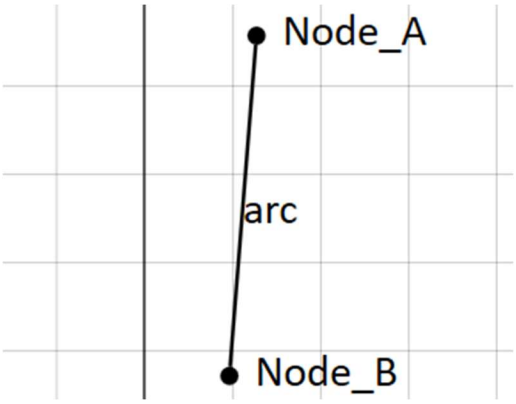


Figure 3.3.1: Nodes, Arcs, and Labels

Nodes, as the name implies, represent points in the graph. They can be added and removed as desired. A node can stand by itself, or it can be connected to an arc or arcs. Functions are provided to find, if any, the connected arcs of a node as well as the nodes at the other end of those arcs. A node can have multiple labels as its identifiers.

Arcs connect nodes in a graph. They can be added or removed. Each arc has a "from" node and a "to" node at its ends. This structure allows nodes to be located via arcs. Like nodes, arcs can have multiple labels as identifiers.

Labels are a property of nodes and arcs. They are often treated as identifiers to find specific nodes and arcs. As a data structure, they are simply strings of text. Multiple labels can exist on a node or an arc. Each of them can be an identifier. Selecting multiple of them at the same time can also create a new identifier, which is not used in our application. It is possible to have repeated labels.

The primary function of GraphSynth is a process called ‘recognize and apply.’ This process takes place with a specified set of rules. While recognizing, GraphSynth looks at the rules in the rule set and a graph to find matching patterns. After matches between certain rules and the graph are found, GraphSynth applies those rules to perform graph replacement, changing the graph that the rules act on. The process repeats until there is no action of graph replacement to perform. Since the process repeats, it is crucial to make sure that there are fewer matches after each repetition. The reducing of matches is realized through a feature in the rules named ‘negation’ It will ignore the nodes with a specified label. The nodes with the label will not be recognized during the next repetition of the process.

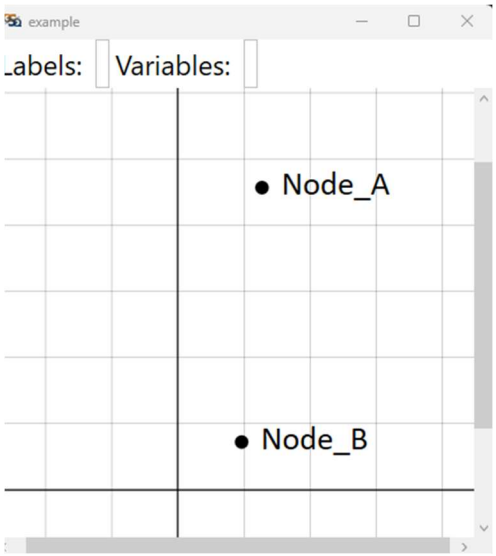


Figure 3.3.2: Seed Graph; the Starting Graph

For example, two nodes labeled “Node_A” and “Node_B” are put in the seed graph as shown in Figure 3.3.2. The rule set contains one rule. As shown in Figure 3.3.3, the left side of the rule defines the pattern to be recognized with the presence of the two nodes. It is also specified on both nodes that label “connected” is to be negated. This allows the system to detect whether the nodes are already connected. If not, the right side adds the label “connected” to both nodes and creates an arc that connects the two nodes. After the ‘recognize and apply’ process is done, the right side of the rule has replaced the patterns on the left side. Figure 3.3.4 shows the result of applying the example rule on the seed graph.

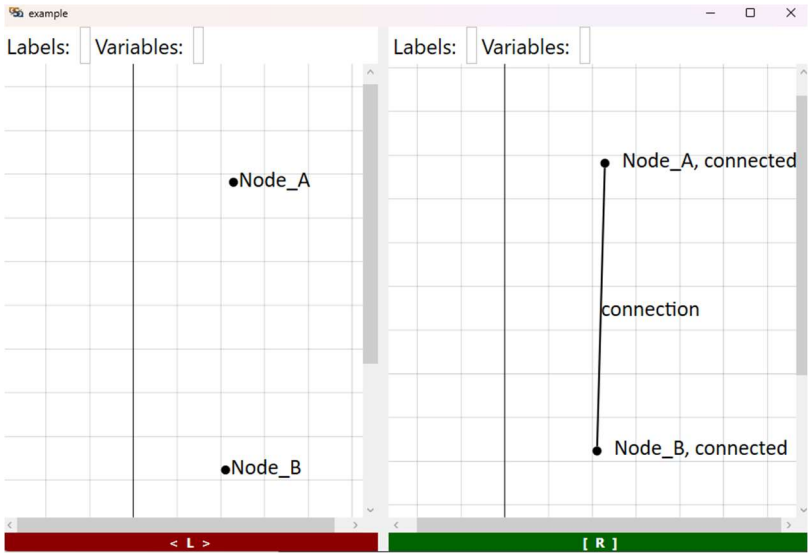


Figure 3.3.3: Rule *example.grxml*

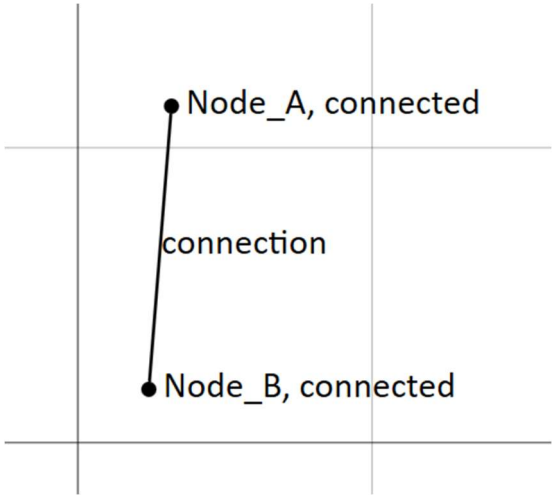


Figure 3.3.4: Result Graph After Rule Application

Thanks to the work done by Cam Tu Le, we have 86 rules at the time of writing. They are divided into two groups with special naming conventions. The rules that start with "add_" refers to the rules that add a component to the circuit; the other rules are connection rules which generate connections between components. These rules were later edited, as described in Section 6.3.1 and discussed in Section 7.3.

3.4 BoGL MQP Summary

BoGL Web: An Application for Generating Bond Graphs on the Web was a Major Qualifying Project (MQP) done during the 2022-2023 school year (Vuolo et al., 2023). Its goal was to design and implement a web-based software application to graphically construct system diagrams and bond graphs. BoGL Web used GraphSynth to create bond graphs, similar to how we will use it to wire circuits. We met with them early on in our project's development to learn about their implementation. This interview, their Literature Review, and their Methodology gave us a good basis for how we wanted to prototype the UI and what coding languages to use for the project. Their implementation chapter also gave us a better understanding of how to create a more user-friendly UI, as well as some key implementation considerations for any front-end or back-end components we developed. These were actively considered when trying to meet our goals and requirements, discussed in the following chapter.

4. Goals & Requirements

Our target audience for this project includes not only students enrolled in ME 3902: Project-Based Engineering Experimentation, which teaches basic usage of Arduino systems, but also those engaged in Major Qualifying Projects (MQPs) and courses like ME 4320: Advanced Engineering Design. These latter experiences require students to integrate multiple sensors and actuators, addressing the complexities involved in such configurations. ME 3902 introduces the Arduino microcontroller and Raspberry Pi microprocessor, so students are expected to have a rudimentary understanding of ECE or CS principles, but no experience with microcontrollers. This forms the profile of the target student or user. They are not experienced with the Arduino, and may have trouble with wiring or coding it, but have some applicable experience in ECE and coding. With this in mind, we were able to devise some goals for this project that could be developed, especially keeping in mind Cam Tu Le's work.

First, the developed application should have all the modules suggested in Le's thesis, easily accessible or distributable. The existing modules were difficult to access and were spread across three applications (web, GraphSynth, and custom). This was unacceptable for the user experience. Thus, the modules must also be unified into one application that can be accessed easily by anyone with a computer. The easiest way to achieve this was to develop an application for the web. Websites such as Wokwi and CircuitO are easy to find and remember. In addition, developing for the web allows the option of using libraries such as node.js, which allow JavaScript or Typescript code to be packaged into an executable download.

Second, the application should seamlessly align with users' intuitive expectations for this type of software, ensuring a user-friendly experience. We targeted users with low experience in wiring, unknown experience in coding, and minimal experience with Arduino microcontrollers. All of these fields had to be as easy as possible to work with for the application to function as intended. Ideally, most of the process should be automated, relying on the user only to select the parts they want or simply enter the intent of the circuit. An additional important part of this was to provide contextual help and suggestions, not only for the circuit but also for the application itself.

To achieve these goals, we also developed several requirements, as seen in Table 4.1. In the table, a "User test" involves evaluation based on surveying prospective users. "Unit tests" refer to manual or automated functionality testing completed prior to user tests.

Table 4.1: Requirements and Testing

Requirement	**Testing**
Create a circuit from individual parts automatically	User test
Generate code to run on circuits automatically	Unit tests & user tests
Simulate circuits in real-time	Unit tests & user tests
Access help module	User tests
Respond appropriately to screen size changes	Unit tests & user tests
Share with other users	Unit tests & user tests

Regarding the requirements listed above, they refer to the main four-module system developed by Cam Tu Le, which includes circuit generation, code generation, circuit simulation, and help. The

first three requirements were essential components of the system, as they automated the process of circuit generation, code generation, and circuit simulation. The help module was also crucial for guiding users through the process.

The last two requirements, responding appropriately to screen size changes and sharing with other users, were new to the application version of the project. These requirements were important for ensuring that the application was user-friendly and accessible to a wide range of users. Sharing a project with other users should be as simple as entering a new URL into the browser's bar, though this functionality should also be accompanied by the option to download the project as a file. If time allows, sharing via image may be a possibility, for example by exporting as a PNG file.

Our vision for ADArC was that a novice user can construct and simulate a circuit from only a list of parts. While the code generated should be enough to run various parts of the circuit, more advanced users may need to modify the code to suit their specific requirements.

Building upon the goals and requirements outlined, the next chapter delves into the methodology employed in this project, examining a range of related projects and tools that have shaped our approach and informed our development process.

5. Methodology

In the beginning of this project, we reviewed related projects and tools in detail. This includes Cam Tu Le's thesis (Le, 2021A), BoGL Web's report (Vuolo et al., 2023), and several circuit generators with different specialties. Each of these was split into a separate research topic and further branched out from there.

From Cam Tu Le's work, we identified the strengths and weaknesses of each of the modules that he developed (explained in Section 3.2), and additionally made notes of the various libraries, tools, and processes that he used to create his project. From BoGL Web's work (explained in Section 3.4), we learned a lot more about the technical aspects of developing the project, from the languages and libraries that they used, to the development process. From the circuit generators discussed in chapter 2 of Le's thesis, we were able to identify what was necessary for the UI, as well as what worked well and what did not. Chapter 6.1 of this paper describes the results of this analysis. One of the circuit generators, Wokwi (Shaked et al., 2024), also provided several open-source tools related to Arduino that proved invaluable to our project.

With the initial research completed, we decided to focus on identifying key features of the final project. We decided to adopt the same features identified by Cam Tu Le in his report: circuit wiring, circuit simulation, code generation, and documentation. Additionally, we decided that the resulting application should be a website to allow the greatest number of users to access it. These main features guided future research, ideas, and problem-solving.

5.1 Solutions for Wiring and Representing Circuits

The first problem to be solved was how to create a GUI displaying the circuit component connections. This was done previously by Cam Tu Le using GraphSynth (DesignEngrLab, n.d.) and a selection of rules to connect components. We further evaluated these rules to verify their correctness and completeness. Those rules cover how to connect components, but the rule application process results in a rather complex graph and an unorganized data structure that is not fit to show to users, especially novice users.

Continuing with our goal of creating our app to be web-based, we found the Wokwi Elements library of web components to be a good way to represent the circuit to the user, and it was convenient for us to use. The library uses WebComponents to show the circuit components as HTML components. The Wokwi simulator uses this method, though it is somewhat unique for doing so. Most other simulators use a 'canvas' HTML element, on which the components are drawn. We decided against using a canvas due to the availability of Wokwi Elements, which drastically reduces the number of components we had to individually create.

We also wanted to be able to share completed circuits with other users, for which a variety of solutions were considered, depending on the type of sharing. We wanted to be able to share them using a URL so users can continue working on their circuits or share them with others with minimal effort. We also considered image sharing, though that was considered significantly less important, as computers can take screenshots that provide the same functionality.

Based on the solutions, we first edited and tested the GraphSynth rules (section 6.3). We then converted Wokwi components to Razor components (section 6.4) as Wokwi provides components written in Typescript, while we needed them in C#. Finally, we drew the connection lines based on pin positions (section 6.3.1, section 7.4).

5.2 Solutions for Circuit Simulation

We also wanted to be able to fully simulate and test the generated circuits in the browser. In our research, we found that Wokwi, one of the examined circuit simulators, had this functionality. Upon inspection, we found that they used an open-source library called AVR8js, which is a JavaScript library that emulates certain Arduino boards. We decided to use this to handle our simulations.

One problem with AVR8js was that it only accepted compiled code. However, within the provided examples was a solution (Shaked, n.d.). The way the example compiled the code was to send it to a Wokwi server (hexi.wokwi.com) using an HTTP request, and the server would return the compiled code. In the example, the code was returned as a hex string, which was then converted to a byte array to be emulated with AVR8js.

Another problem was that the emulator's only function is to change the output state of a set of pin variables. To properly update the visuals, there needed to be code to listen for changes to the pins and translate those into visual effects on-screen. To solve this, we created a translation function for each component, intended to translate pin data into visual output or response data. We decided to add an event to trigger when a component is introduced into the primary circuit, which enables the translation function for that component.

The last problem with AVR8js to solve was the handling of Serial output. Physical Arduino boards would output messages to the console of the Arduino IDE to be displayed. Similarly, AVR8js would also output the messages. Therefore, we need to provide a console that can display those messages. This console could be accomplished with a simple text box in most cases, but Serial input and signal output were a blocking factor. We decided that these were not essential features to the initial design, but could be added later, especially the input.

Eventually, we successfully utilized AVR8js and provided features for compiling and executing the code. We also managed to capture the compile messages and the serial outputs and displayed them in the console. Details about the console can be found briefly in section 6.1.3. The remaining problem was the translation functions mentioned above. We encountered problems with timing and single-threading during the development process. We made some failed attempts to solve the problems, and they were later resolved using the timing-packet solution. The problems and the solution are discussed in section 7.5.1.

5.3 Solutions for Code Generation

This is the least developed module of the original thesis report, and thus the hardest to find solutions for. After some research on the topic, we had two options: the first is to use generative AI, and the second is to design a system to neatly order premade code blocks. We decided to do the second, as generative AI is variable and often wrong and we needed certainty. We did not find a library to do this for us, so we decided that a custom system specifically for our needs would be more convenient.

Additionally, we decided to add a code editor, as being able to see and edit the code is essential for the proper function of the simulation. We found several solutions, namely Ace Editor, CodeJar, Monaco Editor, and CodeMirror (Ace (Ajax.org Cloud9 Editor), n.d.; Medv, n.d.; Microsoft, n.d.; CodeMirror, n.d.). The details can be seen in Table 5.3.1.

Table 5.3.1: Code Editors Comparison

Editor	**Pros**	**Cons**
Ace	• Multicursor • Easy to integrate into sites • Many languages • Some mobile support	• Good docs, little support beyond that
CodeJar	• Small (2kb)	• Poor mobile support • Very few features • Possibly no longer available
Monaco	• Microsoft developed, better support • Basically the same as Visual Studio (VS) Code • Autocomplete and validation, custom available • Multicursor	• No mobile support
CodeMirror	• Mobile Support • Multicursor • Autocomplete and validation, custom available • Custom key binds	• Tab not bound by default

We eventually decided that Monaco editor would be the best choice due to its developer being Microsoft, guaranteeing good support. It also is extremely similar to VS Code (a Microsoft source-code editor), ensuring some familiarity for users. We decided that mobile support was not a requirement, so

the one downside could be ignored. In the case that Monaco was not compatible with the rest of the environment, Ace editor would also be acceptable.

We deployed our code editor using the Monaco editor and added starter code as shown in Figure 5.3.1 to resemble the coding experience with the Arduino IDE. We then used the sample codes from the help website, one of Cam Tu Le's modules, and broke them into snippets. The detailed process of code generation will be explained in section 7.3.

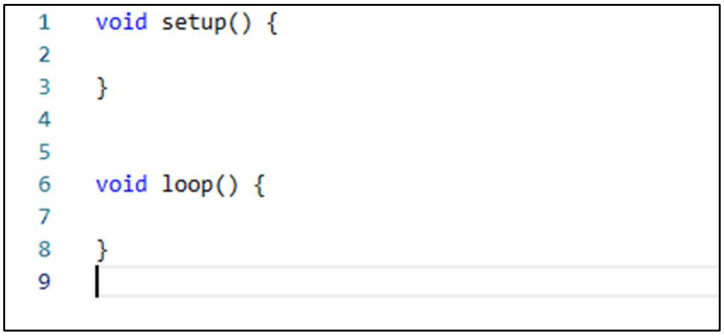


Figure 5.3.1: Default Code Editor

5.4 Solutions for Documentation

As this was one of the most developed sections from the original thesis, this was relatively easy to find a solution for. We decided to just use the previous documentation/help system, pulling the pages into our project and displaying them differently. We did have options regarding how we wanted the display to be accessed, with some advantages and disadvantages discussed in Table 5.4.1.

Table 5.4.1: Documentation Access Points Pros & Cons

Option	**Pros**	**Cons**
Tooltip hover	• Obvious	• UI Clutter
Toolbar button/sub-menu button to external	• Easiest to create	• Hidden, likely never seen • Not context sensitive
Tooltip link/button to external	• Context-sensitive	• Clickable tooltips are known to get in the way of important information
Tooltip button click to popup	• Context-sensitive	• May require reformatting of the help module

Generally, this can be resolved down to two sets of two options: static or context-sensitive and new page or popup. Context-sensitive input means that the output would change based on differences in the input. For us, that meant that context-sensitive help would direct a user to help for the component they clicked on rather than generic help. For the first option, we decided that context-sensitive input was better, eliminating the searching that a user would have to do otherwise. We did, however, decide to also add a separate toolbar button to take a user to the established help module. Context-sensitive input would be most easily achieved with a tooltip, which could appear on hover or on click. We decided that clicking would be better, as the hover method is significantly more likely to lead to visual cluttering. The last choice was between a popup help menu, or simply redirecting to the existing page. We settled on a popup being better, as repeatedly opening tabs that the user needs to close is not friendly to the user. This did lead to concerns about loading all of the popup pages, but we decided that load times were not a huge concern past the initial page loading as most elements remained static after loading the page.

5.5 Programming Languages

With the main components decided, we needed to select a programming language to work with. We were able to narrow the field down to JavaScript, Typescript, and Blazor (C#) thanks to the work done by the BoGL team (Vuolo et al., 2023). We settled on Blazor after deciding that GraphSynth was a requirement. GraphSynth is written in C#, so having good compatibility was the deciding factor. Blazor also had some interoperability features to interface with JavaScript so that other libraries could be integrated into the project (Spasojević, 2022). One potential problem was that Microsoft recommends not letting JavaScript functions affect the DOM, though the extent of the problems is not specified (Microsoft, 2024). This was not an issue in the end.

Ultimately, we wanted to produce a static webpage, meaning that it had defined code that would initially appear the same to every user. By using Blazor WebAssembly, we could avoid needing a server-side of the application at all, saving us from having to code the server in addition to the client. Hosting static web pages is also easier, with many different free hosting services with a variety of features, with a selection of them shown in Table 5.5.1.

Table 5.5.1: Web Hosting Pros/Cons

Site	**Pros**	**Cons**	**Link**
GitHub pages	• Custom URL • Easy updating • No cost	• Soft bandwidth limit	https://pages.github.com/
Cloudflare pages	• GitHub integration for easy updates • No bandwidth limit	• 500 build per month limit	https://pages.cloudflare.com/
Netlify	• Many templates • Feels very all-in-one, helpful tools	• 300 build minute limit (per month) • Many limits	https://www.netlify.com/
Firebase	• Database availability • Performance monitoring/analytics	• Data transfer limit (360 MB/day) • Database not scalable	https://firebase.google.com/
Amazon S3/AWS	• Update from Git • Uptime guarantees	• Only free for 1 year • Various other limits	https://aws.amazon.com/amplify/hosting/
Forge	• Update from Git	• Small bandwidth cap (1gb)	https://getforge.com/
Back4App	• Collaborative development	• Small size limit (250mb)	https://www.back4app.com/
Vercel	• Update from Git • Previews from Git • Collab UI review	• Collaboration not free	https://vercel.com/
Gitlab Pages	• Better for private repos	• Limit on compute minutes each month	https://about.gitlab.com/
Surge.sh	• Update from Git • No limits • Clean URLs	• Mainly command line based, more complicated	https://surge.sh/
Render	• Update from Git • Integrated database	• Bandwidth and build minute limits	https://render.com/

We decided that cost was the most important consideration, followed by ease of use and availability of collaborative features. We were unsure of the impact bandwidth limits or compute minutes would have on the deployed application. We decided to avoid those restrictions to ensure continued deployment of the application. Under these considerations, the best option was GitHub Pages, due to its ease of use and notably low-cost, fee-free payment model. While others such as Surge.sh also had the same or similar features, familiarity was the deciding factor. One of us had used GitHub Pages in the past and was comfortable setting it up. Details on deployment can be found in Chapter 7.11.

5.6 Components List

With the initial research and conceptual design established, we began to delve deeper into many of the topics of interest in each of the fields. The first was the identification and categorization of the circuit components presented by Cam Tu Le's thesis report, a requirement for both the wiring and simulation modules. We wanted to identify components that were essential to include in our system, as well as identify components with both previously designed rules as well as images provided by the Wokwi Elements library.

By going through the Help module and Cam Tu Le's case studies, we created a list to determine for which components we needed to create GraphSynth rules and Razor components. This list could be updated in the future by investigating other existing projects for more components. One way we determined whether to include a component was the viability of generalizing a unique connection rule.

For this reason, we excluded motor shields and basic components such as single resistors, transistors, capacitors, and inductors, as these could not be guaranteed to be added in the correct positions.

At the time of writing, our list contains more than seventy components and can be found in Appendix D. We realized that the number of components is high enough to confuse the users when making a selection. Therefore, the categorization of components is necessary. In support of the design of the components selection panel that will be discussed in section 6.1.2, we decided to make categories and subcategories such that there will be no more than ten components under each lowest category. With deliberate consideration of the requirement and the guidance of the help module, we decided on the categorization shown below in Figure 5.6.1. In the end, time constraints limited the number of completed components to twelve, and thus how many of these categories appeared in the final product.

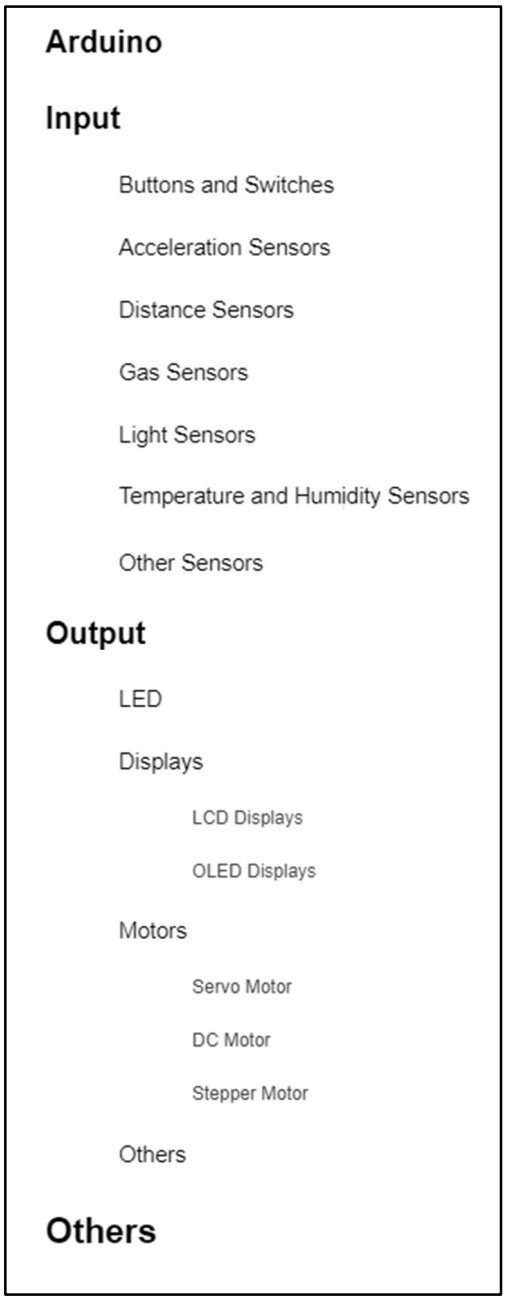


Figure 5.6.1: Planned Categorization of Components

In this chapter we discussed our research into addressing the goals and requirements from the previous chapter, as well as backend solutions for building ADArC. The next chapter covers the design of ADArC.

6. ADArC Design

This chapter addresses all of the considerations made when designing the UI and UX, from the prototypes to the final versions.

6.1 UI Design

To create a UI that is easy for the user to understand and that can support the four core modules of the ADArC, we compared other popular circuit simulators' UIs with each other to form a list of traits we needed for ours. Table 6.1.1 below shows the results of this evaluation. Almost every evaluated tool separated their interfaces into different areas, usually by using physical separators such as button-controlled tabs. Almost all had a primary control toolbar. While all tools had ways to select components, we found that tools with draggable images were the most informative and felt the most intuitive. Following those features, of the tools with coding functionality, most included a form of rudimentary IDE isolated from the rest of the application. We aimed to take the most effective features from each of the other tools, producing five key requirements for our UI mockups moving forward: a Toolbar, Components Window, Simulation Window, Code Window, and Help Window.

Table 6.1.1: Circuit Simulator Comparisons

	Sensor location	**UI organization**	**Drag and drop**	**Immediate simulation updates**	**Code organization, editing, compiling and executing**	**Where is it saved?**	**Context-sensitive help? Tutorial?**
CircuitO	Left panel	In tabs (cannot have two things on the screen)	Yes, but it organizes the parts for you, you can't move them around	Web components	Built-in editor, no compiling and executing	Online, sharing link available	No context-sensitive help. Has a tutorial/guide line before entering the web.
SimulIDE	Left panel	Left: components; Central: circuit canvas; Right: code editor/compiler/debugger	N/A	N/A	N/A	N/A	N/A
PicSimLab	Tabs, no preview, sorted by IO	In tabs	No	Canvas for online version	No editing, can execute precompiled code	User computer, no linking	No, has help docs, not context sensitive. No tutorial
Wokwi	+ button on the canvas	In tabs (Can add and create more files in tabs)	No	Uses web components	Online editing, compiling, running	Can be saved onto online account or downloaded onto device	Help is for individual parts. Not context-based. No tutorial.
Tinkercad	Right panel	Circuit + component panel default, buttons -> panels otherwise	Yes, or click	Uses canvas	Yes, block-based with option for C++, no generation	Online server, option for sharing link	No help, no tutorial
Fritzing	Right panel	Circuit and component windows default, parts separated into tabs	Yes, or click	Yes, download	No, no editor, no compiling and executing	User computer, no/difficult sharing	No help, no tutorial

6.1.1 Prototype

Figure 6.1.1.1 shows an early prototype with rudimentary buttons for each of the essential four modules: circuit generation, circuit simulation, code generation, and help tab. Details of this protype are discussed after the image.

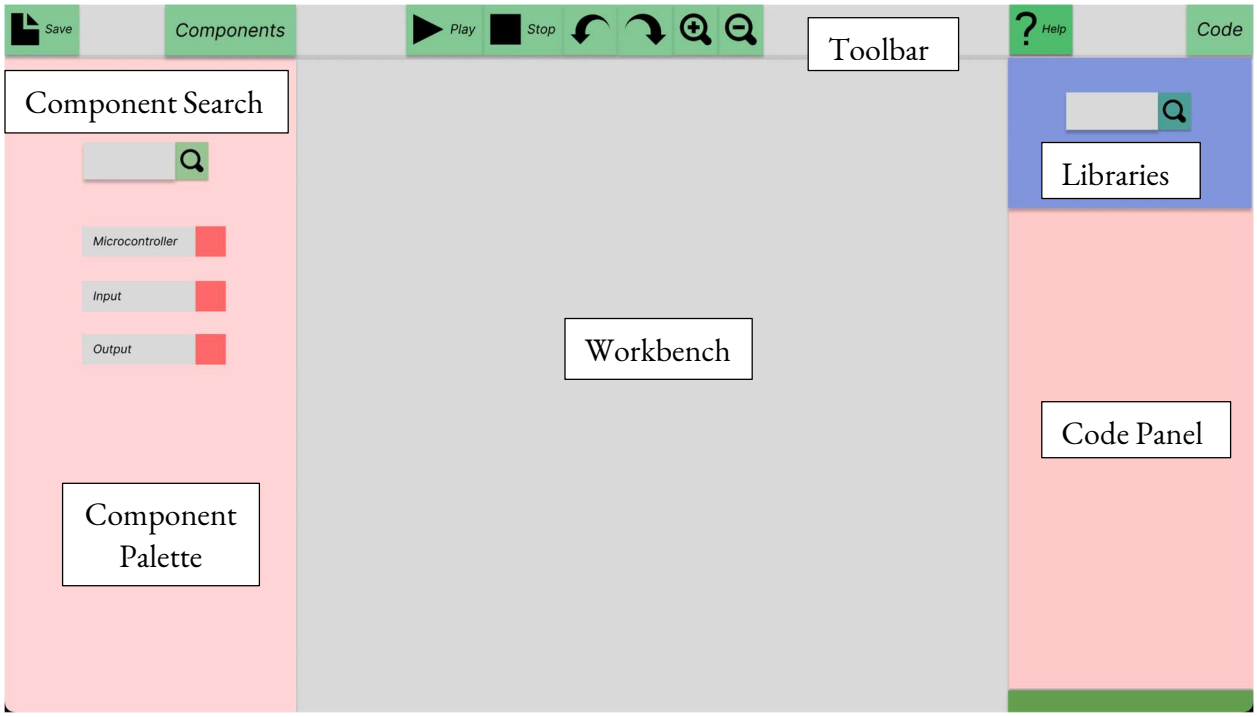


Figure 6.1.1.1: First UI Prototype Mockup (Tested Using Figma)

To create this prototype, we used Figma (Figma, n.d.), which is widely known for creating and testing website design before coding it. We tested how the workflow would look, which is when we realized that the components selection panel (Component Palette) should appear on the left since our brains mainly read information from left to right.

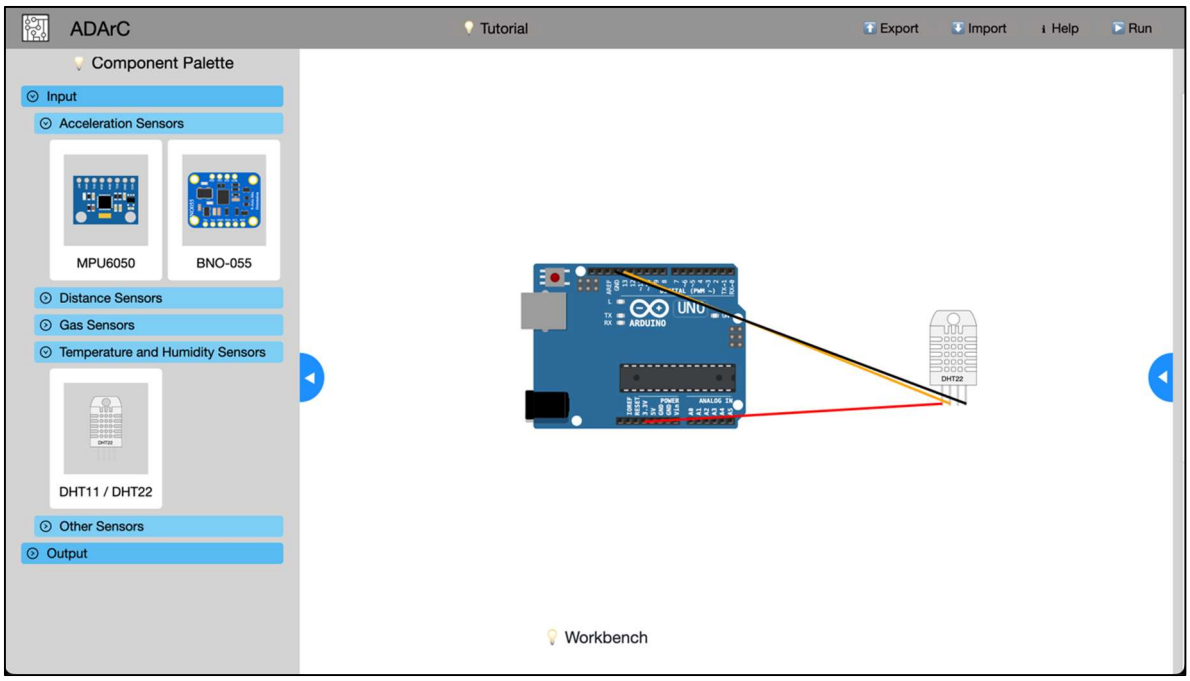
The Component Palette will be a scrollable and categorized list for users to select their desired circuit components.

The circuit image will be in the Workbench in the middle of the screen. The components and code tabs can be minimized (the components on the left and the code on the right). New connections will appear in this area as the user adds new components.

The code panel will be on the right side. It will generate and display code on the upper section according to the components selected. The lower section is where the console will be. It will update every time the user runs the simulation. Note that the upper and lower sections had not been created for this prototype. At the top of the Code Panel, you can search for libraries.

Above the simulation and these side panels, the toolbar will always be visible, and it will allow users to save the current circuit, zoom in or out, undo and redo, play and stop the simulation, and open the help module on an external site.

This UI prototype took multiple iterations to get to the current version, shown below in Figure 6.1.1.2. It takes into consideration research of the aforementioned circuit simulations and Le's insights (Le, 2021A). Some features were left out due to time constraints, like the search bar for libraries in the Code Panel and for components in the Palette. Over time, we moved the expand/minimize buttons to the sides of the panels, reordered the buttons so the Run button is near the Code panel, formed a coherent color palette, and made the panels bigger.



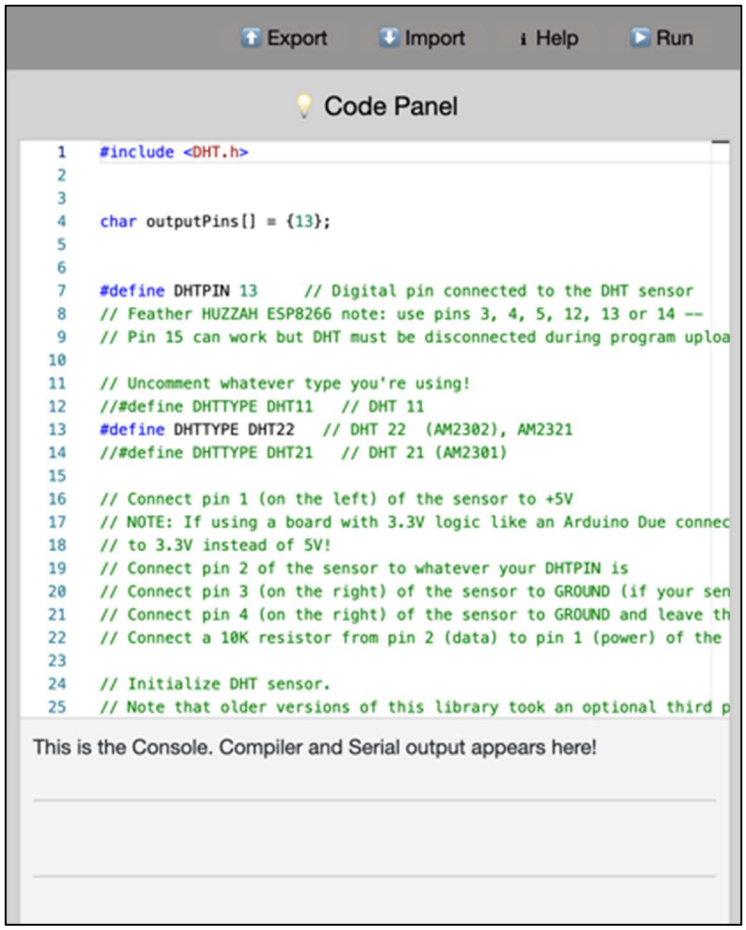


Figure 6.1.1.2: Final UI Design (Code panel shown separately).

In Figure 6.1.1.2, the panel on the left is called the Component Palette, where users can find components organized categorically and drag them to the Workbench, which is the white space in the middle where the circuit creation and simulation take place. In the figure, the Workbench contains an Arduino and a DHT sensor. The Code Panel is hidden on the right since only one panel can be opened at a time. The Code Panel contains the code for the components in the Workbench. It can be compiled and run using the 'Run' button located on the far right of the Toolbar.

6.1.2 Component Palette

One of the most important features that we knew we needed was a way to find and add circuit components to the Workbench. We decided to call it the Component Palette. We needed to display the type of component, a preview image, and its name. We planned to include extra information such as functionality that would appear when hovering over parts. This feature was later included as an option inside the menu that appears when right-clicking components in the Component Palette or Workbench. For the exact implementation, see Figure 7.1.3. We initially created a low-fidelity prototype of this feature, shown below in Figure 6.1.2.1.

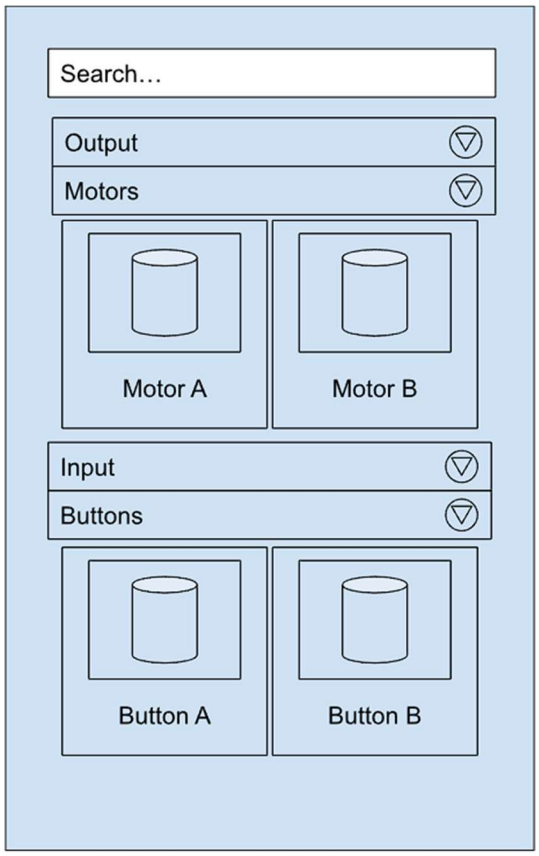


Figure 6.1.2.1: Component Selection Panel Mockup

The early prototype used a system of nested dropdown menus that reveal components at the lowest level. These components are represented as cards, each of which has a name and a preview image. The cards can be dragged and dropped into the simulation area to add the represented component to the circuit. This prototype included a search bar, though that was later scrapped due to disagreements over what to search for. Initially, we also had three components per row, but it became too cluttered, and the text was harder to read, so we decided to organize components in rows of two, in which every sub-category is indented to indicate a hierarchy, as seen in our final design shown in Figure 6.1.2.2. When components are dragged onto the Workbench, they automatically connect to the Arduino. Using a drag and drop system allows all changes to be represented visually instantly. We previously thought of

having a ‘Generate’ button to press after adding all components to the workbench, but that would not let the user observe how those parts are being connected one by one.

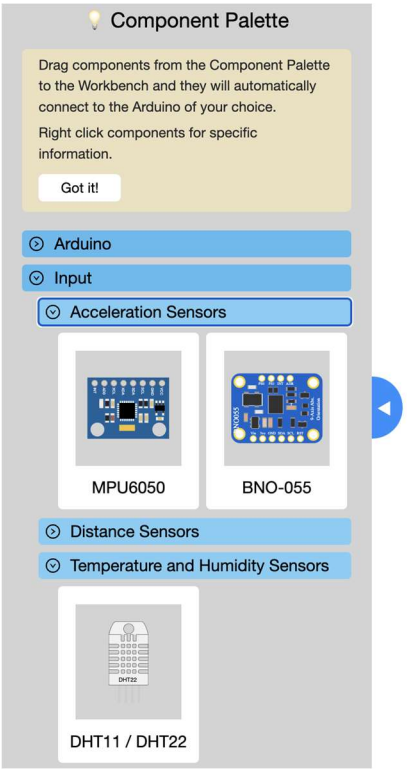


Figure 6.1.2.2: Final Component Selection Panel

6.1.3 Code Panel

Once users have chosen the majority, if not all, of the components, their primary focus is likely to shift toward simulating the code and observing the circuit's behavior. To enhance this experience, we incorporated arrow buttons adjacent to each panel, allowing users to effortlessly minimize or expand them. These buttons were a vibrant blue with an arrow that indicated in which direction the panels would move. Examples can be seen at the far right of Figure 6.1.2.2 and the far left of Figure 6.1.3.1.

Additionally, if one panel is open and an attempt is made to expand another, the currently expanded one automatically minimizes, ensuring a seamless and intuitive interface that directs the user's attention to the most important task at that moment. This also allows the Workbench to always be the main focus, so users can see it change when adding a new component or running the code.

The design of the code panel essentially never changed during the project. We wanted it to be as simple as possible. Our goal was to make it resemble other code editors, thereby increasing the user's sense of familiarity with the feature. The console, however, had several versions, mostly focused on error handling. Originally, all text was black, but in the final design, errors were printed in red and were added to the Code Panel as underlines, shown in line 9 of the code shown on the right of Figure 6.1.3.1. Figure 6.1.3.1 shows the final design, including an error message.

Figure 6.1.3.1: Final Code Panel (Example with an error on the right)

6.1.4 Toolbar

The Toolbar design underwent many iterations. We eventually decided that the must-haves are the 'ADArC' text and logo, the Import and Export buttons, the Help button, and the Run button. We

linked Le's help module to the Help button. When clicking the Run button, it changes to 'Compiling...' and then to 'Stop', as seen on Figure 6.1.4.1. At one point, we had zoom in and out buttons, but zoom as a feature was unexpectedly difficult to implement. Sometimes components would change in size and wires would move out of place, so this feature as a whole was not developed.

The Export button will provide the user with an '.adarc' file that will save their current circuit and code, it can then be imported using the Import button so they can continue working from where they left off.

If the user has a smaller screen or makes their window smaller, the toolbar adapts to that change in size, up to a window size of about 500 pixels wide. Less than that, the application cannot fit the buttons anymore without making them too small to read, so it becomes scrollable from side to side. Figure 6.1.4.1 shows the final design.

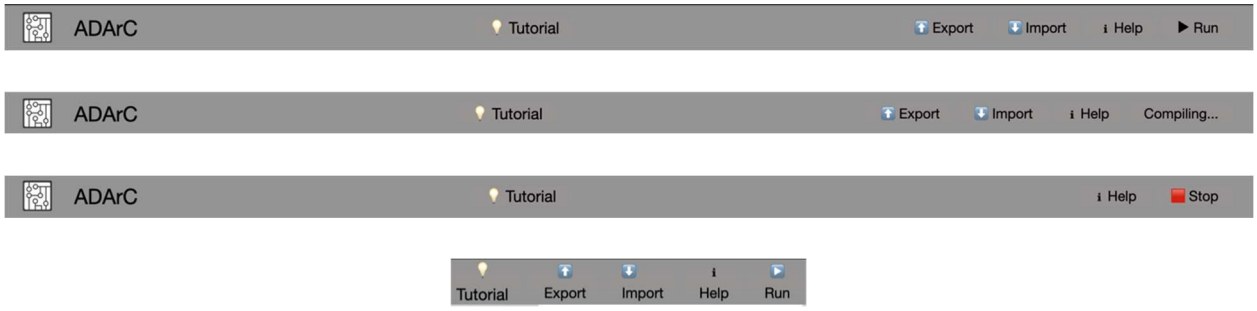


Figure 6.1.4.1: Toolbar (idle, compiling, running, small window, in order)

6.1.5 Help Design

To help ease novice users to our application, we implemented two ways for the user to learn information about the system. The first is a tutorial that covers each step of the design process of ADArC. This was implemented using IntroJS, a JavaScript library containing tools for building user onboarding tours (Mehrabani, 2023). The design of the tutorial can be seen in Figure 6.1.5.1. Our team used IntroJS because it allowed for a simple way to highlight important parts of the projects, allowing users to quickly familiarize themselves the basics of each part of the application, while also showing the ideal process of making an Arduino project using ADArC, as described in Chapter 6.2. The exact content of the tutorial can be found in Appendix E.

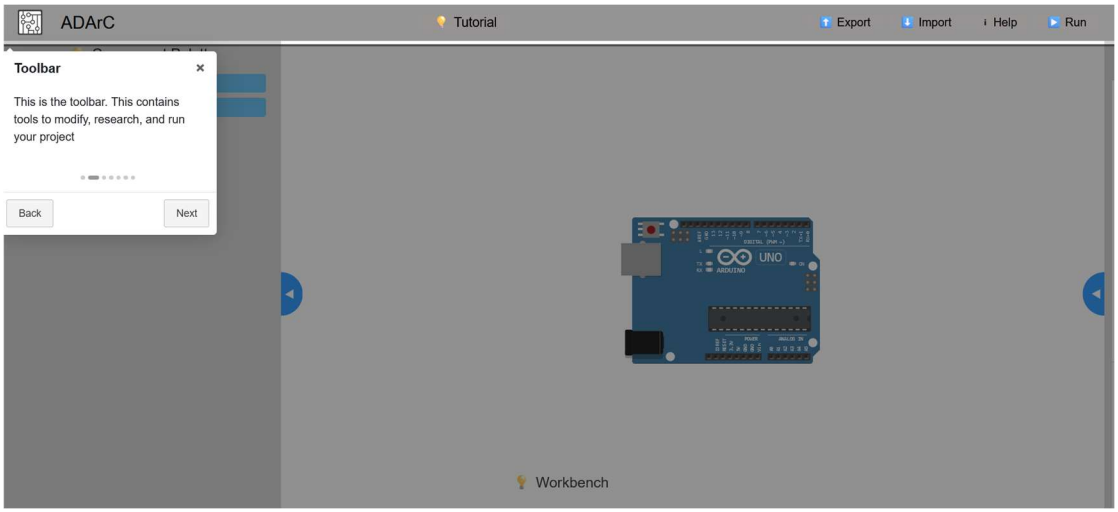


Figure 6.1.5.1: IntroJS highlighting the Toolbar

In case the user does not want to go through the complete tutorial again, we added Tooltips across the app on each major section. They can be accessed by clicking the light bulb icons. Upon doing

so, a light-yellow box appears that explains the functionality of the respective section. An example is shown below in Figure 6.1.5.2 these are displayed on the Component Palette and on the Workbench.

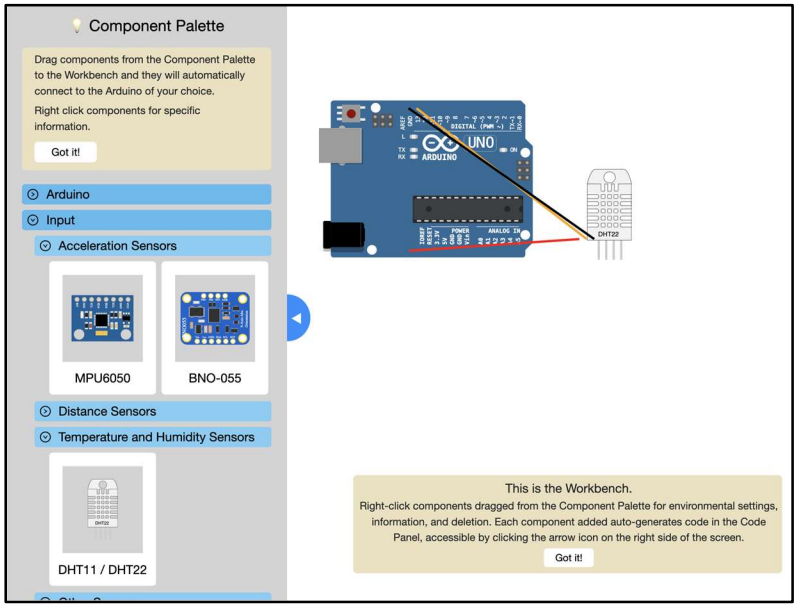


Figure 6.1.5.2: Tooltips are shown in yellow boxes

6.2 Intended User Interaction Order

As additional components are added to the Workbench by the user, the corresponding code with some basic functionality will automatically populate in the Code Panel. Clicking the Play button on the toolbar runs the simulation. Serial outputs and errors will appear on the Console at the bottom of the Code Panel.

There is context-sensitive information provided when hovering over individual components. This allows users to easily receive important information about components without having to visit the

full documentation on an external link. The information ranges from general specifications, variations of the same component, and related links (such as parent topics).

6.3 GraphSynth Rules Edits

GraphSynth rules, as introduced in section 3.3, are the essential part of GraphSynth that enables the functionality of circuit generation for our project. Building upon the foundation of Cam Tu Le's work (Le, 2021A), we have made strategic edits and modifications to his GraphSynth rules to align with our goals and requirements. In this section, we explain the reasoning behind those edits and modifications.

6.3.1 Connection Lines and Color Coding

Our first issue was distinguishing inter-component arcs from component-pin arcs. Our solution was to add a label called "connection" to inter-component arcs, denoting them as component connections. This is illustrated in the grammar rule in Figure 6.3.1.1, highlighted with red underlines. The second issue was distinguishing the wires from each other in the final UI, further increasing real-world parity. To solve the problem, we came up with a system of color codes (seen below in Table 6.3.1.1) and added one of these codes as a label to each component connection. This feature is also shown in Figure 6.3.1.1, marked with blue underline. Aside from the convenience it brings to our development, it is also a good practice to have a consistent color standard for wires to avoid faulty connections. Although there is not a unified color standard for wires in breadboarding circuits, there

are some common conventions such as the one used in the Arduino pinout diagram shown in Figure 6.3.1.2.

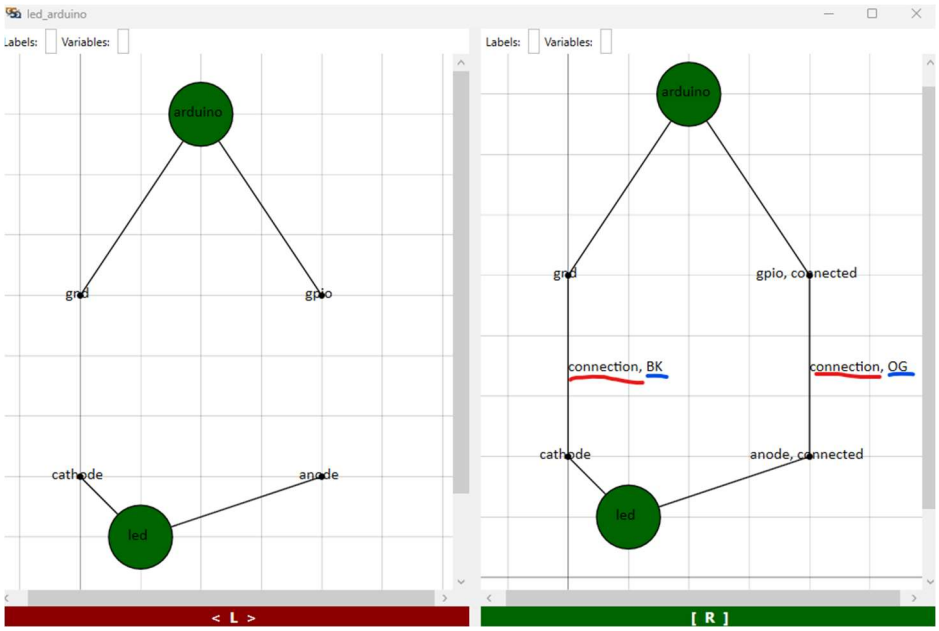


Figure 6.3.1.1: Rule *led_arduino.grxml* specifying the connection between LED and Arduino

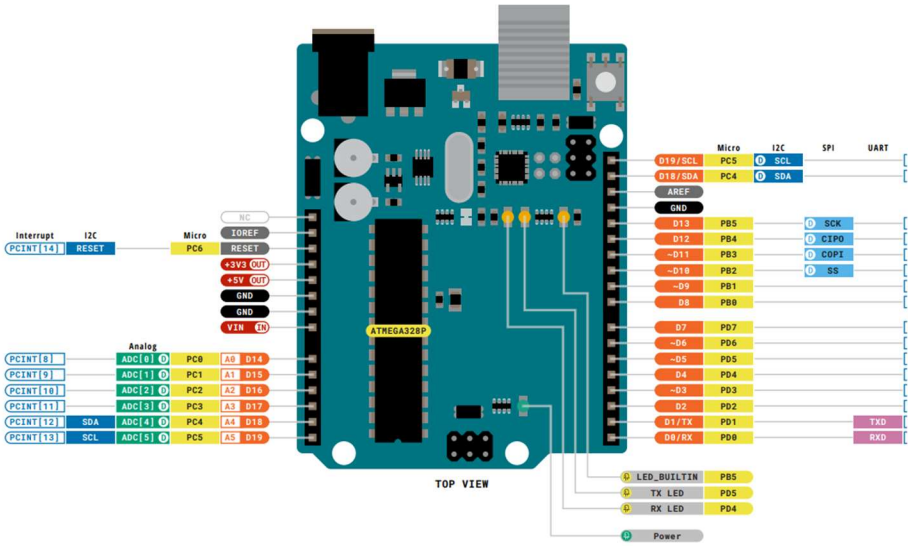


Figure 6.3.1.2: Arduino Uno R3 Pinout Diagram (Arduino, 2022B)

With the help of the pinout diagram, we created Table 6.3.1.1:

Table 6.3.1.1: Color Code for Different Connections

Color	**Color code**	**Type**
Black	BK	Ground
Red	RD	Power
Light Blue	LB	SPI
Dark Blue	DB	I2C
Green	GN	Analog
Violet	VT	UART
Orange	OG	Digital

We also had to decide what path the wires would take. We decided that pathfinding was too difficult to implement, so we settled on simple straight lines. Finally, we decided to allow users to arrange the components themselves, rather than try to arrange them with an algorithm.

6.3.2 Combining Components

When building a circuit, it is often required to use additional components for a specific component to work properly. For example, if an LED is needed in a circuit, it is often necessary to wire a resistor in series to prevent burning the LED. Such practices are common among engineers who have some level of experience with electrical circuitry, which is not expected for users in this project's scope.

Therefore, we decided to combine components that are typically used in this way into one 'combined component.' The visual changes of the 'combined components' are also briefly discussed in this section.

One of the considerations about this decision was the ease of implementation. The envisioned interaction of the program was that when a 'component' is dragged and dropped onto the workbench, the connections should appear automatically. This interaction conflicts when components require other complementary components to work. Since components not manually added by the user will also appear, it may cause confusion. Additionally, on the backend, the system is set up to only add one rule per component added. In an example case, using an LED would require two rules: one to connect the resistor with the LED, and one to connect the set to the Arduino. Ultimately, we decided that fixing this was more difficult than simply combining a standard resistor into the existing LED component. With the implementation of combining components, the 'combined LED component' included a resistor connected in series by default and was treated as one entity (as shown in Figure 6.3.2.1). This eliminated the confusion related to unwanted components being added and allowed us to use only one rule per component.

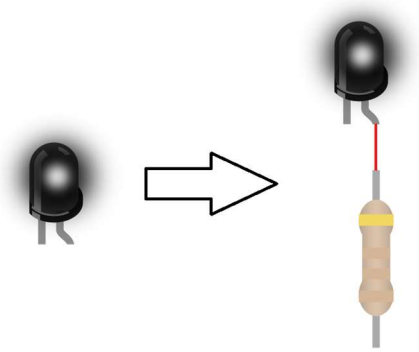


Figure 6.3.2.1: 'LED Component' Before (left) and After (right) Combining.

Another consideration about the decision to use combined components is the user experience. As mentioned at the start of this section, the targeted audience of the program might not find the intermediate connections intuitive. A novice user can be confused if the connection is not displayed as expected. This increases the learning curve of this program, which is against the requirement of this program being "easy to use." With the complementary components combined, the intermediate connections are drawn out by default. The users no longer need to know about the intermediate connections to use this program.

Aside from the LED, there are other components that require combining complementary components. We also created variants for some components because the complementary components are optional. This means that the user has access to combined elements that both include and do not include the complementary components described below. Table 6.3.2.1 shows the combined components in GraphSynth rules. The shaded cells in the table are the components that require rule changes.

An example of rule changes of a loadcell is presented in the subsequent figures. A loadcell needs to be connected to an HX711 to connect to Arduino. As shown in Figure 6.3.2.2, four rules are needed to complete the connections originally. After combining HX711 with the loadcell, the first three rules are also combined into one rule. The detailed changes are presented in Figure 6.3.2.3, Figure 6.3.2.4, and Figure 6.3.2.5.

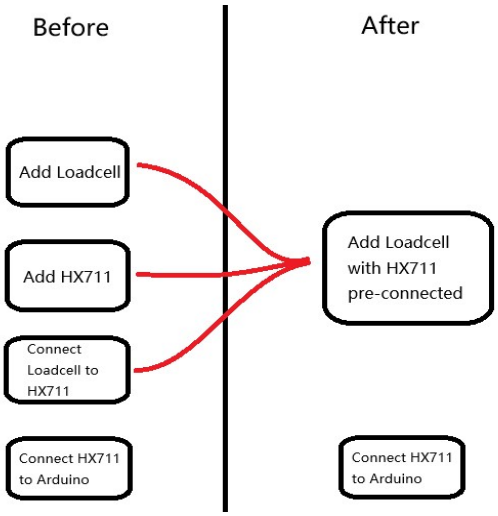


Figure 6.3.2.2: Before & After rule changes

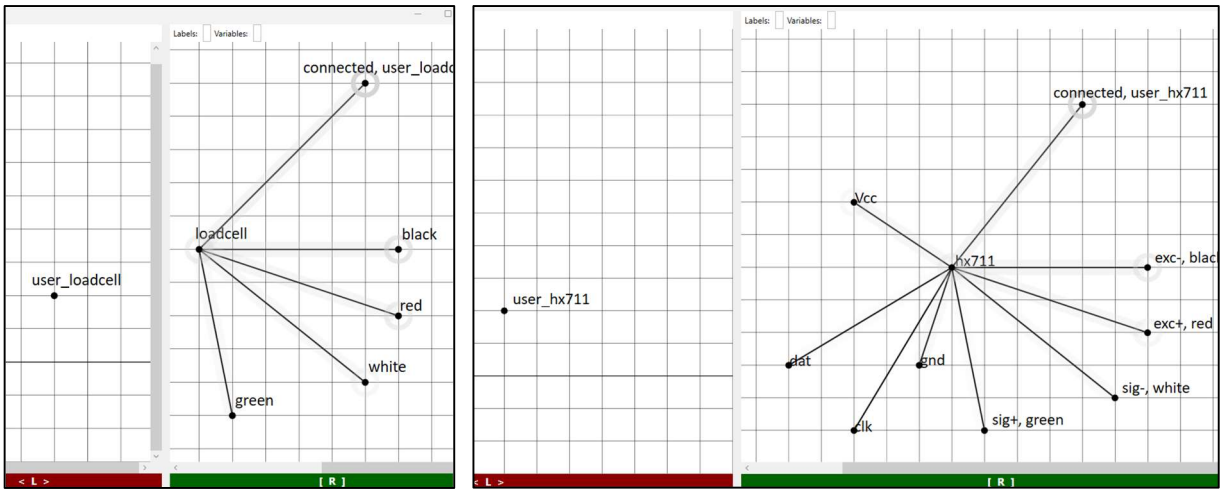


Figure 6.3.2.3: Original *add_loadcell.grxml*(left) and *add_hx711.grxml*(right)

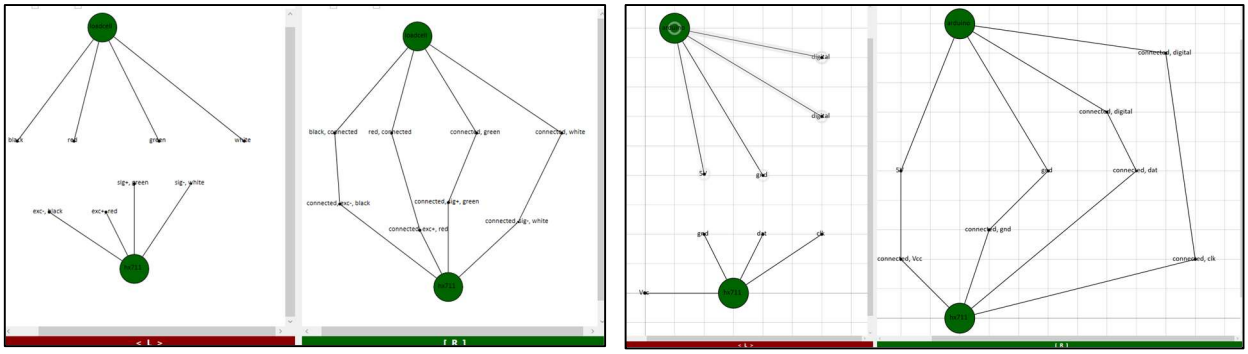


Figure 6.3.2.4: Original *hx711_loadcell.grxml*(left) and *hx711_arduino.grxml*(right)

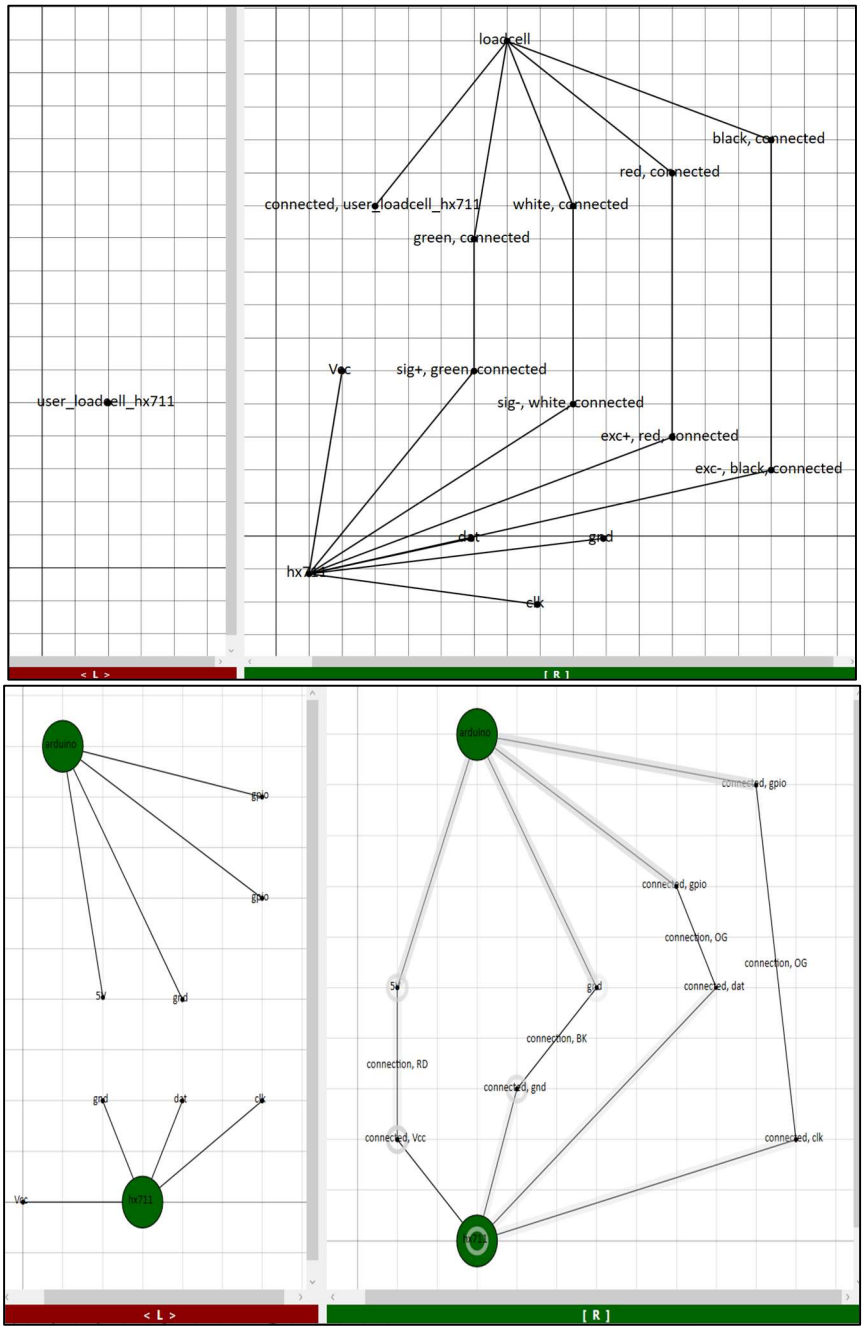


Figure 6.3.2.5: New add_loadcell_hx711.grxml(Top) and hx711_arduino.grxml(Bottom)

Table 6.3.2.1: Combined Components

Original Component Name	**Complementary Component**	**Combined Name**
led_rgb	3 resistors	led_rgb
pushbutton	resistor	pushbutton
ds18b20	resistor	ds18b20
servo	pca9685	servo_pca9685
servo	NONE	servo_direct
servo_9g	pca9685	servo_9g_pca9685
servo_9g	NONE	servo_9g_direct
dc_motor	l298n	dc_motor_l298n
dc_motor	NONE	dc_motor_direct
loadcell	hx711	load_cell_hx711
stepper_nema	l298n	stepper_nema_l298n
stepper_nema	a4988	stepper_nema_a4988
speaker	lm386	speaker_lm386
max6675	k_type_Thermocouple	max6675
esc	brushless_dc_motor	esc
esc	pca9685	esc_pca9685

6.4 Visual Component Design

This section explains the visual design and implementation of our components, both for components that can function independently and components that require another to function.

6.4.1 Wokwi Conversions

The design of our components was largely based on how Wokwi handled their component design and functionality. Wokwi used components called from their Wokwi Elements library (Shaked, 2024), which combined information from an optimized scalar vector graphic (SVG) file with Typescript functionality. These SVG files were designed to match the specification and look of the component it was made to simulate. The file was then completed using functionality from lit-element, a small library used to create Web Components. The inclusion of lit-element is how Wokwi allowed for their components to have visual responses and interactions (Shaked, 2020). However, these components were designed for a pure JavaScript environment. On the other hand, we are using the Blazor system, which can create components, although the syntax is completely different. We copied and edited many of the components from Wokwi Elements to fit Blazor syntax. At the most basic level, Wokwi Elements uses a `render()` function to display HTML. We removed only the function definition, as Blazor allows components to be rendered without a function. Similarly, Wokwi uses a braces-based syntax for injecting variables into static text, while Blazor uses parentheses. Many issues like these were fixed for each component. We initially started with components that had existing GraphSynth rules and an available Wokwi Element. Then we continued to create other components in order of ease of testing and development.

6.4.2 Creating New Components

While most of the components we needed already had SVGs provided by Wokwi Elements, we did need to create SVGs for ten components. We tackled this problem using our experience with the

previous conversions and SVG creation. Our created components followed the same art style and code patterns as the converted components. Cam Tu Le's project had already listed most, if not all, of the necessary data sheets used in the original case studies, which meant creating accurate and recognizable components for students was much easier. The SVGs for each component were all created on a design app called Inkscape (Inkscape, n.d.). Inkscape is a free and open-source vector graphics editor. The main strength of Inkscape came from its ability to properly optimize each SVG of component we make, removing excess comments and metadata that are part of regular SVG files during their creation. An example of a SVG we created is shown below in Figure 6.4.2.1.

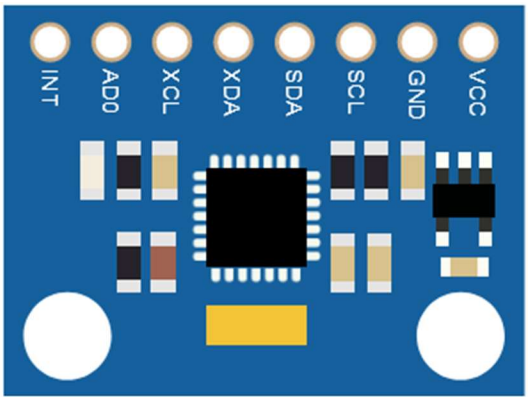


Figure 6.4.2.1: Designed MPU6050

Additionally, each of the combination components had to be created individually. For this project, we only created three of these, one of which can be seen below in figure 6.4.2.2. These weren't used in the demo due to issues implementing their translation functions.

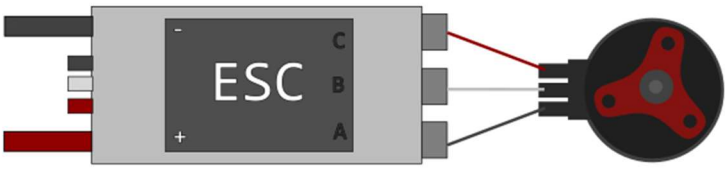


Figure 6.4.2.2: ESC & BLDC Motor combination component

In the current iteration, our Blazor components consist of three parts. The first is the optimized SVG image's code we created earlier, which contains the size, color, and shapes of each part of the component. The second is a pinInfo function, which stores the pin locations for each component. The function came with the original Wokwi Elements, and we decided not to change it. This function mainly serves to assist the GraphSynth connections so the wire connects properly in the circuit image. By the end of development, we had created and converted thirty-three Blazor components that could appear on the Component Palette. The third and final part is a translation function. The translation function describes the functionality of the component to provide the visual output to indicate a signal response from the Arduino. The process for creating translation functions is found in Chapter 7.7.1.

The design decisions described in this chapter were made to allow the system to mimic the design process of a physical Arduino project. It was important to consider how components are stored, what they look like, where the circuit is built, how components are wired, and what the IDE should look like when designing using ADArC. The next chapter will cover how we implemented these design choices.

7. ADArC Implementation

For many of our implementation details, we looked to BoGL Web (Vuolo et al., 2023), another similar past MQP. BoGL closely aligns its implementation with GraphSynth (DesignEngrLab, n.d.), directly including many of GraphSynth's core files. Notably, BoGL's implementation involves several key differences from the original GraphSynth code.

First, it includes namespace adjustments. BoGL adds several imports to GraphSynth files. These adjustments help to better integrate GraphSynth in BoGL Web, reducing potential naming conflicts, and enhancing code organization. Second, BoGL changed a number of variable types in the original files. This entails converting data structures, variables, or objects to match the expectations of the BoGL project, ensuring the compatibility and functionality of the integrated code. Lastly, BoGL maintains uniform code styling. As it imports code from `GraphSynth.Base`, BoGL applies a consistent code styling approach, which includes formatting conventions, indentation, commenting practices, and naming conventions. This commitment to uniformity not only enhances code readability but also simplifies maintenance and future development.

In our project, we do some of these things, but not all of them. In most cases, nothing in our project has changed from the original GraphSynth code. We found it unnecessary to use custom imports as our project structure was different. We did run into some problems when attempting to serialize GraphSynth objects to JSON. We figured out that several of the types GraphSynth used were deprecated and thus could be serialized. In those cases, we changed the type to an updated version. Lastly, we did

not invest the time needed to format GraphSynth's code. This is the extent of our changes to the GraphSynth code in our project.

7.1 UI Implementation

After discussing every design decision and testing expected user interactions (as explained in Section 6.1.1), we continued refining the UI to what you can see in Figure 7.1.1. Every part of the app got a title, and we decided on user-friendly names that were easy to remember. The component selection panel was renamed to Component Palette, the simulation canvas to Workbench, and the code editor and console to the Code Panel. We also added lightbulb icons next to the titles that, when clicked, open popup tooltips to guide new users. The toolbar was also separated into sections. To the very left is the logo and title of the app, then a guided tutorial, then the zoom buttons, and then all other buttons: import, export, help, and run. When selecting the Run button, instead of changing to 'Stop' right away, it shows the word 'Compiling' while it is still loading. This process can be seen in Figure 6.1.4.1.

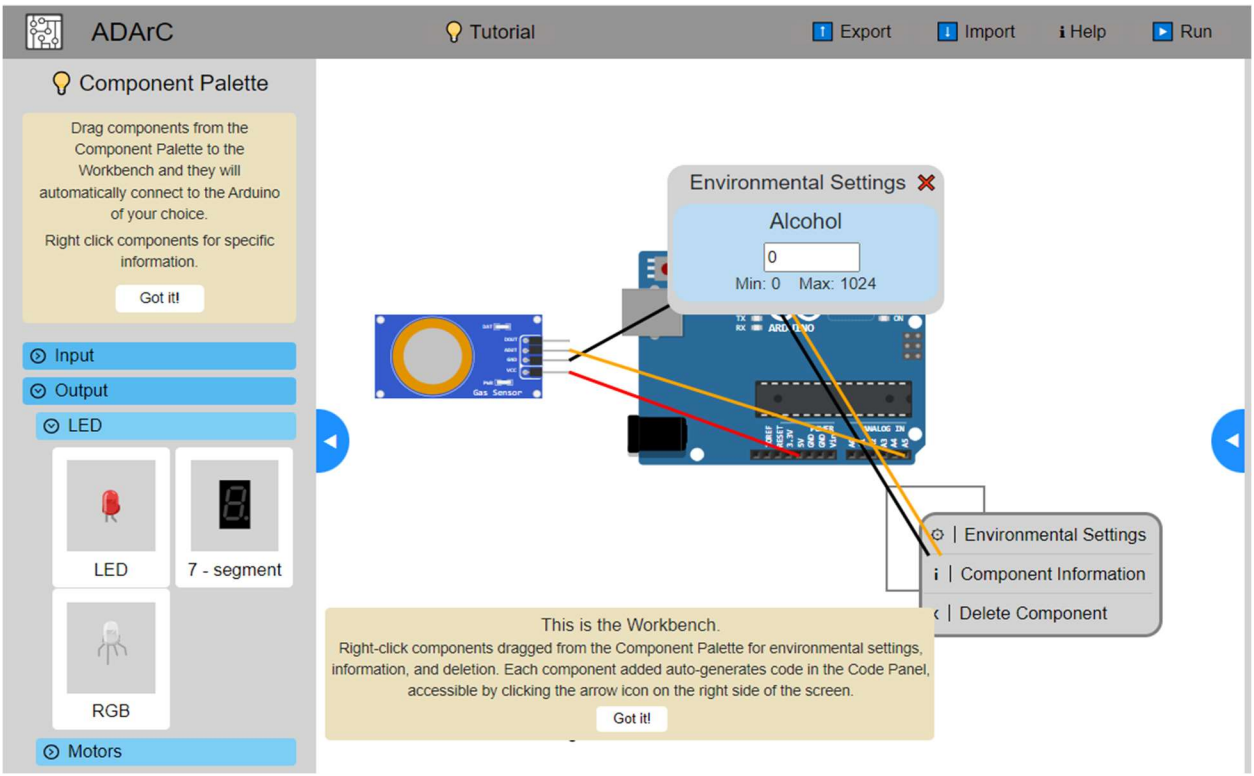


Figure 7.1.1 Final UI Design

Initially, components in the Workbench were going to be deleted by clicking an 'X' icon that would appear at the top right of a component when selecting it. This was eventually rolled into the combined right-click menu seen in the lower part of Figure 7.1.1. Later we added a help button for specific component information and an Environment Settings option. These were also originally separate buttons on the component before being incorporated into the right-click menu. In Figure 7.1.2, the original three-button design can be seen.

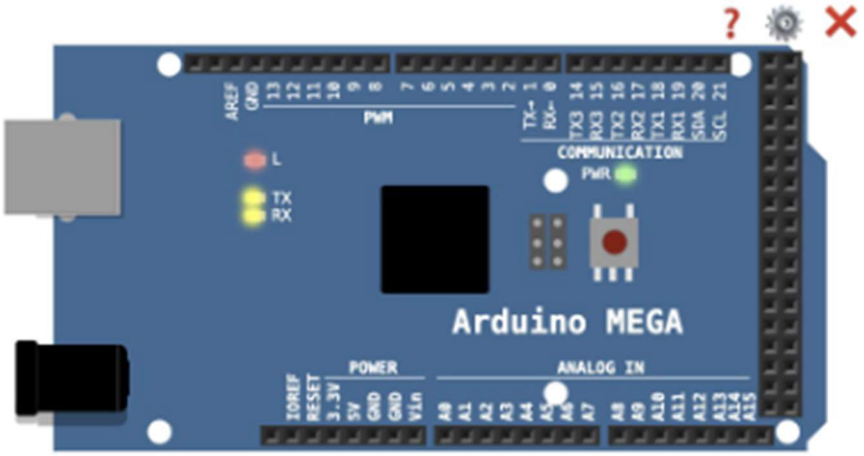


Figure 7.1.2 Initial Three-Button Selection Design

To ensure uniformity in both user interactions and the overall visual appeal of the application, we have adjusted the presentation and activation of buttons. In Figure 7.1.2 is our old design for accessing component information, environmental settings, and deletion. These buttons were visible upon selecting a component. It was later changed to using a right-click on a selected component to access these features, to stay consistent with the right-click menu on the Component Palette (as shown on the left side of Figure 7.1.3).

When right-clicking a component in the Palette, the user is unable to toggle environmental settings. Instead, users can quickly review the available environmental settings for that component should they add it to the Workbench. Additionally, all icons within the menu maintain a consistent color and style for a cohesive aesthetic. These right-click menus disappear after a left-click outside their borders.

In summary, to select components in the Workbench, they have to be clicked, a black outline will appear around it, and when right-clicked, the context menu appears (as shown on the right side of Figure 7.1.3).

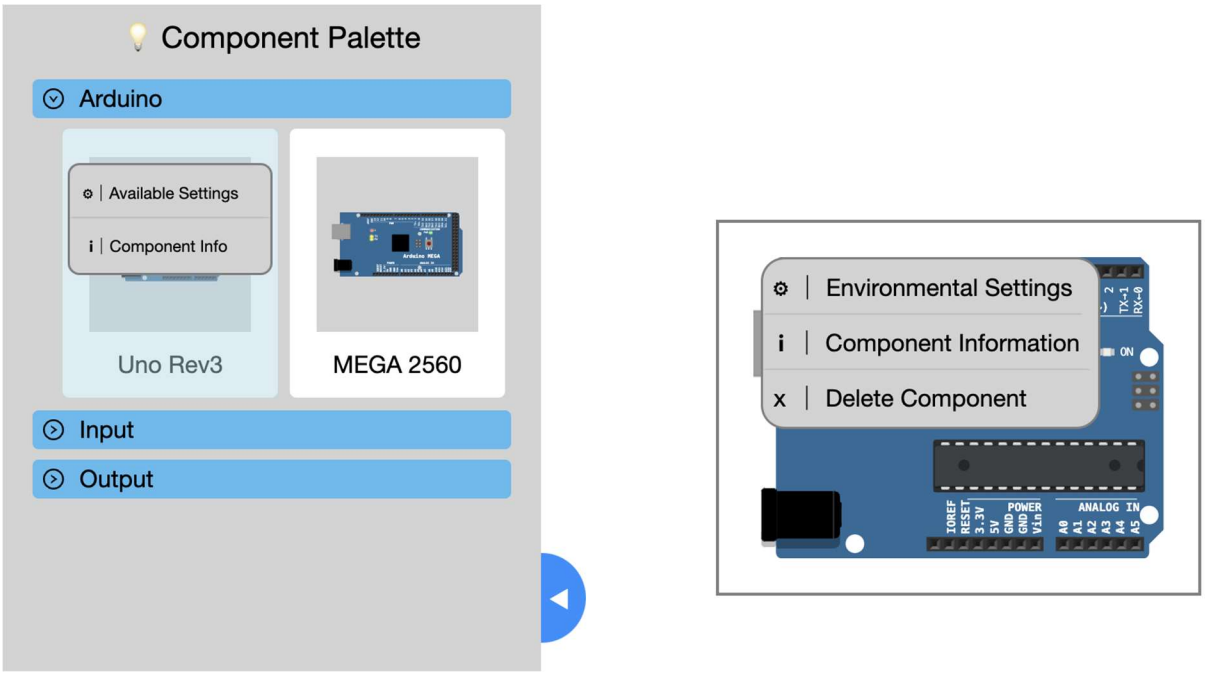


Figure 7.1.3 New Context Menus (right-click)

7.2 Component System

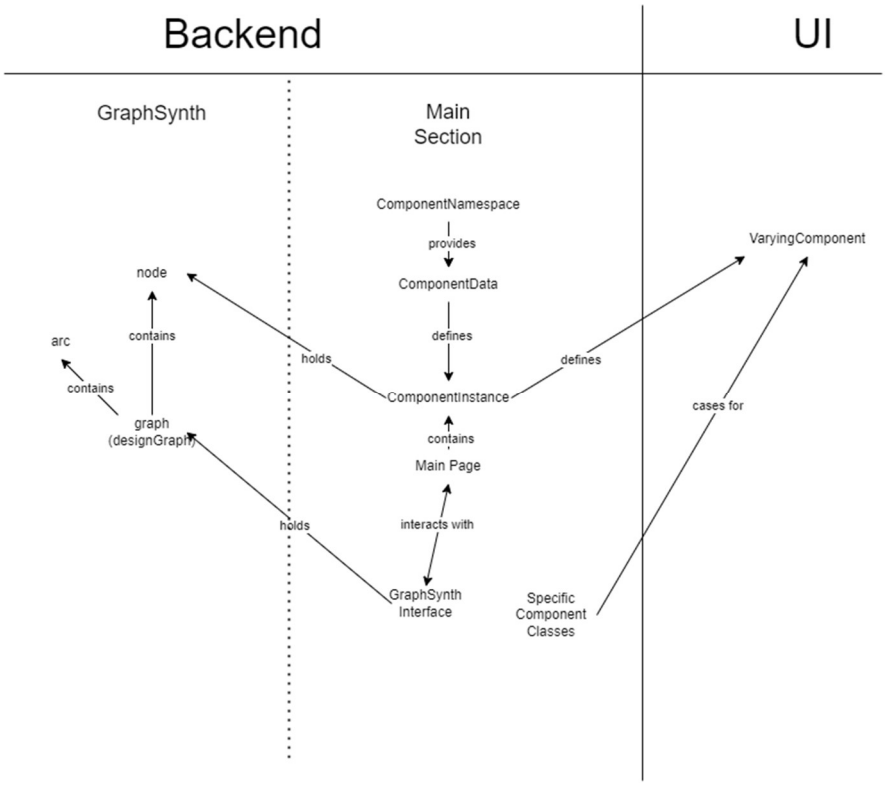


Figure 7.2.1: Internal Application Structure for Components (Arrows Read Tail-to-Head)

We faced several problems during the implementation of particular components, given the amount of information they store and their uniqueness. To begin with, we envisioned a data repository containing definitions which would provide a standard starting template for each component. This has come into existence as the Component Namespace-Data-Instance System seen in the middle of Fig 7.2.1. The component namespace simply stores all the data about each type of component as a ComponentData object, notably keyed by a unique global identifier. For example, the LED contains the name “LED,” its place in the component palette, “Output/LED,” and its size of 40 pixels by 50 pixels.

A ComponentInstance is instantiated using a global ID to get the data, a position, and, in some cases, a starting state. These Instances are stored as a Dictionary in the main class, Index.razor. When a component is added to the workbench, a new entry in the Dictionary is created, keyed by a numeric local identifier which simply increases with each new component. The local identifier serves to distinguish each component on the Workbench uniquely. However, these are still internal objects at this point and still need to be sent to the screen to be drawn. For this, we send the instance to another class called VaryingComponent, which translates the instance into something drawable using the component type stored in the data. Connections are handled by GraphSynth through a wrapper class.

7.3 Adding Components

Adding components to the circuit is an essential step for users to build a circuit. In the UI, users can drag a component from the Palette to the Workbench to add that component to the circuit. To handle the added components in the backend, it seems most appropriate to incorporate them within the 'designGraph' structure introduced by GraphSynth and covered in detail in section 3.3. We interface to the 'designGraph' using the 'BaseGraphClasses.designGraph.addNode(node n)' function. From there, to connect the graph nodes representing the components, we use the "recognize and apply" process, also described in section 3.3, with our GraphSynth rules. Additionally, to increase processing speed, we only apply rules that contain the name of the component added. It results in a designGraph that contains information about the nodes and arcs that will be used for the visual circuit diagram on the Workbench in the UI later. For example, if the user drags a LED from the component palette and drop it on the workbench, a node named "user_led" will be added to the designGraph. As shown in

Figure 7.3.1, there is the representation of an Arduino Uno Rev3 on the left and a newly added node on the right. Subsequently, the rules named "*add_led.grxml*" and "*led_arduino.grxml*" are applied. The resulting designGraph (as shown in Figure 7.3.2) contains nodes and arcs to specify the connections.

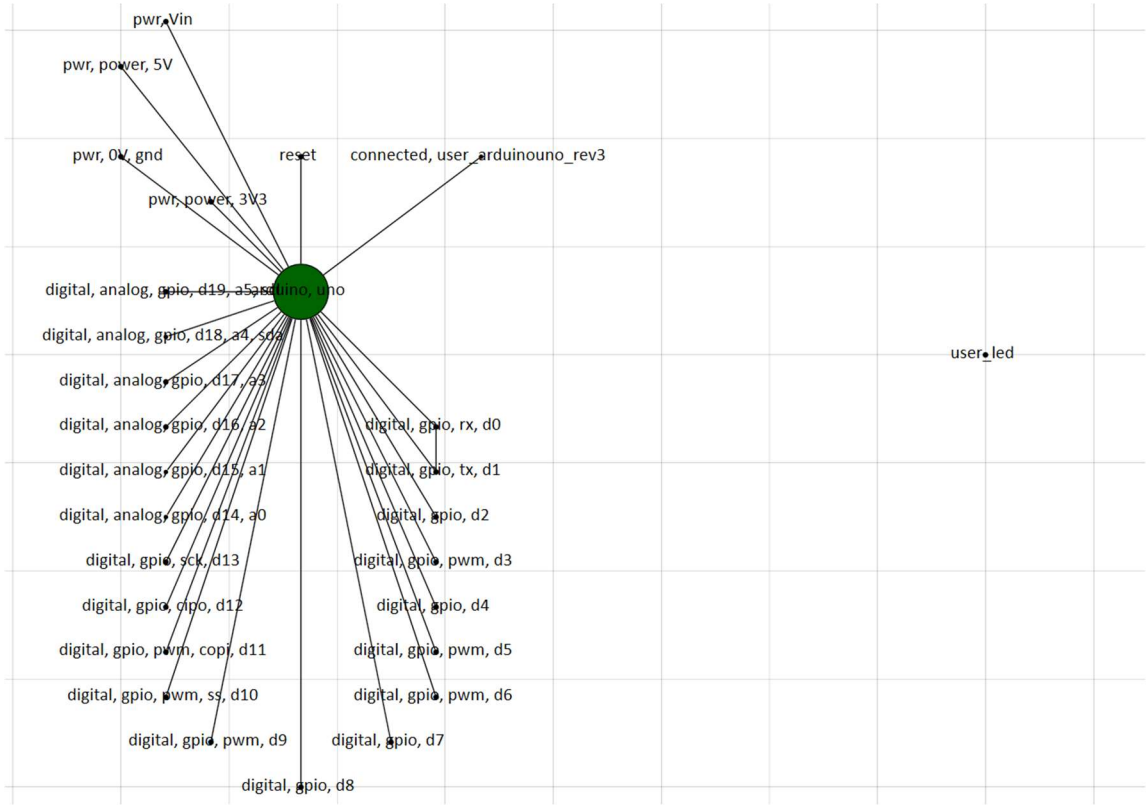


Figure 7.3.1: Initial designGraph when a LED is dragged to the Workbench

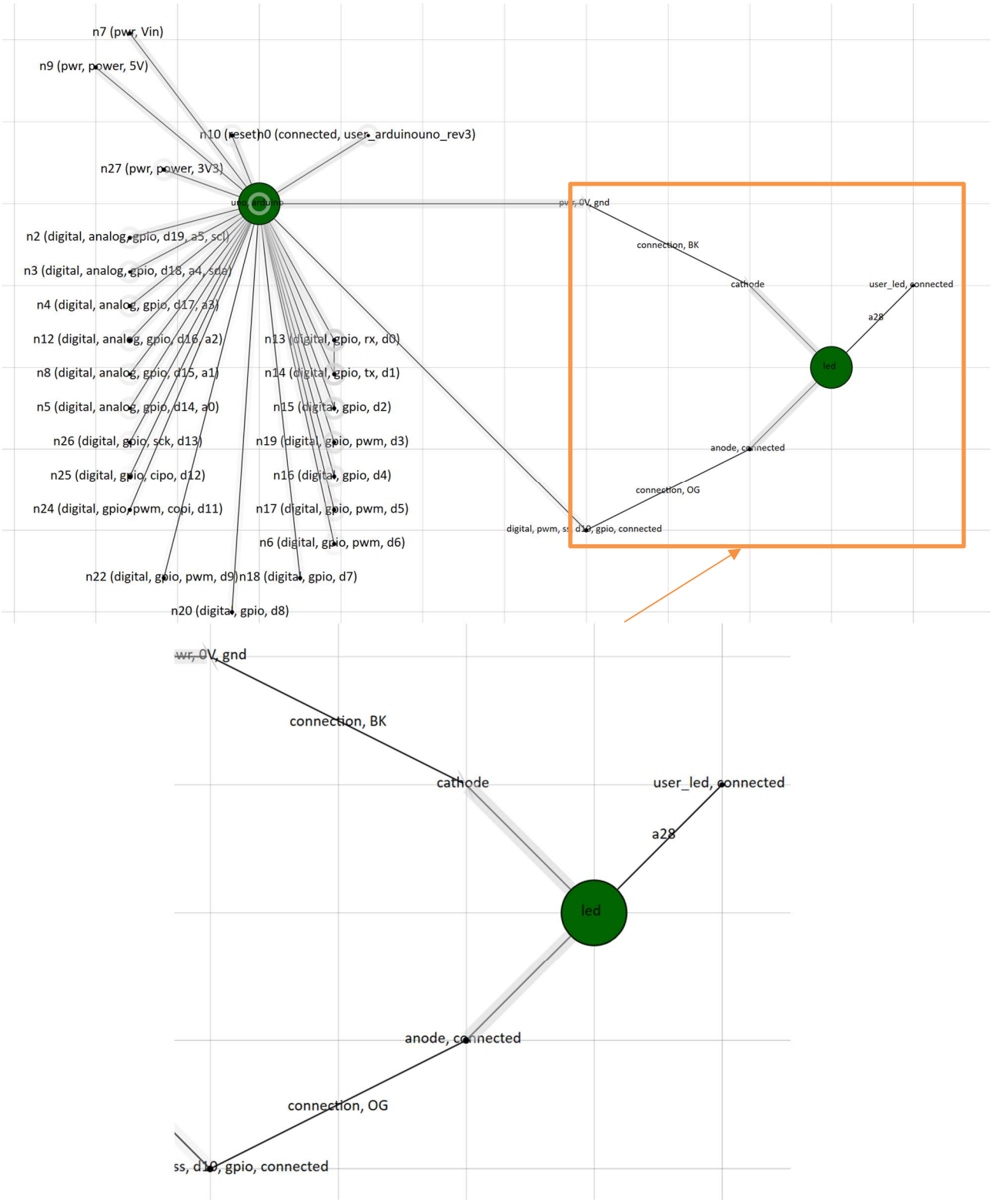


Figure 7.3.2: Final designGraph after rules about LED are applied.

The next step is to use the data in the designGraph for the visual circuit diagram on the Workbench in the UI. While this seems simple, it poses several problems for integration with other parts of the project. The first problem is the lack of easy ways to associate the GraphSynth 'node' with the ComponentInstance. It is problematic because the list of nodes in a designGraph contains nodes from all the components, making it difficult to distinguish them. The second problem is forming the associations with the virtual Arduino pins. To solve the first problem, when the components are added to the graph, the 'node' is added to a variable in the ComponentInstance and all of the new nodes and arcs in the graph are labeled with the component's global ID. The GraphSynth rule application process creates connections between nodes representing component pins. These arcs are labeled "connection" internally and stored. To solve the second problem, every time a new component is added, these connections are reassessed. Then, the ComponentInstance at each end of the arc stores the intermediate connection as a list of InstanceConnections. These are subsequently used to draw the connection lines, representing wires, on the screen later in the process. Continuing with the previous example of adding a LED, as shown in Figure 7.3.3, the nodes and arcs that belong to the LED are labeled "GlobalId:1", and the arcs that represent connection between the Arduino and the LED are labeled "connection".

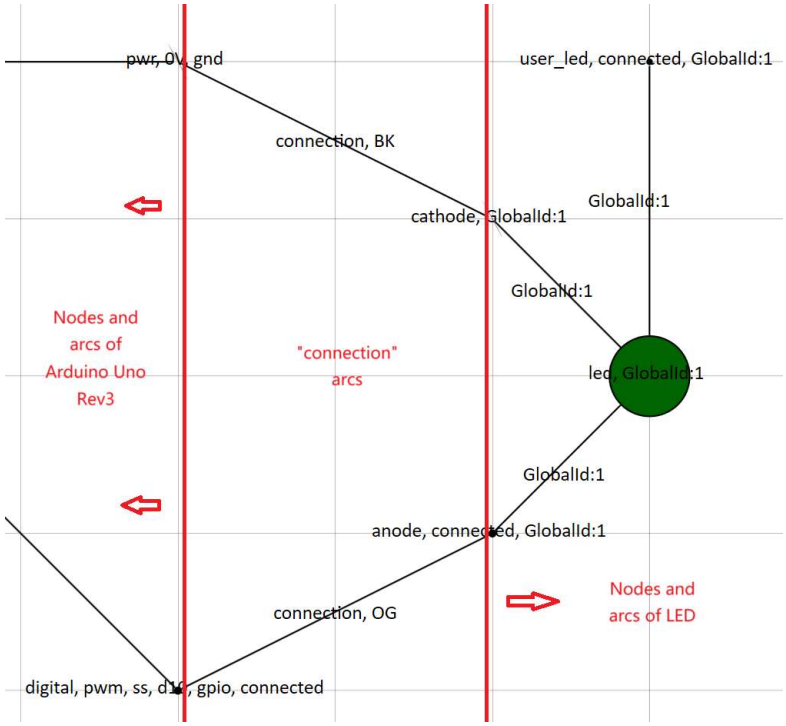


Figure 7.3.3: Final designGraph after rules about LED are applied.

Originally, the Arduino itself was a component that could be added. However, several of these steps rely on a component being connected to an Arduino immediately, so we decided to keep an Arduino in the Workbench permanently while removing it from the Component Palette. This also solved some potential issues with components connecting to multiple Arduinos at once.

7.4 Connection Lines

Drawing the connection lines (i.e., wires) was not difficult, though there were some persistent issues caused by our choice of implementation. We already had information on the position of each component and their pins, so by adding the two together, we could find pin positions easily. After that,

we only had to draw straight lines between the coordinates, so it was relatively simple. To start, we created an empty SVG layer to cover the entirety of the Workbench. This allowed us to use SVG lines. By adding `<line>` tags to the SVG layer, coordinate-based lines could be drawn. However, the SVG layer relies on an internal coordinate system, which meant that it had to be precisely aligned with the main workbench area. At first, the scaling of the SVG was inconsistent with the main screen, and various HTML paddings made it difficult to position the SVG layer properly. Once these issues were solved, it was straightforward to compute the position of each coordinate of the line based on the component position and pin offset. For example, if a component exists at (600, 300) and has a pin offset of (30, 40), the line coordinate becomes (630, 340). We can also drag the canvas around, adding an offset to all components, calculated separately. Continuing the example, assume we have dragged the canvas to an offset of (-100, -200), or up and to the left. The final coordinate becomes (530, 140).

We had many problems trying to integrate these systems with the zoom system. We could not get the coordinates to line up properly due to a scaling mismatch between the SVG layer and the individual components. Eventually, we decided that the lines were more important than zoom and removed the zoom system. There are still some issues with the positioning of the SVG layer in the site's HTML tree (hierarchy of rendering on the z-axis), leading to the lines always being displayed over the right-click menus.

7.5 Code Generation

Our approach to automated code generation was to utilize the pre-existing example code for each component. To align with conventional Arduino code, we broke the example code into snippet

categories including additional libraries, global variable declaration, setup, loop, and self-defined functions. To generate the code for a circuit as a whole, we assemble each snippet in order at predefined positions in the code based on the type of the snippet. An example of where each category is placed can be seen below in Figure 7.5.1. Components often have snippets in most or all of these categories.

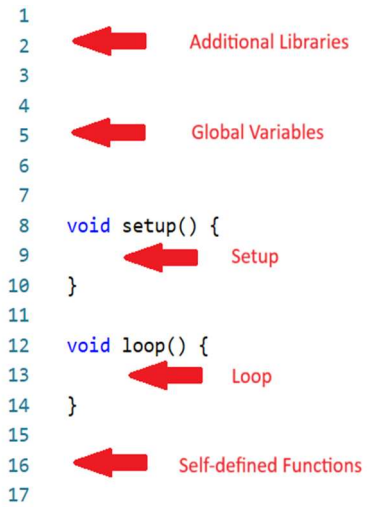


Figure 7.5.1: Positions for Code Snippets

Since the circuit is generated automatically, we do not know which Arduino pin the component will be connected to. We used the syntax '~"pinName"' to indicate the connected Arduino pin, where pinName is a unique label present on the GraphSynth pin node and present in the defined pins for that component. When a component is added, the syntax '~"pinName"' in the snippets is replaced by the actual pin number.

For example, if an LED is integrated into the circuit, it should be connected to pin 13 as illustrated on the left side of Figure 7.5.2. Subsequently, the number "13" will replace the placeholder '~"anode"' in the code snippets related to the LED, prior to entering the code into the console. The

resulting code is displayed on the right side of Figure 7.5.2. The snippets are placed into corresponding locations in the code panel. Since LED is a simple component, the predefined code does not include additional libraries, global variables, or self-defined functions.

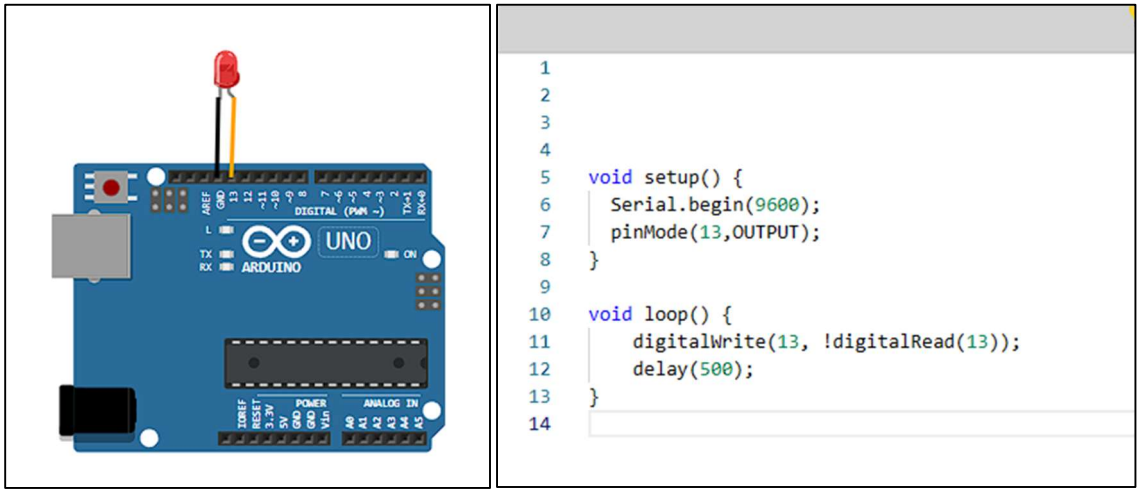


Figure 7.5.2: Circuit of a LED (left) and the generated code (right)

7.6 Component Removal

The ability to remove components from the workbench is a fundamental requirement. Users rely on this feature to edit circuits effectively, tailoring them to their specific needs. On the UI side, we initially had an ‘X’ button (as shown in Figure 7.1.2) on the top right corner of the selected component on the workbench. However, the icon ‘X’ is crowding the workbench with other icons. We decided to change it into a 'delete component' button in the right-click menu (as shown in Figure 7.1.3). When the button is clicked, the selected component disappears along with its connection wires. On the backend, the ComponentInstance about the component is removed and the designGraph is updated accordingly. We implemented a function to modify the designGraph structure utilized by GraphSynth. This

function removes all nodes and arcs associated with the component, along with the arcs representing connections between the component and the Arduino. Additionally, it removes any “connected” labels from previously linked Arduino nodes. For example, there is a LED connected to the Arduino as shown in Figure 7.6.1. The underlined labels are the identifiers for the nodes and arcs that need to be removed. The circled label is the “connected” label that also needs to be removed to restore the availability of the pin. Figure 7.6.2 shows the updated designGraph after the LED is removed.

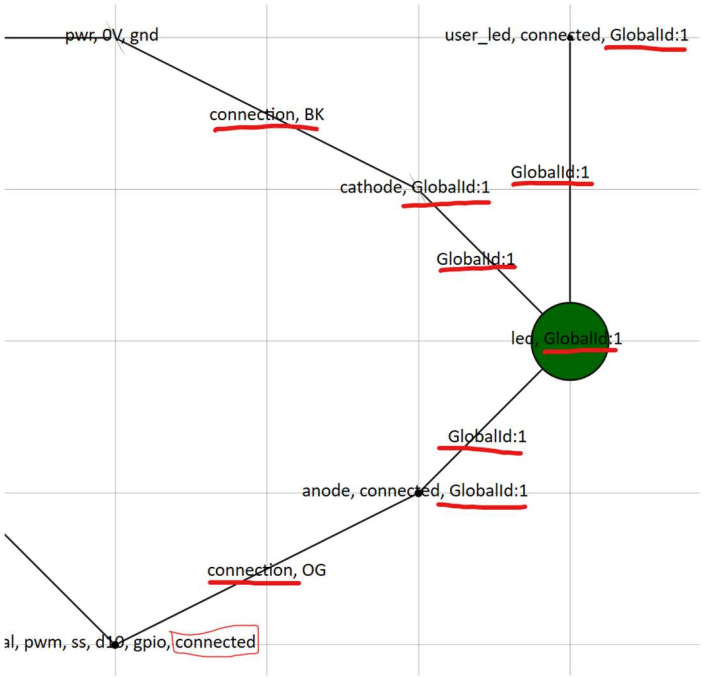


Figure 7.6.1: Nodes, arcs and labels that need to be removed.

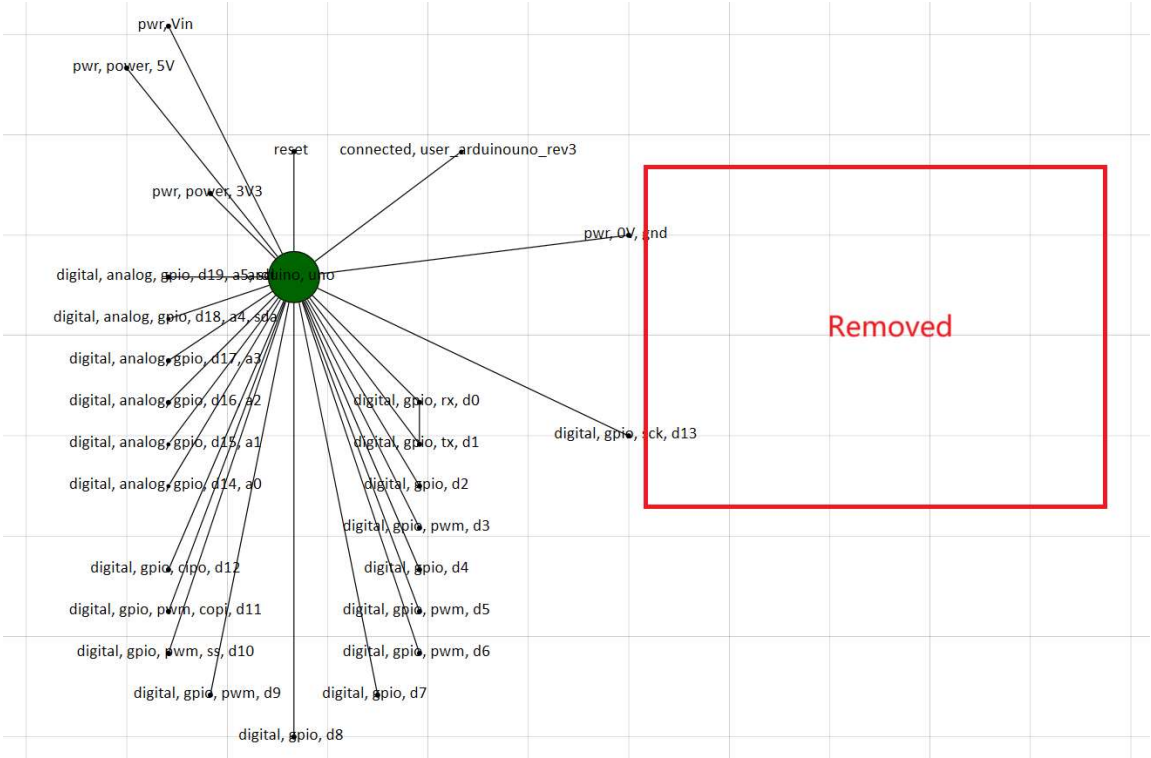


Figure 7.6.2: designGraph after the LED is removed.

7.7 Component Simulation

For the simulation, we use a variety of strategies. The code is mainly written in C#, though we depend on AVR8js (Shaked et al., 2023), an Arduino simulator written in Typescript. Below, Figure 7.7.1 is a simplified flowchart of the simulation process, showing the simulated Arduino on the left.

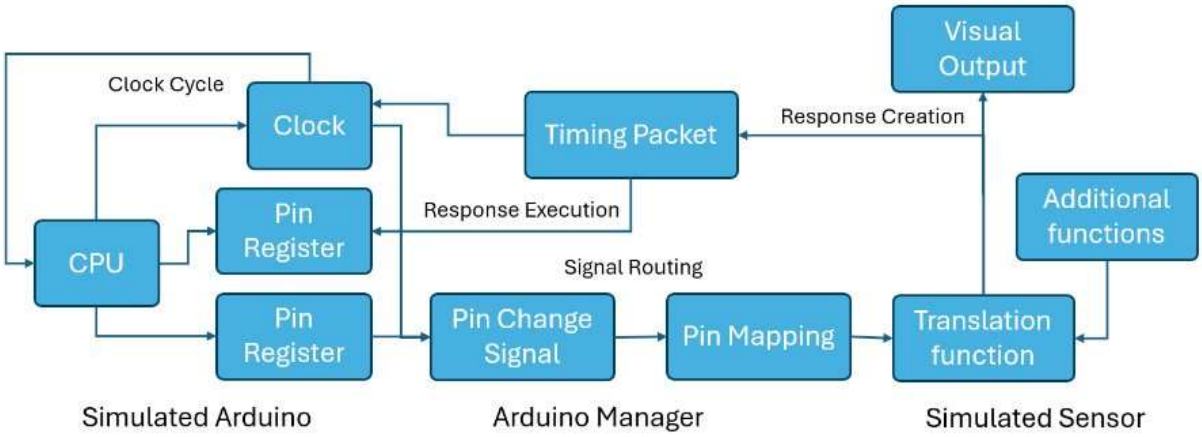


Figure 7.7.1: Simplified flowchart of simulation system

To bridge the gap between C# and Typescript, we use a wrapper class and C#'s interoperability with Typescript. These processes are shown in Figure 7.7.1 as the Arduino Manager. It uses both C# and Typescript code to transfer data. Connections between the Arduino and components are set up based on a system of labels in the GraphSynth graph, including labels on the inter-component connections and the individual nodes that represent the pins. This is explained further in chapter 7.3. These are stored by pin number and local ID for future use in "translation functions," which take Arduino signals and convert them to appropriate visual output.

As an example, a DHT is connected to the Arduino on pin 13. When pin 13 changes, part of pin register D, a change signal containing the data of register D (pins 8 to 13) is sent to the manager. The manager then compares against the last known value of register D to find what has changed. The manager then finds that the DHTs pin has changed and sends details on pin 13 to the DHT for a

response. The DHT calculates its response, passes it to the manager, and the manager updates the Arduino pins.

7.7.1 Translation Functions

The purpose of translation functions is to make components behave correctly in the simulation. In other words, translation functions decode the Arduino signals that are sent to components. In reality, each component communicates with the Arduino board differently according to the respective datasheet. The communication protocol of a component is predetermined by the manufacturer's design. Components only react when exposed to the supported protocol.

The communication within our application can be classified into four categories based on the included components: Pulse Width Modulation (PWM), Inter-Integrated Circuit (I2C) communication protocol, Analog Input, and various other communication protocols. For instance, an RGB LED uses PWM for communication, while the BNO55 acceleration sensor employs the I2C communication protocol. The MQ-3 sensor operates as an analog input and the DHT22 temperature and humidity sensor has its own unique communication protocol. These protocols are predetermined for each component, usually defined in the component’s data sheet. In total, we created translation functions for 14 components. The list of components with translation functions can be found in Appendix G.

Pulse Width Modulation

Pulse Width Modulation (PWM) is a technique used in electronics to control the amount of power delivered to a load by varying the duration of pulses. For Arduino microcontrollers, it is often used to output analog signals with digital means. These pulses occur periodically.

In PWM, the duration of the pulses, also known as the pulse width, is varied in proportion to a modulating signal. This proportion, often referred to as the "duty cycle," is expressed as a percentage. It represents the fraction of time that the signal is in the high state compared to the total period of the signal. The modulated analog output results from multiplying the reference voltage by the duty cycle. The frequency of the cycles depends on the specific pin that outputs the signal. In the case of Arduino Uno Rev 3, it is 490 Hz for pins 3,9,10, 11, and 980 Hz for pins 5 and 6. Figure 7.7.1.1 shows an example signal from five different analog outputs from Arduino. The PWM output from Arduino is generated by calling the function analogWrite(int). Here, the parameter "int" is a number ranging from 0 to 255, which is the desired value we aim to obtain with the translation function.

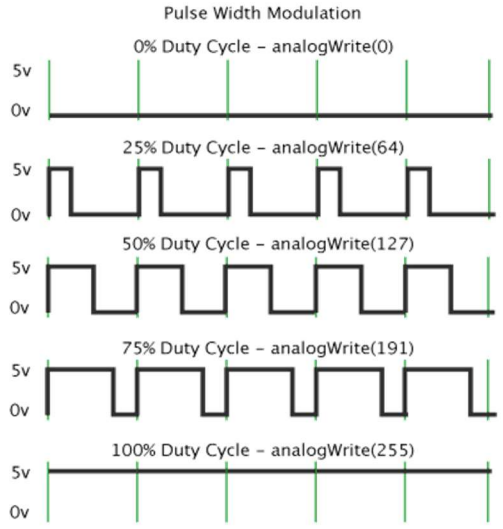


Figure 7.7.1.1: Pulse Width Modulation (Hirzel, 2022)

The first step to translate the PWM signal was to measure the pulse width. It is done by comparing the times at both the rising edge and the falling edge of the pulse. With the pulse width, we determined the duty cycle by dividing the pulse width by the period, which is the inverse of the frequency.

$$period = \frac{1}{frequency}$$

$$Duty\ Cycle = \frac{Pulse\ Width}{period}$$

With the duty cycle, the PWM values can be calculated by multiplying 255 by the duty cycle. That value is then used to perform actions of the component. For example, if Arduino sends a 50% duty cycle signal by executing the function `analogWrite(127)` to the "R" pin of the RGB LED that controls the red, the calculations above will be performed and the number 127 will be acquired. Then the red portion of the RGB LED will have a 50% brightness.

I2C

The Inter-Integrated Circuit (I2C) protocol is a widely used serial communication protocol for connecting microcontrollers to peripheral devices. It utilizes two wires: Serial Data (SDA) and Serial Clock (SCL). I2C features a master-slave architecture, with the master device initiating communication and controlling the communication line, while one or more slave devices respond to commands from the master.

According to Texas Instruments, the I2C protocol includes key conditions for devices to talk to each other effectively (Texas Instruments, 2015). These include:

Start Condition: Marks the initiation of communication on the I2C communication line.

End Condition: Marks the conclusion of communication on the I2C communication line.

Write Operation: Involves transmitting data from the master device to a slave device.

Read Operation: Involves receiving data from a slave device by the master device.

ACK/NACK: Acknowledgment and Non-Acknowledgment signals sent by the receiver to indicate successful or unsuccessful receipt of data.

Repeated Start Condition: Allows the master device to maintain control of the communication line for sequential operations without releasing it.

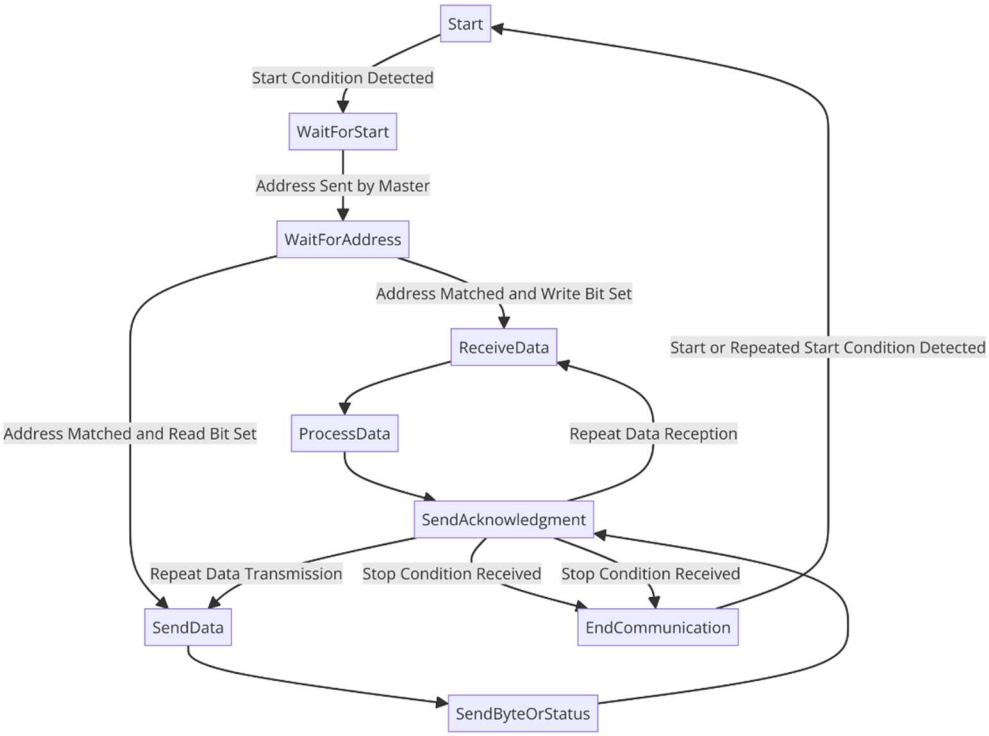


Figure 7.7.1.2: Sequence of actions of I2C state machine

To manage the various conditions and operations in the I2C communication protocol, a state machine was implemented. This state machine comprises different states, each corresponding to a specific condition or operation mentioned above. By transitioning between these states based on the sequence of events on the I2C communication line, the system effectively controls the communication process. The detailed sequence of actions of the state machines is displayed in Figure 7.7.1.2.

Analog input

Components that are analog inputs only send analog signals to the Arduino. There is no signal coming out of the Arduino to such components. Arduino reads the signal after it is converted from analog to digital.

Analog to digital conversion is a crucial process in microcontroller-based systems. It enables the conversion of continuous analog signals into discrete digital values that can be processed by the microcontroller. For Arduino Uno, there is a built-in 10-bit Analog to Digital Converter (ADC). It divides the reference voltage, usually 5V, into 2^10 or 1,024 distinct steps. Therefore, signals within the range of 0-5V can be calculated by the formula below and sent from component to the Arduino.

$$Analog\ Value\ = Step\ Value \times \frac{Reference\ Voltage}{Maximum\ Steps}$$

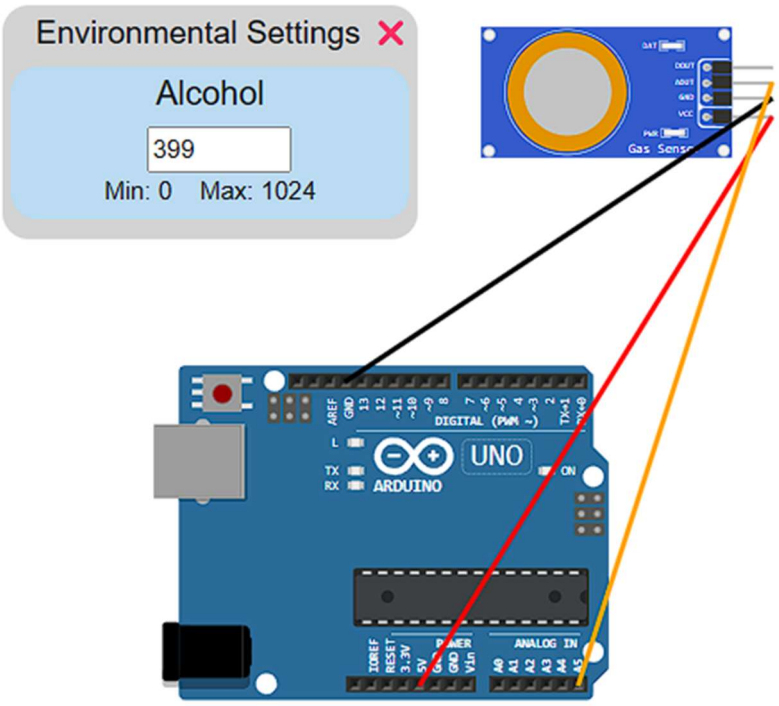


Figure 7.7.1.3: Circuit of Arduino and MQ-3 with Environmental Settings

The values, as results of the conversion, are stored in the analog-read registers in the Arduino Uno. We have direct access to those registers in the simulation. We simulated analog input by allowing the components to directly send the voltage value to the registers. For example, if a MQ-3 alcohol sensor is connected to Arduino and the value it senses is set to 399, the translation function will calculate the voltage value of $399 \times \frac{5}{1024} = 1.95V$ and send this to the analog-read registers. It will be ready to be read using the analogRead() function (Arduino, 2023).

Other communication protocols

There are other components that have unique communication protocols. We translated those communications by using similar approaches as I2C and PWM. All communications of current

components can be translated by the combination of measuring pulse width and state machines. For example, the DHT22 humidity and temperature sensor has its unique communication protocol through a single data line. The Data line at default is maintained at a high voltage level. When the MCU initiates communication with the DHT22, it lowers the Data line voltage from high to low. This shift must persist for at least 18ms to ensure the DHT22 detects the MCU's signal. Afterward, the MCU raises the voltage again and waits for 20-40μs for the DHT22's response (D-Robotics, 2010, p. 6). This high-low-high pattern is the start signal sent out by Arduino indicating the start of the communication. It is recognized by the state machine displayed in Figure 7.7.1.4. The function "sendSignal" after the start signal completes the entire communication and resets the state to the initial state. The "sendSignal" function includes timing-critical actions that involve the "timing packet" system that will be discussed next.

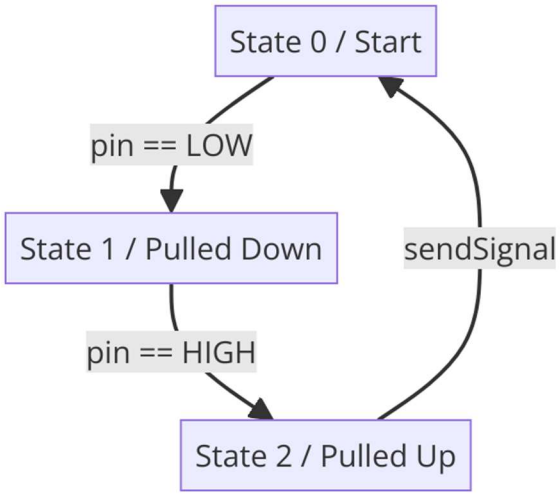


Figure 7.7.1.4: State Machine in DHT22's translation function

Timing-Packet

One issue we had while developing the simulation system was that the simulator ran alongside the C# code in a separate thread, which caused numerous issues when attempting to write translation functions for some communication protocols. For some components, time itself is used to encode and carry information. A discrepancy in timing would cause the signal to be unrecognized or result in incorrect data. Our first attempts to solve this involved attempting to use delays, but these resulted in varying degrees of failure due to threads blocking and, most importantly, inconsistent delays caused by interoperability functions. We tried using web workers to get a separate thread for consistent delay timings, but this was deemed unfeasible to set up, let alone whether it would work. In the end, we used a system called a "timing packet," a bundle of instructions sent with specific timings to implement those instructions. For example, a DHT's timing packet is composed of more than 40 instructions. If the starting cycle is 1000, the first instruction is to turn the pin off after 16 cycles of delay, at 1016. The second instruction is to turn the pin back off after 80 microseconds, or 1280 cycles, at cycle 2296. The origin of these numbers is described below.

A real-life Arduino runs on an "instruction set," defined as a list of the most basic operations the CPU can do. When a code script is compiled, it creates a binary representation of many these instructions. Then, when the script is run, the Arduino processes one instruction at a time. Each of these processes begins on a specific "clock cycle," which the Arduino does sixteen million times per second or a "clock speed" of 16MHz. In contrast, the Arduino simulator library that we are using, AVR8js, does each individual clock cycle and instruction manually in a loop. By preventing the loop from progressing the clock cycle, we essentially pause the Arduino's time. We achieve timing accuracy down to ⅓ of a

microsecond by “pausing time” when an outgoing signal requires a timed response. Note that “pausing time” refers exclusively to the simulated Arduino. The thread that hosts the Arduino progresses at normal speed, looping many times per second.

The final implementation calls for the system to remember the pins that require timed responses as they are computationally slower than other components. Following this, whenever the simulated Arduino changes the state of this pin, it waits for a response, not increasing the clock cycle count or executing any instructions until it receives the response. The response comes in the form of a “timing packet,” which contains the complete protocol and the clock cycle on which to execute each pin change. The pin changes are then executed on the specific intended clock cycle, achieving perfect timing.

7.8 Rules loading

We have added a progress bar at the bottom of the screen that appears when you first load the app. There are 86 rules, and they take less than ten seconds to load. The Component Palette and Workbench are disabled until all rules are completed; thus, the bar notifies the user when these functions unlock. Figure 7.8.1, shown below, displays the loading bar in the middle of its progress.

Figure 7.8.1 Progress Bar

7.9 Environmental Settings

Some components' state and values they sense in the environment can be altered by the user. For example, LEDs can have multiple colors and sensors such as the DHT22 can detect a diverse range of values, such as temperature and humidity. If the user wants the LED to blink once the DHT detects 25 degrees and 20% humidity, they can do so by adjusting these values directly in the Workbench by right-clicking a component, which will summon the context menu, from which they can open the Environmental Settings menu. If a component has no applicable settings, the menu appears empty. Otherwise, applicable settings will appear and can be altered, and the menu can be closed when no longer needed.

There are two types of settings: value-based and dropdown-based. Figure 7.9.1 shows a DHT with its environmental settings menu opened. In it, specific values (in this case humidity percentage and degrees Celsius) can be assigned to each parameter to simulate sensor measurements. Figure 7.9.2 shows the environmental settings for an LED, which has a dropdown with all available colors.

Internally, the Environmental Settings are configured by providing a property in the Component for the setting to affect, then labeling the property with the 'ValueSetting' or 'DropdownSetting' annotation. This is an opt-in system, requiring some setup to be effective, and it does not affect any other part of the system, only that component or its measured values. Generally, these variables are used in rendering or return measurements to the Arduino, so they do not change the connections or code. The annotations provide additional info about the setting for applicable components, such as the value's minimum, maximum, and unit. The name is parsed from the property's

name, while the starting value is provided. These values are passed to and from the ComponentInstance class through a property dictionary, which is done automatically. When values are altered, the console will display an updated output once the code is running.

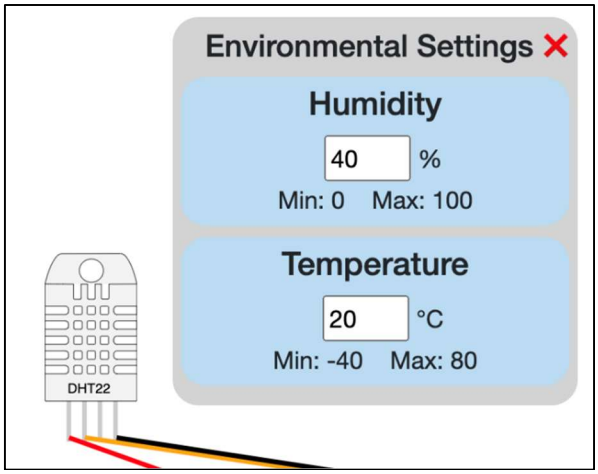


Figure 7.9.1 Environmental Settings Menu Example 1

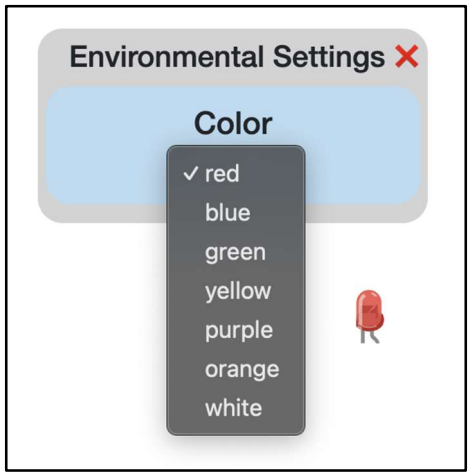


Figure 7.9.2 Environmental Settings Menu Example 2

7.10 Code Console

There were two goals of the console: to handle compiler output and handle Serial output. These each posed their own individual challenges. The basic structure of the console called for output to be separated to denote new lines. Compiler output was returned as a multiline string, meaning that the newline characters in the middle should not generate separators, as the output should be displayed together. Error messages were similar, calling for parsing to remove unnecessary file information and the creation of error lines in the editor part of the panel. On the other hand, Serial output was received one character at a time, calling for separators at newline characters. These issues were solved by introducing a buffer system that only adds a separator line when the buffer is completed, clearing the buffer for a new line.

The implementations described in this chapter are what allows a large application like ADArC to function. However, creating these required active review and testing, especially for the more complex features that were core parts of ADArC. This next chapter covers the testing processes for the wire connection, GraphSynth rules, and translation functions during development.

7.11 Deployment

To deploy the app to the internet, we used GitHub Pages. On our main GitHub, we created a branch called “gh-pages” that would contain the content to be deployed. Additionally, a GitHub script was used to compile the code and push it to the “gh-pages” branch whenever we added code to the main

branch. Another automated script then updated the online page whenever the "gh-pages" branch received changes.

This process spawned two different issues that we tangled with over time. The first issue was browser caching. This is a feature where the browser (Google Chrome, for example) saves a snapshot of a page's content to improve loading times and data usage. However, it also means that a user may not see the most up-to-date version of the application if they visit the site frequently. We were able to avoid the problem by using the "non-cache reload" shortcut, Ctrl-F5. This is not a solution, however.

The second issue was a mismatched development and deployment environment. In the main `index.html` file, there is a line to define the default route (line 10). In testing, this remained as a simple "/" for the default route, but it had to be changed when deployed, as the process added to the route. Thus, whenever the site was deployed, the line had to be changed to "/ADArCWebApp" to avoid errors on the live website.

The implementation described in this chapter is what allows a large application such as ADArC to function. However, creating this required active review and testing, especially for the more complex features that were core parts of ADArC. This next chapter covers the testing processes for the wire connection, GraphSynth rules, and translation functions during development.

8. Testing Process

Throughout the development of our application, manual testing has played a crucial role in ensuring its reliability and functionality. To validate the different features, we designed a simple baseline case, leveraging an Arduino Uno Rev3 microcontroller and an LED component. This served as a common baseline for testing all the major functions within our application.

In this chapter we will explain how GraphSynth rules, connection lines, and component translations were manually tested and debugged.

8.1 GraphSynth Rules

To test the rules with the GraphSynth integration in the ADArC application, we utilized the development console (such as the F12 console on a webpage or the console window in Visual Studio) for debugging and monitoring purposes. We displayed the number of nodes and connections in the development console every time a component was added to the Workbench. This process was not only crucial during the initial stages of development but was also implemented consistently in subsequent phases as part of our debugging system. It helped us determine whether the rules were contributing to any issues within the system. However, we did not maintain a detailed track record of these tests during the later stages of debugging, focusing instead on real-time problem resolution.

During the early stages of this testing process, we examined two specific cases. The first was the baseline case involving an Arduino and an LED. The second case, however, became invalid due to the rule changes outlined in Section 6.3.2. Therefore, this section is dedicated to showing the baseline case.

Specifically, it is crucial to note that the Arduino Uno Rev3 microcontroller adds twenty-seven nodes to the design graph, while the LED component contributes another four nodes.

Table 8.1.1: Common Baseline Case for GraphSynth Rules

Step	Action	Expected Outcome
1	Add Arduino Uno Rev3 microcontroller to workbench	27 nodes, 0 connections
2	Add LED component to workbench	31 nodes, 2 connections

The display of the number of connections was later removed because of the implementation of the connection lines, which represent wires in physical hardware.

8.2 Connection Lines

To assess the correctness of the generated connection wires, we examine the pins they are connected to and their colors. This crucial information is stored in the GraphSynth rules, as introduced in section 6.3.1. By referencing these rules, we can verify whether the generated connection wires align with the expected criteria, thus validating their correctness within the application.

Table 8.2.1: Common Baseline Case for Connection Lines

Step	Action	Expected Outcome
1	Add Arduino Uno Rev3 microcontroller to workbench	No line
2	Add LED component to workbench	A black wire connects GND on Arduino to Cathode on LED A red wire connects a GPIO pin on Arduino to Anode on LED

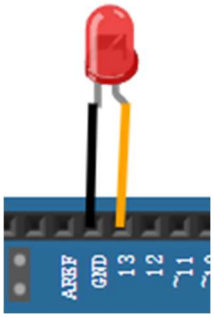


Figure 8.2.1: Desired Result of the Baseline Scenario - Detailed view of the connection pins

8.3 Translation Functions

Translation functions are a crucial part of enabling circuit simulation in our application. They translate the signal coming from the simulated Arduino microcontroller to make components behave correctly in the simulation. Therefore, to test the accuracy of the translation functions, we ran the simulation with each component and the Arduino microcontroller. The expected results were shown in the visual display of the component in the Workbench or in the Console.

We created translation functions for 14 components in total, as shown in Appendix G. Among those, we tested and refined LED, DHT22 humidity and temperature sensor, and MQ-3 alcohol sensor in their respective circuits built using ADArC. The components we tested have different types of translation functions, therefore we added two cases for DHT22 and MQ-3 in addition to the baseline case. Case 2 is a circuit built with the Arduino Uno Rev3 and the DHT22 humidity and temperature sensor. It served as an example for translation functions that utilized timing-packet. Case 3 is a circuit built with the Arduino Uno Rev3 and the MQ-3 alcohol sensor. It served as an example for translation functions that are implemented on analog input components. For Case 2 and Case 3, the environmental settings were modified during runtime to test if the output changed correctly.

Baseline Case:

Table 8.3.1: Steps for Baseline Case

Step	Action
1	Add Arduino Uno Rev3 microcontroller and LED to workbench
2	Run the simulation

Expected Result: LED blinking

Figure 8.3.1 shows the result of LED blinking.

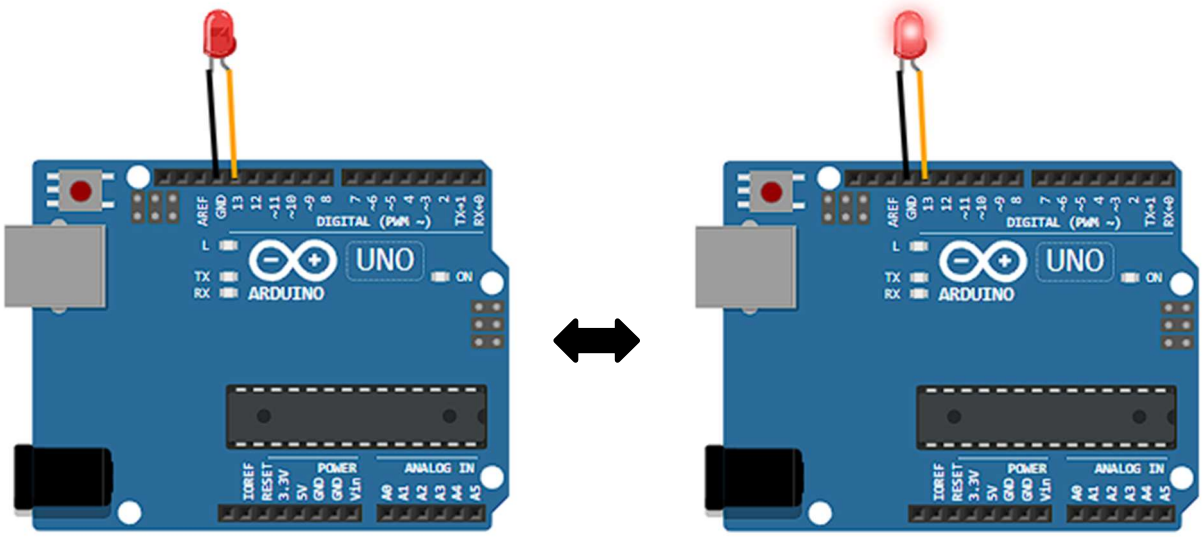


Figure 8.3.1: Screenshots of baseline case circuit running in ADArC

Case 2:

Table 8.3.2: Steps for Case 2

Step	Action
1	Add Arduino Uno Rev3 and DHT22 to workbench
2	Run the simulation
3	Change the environmental settings

Expected Result: Temperature and humidity values should be displayed in the console. Values should be the same as the ones in the environmental settings of the DHT22 sensor.

Figure 8.3.2 shows the result of Case 2. The output values on Console are consistent with input values on the environmental settings of DHT22.

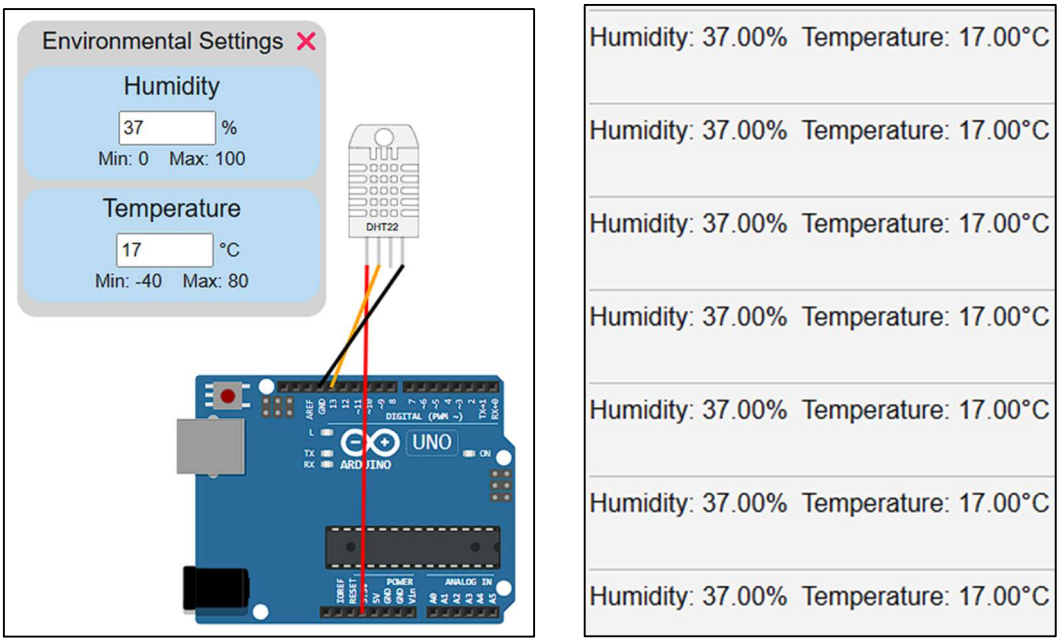


Figure 8.3.2: Circuit of Case 2 in ADArC (left) and Console output messages (right)

Case 3:

Table 8.3.3: Steps for Case 3

Step	Action
1	Add Arduino Uno Rev3 and MQ-3 to workbench
2	Run the simulation
3	Change the environmental settings

Expected Result: Alcohol value should be displayed in the console. The value should be the same as the one in the environmental setting of the MQ-3 sensor.

Figure 8.3.3 shows the result of Case 3. The output values on Console are consistent with input values on the environmental settings of MQ-3.

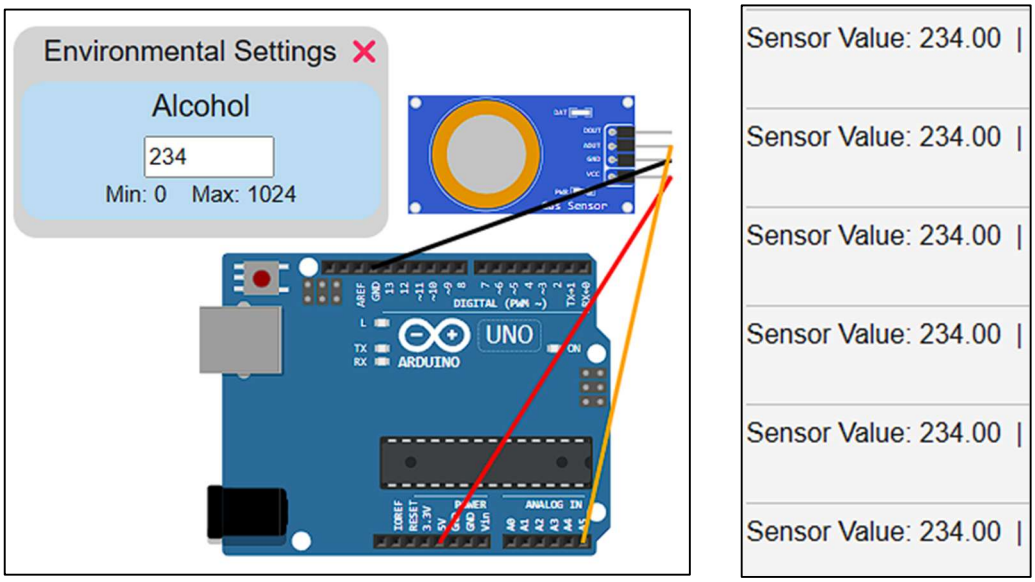


Figure 8.3.3: Circuit of Case 3 in ADArC (left) and Console output messages (right)

These testing processes described in this chapter helped us develop and refine our application through most our final term of development. The result was a functional prototype that we could showcase to a class at the end of C Term 2024. The following chapter presents our User Feedback Survey and the data analysis from it.

9. Data Analysis

In this chapter, we present the findings of a survey conducted with WPI students aimed at assessing the ease of use and effectiveness of ADArC, as well as gathering suggestions that could improve the app in the future.

Our team received an invitation from Professor Radhakrishnan to present our project in one of his classes, 'ME/RBE 4322 Modeling and Analysis of Mechatronic Systems.' In anticipation of this event, we designed a comprehensive survey for students to complete while using ADArC. The survey, found in Appendix A, was structured around five primary areas:

- Background student information
- Component Selection
- Coding & Simulation
- Exporting & Importing Files
- Final Feedback

We used a scale from 1 (easy) to 7 (difficult) for most applicable questions. We supplemented numeric data with short answer questions to gather additional comments regarding their experience with the application. The short answer questions had indications of what we were mostly looking for, such as the UI friendliness, if there was any lag, or whether they had any suggestions of features that would enhance the experience.

9.1 User Feedback Survey Results

In the first section of the survey, we asked about students' experience and proficiencies with similar circuit-building applications. 92% of students in the surveyed ME/RBE class were Mechanical Engineering majors, as expected. Even though the class was composed of mostly Juniors and Seniors, 48.4% considered themselves novices at bread-boarding, while 51.6% considered themselves vaguely familiar with Arduino boards. This information was surprising and reminded us of how important features such as the tooltips and the tutorial are for students using this kind of application.

In the Component Selection section, we discovered that most users found it easy to find their desired components (Figure 9.1.1). Most issues with finding components came from either a lack of understanding of how the components are categorized or how they were named. However, most components matched what students have seen and used in the past. Most features we implemented, especially our component manipulation (Figure 9.1.2), automated wire connections (Figure 9.1.3), and dedicated help pages (Figure 9.1.4) were received well. In their final comments about the Component Palette, some requested a search bar that would consider multiple naming conventions. Others suggested that components could appear in multiple categories. In the following figures, 1 is the lowest score and 7 the highest score (represented in the x-axis), and the number of people is shown on the y-axis.

Rate how easy it was to find your desired components.

29 responses

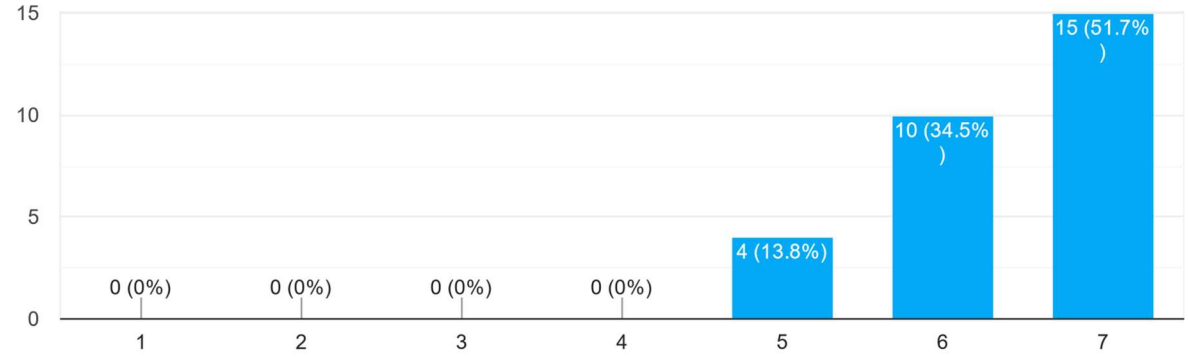


Figure 9.1.1 Results for Ease of Component Access

When a component is selected, how easy did you find it to manipulate it and delete it on the Workbench?

29 responses

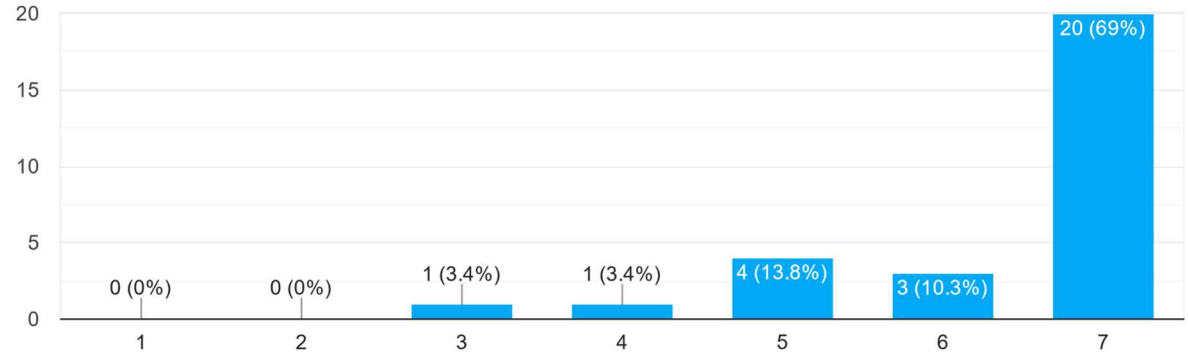


Figure 9.1.2 Results for Component Manipulation

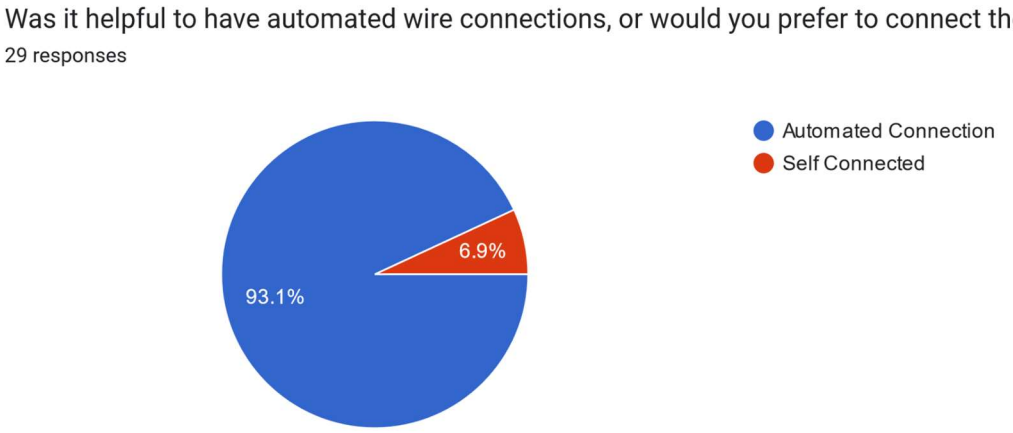


Figure 9.1.3 Results for Automated Wire Connections

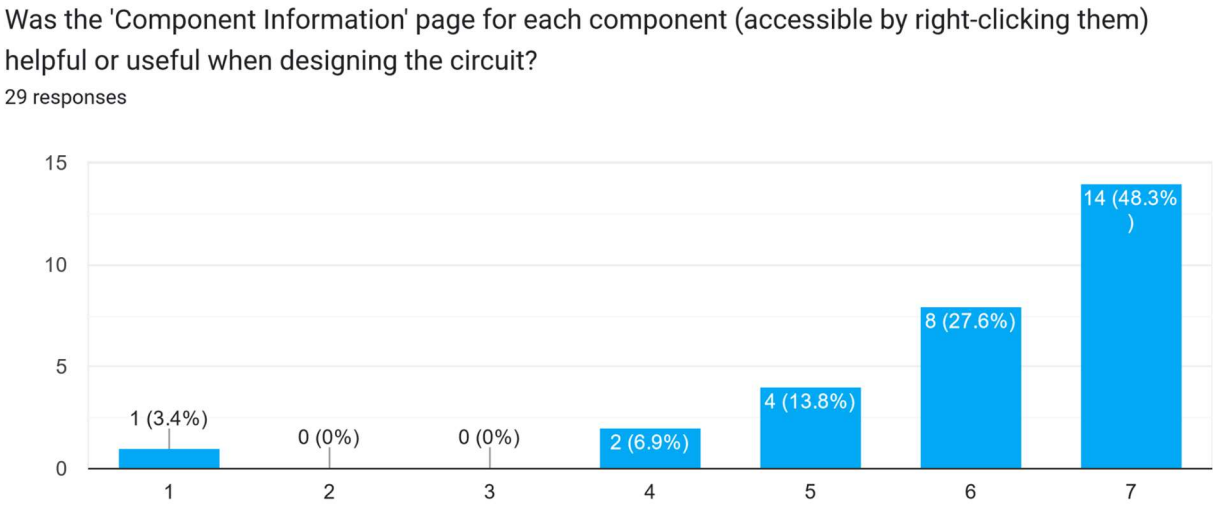


Figure 9.1.4 Results for Component Information

The Coding and ~~&~~Simulation sections of our app ~~was~~were ~~another~~ importan~~tpivotal section~~, and while we had an overall good review for it, user feedback shows that it can be more polished. The system proved to be visually intuitive (Figure 9.1.5). However, environmental sliders had a ~75% approval when it came to simulating real-world conditions (Figure 9.1.6). Code generation was received with moderate success, with most ratings being around an average of 5 (Figure 9.1.7). Nevertheless, we discovered that it could still contain more extensive comments to guide newer users.

To what extent did the simulation prove to be visually intuitive (output on the Console after changing environmental settings on the DHT22)?

29 responses

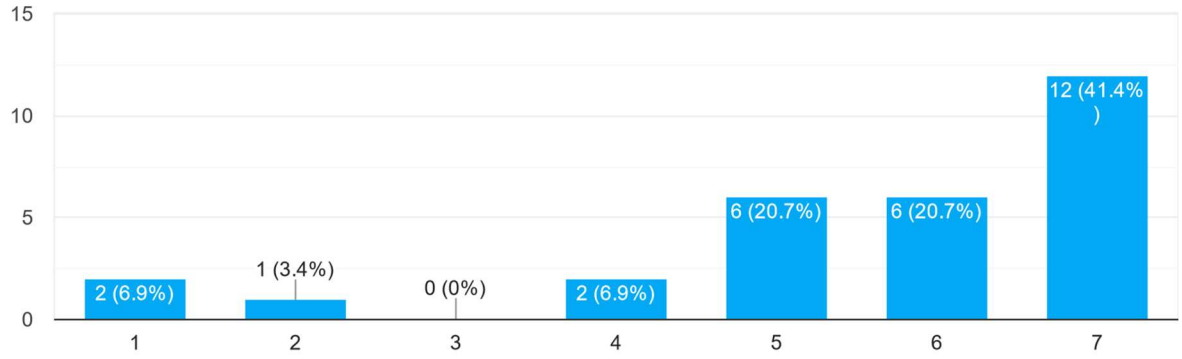


Figure 9.1.5 Results from Question on Visual Intuitiveness

Were the environmental settings helpful in simulating real-world conditions?

29 responses

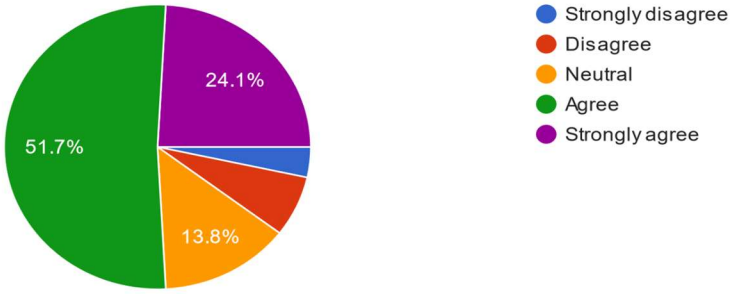


Figure 9.1.6 Results for Environmental Settings

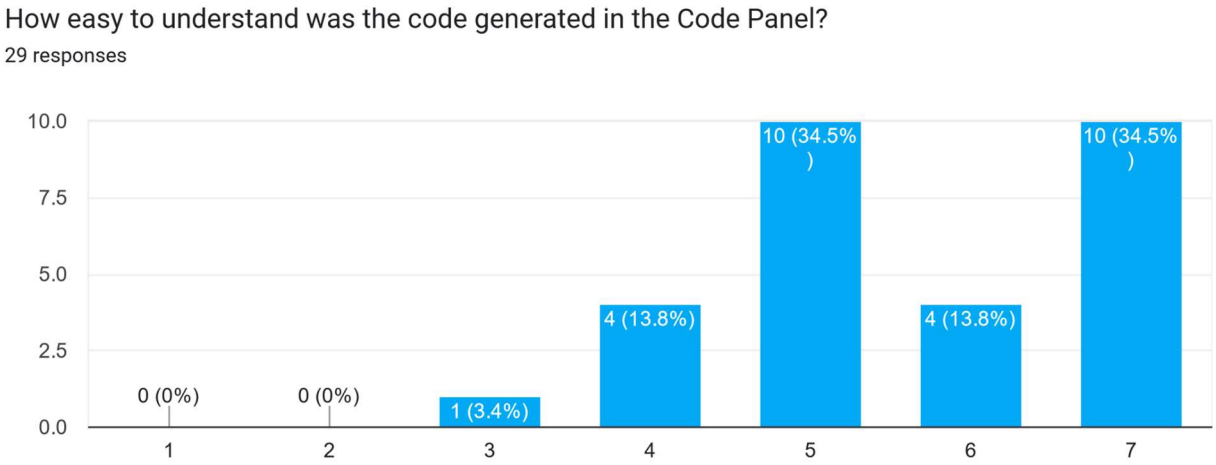


Figure 9.1.7 Results from Code Generation

The feedback on the Exporting & Importing Files section shows that students found these features useful, albeit buggy.

The last section focused on the student's overall experience with ADArC. While we had high levels of satisfaction, 13.8% of the class wouldn't use ADArC over other circuit simulators (Figure 9.1.8), and 20.7% does not have enough experience with similar applications to know if this one is a better option.

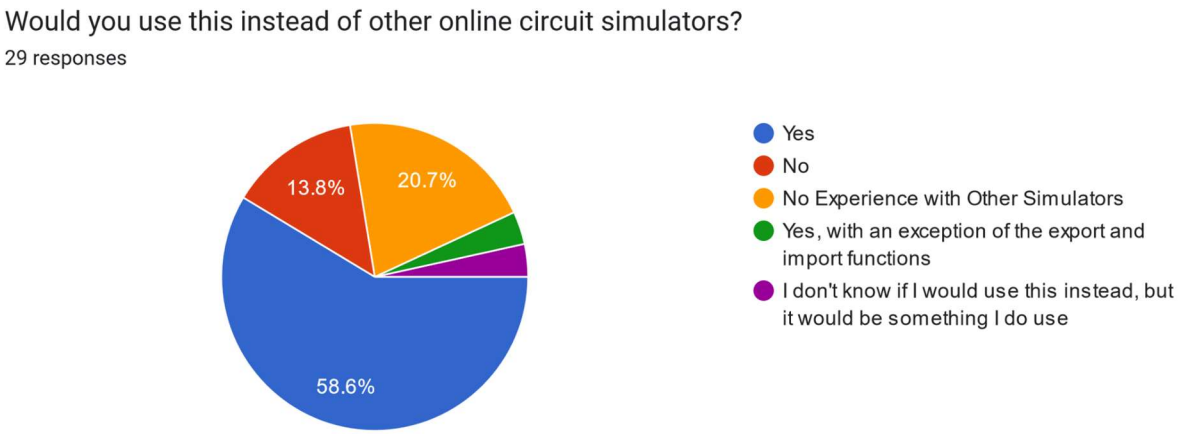


Figure 9.1.8 Results from Circuit Simulator Preference

Overall, the survey was very successful, yielding essential information for future changes. Our team got together at the start of the following term to discuss and record these suggestions. Our team implemented a few of the small-scale changes and bug fixes, and left suggestions for implementing some of the bigger changes in section 11.1: Future Work.

10. Discussion

When we began this project, we set out to develop a circuit builder, code generator, and simulator for Arduino-based microcontroller applications for novice RBE and ME students. Based on the feedback we have received from the User Feedback Survey, our team feels that we have adequately satisfied this goal.

Our first requirement to achieve our goal, mentioned in Table 4.1, was to create a circuit from individual parts automatically. With some assistance from ADArC, all the user needs to do is drag and drop a component from the Component Palette onto the workbench and ADArC handles the rest of the pin connections for novice users. The effectiveness of this tool is shown through Section 2 of the ADArC User Feedback Survey, found in Appendix A. Students gave overwhelmingly positive comments on ADArC's fulfillment of this category, based on Figures 9.1.1, 9.1.2, and 9.1.3.

Our second requirement for this goal was to automatically generate code based on their chosen components. This was handled by breaking preexisting sample code into categories that can be filled when the user adds a component to the workbench, as explained in Chapter 7.5. These code snippets contained comments to help users with the coding aspect of their projects. When tested with a class, students responded with a more moderate level of understanding of the code presented to them, as shown in Figure 9.1.7.

Our third requirement was to enable the user to create real-time simulations of their circuits. This was achieved by creating translation functions that could decode and utilize the Arduino's digital signals, as explained in Chapter 7.7. Some of these signals could be modified by users using each

component's environmental setting. This system had trouble decoding certain signals, which limited the number of functional components we could implement and display to the ME 4322 class. Despite this, we were able to display twelve fully functional sensors and actuators to the ME 4322 class. Students had a good understanding of what was happening during the onscreen simulation, as shown in Figure 9.1.5, and a majority found the environmental setting useful in simulating real-world conditions, as shown in Figure 9.1.6.

Two less important requirements were accessing the help module and sharing their current circuit with others. Both of these were minor requirements that could be achieved as we developed ADArC, but they were still important to increase the educational value of our project. We have included a button on the toolbar to help users access the help module's homepage. A similar link to specific component's help pages can be found under each of their environmental settings. Both links were praised by students for being helpful, but were often noted to be confusing to navigate, full of redundant information, or did not work. The circuit sharing feature was achieved using BoGL Web's method, saving the current state of the project as an .adarc file. This feature works, albeit with a few bugs that occur while restoring components to the proper placement.

One requirement we tried but failed to implement properly was allowing users to zoom in or out on the circuit diagram. Due to how the automated wires are rendered, different levels of zoom would often misplace the wires.

In summary, we feel we successfully achieved most of our primary goals with ADArC and left the groundwork to for the next team to complete the ones we did not have the time to fully develop; and despite facing challenges with some functionalities, the positive feedback and practical results

confirm the effectiveness of the project in enhancing the educational experience for RBE and ME students. This next chapter will conclude our project with suggestions for future work and reflections from the team.

11. Conclusion

In this chapter, we provide conclusions about ADArC's development, give suggestions for future work, and reflect on our performance and progress as team members.

11.1 Future Work

As we finished our work on the project, we identified several areas that could be improved either visually or technically. A brief overview is presented here, with more details in Appendix B. Some of what is presented are bugs, while others are feature suggestions resulting from the survey, and still others are suggestions based on our own experience.

The essential new features to add in the future are zooming, a search bar, better code generation, and most importantly, more components. All of these were identified as issues by the survey respondents and are relatively easy to develop. Another suggested new feature is an undo/redo system, though that would be more difficult to develop.

A feature worth updating in the future, that we did not have the time to look into, would be the help module. Currently, ADArC sends users to an outdated web page containing datasheets and summaries on many components students would typically use. Unfortunately, the help module suffers from being inconvenient to search through and poorly organized. One way to resolve this is by having members of a team, who are more familiar with the hardware side of circuit design, reorganize the webpage to match ADArC's categorization style. Another way to improve the help module would be

to include prerecorded videos to help students better understand how to use ADArC. These features would also improve the utility of the help module.

Some technical problems that are available for future improvement are: porting to React or other JavaScript frameworks; improving GraphSynth performance; providing other microcontroller support; and strengthening the backend system. Unfortunately, there was little time to refactor the project, so the backend is not easy to work with. Making improvements here would mean easier development in the future. Switching to React or other JS frameworks would eliminate the initial loading time and avoid the extremely slow interoperability calls present in the current project. Lastly, while GraphSynth worked for our purposes, we began running into its limitations. Finding a better solution to the problem of component connections would provide significant improvements to performance on medium to large circuits.

Lastly, there are several bugs left in the code. The most notable of these are the Workbench layer ordering placing the connection lines over the environmental settings box, and the component's size being wrong while being dragged. More details are provided in Appendix B, with a list of known issues appearing in Appendix C.

11.2 Reflection

Yangyang Jin:

The skills that I acquired from my ECE Classes were critical in my ability to work effectively on this project. These courses provided me with foundational knowledge in electronics, specifically how to

read and interpret datasheets for electronic components, and detailed insights into Arduino microcontrollers. Such expertise played a crucial role in understanding the technical specifications essential for working with the Arduino microcontroller throughout the project. Additionally, my coursework in CS 2301 Systems Programming proved invaluable as it familiarized me with the basics of C/C# programming languages, which were crucial for completing various coding tasks related to the project.

As the project progressed, I also acquired new skills, including learning how to manage parallel branches effectively in GitHub, which improved collaboration and version control. I gained a better understanding of GraphSynth usage and graph grammar, expanding my knowledge of graph-based systems. Additionally, I improved my writing skills for clear technical communication and developed my coding abilities for handling complex tasks within the ADArC application. Overall, this project was instrumental in my growth and skill development.

Gabriel Buziba:

My role as an ECE major helped pay off when it came to designing systems or tools used for students designing circuits. The knowledge I gained during my time studying the microelectronic track helped me design and build accurate components that students would use. My knowledge of component functionality aided our team in design considerations to maintain accuracy. This played a critical role, especially at the beginning of the project where we needed to organize all the datasheets and information handed over to us by Tu Le. As the project progressed, I gained a lot more skills in time management. Most of my classes allowed me to finish my work within the dedicated hours, so this was

the first real project that had me making more time to work on it. I also was able to sharpen my coding skills for developing systems like this. Most of my coding knowledge had been used to work on embedded or digital systems so working on something more open-ended like this really honed my coding skills.

Andres Negron:

Throughout my CS journey at WPI, I've mostly focused on UI design. I've gained a lot of valuable experience being advised by Professor Brown with everything UI and UX related; from choosing the app's color palette, to the organization of panels, to effectively conveying information through phrases and icons. On the other hand, I learned a lot about what is needed to produce a web application from scratch and how different parts of the code that seem separate work together to display the finished product. The structured nature of the project, spanning three terms, has imparted a sense of professionalism akin to real-world scenarios, teaching me the importance of routine collaboration within a team setting rather than treating the project as a fast-paced homework assignment like I am used to with my other classes.

Casey Wohlers:

Coming into this project, I had little experience in developing full and deployable applications. Most of my prior experience came from a Software Engineering class, where I was not in any position of leadership. I gained a lot of experience in almost every facet of the development process, from market research to coding to deployment options. Along the way, I learned a lot about planning, timing, and

management. Overall, I made more gains in these soft social skills than in the harder coding skills. However, I also gained a lot of experience in .NET development. I had used C# previously for other things but using it for web development though .NET was a great learning experience. I also learned a lot about UI design, even though I mostly focused on backend development. Lastly, I gained experience in using GitHub for version control. I know that many places in our app could be improved, but I am happy with what we have completed.

References

Ace (Ajax.org Cloud9 Editor). (n.d.). *Documentation for Ace (Ajax.org Cloud9 Editor).* http://ace.c9.io

Arduino Team. (2022A, April 11). *Arduino hardware.* Arduino. https://www.arduino.cc/en/hardware

Arduino. (2022B, October 6). *A000066-full-pinout.* Retrieved January 11, 2024, from https://docs.arduino.cc/resources/pinouts/A000066-full-pinout.pdf

Arduino. (2023) *analogRead()* https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/

Arduino. (2024, April 1). *Arduino libraries.* https://www.arduinolibraries.info/libraries.

Arduino. (n.d.). *SD.* https://www.arduino.cc/reference/en/libraries/sd/

AutoDesk. (n.d.). *Tinkercad* Retrieved from https://www.tinkercad.com/

CircuitO. (n.d.). *CircuitO.* Retrieved from https://www.circuito.io/

CodeMirror (n.d.) *Extensible Code Editor.* https://codemirror.net/

DesignEngrLab. (n.d.). *Graphsynth by DesignEngrLab*. GitHub. Retrieved January 11, 2024, from https://designengrlab.github.io/GraphSynth/

Developers, I. W. (n.d.). Inkscape. News. https://inkscape.org/about/

Figma. (n.d.) *Figma*. https://www.figma.com/

Franz, M., Lopes, C. T., Huck, G., Dong, Y., Sumer, O., & Bader, G. D. (2016, January). *Cytoscape.js: a graph theory library for visualisation and analysis. Bioinformatics*, 32(2), Pages 309–311. https://doi.org/10.1093/bioinformatics/btv557

Fritzing. (n.d.). *Fritzing.* Retrieved from http://fritzing.org/

Google. (2023). *Gemini*. https://gemini.google.com/

Guillaume Plique. (2023). *Graphology, a robust and multipurpose Graph object for JavaScript.* (0.25.4). Zenodo. https://doi.org/10.5281/zenodo.8204237

Hirzel, T. (2022, December 15). *Basics of PWM (Pulse Width Modulation)*. Arduino. Retrieved April 8, 2024, from https://docs.arduino.cc/learn/microcontrollers/analog-output/

Inkscape. (n.d.). *Inkscape.* Retrieved from https://inkscape.org/

Jacomy, A., Divol, D., & Heymann, S. (n.d.). *sigma.js*. GitHub. https://github.com/jacomyal/sigma.js

Le, C. (2021A). *An Automated Design Tool for Sensor and Actuator Integration in Mechatronic Systems.* MS, Worcester Polytechnic Institute.

Le, C. (2021B). *ADArC Help Website.* https://adarc-help.mech.website/

Mehrabani, A. (2023). Intro.js documentation. Intro.js Docs. https://introjs.com/docs

Medv, A. (n.d.). *CodeJar – an embeddable code editor for the browser.* https://medv.io/codejar/

Meta Platforms. (n.d.). *React*. Retrieved from https://reactjs.org/

Microsoft. (n.d.). *ASP.NET*. Retrieved from https://dotnet.microsoft.com/

Microsoft. (2024, April 11). *Call JavaScript functions from .NET methods in ASP.NET Core Blazor*. https://learn.microsoft.com/en-us/aspnet/core/blazor/javascript-interoperability/call-javascript-from-dotnet?view=aspnetcore-8.0

Microsoft. (n.d.). *Monaco Editor.* https://github.com/microsoft/monaco-editor

OpenAI. (2021). *ChatGPT*. https://openai.com/chatgpt

OpenJS Foundation. (n.d.A). *Node.js*. Retrieved from https://nodejs.org/

OpenJS Foundation. (n.d.B). *Electron*. Retrieved from https://www.electronjs.org/

PicSimLab. (n.d.). *PicSimLab.* Retrieved from SourceForge. (Download link: https://sourceforge.net/projects/picsim/)

Proteus. (n.d.). *Proteus*. Retrieved from https://www.labcenter.com/

Shaked, U. (2020, January 20). *Recreating The Arduino Pushbutton Using SVG And <lit-element>. Smashing Magazine*. https://www.smashingmagazine.com/2020/01/recreating-arduino-pushbutton-svg/

Shaked, U. (2023). *AVR8js.* GitHub. https://github.com/wokwi/avr8js

Shaked, U. (2024). *Wokwi Elements.* Github. https://github.com/wokwi/wokwi-elements

Shaked, U. (n.d.). *Avr8js Serial.* StackBlitz. https://stackblitz.com/edit/avr8js-serial?file=index.ts

Spasojević, M. (2022, May 23). *How to Call JavaScript Functions with C# in Blazor WebAssembly.* CodeMaze. https://code-maze.com/how-to-call-javascript-code-from-net-blazor-webassembly/

Valdez, J., & Becker, J. (2015, June). *Understanding the I2C Bus.* Texas Instruments. Retrieved from https://www.ti.com/lit/an/slva704/slva704.pdf?ts=1713893308416&ref_url=https%253A%252F%252Fwww.google.com%252F

Virtual Breadboard. (n.d.). *Virtual Breadboard.* Retrieved from https://www.virtualbreadboard.com/

Vuolo, A., Misbach, J., & Earnest, M. (2023). *Convert a Windows-based software (BoGL) to a Web-based system.* (Undergraduate Major Qualifying Project No. E-project-042423-125013-104616). Retrieved from Worcester Polytechnic Institute Electronic Projects Collection: https://digital.wpi.edu/concern/student_works/8336h5121?locale=en

Wokwi. (n.d.). *Wokwi.* Retrieved from https://wokwi.com/

Appendices

Appendix A - ADArC User Feedback Survey Questions

Below is a transcript of the ADArC User Feedback Survey we distributed to Professor Radhakrishnan's ME/RBE 4322 class on February 28th, 2024. It was used to evaluate the performance of our project. The response types have been shortened for brevity.

Section 1. Introduction

Purpose of Survey: The purpose of this survey is to review the user experience with our circuit simulator and test its effectiveness in a class setting. Your responses and names will stay confidential in all records and presentations.

Expectations from Students: You will be asked to answer some questions based on your experience with our app. You will be asked to take screenshots to capture any bugs or issues with the application. Be prepared to time yourself when you enter section 2.

Q1. What's your first and last name?

- *Short Answer*

Q2. What is your major(s)?

- *Short Answer*

Q3. What is your year?

- Freshman
- Sophomore
- Junior
- Senior
- Graduate

Q4. How experienced are you at bread boarding or circuit wiring?

- No Experience
- Novice
- Intermediate
- Experienced

Q5. How experienced are you at using Arduino or other micro-controllers?

- No Experience
- Novice
- Intermediate
- Experienced

Q6. Have you used any of the following Circuit Simulators?

- Wokwi
- Circuit.io
- TinkerCad
- Fritzing
- MultiSim
- Simulink
- None of the Above

Section 2 Component Selection

INSTRUCTIONS

1. Open the following link on another tab, this is the ADArC App: https://automatedcircuitgenerator.github.io/ADArCWebApp/
2. Please drag and drop the components listed in the project below onto the Workbench (the whitespace at the center of the screen) from the Component Palette (on the left of the screen, divided in categories). You should see them automatically connect to the Arduino.
3. To delete a component, select it and then select the "Delete Component" button. (Remove cause of tutorial?)
4. To consult help about that component, select a component and then select the "Component Information" button. (Remove cause of tutorial?)

Components List: Create circuits with the following components

- LED
- DHT22 Temperature and Humidity Sensor
- Seven Segment Display
- MQ-3 Alcohol Sensor

Please time yourself as you go through the design of your project

Q7. Rate how easy it was to find your desired components. (1-7)

- 1) Difficult
- 7) Easy

Q8. Explain what made finding components easy or difficult.

- *Long Answer*

Q9. How well did the components on screen resemble what you would see in the class? (1-7)

- 1) Inaccurate
- 7) Accurate

Q10. Were there any components that didn't match your expectations (visually, in functionality, etc.)?

- *Long Answer*

Q11. When a component is selected, how easy did you find it to manipulate it and delete it on the Workbench? (1-7)

- 1) Difficult
- 7) Easy

Q12. When a component is selected, how easy did you find it to manipulate it and delete it on the Workbench?

- Automated Connection
- Self Connected

Q13. Was the 'Component Information' page for each component (accessible by right-clicking them) helpful or useful when designing the circuit? (1-7)

- 1) Not Helpful
- 7) Helpful

Q14. Any general comments on this section? (Component drag & drop, visual design, help). Are there any components you'd like to see included in the app?

- *Long Answer*

Screenshot of your progress in the app so far

- *Add file*

Section 3. Coding & Simulation

INSTRUCTIONS

1. Upon dragging the components onto the Workbench, you can access the Code Panel by clicking the arrow enclosed in a blue semi-circle on the right side of the screen. Here you'll find corresponding code snippets for each component.
2. Run and compile the code by selecting the "Run" button located at the top right corner of the screen. This action initiates the simulation of the circuit. To halt the simulation, simply click on the "Stop" button in the same location.
3. Once the simulation is active, you can toggle Environmental Settings for each component by right-clicking the desired component and then clicking the "Environmental Settings" button. If that component does not have applicable settings, the menu will appear empty. For this case study, only the DHT22 has Environmental Settings that can be modified.
4. You can close the Environmental Settings menu by clicking the "X" button.

Q15. To what extent did the simulation prove to be visually intuitive (output on the Console after changing environmental settings on the DHT22)? (1-7)

- 1) Not intuitive
- 7) Intuitive

Q16. Were the environmental settings helpful in simulating real-world conditions?

- Strongly disagree
- Disagree
- Neutral
- Agree
- Strongly agree

Q17. How easy to understand was the code generated in the Code Panel? (1-7)

- 1) Hard
- 7) Easy

Q18. Any general comments on this section? (code, environmental settings, import/export)

- *Long Answer*

Screenshot of your progress in the app so far

- *Add file*

Section 4. Exporting & Importing Files

INSTRUCTIONS

1. After playing around with the code, simulation and environmental settings, click the "Export" button. An "adarc-circuit.adarc" file will be downloaded.
2. Open the app on another tab so your Workbench is empty, then click the "Import" button, select that file and your previous circuit and code will be recreated.

Q19. How useful do you find this feature? (1-7)

- 1) Not useful
- 7) Useful

Q20. Comments (specifically, did you find any discrepancies in your circuit or code before and after recreating it)?

- *Long Answer*

Q21. Screenshot of your progress in the app so far

- *Add file*

Q22. Upload '.adarc' file by clicking the Export button in the app.

- *Add file*

Section 5. Final Feedback

Feel free to experiment with the rest of the simulator!

For additional help with the components, click the "Help" button at the top right of the screen.

Q23. How satisfied are you with the circuit simulator? (1-7)

- 1) Dissatisfied
- 7) Satisfied

Q24. Would you recommend it to fellow classmates and friends?(1-7)

- 1) Not likely
- 7) Likely

Q25. Would you use this instead of other online circuit simulators?

- Yes
- No
- No Experience with Other Simulators
- Other...

Q26. Would you use this in a class?

- Yes
- No
- Other...

Q27. Could you provide feedback on the effectiveness of the tooltip pop-ups? Were they informative, did they cover all necessary information, did they appear redundant, and were they perceived as intrusive or disruptive?

- *Long Answer*

Q28. Did the categorization of components make sense?

- Yes
- No

- Other...

Q29. How friendly was the UI in terms of navigation? (1-7)

- 1) Difficult
- 7) Easy

Q30. Do you think new users would benefit from a tutorial/walk-through on the ADArC Website?

- Yes
- No
- Other...

Q31. How long did it take to build the circuit overall?

- *Short Answer*

Q32. How easy was it to understand the sample code generated in the Code Panel (after adding components on the Workbench)?

- 1) Difficult
- 7) Easy

Q33. How fast and responsive was the app overall?

- 1) Slow

- 7) Fast

Q34. Did you experience significant lag on any specific feature?

- *Short Answer*

Q35. Did you encounter any errors while building the circuit? If so, how many times did you encounter them?

- *Short Answer*

Q36. Any final feedback or comments?

- *Long Answer*

Thank you so much for your time testing out the app and for your very valuable feedback!

No screenshot required on this section.

Appendix B - Details on Suggestions for Future Work

This list is ordered by the complexity and approximate time it would take to complete each of these suggestions, from large to small. Some of the following are bugs, some are issues we ran into that troubled us, and others are simply suggestions.

Overhauls

- Completely port to React or similar JavaScript-based framework.
    - Reason: eliminates the need for JavaScript interoperability, a slowdown. Also consolidates the codebase.
    - Severity: Low
    - Classification: Suggestion/ran into
    - Notes: can use Wokwi elements without changes to save time.
- GraphSynth problems.
    - Reason: Poor UI when making rules, mediocre to low performance when connecting, rule loading times, massive slowdowns if connecting to a frequent label
    - Severity: Medium
    - Classification: Annoyance
    - Suggestions: find a new program, preferably in JS/TS, or make a new system (significantly more difficult)
- Other board support.
    - Reason: other board support was considered and dropped as too much work. Theoretically, avr8js supports making custom boards, we did not get into it.
    - Severity: Low/Needs Testing
    - Classification: Suggestion
    - Notes: This is more research than coding. Some Arduino mini variants use the same chip, so those should be fine on their own. It may be the case that the CPUs are similar enough on other boards not to cause problems.
- Reform the backend.
    - Reason: It is bloated and thoroughly unfortunately designed.
    - Severity: High
    - Classification: Strong suggestion
    - Notes: At the very least, implementing an abstract class to standardize individual components is a must. If possible, this should be rolled into the ComponentInstance class.

Probably, the ComponentNamespace and Builder classes should be removed in favor of some kind of database of some kind. Data is possibly fine to keep as templates, possibly not.
    - Creating a list of "things to do" when a component is added would be good. Then, by creating a function for each of those "things," component creation would be greatly simplified, as you would only need to provide the function args. This would also be a great help in removal and developing undo/redo.
- Undo/redo system.
    - Reason: Would probably help ease of use.
    - Severity: Very Low
    - Classification: Suggestion
    - Notes: See BoGLWeb's implementation for more info.

Features

- Search bar for the component palette.
    - Reason: one of the most requested features
    - Severity: Medium
    - Classification: Suggestion
    - Notes: One complaint from users is they did not know the technical name or they could not figure out the placement of components. May be mitigated by better sorting, but having it wouldn't be a bad idea.
- Better code generation.
    - Reason: a common complaint was code readability and functionality.
    - Severity: Medium/High
    - Classification: Strong suggestion
    - Notes: adding comment tags to each component would help a lot. Similarly, removing code from the window on delete would also help.
- Reimplement zoom.
    - Reason: Mostly complete already, just very buggy. We also got some requests for it on the survey.
    - Severity: Medium
    - Classification: Suggestion
    - Notes: None.
- More components!
    - Reason: stock is extremely limited.
    - Severity: Very high

- Classification: Necessity
- Notes: We ran out of time to bugfix these individually. Maybe you can find a better system or the time to finish this.

Bugs

- See 'Known Issues' (Appendix C) for more information.
- Primary issues:
  - Workbench component/line/Env settings ordering.
    - Cause: HTML layer ordering. Fixing the issue would require substantial changes to the way that one of them appears. Essentially, the components and settings are in the same layer, while the lines sit on top. If the components are on top, then the lines will be behind the components. This can be solved by using three layers or only one and using the CSS z-index property.
  - Component dragging image sizes off.
    - Cause: Unknown.

Appendix C - List of Known Issues

- Dragged component from the palette is a few pixels below the mouse.
- Drag scaling off, especially on the DHT.
- Lines appear over component info.
    - It has to do with layer ordering in MainCanvas.razor.

Appendix D: Component List

The full list can be found at https://docs.google.com/spreadsheets/d/13X_gPcTXqMcJu-CODgA-dwPAsBkhFZRd0L-eNx64T18/edit?usp=sharing, but it is also reproduced below.

Simple Category	Technical Name	GraphSynth rules	Wokwi elements
Accelerometer sensor	ADXL345	included	Not included
Accelerometer sensor	MPU6050	included	included
Accelerometer sensor	BNO-055	included	Not included
Audio Amp	LM386	included	Not included
basic electronics	laser led	included	Not included
basic electronics	Resistor	included	included
basic electronics	Potentiometer	Not included	included
basic electronics	photoresistor	Not included	Not included
basic electronics	transistor NPN	Not included	Not included
basic electronics	rectifier diode	Not included	Not included
basic electronics	pushbutton	Not included	included
basic electronics	switch	Not included	example included
basic electronics	relay	Not included	Not included
Bluetooth Module	HM-10	included	Not included
buzzer	ky012	included	Not included
DC motor		included	Not included
Display Module	LCD2004	included	included
Display Module	LCD2004 with I2C interface mounted	included	included
Display Module	LCD1602	included	included
Display Module	LCD1602 with I2C interface mounted	included	included
Display Module	SH1106 OLED SPI	Not included	Not included
Display Module	SH1106 OLED I2C	Not included	Not included
Distance sensor	TF-Luna Lidar	included	Not included
Distance sensor	RPLIDAR-A1 Lidar	included	Not included

Distance sensor	GP2Y0A21YK0F IR Distance Sensor	Not included	Not included
Encoder	E6B2-CWZ3E	included	Not included
gas sensor	MQ-3	included	example included
gas sensor	SGP41	Not included	Not included
Hall effect sensor modules	KY-024	included	Not included
Hall effect sensor modules	KY-003	included	Not included
Hall effect sensor modules	KY-004	Not included	Not included
IR receiver	KY-022	included	example included
LED		included	included
LED 7-segment	Adafruit 0.56"	Not included	example included
LED bar	Adafruit Bi-Color 24 Bargraph	Not included	Not included
LED matrix modules	MAX7219/MAX7221	Not included	Not included
LED matrix modules	HT16K33	Not included	Not included
LED RGB		included	included
LED RGB module	KY-009	Not included	example included
light sensor module	KY-018 LDR sensor	included	example included
light sensor module	BH1750 Ambient Light Sensor	Not included	Not included
light sensor module	TSL2561 Luminosity Sensor	Not included	Not included
light sensor module	TEMT6000 Ambient Light Sensor	Not included	Not included
Load Cell	TAL221	included	included
Load Cell Amplifier	HX711	included	included
microcontroller	Arduino Uno Rev3	included	included
microcontroller	Arduino MEGA 2560	Not included	included
microcontroller	Arduino Uno WIFI Rev2	Not included	Not included
motor driver	ULN2003	included	Not included
motor driver	A4988	included	Not included
motor driver	L298N	included	Not included
motor driver	PCA9685	included	Not included

motor driver	DRV8825	included	Not included
motor driver	DRV8833	included	Not included
motor driver	L293D	Not included	Not included
motor driver	L9910	Not included	Not included
motor driver	TB6600	Not included	Not included
motor driver	TB6612	Not included	Not included
motor driver	TMC2208	Not included	Not included
Motor Shield	L293D	Not included	Not included
PIR motion sensor	HC-SR501	included	example included
sdcard reader		included	included
servo motor	SG90	included	example included
servo motor	DS-7001HV	included	example included
stepper motor	28BYJ-48	Not included	Not included
stepper motor	Nema-17	included	included
temperature sensor	DHT11/DHT22	Not included	included
temperature sensor	SHT31 Temperature & Humidity Sensor	Not included	Not included
temperature sensor	DS18B20	included	Not included
Thermocouple Amplifier	MAX6675	included	Not included
Ultrasonic sensor	HC-SR04	included	included
Voltage regulator	LM2596	included	Not included
	ESC	included	Not included
speaker		included	included

Appendix E – Tutorial Popups

This contains the list of popups the user will receive when selecting the Tutorial Button. This list is in order of how the user designs a circuit in ADArC. Pop-Ups currently included in the system are shown in black, while those not yet implemented are shown in grey.

Welcome to ADArC!

- ADArC stands for Automated Design tool for Arduino Circuits! This is a circuit design tool for building circuits with Arduino Microcontrollers.

The Toolbar

- This toolbar contains tools to save and continue your work, find component information, and run a simulation of your circuit.

Zoom Buttons

- These are the Zoom buttons. The buttons will allow you to change the size of the view of your circuit: helping you see how each component is connected.

Export & Import Buttons

- These are your Export and Import buttons. They are used to save your current circuit and upload your past circuit as .adarc files.

The Help Button

- This button will take you to the dedicated help website, where you can research the datasheets and the uses of each of the available components.

The Run/Compile Button

- This button will compile your Arduino code and start simulating your circuit. This should be selected when you are finished building the circuit.

The Three parts of ADArC

- In order to build and run a simulation of your circuit, ADArC provides three panes: the Component Palette (on the left side), the Workbench (in the center), and the Code Panel (on the right side).

The Component Palette

- This is the Component Palette. Here you can choose from our library of components to create your circuit. Just drag your components to the Workbench, and all the wires will be automatically connected to the Arduino Microcontroller.

The Workbench

- This is where your circuit is built and simulated. Selecting and right-clicking on components will open up additional tools that will aid you in building your circuit.

The Code Panel

- This is where you can inspect and modify the code produced for the Arduino simulation. It follows the same design as the Arduino IDE and contains sample code for each component on the Workbench. If the code isn't what you need, then edit it to suit your goals.

Appendix F – Rule Changes due to combining components

The table below lists the combined components that are treated as single entities.

Table F1: Combined components

servo_pca9685
speaker_lm386
dc_motor_l298n
load_cell_hx711
stepper_nema_l298n
stepper_nema_a4988
servo_9g_pca9685
esc_pca9685

servo_pca9685

Original rules:

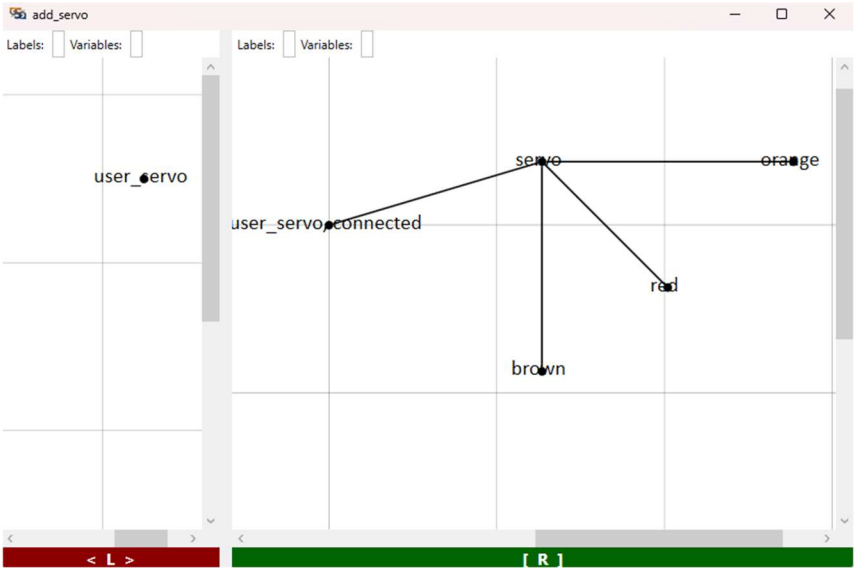


Figure F1: Original *add_servo.grxml* Rule

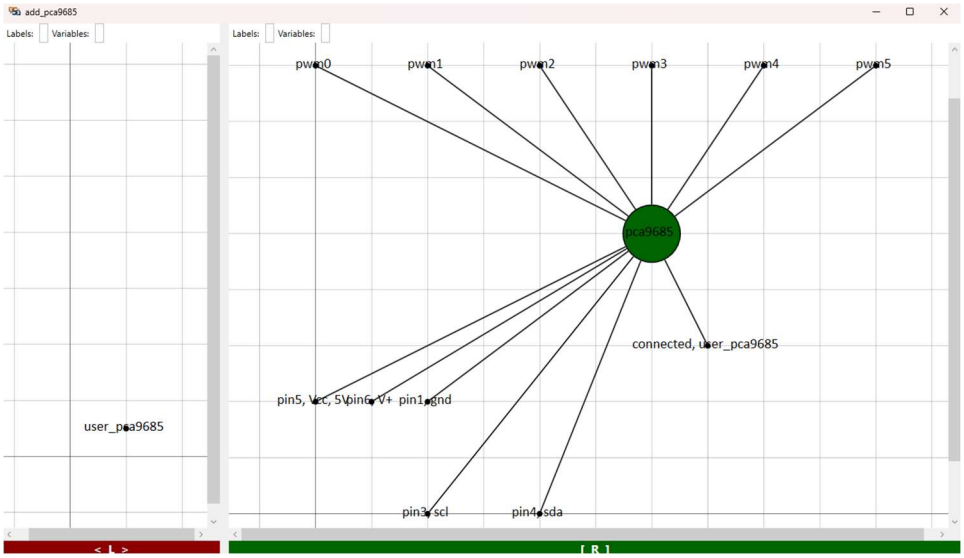


Figure F2: Original *add_pca9685.grxml* Rule

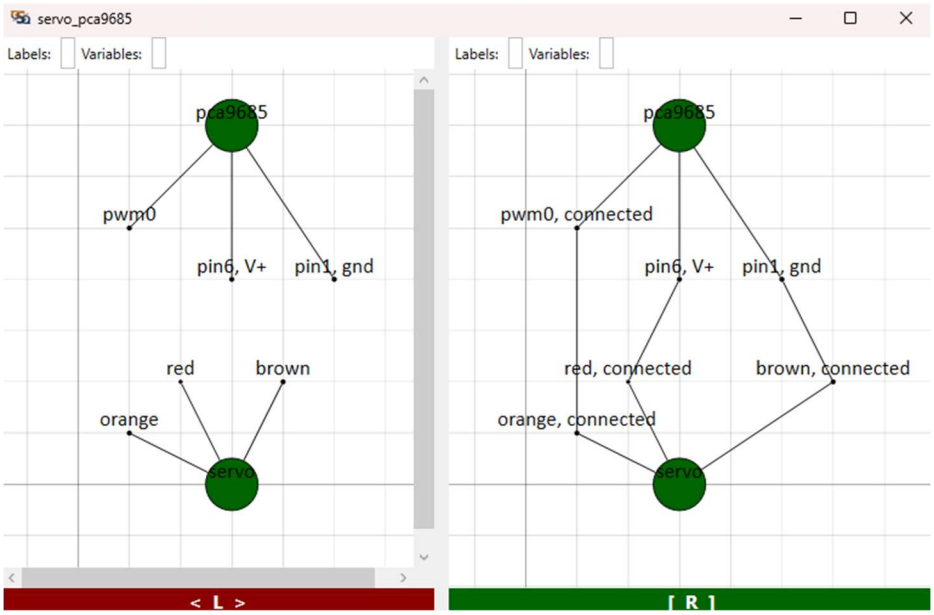


Figure F3: Original *servo_pca9685.grxml* Rule

New rule that combined the original rules:

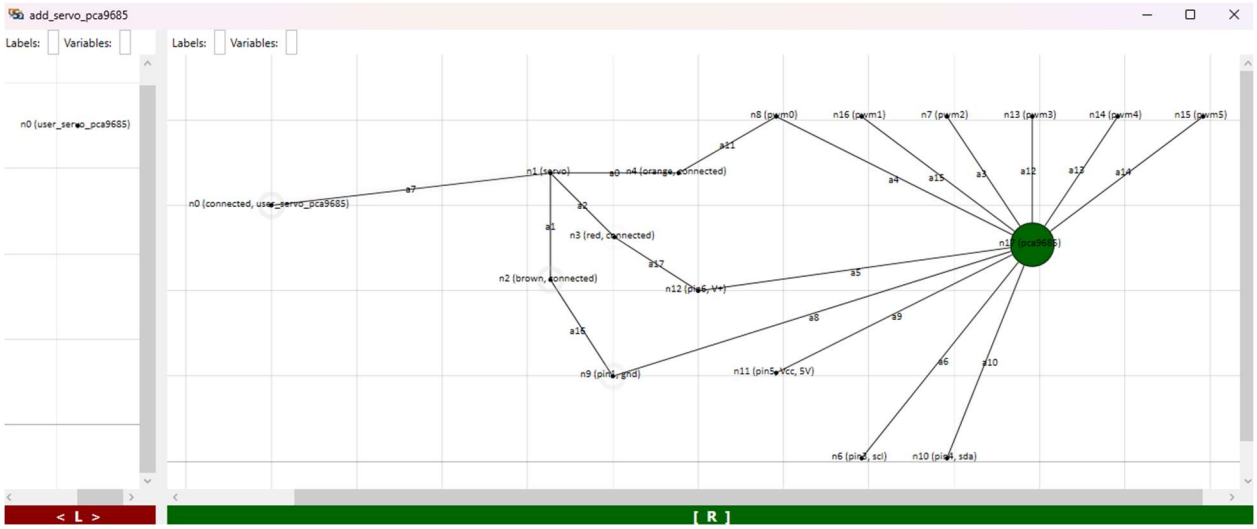


Figure F4: New *add_servo_pca9685.grxml* Rule

speaker_lm386

Original rules:

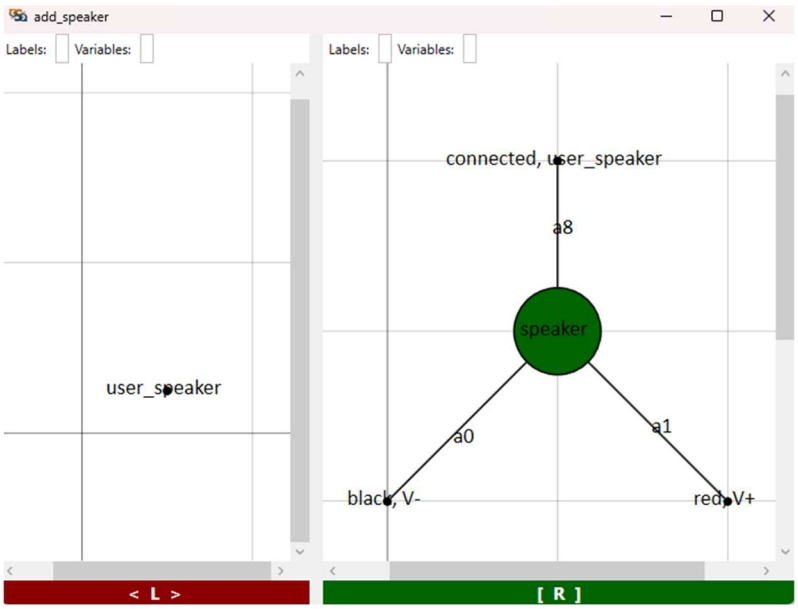


Figure F5: Original *add_speaker.grxml* Rule

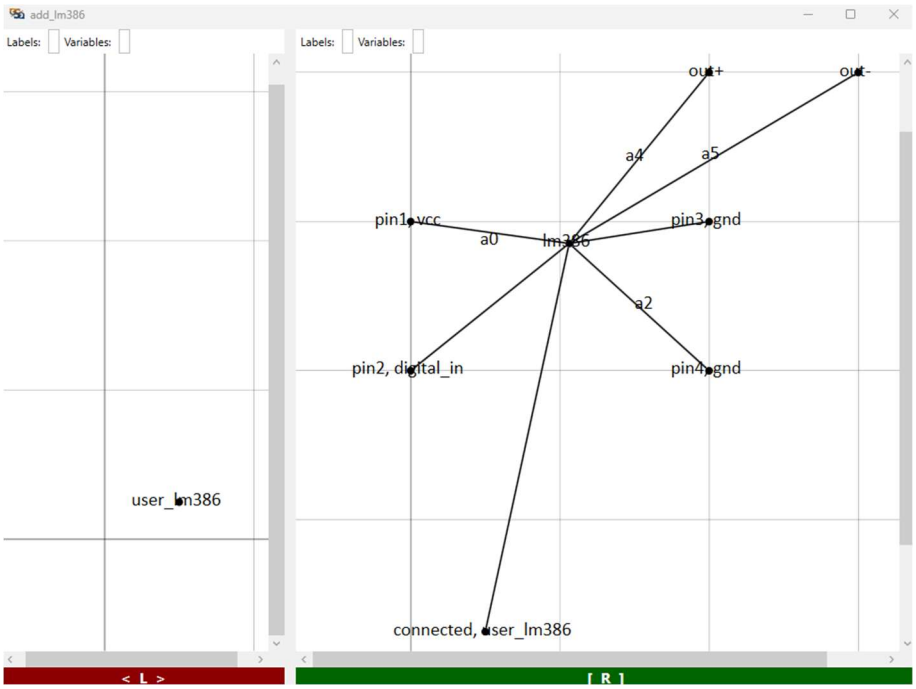


Figure F6: Original *add_lm386.grxml* Rule

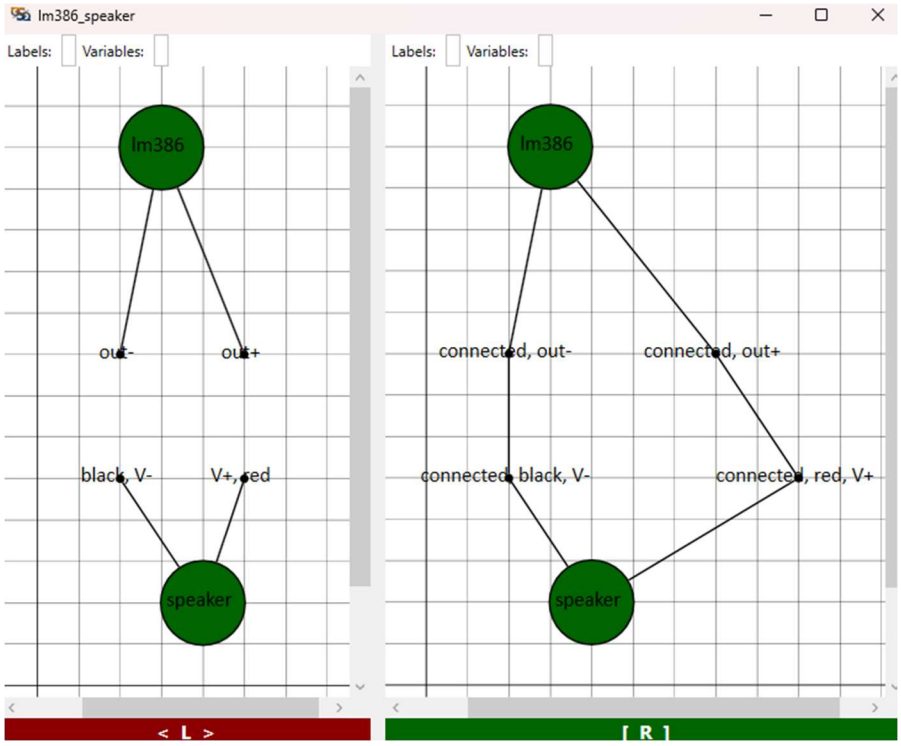


Figure F6: Original *lm386_speaker.grxml* Rule

New rule that combined the original rules:

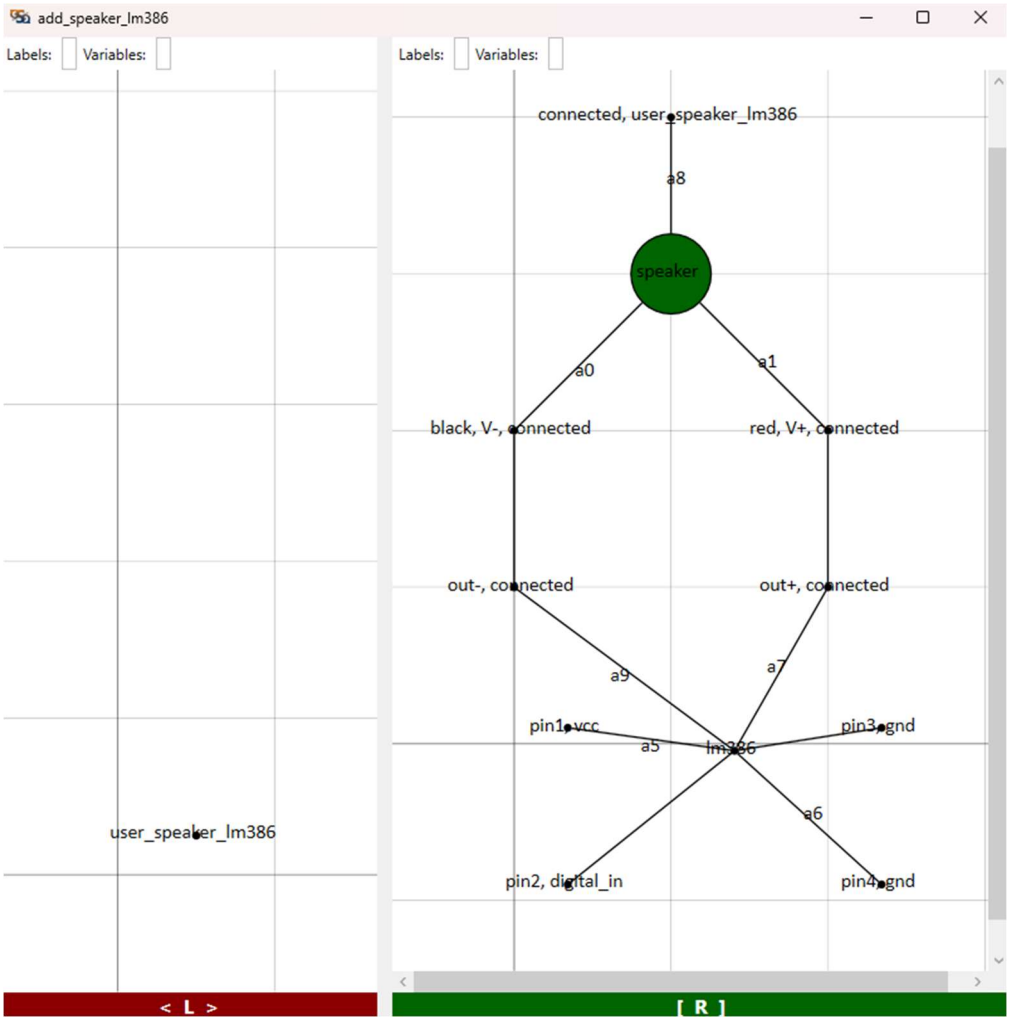


Figure F7: New *add_speaker_lm386.grxml* Rule

Remaining Combined Components

New combined Graph Rules for the remaining components in Table F1 can be found at this location:

https://drive.google.com/drive/folders/13zQRQM8AsaQ1e71UWiKumgkJaxqp8SmV?usp=sharing

Note that the combined rules can be found by the prefix "add_" and the names in Table F1.

Appendix G: List of components with Translation Functions

Table G1: List of components with Translation Functions

1	MPU6050
2	Stepper Motor
3	Servo
4	LED
5	KY012
6	KY003
7	ESC-Brushless DC Motor
8	Seven Segment Display
9	HCSR04
10	Loadcell-HX711
11	DHT22
12	MQ-3
13	BNO55
14	RBG LED

The shaded components were tested and refined.