UNIVERSITY OF CALIFORNIA SAN DIEGO

# “Sudo make me a smart device”: Accelerating and automating the end-to-end design of smart devices.

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

J. Garza Guardado

Committee in charge:

Professor Steven Swanson, Chair
Professor Scott Klemmer
Professor Sorin Lerner
Professor Deian Stefan
Professor Paul H. Siegel

2023

The Dissertation of J. Garza Guardado is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

# DEDICATION

I dedicate this work to my parents and family, who have lovingly supported me through this journey. Without you, this work would not have been possible. Thank you!

# EPIGRAPH

*We are whirlwinds of wind, born from the etheric field of the atmosphere that surrounds us, going around a few times, dusting everything in our path, and then at the end of the energetic-mathematical dynamics that fed us, we dissolve again to be only air, what we have always been and what we will always be.*

*—Athena Swaruu*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGEMENTS

I would like to begin by deeply thanking my advisor, Prof. Steven Swanson, for his invaluable guidance throughout my Masters and Ph.D. I appreciate the time you invested in helping me write research papers, give good presentations, particularly your patience in helping me understand what a research contribution is and is not, as well as your support in personal matters, an advisor both academically and in the life.

In addition, I would also like to thank Devon J. Merrill, my research colleague, who in addition to assisting me in my research helped me navigate my Ph.D., which as a foreign student I greatly appreciate.

Finally, I also thank the members of the committee for their feedback and comments that have helped me improve this work.

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the 19th International Conference on Web Engineering 2019. Garza, Jorge; Merrill, Devon J.; Swanson, Steven, Springer Lecture Notes in Computer Science, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in the Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. Garza, Jorge; Merrill, Devon J.; Swanson, Steven, Association for Computing Machinery, 2021. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is currently being prepared for submission for publication of the material. Garza, Jorge; Swanson, Steven. The dissertation author was the primary investigator and author of this chapter.

# VITA

2012 B. S. in Electrical and Computer Engineering, Monterrey Institute of Technology and Higher Education (ITESM).

2013–2014 Embedded Software Engineer, Torrey Electronics

2014–2016 M. S. in Computer Science (Computer Engineer), University of California San Diego

2015–2016, 2023 Graduate Teaching Assistant,University of California San Diego

2016–2023 Graduate Research Assistant, University of California San Diego

2023 Ph.D. in Computer Science, University of California San Diego

ABSTRACT OF THE DISSERTATION

**“Sudo make me a smart device”: Accelerating and automating the end-to-end design of smart devices.**

by

J. Garza Guardado

Doctor of Philosophy in Computer Science

University of California San Diego, 2023

Professor Steven Swanson, Chair

Smart devices design consists of multiple highly connected electronic device design stages (firmware, PCB layout, application logic, etc.). However, the design tools are separate, specializing in only one design stage, leading to complex design workflows with slow and time-consuming iteration processes. In this dissertation, we explore circuit data information encapsulation techniques for fusing multiple electronic device design steps, creating fast design workflows. We detail these techniques through three design tools: a framework for rapidly generating firmware from web-based applications, an interactive tool for rapidly transforming web pages into functional smart devices, an interactive tool for quickly merging electronic

schematics with real-time detection of common errors. This work also presents design automation techniques that are possible with the encapsulation of circuit data information. Small user studies show how these tools can provide an alternative to traditional design workflows by lowering the barrier to entry for beginners and streamlining design for experts.

# Chapter 1

# Introduction

Electronic devices play an essential role in our modern society. From the automation of many daily tasks, such as thermostats that maintain a comfortable temperature in our home, to multimedia entertainment devices. The expectations of electronic device designs that allow multiple modes of interaction and have multiple sensors have led to an increase in design complexity. The development of electronic devices is divided into multiple steps, for example, software development, low-level code development, component selection, printed circuit boards (PCB) design, among others. These steps vary depending on the electronic device type, smart devices that include screens being the ones with the most number of development steps, as they require high-level application logic development and graphical user interface (GUI) design. These development stages are highly coupled, and any design changes from one stage must be reflected in all development steps, making iterations slow and time-consuming. While the design of electronic devices requires coordination between all steps of development, the design tools at each stage have remained largely disconnected from one another.

This dissertation aims to explore new interactive design tools and techniques that fuse different steps of the end-to-end electronic device design process, with the motivation of reducing development times and design complexity. In particular, this work focuses on new approaches to encapsulate circuit data specifications into modular blocks, necessary for fusing design steps and which at the same time enables automation of low-level design tasks, further accelerating design.

Therefore, the central thesis of this work is the analysis of different encapsulation techniques and interaction design methods that support designers, from beginners to experts, to create fast electronic device designs with ease.

## 1.1 Motivation

The motivations for reducing the design complexity and development times required to create electronic devices are many. In a broad context, lowering the design complexity allows hobbyists, students, and novices to develop their own electronic devices and which in turn provides multiple benefits, from educational to societal benefits.

Educational benefits include assisting students, teachers, and researchers with the creation and exploration of complete electronic device designs [76], which do not remain mere prototypes. Other benefits include, helping designers quickly create devices close to the final product that allow faster transitions of new ideas to the market.

Tools with a low skill floor also help reach designers from diverse fields of study, assisting them with the creation of new types of electronic devices (e.g., biomedical devices). At the same time it promotes the democratization of devices [31, 101] allowing people to become less dependent on large corporations.

On the other hand, shorter development times make it possible to quickly react to emergencies where the creation of a unique electronic device is required [93], not to mention faster iteration times, which enable quicker improvements to devices and ultimately have a positive impact on consumers.

## 1.2 Design Steps Fusion

The current design process for smart devices can take years. Full automation of the design process to develop smart devices that can generate device designs in seconds is not yet possible; however, researchers are addressing this problem by creating interactive design tools

that abstract the design of different design steps by encasulating design information. In this way, designers can provide some design steps to an interactive design tool, such as hardware component selection, some user parameters, and perhaps the GUI and application logic of the device, and from there the design tool. design, along with some automation, can generate some of the design steps. Of course, the designer will also need to work to complete the remaining design steps. With this we may not reduce the design time to seconds, but the design process can be reduced to weeks or even less time.

We call the process of encapsulating circuit or other information from different design steps (e.g., low-level code design, schematic design, PCB design), abstracting the information into simpler parameters, and using these parameters in interactive tools that combine these parameters to create new designs as a *design steps fusion*.

## 1.3 Overview

We created three interactive design tools: Amalgam, Appliancizer, and TypedSchematics, which explore new approaches to achieve design steps fusion. These tools take advantage of the use of circuit data encapsulation for the rapid creation of electronic circuits.

Amalgam and Appliancizer, centers on electronic devices that include the most number of design steps, in this case smart devices with screens. The aim of Amalgam and Appliancizer is to explore the design of a tool that abstracts as much as possible the design of smart devices, while at the same time having the tool the ability to generate production-ready smart devices as quickly as possible. This combination of high abstraction and automation across the end-to-end design process allow us to fully research the possibilities and shortcomings of these types of tools. In other words, both Amalgam and Appliancizer, center on exploring the highest design ceiling possible which can produce manufacturable electronic PCB designs.

Amalgam focuses on an encapsulation technique for deeply integrating hardware devices' firmware functionalities into HTML and CSS. Appliancizer extends the encapsulation and

integrates circuit design data, allowing the tool to generate complete electronic designs, including PCB layout, directly from static web page designs. With Amalgam designers can transform web pages into fully functional electronic devices in a couple of minutes.

Although Amalgam and Appliancizer allow for a fast design with great ease of use, these tools do not permit low-level configurations. Using usiability terminology created by previous HCI researchers [81, 92], this means that both Amalgam and Appliancizer have a low threshold and a low ceiling when it comes to design, implying that both tools are easy for beginners but not as powerful for experts. Furthermore, integrating more modules into each tool libraries requires high expertise, which makes the scalability of these libraries not easy. TypedSchematics presents a tool with a design direction focused on improving the reuse and integration of schematic design blocks with a low-threshold, high-ceiling design experience, a balance that enables easy yet highly configurable schematic designs. In addition, TypedSchematics improves the scalability of the blocks' library by incorporating an annotation syntax into current PCB design tools. Overall, TypedSchematics lets designers rapidly create and merge different schematic blocks into a single schematic, as well as provides detection of design errors in real-time.

In addition to the circuit design reuse methods presented by the three tools, Appliancizer and Amalgam present unique solutions in terms of tool interface design that promote ease of use for designers. Appliancizer allows designers to simulate the interaction between physical components and components on the screen, using the same HTML elements as mock-up devices, improving idea exploration. TypedSchematics in turn introduces a schematic representation model that eliminates the need for ground connections and repetitive voltage, thus reducing interaction costs.

## 1.4 Contributions and Results

For each design tool, this dissertation makes the following contributions:

**Amalgam**

- A toolkit that deeply integrates hardware devices into web programming technologies, enabling rapid development and more flexible design iteration for embedded devices. Amalgam lets developers replace soft interface components with hardware components just by changing a CSS file.

  We implemented Amalgam and evaluated its capabilities by prototyping three devices in a web browser and then "hardening" them into standalone devices. Our results show that Amalgam can significantly reduce the programmer effort required to implement software for electronic devices.

**Appliancizer**

- A computational design that allows designers to rapidly generate functional smart devices, including its firmware and complete PCB layout design, from web-based prototypes, facilitating the prototyping of mixed graphical-tangible interactions and accelerating development.

- Essential interface mapping, a technique that allows mapping the similarities between HTML controls and tangible controls, enabling circuit synthesis, device simulation of digital and tangible interfaces, and low-level code generation. Essential interface mapping makes possible the transformation of HTML elements from graphical to tangible and enables the almost complete automation of the conventional development process for smart devices.

  We demonstrated the capabilities of Appliancizer by transforming both a video player and a smart thermostat web application into physical devices without requiring any changes to the application source code. We have also demonstrated the rapid development capabilities of Appliancizer through a preliminary user study showing a 6.5X speed improvement compared to Amalgam.

**TypedSchematics**

- An interactive design tool that improves the merging process of schematic blocks and enables real-time detection of potential design problems including mis-matched voltages, mis-connected signals in complex interfaces, and missing (but necessary) connections.

- *Mats*, an abstraction technique that eliminates the need for repetitive voltage and ground connections, reducing interaction costs. A flow-based programming paradigm is seamlessly integrated into Mats, making it possible to perform constraint validations of design problems in real time.

  We demonstrate the capabilities of TypedSchematics with two schematic design examples, a thermostat and a temperature logger. A user study shows that, on average, participants' design time of a thermostat schematic was 2.8 times faster with TypedSchematics than with Fusion 360. Feedback from participants via a questionnaire also showed that participants felt completely confident in the correctness of the design with TypedSchematics compared to Fusion 360, where they felt somewhat confident.

## 1.5 Content Structure

The rest of this dissertation is organized as follows: Chapter 2 provides an overview of the different electronic device design steps (e.g., schematic design, PCB assembly). Chapter 2 introduces Amalgam, a technique for encapsulating the low-level code required by circuit blocks within web components and within the programming syntax of web technologies (HTML, CSS, JS). Chapter 3 introduces Appliancizer, a tool that extends Amalgam and allows the automated generation of electronic devices, including PCB designs and low-level code, from web pages. Chapter 4 introduces TypedSchematics, a tool that makes composing schematics from schematic design blocks easier by providing real-time detection of common errors. Chapter 5 presents future work directions and Chapter 6 concludes.

# Chapter 2

# Background

Building an electronic device consists of multiple stages and varies depending on the complexity of the device. In this section we describe the stages required to design a smart device with an interactive screen, a complex electronic device with the largest number of design stages and which requires skills in multiple fields: software, hardware and mechanics. Figure 2.1 shows a diagram of the typical development process cycle for these devices that we briefly describe below.

***GUI & Application logic.*** In this stage, the graphical user interface (GUI) and the software of the device are developed. As embedded device processors are becoming more powerful and priced lower, the use of high-level programming languages (e.g., Python, Java, or web programming languages) is becoming more popular among these devices [45, 55]. Web programming languages (HTML, CSS, and JavaScript) have emerged as a popular solution for designing GUIs on embedded devices due to their capabilities: state-of-the-art for developing rich GUI, first-class networking, and available libraries.

***Device Simulation.*** Embedded device software development is tightly coupled with the hardware IO. Device simulation is often necessary to guide the design of the user experience and choose what form the interface should take. Simulation lets designers iterate quickly, start software

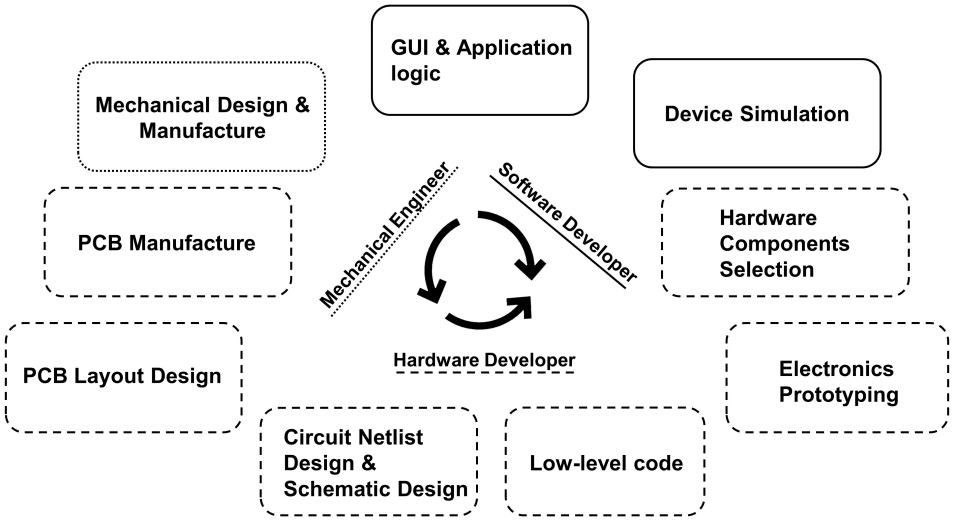



**Figure 2.1.** Conventional development process cycle of modern smart devices. Underlined are the tasks required to be accomplished by developers in each field.

development before the hardware is ready, and identify potential issues early.

***Hardware Components Selection.*** The selection of electrical components is made in this stage. Components such as ICs, LEDs, resistors, capacitors, and sensors are selected based on design specifications and desired device futures. An understanding of the capabilities of each component is required to select each component correctly. User interface decisions drive hardware selection, which, in turn, impacts how the programmer implements aspects of that interface.

***Electronics Prototyping.*** Interactive prototype boards and toolkits such as Arduino [20], BBC micro:bit2 [41], and Raspberry Pi [88], are used to rapidly and inexpensively create prototypes of electronic devices. Multiple prototyping paradigms exist, Lambrichts et al. [64] categorizes these paradigms into three, based on the use of: discrete electronic components only, breakout and development boards, and integrated toolkits consisting of modules specifically designed to work together.

***Low-level code.*** After hardware components selection, code that makes the connection between

the software and each hardware component is needed. We use the term *low-level code* to refer to all the necessary code required to communicate with hardware components. Low-level code is also known as *firmware* in the case of microcontrollers and *device drivers* in the case of processors.

Electronic devices use a variety of wired communication protocols for interfacing with hardware components, such as GPIO, I2C, SPI, and UART. We refer through this work to these protocols as *hardware interfaces*. Each hardware interface can address a wide range of sensors and actuators, including accelerometers, LEDs, servo motors, and light detection sensors as examples. Embedded computing platforms (e.g., Raspberry Pi [88]) provide groups of pins with different capabilities that may cover some or all of the above interfaces. The pin numbering varies for each embedded computing platform. To make the low-level work with the different computing platforms developers also need to map the low-level code with the pins of the computing platform, known as *pin mapping*.

***Circuit Netlist & Schematics.*** At this stage, connections are created between the different selected hardware components. PCB design tools are used to generate circuit netlists as they allow designers to visually interconnect the hardware components required for a device. The visual representation of circuit netlists is known as *PCB schematics*. KiCAD KiCAD and EAGLE [34] are examples of PCB design tools. Pin mappings in the low-level code must match the PCB schematics for the device to function properly.

***PCB Layout Design.*** After completing a schematic design, PCB design tools also provide graphical interfaces for creating PCB layouts. A PCB layout digitally describes the physical placement and routing between hardware components. From a PCB layout, manufacturing files can be generated containing all the specifications required for PCB fabrication; known as *Gerber files*.

***PCB Manufacture.*** The cost of manufacturing PCBs has dropped in recent years, allowing hobbyists and students to create high-quality, long-lasting, and reliable circuits. PCB fabrication industries like JLCPCB [56], for example, provide 2-layer PCB fabrication for $ 2 and 1-week lead times, allowing for low-cost rapid iterations of PCB prototypes. After PCB manufacturing, industries can also provide assembly services for the physical placement and soldering of hardware components, freeing hardware developers from the technical skills required to assemble these devices.

***Mechanical Design & Manufacture.*** At this stage, the manufacture of enclosures and mechanical parts is carried out. Mechanical engineers must know the dimensions of PCBs and hardware components to design mechanical parts. Connector openings, gear sizes, and screw terminal positions can go wrong if they are not known. While the mechanical parts of the electronic device are essential, in this work we only focus on the software and hardware tasks of the design process.

The different stages of the design of electronic devices require many steps and deep experience in each of these areas, which translates into additional costs and time. Because the entire design process is highly coupled, any design change must be reflected at all stages of development, making iterations time-consuming and complex, especially as each task is typically handled by different people with different skill sets.

However, although each of these stages depends on one another, the tools used in each of the stages are loosely coupled, remaining disconnected and separate from one another. In this work, we propose new design tools that fuse several of these design stages using different methods of encapsulating circuit data information. We also explore design automation workflows that are possible with the encapsulation of circuit data information.

# Chapter 3

# Amalgam

Amalgam is a tool that focuses on fusing software or application logic and low-level code design steps of smart devices. Amalgam does this by integrating the low-level code of hardware components found in embedded devices (e.g., a push button) directly into the HTML and CSS syntax. By doing so our system reduces the programming effort required to develop new embedded devices that use web technologies, as well as adds new interesting capabilities to the design of these, for example, hardware manipulation through CSS.

Amalgam's application logic and low-level code steps fusion is achieved by encapsulating low-level code information in HTML web components and low-level code hardware interface specifications and pin assignments (e.g., SPI port, I2C address, GPIO number) into the CSS syntax. Amalgam's library of modules with encapsulated information consists of nine elements that allow the use of different hardware components (e.g., servo motors, RGB LEDs, potentiometers) as HTML elements.

We demonstrate Amalgam's capabilities by exploring three embedded devices built using web programming technologies. Also, we demonstrate how Amalgam reduces programming effort by comparing two traditional approaches of building one of these devices against Amalgam. Results show our system reduces the lines of code required to integrate hardware elements into the device application to a line of code per hardware component added to an embedded device.

## 3.1 Introduction

With the rise of the Internet of Things (IoT) and smart, connected devices in the home, workplace, and environment, the web and its underlying technologies are pushing up against the real world in a wide range of domains. Connected sensors, web-enabled appliances, and personal electronics all require software that interacts seamlessly with the physical world (e.g., the user or the environment), cloud-based services, and local compute resources. The growing demand for these devices means they need to be easy to program.

Building these smart, connected devices requires programmers to manually bridge the gap between tangible user interfaces (e.g., buttons, knobs, and displays), sensors (e.g., temperature, light, and movement), and actuators (e.g., servos, motors, and lab equipment) and software.

On the software side, web technologies – Javascript, CSS, HTML and the universe of libraries available for them – are state-of-the-art for developing rich user interfaces, provide deep integration network services, and are the languages of choice for a large population of developers.

Web programming technologies provide a clean separation between program logic, interface structure, and appearance. They make it simple to re-style an interface for a new device or adapt an existing interface to a new form factor (such as desktop to mobile). These tools are so powerful that they have become the default user interface design tools for fixed-function mobile devices, desktop applications, and mobile applications.

For hardware, the tools of choice remain C and C++ which can easily handle the particulars of controlling hardware components (e.g., interrupts, pin assignments, and device drivers) to control sensors, tactile user inputs, and actuators. However, they make building user interfaces and networking communication more cumbersome.

Creating a seamless experience that blends on-screen, soft controls, hardware interface elements, sensors, and actuators is challenging because the elegant separation that HTML, CSS, and JavaScript does not extend across the hardware/software boundary. In practice, the tools

available for hard and soft elements differ in syntax, operation philosophy, and requirements.

This problem is ubiquitous in modern devices. The interfaces to embedded devices – from personal fitness monitors to home appliances – have sophisticated, polished, and powerful user interfaces. Even small devices (e.g., the Apple Watch) typically run full-blown operating systems that can support high-level languages for graphical user interfaces. According to an annual industry survey of embedded designers, 67% of new embedded designs utilize an operating system and 49% use graphical interfaces [37].

Embedded devices with graphical interfaces can have a blending of *soft* and *hard* components that provide information to or from the user. Examples of soft components can be on-screen buttons, range sliders, and indicators in the form of text or graphics. Hard or physical components include tactile buttons, knobs, and sensors (such as temperatures, heart rate, and so forth), actuators (such as servo motors) and hardware indicators (for example, status lights).

Previous attempts to address this problem simply translate the same complex interfaces into higher-level languages, rather than deeply integrating hard elements into the idioms and tools high-level languages provide. This does not solve the problem: Programmers must still treat physical interface components differently than their soft counterparts.

Indeed, several projects [63, 86] provide JavaScript libraries for controlling robots [5] and general embedded systems [46], but they leave behind the power of CSS and HTML, preventing deeper integration with existing programming toolkits and tools.

As a result, web programmers that want to build software that deeply integrates software and hardware cannot leverage their own experience; the wealth of training, documentation, and message boards; or the myriad web programming frameworks that are available. Instead they must develop custom solutions to bridge the gap.

We propose Amalgam, a toolkit that extends web programming technologies across the hardware/software boundary by seamlessly including hardware devices into Javascript, CSS, and HTML. Amalgam exposes hardware devices like buttons, sensors, lights, and motor as domain object model (DOM) objects that have the same interface as the analogous HTML elements (e.g.,

<button> or <input type=range>).

Amalgam lets programmers *harden* conventional DOM objects into hardware devices using a simple CSS directive. The directive controls whether a particular component appears on-screen or as a physical component, and describes how the device connects physically to compute platform.

As a result, moving a button from on-screen to the real world requires just editing a CSS property and physically connecting the button. Application logic does not change because the interface remains the same. More important, existing frameworks like Angular [1] and JQuery [29] work just as well with hard elements as soft.

This capability extends to output as well. For instance, the programmer can create complex lighting effects by using CSS to animate the color of an RGB LED. Likewise, setting the content of a  that has been hardened into display changes the contents of the display.

We have implemented Amalgam as a Javascript framework and developed a small but useful library of hardware components. We demonstrate Amalgam's capabilities by using these components to create three embedded devices with rich hardware/software interfaces. We demonstrate that Amalgam works seamlessly with existing web-programming frameworks and libraries to build complex, responsive interfaces for these devices. We also describe Amalgam's implementation and measure how Amalgam makes it easier to develop these kinds of devices.

The rest of this paper is organized as follows. Section 3.2 introduces the technologies used throughout the paper. Section 3.3 gives a overview of Amalgam, and Section 3.4 illustrates Amalgam's capabilities by describing three Amalgam devices. Section 3.5 evaluates the impact of Amalgam on developer effort. Finally, Section 3.6 describes related work, and conclusions for this tool are presented in the final chapter, Section 6.1.

## 3.2 Background

Amalgam lets programmers integrate hardware components (e.g., push buttons, LEDs, sensors, etc.) in applications built using web programming technologies like Javascript, CSS, and HTML. It accomplishes this by providing interfaces to those components that mimic the interfaces of "soft" components (e.g., on-screen buttons) provide. Below, we describe the aspects of these technologies that are most relevant to Amalgam and briefly review the interfaces currently available to programmers for using hardware components.

### 3.2.1 Web Technologies

Programming for the web continues to drive a remarkable amount of innovation in programming tools and abstractions. These innovations are visible in the rapid evolution of HTML ( [12], [110]), CSS, and JavaScript (officially ECMAScript [36]) and in the myriad frameworks and libraries available for web programming. These tools ease development of complex, flexible, highly-interactive user interfaces. Crucially, these frameworks rely on the interfaces that HTML, CSS, and JavaScript provide and the programming idioms (e.g., promises and asynchronous operations) that form the foundation of web programming.

**HTML and DOM** The Document Object Model (DOM) [84] specifies a wide range of user interface elements, including buttons, sliders, check boxes made up of HTML tags and attributes. Software interacts with these elements through DOM method calls, CSS styles, and DOM events.

**CSS** CSS uses a declarative language to control the appearance of DOM elements on the screen. Its complexity has grown to allow it control many aspects of a program's behavior (e.g., what controls appear on mobile vs desktop displays). It has also replaced tedious programming for things like animations.

**JavaScript** JavaScript provides an object-oriented programming language for manipulating the DOM and implementing program logic. A defining characteristic of JavaScript is its first-class support for asynchronous operations. Indeed, JavaScript code (especially in interactive web

applications) should be non-blocking.

JavaScript's non-blocking, asynchronous execution model places it at odds with conventional interfaces for hardware devices, in which programs issue synchronous system calls from languages like C, C++, or Python that do not support asynchronous operations by default. While it is possible to "shoehorn" this model into JavaScript, the result is inelegant and prevents programmers from relying on the powerful idioms that JavaScript provides and JavaScript programmers rely on.

**Frameworks** A hallmark of modern web programming is the amazing number and variety of open-source "frameworks" that leverage these core technologies. Frameworks provide powerful improvements in programmability, implement useful features, or solve common web-programming problems.

Individuals and companies have created a diversity of frameworks ranging from the simple to the complex, with more appearing every year. This activity lies at the heart of innovation in web-based user interface programming. All these frameworks are deeply intertwined with (and dependent on the power of) the core web technologies.

**Web Components** Web components [108] are a recent extension to HTML that allows programmers to extend the DOM by defining new tags that behave like native tags. Web components can read, write or listen to the changes to their HTML attributes and react when those attributes changes (e.g., in response to CSS animation). Frameworks like Angular [1] served as a proving ground for these capabilities.

**Web Assembly** Browsers use JIT-accelerated interpreters to execute JavaScript code. This limits performance and prevents languages like C or C++ from running in the browser. Web Assembly [11] eliminates this limitations and lets developers to pre-compile code in any language for execution in the browser.

**Electron** Web browsers use a sandbox environment for web applications which restricts their access to the file system or hardware. Electron [2] is a framework that lets trusted web

applications access the file system or hardware through the use of Node.js [7]. Electron also allows web applications to run as stand alone desktop applications, a capability commonly sought in frameworks for building embedded devices.

### 3.2.2 Tools for Programming Devices

Innovation in programming tools for controlling low-level hardware components that let devices interact with users and the real world (e.g., buttons, sensors, LEDs) have undergone a renaissance in parallel with web programming. Software tools like the Arduino development environment [20] and hardware platforms like Raspberry Pi [88] have made it possible for relative novices to build complex electronic devices. The result has been an explosion of creativity in the "maker" community.

Arduino and Raspberry Pi provide access to hardware through C/C++ or Python interfaces that let programmers manipulate IO pins and peripherals attached to micro-controllers or embedded processors. They hide the details of the underlying hardware and provide a measure of portability across different platforms. For instance, the same code will run (with a little effort) on Raspberry Pi and Beagle Bone Black [26].

Despite being simpler than the register-based (and very nonportable) interface that "raw" hardware provides, the interfaces are still low-level: they provide simple function calls for toggling an IO line or reading an analog voltage on a processor pin rather than smoothly dimming an LED or reporting that a particular button has been pushed.

Building complex applications with rich user interfaces using these interfaces is possible, but it is complex. A common route is to use GUI toolkits like Qt [8] that are C/C++-based and then use multiple threads to handle asynchronous events (e.g., button presses). These toolkits also lack many features that make web programming tools so powerful: first-class networking integration, an abundance of frameworks, and sophisticated, declarative control over the interface's appearance (i.e., CSS).

This approach also makes prototyping and refining interfaces with Qt and low-level

hardware interfaces cumbersome. Implementing a simple design change (e.g., converting an on-screen button to a physical button) requires significant changes to application logic. The effort required discourages rapid and frequent design iteration.

Finally, and most importantly, toolkits like Qt lack the user base, vibrant online communities, rich suite of libraries, and near universal availability of web-based tools for building GUIs. As a result, the learning curve is steep and there is less help available online.

Combined, these challenges mean that building electronic devices that meld soft (i.e., on screen) controls with hard (i.e., tactile) controls is hard. GUI designers and web developers must "retool" to program hardware and, likewise, embedded systems programmers must bridge the gap to the browser to leverage web technologies.

### 3.2.3 Hardware Interfaces

Embedded computing devices use a variety of electrical interfaces to connect to hardware components. These include general purpose I/O (GPIO) for single-bit digital inputs and outputs (e.g., to detect a butt press or turn on an LED), analog to digital converters (ADC) input to measure an external voltage (e.g., to measure lighting levels or temperature), analog outputs (e.g., to dim an LED or control a motor's speed). In addition, Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C) busses provide standard mechanism for communicating with more complex peripherals (e.g., LED arrays or simple displays). Finally, a single physical pin can serve multiple roles (e.g., the I2C pins can also serve as GPIO pins).

Each embedded computing platform (e.g. Raspberry Pi or Beagle Bone Black) will provide a different set of pins with different capabilities cover some or all of the above interfaces.

Functionally similar devices might require different interfaces, so there is great diversity in how, for instance, a button might connect to the system.

**Table 3.1. A Selection of Hardware Components** Amalgam can connect to a huge array of hardware components. These are the components our examples use.

| Name | Description | Hardware Interface | Uses |
|---|---|---|---|
| Rotary Potentiometer ( or "pot") | A knob with a fixed range | ADC or I2C | Analog volume controls, light dimmers |
| Rotary encoder | A knob that turns forever in either direction | ADC or I2C | Menu selection, digital volume control |
| Linear motorized pot | A slider that can also move itself to specified location | ADC or I2C and GPIO | Audio mixing boards |
| RGB LED | An LED that can emit any color of light | SPI | Indicators, decoration |
| Tactile push-button | A "clicky" button | GPIO | Buttons, controls, switches |
| Servo motors | Precisely controllable motors | I2C | Robot joints, actuators |
| LCD text display | Black and white, high contrast display | Serial | Calculators, inexpensive text displays |
| LED numerical display | Illuminated "7-segment" displays | SPI | Inexpensive numerical displays |
| Load cell | A pressure sensor | SPI | Digital scales |

### 3.2.4 Hardware Components

Arduino and Raspberry Pi can connect to an enormous range of hardware components. Table 3.1 briefly describes the components we use in our example code (Section 3.3) and example devices (Section 3.4).

## 3.3 Amalgam

Amalgam is a Web API that integrates hardware components into web programming tools in a natural and transparent way. It provides a new style attribute (`hardware`) that converts an on-screen element of a web-based interfaces into a hardware device. We call this process *hardening* the on-screen element. Hardening allows, for instance, the replacement of an on-screen button with a physical button. The element's interface remains the same so application's software does not need to change.

We have implemented Amalgam as a JavaScript library. It leverages Web Components

a) Web Application HTML (Soft elements)

```
<div id="display">0</div>
<input id="slider" type="range" min="0" max="9" value="0"
oninput="getElementById('display').innerHTML = this.value"/>
```

b) Amalgam-enhanced CSS

```
#display {
 hardware: physical-seven-segments ( spi-port:
                                     url('/dev/spidev0.0') );
}

#slider {
 hardware: physical-pot ( adc-channel: 1, i2c-addr: 0x48,
                          i2c-port: url('/dev/i2c-1') );
}
```

c) Software components


d) Hardware components

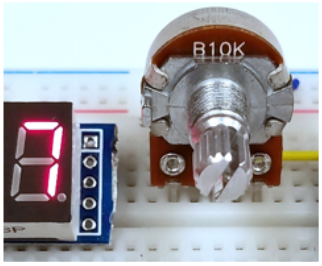



**Figure 3.1. Styling hardware with Amalgam** a) Describes the interface for a simple numerical display controlled by a slider. Applying an Amalgam-enhanced CSS style sheet b), produces the same on-screen version of the interface c) implemented in hardware d).

and Web Assembly to build DOM elements that interface with hardware, and it provides a simple compiler that parses a web page's CSS style sheets and hardens elements according the `hardware` directives it finds.

This section describes Amalgam's programming interface, presents a simple example of Amalgam in action, describes the library of physical components we have implemented, and what is required to create a new one.

### 3.3.1 Overview

Amalgam's programming interface is simple by design. It lets programmers convert existing DOM elements, which we call *soft elements*, into hardened elements that exist in the real world.

Figure 3.1 shows a simple Amalgam application. In the figure, (a) shows the HTML for a range `<input>` and a `<div>` along with the event callbacks to ensure that the `<div>` displays the value of the `<input>`. (c) depicts the web page in browser running on a Raspberry Pi.

The CSS code in (b) hardens both elements by setting their `hardware` attribute. It converts the `<div>` into a seven-segment LED display and the `<input>` into a rotary knob potentiometer. The photo (d) shows the hardware. Turning the knob, updates the display. No other changes are necessary to the code.

The value of the `hardware` attribute describes what kind of hardware to use (in this case, the knob and the display) and how the two components connect to the Raspberry Pi. In this case, the display connects via SPI and the potentiometer connects to the first channel of an analog-to-digital (ADC) integrated circuit connected via I2C

Once hardened, the components continues to behave just like the original soft component. It has the same DOM methods and emits the same events (e.g., when the knob moves, the `<input>` emits `onchange`). If the hardened component has a display capability (e.g., an RGB LED), the programmer can style or animate it with CSS.

Since hard elements have the same interface as soft elements, existing application code requires no modification. In particular, web programming frameworks function as expected without any changes.

Amalgam can also *soften* hardware components by replacing them with on-screen, simulated components. For instance, during software development and testing for a smart light switch, an `<input>` could emulate the input from a light sensor to facilitate development and testing without access to actual hardware: The developer could move slider to control the simulated light level.

We have implemented a small library (Section 3.3.3) of hardware components that Amalgam can use to harden DOM elements, but implementing new components is simple.

Cleanly integrating hardware components into web programming technologies offers multiple benefits.

- **Easy Hardware Emulation** Amalgam decouples application design from hardware design. Software developers can implement the software for a soft version of a device long before the hardware is complete.
- **Faster Design Iteration** Developers can rapidly explore different designs by hardening and softening different parts of a user interface without needing to modify application logic.
- **Faster Development** Amalgam lets developers leverage the universe of available JavaScript frameworks to quickly build complex applications.
- **Automatic Web Integration** Because they are web applications, Amalgam applications have first-class access to web services, the cloud, etc.

Amalgam makes it easy for programmers to control hardware devices, but they still need to assemble the device. And they need some familiarity with the hardware and its limitations. Amalgam can replace any HTML element with any hard component, but the programmer must be aware potential limitations. For example, setting the `value` attribute for `<input>` hardened into a normal potentiometer will not move the potentiometer. If the programmer needs that capability, she should use a rotary encoder or motorized pot. Likewise, the designer must be aware of which pins connect the hardware components and if those connections change, the programmer must update the CSS.

### 3.3.2 Amalgam Platforms

Amalgam applications run an Amalgam *platform*. A platform includes a computing device (e.g., a RaspberryPi), a web programming runtime that includes JavaScript, DOM, and CSS (e.g., a web browser) that the applications run in and an *hardware abstraction layer (HAL)* that provides low-level access to hardware. A web server (running on the platform or remotely) serves the application. Figure 3.2 shows the components of an Amalgam platform and their relationships to an Amalgam application.

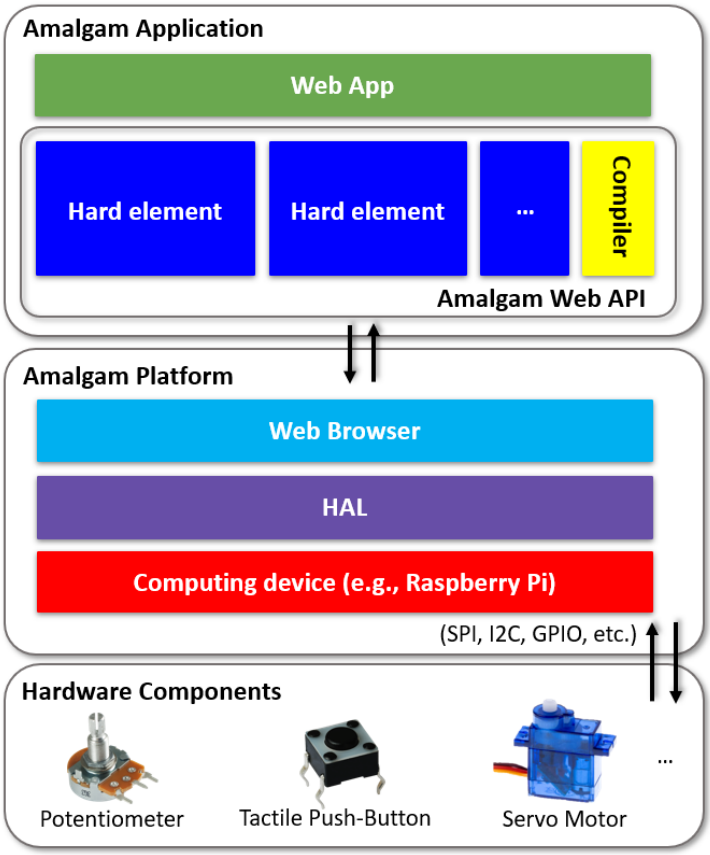



**Figure 3.2. Amalgam Platform:** An Amalgam application, a web app that includes the Amalgam Web API, can run on Amalgam Platforms which includes a web browser engine that can communicate with hardware through the HAL.

The HAL exposes a standard software interface (i.e., function calls) to common hardware interfaces (i.e., electrical connections to the platform hardware). The prototype Amalgam HAL is a user space implementation of Arduino Reference Language [20] for Linux, called Linuxduino [6]. It accesses low-level hardware through Linux's standard drivers, so it should be portable across the many Linux-based embedded systems that are available. Since it is Arduino-compatible, it supports a huge array of hardware components.

We compile Linuxduino to web assembly, so it runs directly in the JavaScript runtime.

The main difference between different Linux-based platforms is the set of electrical interfaces they provide. For instance, Raspberry PI provides 26 digital IO pins, two I2C interfaces and two SPI interfaces, but no analog inputs or outputs. Beaglebone Black has 25 digital pins, six analog outputs, and seven analog inputs, one I2C bus and one SPI bus. In addition, the platforms use different naming schemes for their pins.

These differences are visible to Amalgam programmers so moving an Amalgam program between platforms requires adjustments to the CSS that hardens the components. Likewise, if the hardware designer changes which pins connect a particular device to the platform, the CSS must change as well.

### 3.3.3 The Amalgam Library

To explore Amalgam's ability to accelerate the design of complex hard/soft interfaces, we built seven hard elements that match existing soft elements HTML interfaces. Table 3.2 summarizes the elements Amalgam currently supports.

The range of possible hard elements is broader than the set of elements that HTML provides, because many different hardware devices can replace a single HTML element and even similar hardware devices may connect to the system through different interfaces.

For instance, the example in Figure 3.1 hardened the `<input>` into a potentiometer (or "pot") that connected via one ADC channel. It could have instead used a "rotary encoder" that connects via two digital IO lines or a "motorized slide pot" that requires an ADC line and three

digital IO lines, two lines to control the motor direction and one to get the user slider touch feedback.

The Amalgam library has entries for each of these alternatives. Their internal software implementations are quite different despite appearing the same to the application (i.e., as a range `<input>`).

Adding new hard elements to Amalgam requires two steps. The first is creating a web component that will interface with the hardware device. The component encapsulates the firmware (written in JavaScript) that controls the hardware via the HAL.

For example, List 3.1 is a class that implements a hard button by extending `HTMLElement` and providing three methods: `get_observedAttributes()` defines the attributes this element supports. Whenever an attribute changes, including when an element is harden, `attributeChangedCallback()` runs and gets updated. Finally, the runtime invokes `connectedCallback()` once after the attributes are updated, indicating that the web component is ready. Here initialization and configuration of hardware is carried out. In this case, a given gpio number, set in the gpio attribute value, is configured as input and is physically connected to a button. After that another function sets up a call back that polls the IO pin every 200ms, which is enough to detect a button press, and emulates a click when it detects a physical button press (lines 13 - 18). For this example the gpio number can only be initialized once but it works for our embedded device prototypes requirements.

Another example, in List 3.2 a class implements an RGB LED. Here the hard element listens for changes to the style attribute. If there is a change, the background-color CSS property gets computed and its color value is sent to a RGB LED using the SPI protocol. The servo-motor hard element has a similar implementation but instead it computes the transform CSS property and then the rotation angle is sent to the servo motor.

The final step is to register the new class with Amalgam so the programmer can use it to harden elements. The code in List 3.1 defines a new HTML tag called `<physical-button>` that the programmer can use directly (e.g., `<physical-button`

**List 3.1.** A hard element button code example which consist of a typical web component code plus calls to hardware using the Linuxduino HAL library.

```
class PHYSICAL_BUTTON extends HTMLElement {
 constructor () { super(); this.gpio; }

  // Monitor attribute changes.
  static get observedAttributes() {
    return ['onclick', 'gpio'];
  }

  connectedCallback() {
    // Initialize GPIO
    Linuxduino.pinMode(this.gpio, Linuxduino.INPUT);
    // Start Reading GPIO
    setInterval( () => {
      // Call 'onclick' if physical button pressed
      if (Linuxduino.digitalRead(this.gpio)  == Linuxduino.HIGH) {
        this.click();
      }
    },200);
  }

 // Respond to attribute changes.
 attributeChangedCallback(attr, oldValue, newValue){
    if (attr == 'gpio') {
      this.gpio = parseFloat(newValue);
    }
 }

}

customElements.define('physical-button',PHYSICAL_BUTTON);
```

**List 3.2.** A hard element code example which consist of a typical web component that listens for color changes in the background-color style property to effect those changes in a RGB LED hardware component.

```
1  class PHYSICAL_RGB extends HTMLElement {
2   constructor () {super(); this.spi = null; this.spiPort;}
3
4   // Monitor attribute changes.
5   static get observedAttributes() {
6     return ['style', 'spi-port'];
7   }
8
9   connectedCallback() {
10   // Initialize SPI
11   this.spi = new Linuxduino.SPI();
12   this.spi.begin(this.spiPort);
13  }
14
15  // Respond to attribute changes.
16  attributeChangedCallback(attr, oldValue, newValue){
17   if (attr == 'style') {
18    // Get background color
19    var backgroundColor = window.getComputedStyle(this, null)["
          background-color"];
20    // Effect the changes on the physical RGB
21    setPixelColor(this.spi,rgb2hex(backgroundColor));
22   } else if (attr == 'spi-port') {
23    this.spiPort = newValue;
24   }
25  }
26 }
27
28 customElements.define('physical-rgb-led',
29    PHYSICAL_RGB);
```

**Table 3.2. Amalgam's hard elements**

| *Hard element tag* | *Amalgam version* | *Compatible soft element tag* | *Notes* |
|---|---|---|---|
| `<physical-pot>` | Rotary Potentiometer ( or "pot") | `<input type=''range''>` | Triggers 'oninput' when potentiometer input value is changed. |
| `<physical-encoder>` | Rotary encoder | `<input type=''range''>` | Triggers 'oninput' when potentiometer input value is changed. |
| `<physical-motorized-pot>` | Linear motorized pot | `<input type=''range''>` | Triggers 'oninput' when potentiometer input value is changed, also setting the 'value' attribute can set the slider position. |
| `<physical-rgb-led>` | RGB LED | `<div>` | Color is set to the CSS background-color property. |
| `<physical-button>` | Tactile push-button | `<button>` | Triggers 'onclick' event at button press. |
| `<physical-servo-motor>` | Servo motor | `<div>` | Servo angle is set to angle rotation of CSS transform propery. |
| `<physical-lcd>` | LCD text display | `<span>` | Text is set with a hard attribute. |
| `<physical-seven-segments>` | LED numerical display | `<span>` | Numbers are set with a hard attribute. |
| `<physical-weight-sensor>` | Load cell | `<input type=''range''>` | Measured weight is available via Angular ng-bind attribute. |

`onclick="foo()" gpio="1"></physical-button>`) to create a hard button. The registration process makes the Amalgam CSS compiler aware of the class so it can replace an existing tag (e.g., a `<button>`) with a `<physical-button>`.

### 3.3.4 Amalgam-enhanced CSS Style

The `hardware` CSS style attribute controls if and how Amalgam hardens a DOM element. The value of `hardware` describes which hard component should replace the software component and describes which electrical interfaces the corresponding hardware device will connect to.

Figure 3.3 (b) shows the CSS code required to harden a soft element (a). Each value for `hardware` starts with the name of the hard component that Amalgam will use to replace the soft element. The remaining arguments are of the form *`attr(value)`* that Amalgam uses to set the

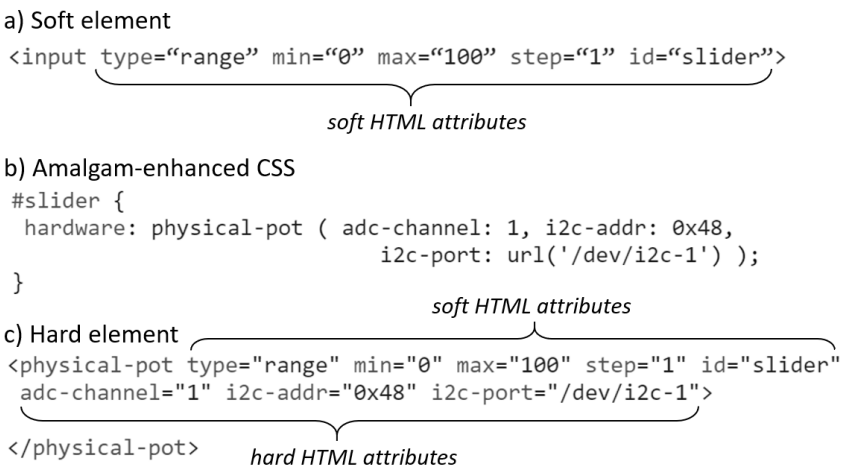



**Figure 3.3. Amalgam compiler hardening of soft elements,** a) Shows a soft element selected by an Amalgam-enchanced CSS property in b). After compilation c) shows the hard element HTML which replaces a).

hard HTML attributes on the hard element (c) it creates. The hard element as well will inherit soft HTML attributes from the soft element in (a) to keep the same web application functionality without any changes.

Our prototype implementation uses a JavaScript CSS processor to scan a pages style sheets for `hardware` declarations, and then harden the elements appropriately.

## 3.4 Examples

To demonstrate Amalgam's capabilities, we built three electronic devices: A video player appliance, an electronic commercial scale (e.g., for weighing items in a grocery store), and a dancing speaker. Each electronic device started with an on-screen, soft prototype. Then, we hardened some of the soft components and built a prototype of the physical device. The Amalgam platform we use comprises Raspberry Pi, running Ubuntu Linux and Electron.

### 3.4.1 Video Player

The video player appliance (Figure 3.4) demonstrates Amalgam's ability to transform an existing web page into the software for a physical device. The left side of the figure shows


**Figure 3.4. Video Player:** At left, a demo of a video player which provides a familiar on-scree interface for playing videos. At right, Amalgam allows the same demo to drive a fully-tactile interface, the only difference being the style sheet.

the soft video player built with the Youtube Player API [111]. It provides a familiar on-screen interface for playing videos, including the slider that displays and controls the current spot in the video.

The right side of the figure shows the appliance we built. Videos appear on the screen, but all the rest of interface is hard. The only difference in software the CSS directives to hardened the three buttons, the volume control, and slider (List 3.3).

The appliance mimics all the behavior of the original, including the progress bar. We hardened it into a motorized linear pot that both the software and the user can move, so the user can "scrub" through the video by moving the slider.

### 3.4.2 Commercial Scale

The scale we built (Figure 3.5) shows how Amalgam can simplify and accelerated iterations during prototyping. The scale has two users: the customer and the salesperson. The salesperson can select products from an illustrated list, see the price per pound, weigh the item, and adjust the scale by zeroing it or setting a tare weight (to account for the weight of a container), and show the total. The customer can see the items name and total price displayed on as second display on the back of the scale.

The soft version (at left) implements all this interface and all the application logic using soft elements. In particular, it softens the load cell (that measures weight) and the second display

**List 3.3.** Video Player Code. At the top we show only two of the software components of the Video Player web application, the play and pause button and the progress bar. At the bottom the Amalgam-enhanced CSS required to harden the software elements, process that is carried out by the compiler at run-time.

```
<!-- Soft elements -->
<button onclick="playPause()" id="playPause">Play/Pause</button>
<input type="range" min="0" max="10" step="1" value="0" id="progressBar">

<!-- Amalgam-enchanced CSS -->
<style>
#playPause {
  hardware: physical-button(gpio:var(--gpio5));
}

#progressBar {
  hardware: physical-motorized-pot
  (motora:var(--gpio23), motorb:var(--gpio24),
  touch:var(--gpio25), adc-channel: 2,
  i2c-addr: 0x48, i2c-port:url("/dev/i2c-1"));
}
</style>
```

on the back of the scale (both shown as gray overlays). The developer can perfect the application logic (including varying the weight on the virtual scale) in a web browser without access to real hardware.

We implemented the soft version using Angular [1], a sophisticated model-view-controller library. Among other things, Angular makes it trivial to "bind" the output of the load cell to the weight display. Since Amalgam's hard elements have the same interface as normal DOM elements, this works just as easily with hard elements.

The rest of the figure show two prototype versions. The design in the center provides a completely soft salesperson interface. The one at right uses hard components for the buttons and numeric displays. Both of them have a hard customer display. The only software difference between all three versions is a few lines of CSS.

### 3.4.3 Dancing Speaker

Our dancing speaker (Figure 3.6) highlights the power of using CSS animations to control hardware. The speaker plays and "dances" to music by waving its arms and flashing its lights in time to the music. It is a fanciful design that a "maker" might assemble as a hobby project.

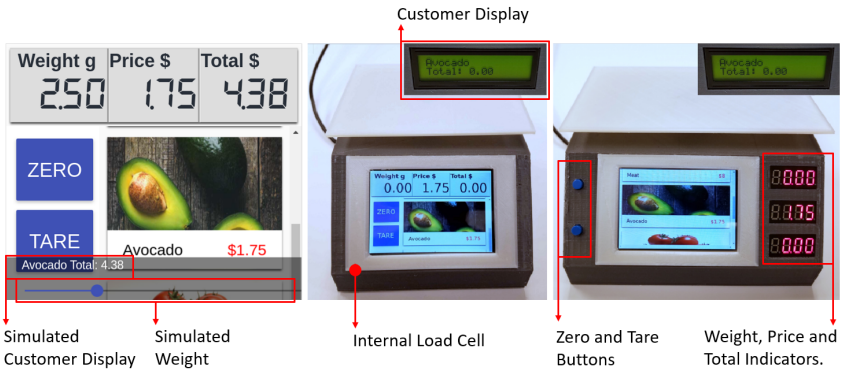



**Figure 3.5. Evolving Scale:** Amalgam allows a spectrum of different implementations with minimal developer effort. From left to right: a software-only mock-up includes a virtual, on screen load cell and supports software development; a "soft" interface version that uses a touch screen for the main screen and buttons and an LCD for the rear screen; and hybrid version that uses 7-segment displays and tactile buttons.

Assembling the hardware for the dancing elements is simple, but writing the software to control (e.g., beat detection and complex coordinated transitions) them would be very complex. Instead of writing that code from scratch, our design leverages an unmodified, third-party library called Rythm.js [9] that can make any website dance in time to the music. Rythm.js uses CSS classes (e.g., "rythm color1" and "rythm twist1") to represent background color and angle rotation changes to `<div>` tags.

Applying those classes to the servos and LEDs makes them dance just as well. List 3.4 shows the HTML and CSS implementation of our dancing speaker.

## 3.5 Impact on development time

Amalgam's goal is to make it easier for developers to build physical devices with rich interfaces. To quantify its effectiveness, we built two other versions of the video player: One using pure JavaScript and another using JavaScript and C.

The "C+JS" version uses a simple server implemented in C that exposes hardware components via a TCP socket. The JavaScript that implements the application logic communicates with it via TCP sockets.

**Figure 3.6. Our dancing speaker**

**List 3.4.** Dancing speaker code implementation using Amalgam-enhanced CSS.

```
<button onclick="playPause()" id="playPause"
  style="hardware: physical-button( gpio: var(--gpio5) )"> playPause
</button>  <!-- Play/Pause button -->
<button onclick="prevSong()" id="prev"
  style="hardware: physical-button( gpio: var(--gpio6) )"> Prev
</button>  <!-- Previous Song button -->
<button onclick="nextSong()" id="next"
  style="hardware: physical-button( gpio: var(--gpio12) )"> Next
</button> <!-- Next Song button -->
<input type="range" min="0" max="1" step="0.1" value="1" id="slider"
  style="hardware: physical-pot( adc-channel: 1, i2c-port: url('/dev/i2c-1')
      ,
  i2c-addr: 0x48"> <!-- Volume -->
<div class="rythm color1"
  style="hardware: physical-rgb-led( spi-port: url('/dev/spidev0.0') )">
</div> <!-- RGB LEDs -->
<div class="rythm twist1"
  style="hardware: physical-servo-motor( servo-channel: 0, i2c-port: url('
      /dev/i2c-1' ), i2c-addr: 0x48 )">
</div> <!-- Servo Motor 1 -->
<div class="rythm twist2"
  style="hardware: physical-servo-motor( servo-channel: 3, i2c-port: url('
      /dev/i2c-1' ),  i2c-addr: 0x40 )">
</div> <!-- Servo Motor 2 -->
```

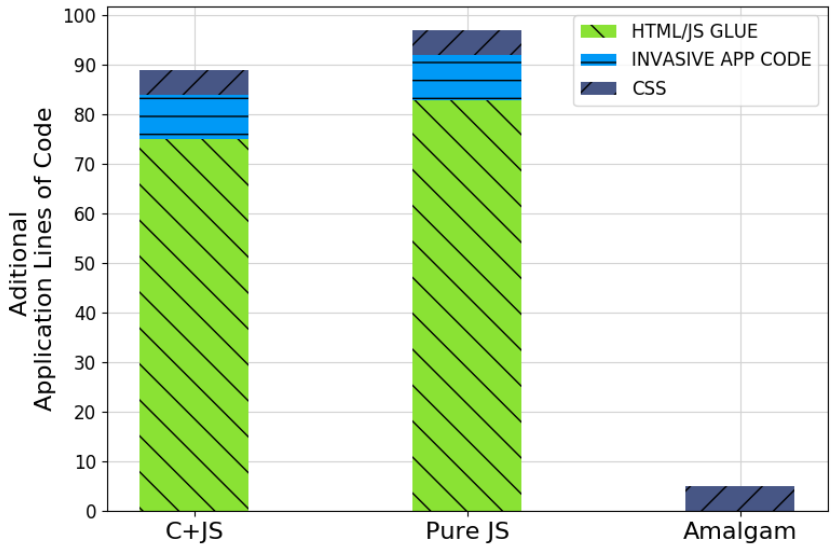



**Figure 3.7. Programming Effort:** Deeply integration of hardware components interfaces into the web languages allows Amalgam to reduce the lines of code needed to integrate these components into web application based electronic devices, therefore reducing development time.

The "Pure JS" version calls the HAL directly to control the hardware and implements the same functionality that Amalgam's hard elements implement internally. We refer to this as *glue code.*

Figure 3.7 compares the lines of code (LOC) required to integrate the hardware components into each version of the application. The measurements do not include the frameworks, libraries, or the server and communication code for C+JS.

We also include the lines of code added or changed in the application code to accommodate the change from soft element to hard elements (labeled as "invasive" changes).

The figure shows that Amalgam vastly reduces the effort required to harden components: five lines of CSS in one file compared to over eighty lines of JavaScript and CSS spread throughout the application for Pure JS and C+JS. Amalgam avoids invasive changes completely.

## 3.6 Related Work

Amalgam seamlessly integrates hardware components into HTML, CSS, and Javascript to reduce development effort and facilitate faster prototyping. Below, we place Amalgam in

context with other projects with similar goals.

### 3.6.1 Integration of Hardware to Web Technologies

Several previous projects have focused on the integration of hardware to web technologies. In particular, the Web of Things [51], and IoT protocols such as MQTT [70] and SOAP [33]. These IoT protocols use web programming technologies (e.g., HTTP, Web Sockets, XML, etc.) to interface remotely with hardware devices which have integrated sensors and actuators. As hardware devices become more powerful at a reduced cost [96] embedded developers are looking to use web programming technologies to also interface locally with hardware. Related efforts adapt JavaScript to run on constrained devices (e.g JerryScript [46]).

Web Browsers have become an extensively used platform that can run across heterogeneous hardware and software platforms, and they provide access to a limited number of hardware components like cameras and microphones [4] via standardized JavaScript APIs.

As web technologies are becoming popular on embedded, mobile devices, other standards for interfacing with hardware components have been included, such as Bluetooth low energy [10] and sensors like accelerometers, gyroscopes and ambient light detection [3]. Still, web browsers standards have not been able to keep up with the myriads of hardware components currently available.

Developers who want to use non-standard (or less common) hardware components with web technologies must do so in an ad hoc manner by developing custom communication protocols or "glue" libraries to provide access in JavaScript. Projects like Jhonny-Five [5] provide these facilities for some hardware devices, but it does not integrate cleanly CSS or HTML. It also does not provide easy access to generic interfaces like I2C and SPI, limiting its generality.

### 3.6.2 Adding Hardware to the Web's Semantics

Work has also been done to integrate hardware into the web programming languages but for remote devices. Projects like GSN [14], LSN [66] and OpentIoT [61], which facilitate the

integration of IoT diveces to their networks using IoT Protocols, have gone a step further and have also integrated sensors data into the web syntax. Semantic Sensor Network (SSN) [27] has been developed from these projects and adds sensor data and hardware components description (e.g., of temperature sensors and barometers) into the XML syntax. However, these systems focus communication among devices rather than communication between the software and locally attached hardware components. Amalgam is, in fact, complimentary to these efforts because it would simplify programming the sensor nodes.

### 3.6.3 Programming hardware in high-level languages

Abstracting hardware interfaces into high-level languages facilitates the development of devices by being able to use the high-level languages capabilities (e.g objects, garbage collection, etc). Previous work exists for adding hardware interfaces into the high-level languages. MircoPython [105] for example lets you program micro-controllers using the python language. Blockly [47] on the other use a visual programmer editor where ports exists to be used for different languages. By facilitating development these languages have been used to teach first-time learners how to program embedded devices [59]. These projects have similar goals (i.e., exposing hardware via a language's native idioms and programming constructs), but pursue that goal in the context of other languages.

### 3.6.4 Rapid Development of Embedded Devices

Many tools exist for the rapid software development of embedded devices. The Arduino Language [20], minimizes the time to develop of embedded software on microcontroller platforms by hiding their low level complexity behind a simple library. TinyLink [50] reduces the lines of code by providing tools that generate the underlying hardware interfaces and binaries required for a target platform. Microsoft .NET Gadgeteer [107] uses a modular hardware platform that is deeply integrated into the Microsoft Visual Studio IDE. Gadgeteer provides hardware abstraction libraries for each supported module and facilitate development by using C# as its

main programming language.

Amalgam is similar in some respects to both Arduino and the software support in Gadgeteer: All three projects aim to integrate hardware support into the host language (C for Arduino, C# for Gadgeteer, and Javascript/CSS/HTML for Amalgam). Amalgam, however, improves on the usability of the others by leveraging the flexibility and power of web programming technologies.

## 3.7 Acknowledgement

Chapter 3, in part, is a reprint of the material as it appears in the Proceedings of the 19th International Conference on Web Engineering 2019. Garza, Jorge; Merrill, Devon J.; Swanson, Steven, Springer Lecture Notes in Computer Science, 2019. The dissertation author was the primary investigator and author of this paper.

# Chapter 4

# Appliancizer

Appliancizer, our interactive design tool, builds on top of Amalgam and further fuses the PCB design, circuit netlist (schematic) design, and device simulation steps, making possible the automatic generation of Schematics, PCB design files, and low-level code from web pages plus the selection of hardware components. With Appliancizer web pages can be rapidly transformed into completely functional smart devices.

For device simulation, Appliancizer exploits the similarities between graphical and tangible interfaces, allowing the simulation of smart devices physical interfaces using HTML elements as mockup hardware components.

An encapsulation technique we call *essential interface mapping*, allows our tool to convert graphical user interface elements (e.g., an HTML button) into tangible interface components (e.g., a physical button) without the need to change the application source code.

Appliancizer makes the prototyping of mixed graphical-tangible interactions as easy as modifying a web page and allows designers to leverage the well-developed ecosystem of web technologies. We demonstrate how our tool simplifies and accelerates the design of complex electronic devices through the development of two smart devices with Appliancizer.

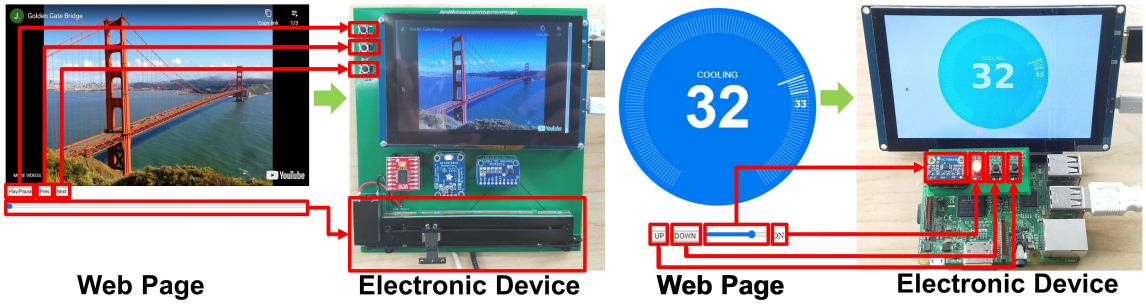



**Figure 4.1.** Appliancizer allows web pages to be transformed into fully functional PCB prototypes for rapid prototyping of smart electronic devices. Left a), we show a video player web page that provides a familiar on-screen interface for playing videos. Left b), a thermostat simulated on a web page with basic controls for adjusting the temperature. Right of the web pages shows automatically generated smart electronic devices running the same web application but with a tactile interface.

## 4.1 Introduction

Electronic devices play an essential role in our modern society. From the automation of many daily tasks, such as thermostats that maintain a comfortable temperature in our home, to multimedia entertainment devices. With the arrival of smart devices, such as smartphones, users expectations of their interaction with electronic devices have changed. Smart devices are expected to have rich interfaces that combine graphical displays with physical buttons and controls. Smart devices must also be connected to the internet and integrated into ecosystems to communicate with other devices [30]. Developing smart devices requires both designing a physical device as well as web-oriented interactions and rich graphical interfaces.

Web technologies such as HTML, CSS, and JavaScript have proven adept at creating interactive interfaces and facilitating integration with online services. The myriad available libraries make creating compelling animations, multimedia playback, and interacting with web-services possible with a few lines of code. These capabilities make web technologies ideal for designing interfaces for smart devices. For example, Tizen [95] and Firefox OS [83] use web-based architectures for smart devices. However, the many details involved with integrating web technologies with tangible interfaces slows new product development.

The conventional process for developing smart devices that combine graphical and tangible interactions requires substantial expertise in a range of specialties – user interface design, PCB design, and embedded programming [68]. This makes it hard to quickly, agilely, and cheaply prototype these complex devices [74]. Even though the accessibility to manufacturing and assembly services has improved, there is still a gap between having access to manufacturing technology and the skills and knowledge required to build these devices [73].

To bridge the gap, several design tools, such as JITPCB [21], EDG [91] and TAC [19], automate the generation of circuits from software by creating new hardware description languages at the circuit netlist or schematic-level. Enabling the programmatic or visual creation of circuits, without requiring expertise in schematic creation or PCB layout design. However, these tools do not target smart devices (that is, devices with screens and web connectivity). They also require developers to learn novel hardware description languages or syntax. To the best of our knowledge, no tool integrates graphical and physical user interfaces using existing web technologies.

In this paper, we address this shortcoming with Appliancizer, a computational design tool that allows designers to prototype electronic devices as web pages, which are then automatically converted into physical smart devices with tactile interfaces. Appliancizer provides an interactive online IDE[1] that allows users to simulate the interactions between hardware components and on-screen elements. Appliancizer guides users through the device design workflow, abstracting away the complexity of hardware design. With our computational design tool, the process of designing smart devices with both graphical and tactile interfaces (mixed interfaces) is simplified to only GUI and software development.

Compared to previous work such as JITPCB, EDG, and TAC, our approach uses ubiquitous web technologies and its rich GUI Model-View-Controller (MVC) model to create circuits. In Appliancizer, HTML is our hardware description language, therefore there is no need to add any extra configurations or change the source code of the web-based prototypes. This facilitates rapid prototyping of devices with both physical and graphical interfaces. As a consequence,

[1] Appliancizer interactive online IDE - https://appliancizer.com/.

web developers (and others without experience with embedded systems) may more easily create embedded devices.

Appliancizer uses a modular design approach that encapsulates multiple reusable circuit schematic pieces with their required low-level code and other circuit specifications, referred through the paper as *tangible controls* (e.g., physical buttons, sliders, LED's required schematics, hardware interfaces, low-level code, etc). Tangible controls are designed by experts and reusable by novices. Each tangible control maps to one of the three supported HTML GUI elements (buttons, sliders, and text). For example, both knob and temperature sensor tangible controls map to the HTML slider element. By combining the different HTML GUI elements, Appliancizer can automatically generate new circuit board designs and the required low-level code.

Appliancizer uses and demonstrates *essential interface mapping*. A novel technique that allows the automated generation of (i) a simulated interface of the smart device, (ii) schematic and PCB layout ready for manufacture, (iii) the low-level code required to make the device work. Appliancizer also explores the use of a custom web browser for easier application deployment, an approach enabled by our essential interface mapping technique.

Our tool is built on top of our previous work Amalgam [43]. Amalgam is a framework that extends the HTML DOM to the physical domain allowing software communication between graphical and tangible elements. However, Amalgam does nothing to assist the user with hardware design and application deployment. In contrast with Appliancizer, Amalgam users must design the PCB board manually, program the pin mappings, and learn complex Linux commands to run the web application.

The rest of this paper is organized as follows. Section 4.2 describes previous work and techniques for accelerating and reducing the complexity of the conventional development process. Section 4.3 and Section 4.4 introduce Appliancizer and our essential interface mapping technique. Section 4.5 demonstrates our tool and technique capabilities by describing two devices made with our tool. Section 4.6 evaluates with a preliminary user study the rapid prototyping capabilities of devices with Appliancizer. Section 4.7 discusses the advantages and limitations of our work.

Conclusions and final remarks for this tool are presented in the final chapter, Section 6.2.

## 4.2 Related Work

In this section, we review previous work that helps reduce the complexity of the design process required for the development and prototyping of electronics devices, as well as techniques relevant to this paper.

### 4.2.1 Rapid prototyping of electronic devices

Developing electronic devices is a complex process, therefore it is important to place our design tool in the context of the field. In this section, we discuss existing systems and how they interact with the different stages of electronic device development.

Several non-PCB prototyping systems have developed solutions to accelerate one or more steps of the conventional development process of electronic devices described in Section 2.

#### Tangible interface simulation

Many commercial tools [80, 89] offer low-level circuit simulation, targeted toward expert users. ThinkerCad Circuits [104], in particular, provides an accessible graphical environment allowing less-experienced users to simulate hardware component behavior and low-level code using a virtual breadboard.

Compared to these tools, Appliancizer uses higher-level abstraction. Appliancizer simulates interface behavior, not electrical circuits. Only essential interface events between application software and the hardware are simulated.

#### Rapid circuit prototyping using modular design

In modular design, hardware and software complexity is divided into different modules that can be reused. Modular design techniques are commonly used to achieve faster prototyping compared to the conventional development process. Multiple authoring tools allow rapid

interactive exploration of circuit designs at the physical level using a modular design.

Microsoft .NET Gadgeteer, MakerWear, littleBits, and SoftMod [107, 58, 24, 65] use prefabricated circuit modules as electronic building blocks that can snap together to create a range of electronic devices. These systems rely on proprietary toolkits. This limits support to only the proprietary hardware modules created explicitly for these systems. While toolkits attempt to standardize modules (for example, Grove [49]), these proprietary systems may be subject to limited future support and licensing fees.

In contrast, Appliancizer creates custom PCB prototypes using the universally used PCB schematics as modules. These PCB schematics can be directly used or modified to create a final commercial device. Additionally, schematic modules can easily be added to our tool as EAGLE files and updated without requiring expert designers to go through the additional steps to manufacture them. Appliancizer schematic modules also use the same off-the-shelf components (e.g., buttons, sensors, LED's, etc) that will be used in the final device.

**Accelerating low-level code development**

Modular design is also used for low-level code in .NET Gadgeteer, Arduino, and others [23, 20, 50] by providing higher-level libraries. These systems successfully remove some of the expert knowledge needed to program embedded devices (e.g. microcontroller registers manipulation) but still require user implementation of low-level communication with tangible interface components.

Appliancizer is built on top of Amalgam [43]. Amalgam deeply integrates different low-level code modules into the HTML and CSS syntax creating a clear separation between software code and low-level code. With Amalgam, the low-level code modules can be mapped to trigger HTML DOM events (e.g. onclick), therefore requiring zero low-level code development. However, Amalgam by itself still requires the user to know other low-level details such as pin mappings and Linux device file names.

Amalgam allows Appliancizer to present the same JavaScript events for tangible interface

components as graphical interface components. However, by using tangible controls and essential interface mapping Appliancizer completely abstracts all low-level details, making developing tangible interfaces as easy as developing graphical interfaces in HTML and JavaScript.

**Mechanical design and mixed interfaces**

Makers' Marks [94], RetroFab [90], and .NET Gadgeteer SolidWorks pluginprovide rapid design facilities for the integration of mechanical parts (and device enclosures) with tangible interface components. These systems often make a clear division between the mechanical, hardware, and software domains resulting in a mental gap for designers.

f3.js [57] works across domains and provides designers with a single codebase allowing users to interactively design the device enclosures with the aid of an IDE in conjunction with the device software code. This technique provides a mixed interface generation from a single codebase where one part of the code is utilized for the device functionality and the other for the device mechanical design.

Compared to these tools, Appliancizer focuses on helping users with the PCB design of the device and not on the mechanical design. Similar to f3.js, our tool provides a single codebase for the generation of a mixed interface interaction with the electronic device.

Amalgam allows Appliancizer to present the same JavaScript events for tangible interface components as graphical interface components. However, by using tangible controls and essential interface mapping Appliancizer completely abstracts all low-level details, making developing tangible interfaces as easy as developing graphical interfaces in HTML and JavaScript.

### 4.2.2 Automating PCB development

Current PCB design tools operate at lower abstraction levels, limiting the design process to the circuit netlist and PCB layout design tasks described in Section 2. Furthermore, as mentioned by Lin et al. these tools have stayed fundamentally the same over the years, without providing design assistance or improvements for novices [68]. Recent work has proposed

integrating the high-level abstractions of circuit design specifications into programming languages to allow the automation of multiple design steps. This transformation process is similar to the synthesis of hardware description languages (e.g., Verilog [103] and VHDL [82]) into gate-level blocks (for example, NAND, NOR, Flip-Flops) but for board-level circuits. Hence, this process is also called synthesis for circuits. Synthesis for circuits translates high-level hardware description languages into circuit netlist or schematic modules of commonly used circuit designs (for example, an LED with a resistor) instead of gate blocks. The benefits of synthesis for circuits include: accelerated development; avoiding the need for in-depth knowledge of electronics; and opening electronic design to new communities with a different set of skills, such as software developers.

Just in Time Printed Circuit Board (JITPCB) [21], Embedded Design Generation (EDG) [91] and Trigger-Action-Circuits (TAC) [19] are examples of tools that go beyond the traditional schematic capture process and provide board-level hardware description languages that allow synthesis for circuits. However, despite achieving faster development times and a notable reduction of complexity in comparison to the conventional development cycle, these synthesis tools integrate the hardware abstractions at the software level, requiring developers to learn extra development steps or limiting the functionality of the software application. JITPCB, for example, requires users to learn a separate programming language, Stanza [87], and manually specify which schematic modules to use in conjunction with the software. EDG requires users to learn an API to generate circuit diagrams. These two tools have benefits to experts doing low-level circuit design. TAC further abstracts the circuit specifications as a visual programming language but limits the application logic complexity by only having an if-then programming model [106]. In contrast to these systems, Appliancizer uses a well-developed programming language specifically designed for creating UIs with a focus on network interactions (HTML and JavaScript).

Appliancizer integrates the hardware abstractions at the HTML GUI level and leverages its rich MVC model to create circuits. In Appliancizer, HTML is the hardware description language, therefore, our tool is also defined as a GUI-based hardware description language for

board-level circuits. Combining this abstraction with our essential interface mapping allows for circuit synthesis without requiring users to learn extra development steps or even modify the software application source code. This allows Appliancizer users not to have to design schematics or PCBs layouts to create responsive, tangible interfaces

At the level of workflow automation, and taking as a model the diagram described in Figure 2.1, of the synthesis tools described above, only JITPCB can generate full PCB layout designs from a software programming language. EDG and TAC limit their work at the circuit netlist design step. Appliancizer's essential interface mapping technique allows workflow automation that spans from the device simulation to the PCB layout design tasks of the smart devices development process. Our tool generates complete PCB Gerber files that can be sent to manufacture without modifications.

### 4.2.3 Mapping graphical and tangible interfaces

A modular design concept that enables rapid prototyping has prevailed for the computational design of tangible interfaces that span from desktop computers to embedded devices. Especially post-WIMP, many of these tools have exploited the correlations that exist between graphical and tangible interfaces to accelerate prototyping.

Phidgets [48] first proposed packaging hardware components and their required low-level code into modules that can interact with on-screen GUI elements, allowing users to prototype tangible interfaces faster. Similarly, d.tools [52] uses the concept of phidgets to graphically prototype hardware and screen transitions of electronic devices. D.tools exploited these similarities by creating physical and virtual duals of components that are mapped to make tangible components interact with graphical ones. Further exploration of the mapping between graphical and tangible interfaces is done by iStuff [22] and the work by Coyotte et al [28] to accelerate the prototyping of tangible or mixed interfaces. These prior tools rely on a full desktop event handling system for mediating between physical device components and the software application logic. Appliancizer allows users to first prototype smart devices as GUIs and then

generate a fully functional physical version with minimal effort. Appliancizer is, to the best of our knowledge, the first system to use graphical-tangible interface mapping to fully automate the hardware design of standalone smart devices.

## 4.3 Appliancizer

To lower the PCB design entry bar and speed up the prototyping of mixed graphical-tangible interactions required for smart devices design, we created Appliancizer. Appliancizer is an online computational design tool that enables designers to transform web pages into fully functional PCB prototypes of smart electronic devices. With our tool, HTML-based interfaces are transformed into mixed interfaces, interfaces that include GUI and tactile elements. Compared to the conventional development process that requires designers to have extensive expertise in multiple fields, Appliancizer simplifies smart device PCB prototype development by requiring only front-end web design knowledge and minimal hardware knowledge. In addition, we release Appliancizer's implementation as open-source software[2]. In this section, we provide an overview of the design process with our tool.

### 4.3.1 Design process overview with Appliancizer

Appliancizer simplifies the design workflow required to prototype smart device PCBs by providing simple steps to build these devices. Figure 4.2 shows the interface of the tool. Designers first start by developing a front-end web application consisting of HTML, CSS, and JavaScript code, much as they would for a normal web-based application. Our tool provides a built-in front-end web editor for user convenience which can be accessed with the add a new web page button (a). Designers create their web-based prototypes with the clear idea that the prototype will be converted into a device and use the three supported HTML elements as mock-up hardware components to begin prototyping the interactions between graphical and potential tangible interfaces. The three supported HTML elements, listed in Table 4.1, are defined as

[2]Appliancizer source code - https://github.com/NVSL/Appliancizer.

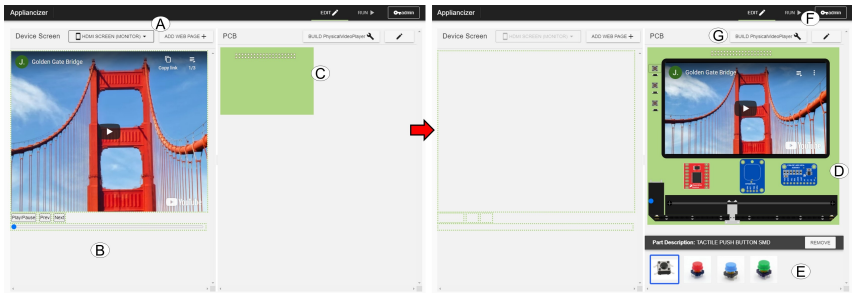



**Figure 4.2.** Appliancizer IDE. Figure shows a) button for adding web applications, b) virtual screen display area, c) empty virtual PCB, d) virtual PCB with virtual hardware components, e) tangible controls hardware components selection area, tangible interface simulation f), and a build button for generating the PCB layout, display required parts, and show application deployment steps g).

*HTML controls* through the rest of the paper.

After the new web-based prototype is developed, users can then upload the web application to a virtual screen display within our IDE that represents what users will see on the smart device screen. Our tool will highlight a green dotted border around the HTML controls that can be transformed into tangible interfaces. Otherwise, a red dotted border will be displayed (b).

The available HTML controls to transform can be dragged and dropped into a resizable virtual PCB (c) and (d), creating a clear separation between the graphical interfaces and the potential tangible interfaces. Each HTML control dropped into the virtual PCB maps to tangible controls which the user can select (e). Appliancizer supported tangible controls are listed in Table 4.2. Examples of tangible controls that the user can select are tactile buttons, LED's, and temperature sensors hardware components.

Accompanying components like resistors or any other hardware components required by the main tangible control hardware component are also automatically added to the virtual PCB. A simulation mode enables developers to simulate the potential tangible interfaces of the smart device (f). Finally, with a simple click, developers can get the list of all parts with a link showing where to buy them, Gerber files ready for manufacture automatically generated from the virtual PCB, and simple steps to deploy and run the web application on the new smart device d). The generated smart device for the virtual PCB in d) is illustrated in Figure 4.1 a).

### 4.3.2 Graphical to tangible interface process overview

With Appliancizer, HTML controls can be moved off-screen to become tangible interface components. Figure 4.3 shows a more detailed view of the process for transforming a graphical interface into a tangible one by showing how internally a button and span HTML controls are transformed into tangible interfaces with which a user can interact. After the HTML controls in a web page a), b) are dragged and dropped into the virtual PCB c). Our essential interface mapping matches the essential IO data similarities that exist between dragged HTML and similar hardware components. An HTML control button, for example, matches tactile pushbuttons perfectly as both produce an ON or OFF essential input data.

For each hardware component and accompanying components in the tangible control module, schematic modules describing the interconnection between the components are provided by our tool. Each schematic module represents an HTML control. By combining the different HTML controls, new circuit board designs can be generated. When ready for manufacturing, a complete schematic is generated from each separate schematic module d). A PCB layout is then generated from the complete schematic e). Finally, f) shows the final PCB after manufacture.

Using a modular design approach our tool also encapsulates the required low-level code for communicating with each hardware component. For deployment, a web application is also generated with the required low-level hardware code allowing its interaction with hardware components. A custom operating system image, provided by our tool, allows users to navigate and run the web application. Finally, at runtime, the selected HTML elements are replaced by *hard HTML elements* [43] consisting of web components [108] and low-level code that interface with the selected tangible controls hardware components. Figure 4.1 shows two examples of smart devices, a video player and a thermostat. Its PCBs were automatically created from a web page, and their hardened HTML elements allow physical interaction with the device's web application.

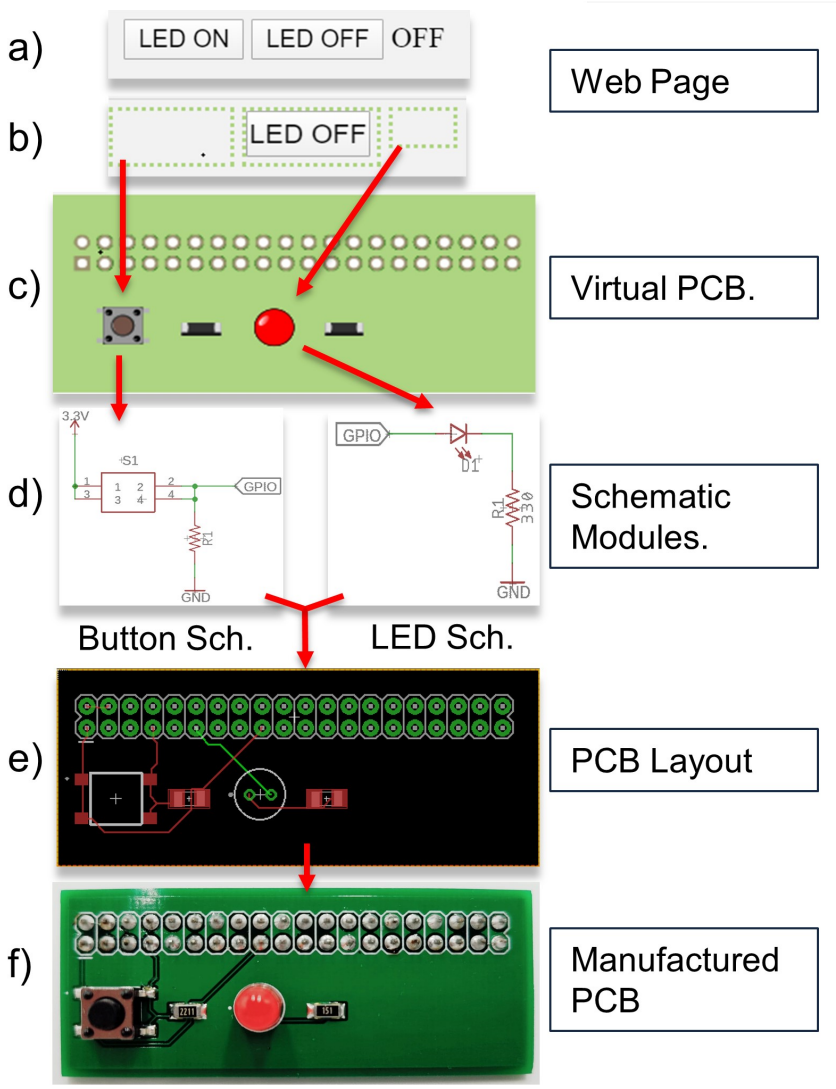



**Figure 4.3.** Graphical to tangible interface process. Two HTML controls a `<button>` and `<span>` are transformed into tangible interfaces while one HTML element, a `<button>`, is left as a graphical on-screen interface element.

### 4.3.3 Appliancizer tangible controls

To better explain our tool, we define as tangible controls as the encapsulation of high-level circuit specifications for hardware components (e.g. LEDs, pushbuttons, etc). The circuit specifications that are encapsulated into each tangible control and the method of integration with development tools affect the system capabilities. Examples of circuit specifications that can be encapsulated are low-level code for interfacing with the hardware components, circuit netlists, voltages, and hardware communication protocols required by the hardware parts, among others.

Figure 4.4 on the right shows an overview of all the circuit specifications that are encapsulated for each tangible interface component provided by our tool. Each Appliancizer tangible control consists of a schematic module, a low-level code block, required hardware components, and the hardware interfaces required to communicate with the hardware components.

The tangible controls concept has been implemented in previous tools, described in Section 4.2.2 and Section 4.2.3, each encapsulating different hardware features. Phidgets and Amalgam [43] for example have explored the use of tangible controls, but limit the encapsulation to contain only low-level code, leaving the PCB schematic and layout design to the developer. In contrast, Appliancizer tangible controls provide additional circuit specifications that allow the generation of PCB schematics and layouts. JITPCB, EDG, and TAC which similarly allow synthesis, limit the integration and exploration of tangible controls to the software level. On the other hand, Appliancizer integrates tangible controls at the graphical interface level, allowing the essential interface mapping of graphical and tangible modules.

Appliancizer tangible controls are designed by experts and reusable by novices. Compared to existing non-PCB rapid prototyping tools, experts designing these modules can add and update the modules as familiar plain-text PCB schematic files, rather than designing and manufacturing toolkit-compatible modules. Appliancizer supported tangible controls are listed in Table 4.2.

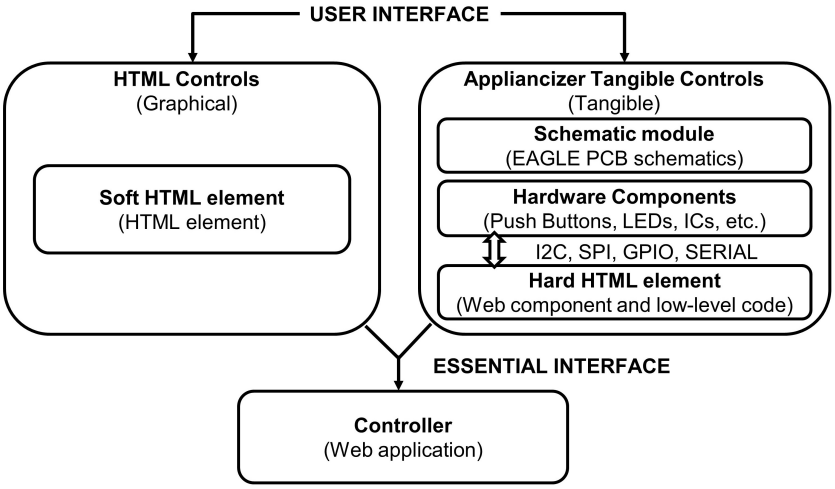



**Figure 4.4.** HTML controls vs Appliancizer tangible controls: To consider the wide range of hardware components we considered mapping at the essential interface level, thus encompassing hardware components that may not interact with the user, such as sensors.

## 4.4 essential interface mapping

By forming well-established mappings the translation between graphical and tangible elements, or to be more precise between HTML controls and tangible controls, can be performed, enabling design automation of smart devices that require rich GUI. In previous works, iStuff [22] and the work of Coyotte et al [28] limited the exploration of interface mappings at the user interface level. Therefore, preventing the analysis of hardware components that do not require direct physical interaction with the user (e.g, a temperature sensor).

While, from the perspective of HCI is not necessary, for fabrication the goal is to map as many HTML elements with hardware components as possible, even if the user never interacts physically with the hardware components. A mapping that covers a wide range of hardware components allows the creation of more diverse smart devices. Therefore, to cover a wider range of hardware components we suggest mapping HTML controls and tangible controls at the essential interface level. Opening the analysis of how graphical interface elements can also be used to prototype the behavior of hardware components that do require direct physical interaction.

**Table 4.1.** Mapping between HTML controls and hardware components

| *HTML controls* | *Essential interface* | *Hardware components* |
|---|---|---|
| `<span>` | Multi-byte Output | **Actuators**, LEDs, Relays, Servos, LCD Displays, etc. |
| `<input type="range">` | Numeric Input & Output | **Sensors**, Temperature Sensor, Potentiometers, etc. |
| `<button>` | 1-bit Input | **Interrupts**, Push Buttons, Status Signals, etc. |

Figure 4.4 shows a comparison between HTML controls and our Appliancizer tangible controls using the Model-View-Controller architecture model as a reference. Both the HTML controls and the communication of the Appliancizer tangible control with the controller are abstracted at the level of the essential interface, or the most essential data that is sent to the web application.

By studying the similarities of the HTML elements and essential interfaces of the different hardware components, we created a relational table that contains a mapping between the two. Table 4.1 shows our mapping table that we organize between HTML controls and a classification of different hardware components (e.g., sensors, actuators, etc.). One of the things we considered when creating this mapping is being able to address as many hardware components as possible with the least amount of HTML elements. This makes application development easier by not requiring specialized HTML elements to match specific hardware components. We describe each essential interface mapping in detail below.

### Multi-byte Output

A `<span>` or text HTML element provides information to the user. Therefore, we consider this element to be mapped to actuators that require one or multiple bytes of information sent by the web application to react, such as LEDs, servos, and LCDs. Once the interface becomes tangible, the data is transmitted to the low-level code and transformed accordingly for each hardware component. For example, for a LED, the web application must produce an ”ON”

or ”OFF” text to turn the LED on or off. In the case of a servo motor, the number of text in angles (e.g. ”45”) will make the servo rotate 45 degrees. Similarly, a 16×2 LCD screen will only display the raw text produced by the software application.

**Numeric Input & Output**

`<input type="range">` range sliders naturally match any sensor that produces a range of values. These sensors can vary from physical knobs or sliders that require human interaction to effect changes to temperature, light, and humidity sensors that interact with the environment. While range sliders are perfect for providing input data, web applications can also change the slider position through code, making the essential interface also a good candidate for outputting data to hardware components that require a range of values to actuate (e.g. servos).

**1-bit Input**

The `<button>` HTML element remains as the most intuitive for providing 1-bit input data to software applications. However, hardware components that produce 1-bit inputs are not limited to tactile pushbuttons. Integrated circuits that trigger interrupts when special conditions are met can be used as well.

**The screen component**

Mixed interface smart electronic devices require the use of a display to allow digital interaction with the device. Devices made with Appliancizer by default will display all the HTML GUI elements left in the virtual screen display. Developers can then connect any HDMI display to the device to interact with HTML elements on the device screen. However, in certain applications, the integration of the device screen with the PCB is required. For these applications, Appliancizer allows the `<iframe>` element to be dropped inside the virtual PCB where different HDMI displays with varying sizes can be selected allowing developers to populate the virtual hardware components adjusting to the screen display dimensions, as is illustrated in Figure 4.2 d).

**Table 4.2.** Appliancizer tangible controls library

| *Tangible controls* | *Essential interface* |
|---|---|
| LED (Red, Green, Blue) Thruhole & SMD | Multi-byte Output |
| Servo Motor | Multi-bytes Output |
| 16×2 LCD Display | Multi-bytes Output |
| Potentiometer | Numeric Input & Output |
| Motorized Potentiometer | Numeric Input & Output |
| Temperature Sensor | Numeric Input & Output |
| Tactile Push Button Thruhole & SMD | 1-bit Input |

### 4.4.1 Appliancizer tangible controls library

The implemented Appliancizer tangible controls that can be selected for each HTML control, listed by their essential interface, is shown in Table 4.2. Each off-the-shelf hardware component that can be used with our tool can come in different sizes and with variations, as is the case with LEDs that can come in thru-hole or surface mount device (SMD) versions and varying colors. With Appliancizer, low-level code can be written once and reused for similar versions of the hardware part. However, a schematic module must be created for each part to address the changes in sizes and shapes.

While some of our tangible controls use solderable breakout boards to facilitate assembly. With our approach, each discrete electronic component soldered onto the breakout board PCB can be directly integrated into the Appliancizer-generated PCB layout, as we do with the button and LED tangible controls.

### 4.4.2 Simulation of the device tangible interface

Device simulation enables developers to prototype and test the software of the device even before the device is manufactured, allowing experimentation and avoiding repeated prototyping to analyze the system behavior. Full simulation of the device hardware is expensive in terms of the computational resources needed. A popular and less expensive technique is to display an on-screen mock-up of the tangible interfaces that allow software developers to trigger simulated

essential interface data events sent to the software application by the hardware components. This approach is common in mobile application development [72], where virtual buttons and sensors are used to represent physical components such as volume, home, keyboard buttons, and sensors allowing developers to debug their mobile applications without requiring a physical device.

Our essential interface mapping technique inherently allows the creation of a virtual tangible interface for smart devices, since both, HTML controls and Appliancizer tangible control, are matched by the same essential interface data. After the HTML controls are removed from the virtual screen display and dropped into the virtual PCB area, developers can interact with the virtual hardware components placed in the virtual PCB when in simulation mode. Changes to the virtual hardware components will produce the same effects in the web application since it's the same web GUI but split into a graphical and a simulated tangible interface by the developer. Taking as an example the virtual button in Figure 4.3 (c), in simulation, any 'onclick' events will still be sent to the controlling web application code. Then, after PCB fabrication (f) and deployment of the application, pressing the physical button will trigger the same events in the web application as its virtual counterpart.

### 4.4.3 Automated PCB layout generation

Adding circuit characteristics to tangible control modules also allows Appliancizer to automatically generate the PCB layout required for the smart device. The PCB layout generation is divided into three tasks: hardware component selection, complete schematic generation, and hardware components placement. Below we describe these tasks and conclude with the PCB manufacture and assembly process.

#### Hardware component selection

Once the HTML controls are placed on the virtual PCB, developers can select among the hardware components implemented in Table 4.2, only those that match the essential interface mapping. Figure 4.2 (e) shows an example of the hardware components that can be selected for

the <button> HTML control.

**Complete schematic generation**

To automatically generate the complete schematic we use a modular design technique at the circuit level. In this technique, schematic pieces of commonly used electronic designs (e.g., LED and resistor, etc) are separated into modules, as is shown in Figure 4.3 (d). Each schematic module consists of a circuit with power nets, ground nets, internally connected nets, and disconnected nets.

Internally connected nets describe the connection of the main tangible interface components (e.g, a tactile button) with their accompanying hardware components (e.g, resistors, capacitors). Disconnected nets are left for the hardware interfaces (e.g., GPIO, I2C, etc) required for each hardware component. At build, each disconnect net is greedily assigned to the next available hardware interface of the computing device platform (e.g., GPIO interface to pin 4). In case there are no more hardware interfaces can be assigned we alert the user. Similar circuit netlist or schematic generation techniques have been implemented by JITPCB, EDG, TAC, and more recently Ehidna [75] for devices that use microcontrollers. In our tool, however, we target systems on chips (SoC) computing devices that can run operative systems which have become more popular in new embedded designs [37].

**Hardware components placement**

To generate a PCB layout the physical position of hardware components and the size of the PCB must be known. After the HTML controls are dropped into the virtual PCB and the desired hardware component is selected, a 2D virtual part image that represents the physical dimensions of the real hardware part can be positioned by the designer in the resizable virtual PCB. Accompanying hardware components for each Appliancizer tangible control can be moved as well. After virtual placement, the position of each virtual hardware component and the size of the virtual PCB are then used to create the PCB layout, described in more detail in Section 4.4.3.

Compared to tools like JITPCB, which require adding placement and rotation of hardware components programmatically, with Appliancizer, designers visually move and place the virtual hardware parts in their desired locations inside the virtual PCB. Although the hardware components placement could also be generated automatically, we choose to use a WYSIWYG methodology as we believe is more intuitive for the user.

**Generating the PCB layout**

After the complete schematic is generated and knowing the hardware components positions, the PCB layout generation process takes place. From the complete schematic, we generate the PCB layout positioning all hardware components inside the PCB to match the virtual PCB design. After placement, we run the auto-routing process provided by EAGLE, a third party PCB CAD used by our tool, to generate the routed PCB. Finally, after the PCB layout routing completes, the Gerber files are also generated using EAGLE. The Gerber files are then sent to a PCB fabrication service for its manufacture. Figure 4.3 shows in (e) the PCB layout after is routed, and (f) the PCB after manufactured. The physical hardware parts were assembled manually, however, PCB services can also provide assembly. To merge the different schematic modules and generate the PCB layout we use Swoop [99], a tool we created to edit, create, and manipulate EAGLE files. Only the automatic routing process and the Gerber file generation tasks are performed using EAGLE.

### 4.4.4 Automated low-level code generation

At build time, our tool also generates all the low-level code required to make the smart device work. The generation of the low-level is divided into two steps: A pin mapping process and the generation of the user web-application integrated with the required low-level code for the device.

**Pin mappings**

Appliancizer also generates hardware interfaces mappings at the software level, known as pin mappings. The pin mappings indicate which IO pins each tangible control low-level code should use and must match the schematic of the device (e.g., button GPIO interface to pin 4). tangible control are not limited to one hardware interface, Figure 4.1 device (a) shows a motorized potentiometer which requires two GPIO interfaces to move the motor and one I2C interface to get its position.

To automatically generate the pin mappings each tangible control is encoded with the hardware interfaces required by the low-level code to communicate with the hardware components. In the same complete schematic generation process described Section 4.4.3, the required hardware interfaces by each tangible control are greedily assigned to the next available computing device header pins. However, in this case, the resulting product is a Amalgam-enhanced CSS file used by Amalagam to map the low-level of each tangible control with the correct hardware pins and that match the generated schematic. The generated CSS file is used by Amalgam for the *hardening* [43] process to take place.

**Web application low-level code integration**

After the pin mapping process, we add to the user web application all the low-level code from each selected tangible control. Finally, the user web application, integrated with the required tangible control modules and a CSS file indicating to Amalgam how to harden the soft HTML elements (HTML controls) with the correct tangible control, is hosted on our server. A web link to the hosted web application is provided to the user which they can use to navigate to the generated web application integrated with low-level code.

### 4.4.5 Application deployment

The conventional application deployment process requires users to learn complex Linux shell commands for application deployment, making it challenging for novices. To facilitate

the process, we created a custom OS that allows users to simply navigate to the generated web link described in Section 4.4.4. Links are shareable making it possible for other users to test applications created by others. Our custom OS can be downloaded from our tool.

At boot, our custom OS provides a familiar web browser interface where users can introduce the generated web link to navigate to the generated web application. Our custom OS also contains Amalgam which is executed when a new web page is loaded, making our web browser behave differently. Unlike standard web browsers, our custom OS web browser will harden the selected HTML control by the developer, extending the HTML DOM to the physical world and allowing physical interaction with the web page. The hardening process is described in depth in [43].

**A smart device or a web page?**

One of the unique characteristics that our essential interface mapping technique offers is the capability of having two versions of the same application, one fully graphical and the other one with a mix of graphical and tangible interfaces. After our tool generates the web application integrated with low-level code, the web application is still accessible from a standard web browser and will render all HTML elements on the screen. However, when the same web page is accessed through our custom OS the user-selected HTML elements to transform are removed from the screen and their interface becomes tangible. Figure 4.5 shows the same web application described in Figure 4.3. Accessing the web application from a standard web browser will result in a fully graphical interface (a). Otherwise, if the web application is accessed from our custom OS with the generated PCB mounted, will result in a fully functional smart device (b).

## 4.5 Examples

To demonstrate our essential interface mapping technique we created two smart devices, a physical video player and a smart thermostat, using Appliancizer. Figure 4.1 illustrates these

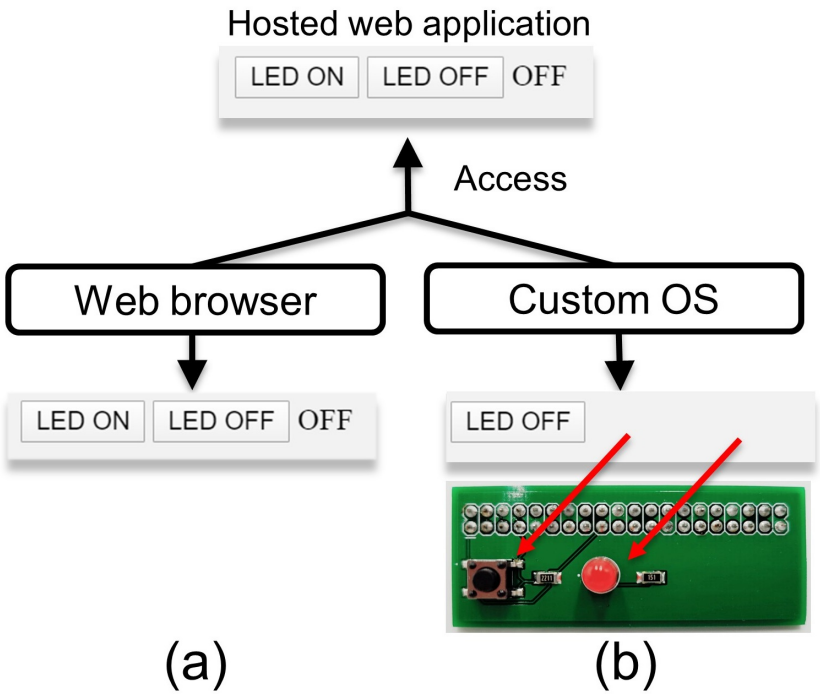



**Figure 4.5.** A essential interface mapping technique allows the same application to be accessed as a fully graphical interface application (a) or a mix of graphical and a tangible interface (b).

two electronic devices. Below we describe each smart device in detail.

### 4.5.1 Physical Video Player

The first smart device we created is a physical video player which was transformed into a smart device directly from a video player web application. To create the smart device we first built a soft video player using the YouTube API [111] by following a tutorial available online. Then, we loaded the application to Appliancizer and moved the player HTML controls to the virtual PCB, as shown in Figure 4.2. Finally, we sent the generated PCB to manufacture resulting in the smart device shown in Figure 4.1 (a). The physical video player device mimics all the behavior of the original soft version, including the progress bar which moves as the video progresses and which the user can actuate to change the video to a specific time. The user can press the physical push buttons to play or stop the video, and move to the next or previous video. No modifications were made to the source code of the web application for deployment.

Originally our physical video player only had a play and pause button. After testing the video player smart device we wanted to include physical previous and next buttons. Since our essential interface mapping technique makes web applications work as web-based prototypes of smart devices. The buttons were simply added as developing any other web page, without requiring adding low-level code or extra configurations. The old device PCB was replaced by the newly generated PCB and the custom OS web browser was simply refreshed to update the mappings, allowing physical interaction with the added buttons.

### 4.5.2 Smart Thermostat

For the second smart device, we took inspiration from the Nest thermostat [102] that provides an intuitive GUI for controlling the temperature inside a room. However, instead of following a tutorial and writing the code ourselves, we decided to search existing web applications from CodePen [25], a popular online editor for sharing front-end web applications. We found that the use of HTML elements as hardware mock-up components is an approach intuitively used for prototyping smart devices graphically and quickly discovering a web-based NEST prototype [25]. Although, the initial proposal of the web application shared by the community was to graphically simulate the Nest thermostat tangible interface, with Appliancizer we transformed the web application into a real smart thermostat.

The soft version of the smart thermostat includes two HTML buttons to increase or decrease the target temperature, an HTML span tag that indicates if the weather to turn ON or OFF the cooling compressor, and finally, a range slider that simulates the ambient temperature of the room. Correspondingly, the built physical version of the devices consists of two physical buttons, a LED that physically simulates the ON and OFF state of the compressor, and a temperature sensor that indicates the ambient temperature with a range of values. However, compared to the motorized potentiometer that also provides an interface to programmatically control the position of the slider using a DC motor, the temperature sensor is limited to only transmitting an input range of values to the web application. Therefore, any initial value set to

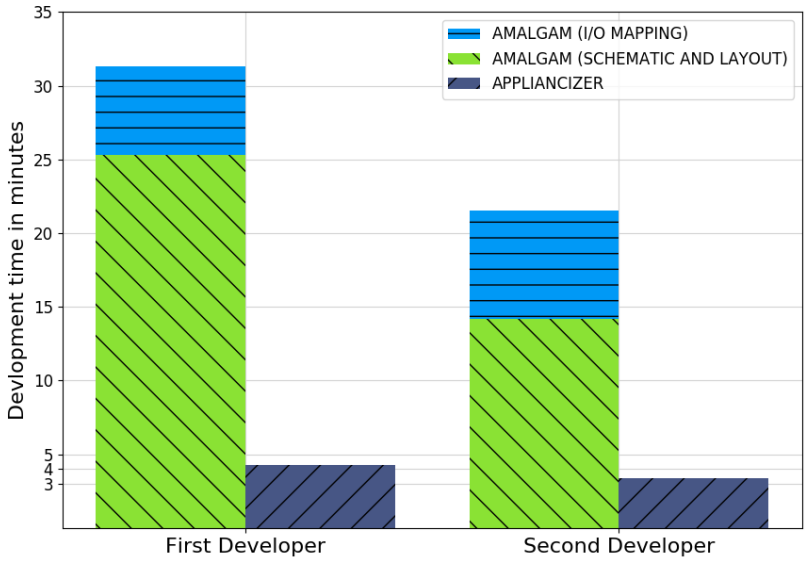



**Figure 4.6.** Development time comparison between Appliancizer and Amalgam for designing a smart thermostat.

the soft slider version will be ignored by the low-level code of the temperature sensor tangible control. Finally, for this device we left the screen component as a soft element, allowing the PCB to be placed at the back or at a different angle from the screen display.

## 4.6 Preliminary Study

To gauge how our essential interface mapping technique speeds prototyping of smart devices that require mixed interfaces we ask two developers with prior experience designing PCBs to build the same smart thermostat described in Section 4.5.2 using both Appliancizer and Amalgam. The web application of the smart thermostat was provided for both tools. For Amalgam, the hardware footprints library was provided as well, requiring users to only select the hardware components for laying out the PCB.

Figure 4.6 shows the development time difference between Appliancizer and Amalgam. It took developers an average time of 4 minutes with Appliancizer versus an average of 26 minutes with Amalgam to design a smart thermostat device. A speed-up of 6.5X in development time with Appliancizer. These results were expected since Amalgam, compared to Appliancizer, requires

users to generate the schematic and PCB layout manually. Furthermore, with Amalgam, users also require to program the pin mappings for the low-level code to work with the selected computing device. These two separate tasks are shown correspondingly for Amalgam in Figure 4.6.

## 4.7 Discussion

The conventional design, test, and prototype process for smart devices that require mixed graphical and tangible interfaces are composed of complex tasks. To mitigate the challenges we introduce a essential interface mapping technique that allows transforming HTML GUI elements within web pages into tangible ones. To demonstrate our technique we created Appliancizer, an online computational design tool that allows designers to generate functional smart devices from web-based prototypes. In this section, we discuss some observed capabilities and limitations of our essential interface mapping technique and compare Appliancizer to previous work.

### 4.7.1 Rapid prototyping of smart devices

While previous tools have achieved synthesis for board-level circuits [21, 19, 91], they limit the high-level hardware abstractions of circuits at the software code level, requiring developers to learn complex integration steps or limiting the software application functionality. In our work, we consider the abstraction at the graphical interface level for smart electronic devices that require graphical and tangible interfaces. Previous tools at this level have demonstrated a relationship between graphical and tangible interfaces [48, 22, 28]. However, they are limited to systems that require the desktop computer as a mediator and are incapable of generating full stand-alone devices. In contrast, our tool takes advantage of these similarities for synthesis, allowing us to transform graphical interfaces into tangible ones and automate different tasks of the design process, without requiring developers to change the application source.

Using a essential interface mapping technique, the development of a smart device comes down to selecting which HTML elements to transform and selecting the desired hardware components that match the essential interface of the selected HTML elements, enabling rapid

prototyping down to the circuit board level design.

Section 4.6 demonstrates Appliancizer rapid prototyping of smart devices. Amalgam developers spend a significant part of their time creating schematics and PCB layouts of the same circuit blocks. Comments from one of our users acknowledge that

”even remembering how to connect commonly used circuit designs takes time”

After comparing the PCB layouts generated automatically by Appliancizer and the ones manually designed by both users, we discover no significant differences in the routing or design.

### 4.7.2 Fast iterations

Compared to the conventional design process of standard PCB software tools that require developers to re-design the PCB and implement new low-level code after adding or removing hardware components, our tool, on the other hand, provides a faster iteration process by automating these tasks. With Appliancizer users only need to choose to leave some items inside the web page as graphical elements and make others tangible, enabling fast iterations of the same smart device design.

### 4.7.3 Web-based prototypes

Aside from board-level rapid prototyping, our essential interface mapping technique enables users with the capability to rapidly explore physical prototypes using HTML GUIs, also referred to as web-based prototypes. For our smart thermostat application, even if the web community of original developers for the smart thermostat web application ruled out the idea of having a physical version of the device, they were able to efficiently explore the physical interaction with the device using HTML elements as mock-up hardware components. Even before the existence of our tool, the use of GUI elements as mock-up hardware components seems intuitive for designers for prototyping smart device’s physical interactions with on-screen elements. Appliancizer’s simulated tangible interface provides the user with a clear division of on-screen HTML elements and those used as mock-up hardware components. This allows a

clearer exploration of the interactions between the graphical and tangible interfaces of the smart device. After interaction design exploration using the simulated tangible interface, our tool can then transform the mock-up hardware components into real tangible interfaces.

### 4.7.4 Web browsing smart devices

Another observed ability of our essential interface mapping technique is the ability to have two versions for the same web application, one that interacts fully digitally with the user and another in which some HTML elements can physically interact. Since both versions are encapsulated in the same application, applications created with Appliancizer inherit all the properties of web technologies. Therefore, allowing smart device applications to also be requested, displayed, and executed using standard web browser techniques. Our custom operating system implements the idea by allowing users to navigate to web applications that contain the low-level that makes the smart device work. To run the physical video player and smart thermostat applications, the only required modification is to change the PCB generated and navigate to the web URL link provided by Appliancizer. Finally, if the web application source code is modified, a simple page refresh allows the device to be updated.

Since our custom works like a browser for smart devices, users can navigate to smart devices web applications created by other developers and easily test their functionality after the PCB for the corresponding web application is replaced.

### 4.7.5 Limitations

Our tool has several limitations including limitations of the auto-routing algorithms which limit the complexity of supported PCB layouts and the compulsory use of JavaScript which may limit real-time performance. However, we narrow the discussion to the limitations presented in our essential interface mapping technique:

### Limited HTML controls

Appliancizer supports a limited number of HTML elements that map to tangible interfaces. HTML radio buttons, checkboxes, lists, and others could potentially be used to map tangible interfaces allowing more intuitive web-based design prototypes of smart devices. For a physical switch, a `<span>` HTML element with changing text ON and OFF is well supported with our current implementation. However, a checkbox HTML element may provide a better fit when prototyping.

### Expressing hardware components capabilities and limitations

Hardware components have different capabilities which make them not always fully match with their HTML element counterpart. In Figure 4.1 the motorized potentiometer and the temperature sensor both map to the HTML slider element. The HTML slider element's value can be set pragmatically. However, while the motorized potentiometer also can be set programmatically (with some latency) the temperature sensor component cannot. Therefore, designers must be aware that, for some hardware components, the behavior will not exactly match the HTML element behavior when the graphical interface is transformed into a tangible one.

### Mapping limitations

A essential interface mapping that reaches the most number of hardware components allows creating more varieties of smart devices. However, our current essential interface mapping approach is limited for some hardware components.

Appliancizer is well fitted for hardware sensors and actuators that require low essential data transfers. Hardware components that require transferring intensive information such as SD Cards, Ethernet ICs, etc are not suitable for our tool.

Among the suitable hardware components, our tool also presents some limitations. For example, mapping hardware components that require multiple essential interfaces. An example

of one of these hardware components is the inertial measurement unit (IMU) that reports acceleration in three different axes (e.g, X, Y Z). For each axis, a numeric range of values is produced. This requires more than one slider element to fully map the IMU essential interfaces.

## 4.8 Acknowledgement

Chapter 4, in part, is a reprint of the material as it appears in the Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems. Garza, Jorge; Merrill, Devon J.; Swanson, Steven, Association for Computing Machinery, 2021. The dissertation author was the primary investigator and author of this paper.

# Chapter 5

# TypedSchematics: Block-based Electronic Schematic Design with Real-time Detection of Common Errors

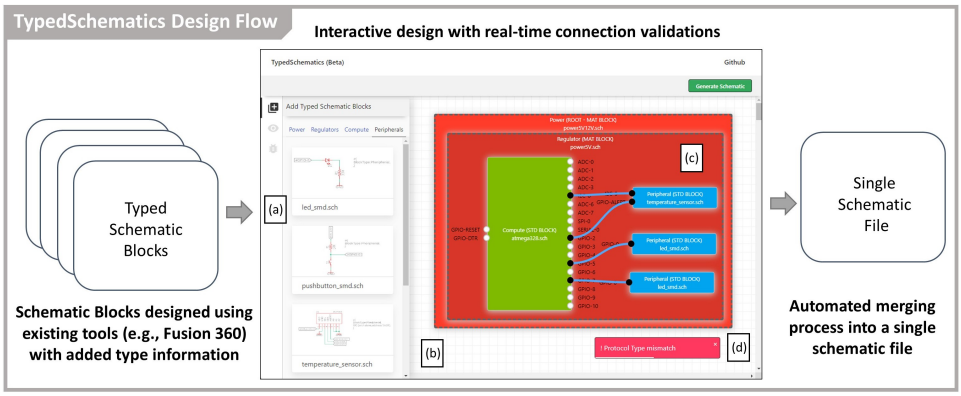



**Figure 5.1.** In the TypedSchematics design flow, the community creates reusable schematic blocks using conventional schematic capture tools. Input and output signals of schematic blocks are typed with information using our annotation syntax. Typed schematic blocks are imported to our interactive tool and usable by any designer. A simplified schematic representation model reduces the number of interactions required to connect schematic blocks. Real-time connection error validation allow schematic blocks to be connected safely. Schematic blocks are automatically merged into a complete schematic in a standard format, allowing designers to quickly proceed with PCB design.

Appliancizer's encapsulation of modules containing circuit information enables the fusion of multiple electronic design steps and design automation. However, when it comes to editing these modules or scaling the library of existing modules, the process is complex and unintuitive, often requiring in-depth knowledge of the tool's internals. Furthermore, with

Appliancizer the pin mapping is done automatically, not allowing the user to select how to connect the different modules. In summary, Appliancizer has narrow design walls when it comes to circuit configuration and design.

In this work, we introduce Typedschematics, an interactive design tool that primarily focuses on an encapsulation technique to make modules easier to configure and scale, while providing designers with the ability to configure module connections and compose them interactively. In other words, Typedschematics seeks a balance that allows for fast generation of design but with highly configurable modules that contain circuit information, modules which are referred throughout this work as blocks.

To create a better encapsulation technique, TypedSchemaics first focuses on the schematic design step of circuit design, allowing designers to merge different electronic schematic blocks. The design of the PCB layout is reserved for future work. A real-time detection of common schematic design errors is introduced to safely compose schematic blocks. In addition, we are introducing a new schematic model representation for schematic blocks that reduces interaction costs and speeds up the design process.

We demonstrate the capabilities of our tool with examples as well as a user study comparing TypedSchematics to Fusion 360, a schematic design tool with schematic blocks creation capabilities often used by beginners.

## 5.1 Introduction

Interactive electronic schematic design has been established as a de facto process that precedes PCB layout design. In this interactive design, users draw schematics by placing virtual electronic component symbols on virtual sheets and connect their signals together, a model used in many popular schematic capture tools (e.g., KiCad, Altium, Fusion 360). The falling cost of circuit board manufacturing processes has made hardware design more accessible than ever before, opening up professional-quality circuit design to students, hobbyists, and others but

making the weaknesses of schematic capture tools more evident.

The growing community of circuit designers has noted that one of the biggest problems with schematic capture tools is the process of reusing and merging schematic blocks [68]. While some current circuit design tools provide some forms of schematic reuse, these mechanisms do not provide information about the capabilities of the blocks, for example, signal types, required voltages, among others. This makes composing reusable blocks to build new designs fraught with design challenges. Design challenges that constantly lead to connection errors and require designers to carefully analyze each schematic block they wish to combine, making schematic block reuse a less popular option among circuit designers.

Commercial block-based circuit design tools such as Sparkfun's À La Carte (ALC) [38] and Altium's Upverter Modular [18] have taken small steps to improve this situation. These tools allow designers to choose different pre-tested circuit blocks, such as microcontrollers, sensors, and connectors, and allow their integration to create complete new designs. Yet, although these tools have novel interactive design flows, the processes for merging schematic blocks still requires one or more engineers to manually create the full schematic as well as to check for any connection errors, an engineering design fee that can approach $1,000 per design [39].

While popular opinions deny the need for better reuse processes [16], the rise and interest in tools that can simplify the reuse process for schematics blocks suggests otherwise. Comparing the current state of reuse capabilities in schematic design with software design would be equivalent to software designers figuring out which data types (e.g., bool, int, char) are needed as input parameters and return values for each function by reading its code. In addition to understanding the input and output data types required by functions, data type information in software makes possible the detection of common data type errors at compile or real time, also known as type checking, a form of constraint validation. In schematics, the analog for software's data types are electrical interfaces that specify the compatible voltages, electrical currents, and data transfer interfaces (e.g., Serial, I2C, or PCIe). These interface types, if properly encoded, can provide "type" checking for hardware designs [43, 18, 44].

Although recent work [67, 21] has explored the addition of type information as well as the use of constraint validation mechanisms of connections for building circuits by proposing the use of board-level hardware description languages (HDLs). These tools are code-oriented and decoupled from the traditional visual and interactive schematic and circuit design flow, requiring circuit designers to learn programming languages. A learning task that requires a steep learning curve for novices or beginners without the help of visual and interactive interfaces. In addition, the use of code-based schematic model representations makes it impossible to reuse existing schematic designs. To date, there is no visual interactive tool that integrates the benefits of adding type information to schematic blocks, as well as implements error detection mechanisms in the merge process.

In this paper, we address these shortcomings with *TypedSchematics*, an interactive design tool that allows designers to annotate existing schematic blocks with type information enabling real-time detection of connection errors. To detect connection errors, TypedSchematics implements a similar approach to type checking using hardware protocol interfaces as a base which we call *interface checking*. A method that makes schematic blocks reuse and the merging process safer and faster.

Compared to board-level HDLs, TypedSchematics leverages the already established traditional design flow for easier adoption. An expressive syntax facilitates annotating schematic blocks with type information for easier integration into our interactive tool.

In contrast to existing commercial solutions, such as Sparkfun ALC and Altium Upverter, which require external engineers to integrate or merge schematic design blocks, our typed syntax makes possible the automation of the merging step, allowing the rapid and automatic generation of full schematic designs.

In creating TypedSchematics, we noticed that in the process of merging schematic blocks, connecting power signals (e.g., Ground, VDD) becomes a repetitive task, as well as causing a pile up of wires or tags, producing a less clear design layout.

To create a clearer layout, TypedSchematics also introduces *"Mats"*, a new visual

schematic representation model for schematic blocks that uses a combination of tree and graph structures. This new approach allows breaking down the traditional structure consisting of a single schematic graph into one that is more efficient for schematic blocks, eliminating unnecessary signal connections and reducing interaction costs.

A data flow programming paradigm is also seamlessly integrated with the visual schematic representation model for running interface checking on block connections.

Overall, the contributions of our work are as follows:

- A method of adding type information to existing schematic blocks that enables real-time detection of connection errors, facilitating the process of merging schematic blocks using interactive designs.
- The introduction of Mats, a schematic representation model for schematic blocks that reduces design interaction costs.

The rest of this paper is organized as follows. Section 5.2 describes previous work and techniques for reusing schematic designs and discusses existing block-based interfaces for circuit design. Section 5.3 explains some of the design challenges with schematic blocks. Section 5.4 introduces TypedSchematics. Section 5.5 demonstrates our tool capabilities by describing two devices made with TypedSchematics. Section 5.6 and Section 5.7 demonstrates how TypedSchematics addresses the design challenges with a user study. Conclusions and final remarks for this tool are presented in the final chapter, Section 6.3.

## 5.2 Related Work

Our work focuses on a new interactive schematic design flow that improves schematic blocks merging process by making use of the addition of type information to schematic blocks enabling type checking, as well as the introduction of ”Mats”, a new visual schematic representation model that reduces design interaction costs. In this section, we review previous work related to our research.

### 5.2.1 HCI and Circuit Design

Recently, there is a growing interest in the HCI community in tools that help and assist circuit designers in the different stages of circuit development. Previous research has work on projects that assist designers in the initial circuit prototyping stages that use breadboards [62, 32]. Other works have focused on improving the debugging and visualization of analog and digital circuits data like currents and voltages [97, 109]. While other works [53, 98] focus on the final stages of circuit design, such as supporting debugging and mass production of PCBs. Our work, on the other hand, focuses on the schematic design stage, a stage that usually follows after the initial prototyping of circuits on breadboards or development boards. Schematic design is also a preliminary stage to PCB design. Schematic design provides the designer with a simpler visual interactive format for connecting electronic components, where even dimensional blocks are used as opposed to irregular electronic components footprint shapes.

After schematic and PCB layout design, PCBs are manufactured, providing a more stable and durable circuit packaging format instead of loose wires. In recent years, with the miniaturization of electronic components and the reduction in PCB manufacturing costs, has led some designers to skip the initial prototyping stage with breadboards and use the schematic design and PCB layout stages as the main prototyping platform, making schematic capture tools more relevant than ever.

### 5.2.2 Schematic Capture Tools and Design Reuse

Commercial PCB design tools like Fusion 360 [34], KiCad [60], and Altium [15] primarily consist of a schematic capture software and a PCB layout software. Our tool focuses solely on the schematic design aspect of the PCB design process. Modern commercially available schematic capture tools have methods for reusing schematic blocks. Below we explain these methods and their limitations.

**Hierarchical Sheets** also known as Multi-Sheet Hierarchical Design [17] allows designers to abstract complexity from schematics into reusable blocks. In this method, the designer creates a schematic subsheet and selects signals in the design to export. On another sheet, designers can draw one or more reusable blocks that reference the created subsheet and exported signals. The exported signals serve as inputs and outputs for the reusable block to be connected with other elements within the schematic sheet. Fusion 360, KiCad, and Altium all implement some form of hierarchical sheets.

Hierarchical sheets provide schematic reuse, however it is limited to the same project, without the ability to export the design to other projects.

**Design Snippets** [16] in Altium and similarly ModularDesignBlocks [35] in Fusion 360, its an improvement on reusability, allowing parts of schematics to be copied and pasted into other designs. This method allows the export and import of schematic blocks, however it does not have any abstraction that clearly defines the inputs and outputs of the blocks, thus behaving more like a sophisticated copy and paste tool.

**Device Sheets** [16], created by Altium, allows having hierarchical sheets with the ability to be exportable and used in other projects. Device Sheets can also be versioned, called Managed Device Sheets. Device Sheets is the most advanced form of schematic block reuse in commercial available schematic design tools.

While previous approaches allow reuse of schematic blocks, the lack of connection error detection mechanisms prevents designers from safely connecting schematic blocks together. In a small group of expert designers who can communicate with each other efficiently, the use of connection validations may be pointless. However, when it comes to exporting schematic blocks to larger communities of both experts and novices, connection validations are a must and currently non-existent. The lack of connection validations may be one of the reasons why

schematic block reuse in the hardware community is not as common as it is in the software community.

### 5.2.3 Block-based Circuit Tools

Block-based or modular design is an efficient approach for reusing designs; this method relies on the encapsulation of information to abstract parts of a system, used in both software and hardware. Altium Upverter formerly Gpetto [18] and SparkFun À La Carte (ALC) [38] are block-based design tools that encapsulate circuit design information for reuse. ALC separates hardware components by functionality (e.g., power, sensors, connectors) and allows designers to select hardware components using a shopping cart-style layout. However, designers are limited to selecting blocks and cannot select the block inputs and outputs to use. Altium Upverter goes further and provides a UI that allows designers to connect blocks inputs and outputs using drag and drop menus. Altium Uperter UI also allows dragging and dropping circuit blocks onto a virtual PCB, used later for PCB fabrication.

While these tools allow rapid creation of circuit designs the circuit blocks are fixed and are only extensible by the developers of the tools. In addition, the process of merging and validating circuit blocks connections is left to external engineers.

None of these tools have the ability to add new community-designed circuit blocks to the tools, as well as algorithms for validating connections. These processes fall to the engineers behind the tools.

### 5.2.4 Higher Abstraction Tools for Circuits and Design Reuse

Previous research work has focused on increasing the abstraction of circuit design by making use of modular design and reusing common circuit designs. Trigger-Action-Circuits [106] reuses popular circuit implementations and abstracts the design to a behavioral level and makes possible the automated generation of circuit diagrams from a if-then programming approach. Appliancizer [44], on the other hand, abstracts the circuits to the GUI level, allowing for schematics

and pcb layout automated generation from web interfaces. While these tools allow the creation of circuit designs almost automatically, the high abstraction eliminates the ability to configure low aspects of the design, for example, pin and wire connection selections between circuit blocks are impossible in these tools. Also, circuit blocks require a large amount of hardware information. Making the encapsulation syntax complex, requiring expert developers familiar with the tool to create these blocks. For example, Trigger-Action-Circuits block developers require XML knowledge, while Appliancizer block developers require HTML/CSS knowledge.

Unlike these tools, TypedSchematics does not try to automate the synthesis of schematics, it aims to make the schematic design process easier and less error prone.

### 5.2.5 Board-Level HDLs

Other work, such as Polymorphic Block [67] and JITPCB [21], enable reuse of circuit design blocks with connection constraint validations with the use of hardware description languages (HDLs) for board-level circuits. These tools use code to represent the connections between electrical components or blocks of components instead of using visual circuit diagrams as schematics. An advantage of using HDLs to build circuits is the ability to create programs that can recommend circuit blocks to use or even build entire circuits automatically by running constraint validation solvers until the correct circuits are found that fit the system requirements. While this approach has many benefits in terms of automation, it requires learning a completely different set of tools and workflows (e.g., code syntax, compilation process, libraries), resulting in designers having a steep learning curve compared to the traditional use of visual design workflows.

### 5.2.6 Visual Schematic Representation Models

The visual representation model of schematics consists of connections of circuit symbols (e.g., resistors, capacitors, integrated circuits) and uses a graph structure algorithm, using nodes to represent symbols and edges for possible connections. This visual model has not changed

much since the standardization of circuit symbols [13] and the introduction of the first schematic capture software [71].

However, the search to abstract the complexity of circuit design and the appearance of new circuit design surfaces, in addition to PCBs, has led to rethinking the model for specific applications that bring special benefits to their users. Aesthetic Electronics [69], for example, combines the schematic model with graphic design tools bringing improvement to the fabrication of circuits embedded in fabrics or ornaments. Appliancizer [44] combines web design (HTML, CSS, JS) with circuit abstractions for the rapid creation of gadgets that feature screens. While Polymorphic blocks [67] uses code as the basis of representation of the schematic model which allows the automation of circuit designs. Our tool as well, adds enhancements to the visual schematic representation model for the particular use with schematic blocks.

## 5.3 Design Challenges

Schematic design using schematic blocks presents unique challenges compared to traditional schematic design. While in traditional schematic design, designers can access data sheets to understand how to connect the electronic component symbols. Current development with schematic blocks lacks standards to provide schematic blocks information, leaving designers in the dark when it comes to connecting blocks made by other designers. Below, we describe in detail some of the challenges that arise when composing schematic blocks with the lack of information.

### 5.3.1 Hardware Protocol Connections

Hardware protocols (e.g., I2C, SPI, GPIO) have become a popular way for transmitting information between ICs (Integrated Circuits), making these protocols ideal for connecting separate circuit designs, and therefore often used in block schematics.

These hardware protocols range from a single wire (e.g., GPIO) to any number of wires, and each protocol has different special ways of connecting these wires. A single mistake

connecting one of these wires can produce a circuit error. For example, I2C, which consists of a two-wire protocol called Serial Clock (SCL) and Serial Data (SDA), cannot mix its wires (e.g., SCL with SDL). Knowing how to connect these protocols creates the first challenge when combining schematic blocks, requiring designers to know these protocols properly in order to connect them correctly.

Furthermore, these protocols also integrate different capabilities making them have connection restrictions in special circumstances. One example is the hardware protocol SPI which uses three wires to transmit data, one clock wire (SCK) and two data wires (MISO, MOSI), allowing faster data transfer rates than the I2C hardware protocol. This protocol requires one IC to be configured as Master and another IC as Slave, making other configuration connections impossible otherwise. Similarly, the I2C protocol requires each IC to have a unique 7-bit address (e.g., '0$\times$1F'), making duplicate I2C addresses connections imposible. If any of these extra capabilities are not met, the circuit fails, making this another challenge when it comes to connecting schematic blocks together, since this information is rarely included in schematic designs.

### 5.3.2 Supply Voltages

All schematic blocks require at least a voltage and ground to power the block's internal circuitry. The required voltage may vary depending on the schematic block design, with some requiring a fixed voltage (e.g., 3.3V, 5V) and others allowing a range of voltages (e.g., 3.3V-5V), depending on the block's internal ICs or circuitry capabilities. Existing schematic design tools lack the ability to annotate supply voltage information on schematic blocks, making finding the required block's voltage another design challenge. Since all schematic blocks require power and ground signals, connecting these wires becomes a repetitive task, often resulting in stacking of wires in the design.

### 5.3.3 Unknown Signals and Optional Connections

While block diagram designers generally try to annotate input and output signals with meaningful names so that it is understood if these signals are part of a protocol (e.g., SDA, SCL) or not. Since there are no standards for annotating signals, the block signals and their capabilities are generally unknown. Making designers have to review and understand the entire schematic and even the datasheet for each component. Therefore, making this is one of the biggest challenges in schematic blocks design. An unknown signal example might be a signal named ”Reset”, which although the name is meaningful, does not indicate whether it is part of any hardware protocol or its connection capabilities.

Finally, there are signals that can be optional. Interrupt signals, which serve to signal microcontrollers when data is ready, are one example. Interrupt signals prevent microcontrollers from needing to wait for information in a loop. While the firmware in some designs takes advantage of this signal, simpler designs that constantly pull the data do not require it. Not knowing whether a signal is optional presents another challenge, leaving designers unsure whether or not specific signals are necessary for the block's proper function.

## 5.4 TypedSchematics

To improve the process of merging electronic schematic blocks and reduce its design challenges, we created TypedSchematics [1]. TypedSchematics reduces errors in building complex schematics by enabling reuse and automatically enforcing constraints. TypedSchematics allows designers to add type information to schematic blocks, enabling the use of real-time connection validation mechanisms and allowing for effortless integration with other schematic blocks. Once schematic block connections are complete, full schematics are generated automatically.

TypedSchematics also improves the schematic representation model for schematic blocks with the introduction of Mats, interactive design blocks with grouping capabilities aside from wire

[1] TypedScheamtics is available online at url-retracted

connections, enabling the automatic connection of power signals, therefore reducing interaction costs. Below is an overview of the design process with our tool and system design.

### 5.4.1 Design Process Overview

Figure 5.1 gives an overview of the design flow with our tool. With TypedSchematics, designers start by adding type information to schematic blocks created with existing schematic capture tools (e.g., Fusion 360), thus creating *typed schematic blocks*. The type information follows minimal syntax of symbols and naming conventions for declaring interfaces that are then written as signal names or as text in schematic blocks. Typed schematic blocks, as well as the schematic image of these blocks, are added to our interactive tool by experts and displayed in a sidebar for designers to use (a).

Typed schematic blocks are classified into four groups based on their capabilities: power, regulators, and computer peripherals. Designers then add typed schematic blocks to a workspace (b) where each typed schematic block is rendered as an Interactive block (c), each with different interactive capabilities depending on their classification. The rendered blocks are connected using our schematic representation model, Mats. The Mats design pattern allows grouping compute and peripheral blocks inside power and regulator blocks, avoiding having to connect ground and voltage signals for all schematic blocks, making design faster.

For each block schematic connection, our tool runs an interface checking algorithm in real-time, which validates voltages and interfaces connections, among others. In case of a connection error, a message is displayed to designers indicating the cause of the error (d). A process that makes the merging process safer and faster. After designers finish connecting the desired schematic blocks, with a simple click, our tool can generate the complete schematic in a file format usable by existing schematic design tools (e.g., Fusion 360), enabling designers to immediately continue with the PCB layout design process.

### 5.4.2 Typed Schematic Blocks

Typed schematic blocks are electronic schematics annotated with type information using our syntax style. Similar to programming languages, our tool allows circuit designers to declare types within schematic blocks using existing schematic capture tools (e.g., Fusion 360). While programming languages use data types to classify data with different attributes and data sizes (e.g., bool, int, char), our tool proposes the use of hardware interface protocols (e.g., GPIO, I2C, SPI) as well as for power signals (e.g., 3.3V, 5V) for type declarations. We call type declarations for schematic blocks as *interface types*. Interface types declaration annotations are made directly on signal nets of schematic diagrams. The following sections explain in detail the annotation process to create typed schematic blocks.

### 5.4.3 Syntax style

The syntax style for interface types declarations is divided in three categories. The first protocol interface types which are used as connection edges for the blocks. The second power interface types which describe the power capabilities of the blocks. Finally, a global attributes syntax to declare further block information, as well as added capabilities of interface types. Our syntax was based on common patterns followed by designers to annotate signal names, explained in more detail in Section 5.4.4, making it easier for designers to understand. Our syntax is also minimal, Table 5.1 shows the full syntax style for declaring interface types, below we explain each syntax category in detail.

#### Protocol Interface Types

Protocol interface types are declared with the '#' ASCII prefix followed by the protocol name (e.g., SPI, I2C, GPIO). Protocol interface types can consist of multiple signals. The Inter-Integrated Circuit (I2C) protocol is an example, this protocol is made up of two signals usually named Serial Clock Line (SCL) and Serial Data Line (SDA). The symbol '.' can be used to group signals of the same protocol (e.g., #I2C.SDA, #I2C.SCL). Our interactive Type

**Table 5.1.** Typed Schematic Blocks Annotation Syntax

| Types | Syntax |
|---|---|
| Protocol | #Protocol.Signal_Voltage |
| Power | @(VIN or VOUT)_Voltage |
| Power | GND |
| Global Attributes | #Global Attributes |
| **Optional Symbols** | **Explanation** |
| - | Alternate name |
| ! | Protocol not required |

Schematic tool displays protocol signals as one wire, avoiding the need for users to connect each protocol signal one by one.

It is common for microcontrollers or integrated circuits to have more than one same protocol, for example having multiple General Purpose Inputs/Outputs (GPIO), each with different functionalities. The ASCII '-' symbol gives the interface protocol an alternate name distinguishable from the others, allowing the enumeration or description of functionality (e.g., #GPIO-0, #GPIO0-1, #GPIO-RESET).

Finally, it is also common in schematic block designs to leave some connections as optional. The symbol '!' allows to indicate that the connection of a protocol interface type is optional (e.g., #GPIO-0!).

**Power Interface Types**

Power interface types are declared with the '@' ASCII prefix followed by two reserved words 'VIN' or 'VOUT' which indicates our tool if the schematic block receives power, outputs power, or both. If the typed schematic block outputs power, it is classified as a power block. The classification of the different blocks is covered in detail in Section 5.4.5.

Following the previous syntax, the ASCII '_' symbol is used to indicate the voltage restrictions of the power signals, which allows designers to specify fixed voltages or a range of allowed voltages (e.g., VOUT_3V, VIN_5V-9V). By knowing the input and output voltages

constraints, voltage validation is achieved.

Finally, The word 'GND', a common abbreviation for "Ground", is used as a reserved keyword. Our tool grounds all schematic blocks via signals declared as 'GND', when generating the full schematic.

**Global Attributes**

Our syntax also allows designer to include global attributes for schematic blocks such as the block classification (e.g., Power, Peripheral) as well as additional information for the interface protocols such as the address of the I2C protocol interface type or whether the SPI protocol is master or slave, allowing for more refined validations. These global attributes are defined within the schematic as text comments using the syntax '# ¡global attributes¿ '.

### 5.4.4 Syntax Implementation Motivation

The implementation of our syntax for adding type information to schematic blocks was based on the analysis of different schematic diagrams from the community of circuit designers, both amateurs and experts. In the analyzed schematics, it was observed that designers divide schematic diagrams by their functionality, separating schematic diagrams in most cases by four classifications: power, connectors, computing modules, and peripherals. We also found that most of the time schematic diagrams have named signals based on their protocol or functionality (e.g., SDA, SCL, RESET, ENABLE). Signals used for power have names that describe the signal voltage (e.g., 3.3V, 5V), and in the case of ground signals, a single signal name is used, usually GND.

Our syntax style uses the same design patterns already observed in the circuit designers community, standardizing the syntax for better sharing and reuse of schematic blocks.

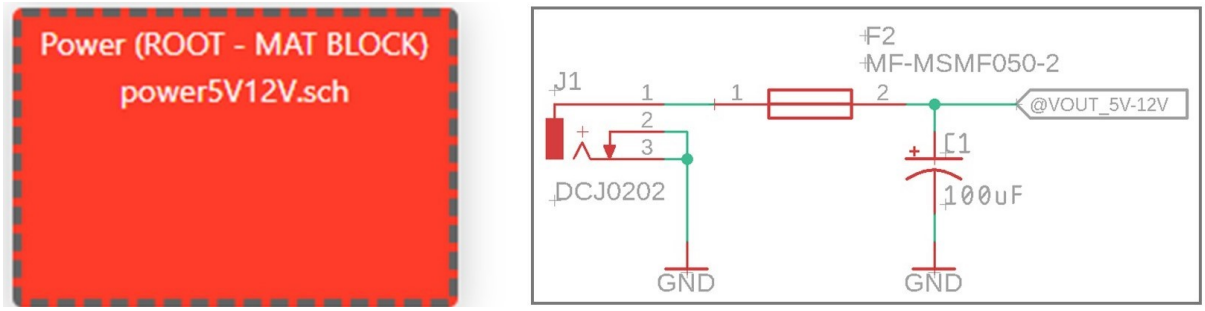



**Figure 5.2.** A power supply connector typed schematic block (right) and its interactive block, a power block (left).

### 5.4.5 Interactive block-based schematic design

Typed schematic blocks are imported directly into our block-based schematic design tool, TypedSchematics. If accepted, the imported blocks are rendered as visual Interactive blocks when added to the workspace area.

The visual rendering of each interactive block varies according to the typed schematic block classification, which can be: Power, Regulators, Compute Modules, or Peripherals. Each classification is explained below.

**Power** blocks are capable of supplying power to other blocks and are the root of the tree structure that contains the entire schematic design, described in more detail in Section 5.4.6. Power blocks are defined declaring a signal with a voltage output (VOUT), for example VOUT_5V-9V, and no voltage input (VIN). Power blocks contain connectors that supply power to the circuit (e.g., power connector, USB connector). Figure 5.2 left, shows an example of an interactive power block, on the right its corresponding typed schematic block consisting of a power connector, a resettable fuse for short circuit protection and a voltage stabilizing capacitor.

**Regulator** blocks step the voltage up or down to supply blocks with their required voltage. Regulator blocks are defined declaring both a voltage input (e.g., VIN_5V-9V) and a voltage output (e.g., VOUT_5V). Figure 5.3 left, shows an example of an interactive regulator

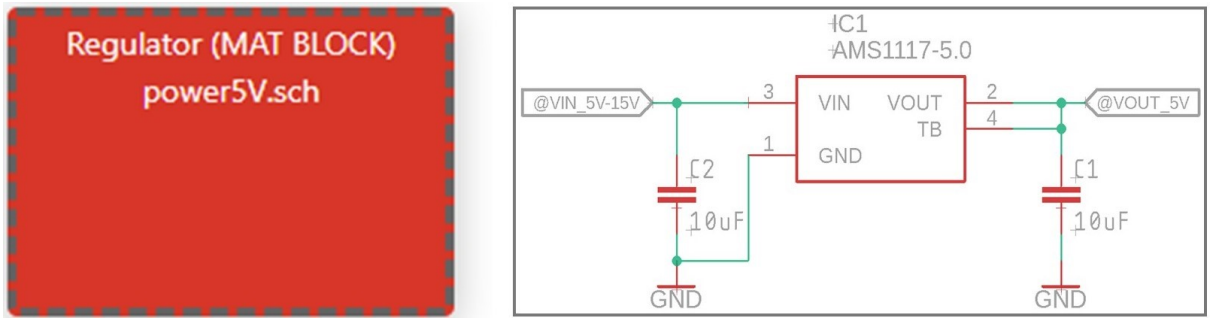



**Figure 5.3.** A 5V regulator typed schematic block (right) and its interactive block, a regulator block (left).

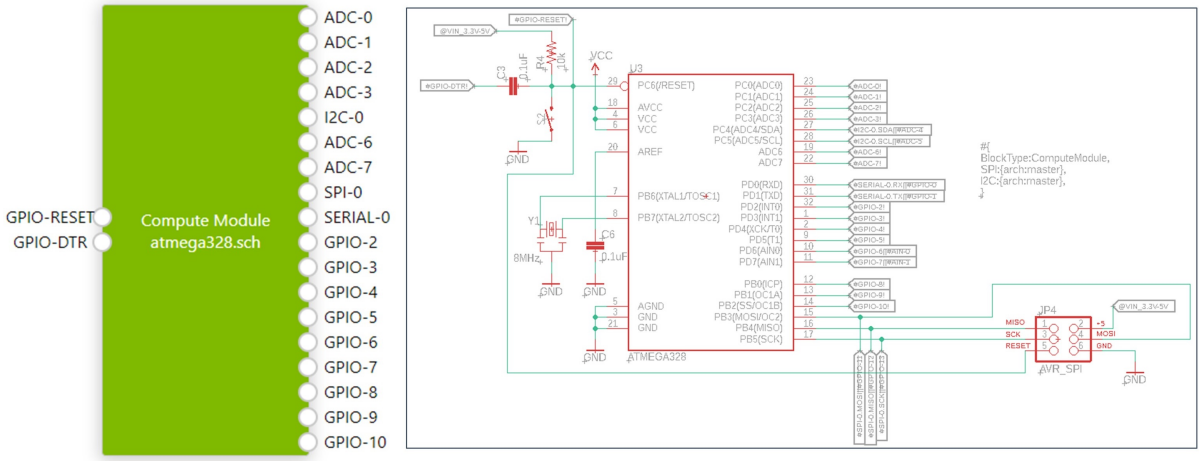



**Figure 5.4.** An Atmega328 microcontroller typed schematic block (right) and its interactive block, a compute module block (left).

block, on the right its typed schematic block consisting of a 5-volt AMS1117 IC [100] power regulator. The input voltage range was taken directly from the power IC datasheet.

**Compute module** blocks are intended for microcontrollers or processors capable of communicating with peripherals. These blocks require an input voltage and have multiple protocol interface types to address a variety of sensors and actuators. Figure 5.4 left, shows an example of a rendered compute module block, on the right its corresponding schematic consisting of an 8-bit Atmega328 [77] microcontroller and accompanying electrical components: an 8 MHz crystal, programming pin headers and a reset switch, commonly used in designs.

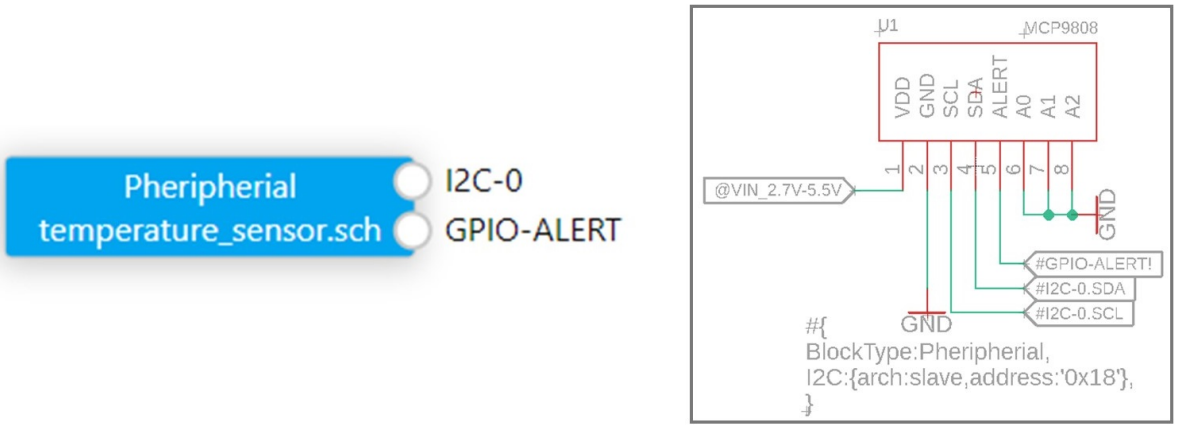



**Figure 5.5.** A temperature sensor typed schematic block (right) and its interactive block, a peripheral block (left).

**Peripheral** blocks are intended for sensors or actuators. These blocks also require an input voltage and have fewer protocol interface types compared to compute module blocks. Figure 5.5 left, shows an example of a rendered peripheral block, on the right its corresponding schematic consisting of a temperature sensor IC MCP9808 [78] with a fixed I2C address '0×18' declared as an I2C extra property using the global attributes comment block.

### 5.4.6 Mats Schematic Model Representation

On-screen interactive blocks are divided into two types: *mat blocks* and *general blocks*, each with different interaction capabilities and connection structures. General blocks behave like graph nodes, allowing for edge connections and positioning along the workspace. Mat blocks inherit the capabilities of general blocks and also allow any block to sit on them, hence the name Mat. Blocks classified as power and regulators are mat block types, while blocks classified as computing and peripheral are general block types.

Compared to the schematic model representation of conventional schematic design tools, that use a general graph structure to interactively connect electrical symbols. TypedSchematics allows the structure to be divided into two, a tree graph structure for mat blocks and a graph for

other blocks. We call this new divided structure *Mats*.

The motivation for the implementation of Mats was given after observing the repetitive task of connecting power signals between designers using existing schematic design tools. Mats eliminate the need for such repetitive power signal connections. Each of the structures in the Mats schematic representation model and their capabilities are further described below.

**Tree Graph and Mat Blocks**

The tree structure for mat blocks allows for the elimination of manual power connections required by all typed schematic blocks that would otherwise have to be made if represented by a single graph structure. This simplifies the visual diagram of typed schematic blocks by reducing clutter, and reducing the cost of interactions by freeing the designers from having to specify the input voltages for each block.

Only blocks classified as power and regulator appear as mat block types. Mat blocks are resizable and allow other blocks to sit on them. When any type of blocks is dragged and dropped on top of the mat blocks, the output voltage of the mat block is connected to the input voltage of the dropped block. The power mats form a tree with the global power supply mat as its root. As an example, Figure 5.6 (b) shows a power block containing two regulator blocks, one 5 volts and one 3.3 volts, (a) shows the tree graph structure representation for these three mat blocks.

**Connecting General Components**

Non-power supply components (i.e., *general blocks*) can be linked together in a general graph structure. This structure allows connections between edges of blocks, as normal schematic design tools. General blocks can also be added to mat blocks. The graph structure is exemplified in Figure 5.7, where (b) shows a compute module block connected to three peripheral blocks; two LEDs, and a temperature sensor, while (a) shows the representation of the internal graph structure.

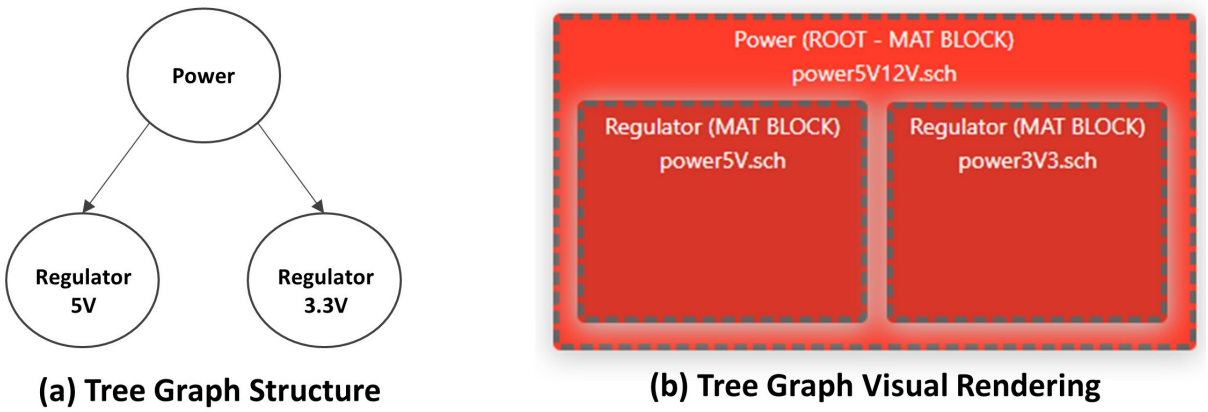



**Figure 5.6.** Visual rendering of two mat blocks, consisting of 5V and 3.3V regulator blocks, connected to a root mat block consisting of a power block (b) and its tree graph structure model representation (a).

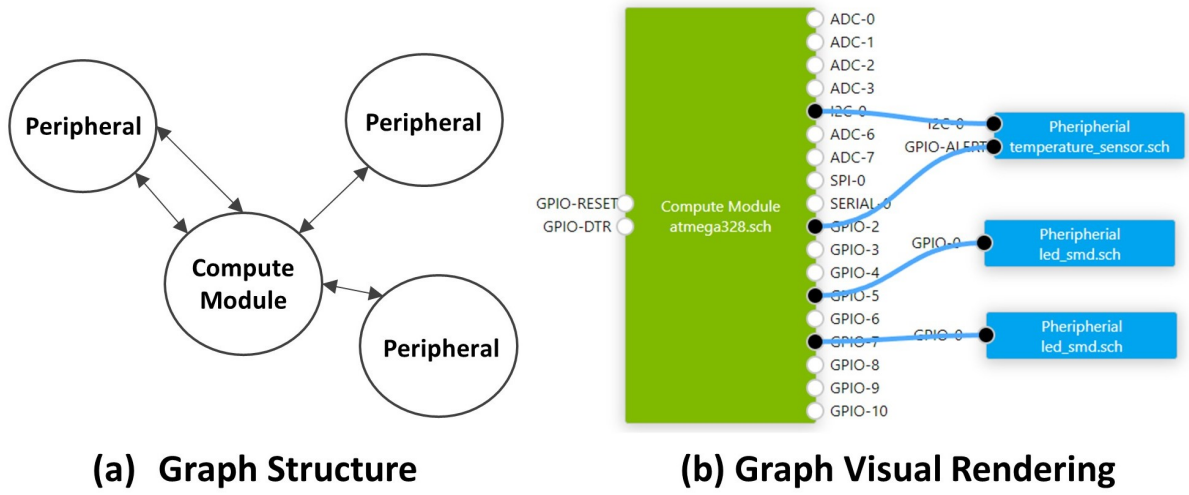



**Figure 5.7.** Visual rendering of multiple general blocks, consisting of a compute module and three peripheral blocks, connected together (b) and their general graph structure model representation (a).

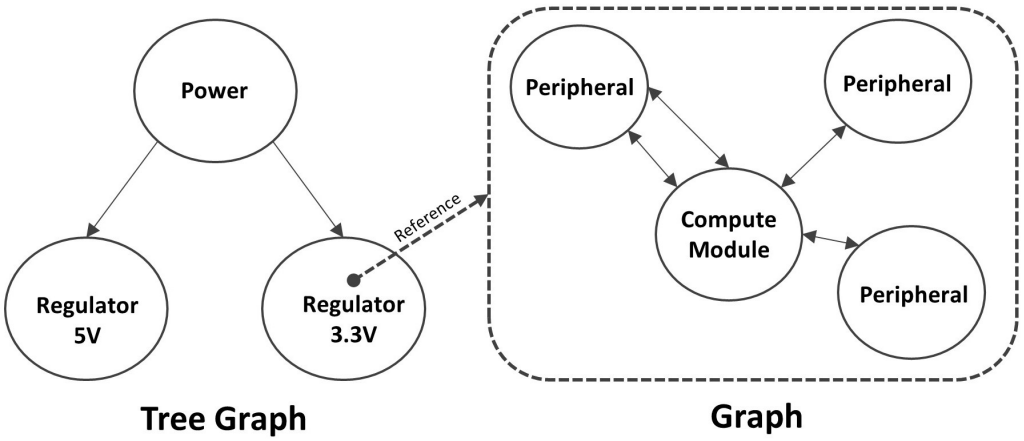



**Figure 5.8.** A tree graph structure (left) connected to a general graph structure (right) by reference, forming the Mats model representation for schematic blocks.

#### Combining Mats and General Blocks

The tree and graph structures are joined together when general blocks are dropped into mat blocks. The combination of these allow the designer to deal with the power-related aspects of the design separately from the signal connection aspects. Figure 5.8 shows an example of combining the power mats in Figure 5.6 with the general components in Figure 5.7.

### 5.4.7 Real-time Interface Checking

When making connections between schematic blocks, either by connecting wires between general blocks or by dragging and dropping blocks onto mat blocks, TypedSchematic checks if the connections are valid through interface checking. Interface checking runs in real time. To perform interface checking in real time, a flow-based programming paradigm is seamlessly integrated into the tree and general graph structures described in Section 5.4.6 and Section 5.4.6. When designers make connections, interface type information is acquired using a message passing algorithm and validations are performed. The validation routines are stored as code files allowing interface types to scale.

Currently, our tool performs four types of interface checkings described below, an error is displayed to the designer when the constraints are not met.

### Protocol Interface Matching

A protocol matching interface checking is triggered when connecting wires between general blocks. This validation verifies that protocol interfaces are of the same type, an error is displayed in case of protocol types mismatch.

### Voltage Interface Matching

Voltage matching interface checking is triggered when blocks are dragged and dropped onto mat blocks. This validation verifies that the voltage interface type of the dropped block element and that of the mat blocks fits together. For example, if the designer wants to add a compute module block that requires 5V to a mat block with an output voltage of 3V, the action is not allowed.

### Required Interface

With the use of the '!' symbol, described in Section 5.4.3, designers can indicate whether protocol interface types are required or not. This interface checking verifies, when attempting to generate the full schematic, that no required interface connections are missing.

### Additional Interface Checkings

With the use of global attributes, described in Section 5.4.3, additional capabilities can be added to interface types. Interface checking for additional interface type capabilities can be extended by designers. For example, the I2C protocol interface type includes address validations in which if there are duplicate addresses (i.e., two sensors with the same I2C address) the connection is invalid. Another example is the SPI protocol interface type, which includes validations that require a master-slave connection, rendering master-master or slave-slave connections invalid.

In addition to these interface checkings, design errors are also verified, such as trying to generate a schematic with no schematic blocks.

### 5.4.8 Automated Composition of Schematic Blocks into a Single Schematic

If there are no design errors, TypedSchematics automatically merges all typed schematic blocks into a single schematic file. In this step, the tree and general graph structures are converted into a single graph and the signals of each interface type are connected according to the design.

The final schematic design is compatible with existing schematic design tools, allowing designers to quickly move To the PCB layout design process.

## 5.5 Design Examples

We demonstrate TypedSchematics Mats schematic model representation capabilities by building two schematic designs, a thermostat and a temperature logger.

### 5.5.1 Thermostat

A thermostat is a common circuit design used in many commercial electronic devices, including refrigerators, air conditioners, and ovens. Thermostats regulate temperature by activating actuators (e.g., cooling compressors) when the measured temperature has reached a user-entered set point.

Figure Figure 5.9 shows a minimal design of a thermostat designed with schematic blocks using TypedSchematics (a) and Fusion 360 (b) to provide a better visual comparison. Both tools include the same schematic blocks. As peripherals the design includes: a MCP9808 [78] temperature sensor, two buttons to set the temperature, an LED indicator that simulates actuators activation, and a display consisting of four 7-segment displays and drived by a HTK16K33 IC [54]. For computation, an Atmega328 [77] with a 8 MHz crystal. A DC jack supplies power to the entire design, while a 5V regulator provides the correct voltage to all schematic blocks.

At the level of hardware protocol interfaces, both the display and the temperature sensor require an I2C connection. The microcontroller only has one I2C port available, however both I2C interfaces can be connected to the same port using daisy-chaining, an I2C capability. To

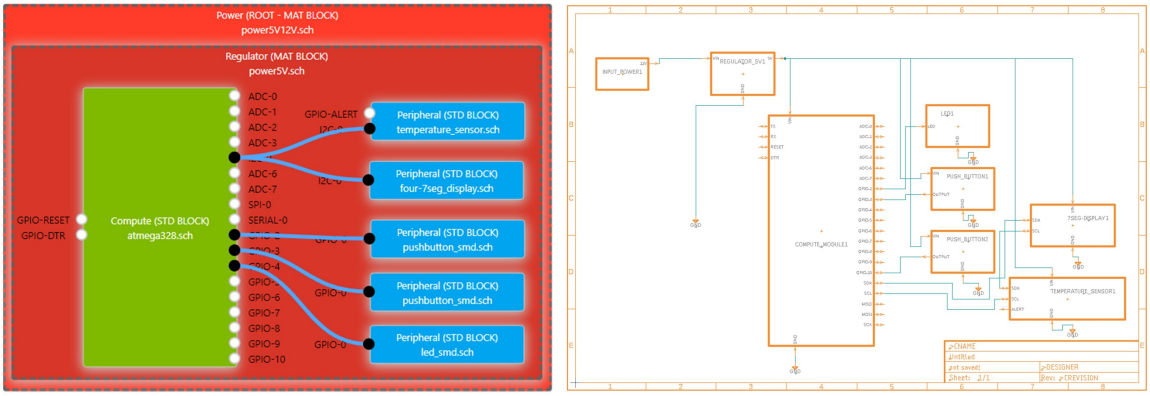



**Figure 5.9.** Schematic blocks design of a thermostat schematic using TypedSchematics (a) and Fusion 360 (b). Compared to Fusion 360, TypedSchematics eliminates the need for multiple voltage and ground connections with the use of Mat blocks, reducing interaction costs and wire clutter. TypedSchematics also combines all interface signals into one, further reducing wires and eliminating the need to know how to connect each individual interface signal. When the design is ready, TypedScheamtics warns designers if connections are missing. Fusion 360 (b) design was implemented by P02.

achieve I2C daisy-chaining both the display and the temperature sensor must have different I2C addresses. The temperature sensor also has an interrupt signal to tell the microcontroller when I2C data is available. This signal interface is of type GPIO and its alternative name is ALERT, as indicated in the data sheet. Finally the LED and the two buttons are GPIO type interfaces.

Compared to Fusion 360, TypedSchematics eliminates the need for multiple voltage and ground connections with the use of Mat blocks, shown in (a) as the red blocks. In Fusion 360, users must also connect each interface wire individually, for example SDA and SCL to make an I2C connection. In contrast, TypedSchematics combines all interface signals into a single one, further reducing wires and eliminating the need to know the wires should be connected for each interface.

Fusion 360 also makes it hard to identify the required interface for some signals, for example OUTPUT, LED, and ALERT in the design, since there is no standard for naming signals. The TypesSchematics annotation syntax ensures that all signals are defined with an interface and an optional alternative name (e.g., GPIO-ALERT).

Finally, in the thermostat design, only the temperature sensor ALERT interrupt signal is optional. In Fusion 360 it is impossible to know if the other signals should be connected or not without looking at each component datasheet. On the contrary, when using TypedSchematics our tool generates an error in case of any missing connection.

### 5.5.2 Temperature Logger

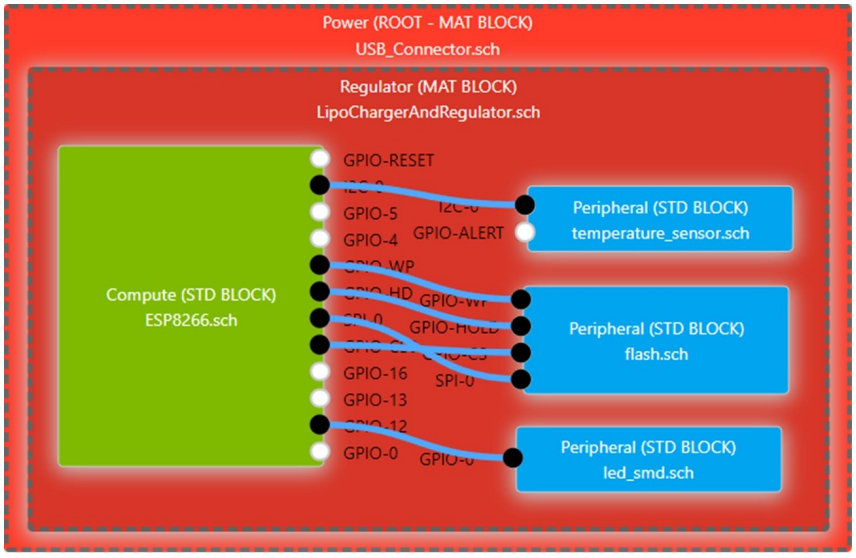
In the following design example, we present a temperature logger with Wi-Fi capabilities and a rechargeable battery, common components of IoT devices.

Figure 5.10 (a) shows the temperature logger schematic blocks design. This design reuses the LED and temperature sensor peripheral blocks used in the thermostat design and adds a 8 MB flash IC [79] to log temperature data. The compute block is based on the ESP8266 microcontroller [40] with built-in Wi-Fi, used in many IoT designs. Power is supplied via a USB connector, which feeds a LiPO battery charger circuitry with included regulator. While the design structure is similar to the thermostat described in Figure 5.9. The capabilities of the circuit are completely different, demonstrating the simplicity of the classification chosen for the schematic blocks described in Section 5.4.5 and how it fits into multiple designs.

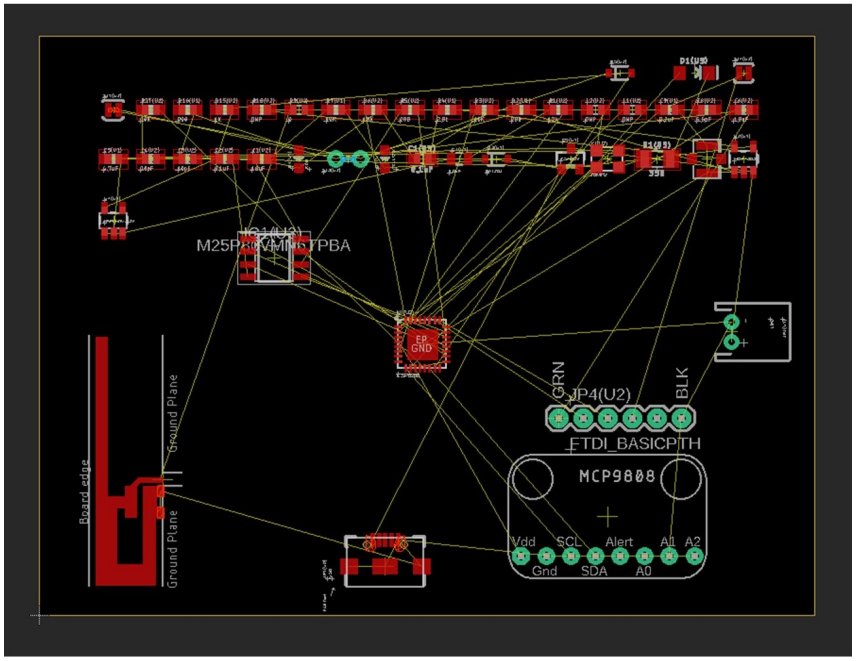
TypedSchematics automatically merges the different blocks into a single schematic, helping designers jump right into PCB layout design. Figure 5.10 (b) shows the starting PCB layout that is generated from the temperature logger design. Each protocol wires are correctly connected between electronic components. Power signals are properly distributed and connected to all corresponding electronic components.

## 5.6 User Study

To demonstrate how TypedSchematics supports designers in the schematic blocks composition process with the addition of safety futures and a new schematic model representation approach, we conducted a small user study. In the user study, participants were given the task of designing a schematic using both TypedSchematics and Fusion 360, a popular schematic design




(a)




(b)

**Figure 5.10.** Schematic block design of a temperature logger with Wi-Fi capabilities and a rechargeable battery. TypedSchematics helps designers jump right into PCB layout design with the automated composition of schematic blocks.

tool.

Overall, our user study focused on the process of connecting schematic blocks together. The realism of the schematic designs and the merging process were emphasized, choosing as a design a thermostat described in Figure 5.9. For Fusion 360, the hierarchical sheet design reuse approach was used for connecting schematic blocks. Below, we describe the study. Section 5.7 describes our findings.

Although our user study has a quantitative section, for the most part it centers on the qualitative aspect, taking in consideration the feedback and opinions of participants.

### 5.6.1 Participants

We recruited two participants (P01 and P02) through email advertising at our university, both participants graduate students with intermediate Fusion 360 schematic design knowledge. Participants were rewarded with a $40 Amazon gift card for a two-hour user study.

### 5.6.2 Structure

The user study consisted of three phases. First, a video training to teach participants how to use TypedSchematics and reinforce their knowledge of block schematic manipulation using Fusion 360. Second, a zoom interview where the participants developed a temperature controller, described in Figure 5.9, in both TypedSchematics and Fusion 360. Finally, a questionnaire where the participants were asked questions comparing the usability of both tools on multiple aspects, as well as their feedback. We discuss more about each phase below.

To prepare participants in the use of TypeSchematics, a video [2] was recorded to teach the different classifications of the interactive blocks, as well as the differences and capabilities between standard and mat blocks. In this stage, they designed a circuit to make an LED blink, the ”hello word” of hardware design. Likewise, they were taught the new schematic model representation, Mats. Knowledge of how to connect block schematics using Fusion 360 was also

[2] TypedSchematics video tutorial - url-retracted

reinforced through a video tutorial provided by Fusion 360's developers, as the use of schematic blocks is not something that all designers implement in their designs.

In the 1-hour zoom meeting, participants were guided through the process of creating a blinking LED circuit using Fusion 360 and TypedSchematics, having now the opportunity to ask any questions. Subsequently, they were asked to design a thermostat and were provided with all the necessary schematic blocks for the design, in both Fusion 360 and TypedSchematics.

In the case of Fusion 360, participants were emailed a single schematic file containing all required schematic blocks. For TypedSchematics, participants were sent a web link, different from the one used for the tutorial [3], containing more schematic blocks.

In Fusion 360, the Hierarchical Sheets approach described in Section 5.2.2 was used as it is the only method that encapsulates schematics into reusable blocks in Fusion 360.

The participant's computer screen was recorded throughout the temperature controller design process, as well as their voices. After finishing both designs, they were asked for their opinion on their design experience using both tools, ending the zoom meeting.

The questionnaire consisted of 15 questions divided into three sections: schematic blocks merging process, Mats schematic model representation, and general questions. The first two sections of questions centered on understanding the participants' usability preferences regarding TypedSchematics compared to Fusion 360. The general questions, centered on understanding any possible barriers to participants in adopting TypedSchematics for future designs. For each section, participants also had the option to leave feedback.

One of the concerns with the Mats schematic model representation was that users might not fully understand the new design pattern and how it works internally. That is why the questionnaire also included four questions to evaluate the participants' understanding of Mats.

[3] TypedSchematics user study - url-retracted

## 5.7 Results

Participants were able to complete the thermostat design using TypedSchemtics and Fusion 360. However, the results show significant differences in design experience and design times. Below, we show these differences for each of the TypedSchematics contributions: the Mats schematic model representation and the real-time detection of connection errors, as well as discuss the overall experience of participants.

### 5.7.1 Mats schematic model representation

To assess how Mats reduces interaction costs versus existing schematic model representations, the thermostat design time was compared in both tools. To make these measurements, the recordings were used where the start and end times of each design were obtained. The results, shown in Figure 5.11, show that, on average, the design time of the participants was 2.8 times faster with TypedSchematics than with Fusion 360. Taking the participants on average 16 minutes to merge the schematics blocks in Fusion 360 and 5.6 minutes in TypedSchematics.

Although the results show better design performance with TypedSchematics, a big concern for us was the usability and ease of learning from participants, since the schematic model representation approach is different from the traditional one.

To assess the ease of learning, we asked the questions focused on Mats, which is new compared to the traditional approach. We ask questions concerning the different types of mat blocks (power and regulator blocks), and what connections they connect internally (voltage and ground), among other things. Both participants completed the quiz without any errors.

In terms of usability, both participants preferred the TypedSchematics design style for merging schematics blocks compared to Fusion 360. Overall, participants felt that using mat blocks helps reduce the visual information overload that occurs with the accumulation of power signals, making the design look cleaner and less cluttered.

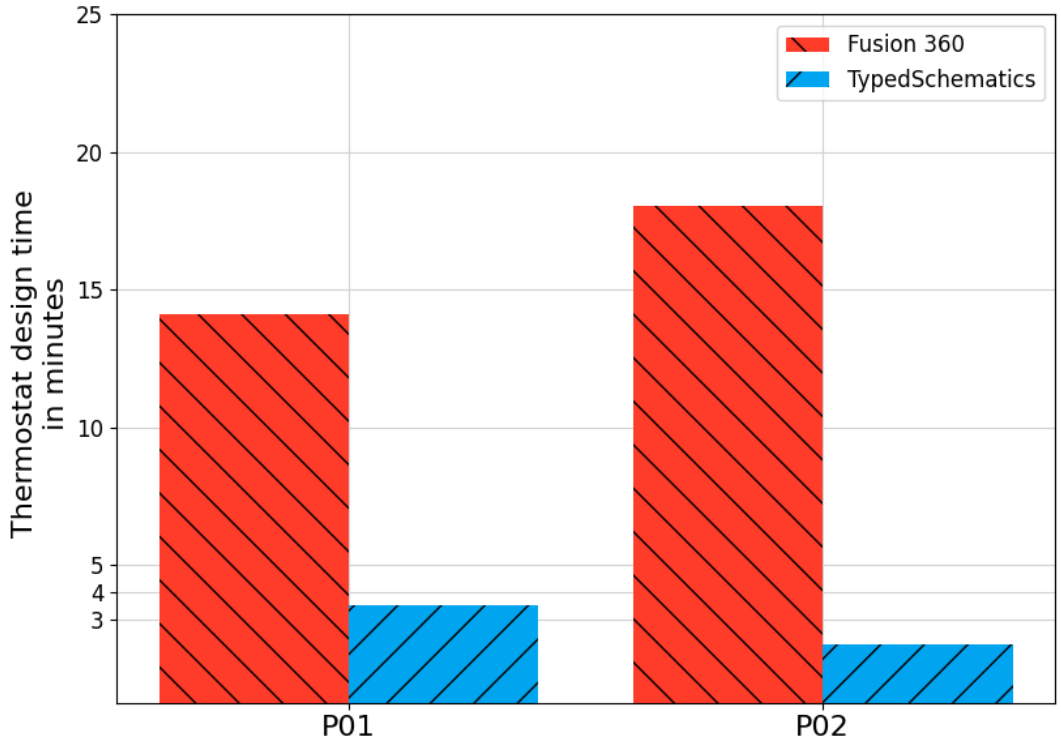



**Figure 5.11.** Design time of a thermostat schematic with schematic blocks, described in Figure 5.9, using Fusion 360 and TypedSchemtics. On average, participants' design time was 2.8 times faster with TypedSchematics than with Fusion 360.

### 5.7.2 Real-time Detection of Common Connection Errors

Real-time detection of common connection errors was assessed through confidence weighted multiple-choice questions regarding connections, to better understand the participants confidence levels when making connections. These questions were divided into protocol and power interface type connections. The results showed that participants felt completely confident making protocol and power interface type connections using TypedSchematics. Whereas with Fusion 360, participants felt somewhat or fairly confident when making the same connections. In general, participants feel more comfortable connecting schematics blocks using TypedSchematics real-time interface checking.

### 5.7.3 Participants Experience and Observations

Through the user study, we made several observations that further show the design differences with TypedSchematics compared to Fusion 360. In this section, we discuss these observations and also quote some of the user experiences that were gathered through the recordings and the questionnaire.

#### Deciding which Power Regulator Block to Use

In the user study it was observed that in Fusion 360 the participants had trouble deciding which voltage regulator to use, whether 3.3V or 5V, to power the underlying blocks. The reason for this indecision is that it is necessary to know the supply voltage required for each of the schematic blocks in order to know which voltage regulator to use. Since there is no information in Fusion 360 indicating the required voltage or any type of checking, participants spent a great time analyzing each schematic block's internal diagrams to figure out which voltage regulator to use.

The correct regulator to use for the thermostat is 5V as only the display has this restriction while all other schematic blocks support a voltage range of 3.3V to 5V. However, in the user study, P01 started with a 3.3V voltage regulator and eventually, realizing that the display required

5V, decided to add an extra 5V voltage regulator just for the display. Although the atmega328 microcontroller accepts an input and output voltage of 3.3V and 5V, making this design valid, in stricter designs or with the use of another microcontroller, this design is incorrect, and could cause serious damage to the entire circuit. This is due to the difference in communication wire voltages between the microcontroller and the display. P02 also used a 3.3V regulator early on and realized that a 5V regulator was required near the end of the design, taking more design time to make the change. However, P02 did not add any extra voltage regulators. In TypedSchematics, participants did not encounter any of these problems with the assistance of the real-time interface checking, which guided participants through the design process by displaying useful error information when they made connection errors.

**Unsure of How to Connect Hardware Protocol Interfaces**

When connecting the I2C hardware protocol interface, both participants made comments indicating they were not sure how to connect the I2C interface correctly. In particular, both participants were unsure how to connect multiple I2C interface blocks, forgetting that the I2C can be daisy-chained, and asking in Fusion 360 if all the SDA wires shown in Figure 5.9 (b) should be connected together or not, same for the SCL wires. For this case, both participants received a brief guide on how to connect these wires; otherwise, the user study would have exceeded the allotted time. In TypedSchematics, since protocol interface types are represented by a single ”wire” on the screen and automatically connected at generation time, the designers were completely confident and did not ask any questions.

For both designs, neither participant asked about the I2C address conflicts that can arise with I2C daisy chaining. The likely cause is that, in the case of Fusion 360, which has no method to detect these conflicts, such a conflict would have led to both participants having a design error.

**Unsure of Optional Connections**

The MCP9808 temperature sensor I2C interrupt signal is optional for the thermostat design, shown in Figure 5.9 (b) as ALERT. P02, unsure of the interface and capabilities of this signal in Fusion 360, searched online for known datasheets and e-blogs for more information. After spending 1.5 minutes researching the capabilities of this signal, the participant discovered that the connection interface type was of GPIO and that was optional, leaving the signal unconnected. P01 treated the ALERT signal as required and made a wire connection. In the case of TypedSchematic, none of the participants had a problem determining if some signals are optional, as our tool alerts the designer in case of missing connections.

**Participants overall experience**

When comparing Fusion 360 to Type Schematics, both participants found the design experience with TypedSchematics easier than with Fusion 360. P02 went further and described the Fusion 360 experience as follows:

> ”Is not fun. Schematic design, which is a high level design compared to PCB layout, should be the easy part”.

P02 also described the design experience with TypedSchematics as follows:

> ”Is not work at all, is very simple to use”

Finally, from the answers collected from the questionnaire, the only limitations of participants to adopt TypedSchematics in new designs is the lack of a large library of schematic blocks and inclusion of more interface types (e.g., USB, HDMI). Which is to be expected given that our tool is experimental and intended for research purposes only.

Additionally, participants also find this tool a good fit as a design template generator of schematics for quickly getting started with new designs or for exploring new design ideas.

## 5.8 Acknowledgement

Chapter 5, in part, is currently being prepared for submission for publication of the material. Garza, Jorge; Swanson, Steven. The dissertation author was the primary investigator and author of this chapter.

# Chapter 6

# Conclusion

We have explored three different encapsulation techniques to fuse different steps of the electronic device design process through the presented design tools: Amalgam, Appliancizer and TypedSchematics.

From the results of these projects, we can show that it is possible to greatly accelerate the development and exploration of electronic device designs by encapsulating circuit information and automating steps. At the same time, we also show that the abstraction of circuit information improves and reduces design complexity. Concluding remarks on each specific project are presented below.

## 6.1 Amalgam

Amalgam deeply encapsulates low-level hardware code into web programming technologies, fusing the low-level code design and Application logic and GUI smart device design steps described in Section 2. This enables rapid development and more flexible design iteration for smart embedded devices.

Amalgam's framework lets developers replace soft interface components with hardware components just by changing a CSS file. A compiler was designed to swap selected HTML elements with web components integrated with low-level code at web page's run time, granting Amalgam the ability to switch between pure soft and soft plus hard software code versions,

avoiding this way invasive changes to the application code. We implemented Amalgam and evaluated its capabilities by prototyping three network-connected electronic devices in a web browser and then "hardening" them into stand-alone devices.

Our results show that Amalgam vastly reduces the effort required to integrate low-level code required to communicate hardware components into the application code. Comparing the same code for a video player smart device, only five lines of CSS code were required with Amalgam compared to over eighty lines of code with the traditional approach.

## 6.2 Appliancizer

Appliancizer builds on top of Amalgam and further fuses the schematic and PCB layout design steps for smart devices, allowing designers to rapidly generate functional smart devices from web-based prototypes, facilitating the prototyping of mixed graphical-tangible interactions. Appliancizer encapsulates multiple reusable circuit schematic pieces with their required low-level code, allowing for automated generation of circuit netlists. Further encapsulation of virtual 2D components that are drag-and-drop into a resizable virtual PCB makes it possible to know the physical position of hardware components, enabling automated PCB generation.

An encapsulation technique we call essential interface mapping maps the similarities between HTML controls and tangible controls which enables the direct transformation of HTML elements from graphical to tangible and enables the almost complete automation of the development process for smart devices described in Section 2. Essential interface mapping also allows the simulation of smart devices physical interfaces using HTML elements as mockup hardware components.

Appliancizer makes graphical and tangible interaction prototyping as easy as modifying a web page and allowing designers to leverage the well-developed ecosystem of web technologies. This accelerates development and reduces complexity for designers. We have demonstrated the capabilities of our tool and novel techniques by transforming both a video player and a smart

thermostat web application into physical devices without requiring any changes to the application source code. We have also demonstrated the rapid development capabilities of our tool through a preliminary user study which show a speed-up of 6.5X in development time compared to Amalgam.

## 6.3 TypedSchematics

TypedSchematics introduces a circuit data encapsulation technique for electronic schematics that enables the creation of an interactive design tool with a low threshold but a high ceiling, a balance that allows for fast and easy yet highly configurable schematic designs. This encapsulation technique centers on annotating schematics with hardware interface types (e.g., I2C, SPI), giving schematic blocks knowledge of required connections rather than having wires with unknown connection types, and allowing the manual connection of inputs of schematic blocks in a faster and easier way.

Compared to block-based design tools for rapid design, such as Sparkfun ALC and Upverter Modular, which rely on existing schematic design tools (e.g., Fusion 360) for composing schematic blocks. These tools present multiple design challenges including: correctly connecting hardware protocol interfaces, supplying the correct voltages to the schematic blocks, detecting if some signals on the schematic blocks are optional.

TypedSchematics addresses these design challenges by separating power-related connections from connections between general components. It also provides real-time, interactive feedback on potential design problems including mis-matched voltages, mis-connected signals in complex interfaces, and missing (but necessary) connections.

TypedSchematics's mat abstraction eliminates the need for repetitive voltage and ground connections, reducing interaction costs. A flow-based programming paradigm is seamlessly integrated into Mats, making it possible to perform interface checking in real time.

In addition, TypedSchematics improves the scalability of the blocks' library by incor-

porating the annotation syntax into current PCB design tools. Overall, TypedSchematics lets designers rapidly create and merge different schematic blocks into a single schematic, as well as provides detection of design errors in real-time.

We demonstrate the capabilities of TypedSchematics with two schematic design examples, a thermostat and a temperature logger. We further demonstrate the design challenges of existing schematic design tools and how TypedSchematics addresses these challenges, as well as reduces interaction costs, through a user study. Results show that, on average, participants' design time of a thermostat schematic was 2.8 times faster with TypedSchematics than with Fusion 360. Feedback from participants via a questionnaire also showed that participants felt completely confident in the correctness of the design with TypedSchematics compared to Fusion 360, where they felt somewhat confident.

## 6.4 Future Work

### 6.4.1 Increasing design abstraction through AI

Recent research has shown the possibility of generating HTML interfaces from text descriptions [85]. On the other hand, Appliancizer allows designers to quickly generate electronic devices with mixed graphical and physical interfaces from HTML interfaces. Therefore, we see possible the integration of these two tools for the creation of a creativity tool that allows an even faster design flow for the creation of smart devices.

### 6.4.2 PCB-ready breakout boards

Through our work, the electronic prototyping design phase was set aside, without fusing this stage with any tool. However, we see the possibility of integrating this step into Typed-Schematics. Since breakout boards are already built with a modular design in mind, we see the possibility of integrating the design of schematics blocks into TypedSchematics. An idea that would enable the rapid transition and generation of PCB designs from prototypes, bridging the

gap between prototyping and production. This idea is under proposal and has been presented as a position paper at CHI 2023 ”Beyond Prototyping Boards: Future Paradigms for Electronics Toolkits” [42].

## 6.5 Final Conclusion

The fusion of different electronic design steps, design steps fusion, is achievable through circuit information encapsulation which accelerates the design of electronic devices through reuse and automation. Circuit information encapsulation methods and techniques affect how different steps of electronic design can be connected together. Throughout this dissertation, we have explored different encapsulation methods and how they enable and influence new design flows. Through user studies we show that, in addition to speeding up design, fusing and encapsulation reduce design complexity. Ultimately, this translates into electronic design tools that allow experts to work more rapidly and efficiently through design reuse and automation, and makes electronics design more accessible to beginners with the reduction in design complexity.

# Bibliography


[1] AngularJS - Superheroic JavaScript MVW Framework. https://angularjs.org/.

[2] Electron — Build cross platform desktop apps with JavaScript, HTML, and CSS. https://electronjs.org/.

[3] Generic Sensor API. https://www.w3.org/TR/generic-sensor.

[4] HTML Media Capture. https://www.w3.org/TR/html-media-capture/.

[5] Johnny-Five: The JavaScript Robotics & IoT Platform. http://johnny-five.io/.

[6] Linuxduino - A javscript library for communicating with hardware in a Arduino style programming for any Linux platform. Available online at: http://www.w3.org/TR/html5.

[7] Node.js. https://nodejs.org/en/.

[8] Qt - Cross-platform software development for embedded & desktop. https://www.qt.io/.

[9] Rythm.js - GitHub Pages. https://okazari.github.io/Rythm.js/.

[10] Web Bluetooth Community Group. www.w3.org/community/web-bluetooth.

[11] WebAssembly. https://webassembly.org/.

[12] World Wide Web Consortium, HTML5 Specification. Available online at: http://www.w3.org/TR/html5.

[13] Ieee standard american national standard canadian standard graphic symbols for electrical and electronics diagrams (including reference designation letters). *IEEE Std 315-1975 (Reaffirmed 1993)*, pages i–244, 1993.

[14] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *Mobile Data Management, 2007 International Conference on*, pages 198–205. IEEE, 2007.

[15] Altium. Altium designer. (2022), 2022.

[16] Altium. Designing for reuse in altium designer (2018), 2022.

[17] Altium. Multi-sheet & hierarchical designs in altium designer (2020), 2022.

[18] Altium. Upverter modular (2022), 2022.

[19] Fraser Anderson, Tovi Grossman, and George Fitzmaurice. Trigger-action-circuits: Leveraging generative design to enable novices to design and build circuitry. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 331–342. ACM, 2017.

[20] Arduino. Arduino reference, 2020.

[21] Jonathan Bachrach, David Biancolin, Austin Buchan, Duncan W Haldane, and Richard Lin. Jitpcb. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2230–2236. IEEE, 2016.

[22] Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. istuff: a physical

user interface toolkit for ubiquitous computing environments. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 537–544, 2003.

[23] Hernando Barragán. Wiring: Prototyping physical interaction design. *Interaction Design Institute, Ivrea, Italy*, 2004.

[24] Ayah Bdeir. Electronics as material: Littlebits. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*, TEI '09, pages 397–400, New York, NY, USA, 2009. ACM.

[25] CodePen. Build, test, and discover front-end code - nest thermostat control, 2020.

[26] Gerald Coley. Beaglebone black system reference manual. *Texas Instruments, Dallas*, 2013.

[27] Michael Compton, Payam Barnaghi, Luis Bermudez, RaúL GarcíA-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, et al. The ssn ontology of the w3c semantic sensor network incubator group. *Web semantics: science, services and agents on the World Wide Web*, 17:25–32, 2012.

[28] Adrien Coyette and Jean Vanderdonckt. Prototyping digital, physical, and mixed user interfaces by sketching. In *Workshop on User Interface eXtensible Markup Language UsiXML, France, Paris*, 2010.

[29] Kris De Volder. Jquery: A generic code browser with a declarative configuration language. In *International Symposium on Practical Aspects of Declarative Languages*, pages 88–102. Springer, 2006.

[30] Flavia C Delicato, Paulo F Pires, Thais Batista, Everton Cavalcante, Bruno Costa, and Thomaz Barros. Towards an iot ecosystem. In *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems*, pages 25–28, 2013.

[31] Carl DiSalvo, Jonathan Lukens, Thomas Lodato, Tom Jenkins, and Tanyoung Kim. Making public things: how hci design can express matters of concern. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2397–2406, 2014.

[32] Daniel Drew, Julie L Newcomb, William McGrath, Filip Maksimovic, David Mellis, and Björn Hartmann. The toastboard: Ubiquitous instrumentation and automated checking of breadboarded circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 677–686, 2016.

[33] Lars Dürkop, Jahanzaib Imtiaz, Henning Trsek, and Jürgen Jasperneite. Service-oriented architecture for the autoconfiguration of real-time ethernet systems. In *3rd Annual Colloquium Communication in Automation (KommA)*, 2012.

[34] EAGLE. Pcb design software - autodesk, 2020.

[35] Autodesk Eagle. Modular design blocks, 2022.

[36] ECMA Ecma. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,*, 1999.

[37] EETimes. 2017 Embedded Markets Study: Integrating IoT and Advanced Technology Designs, Application Development Processing Enviroments, Apr. 2017. https://m.eet.com/media/1246048/2017-embedded-market-study.pdf.

[38] SparkFun Electronics. Sparkfun À la carte. (2022), 2022.

[39] SparkFun Electronics. Sparkfun À la carte faq. (2022), 2022.

[40] ESPRESSIF. A cost-effective and highly integrated wi-fi mcu for iot applications, 2022.

[41] Micro:bit Educational Foundation. Meet the new bbc micro:bit, 2023.
[42] J Garza and Steven Swanson. Pcb-ready breakout boards: Bridging the gap between electronics prototyping and production. *arXiv preprint arXiv:2303.06620*, 2023.
[43] Jorge Garza, Devon J Merrill, and Steven Swanson. Amalgam: Hardware hacking for web developers with style (sheets). In *International Conference on Web Engineering*, pages 315–330. Springer, 2019.
[44] Jorge Garza, Devon J Merrill, and Steven Swanson. Appliancizer: Transforming web pages into electronic devices. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2021.
[45] Julien Gascon-Samson, Kumseok Jung, Shivanshu Goyal, Armin Rezaiean-Asel, and Karthik Pattabiraman. Thingsmigrate: Platform-independent migration of stateful javascript iot applications. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
[46] Evgeny Gavrin, Sung-Jae Lee, Ruben Ayrapetyan, and Andrey Shitov. Ultra lightweight javascript engine for internet of things. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 19–20, New York, NY, USA, 2015. ACM.
[47] Google. Blockly: A visual programming editor, 2022.
[48] Saul Greenberg and Chester Fitchett. Phidgets: Easy development of physical interfaces through physical widgets. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 209–218, New York, NY, USA, 2001. ACM.
[49] Grove. Grove modules - seeed studio, 2020.
[50] Gaoyang Guan, Wei Dong, Yi Gao, Kaibo Fu, and Zhihao Cheng. Tinylink: A holistic system for rapid development of iot applications. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 383–395. ACM, 2017.
[51] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, volume 15, 2009.
[52] Bjoern Hartmann, Scott R Klemmer, Michael Bernstein, and Nirav Mehta. d. tools: Visually prototyping physical uis through statecharts. In *in Extended Abstracts of UIST 2005*. Citeseer, 2005.
[53] Steve Hodges. Democratizing the production of interactive hardware. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 5–6, 2020.
[54] HOLTEK. Ram mapping 16*8 led controller driver with keyscan - ht16k33, 2022.
[55] Utkarshani Jaimini and Mayur Dhaniwala. Javascript empowered internet of things. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 2373–2377. IEEE, 2016.
[56] JLCPCB. Low cost pcb prototype & pcb fabrication, 2020.
[57] Jun Kato and Masataka Goto. F3. js: A parametric design tool for physical computing de-

vices for both interaction designers and end-users. In *Proceedings of the 2017 Conference on Designing Interactive Systems*, pages 1099–1110, 2017.

[58] Majeed Kazemitabaar, Jason McPeak, Alexander Jiao, Liang He, Thomas Outing, and Jon E. Froehlich. Makerwear: A tangible approach to interactive wearable creation for children. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 133–145, New York, NY, USA, 2017. ACM.

[59] M. Khamphroo, N. Kwankeo, K. Kaemarungsi, and K. Fukawa. Micropython-based educational mobile robot for computer coding learning. In *2017 8th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES)*, pages 1–6, May 2017.

[60] KiCad. Kicad eda, 2022.

[61] Jaeho Kim and Jang-Won Lee. Openiot: An open service framework for the internet of things. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 89–93. IEEE, 2014.

[62] Yoonji Kim, Hyein Lee, Ramkrishna Prasad, Seungwoo Je, Youngkyung Choi, Daniel Ashbrook, Ian Oakley, and Andrea Bianchi. Schemaboard: Supporting correct assembly of schematic circuits using dynamic in-situ visualization. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 987–998, 2020.

[63] Roman Kuc, Edward W Jackson, and Alexander Kuc. Teaching introductory autonomous robotics with javascript simulations and actual robots. *IEEE Transactions on Education*, 47(1):74–82, 2004.

[64] Mannu Lambrichts, Raf Ramakers, Steve Hodges, Sven Coppers, and James Devine. A survey and taxonomy of electronics toolkits for interactive and ubiquitous device prototyping. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(2):1–24, 2021.

[65] Mannu Lambrichts, Jose Maria Tijerina, and Raf Ramakers. Softmod: A soft modular plug-and-play kit for prototyping electronic systems. In *Proceedings of the Fourteenth International Conference on Tangible, Embedded, and Embodied Interaction*, pages 287–298, 2020.

[66] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Josiane Xavier Parreira, and Manfred Hauswirth. The linked sensor middleware–connecting the real world and the semantic web. *Proceedings of the Semantic Web Challenge*, 152:22–23, 2011.

[67] Richard Lin, Rohit Ramesh, Connie Chi, Nikhil Jain, Ryan Nuqui, Prabal Dutta, and Björn Hartmann. Polymorphic blocks: Unifying high-level specification and low-level control for circuit board design. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 529–540, 2020.

[68] Richard Lin, Rohit Ramesh, Antonio Iannopollo, Alberto Sangiovanni Vincentelli, Prabal Dutta, Elad Alon, and Björn Hartmann. Beyond schematic capture: Meaningful abstractions for better electronics design tools. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2019.

[69] Joanne Lo, Cesar Torres, Isabel Yang, Jasper O'Leary, Danny Kaufman, Wilmot Li, Mira Dontcheva, and Eric Paulos. Aesthetic electronics: Designing, sketching, and fabricating circuits through digital exploration. In *Proceedings of the 29th Annual Symposium on*

*User Interface Software and Technology*, pages 665–676, 2016.

[70] D Locke. Mq telemetry transport (mqtt) v3. 1 protocol specification. ibm developerworks technical library (2010), 2010.

[71] Mike Marsh. *Designing PCBs for Surface-Mount Assemblies*, pages 504–510. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[72] Victor Matos. Android environment emulator. *Cleveland State University*, 20(11), 2009.

[73] David A Mellis and Leah Buechley. Do-it-yourself cellphones: an investigation into the possibilities and limits of high-tech diy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1723–1732, 2014.

[74] David A Mellis, Leah Buechley, Mitchel Resnick, and Björn Hartmann. Engaging amateurs in the design, fabrication, and assembly of electronic devices. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, pages 1270–1281. ACM, 2016.

[75] Devon J Merrill, Jorge Garza, and Steven Swanson. Echidna: mixed-domain computational implementation via decision trees. In *Proceedings of the ACM Symposium on Computational Fabrication*, pages 1–12, 2019.

[76] Devon J Merrill and Steven Swanson. Reducing instructor workload in an introductory robotics course via computational design. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 592–598, 2019.

[77] Microchip. Atmega328, 2022.

[78] Microchip. Mcp9808, 2022.

[79] Micron. 8mb low-voltage serial nor flash: M25p80, 2022.

[80] Multisim. Live Online Circuit Simulator, 2020.

[81] Brad Myers, Scott E Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28, 2000.

[82] Zainalabedin Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[83] Mozilla Developer Network. Firefox os, 2013, 2013.

[84] Gavin Nicol, Lauren Wood, Mike Champion, and Steve Byrne. Document object model (dom) level 3 core specification. *W3C Working Draft*, 13:1–146, 2001.

[85] OpenAI. Gpt-3 demo, code generation, 2023.

[86] Sarah Osentoski, Graylin Jay, Christopher Crick, Benjamin Pitzer, Charles DuHadway, and Odest Chadwicke Jenkins. Robots as web services: Reproducible experimentation and application development using rosjs. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 6078–6083. IEEE, 2011.

[87] P. Li and J. Bachrach, "The Stanza Language", 2015.

[88] Raspberry Pi. Teach, learn, and make with raspberry pi, 2015.

[89] Proteus. PCB Design and Circuit Simulator Software, 2020.

[90] Raf Ramakers, Fraser Anderson, Tovi Grossman, and George Fitzmaurice. Retrofab: A design tool for retrofitting physical interfaces using actuators, sensors and 3d printing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 409–419, 2016.

[91] Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn

Hartmann, and Prabal Dutta. Turning coders into makers: the promise of embedded design generation. In *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication*, page 4. ACM, 2017.

[92] Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, Ted Selker, and Mike Eisenberg. Design principles for tools to support creative thinking. 2005.

[93] Annika Richterich. When open source design is vital: critical making of diy healthcare equipment during the covid-19 pandemic. *Health Sociology Review*, pages 1–10, 2020.

[94] Valkyrie Savage, Sean Follmer, Jingyi Li, and Björn Hartmann. Makers' marks: Physical markup for designing and fabricating functional objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 103–108, 2015.

[95] Sunil Saxena and HV Kwon. Tizen architecture. In *Tizen Developer Conference, San Francisco, California*, 2012.

[96] M. Schlett. Trends in embedded-microprocessor design. *Computer*, 31(8):44–49, Aug 1998.

[97] Evan Strasnick, Maneesh Agrawala, and Sean Follmer. Coupling simulation and hardware for interactive circuit debugging. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–15, 2021.

[98] Evan Strasnick, Sean Follmer, and Maneesh Agrawala. Pinpoint: A pcb debugging pipeline using interruptible routing and instrumentation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2019.

[99] Swoop. A python library for eagle pcb design files, 2015.

[100] Advanced Monolithic Systems. 1a low dropout voltage regulator, 2022.

[101] Joshua G Tanenbaum, Amanda M Williams, Audrey Desjardins, and Karen Tanenbaum. Democratizing technology: pleasure, utility and expressiveness in diy and maker practice. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2603–2612, 2013.

[102] Nest Learning Thermostat. v1. 0 [product brochure]. *Nest Labs, plus publication date addendum*, 2011.

[103] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[104] TinkerCad Circuits, 2020.

[105] Nicholas H Tollervey. Programming with micropython: Embedded programming with microcontrollers and python. 2017.

[106] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 803–812, New York, NY, USA, 2014. ACM.

[107] Nicolas Villar, James Scott, Steve Hodges, Kerry Hammil, and Colin Miller. . net gadgeteer: a platform for custom devices. In *International Conference on Pervasive Computing*, pages 216–233. Springer, 2012.

[108] Webcomponents. Specifications, 2020.

[109] Te-Yen Wu, Hao-Ping Shen, Yu-Chian Wu, Yu-An Chen, Pin-Sung Ku, Ming-Wei Hsu, Jun-You Liu, Yu-Chih Lin, and Mike Y Chen. Currentviz: Sensing and visualizing electric current flows of breadboarded circuits. In *Proceedings of the 30th Annual ACM*

*Symposium on User Interface Software and Technology*, pages 343–349, 2017.

[110] Stelios Xinogalos, Kostas E. Psannis, and Angelo Sifaleras. Recent advances delivered by html 5 in mobile cloud computing applications: A survey. In *Proceedings of the Fifth Balkan Conference in Informatics*, BCI '12, pages 199–204, New York, NY, USA, 2012. ACM.

[111] YouTube Player API Reference for iframe Embeds - Google Developers, 2020.