UNIVERSITY OF CALIFORNIA SAN DIEGO

Hungry for Fully Automated Design of Embedded Systems?

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

in

Computer Science (Computer Engineering)

by

Devon James Merrill

Committee in charge:

Professor Steven Swanson,
Professor Chung-Kuan Cheng
Professor Ryan Kastner
Professor Leo Porter
Professor Paul H. Siegel

2021

The dissertation of Devon James Merrill is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

EPIGRAPH

The popular stereotype of the researcher is that of a skeptic and a pessimist. Nothing could be further from the truth! Scientists must be optimists at heart, in order to block out the incessant chorus of those who say "It cannot be done."

—Academician Prokhor Zakharov

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Professor Steven Swanson for his support as the chair of my committee. Without his guidance, encouragement, and advice, this work could not have been completed.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the 50th ACM Technical Symposium on Computer Science Education 2019. Merrill, Devon; Swanson, Steven, Association for Computing Machinery, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the ACM Symposium on Computational Fabrication 2019. Merrill, Devon; Garza, Jorge; Swanson, Steven, Association for Computing Machinery, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is currently being prepared for submission for publication of the material. Merrill, Devon; Lin, Ting-Chou; Holtz, Chester; Wu, Yen-Yi; Garza, Jorge; Swanson, Steven; and Cheng, Chung-Kuan. The dissertation author was the primary author of this chapter.

VITA

2014	B. S. in Computer Engineering, University of California San Diego
2015-2018	Graduate Teaching Assistant, University of California San Diego
2016, 2017	Guest Instructor, San Diego Mesa College
2017	M. S. in Computer Science, University of California San Diego
2019	Instructor, University of California San Diego
2021	Ph. D. in Computer Science (Computer Engineering), University of California San Diego

ABSTRACT OF THE DISSERTATION

Hungry for Fully Automated Design of Embedded Systems?

by

Devon James Merrill

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2021

Professor Steven Swanson,

Computational design techniques have the potential to fully automate the process of designing embedded electromechanical systems, including component selection, PCB design, firmware creation, and more. The benefits of a practical, fully-automated PCB layout flow are numerous and far-ranging, including the potential for new embedded-system design interfaces, new pedagogical methodologies, and vastly reduced prototype turnaround time. While an automated system that designs smartphones and server motherboards competitive with human engineers is beyond the current state-of-the-art, this dissertation details several computational design systems that automate the design of less complex embedded systems, primarily at the PCB level: a pedagogical computational design tool for small robots; a methodology for creating

advanced computational design tools for electromechanical systems using heuristic search over a decision tree; and a framework and benchmarks for fully-automated PCB layout.

Chapter 1

Introduction

There are too many electronics that need to be designed and not enough people to design them. The current methods and tools we use to make electronics are outdated and unnecessarily tedious, even for experts. If we can make computers do the work of designing, we solve those two problems.

This dissertation is about automating the engineering of embedded electronic systems with a focus on design at the level of printed circuit boards (PCBs). The following chapters discuss several projects that automate embedded systems design using computation. The first of these projects is a university class taught using a computational design tool that facilitates the design of PCB-based robots. The second is a computational design tool called Echidna. Echidna designs complex electronic and mechanical systems from a functional specification. The third is a framework and set of benchmarks for fully-automated, no-human-in-the-loop layout for PCBs.

This introductory chapter begins with a discussion of the confluence of commercial, academic, and pedagogical circumstances around computational design for embedded devices which has motivated the work presented in this dissertation. This is followed by briefly defining computational design compared to computer-aided design in general. Then a brief summary of the projects and a summary of the results follows. The three projects are detailed in the following

chapters.

1.1 Motivation

There are systematic, industry-wide problems can be solved by the development of software tools that apply software engineering techniques to embedded systems design. This work explores the question of what technical innovations are necessary to create such tools and presents several tools which do solve these problems for simple embedded devices.

In both academic research and commercial applications, the benefits of a practical, fully-automated PCB layout flow would be numerous and far-ranging, including the potential for new embedded-system design interfaces, new pedagogical methodologies, and vastly reduced prototype turnaround time. Previous research in the area of PCB layout automation lacks practical application to industry, however, and academic research has fallen behind commercial practice. For example, multilayer PCBs, non-uniform pad shapes, pad entry, variable trace width, among others, are ignored in PCB research tools. As a potential cause, researchers working on CAD for PCBs are often not familiar with commercial PCB design practices.

Illustrative of the industry (and academic) inertial bias against fully automated design, our discussions with PCB engineers reveal an entrenched skepticism towards completely automated techniques for PCB layout resulting from the failure of past commercial attempts. Overall, it appears that experts in embedded systems and PCB design do not believe automated tools can produce commercially-relevant outputs and that humans are required to guide or adjust the output of any automated tool. Also, to embrace computational design would require large changes to the current industry workflow, not incremental improvements to the current process. Only robust, realistic research can overcome this distrust.

If a practical, fully-automated PCB layout flow can be achieved, the benefits would be numerous and far-ranging, including the potential for new embedded-system design interfaces,

new pedagogical methodologies, and vastly reduced prototype turnaround time. Computational engineering systems for embedded electronics (including Echidna) are limited to low-density electronics. The commercial tools that are available are limited in interfacability, automatability, and ease-of-use for non-experts. The limited commercial tools limit the application of computational design systems for advanced PCB designs.

Embedded systems, being pervasive in modern life, are of increasing interest and relevance for non-computer-science majors. Automating embedded system design can improve student experiences in these circumstances, especially, as we have found, in small-group interaction. There is growing enrollment in CS, which has increased class size. Increased class size has a negative impact on student-teacher interaction. Small group interaction and student-teacher interaction are important in student retention, especially in computer science, where student retention is a major problem.

Outside of the classroom, the popularity of the Maker movement has driven a surge in the hobbyist projects in designing, building, and programming embedded electronics projects at all levels. Hobbyists with little or no basic knowledge in CS must spend time learning the basics of programming and electronics as a prerequisite to completing their projects. For experienced engineers, much time is spent applying procedural knowledge and well-known rules, a tedious and error-prone process. Partial design reuse and open, standardized libraries ameliorate similar problems in software engineering. However, in embedded systems, there is little formal reuse, and libraries are often coded in proprietary formats. Much of the design process is rote application of well-known rules, a tedious and error-prone process.

1.2 Background

Engineers have used computers to automate aspects of engineering in numerous fields, including electronics engineering, since the advent of computers. Particularly, the field of

computer-aided design (CAD) for integrated circuits (IC) has made the complexity and capabilities of modern computers possible. With billions of individual transistors, the ICs in today's computers could not be designed without substantial automation. For printed circuit boards, commercial CAD systems are used in drafting, human-guided automatic routing, design rule checking, and electrical simulation.

Beyond computer-aided design is computational design. Computational design is the use of computer programs to automatically create an implementation based on a design, which is provided by a user. The process of creating an embedded system includes both this design aspect and the implementation aspect. A design is a functional specification that describes what an artifact (in this work, an embedded system) does. As an example, the design of a hammer is: a tool that can be used for the jobs of breaking things and driving nails. A design is the "why" of the artifact. An implementation is the sum of the technical information that describes how the artifact is to be made (that is, manufactured). A possible implementation of a hammer could be: a heavy metal head mounted at a right angle on the end of a wooden handle. An implementation is the "how" of the artifact.

In a conventional CAD workflow, implementation is performed by a user with the assistance of CAD tools. Conventional CAD tools allow the user to specify every detail of the implementation. CAD tools are considered powerful if they allow the user to easily understand and specify complex implementation details. In contrast, computational design tools hide the implementation details from the user to allow the user to focus on design. Computational design tools are considered powerful if they can create implementations of designs without guidance from the user.

When using a computational design system, the human user does "pure" design work, that is, inputting the functional specification and, possibly, selecting from several automatically-generated implementations; the implementation is done completely by a computational system.

1.3 This Work

We attack the question of computational design for electronics from three different directions that approach this problem from a technical and practical base – demonstrating both that it is possible, and that it can fundamentally alter how people design. First, we taught a robot design class using an application-specific computational design tool. Second, we designed and implemented a more general-purpose computational design tool for embedded systems. Third, we created a framework for automated PCB design to address the primary bottleneck in contemporary computational design systems for embedded systems.

As a practical study of our initial computational design system, we designed and taught a one-unit introductory physical computing course with first-year computer science students based on constructionist learning theory [HP91], the theory that making a thing creates knowledge, as opposed to passively receiving information, which does not.

A primary goal of the course was to determine if students could successfully complete a new and complex task with intense coaching but minimal instructor assistance. Students engaged in problem-based learning with a randomly assigned partner and were tasked with designing, building, and programming a simple robot with specific criterion-assessed objectives. The robots included a custom PCB and were built with off-the-shelf electronic components. Creating custom PCBs is time-consuming, especially for students with limited experience. We created a computational design tool to streamline the process. This tool, named Robot Factory, uses computational design techniques to create a custom PCB design, custom firmware, a parts list, assembly instructions, and assisted with instructor management of the various designs. Without Robot Factory, the class would have required an impractical amount of instructional support.

Robot Factory evolved into a more complex computational design tool, named Echidna. Echidna automates the mechanical, electronic, and software implementation of embedded systems. The input to Echidna is a set of functions (or functional components); output is a set of "utility"

components and the connections between all the components. This work differs from other computational design tools for electronics by employing multidomain codesign via a heuristic search, rather than templating, to codesign the software, mechanics, and electronics of devices. A heuristic search of the design space allows Echidna to design more varied and complex embedded devices without using a fixed topology. Echidna uses a library of components with typed interfaces to produce a custom topology for each design.

The lack of a fully-automated PCB layout tool limited Echidna from creating embedded systems with high-density PCBs and advanced electrical constraints. To support Echidna and other tools like it, we developed an open PCB layout framework. To use the framework, users input unplaced, unrouted PCB designs into the framework; the toolchain output placed and routed PCB layouts. Included with the framework are benchmarks that provide an open and standardized measure of PCB layout performance to aid future research for advanced PCB layout. The framework is open-source software capable of laying out advanced PCB.

1.4 Results

The projects presented in this dissertation successfully met their ends. There was an improvement in the GPA of first-quarter university students who completed the class using our computational design tool, indicating that computational design systems are useful in a pedagogical context. We demonstrated that end-to-end automation is possible for generic embedded systems with custom topologies. The complete process of embedded engineering can be fully-automated as computational design. And, we expanded the capabilities of computational design tools for embedded systems by demonstrating a fully-automated, no-human-in-the-tool DRV-free layout of a complex single-board computer (SBC), the BeagleBone Black, using low-speed constraints.

Each project is detailed in the following chapters: Chapter 2 presents our experience using

a computational design system for small robots in the context of a university engineering class. Chapter 3 presents Echidna, a system that automates the creation of embedded systems (including PCB design, component selection, API generation, and so on) using heuristic search over a decision tree. Chapter 4 presents a framework and benchmarks for the no-human-in-the-loop layout of advanced PCBs. Chapter 5 concludes.

Chapter 2

Robot Parade: A Robotics Course via Computational Design

Physical computing has essential benefits for novice engineers and computer scientists. However, lab time and hardware debugging come with a high cost of instructor time and effort. We implemented a computational design tool that simplifies printed circuit board (PCB) design and manufacture, assembly, and programming to reduce this workload. We pilot tested our computational design tool in a one-unit introductory physical computing course for 196 CS1 students. The students designed, assembled, and programmed a custom robot with minimal instructor assistance. The robots are Arduino-based, and each included a student-designed PCB. The students assembled the robots from off-the-shelf electronic components according to automatically generated assembly instructions. Students programmed their robots using simple APIs that were automatically generated and customized for each unique robot. A minimum of two quarters after the completion of the course, the grade point average (GPA) for the students who completed the course, was found to be 0.15 higher than a comparison group of similar students ($n = 498$, $p < 0.05$). We present a detailed description of our computational design tool and course curriculum, identify challenges encountered by the students and instructional staff, make

recommendations to increase student achievement, and address the scalability of the course.

2.1 Introduction

A diverse range of computing devices is becoming deeply embedded in almost all aspects of our day-to-day lives. The increasing importance of computing and its growing economic impact drive enormous growth in undergraduate demand for computing courses. These two trends lead to challenges in providing large numbers of students with experiences that straddle the boundary between software and the real world. Furthermore, growing enrollments often lead to large undergraduate classes that can be impersonal and make it difficult to have effective small-group interactions.

We have developed Robot Parade, a lightweight lab course for freshmen computer science students taken concurrently with CS1, usually during their first quarter at college. We built the course to show students that they can conceive, build, and program a complete computing device of their design. Robot Parade aims to provide a low-stress, enjoyable venue for students to practice what they learn in CS1 while interacting with other students in a smaller, more intimate setting than the large lecture format of our CS1 course can offer. Students complete nearly all the coursework in class during weekly two-hour labs, held nine times during the quarter.

Using robots in introductory programming courses can improve attitudes toward computers in general [KL16], and this approach has been developed as a direct application of Piaget's constructionism learning theory (e.g., by Rosenblat and Choset [RC00]). However, teaching these courses can be a resource- and time-intensive. Rosenblat's course, for instance, relied on a low student-to-instructor ratio and pairing experienced senior students with less advanced students. These problems are especially acute for courses targeted at less-experienced students.

We developed Robot Parade to make a hands-on programming and building experience accessible to students during their first term studying computer science in college. A key

motivation for targeting students early is to reduce attrition among underrepresented groups of CS and CE majors.

The course centers around a single project: students design, build, and program a simple wheeled robot that can sense and react to the environment. The project includes creating and assembling a custom printed circuit board (PCB).

Enabling first-term students who are learning to program and have little (or no) experience with electronics to complete the project has required us to carefully craft the course's content and schedule and develop a robot design tool that is accessible to our students.

During the first half of the course, students complete several robot programming labs using pre-built robots that include all the elements they will eventually use in their designs. Each lab focuses on a different aspect of sensing or control (such as driving, controlling lights and a speaker, using a distance sensor). These labs help students learn the basics of Arduino-style microcontroller programming, and grapple with the vagaries of using code to gather data from sensors and control physical devices. They also encounter the rudiments of C++ (we teach CS1 in Java), gain experience assembling electronics (i.e., soldering and wiring), and learn their way around the "Makerspace" where we teach the class and some of the tools (e.g., 3D printers) it provides.

Next, they design a robot to carry out a task of their choosing. "Tasks" include things like "look and act like a duck," "be polite to other robots," or "drive around and pop wheelies."

The key component of the robot is a printed circuit board (PCB) that will host its electronics and serve as its body. Designing PCBs is a complex task that usually requires experience with electronics and the steep learning curve of conventional PCB design tools.

To make PCB design tractable for our students, we have built a web-based, what-you-see-is-what-you-get (WYSIWYG) robot design tool called *Robot Factory*. Robot Factory lets students design a working robot in a single class period with very little training. Behind the scenes, Robot Factory uses computational design to algorithmically generate a PCB that implements the

circuitry for their design.

We have their PCBs manufactured, and the students solder them together by following customized assembly instructions for their robot that Robot Factory generates. Students program the robots with the Arduino IDE using a customized library that Robot Factory generates to match their particular robot design.

We have taught Robot Parade in five academic quarters over the last three years. Space is limited primarily by instructor "bandwidth," so we select students by lottery from those that submit a simple application. We typically run two sections of 20-30 students working in pairs. We have been working steadily to increase the number of students that may enroll in the course by streamlining our curriculum and refining our robot design tool.

We recruit students who concurrently take a CS1 course alongside our robot course. The majority of the participants do not initially have basic programming skills such as knowledge of variables, functions, and control structures.

The following sections describe Robot Factory and its interface (2.2), detail the course content (2.3), provide a pilot evaluation of the course's impact on student academic success (2.4), and discuss plans for future improvements to the course (2.5). 2.6 draws conclusions from our work.

2.2 Robot Factory

We want students to have the experience of building a real, working robot without requiring them to scale the steep learning curves that hardware design tools usually require. To make the task tractable and allow the students to focus on the *design* of the robot rather than the details of its implementation, we developed a computational design tool called Robot Factory that makes designing a robot easy and automates much of the implementation.

Robot Factory caters to students with no electronics design or programming experience,

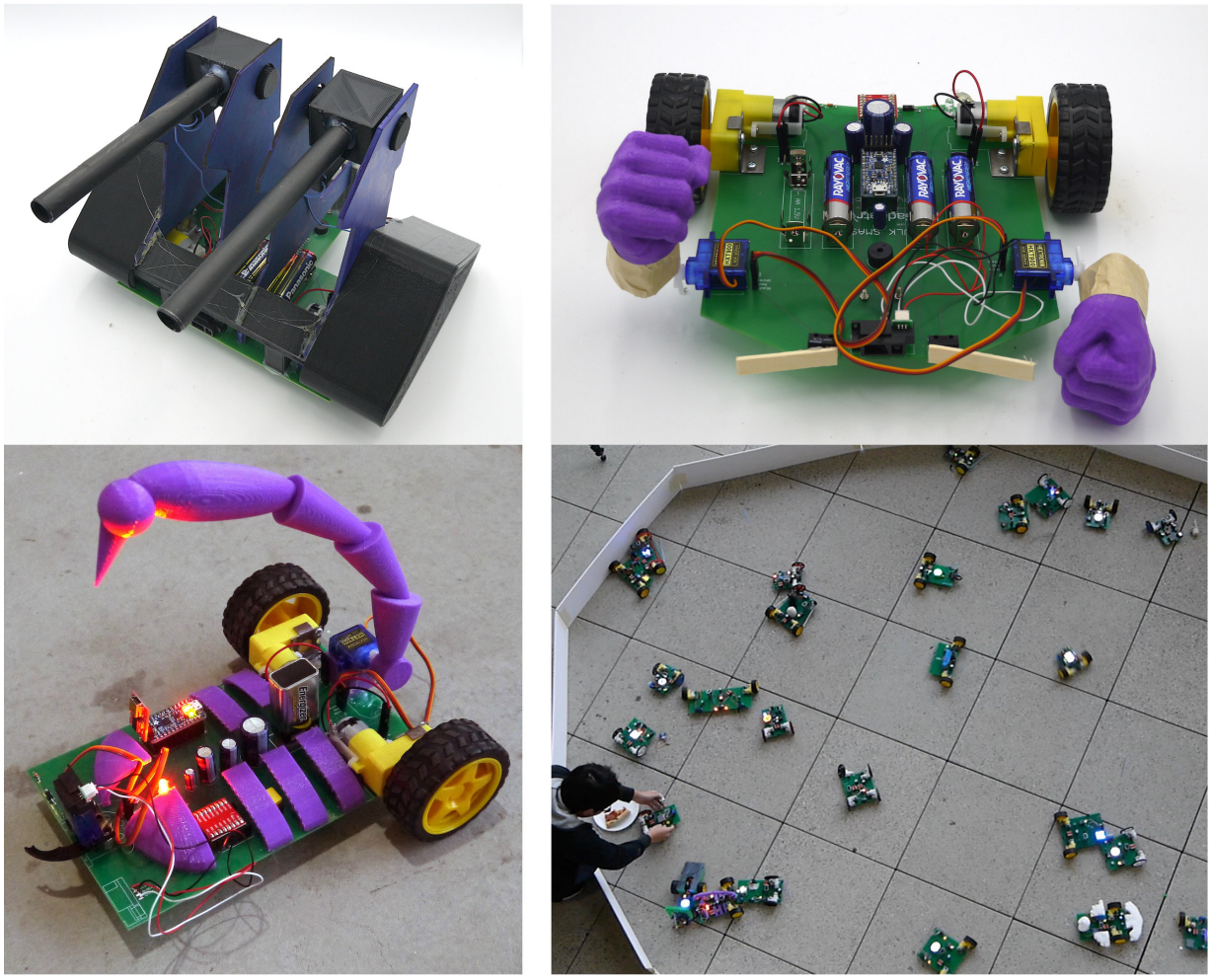

Figure 2.1: Several student designed robots.

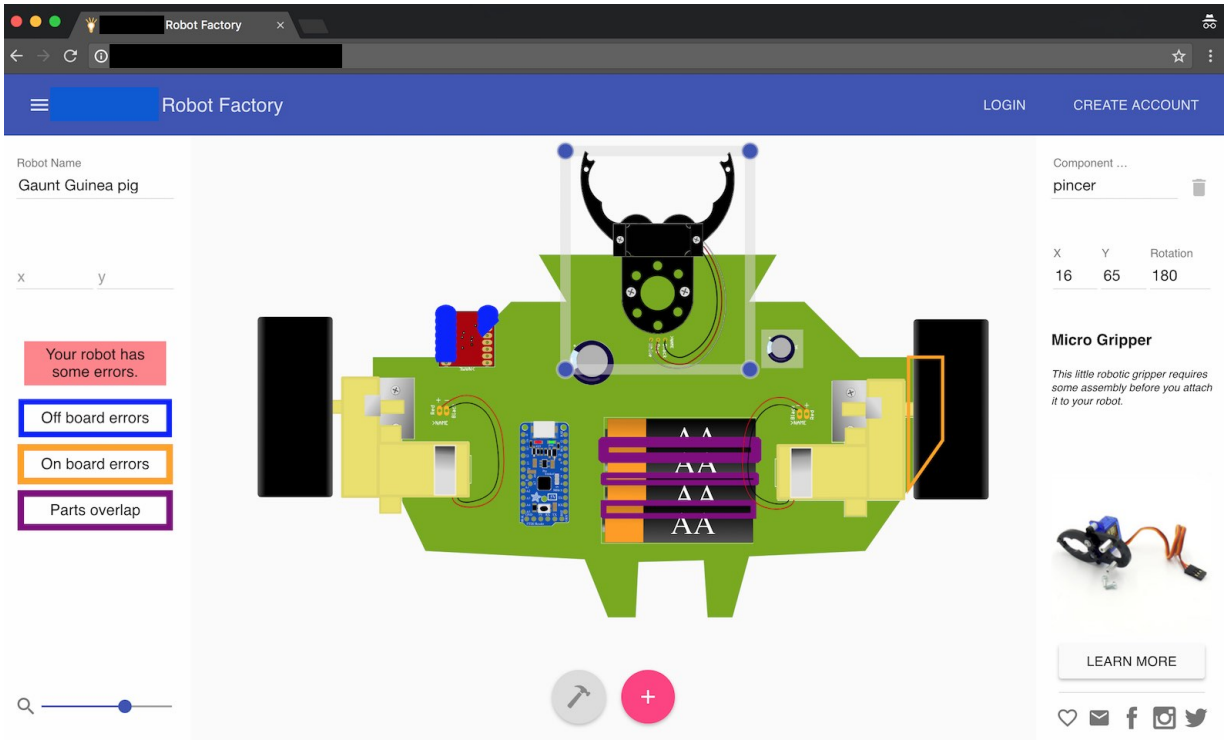

Figure 2.2: The computational design tool GUI. An in-progress robot design showing several types of placement errors.

and it minimizes the amount of instructor assistance and prompting required for success. To achieve both of these goals at once, we built it to be simple to use, provide real-time feedback about potential design problems, and use a familiar "drag-and-drop" paradigm for design. As a result, students can create a valid robot design after just a short two-minute introductory demonstration.

Robot Factory also generates an easy-to-use high-level application programming interface (API) for each robot. The API uses concepts familiar to students taking CS1, so programming the robots is relatively easy as well.

Below, we describe the tool's interface and give an overview of how it works internally. It is available for anyone to use (http://robots.gadgetron.build/).

2.2.1 User Experience

Robot Factory provides a web-based, drag-and-drop GUI that students use to design the shape of their robot and the components it will comprise. When it opens, it presents students with a minimal robot design on a green polygon that represents the PCB. The minimal design includes a ProTrinket [Ind] Arduino board and four AA-size batteries.

Students add *functional components* to the PCB from a library. They can choose from motorized wheels, buttons, single LEDs, LED arrays, IR distance sensors, actuated claws, bump switches, and a few other items.

Robot Factory automatically adds *utility components* that the robot must include to work properly and synthesizes the PCB circuits to connect all the components together. Utility components include resistors and capacitors, motor drivers, and protection diodes. Large utility components that will significantly affect the appearance of the robot (e.g., the batteries and large capacitors) are visible in the user interface so students can choose where to place them. The tool automatically places small utility components (e.g., resistors).

After students are satisfied with their design, they press the "build" button. This button sends the robot specification to synthesis servers running in the cloud.

Placement errors

The locations of the components are subject to several constraints that the student designs must satisfy (the wheels must be off the board, parts cannot overlap, and so on). Typical PCB design tools provide a separate design rule check (DRC) command to check some of these constraints, but the output can be confusing, the tools tend to generate many redundant errors, and it can take several iterations to resolve all the problems. Furthermore, some errors can safely be ignored, but knowing which ones are spurious requires expert knowledge.

Robot Factory makes it easy to design correct robots with real-time feedback about placement restrictions. Difference colors indicate each of the three types of constraint (must be

on the board, must be off the board, and no overlap). When a constraint is violated, the area of the component that violates the constraint is highlighted with a pulsing outline. The first time each type of error is detected, a pop-up window explains the error and how to resolve it.

If an error exists, a prominent indicator in the GUI informs the user, "Your robot has errors." If no component has a placement error, the indicator becomes green and reads, "Your robot looks great!" Students can only submit error-free designs.

To ensure students learn to deal with errors quickly, we place the initial design components, so they overlap. Students typically respond by resolving this placement error as their first interaction with the tool. This first interaction facilitates student willingness to experiment with component placement and resolve errors on their own.

Board shape

Letting students express their creativity is an important part of the class, but our initial implementation of the tool restricted students to rectangular boards, limiting the range of designs. Since then, we have added support for more complex shapes.

When we first introduced the ability to create complex robot shapes, we noticed that almost every student-created robot still had a rectangular board. We believed that this is because the starting board shape was rectangular. In the latest iteration of the tool, we randomized the vertices of the starting polygon, which led to a much larger diversity of student designs. For instance, one team designed their board in the shape of the letters "LA" and then programmed the robot to dance to music from the movie "LA LA Land."

2.2.2 Back-end

Robot Factory must be as automatic as possible to support scaling the number of students we can support in a single class. In the vast majority of cases, translating a student's design into the hardware and software required to implement it must not require any direct intervention

from the course staff. Furthermore, it must, to the extent possible, guide the students through the process of assembling and programming their robot.

Robot Factory synthesizes a complete manufacturing specification from the students' designs. The manufacturing specification includes PCB computer-aided manufacturing (CAM) files, C++ API files, assembly instructions, and bill of materials. The complete process takes between one and five minutes, with the majority of that time spent on auto-placement and routing for the circuit board.

Synthesizing a complete manufacturing specification from an incomplete graphical specification is a multi-step process. We discuss each of the steps below.

Component-level synthesis

Component-level synthesis makes the necessary electrical connections between the functional components in the design and adds utility components that are not visible in the tool's interface. For example, students add motorized wheels to their design and choose where to put them, but the motors require more current than the microcontroller can provide. In response, Robot Factory automatically adds and connects a motor driver board and a large capacitor. The motor driver board requires several GPIO connections to the microcontroller, logic-level power, and motor power. So, Robot Factory connects (and adds if not already present) batteries and the microcontroller.

Some robot designs are not synthesizable. For example, if the student adds 20 LEDs, but the microcontroller only has 10 GPIO pins available, not all of the LEDs can be connected. Our tool reports this error to the student. It lists all components that use GPIO pins and suggests removing some of them.

We provide guidelines to the students, so these errors occur infrequently, but the details of these connection constraints are intricate. For example, many of the Arduino's pins are multi-function, and some functions require access to limited microcontroller resources such as timers.

We encode these constraints in a generalized, abstract interface specification for each component. Using heuristic search, Robot Factory can perform component-level synthesis on student robot designs in less than 30 seconds.

Auto-placement

Components must be placed on the PCB without violating their placement constraints. Robot Factory's real-time feedback ensures that the students have correctly placed the functional components and that the utility components are visible. Robot Factory adds additional utility components and must find places for them on the PCB. A general solution to the placement problem is not feasible (it is at least NP-complete), but our robot designs are not space-constrained, so a simple "first fit" is appropriate and usually completes in under 15 seconds. If auto-placement cannot find a valid placement, Robot Factory returns an error and asks the student to make more open space in their design. The auto-placer has never failed in practice.

Electrical auto-routing

The routing process translates the logical connections created during component-level synthesis into metal traces on the PCB.

PCB routing can be very challenging. As with placement, however, the relative simplicity of Robot Factory designs lets us successfully automate the process. Robot Factory uses Autodesk Eagle's autorouter.

If a design fails to route, synthesis aborts, and the student is prompted to try an alternate placement of their components. Out of the more than 100 designs students have created with our tool, fewer than five have failed to automatically route.

Source code generation

Students in the course are concurrently taking a first programming course taught in Java. The robots we generate are Arduino-based, so they must program in (a simple subset of) C++. Java and C++ are similar enough that students are capable of creating simple programs for their robots. We typically spend about 15 minutes discussing the differences between Java and C++, and students do not usually struggle with the transition.

The Arduino API presents a more considerable obstacle. Arduino provides a low-level pin-oriented API that can read and write digital and analog values to and from pins on the microcontroller. The link between these low-level operations and higher-level robot behaviors are not always simple. For instance, driving forward requires manipulating several pins at once.

To streamline the process of programming their robots, Robot Factory provides a "starter" program built specifically for each robot. The program runs "out of the box" on their robot and tests the operation of each component. It flashes any LEDs, drives forward, and so on. The starter files also instantiate a C++ object for each functional component in the design with a name that corresponds to the name of the component the student specified in GUI. These objects support simple, intuitive methods such as LED.turnOn(), LED.turnOff(), wheels.forward(), and wheels.turnLeft().

These starter programs are very effective in reducing instructor workload in class. When used without modification, they quickly identify malfunctioning components. Troubleshooting speeds up tremendously because students can accurately report and demonstrate issues.

If a robot is not working when loaded with student code, students tend to blame the robot's hardware. By reloading the starter code, they can retest the hardware quickly. If the test code works, they know it is a problem with their code, and they usually attempt to fix it themselves.

Assembly Instructions

Since each robot is different, the process of assembling them varies. To help students assemble their robots, Robot Factory provides customized assembly instructions as a web page that includes an illustrated parts list, shows them where each part goes, and provide instructions for assembly. The instructions also include videos illustrating basic soldering techniques.

2.2.3 Design management

Student teams create an account in our tool that includes a team name, password, recovery email address, and association with a class section.

When a student's design is finished, the design is saved and becomes visible to instructors. When the students are satisfied with the design, an instructor confirms the design, and then it is ready to be manufactured. When all the designs are ready, Robot Factory generates a combined bill of materials and makes the PCB design files easily available for ordering.

2.3 Course Content

Robot Parade has two phases. In the first, students program robots similar to those they will eventually build. They also learn what is feasible with the components that Robot Factory provides. In the second phase, they use this knowledge to guide the design of their own robots and then assemble and program them.

We teach the class in a makerspace. Students use 3D printers and other tools to create decorations for their robots and soldering equipment for assembly. We encourage students to use other equipment or bring in materials for decoration. The only out-of-class homework assignment in the course is to 3D something by the end of the fourth week. This helps engage them with the tools in the makerspace.

2.3.1 Grading

Robot Parade is meant to be a low-stress, fun opportunity for students to practice programming and broaden their horizons within computer science. The grading structure reflects this goal. The course is "Pass/Fail" and does not affect student GPA. We record attendance and the completion of the lab assignments. We use these records to identify students who may need additional coaching and encouragement, not for assigning grades.

2.3.2 Introductory Labs

The course begins with two introductory labs in which the students program two predesigned robots to complete a variety of tasks. The labs are spread across three to four class meetings (a total of 6-8 hours). Groups work at their own pace with some encouragement from the course staff to stay on schedule.

The first lab involves driving the robot around on paper and using a marker to trace the path the robot has taken. This directly connects to a "turtle graphics" assignments they are working on in their concurrent CS1 course and that is common as an introduction to both procedural and object-oriented programming [CC00]. Students are eager to engage in the familiar task and quickly connect with the platform.

Drawing with a robot and drawing with a virtual "turtle" provide vivid lessons about the differences between programming the digital and physical worlds. For example, turning the robot precisely is hard while turning a turtle precisely is trivial.

The major tasks assigned in the first lab are to "draw your favorite shape" using the robot and marker, and to program the robot to "sing and dance" using LEDs and a small speaker on the robot.

Both of these assignments are freeform and credit is given for any attempt. However, if there is time left in class, we encourage them to do more. Although the students know that they

will receive the same mark for any attempt, they become invested in the free-form assignments and often create elaborate programs.

The second lab involves interacting with the environment using a distance sensor and a motorized gripper. Students program the robot to maintain a constant distance from a moving object, "explore the room without getting stuck," and locate, grasp, and move a 3D-printed object. Students are told that they must use a state machine programming pattern to complete the final task, and we preface the lab with a brief lecture about the state machines and how they apply to robots.

We have taken steps to reduce the dependence of students on course staff during labs. The most helpful of these has been to put together a troubleshooting list. This list contains seven solutions to common problems that students have. These problems are:

1. Plugging the programmer in the wrong orientation
2. Not plugging the USB cable in (properly or at all)
3. Selecting the incorrect serial port
4. Selecting the incorrect Arduino board type
5. Selecting the incorrect programmer type
6. Using the wrong version of the Arduino IDE
7. Having a dead battery

We have found that if the student's trouble is not on this list, then the problem generally cannot be solved quickly in class and the student should move to another workstation or switch to a different robot.

By using this list we have been able to focus the majority of in-class instructor effort on coaching program design and encouragement. However, the students need frequent reminders to check the list if they have trouble.

2.3.3 Robot Design

We weave the concept of creating a robot into the class from the beginning, culminating in "design day" when students design robots with Robot Factory.

During the introductory labs, we encourage students to come up with a concrete, specific concept for their robot. This leads to the more interesting designs. Some examples of successful concepts include Polite Bot (which would drive around a tip a 3D printed top hat at other robots), Duck Bot (which acted like a duck), Smash Bot (which would wave around a big pair of 3D-printed fists. See 2.1, top-right), R2-D2 bot (which looked like R2-D2 from Star Wars), and so on. Concepts like "it's going to drive around and grab things" lead to less interesting designs and less student engagement.

The bulk of the design process takes most teams 60-90 minutes, but many teams spend the full two-hour class period tweaking their design. By the end of the class period, all the student designs are ready to be manufactured.

2.3.4 Robot Assembly Logistics

The next step is manufacturing the PCBs and assembling the robots. We order the boards from Advanced Circuits [adv], quick-turn PCB manufacturer that will make boards for students for $33/board. The boards take at least seven days to arrive, so there is effectively a 2-week delay between design and assembly. The students spend the intervening class period working on 3D-printed decorations for their robots.

We have engineered the robots so that the most expensive components (the distance sensor, servo motors, drive motors, microcontroller, and LED arrays) are reusable. They all have "headers" that plug into sockets that students solder to the board. The components that are soldered to the board (resistors, capacitors, diodes, and so on) are disposable. Altogether, it costs about $50 to build each robot.

2.3.5 Robot Assembly in Class

Assembling the robots requires basic soldering as well as screwing and hot-gluing some components together. Most students have never assembled PCB or soldered before.

Assembly proceeds in several stages. These steps typically consume between one and two class periods.

Lecture

We give a very brief lecture about the assembly process (described below) and explain the notion of polarized components (that is, components with a symmetrical arrangement of pins that must be inserted in the correct orientation) and how to tell if they have them assembled correctly.

Part collection

Before class, the course staff sets out trays with all the components the students will need. The trays are arranged in the same order that the parts appear on illustrated parts lists for their robots and are labeled with the same pictures. Students take a small plastic bin, form a line, and move down the row of bins collecting their parts.

Dry fitting

Students "dry fit" the components to their PCBs by using tape and inserting and bending the wire leads to hold them in place. A member of the course staff checks that everything is correct.

Solder lesson

An instructor gives a quick soldering lesson to between one and three groups at a time. It covers how soldering works, what a good solder joint looks like, and a demonstration of how to solder each kind of connection they will need to make on their board.

Soldering!

Students solder all the components to their PCB.

Solder check and final assembly

Course staff checks each soldering job. If it looks good, the students trim the leads on the components. Then, they plug in the microcontroller and other reusable parts. They have a robot!

Testing

The last step is to run the test program that Robot Factory provides. If something does not work, the course staff helps them resolve the issue.

2.3.6 Robot Programming

The rest of class time in the course (1-2 weeks) is allocated to programming, decorating, and fine-tuning the software for the robots. We coach the students to use a state machine pattern to enable responsive interactivity. Because students have spent several weeks in the beginning of the course programming similar robots, they are generally able to program their own robot without assistance.

We notice an over-reliance on delay statements for program coordination (similar to other observations of early programmers [WHHF18]). We are continuing to develop strategies to help students be successful in creating real-time interactive programs in an embedded environment.

2.4 Evaluation

We have taught the course in five academic quarters, Fall 2015, Spring 2016, Fall 2016, Winter 2017, and Fall 2017 and performed a preliminary analysis of how taking the class impacts

student success over time. Between Fall 2015 and Fall 2017, 498 students applied to take the course. We admitted students from the applicant pool using a random lottery.

Our analysis shows that completing Robot Parade resulted in a statistically significant increase in overall GPA (Welch's t-test p=0.0011). The mean, cumulative GPA of students who completed the course, as of Summer 2018, is 3.43 (196 students, standard deviation 0.45 points, 4.0 scale). The mean GPA of the comparison group of students who applied to take the course, as of Summer 2018, is 3.29 (302 students, sd=0.56). The difference between the means is 0.148 points.

This pilot analysis indicates that completion of Robot Parade resulted in improved academic performance, and we are encouraged to continue developing the course. A full analysis, including the effect on major retention and with regards to gender is in progress.

Anecdotally, students enjoy designing, building, and programming (and decorating) their robots a great deal. Students attribute unique personalities to their robots, and to other students robots. Many students create elaborate stories about their robots' motivations and behaviors. We have observed that these student story- telling behaviors seem to be absent or strongly attenuated in similar first-year seminar robotics classes where all students are given identical robots. Students "owning" the robot that they designed and built themselves seems to be very engaging and motivating, echoing the findings of previous work [6]. Students often express a desire to keep and to continue working on their robot after the course is completed.

Also, because of the PCBs that forms the core of the robots, students feel that their robots are "real" electronics, as opposed to bread-boarded electronics or toolkits such as Lego Mindstorm. The concept of students designing and building "real" embedded devices seems to be empowering and motivating to the students.

2.5 Future Work

Although the Robot Parade class and The Robot Factory design tool have met their original technical goals, we will continue to evolve the course with two further ambitions in mind.

The first is to continue increasing class size. To date, we have increased the scale of the class from 5 robots per class to 15 robots per class without adding course staff. We have plans to scale the course to 60 students (30 robots) per section in the next iteration (Fall 2018). Scaling the initial labs to that size should be easy, but we expect that efficiently assembling that many robots will become challenging.

One option would be to use solder-free assembly. To achieve this we could build (or buy) a standardized robot board and have students attach it to a laser-cut frame that would correspond to the robot shape they designed with The Robot Factory. They would connect electrical components with "hookup" wire. The resulting robots would be cheaper and easier to assemble, and it would open up more time in the course for programming, but we are worried they would be less engaging and exciting for the students.

Our second ambition is to integrate the course more deeply into our CS curriculum. Currently, the course is optional and loosely coupled to the introductory CS courses. Coupling it more tightly or expanding it to be an alternative format for introductory computing, perhaps in conjunction with our robotics program, would be very exciting.

2.6 Conclusion

Robot Factory makes it possible for Robot Parade to give students hands-on experience building working robots early in their computer science careers. Of equal practical importance, Robot Factory makes it easier for student to guide themselves through key parts of the design process, facilitates debugging, and eases the burden on course staff of organizing and ordering robot components.

The feedback from students and the growing demand for the course shows that there is a great appetite for this kind of class among CS students. Our preliminary data suggest it may have a positive impact on their long-term success. We plan to continue refining the class and exploring how to increase the benefits to students.

2.7 Acknowledgement

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the 50th ACM Technical Symposium on Computer Science Education 2019. Merrill, Devon; Swanson, Steven, Association for Computing Machinery, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Echidna: Computational Design for Embedded Devices

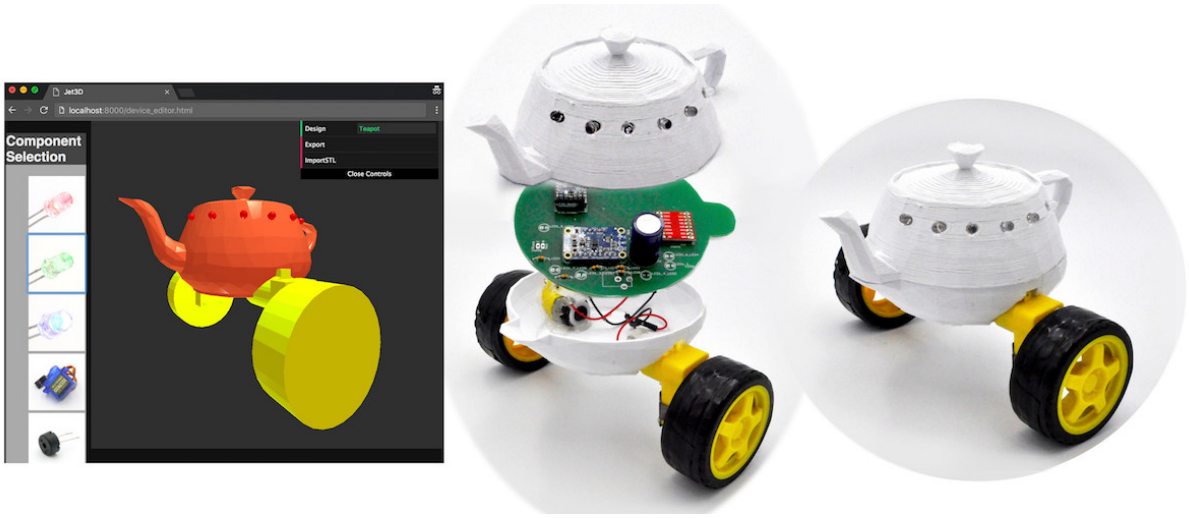

Figure 3.1: Overview of Echidna.
Users start with a functional, graphical specification (left). Echidna creates ready to fabricate manufacturing files. The user then assembles the fabricated components via generated assembly instructions into a complete device (right).

The Robot Parade class and tool enabled novices to design, assemble, and program simple robots with minimal instructional assistance. However, our vision of end-to-end computational design for embedded devices is larger:

The tool should not presuppose such a narrow class of devices as the robots demonstrated in Chapter 2; the tool should be powerful enough to be of some use to experts; and the designs generated should be, ideally, Pareto-optimal for their design objective.

Echidna is an evolution of the Robot Parade tool that implements a heuristic search algorithm over a generalized design space. For the classes of devices tested (including more complex electronics, mechanical components, and software libraries), Echidna meets those goals, with some concessions given to computation time at the expense of optimality.

3.1 Computerized tools for design and engineering

Conventional computer-aided design (CAD) tools focus on the "how" of a design. That is, how will a device fulfill its functions? The answer to this question is a set of *technical constraints*. Users of CAD tools are expected to know how a device will work and specify it exactly. Conventional CAD tools manage, analyze, validate, and accelerate specifying these technical constraints.

On the other hand, "computational design" tools facilitate specifying and exploring the "what" of a design. That is, what will the device do? What will be the functions of the device? These are *functional constraints* to contrast the technical constraints of a design.

Some computational design tools, including Echidna, provide an automated implementation of designs. That is, the tools help the user to create an incomplete set of constraints – functional, technical, or both – and the tool outputs a more complete set of technical constraints. Ideally, the output constraints are complete enough that the device can be created without further user input.

For example, consider the functional specification given by the 3D model in the left of Figure 3.1. The designer wants to build a two-wheeled teapot-shaped device. The functional constraints are the shape of the robot and the placement of some functional components. The

specification details how the mechatronic device should function: light up and wheel around. It does not provide the information required for the assembly and control of the device.

The user wants the device on the right, a fabricated, functional artifact. A traditional CAD workflow would have an engineer (or several) choose and connect the internal components, specify the circuit board placement and routing, prepare the 3D printing files for a case of the correct shape, and code an API and firmware. With this complete specification, the parts of the device can be assembled, printed, and programmed to create the physical artifact.

Our computational design system does the work of that engineer by accepting, as input, the functional specification on the left of Figure 3.1 and outputting ready-to-go manufacturing and programming files.

We do not consider our system to be a computational fabrication system because the fabrication happens offsite, using conventional methods. However, many of the generated manufacturing files are loaded directly into computer-controlled machines (CNC machines) without further human intervention.

How our system relates to several computational design, fabrication, and implementation systems are discussed in Section 3.8.

3.2 Project Overview

Our system is an automated design system. It takes a high-level specification and outputs files that are ready to be sent to a manufacturer. If we consider the manufacturer to be part of the system (the files can be sent directly by the system), the output of our system is finished – or at least ready-to-be-programmed – devices.

This automated implementation system is modular. These are the main components: a specification interface, a high-level synthesis engine, and low-level domain-specific processors. These components can be swapped out, removed, or replaced to produce devices of different

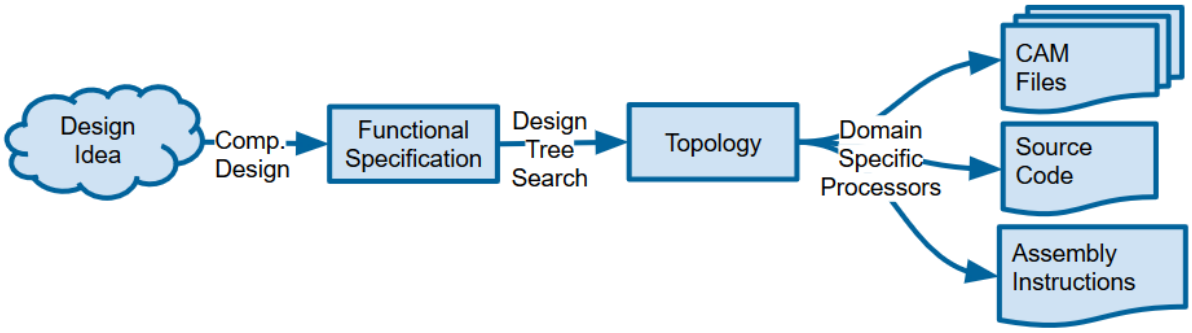

Figure 3.2: Computational design and implementation flow.
Using computational design, users create a specification from their design idea. Our automated system uses design tree search to generate a topology for the design. Then domain specific processors create a complete technical specification.

levels of completion from specifications with different levels of abstractness. This paper focuses on a specific embodiment of the system which takes a graphical device specification and outputs a complete electromechanical device implementation with example firmware.

Specification interface

The intent of this paper is to focus on the computational implementation side of our toolchain. However, we will briefly discuss two graphical constraint specification frontends to give context to the implementation backend.

We have two graphical interfaces that we have used with our system, one two-dimensional and one three-dimensional. Each graphical interface creates a functional specification with two types of constraints: geometric (shape) constraints, and functional component placements.

The two-dimensional interface specifies a circuit board shape and the location of components on that circuit board. This interface has some elements of a computational design system in that it gives guidance to the user with regards to invalid placement and the like. Figure 3.6 show some devices designed with this interface. This interface has been used with our system by first-quarter university students to create approximately 200 electromechanical devices over several years in an introductory electronics course. Please see our previously published experience

report for further details on this course and use of this GUI in an educational context [MS19].

The three-dimensional interface is similar. Except, instead of a 2D PCB shape, a 3D "shell" is specified and components are placed on and around it in three dimensions. Unlike the 2D GUI, the 3D interface has not yet been tested "in the wild." Figures 3.1, 3.4, and 3.5 show the 3D GUI.

The output of these interfaces is a design specification which includes a 2D or 3D geometric shape specification and a set of functional components (possibly with parameters, like color or speed) arranged in space.

High-level synthesis engine

This paper focuses on our system as a computational implementation system, not as computational design. So while the specification interface must be present, the main focus of this paper will be on the process of transforming the functional specification provided by the user interface into a set of technical constraints that are ready to manufacture.

The first step of this process is to iteratively satisfy the constraints required by each element of the user's functional specification. This is done by abstracting the functional elements as atomic components with interfaces. Each interface either must or may be connected to another type of interface. Finding a set of connections such that each required interface is connected in a valid way is the first (and most difficult) step of automated implementation.

Section 3.3 formalizes our approach to this problem and Section 3.4 describes our approach to solving this problem as search over decision trees.

Low-level domain-specific processors

After the high-level connections are made and the component list has been finalized, there are several different domain-specific processes that produce the final, low-level technical specifications of the design.

These include PCB place and route, API generation, and 3D printing file preparation. Our system uses either very simple, commercial, or open source solutions for these processes and they are discussed briefly in Section 3.5.4. Special consideration is given to the problem of API generation for arbitrary mixed-domain topologies in Section 3.5.5.

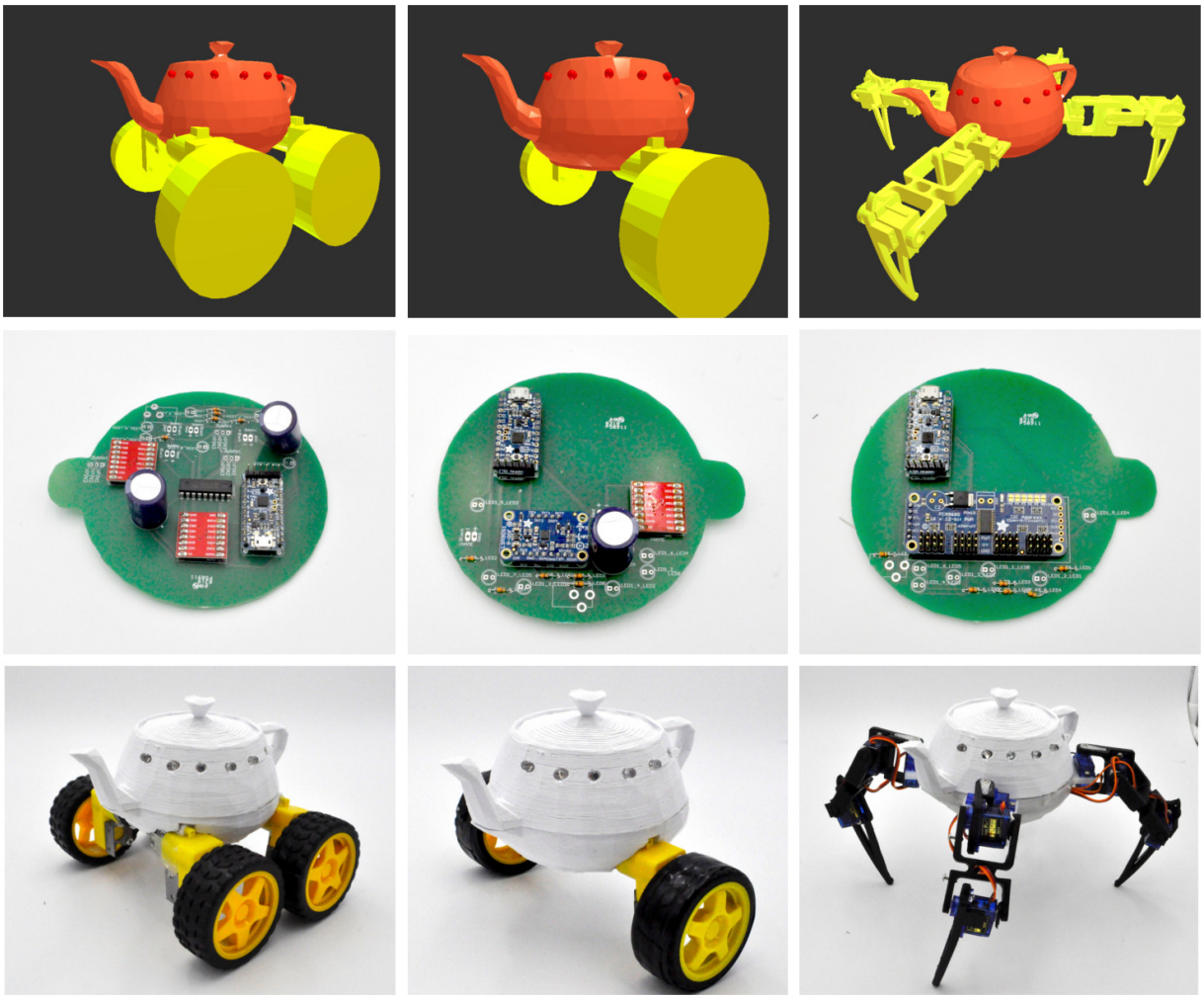

Figure 3.3: Synthesized robots.
These graphical initial design specifications (top row) are presented with their corresponding synthesized circuit boards (center row) and fabricated designs (bottom row).

3.3 Multi-Domain Design as Search

Echidna models computational design as a search over a tree of partial designs. Search starts at the root – a design that embodies the designers intent but lacks the internal parts and connections required to function.

Child nodes are designs that are the product of *transformations* – connecting interfaces of components that define a relationship between the components ranging from electrical connectivity to physical attachment. Echidna expands and searches the tree until it arrives at a completed realizable design.

The following section first describes Echidna's approach and then formalizes our notion of a design, the components it contains, the connections between those components, and the transformations that operate on them. We begin with an overview of how these parts fit together and an example to illustrate the formalism we will use.

3.3.1 Overview

Echidna designs are collections of components with connections between them. We consider mechanical components such as a leg, wheel, drive train, or button; electrical components such as microcontrollers, batteries, motor controllers ICs; and software components such as control APIs and software libraries.

The designer describes the device they wish to build by providing a list of functional components of the device and describing its physical shape. For instance, the functional components of an alarm clock would include the display, a speaker, several buttons, and shape for the clock.

Figure 3.4 shows Echidna's graphical user interface (GUI) for specifying a design. For simplicity, the interface assumes there is a single 3D "body" that represents the shape of the design. The designer can then attach components to the body. We refer to these end components as functional components.

After each desired functional component is connected to the body then extra components need to be added, such as a microcontroller or drivers. These extra components are the internal components that enable correct operation for each of the functional components added to the body.

Echidna specifies the relations between components as connections between component interfaces. The complete design requires connections between the interfaces of both functional components and internal components. For instance, the output interface of the power supply would connect to the power input interface of the microcontroller. Likewise, the display's "mounted-to" interface might connect to the "mounted-on" interface of the clock's case. The designer specifies some of these relationships using the GUI (such as the location of the display), but many are left for Echidna to create as it searches the space of designs.

We distinguish between interfaces that must be connected for the component to function (the *required interfaces*), and others. We refer to the optional interfaces as *provided interfaces* because they typically provide a capability that another component needs. The microcontroller's power input interface is required, but its general purpose input/output (GPIO) interfaces are provided.

Our search-based design system creates a complete design by starting with an incomplete design and then adding components and connections until no required interfaces remain unconnected.

3.3.2 Designs

Our algorithm operates on trees of *designs*. A design is a graph of components and the connections between the components. Each edge in the graph represents the connection of two components via specific interfaces. Two components may have more than one edge if the more than one pair of interfaces connects the components.

Our model considers several special types of designs:

Initial design specifications

An initial design specification is the entry point into an exploration of the design space. All other designs are derived from the initial design. Initial designs contain only user added components.

The range of possible initial design specifications defines the design space available to the user.

Incomplete designs

Incomplete designs have at least one component with an unconnected required interface. Echidna may further explore these partial designs by applying transformations. After a series of composed transformations, an incomplete design may become a complete design.

Complete designs

Complete designs do not contain any components with unconnected required interfaces. When a complete design is found, the search algorithm can terminate.

Dead-end designs

Dead-end designs are incomplete designs that Echidna cannot transform into a complete design through any allowed transformation. Dead-end designs root a sub-tree in the design tree that does not contain any complete designs.

If no transformations are possible on a design, then it is a leaf of the design tree and it is *trivially dead-end*.

3.3.3 Components

Components represent atomic units of a design.

For example, the Echidna library offers components such as buttons, lights, speakers, wheels, legs, knobs, sensors, microcontrollers, motor drivers, power supplies, resistors, capacitors, software libraries, and control APIs.

Some components add functionality to a design; a motorized wheel component adds mobility. Other components support the functional components; a motor driver component is necessary for the motorized wheel to function.

Users add functional components to an initial design specification to define their design intention. Echidna adds internal components to a design to support the functional components.

A component is represented by a set of internal parameters, a set of provided interfaces, and a set of required interfaces.

Echidna components contain mechanical, electronic, and software aspects. For example, a robotic end effector component incorporates a mechanical model of the physical structure, a kinematic model of the actuator, an electronic interface for the actuator, and a positioning API. These aspects of the component are exposed to Echidna as the interfaces and parameters.

3.3.4 Interfaces

Interfaces connect components. They represent the requirement or ability to provide electronic signals, mechanical attachment, software function arguments, and the other ways that the components interact. Echidna expresses interfaces by a set of parameters, a satisfaction predicate, and a component transformation.

The parameters of an interface holds information needed to determine compatibility with other interfaces. This information may include size and shape, voltage and current ranges, input/output direction, location, and other necessary information.

The satisfaction predicate is a function that evaluates to true when a valid connection is possible between the interface and another interface in the design. When the satisfaction predicate is true for two interfaces, the interfaces satisfy each other. For example, the satisfaction predicate

of a 5 V, 100 mA power input will evaluate to true for a 5 V, 150 mA power supply. But, it will evaluate to false for any 4 V power supply, any power supply with capacity less than 100 mA, or any interface that is not a power supply at all.

An interface's component transformation alters the component that contains it. Component transformations are triggered when an interface is connected. For instance, connecting a serial interface may mean that a GPIO interface is unavailable. Or, connecting a mechanical interface may change the needed actuation torque parameter of a motor.

Specifically, two things may change in a component through component transformation. First, the number and type of unconnected interfaces may change. Second, the parameters of the component may change.

We require that previously connected interfaces remain connected in the component, but any unconnected interfaces may be removed by the component transformation. In addition, the transformation may add new, unconnected interfaces to the component.

Hierarchical interfaces

We have found that composing interfaces provides a useful method for creating new interfaces for new components. For example, many interfaces (such as a full-duplex serial port) are made of a set of single wire digital I/O connections. We specify the higher-order interface (the serial port) as a collection of lower-order interfaces (the digital I/O wires). This can lead to the situation where the same low-order interface is used by more than one higher-order interface. If any of an interfaces sub-interfaces are unavailable for connection (perhaps due to previously being connected) then the whole higher-order interface is also unavailable.

3.3.5 Design transformations

A design transformation returns an output design which contains the same functional elements as the input design. The output design also contains exactly one more connection than

the input design and at most one more component.

With this type of transformation it may be possible to find a series of transformations that, when composed, transform an initial design specification into a complete design that retains all the function components of the initial design specification.

A design transformation is the application of two component transformations when making a connection. We allow one of the two components to be a new instance of a component from the library.

3.4 Design Tree Search Algorithm

Given an initial design comprised of functional components, Echidna's goal is to find a corresponding complete design. It achieves this starting at the root design (the initial design specification), and expanding the tree by adding components and making connections.

Below, we discuss the design tree in more detail, describe the basic search algorithm we use, and the heuristics that allow it to complete in a reasonable amount of time.

3.4.1 Design Tree

A *design tree* is a decision tree. Each node in the tree represents a (potentially incomplete) design, and the decision made at each node is how to transform the node's design to produce a new (child) design. Section 3.3.5 describes the two transformations we allow: connecting interfaces on two components or adding a new component and then connecting one of its interfaces to an existing component.

The leaves of the tree are designs that cannot be further transformed. There are two reasons this occurs. First, if the design's components have no unconnected, required interfaces; the design is complete. Second, if we cannot connect any interfaces between the design's components *and* adding an additional component from the library cannot change this property, then the design is a

dead end.

The large number of possible transformations make design trees extremely large. Consider a design with r required interfaces from its included components, p provided interfaces from its included components, and l provided interfaces from library components. This design may have $r(p+l)$ children in the worst case. As r and p are roughly proportional to the number of included components, $|C|$, and l is roughly proportional to the number of components in the library $|L|$, we can also say the number of children is on the order of $|C|^2 + |C||L|$. For our examples (Section 4.10), Echidna finds complete designs at depths around 400 that have up to 400 components. At that depth, the expected fanout leads to an effectively infinite number of designs.

3.4.2 Search algorithm constraints

The properties of design trees mean that many conventional tree search algorithms will not work. Depth-first search fails when it descends down a fruitless subtree (for example, one that includes an ill-advised transformation early on). The tree's high fanout makes enumerating even the second level of the tree intractable. This prevents naive breadth-first search.

Instead, an efficent computational implementation system needs an algorithm that meets two requirements. First, it should avoid the need to enumerate all the children of any design. Second, it must detect and avoid fruitless subtrees.

To achieve these goals, Echidna adopts a frontier-based, heuristic search algorithm. Echidna updates the value of the heuristic for each design as the search progresses.

The heuristic is a weighted sum that combines estimates of the design's cost, the expected difficulty in completing the design, and the estimated likelihood that the design is the root of a fruitless subtree.

We describe each of these components in detail below.

Cost

The cost factor models the designer's preference for cheaper, simpler designs. Echidna calculates design cost, cost(D), as the sum of the monetary costs of the components in the design. The cost function may also encode hand annotated preferences of the designer. For example, a lower cost for components that are spatially located close to each other; this would represent a preference for compact designs.

Completion

Our computational implementation system measures the difficulty to complete a design using the quantity and types of unconnected interfaces. The value accounts for required and provided interfaces differently.

Intuitively, more unconnected required interfaces means the design is far from completion. Therefore, designs with more unconnected required interfaces will be more difficult to complete. Furthermore, some required interfaces are more difficult to satisfy than others. For instance, an I2C bus connection is harder to satisfy than a general-purpose IO (GPIO) connection because the I2C bus must connect to particular pins on a microcontroller, while any pin will work for GPIO. In this case Echidna assigns a higher *complexity* to the I2C interface.

Unconnected provided interfaces, on the other hand, make a design easier to complete because they represent unused "features" of the design's components. The higher the complexity of the unconnected, provided interfaces, the more valuable they are.

To compute the net impact of unconnected interfaces, Echidna computes the sum of the complexity of for all unconnected interfaces of a design D as

$$\text{completion}(D) = \sum_{p \in P} \text{complexity}(p) - \sum_{r \in R} \text{complexity}(r) \tag{3.1}$$

where P is the set of provided interfaces of the components of D and R is the set of

required interface of the components of D.

The complexity of the interfaces is determined partially by humans. Lower-order interfaces are hand-annotated in the library based on the presumed rarity of compatible interfaces. The complexity of higher-order interfaces is automatically determined by summing the complexities of lower-order interfaces. Intuitively this makes sense, more complex (hierarchically) interfaces have higher complexity ratings.

In regards to electrical interfaces, we have seen that a default value of 1 for base interfaces works well, with a value of 2 for all other interfaces. Because most pins are dual function, multi-function pins are automatically assigned an appropriate complexity value due to hierarchical specification.

Dead-end heuristic

The final factor in Echidna's design evaluation estimates whether a design is in a fruitless sub-tree. The heuristic assumes that the more trivially dead-end descendents a node has, the more likely it is that the node is a dead-end node. Our estimate for a parent design, D, is updated each time a new design is found in the sub-tree with D at its root. To calculate the dead-end heuristic, we keep track of the proportion of trivially dead-end descendents of each node. The dead-end heuristic, dead(D), is the ratio of the number of trivially dead-end descendents of D to the total number of descendents of D visited so far.

Here, the value of dead(D) influences the exploration of the children of D which, in turn, influences dead(D). In practice, this has the effect of encouraging the exploration of designs without switching, until a dead-end child is found. Then, that design is avoided, until all other sibling designs have a similar number of explored dead-end children.

Combined design heuristic

The full selection heuristic is

$$h(D) = \alpha\text{cost}(D) - \beta\text{completion}(D) + \gamma\text{dead}(D) \tag{3.2}$$

where α, β and γ are weights for the sub-heuristics.

We find that these weights should be set such that $\gamma >> \beta >> \alpha$. This allows search to abandon bad sub-trees quickly. Ties are broken in favor of the deepest design (that is, the ones farther from the root) and then by cost.

3.4.3 Search algorithm

Echidna's search algorithm is a frontier-based search algorithm similar to A*. It maintains a frontier of candidate designs. Initially, the frontier contains only the root.

At each step, we choose the best design, D according to Equation 3.2, and try to connect an unsatisfied required interface on one of its components to an unused provided interface on another. When making this connection, we prefer to connect the required interface with higher complexity, and provided interfaces with lower complexity.

If we cannot make a connection among the existing components, we expand the set of candidate provided interfaces to include the interfaces to components in the library. If we find a component that provides an interface that can satisfy another interface in the design, we add the component and make the connection. If several components provide the interface, we choose the cheapest one. We then add the new design to the frontier.

If the new design is complete, we return that design.

If no connections can be made, D is a dead end, and we remove it from the frontier and update the count of dead descendents for each of its ancestors.

If the frontier is not empty, we select the new best design, and repeat the process.

3.4.4 Evaluation

We do not claim that this algorithm is necessarily optimal. However, using heuristics, it is able to quickly and automatically create topologies for a variety of the class of devices we are interested in. Also topologies are created in a way that guarantees that they are correct by construction (assuming a bug free library). These properties make decision-tree search a good choice for our computational implementation system.

Our experiences have indicated that this algorithm generated reasonable topologies; the topologies generated by this search algorithm are very similar to those generated by a human. For example, the three devices shown in Figure 3.6 have the same topologies as those generated by a human, given the same constraints (and allowing for variation from swapping equivalent pins). Further evaluation of algorithm performance is given in Section 4.10.

3.5 Implementation

We implemented design tree search as the back end of our full system that allows novice users to create a variety of mechanical and electronic devices without design experience in these domains.

3.5.1 System architecture

Our system is composed of several, quite different, parts.

In production, our web-based GUI runs on Google's App Engine and is written in Python and Typescript.

While the GUI is in use, geometric information passed to servers to check overlap and other quick-to-computer feedback.

When a use submits a design for implementation, the specification (geometric constraints and component locations) are pushed to a job queue. A bank of servers poll this queue, waiting to

start the more computationally intensive task of implementation. These servers require several minutes of setup time so must be aggressively autoscaled if demand is expected..

When a job is pulled, design tree search (implemented in Python) is used to finalize the component list and the connections between the components. This creates the device topology. After design tree search, the device topology is handed off to file converters that reformat the topology into forms which can be fed through the diverse set of domain specific processors used. These include PCB files (EAGLE sch and brd files), Jinja template files for source code compilation, svg graphics and html files for assembly instructions, and other. The glue code that ties these processors together is implemented in Python.

The domain specific processors then work to create the final technical specification of the device. The most computationally intensive of these processes is PCB place and route. Once all the final technical files are created, they are compressed and uploaded to a cloud database and a link is given to the user to download the implementation. At each stage of the process the user is updated on the current stage of implementation. If a stage fails, that stage can send a human-readable error message back to the user interface.

3.5.2 Graphical user interface

Our system begins with a high-level graphical specification created through a drag-and-drop interface.

Three-dimensional devices

Users begin with a basic three dimensional model that serves as the body of the design. From a pallet, users add functional components such as lights, motors, legs, buzzers, sensors, and so on. Some components must be directly attached to the exterior of the body. Others may be placed in space around the body. Still others, such as internal sensors, are not spatially placed by the user but are added by selection only.

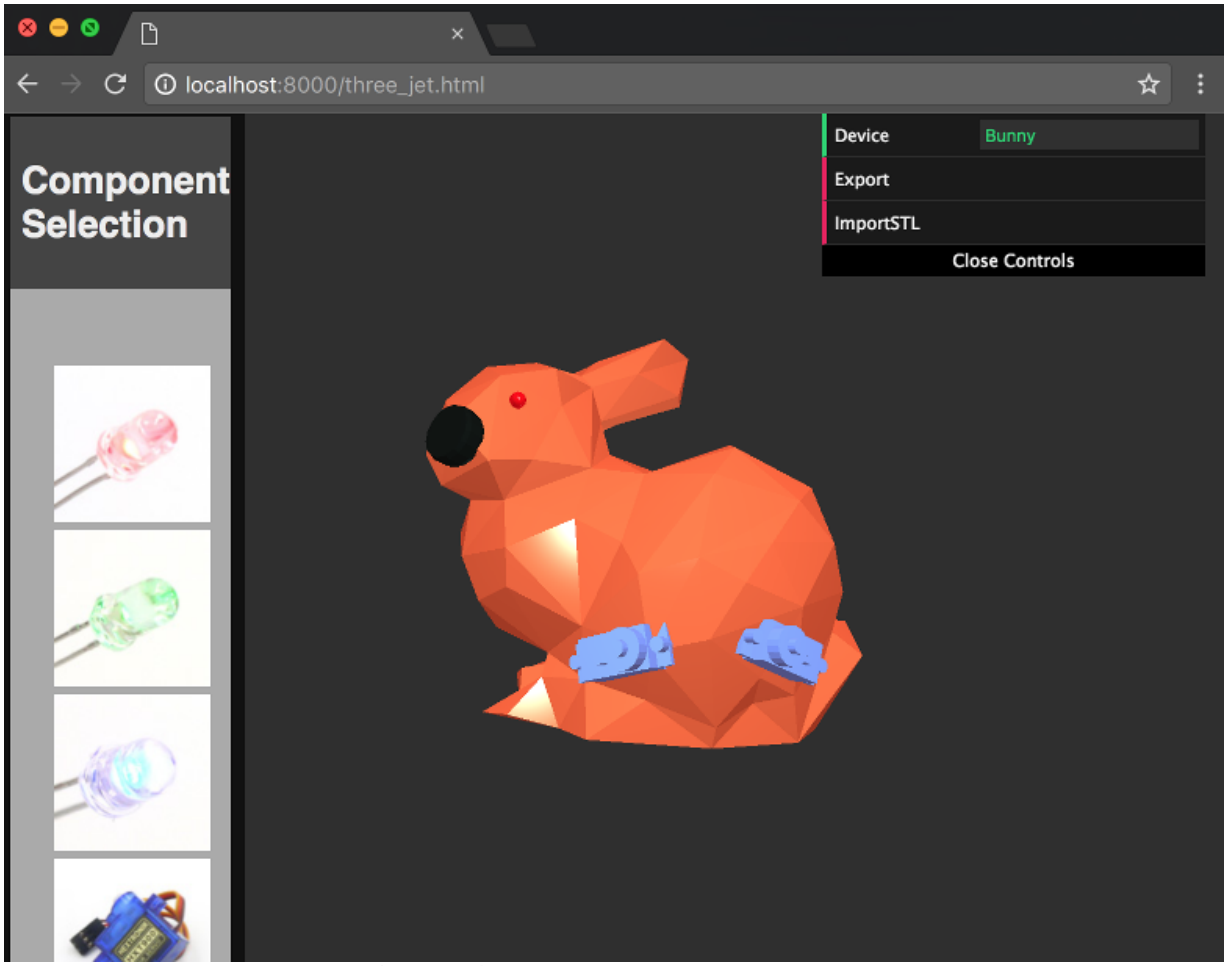

Figure 3.4: GUI for specifying 3D devices.

The user also selects a cross section of the body that forms the location and shape of the internal circuit board.

The graphical user interface can also act in a circuit board only mode which enables the user to graphically place functional components directly on a circuit board to create devices (Figure 3.6).

Two-dimensional devices

Similarly to the three-dimensional version, users drag and drop components from a library onto their device. However, instead of a cross-section of a 3D form providing the PCB shape, the PCB geometry is specified directly. Designs created with this GUI do not require a 3D printed

component. The devices shown in Figure 3.6 were designed with the 2D GUI. This version of the GUI has been used by novices in a classroom setting to create simple electronic devices and robots [MS19]. The 2D GUI and computational implementation system as used in the course is available at `http://robots.gadgetron.build`.

3.5.3 Component library

We use a component library with over 300 different components. These components include mechanical components such as robotic legs, sections of robotic arms, motorized wheels, and buttons; electrical components such as lights, microcontrollers, and sensors; and software components such as control APIs.

To use these components in designs, our system needs a variety of information about each component. This information typically includes a subset of the following: 3D models, mechanical drawing, electrical schematics, source code snippets, weight, assembly instructions, and price information. In addition, each component entry contains detailed interface information that implements the interface objects defined in Section 3.3.4.

Limitations our of library are discussed in Section 3.7.

3.5.4 Domain specific implementation

Once Echidna creates a multi-domain design topology, the topology is processed by several domain specific procedures. These procedures involve completing templates with information from the design topology, collating and converting files, and processing with commercial software.

3D printed components

Echidna modifies the 3D printable shell with mounting holes and brackets that are specific to the type and location of each components. Echidna creates mounting holes by removing

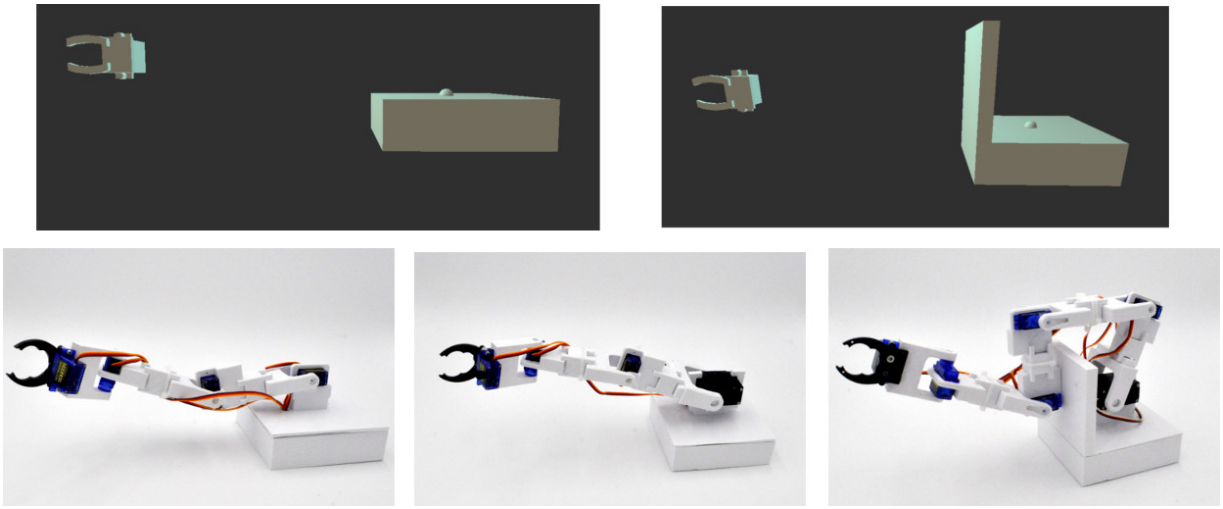

Figure 3.5: Synthesized robotic arms.
Initial design specification for base configuration and heavy-lifting configuration (top left), initial design specification for obstacle configuration (top right), fabricate base configuration (bottom left), fabricate heavy configuration (bottom center), and fabricate obstacle configuration (bottom right).

material in the shape of the component. For some components, especially actuators, Echidna adds material to act as mounting substrate. 3D printing files are prepared and modified using OpenSCAD and Ultimaker Cura, both open source software.

Printed circuit boards

Echidna creates a circuit schematic for the electronic components via the design topology. From the schematic, Echidna creates a board layout using a simple packing algorithm (greedy linear search). The outline of the circuit board is the shape of the internal wall of the shell at the user-selected cross-section (less any interfering mounted components). The electrical connections on the circuit board are routed using Autodesk EAGLE, a commercial autorouter. If placement or routing fails, a warning is displayed to the user. This warning makes the suggestion to increase the empty space on the board by increasing the size of the device. We have found that very few (¡3%) of student created design are not able to be automatically placed and routed [MS19]. These failures could be address by integration with superior autorouters, but EAGLE was chosen due to

its ubiquity and open file format.

Source code

Echidna generates firmware source code through parameterized templates. Each controllable pin of the microcontroller or single-board computer is bound to a variable in a template. These variables then parameterize software object constructors based on information from the design topology. These objects form a user-friendly API for writing custom programs. Echidna also creates an example program which cycles through the basic functions of each component present in the design topology. The example program may be used as a basic test of the device or as a starting point for a custom program.

3.5.5 Generating APIs for complex topologies

Echidna generates control APIs for components that require the coordination of several other components. Some component interfaces specify API calls and their parameters. During synthesis, Echidna tracks these parameter symbolically. These components also specify how the API functions they require relate to the API functions they provide, defining a transfer function.

For example, a microcontroller's transfer function may take a `pin_1(voltage=5)` constraint and transform it into a `digitalWrite(pin_1, HIGH)` API call. This transform may seem trivial, but these simple transfer function can be automatically combined to create API for arbitrary device topologies.

When Echidna connects a component, Echidna parameterizes that component's transfer function with the constraints of the previous components. This creates a new system of (more complex) constraints that Echidna propagates further.

If the final constraint system is purely geometric, specialized inverse kinematic solvers are optimized for the problem. Mixed geometric and logical constraint systems can be solved by general constraint satisfaction tools such as dReal [GKC13] to provide API implementations.

In Echidna, we have successfully used both the symbolic computation package SymPy [MSP+17] and dReal for API implementations and connection type checking. Echidna stores both variables and composed constraints symbolically in the components' parameters (Section 3.3.3). The component transformations can then access these symbols during design topology synthesis.

The key feature of our approach is that it is domain and topology agnostic. While, for example, digital control and kinematic control are well studied, and systematic methods have been developed to solve problems in each domain, our system of composing constraints allows each domain to be solved in the same process, with arbitrary topologies, without human intervention. This obviates the need to code specialized control and solvers for each new component and domain.

Kinematic control example

As an example, a robot arm's end effector (Figure 3.5), could require a `move_to(x,y)` API where `x` and `y` are absolute coordinates. If the end effector is attached to a series of arm segments, then each segment adds mechanical constraints to the inverse kinematic system that the movement API must solve.

Again, Echidna generates the API by composing the transfer functions of each component.

A single segment's transfer function relates the inputs, x^{start}, y^{start}, and r to the outputs x, y by the equation:

$$(x,y) \in \{(a,b) | a = x^{start} + l \times \sin(r), b = y^{start} + l \times \cos(r)\} \tag{3.3}$$

where r is the segment actuator's rotation, l is the segments length, and (x^{start}, y^{start}) is the position of the segment's start.

For a chain of segments, let $(x_k^{start}, y_k^{start})$ be the location of the start of a segment k and (x_k^{end}, y_k^{end}) be the location of the end of segment k, and r_k be the actuator rotation for segment k

and l_k be the length of segment k.

The constraints in an n-segment chain are described by this non-linear system (equations 3.4 through 3.6).

$$(\mathtt{x},\mathtt{y}) = (x_1^{end}, y_1^{end}) \tag{3.4}$$

$$\bigwedge_{i=2}^{n} (x_{i-1}^{start}, y_{i-1}^{start}) = (x_i^{end}, y_i^{end}) \tag{3.5}$$

$$\bigwedge_{i=1}^{n} \Big[(x_i^{end}, y_i^{end}) \in \{(a,b) | a = {x_i}^{start} + l_i \times \sin(r_i), b = y_i^{start} + l_i \times \cos(r_i)\}\Big] \tag{3.6}$$

Here the transfer function of each mechanical component transforms the propagated constraints in two ways. First, it adds a spatial constraint to the starting end of the attached component (Equation 3.5). It also appends the trigonometric constraints to the system with a conjunction (Equation 3.6).

When it comes time to make an API call, `move(x,y)`'s parameters are set and the rest of the free variables can be solved for via inverse kinematics (or an error is raised if the system cannot be solved).

Digital control example

As another example, an LED component may require a `blink_once()` API. If the LED is connected to the microcontroller (MCU) through an IO expanders, the LED cannot be manipulated directly by the microcontroller. Echidna generates this API implementation by composing the functions that relate the inputs and outputs of the microcontroller, the IO expander, and the signal inputs of the LED.

This example follows the generation of the `blink_once()` API during synthesis as an ordered list of constraints.

The value of the symbolic constraints in each partial design are formatted as:

```
1: component.interface(key=value, ...)
```

In the initial design specification, the LED's API constraint is unmodified. The initial ordered list of symbolic constraints is:

```
1: LED.anode(state=DIGITAL_HIGH)
2: generic.API(method=delay, parameter=onPeriod)
3: LED.anode(state=DIGITAL_LOW)
4: generic.API(method=delay, parameter=offPeriod)
```

Echidna adds a GPIO expander and connects the expander's `out_1` interface to the LED's `anode` interface:

```
1: expander.out_1(state=DIGITAL_HIGH)
2: generic.API(method=delay parameters=onPeriod)
3: expander.out_1(state=DIGITAL_LOW)
4: generic.API(method=delay parameters=offPeriod)
```

The IO expander's transfer function determines how to expand constraints 1 and 4, where `0x7E` is `LED_IC`'s I2C address, `0x01 0xFF` is the command to bring `out_1` high, and `0x01 0x00` is the command to bring `out_1` low:

```
1:  expander.I2C_data(type="addr", data=0x7E)
2:  expander.I2C_data(type="data", data=0x01)
3:  expander.I2C_data(type="data", data=0xFF)
4:  expander.I2C_data(type="end")
5:  generic.API(method=delay, parameter=onPeriod)
6:  expander.I2C_data(type="addr", data=0x7E)
7:  expander.I2C_data(type="data", data=0x01)
8:  expander.I2C_data(type="data", data=0x00)
9:  expander.I2C_data(type="end")
10: generic.API(method=delay, parameter=offPeriod)
```

Echidna adds a microcontroller (MCU) and connects the I2C bus interface to the expander.

```
1:  MCU.I2C_data(type="addr", data=0x7E)
```

```
2:  MCU.I2C_data(type="data", data=0x01)
3:  MCU.I2C_data(type="data", data=0xFF)
4:  MCU.I2C_data(type="end")
5:  generic.API(method=delay, parameter=onPeriod)
6:  MCU.I2C_data(type="addr", data=0x7E)
7:  MCU.I2C_data(type="data", data=0x01)
8:  MCU.I2C_data(type="data", data=0x00)
9:  MCU.I2C_data(type="end")
10: generic.API(method=delay, parameter=offPeriod)
```

The MCU's transfer function transforms the symbolic constraints to call to the MCU's API interface:

```
1:  MCU.API(method=beginTransmissionI2C, parameter=0x7E)
2:  MCU.API(method=writeI2C, parameter=0x01)
3:  MCU.API(method=writeI2C, parameter=0xFF)
4:  MCU.API(method=endTransmissionI2C()
5:  MCU.API(method=delay, parameter=onPeriod)
6:  MCU.API(method=beginTransmissionI2C, parameter=0x7E)
7:  MCU.API(method=writeI2C, parameter=0x01)
8:  MCU.API(method=writeI2C, parameter=0x00)
9:  MCU.API(method=endTransmissionI2C)
10: MCU.API(method=delay, parameter=offPeriod)
```

The API constraints are transformed into source code by a parameterized template:

```
blinkOnce(onPeriod, offPeriod) {
    beginTransmissionI2C(0x7E);
    writeI2C(0x01);
    writeI2C(0xFF);
    endTransmissionI2C();
    delay(onPeriod);
    beginTransmissionI2C(0x7E);
    writeI2C(0x01);
```

```
    writeI2C(0x00);
    endTransmissionI2C();
    delay(offPeriod);
}
```

3.6 Results

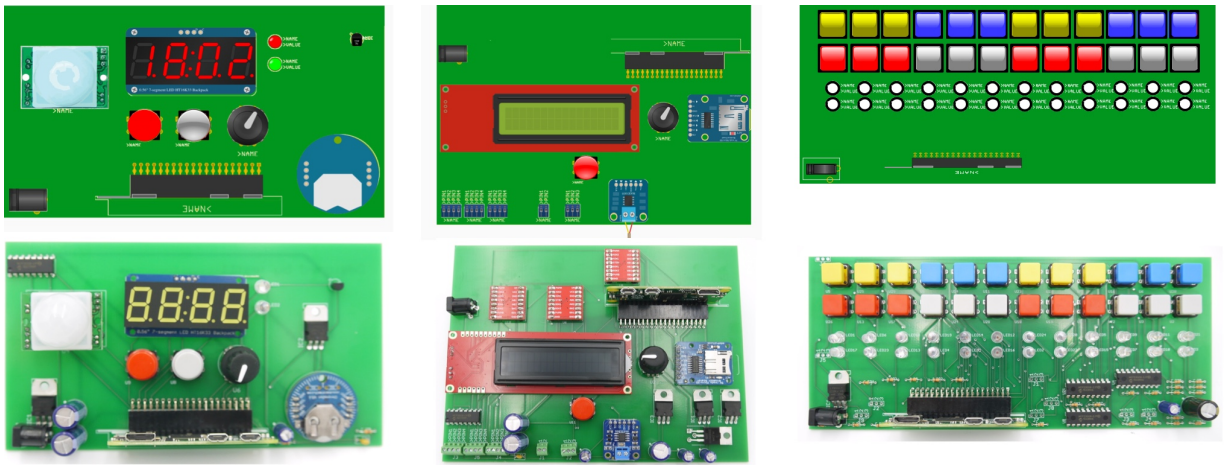

Figure 3.6: Synthesized circuit boards.
These fabricated circuit boards (bottom) are paired with how they appear in the 2D version of our GUI (top). They include a thermostat (left), a 3D printer control board (center), and a 2x12 music sequencer (right). Sequencers 4x12 and 8x12 not shown.

To demonstrate our computational implementation system we have used it to design and fabricate several devices: a small teapot-shaped robot to demonstrate how changes to the mechanical design components can affect the electronic and software internal components; a robotic arm to demonstrate how design tree search can find solutions to mechanical constraints; finally, a selection of embedded device control boards to demonstrate how design tree search scales to hundreds of components. We implemented Echidna's search algorithm in Python.

3.6.1 Teapot robots

We have designed three configurations of a self-contained, motive robot (Figure 3.3). Each configuration has a different method of motion specified in its initial design specification. The initial specification also includes six LED lights for each configuration.

These robots have in common several internal components added through design tree search. These are a microcontroller unit, a battery, and six current limiting resistors for the LEDs. In addition, Echidna adds several other internal components to each configuration, including control APIs.

Wheeled configurations

Two of the configurations use motorized wheels. For each pair of wheels, Echidna adds a motor driver board and a large capacitor.

The configuration with four wheels is short on GPIO pins from the microcontroller. In this configuration, Echidna automatically adds and configures an IO expander component.

The configuration with two wheels does not need an IO expander. However, a design with only two wheels cannot satisfy the four wheel control API. Instead, Echidna adds a self-balancing two-wheel API. The self-balancing API component has a tilt sensing required interface, so Echidna adds an IMU.

Legged configuration

This configuration does not require motor driver boards. Instead, Echidna selects a servo control board and gait control API to satisfy the leg components.

3.6.2 Robotic arm

This example of a robotic arm and gripper demonstrates how geometric and mechanical constraints can be satisfied by design tree search (Figure 3.5). In each of the three configuration, a base and gripper are added as design components. The gripper is not mounted to the base, and our system must complete the robotic arm. The synthesized mechanical components differ in each configuration, but the synthesized internal electronics are similar to the legged teapot robot.

Base configuration

In this configuration, design tree search first finds the required interface on the gripper and matches a robotic arm segment. This new segment introduces another similar required interface. The final link (to the base) is completed by one last segment.

Heavy lifting configuration

In this configuration, we increased the weight of the end effector as a parameter of the component specification. When a new arm segment is added, the segments weight adds to the arm's total weight. This combined weight is propagated to the required interface of each new segment.

When we increased the weight of the end effector, Echidna selects a more powerful motor for the final end segment to lift the increased propagated weight.

Designed around obstacles

In this configuration, we modify the base such that the direct path from the end effector to the attachment point on the base is blocked. Here, Echidna designs the arm around the obstacle. The weight of the additional arm segments causes the last segment to again use a more powerful motor.

3.6.3 Embedded circuit boards

When run in circuit-board-only mode, our system allows users to place design components directly on a circuit board. This is useful to demonstrate our system's scaling properties. Using this mode, we designed five devices. These are: a thermostat, a 3D printer control board, and three configurations of a musical sequencer board (Figure 3.6). The three configurations of the music sequencer use different numbers of rows of 12 buttons and lights.

The thermostat, printer controller, and two row sequencers were fabricated and appear in Figure 3.6 along with how they appear in the 2D version on our GUI.

Table 3.1 lists the design characteristics of the five designs. We obtained these synthesis results on a system with a 2.7 GHz Intel Core i5 processor. As the number of design elements increases, so does the required synthesis time.

Thermostat

The thermostat's user added components include a real-time clock, an occupancy sensor, and a temperature sensor. Echidna adds and connects several power regulation components, signal filtering components, current limiting resistors, and an analog to digital converter (ADC). In this design, Echidna's heuristics select the correct path through the design tree without backtracking and completes in less than one second. Here, synthesis time is dominated by start up overhead.

Table 3.1: Topology synthesis characteristics of example circuits.

	Thermo-stat	3D Printer	Sequencer 2x12	Sequencer 4x12	Sequencer 8x12
Design elements	12	18	50	98	194
Internal elements	9	19	39	99	189
Connections	50	101	97	200	406
Search time	**0.3s**	**0.9s**	**2.9s**	**8.9s**	**43.7s**

3D printer control board

The 3D printer control board's user added components include several motor control hookups, a thermocouple hookup, several high-current control hookups, and an SD card reader. Echidna adds and connects several motor drivers, high current transistors, power regulation and filtering components, current limiting resistors, and an ADC.

Sequencers

The sequencers include a large number of input components (buttons) and output components (LEDs). These necessitate the addition of several digital IO expander ICs.

As the number of buttons and lights increases, more IO expanders are required. This necessitates exploring and discarding many dead-end subtrees, as our search algorithm prefers designs with fewer components. The smaller designs complete in several seconds. For the largest design with 383 components, the synthesis time is 44 seconds.

3.7 Limitations of the Library

To develop, use, and evaluate our tool, we created a library of components and interfaces. As we have described, this library includes electronic, mechanical, and software components and interfaces. Our experiences with the tool have shown that our component library is sufficient to create interesting devices. Still, there are many limitations to the current library. In the current system, the library is the primary limiter of the system's usefulness.

For example, the teapot robot: if a user were to place four wheels in a row, then the current system would not know to add a balancing system to the robot. The four-wheel API and control system (as currently implemented) does not require this constraint. We believe that such a constraint *could* be implemented using our interface system; the robotic arm example shows that the implementation system is able to handle moderately complex geometric constrains.

As an other example, the currently the user cannot specify degrees of freedom of the robotic arm. If a suitable component existed in the library (it does not), then the implementation system might connect the pincer directly to the base using a component with zero degrees of freedom (a solid beam). We believe that a degree-of-freedom constraint *could* be added using our interface system; such a constraint is very similar to those used by electrical components that require specific ranges of voltage and current.

There are a large number of such example. We believe that we have demonstrated that our computational implementation system reasonably general and fast for our use cases (small embedded devices). However, our tree-based implementation algorithm (specifically the heuristics) were developed and tested using our library. If the library were expanded, extended to other domains, or if the components were made to be significantly more complex, then further heuristics and connection algorithms may need to be develop.

Anecdotally, we have found that library creation, maintenance, and debugging quickly require the largest portion of effort as we add capabilities to Echidna and other computational implementation systems that we have developed. We feel that this issue poses the largest barrier to the creation and adoption of similar computational implementation tools, especially in an academic context.

3.8 Related Work

Our approach to computational design and implementation spans computational geometry, robotics, electronic design automation (EDA), and programming languages. In the following section we detail our contribution to computational design and fabrication by comparing it to related works.

Library-based synthesis

Component-based synthesis is the construction of a network of elementary components, given a library of components such that the constructed network behaves in the intended way. The set of components chosen, and the connections between them constitute a *topology*.

The problem of synthesis from a component library has been studied in the context of software [LV09, SLTB+06, PR90] in particular, with research emerging in the mixed hardware-software domains [IoTSV17, RLI+17b]. The complexity of the components in the library can make it difficult to reuse the computational techniques provided by a design tool. For example, [SSL+14] system in *Design and Fabrication by Example* requires CAD models that are detailed down to the screw. Our algorithm uses a library of individual components that are described using Python.

Design for Fabrication

Computational design and implementation systems exist in many domains. Given a design, they fill in the gaps that stand between the specification and an artifact. Implementation is often framed as a task-specific problem and as a result, many systems in this genre target a narrow domain *e.g.* plush toys [BCC17], furniture [LHAZ15, SFJ+17], clothing [SYN+15], mechanical objects [BCT+15, ZAC+17], or trigger-action circuits [AGF17b].

Echidna brings a domain-agnostic approach to implementation that enables the creation of complex design topologies from components that span the mechanical, electrical, and software domains.

Embedded electronics domain

EDASolver [eda14] is a tool that connects complex components and interfaces, but cannot infer when additional components are needed to satisfy the requirements of a system. Echidna is

able to add many components if required to complete a design.

Just in Time Printed Circuit Board (JITPCB) [BBB+16] synthesizes high-level specifications into circuit netlists. Like our system, the JITPCB uses EAGLE for PCB routing. Integral to JITPCB is number of slave microcontroller that make up a required chain-of-stars topology. Our system instead allows any topology that can be generated by connecting interfaces.

Embedded Design Generation (EDG) [RLI+17a] tackles synthesizing small embedded circuits. EDG solves for the components needed in a circuit from constraint descriptions. Solving the constraints for even small designs can take hours or days. EDG uses the Z3 [DMB08] constraint solver. Our system, using design tree search, significantly out performs EDG without requiring library pruning for acceptable runtimes.

Trigger-Action Circuits (TAC) [AGF17b] synthesizes both small circuits and firmware with an eye towards novice prototyping. This work enables electronics novices to specify an ”if this then that” circuit that senses a stimulus from the environment and executes an action. For example, ”if the temperature is above 72 degrees, then turn on the fan”. The system lets the user specify just the functionality using a graphical programming interface and then synthesizes an implementation along with instructions. TAC is important in that they incorporate behavior into the computational design process; this work does not address many computational implementation questions. We see TAC as a possible design frontend for Echidna's fast implementation backend. TAC uses breadth-first search which does not scale to large designs and libraries.

Mixed-Domain Computational Design

Recent computational design and implementation projects focus on creating devices which incorporate mechanical, electronic, and software elements. Some of these focus on a narrow class of devices such as walking robots [MTN+15, Cor13] or multicopters [DSZ+16].

These systems maintain disparate topologies for the mechanical, electrical and software components of a design. There are strong constraints in the mechanical domain, no topology for

the electrical components, and a fixed software topology. In contrast, Echidna generates multiple- and mixed-domain topologies from nil.

[CSF13] propose a method for inferring energy transfer components in physical modeling languages. This technique is able to infer abstract design topologies (such as an abstract pipe to transfer hot air or a wire to transfer current). It focuses on generating comprehensible descriptions, not fabricatable implementations. In contrast, Echidna generates concrete topologies and fabricatable implementations from a much larger library of components.

3.9 Acknowledgement

Chapter 3, in part, is a reprint of the material as it appears Proceedings of the ACM Symposium on Computational Fabrication 2019. Merrill, Devon; Garza, Jorge; Swanson, Steven, Association for Computing Machinery, 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 4

OpenROAD PCB:

A Framework for PCB Layout Automation

We have shown that computational design and implementation can efficiently create interesting embedded devices. Also, in Chapter 3, we identified library creation and management to be significant, if not primary, difficulties in the development of computational design tools for embedded devices. In an academic context, library development does not necessarily pose an interesting research problem. It only requires a large amount of engineer-hours for creation and debugging. Conventional software engineering techniques can likely rise to this challenge. In the domain of PCBs specifically, the unavailability of competent layout tools also bottlenecks computational design tools.

The wide gap between PCB layout research and commercial practice limits the usefulness of our community's research contributions to industry. PCB layout researchers can emulate the steps taken by the VLSI community to solve this problem: specifically, open benchmarks and open and automated workflows. VLSI workflows are available that can take a design from a netlist to layout with minimal human interaction. These workflows enable the evaluation of a contribution in context, for example, in evaluating whether a change in layout methodology improves routability.

They also facilitate research contributions, for example, allowing experimentation with routing heuristics without coding a router from scratch.

We have implemented and are releasing a PCB layout framework, which has the unique combination of being fully-automated and open-source. Ours is the first PCB layout framework that can provide the same benefits to PCB layout research as existing VLSI workflows provide in that domain. We also propose a method for objectively evaluating PCB layouts: a set of benchmarks, a proposal for adopting a standard open benchmark format, and a list of evaluation metrics. The PCB benchmarks are based on real designs, ranging from 19 to 57 components and from 15 to 99 nets. We offer an initial evaluation of our framework using our proposed evaluation method to show that our framework produces layouts comparable to manual layouts. We believe that these contributions will help close the gap between PCB layout research and commercial practice, supporting the production and evaluation of industry-relevant PCB layout research.

4.1 Introduction

Automated printed circuit board (PCB) layout has been of interest to the design automation community for decades [FI65, Gey71, Sou78]. Despite such interest, this area of research has fallen far behind current commercial tools. Prima facie, PCB layout is tantalizingly automatable due to its easy-to-grasp problem domain of placing and connecting points in two dimensions, and its apparent similarity to several classic computer science problems, including packing and pathfinding. In practice, complex design rules and special cases that are often never formalized lead to a quagmire of complications that remain obscure to those without extended, practical PCB layout experience. A 2010 survey by Yan and Wong [YW10] showed that a "huge" amount of research has been devoted to the PCB routing problem. The significant projects addressed in Yan and Wong's survey operate in simplified problem regimes, however, and it is unclear whether they could be integrated into a real PCB layout flow.

For example, many of the problem formulations assume single-layer designs. This is never the case for the high-end designs that might make use of the routing techniques surveyed. The vastly reduced complexity of single-layer routing compared to multi-layer routing makes it unclear if the techniques can ever be applied to real layout problems. The simplification also hides many other nuances, such as the large vias that typify the PCB routing problem. In essence, there is a "reality gap" between current academic work and the practice of designing PCBs.

If a practical, fully-automated PCB layout flow can be achieved, the benefits would be numerous and far-ranging, including the potential for new embedded-system design interfaces, new pedagogical methodologies, and vastly reduced prototype turnaround time.

Our discussions with PCB layout engineers while preparing this work indicate some distrust towards automation techniques in their field. Often cited is a belief that automated tools cannot produce manufacturable design and that a human will inevitably need to rework the automated tools' outputs. Robust, realistic research is needed to overcome this distrust.

For PCB research to be robust, open-source research tools are needed that can produce manufacturable PCB layout. In addition to the availability of the source code, a standard set of metrics and benchmarks must be agreed upon to allow reproducibility and objective measurement of results.

We summarize the key contributions of our work as:

1. The implementation of an open-source framework for PCB layout automation, including the release of our open-source placer, router, and design database.

2. A set of metrics useful for evaluating the quality of a low-speed PCB layout.

3. Eleven real PCB designs for evaluating PCB layout automation, each with manual layout for comparison.

4. The evaluation of our placer and router with an existing open-source router to demonstrate an end-to-end flow.

The remainder of this paper is organized as follows: Section 4.2 relates this work to other work in the field; Section 4.3 compares PCB layout to the similar field of VLSI layout; Section 4.4 details our open-source PCB layout framework; Sections 4.5 and 4.6 briefly discuss the PCB-relevant aspects of our layout algorithms; Section 4.7 discusses metrics for evaluating PCB layout; Section 4.8 discusses the preparation and technical details of the benchmark designs we are releasing; Section 4.9 explains our choice of the KiCad PCB layout format as a standard format for PCB benchmarks; Section 4.10 presents our evaluation of automated and manual layout for the benchmark designs; Section 4.11 proposes future directions for our framework and PCB benchmarks.

4.2 Relation to Previous Work

The dearth of fully-automated PCB layout tools limits computational design research for embedded systems. Tools like Trigger-Action-Circuits [AGF17a], EDG Methodology [RLI+17c], TinyLink [GDG+17] and Geppetto [gum] provide novel ways of generating circuit netlists from high-level descriptions, with the goal of enabling the creation of embedded devices by non-experts. Without a fully-automated PCB layout flow, however, these processes cannot fully realize their goal. Geppetto, for example, must offload track routing to human engineers "behind the scenes," increasing both cost and turnaround time.

For researchers who *have* cobbled together fully-automated PCB flows, the restrictions – availability of appropriate licences, lack of exposed tuning parameters, difficulties with automating and interfacing, no ability to extend capabilities– of closed-source commercial tools limit the complexity of designs that are possible. The tools Echidna [MGS19] and JITPCB [RLI+17c] use Autodesk's EAGLE autorouter to create a fully-automated layout flow. Those researchers chose EAGLE because EAGLE's plain-text file format creates the possibility of an automated interface. The proprietary format restricts extension through a license agreement, however, and because

EAGLE is closed-source, the router cannot be modified or extended.

The OpenROAD [ANP+19] project has the ambitious goal of creating a no-human-in-the-loop VLSI flow with a target of $500 for a one-off IC prototype. If the project is successful, the design of the host PCB and package will account for the majority of the remaining hardware costs.

The creation of viable PCBs at "the push of a button," while a major task, would enable a plethora of research and commercial projects. Our work in developing benchmarks and a framework for fully-automated PCB layout is a fundamental step towards achieving that goal.

4.3 Comparison to VLSI Layout

The previous work on practically applied layout algorithms for PCBs may be sparse, but similar flows exist for VLSI. We have identified key differences in both placement and routing that prevent the direct application of VLSI tools for PCB layout.

Differences between PCB and VLSI placement

As an initial experiment, we applied the state of the art analytical IC placers [LCC+15, CKKW18] to the PCB placement problem and identified several issues that prevented convergence to good solutions.

1. The vanilla RePlace implementation fails to successfully find good rotations for individual components.

2. The smoothness of the cost and convergence of the solution is significantly dependent on design-dependent parameters, such as filler cells and boundary anchor weights.

3. RePlace is natively incapable of dealing with non-rectilinear geometries.

The failure of RePlace to find acceptable placement for PCB layouts led us to identify these key difference between PCB and VLSI placement:

1. Scale: PCB designs contain orders of magnitude fewer components than VLSI. The number of components in a PCB placement problem range from dozens in small designs to thousands in very large designs, whereas VLSI designs may contain billions of components.

2. Shape of components: PCB components may have complex polygonal shapes, as opposed to the rectilinear cells of IC design. The shape of any single component is a set of arbitrary polygons that may be spatially disconnected or non-convex.

3. Two-sided and 3D placement: PCB components may be placed on either the top of the bottom of the PCB. When a component is placed on the bottom of the PCB, the component geometry (outline, pin location) is mirrored. In some cases, components may overlap. For example, small components may be placed underneath the overhangs created by daughter-cards. While some PCB tools are able to encode 3D component geometry, we are unaware of any PCB layout tool that assists with 3D component placement.

4. Rotation of components: PCB components may be freely rotated during placement, and 45-degree rotations are not uncommon in PCB layouts. Human designers generally resist creating designs with odd angles (other than 45-degrees), ostensibly for aesthetic or legibility purposes.

Our work leverages the fact that PCBs are composed of far fewer components compared to VLSI. Our placer uses stochastic combinatorial optimization which, although typically incapable of placing hundreds of thousands of cells, we successfully apply it to PCB designs. Such generic global optimization techniques have been demonstrated to perform well on the PCB placement problem. For example, Jain and Gea[JG96] demonstrated the efficacy of genetic algorithms on layout problems involving multiple objectives. In this work, we apply simulated annealing[KGV83] to jointly optimize wirelength, overlap/density, and routability.

Differences between PCB and VLSI routing

Similar to the placement problem, there are several key differences that prevent the use of standard VLSI routing tools for PCBs:

1. Large vias: Vias can be much wider than the routing trace width. Through-hole type vias penetrate the whole PCB, blocking routing on all layers. Laser vias are smaller than through-hole (generally still larger than trace width) and are significantly more expensive.

2. Free angle routing: Because of the large and expensive vias, PCB traces are commonly routed at odd angles through the conductor layers to limit vias use. Traces are not restricted to a layer's "preferred direction." Commonly, PCB traces are routed using "eight-direction routing," or multiples of 45 degrees.

3. Layer count: Routing layer count should be minimized because the number of conductor layers is the most significant contributor to the cost of PCB manufacture.

Canonical approaches for routing PCBs include Mazeroute and variants [OW06]. Topological-based routing has emerged as a flexible alternative, able to handle a variety of cases where Mazerouting fails [SJDD93]. Our router adopts a grid-based cost-driven rip-up and reroute router.

In conclusion, the differences between VLSI and PCB have led us to create a new placer, router, and design database focused on the issues unique to PCB layout, rather than attempting to modify an existing IC flow.

4.4 An Automated Open-source Flexible PCB Layout Framework

Our framework is unique in being both open-source and fully-automated. The numerous benefits of open-source are critical to our project and its impact. These benefits have been

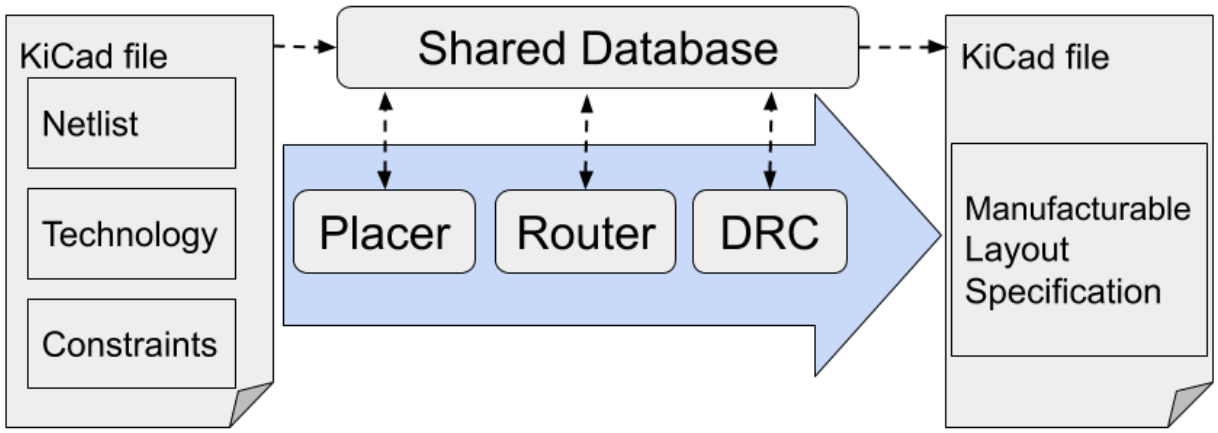

Figure 4.1: A diagram of the framework and flow. The dashed arrows represent passing layout information. The large blue arrow shows the flow from incomplete layout to completed layout.

thoroughly discussed in the context of EDA [HLGW19]. A push-button flow enhances the applicability of such a framework, freeing researchers from details that are irrelevant to their particular work, and allowing different parts of the layout process to be isolated and tested across multiple tools in the context of an entire flow [KLL14].

An overview of our framework is illustrated in Figure 4.1. We are releasing the following modules as part of our framework:

1. shared design database
2. placer
3. autorouter
4. design rule checker (DRC)

The framework is highly flexible, allowing for each stage –placement, routing, and design rule checking– of the flow to be interchanged between several options as detailed in Sections 4.4.2, 4.4.3 and 4.4.4. Included with this work, we release a fully-automated, open-source tool for each stage. Alternatively, each stage can be completed using other external tools with varying degrees of automation. A scripting API facilitates the creation of additional stages and modification to the flow. Each of the framework components that we have implemented,

including the database, exposes both a C++ and Python API. Iteration between stages is scriptable, if desired, and manual placement, manual routing, or design inspection can each be inserted at any stage.

4.4.1 Database

Our shared design database is a utility for importing and exporting PCB layout designs and exposes an API that enables easy reading and writing of PCB layout information in both C++ and Python. API documentation is available in the project repository. Our database can also interface with other tools (for example, extant placers and routers) using a file-based interface.

The database can read and write to the KiCad PCB layout format (`.kicad_pcb`), a plain-text, fully open-source PCB layout format. See Section 4.9 for a discussion of our selection of this file format.

Additionally, a partial, intermediate layout can be written out at any stage. These partial layout files can be imported by KiCad and manually inspected or modified using the KiCad GUI. The manually modified files can be reloaded by the database and further processed by the automated tools.

Upon completion of the layout, Gerber files can be generated from KiCad PCB files for multiple manufactures. PCB manufacturers accept Gerber files as a standard manufacturing specification.

4.4.2 Placer

Our framework may use any combination of the following:

1. Our placer: We use a simulated annealing (SA) algorithm to find a placement that is optimized with regards to the sum of an overlap penalization term, unrouted air-wire lengths, and a routing congestion prediction metric[SJ07]. Our placer can be used as part of

a full-automated, push-button flow. See Sections 4.5 for algorithm details and Section 4.10 for evaluation.

2. KiCad autoplacer: KiCad has a utility for spreading components on the PCB. KiCad's autoplacer attempts to minimize air-wire length, with a preference for compacting components into the lower-right corner of the board. KiCad's autoplacer requires use of the KiCad GUI.

3. Other external tools: Our framework can be used with any other placement tool compatible with the KiCad file types or capable of being modified to interface with our database API.

4. Manual placement: Manual placement can be performed by placing and locking components in the design through the KiCad GUI. The design can then be exported to or imported from the KiCad GUI at any stage.

Placement constraints

PCB layout generally starts with a partial placement for off-board interconnects and user interface components. These components are "locked" and may not be moved by the automated placer. Interface components like USB connectors, pin headers, power plugs, reset buttons are commonly locked.

Components must be placed within a board outline, the geometry of which is given as a set of line segments and arcs that form a continuous outline of the PCB's perimeter. All component pads and pins must be placed completely within the PCB's perimeter.

4.4.3 Router

Our framework may use any combination of the following:

1. Our autorouter: Our own autorouter uses a grid-based A*-search and cost-based rip-up and reroute. Following the general practice of VLSI routing, our autorouter prioritizes layout-vs-schematic (LvS) completion over design rule violation (DRV) cleanness, differentiating it

from existing PCB autorouters, which refuse to route nets unless a DRV-clean solution can be found. Our autorouter can be used as part of a full-automated, push-button flow. Details of the algorithm can be found in Section 4.6. Our autorouter is evaluated in Section 4.10.

2. FreeRouting: Multiple stand-alone autorouters are compatible with KiCad, the most prominent being FreeRouting, which uses a topological routing algorithm. FreeRouting only performs DRV-clean routing, and will not output routes that violate the design rules. This is because FreeRouting assumes active human interaction and verification. FreeRouting can be run in an automated way, but requires file conversion through KiCad. FreeRouting is open-source software, but does not appear to be actively maintained. We evaluate FreeRouting in Section 4.10.

3. Other stand-alone autorouters: Any autorouter can be used with our flow provided that it is either compatible with KiCad files or that it can be modified to interface with our database's API.

4. Manual routing: Routing can be performed manually using the KiCad GUI, or using any graphical layout program that is compatible with KiCad.

4.4.4 Design Rule Checker

We have implemented an automated design rule checker (DRC) that can interact with our database. KiCad also has an integrated DRC, but it has been designed to be used within the KiCad GUI and is difficult to script and modify.

4.4.5 Public availability

We offer all parts of our framework with source code and the BSD 3 license[1]. The framework source code is available on GitHub[2]. We have attempted to set up this work to be a maintainable open-source project. Continuous integration tests run through the Jenkins framework and API documentation is available in the GitHub repository. An install script is available for Ubuntu.

4.5 Simulated Annealing for PCB Placement

Our placer uses a simulated annealing strategy with a mixed objective function. Simulated annealing is a standard algorithm [KGV83]; this section briefly outlines some of the issues specific to PCB placement.

Our objective function penalizes the sum of half-perimeter wire length (HPWL) over all nets, a routing congestion metric[SJ07], and a component overlap score. We use the Boost.Polygon library for calculating component overlap. At the beginning of annealing, the objective function places 100% of weight on HPWL and congestion and none on overlap. As annealing progresses, the objective function is reweighted, placing more weight on overlap, proportional to the remaining annealing iterations. Annealing ends with overlap weighted at 100%. To maintain the layout solution in a low HPWL-congestion space, the move size is reduced as HPWL-congestion weight is decreased. At the end of annealing, the allowed component move radius (in Manhattan space) is one millimeter.

We have experimented with several formulas for penalizing overlap. Our best results have been achieved using the following three-term measure: Let $o_{a,b}$ be the area of overlap between the courtyard polygons of components (in mm^2) a and b, $d_{a,b}$ be the distance between the geometric

[1]The BSD 3 license is a permissive open-source license.

[2]`https://github.com/ICCAD2020-Submission271/PCB-Layout-Framework`

centers of the courtyard polygons of components a and b, and $k_{a,b}$ be the indicator function $k_{a,b} = 1$ if $o_{a,b} > 0$; else 0.

$$overlap\ penalty = \sum_{i,j} o_{i,j}^2 + o_{i,j} + \frac{k_{i,j}}{1 + d_{i,j}}$$

Penalizing the overlap quadratically encourages small overlap between all the components. The linear term produces better behavior when overlap between two components is less than 1 mm^2 (when the edges are barely overlapping). Also, when the allowed move size is small, the last term has the effect of "pushing" small components towards the edge of larger components in the case that their courtyard polygon is wholly within the larger components.

Components are also allowed to rotate during placement. We fix the ratio of rotation moves at 25%. We reject all moves that would place a component outside the board outline.

4.6 A* Search for PCB Routing

Our router uses an iterative cost-based rip-up and reroute strategy based on A* search. This section The vector-based PCB design is rasterized onto a grid. The pads of each component are rasterized and represent obstacles during routing. Each net is then routed using minimum-cost graph search. When a point-to-point connection is made, a cost is added to each grid-cell that the new trace occupies. For multi-pin nets, the search frontier is initialized as the set of grid-cells occupied by previously routed connections for that net. After every net has been routed to minimum cost, nets are ripped up (their costs removed from the routing grid) and rerouted, one by one. Ripping up and rerouting all nets once represents one iteration of our router algorithm. The results for our router, presented in Section 4.10, were routed with 20 iterations.

Because the trace-width is wider than the grid resolution (typically by a factor of 6 or more) we use a partial cost update method for the search-frontier, based on a rasterized circle whose radius is the trace width. That is, when we expand into a new cell, we only add and subtract

the cost of the grid-cells at the edge of the circle. Large vias cause a considerable slow-down for this algorithm. They require a much larger cost update (their radius is larger than the trace width) and they require cost lookups on all layers. Allowing only laser vias (which are smaller than through-hole vias and only penetrate one layer) results in an approximate 4x speedup over through-hole vias. To handle 45-degree routing we allow traces to expand into their 45-degree corner neighbors with an increased cost of factor $\sqrt{2}$. This requires the use of floating-point numbers to store cost.

Adding support for high-speed nets and other advanced constraints is ongoing.

Our discussions with layout engineers have led us to penalize trace bending. The layout engineers that examined our results commented that the result looked "cleaner" and they were more accepting of the automated layouts after this change. However, they acknowledged that the bends would not affect electrical correctness.

4.7 Metrics

Metrics are standards used to objectively evaluate layout quality and quantitative comparison between systems without relying on aesthetics. We identified design-rule violations (DRVs), through-hole via count, laser via count, and laser via layer count as the key metrics because they are direct outputs of the layout flow and each affects either design correctness or design cost.

4.7.1 DRVs

In the context of routing, DRVs include pad-to-trace, pad-to-via, trace-to-trace, trace-to-via, and via-to-via clearance violations.

In the context of placement, DRVs include pad-to-pad clearance violations, overlap of component outlines, and off-board placement (placement of components completely or partially outside of the board outline).

Numerous other advanced design rules affect PCB layout quality, including but not limited to power delivery, thermal dissipation, differential pair routing quality, bus length matching, signal integrity, pad entry, high-speed net topology, and decoupling capacitor placement. Commercial layout tools have mixed support for these advanced rules. To the best of our knowledge, no commercial layout tool is able to encode and check for all of the advanced design rules required of a modern, high-speed PCB design. These tools typically rely on "app notes" written and checked by human engineers. KiCad, specifically, does not currently support advanced constraints, but support is in the road map for Version 6, the next major version.

The majority of our benchmark designs do not require advanced design rules. The few that do did not have these parameters formally specified in their design files.

Our framework, which supports simple design rules, is a necessary first step towards implementing a future framework that supports more advanced rules. We are currently adding support for some advanced rules to our placer and router (see Section 4.11).

4.7.2 LvS errors

An LvS error occurs when a connection that appears in a design's netlist does not have a corresponding electrical connection after routing. Our experience is that power and ground nets are most likely to be subject to a LvS error due to their relatively large number of connections. Our router prioritizes LvS connectivity over DRV cleanness and never produces LvS errors, but all other extant autorouters will produce LvS errors if they cannot find a DRV clean solution.

4.7.3 Via count

Vias are conductive connections between conductive layers and can be either through-hole or laser. Per-piece board cost increases with each through-hole via in a design. Through-hole vias penetrate and connect every layer of a PCB; laser vias connect only two adjacent layers and

can be made smaller than through-hole vias. Our benchmarks include design rules that specify minimum via sizes.

Through-hole via count equals the number of through-hole vias used in a layout.

Minimum through-hole via diameter increases with the number of conductor layers due to the difficulty of plating vias with tall aspect ratios in thick PCBs. Exact minimum through-hole via diameter varies between manufacturers, but our benchmarks contain per-net minimum through-hole via sizes as design rules.

Laser via count equals the number of laser vias used in a layout and should be minimized because they are a common failure point in PCBs due to thermal and mechanical stress.

Laser via layer count equals the number of adjacent layer pairs connected by laser vias. For example, if three adjacent conductor layers (with two layers of dielectric) are connected by laser vias, the laser via layer count is two.

Each pair of layers connected by laser vias increases the manufacturing cost by approximately 40% due to process cost and yield losses.

4.7.4 Other metrics

We report two other metrics to differentiate the tools evaluated in this work:

CPU time is the total run time of a layout program. The scale of run times reported (less than one hour) have little effect on cost, but run time may have more significant consequences for designs more complex than those discussed in this work.

Total trace length is the combined linear length of all traces in a routed design. This metric does not affect design correctness or cost, but may indicate general "competence" of a router in avoiding detours.

Table 4.1: Design characteristics of PCB benchmarks.

design	#components	#nets	#pins	#locked	design area
bm1	60	99	319	18	4890.44
bm2	19	34	77	4	874.01
bm3	58	80	229	10	1358.78
bm4	48	54	163	7	1432.23
bm5	34	38	138	4	1220.87
bm6	28	52	140	6	1143.52
bm7	8	15	40	1	193.01
bm8	36	70	188	4	4342.17
bm9	61	63	314	4	5520.65
bm10	58	35	233	6	2920.12
bm11	46	69	207	7	1176.24

#pins includes unconnected pins. #locked is the number of fixed location components. Design area is the area of the smallest rectangle that covers the board outline, in square millimeters.

4.8 PCB Benchmarks

In this work, benchmarks serve as standard problems against which to measure the performance of competing tools. (N.B. This differs from the use of the word benchmark to indicate searching the industry for best practices[alt].) We present a collection of 11 PCB designs to serve as benchmarks for automated layout[3].

These designs were selected because they contain a variety of complexity levels, are small enough to be completed in a reasonable time using our automated framework, and are simple enough that they do not contain many advanced constraints, such as trace-length matching or differential pair coupling. Each benchmark design is presented with a full manual placement and routing with the intent of providing a "known good" point of comparison for automated layouts. The benchmarks presented were prepared from layouts that have been manufactured and functionally tested. Further details on each design, including license information, are available in the GitHub repository.

[3] `https://github.com/ICCAD2020-Submission271/PCBBenchmarks`

4.8.1 Summary of benchmark designs

For each benchmark design we present a short summary of the features of the design. The designs and some of their characteristics are listed in Table 4.1. Additional information on each design is available in the design repository.

bm1

Based on a well known microcontroller (MCU) carrier board (Arduino Mega), bm1 contains a high pincount MCU.

bm2

Based on the Adalogger FeatherWing, a commercial design, bm2 contains two large, oddly shaped components: an SD card slot, and a coin cell battery. The remaining area available for placement is fully enclosed by these two components and interface pins along the sides.

bm3

A quadcopter controller board, bm3 contains an antenna and four motor driver circuits to power the propellers.

bm4

An audio processor breakout board, bm4 contains a headphone jack and a MCU. This MCU has fewer pins than the MCU in bm1, however, this MCU has a large ground pad that prevents routing through the MCU's footprint.

bm5

A USB to lithium battery charger, bm5 contains several different connector components and the components of the charging circuit have unusual footprints.

bm6

An MCU carrier board, bm6 has a very narrow aspect ratio. We have found that narrow PCBs are difficult to place with simulated annealing. Random moves often go outside of the board boundary and components can become stuck in local minima as the movement step size decreases during cooling.

bm7

An accelerometer breakout board, bm7 is useful as a sanity check and can be automatically completed in less than one minute. It can be routed using only one layer.

bm8

An MCU carrier board, bm8 only contains through-hole components and does not contain any surface mount components (SMDs).

bm9

A remote control board, bm9 contains many locked user-interface components including buttons, switches, and LEDs. It is also a four-layer design.

bm10

Another quadcopter design, bm10 contains larger motor driver components and four routing layers.

bm11

Based on the Adafruit Feather nRF52 Pro Bluetooth LE development board, bm11 contains a high pincount Bluetooth modem and a very tight placement area.

4.8.2 Benchmark preparation

Human-generated PCB designs lack the regularity that would make them suitable for automatic layout. Human designers are inconsistent in their use of the layout format features. We took care, therefore, to process each design file to conform to a set of stylistic standards. We also added missing information that was not explicitly encoded in the original design files.

We regularized each design with the following steps:

1. Designs not originally in the KiCad layout format were converted to that format;
2. Images, text, and other non-functional copper elements were removed;
3. Copper pour areas (also known as copper fill or zones) were removed, and nets that relied on copper pour for connectivity were manually routed;
4. Silk-screen-only components and other non-copper components were removed;
5. Silk-screen outlines and designator text were replaced with polygons in the front and back courtyard layers (CrtYd.F and CrtYd.B). These courtyard polygon serve as component outlines for placement purposes;
6. Board outlines were simplified in some cases and made continuous;
7. Connectors and interface components were marked as locked as appropriate;
8. Net classes were added to specify trace width;
9. Components, traces, and design rules were moved or modified so that the KiCad DRC reports zero errors.

4.9 Open-source PCB Layout Format

The IC community has standardized on LEF/DEF as an open layout format, enabling research and commercial tools to be interoperable. Interoperability allows research tools to be inserted into, and even replace parts of, commercial tools chains. Also, LEF/DEF can be freely extended by anyone to add advanced features as the needs of IC design evolve. PCB layout research would also realize significant benefit from a standard layout format.

We propose the KiCad PCB layout format (`.kicad_pcb`) as a preferred standard format for PCB layout benchmarks due to:

1. an active development community,
2. the openness of the format, and
3. its availability as a widely compatible, open-source tool.

KiCad is a full EDA suite for PCB, including schematic capture, manual layout tools, 3D board modeling, and CAM generation. Created in 1992, KiCad is an open-source tool for PCB layout with a very active development community that provides regular updates. The Git repository shows dozens of updates and bug fixes per week[4]. The European Organization for Nuclear Research (CERN) is an active contributor and recommends its use for open hardware projects[cer].

Unlike commercial PCB layout tools, which each have their own proprietary, often encoded file format, the KiCad layout format is plain-text and can be freely extended. KiCad format documentation is licensed under Creative Commons Attribution License version 3.0.

[4] https://gitlab.com/kicad/code/kicad

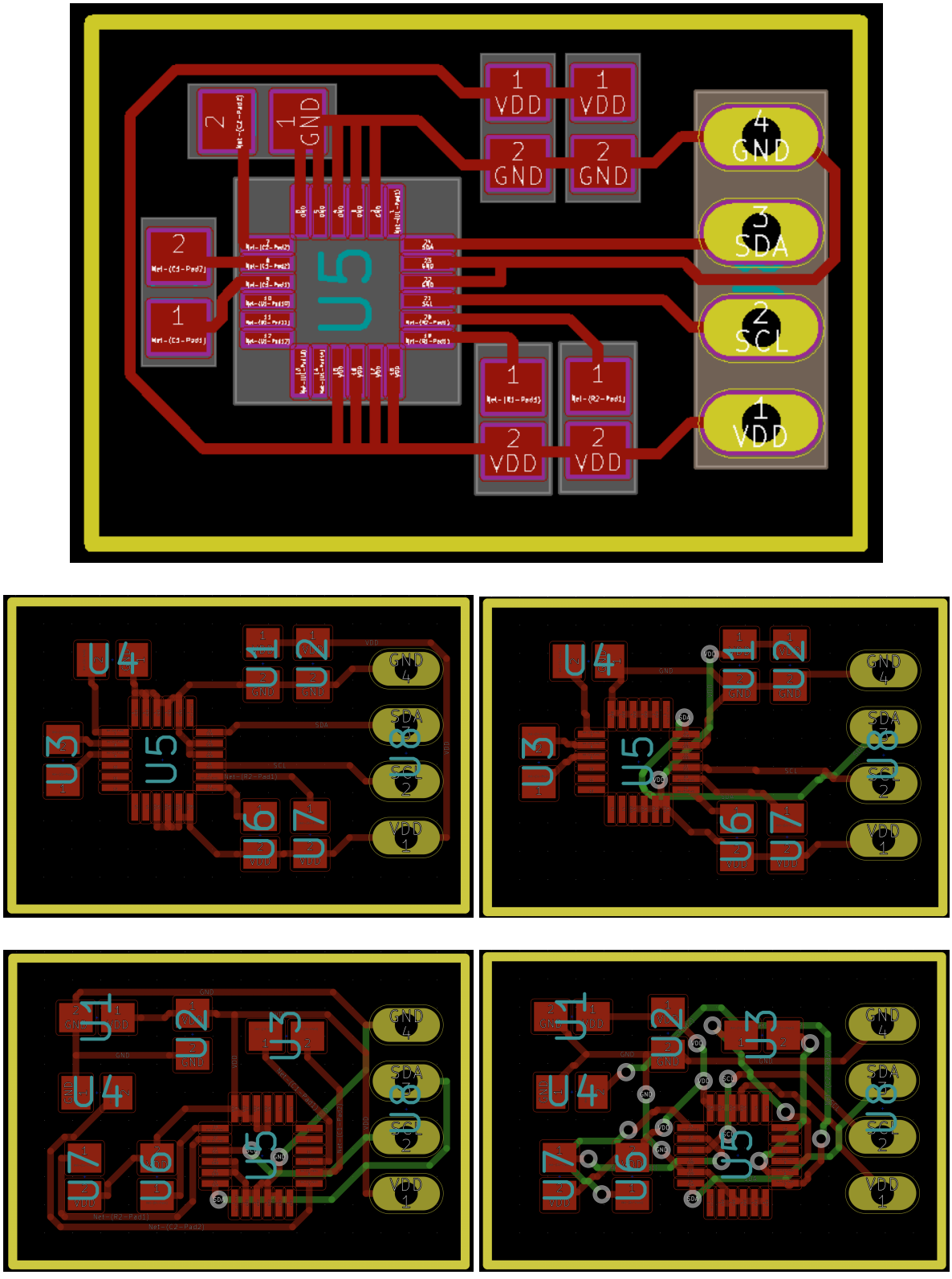

Figure 4.2: Several bm7 layouts. Manual placement and route (top); manual placement and FreeRouting (middle left); manual placement and our router (middle right); SA placer and FreeRouting (bottom left); SA placement and our router (bottom right).

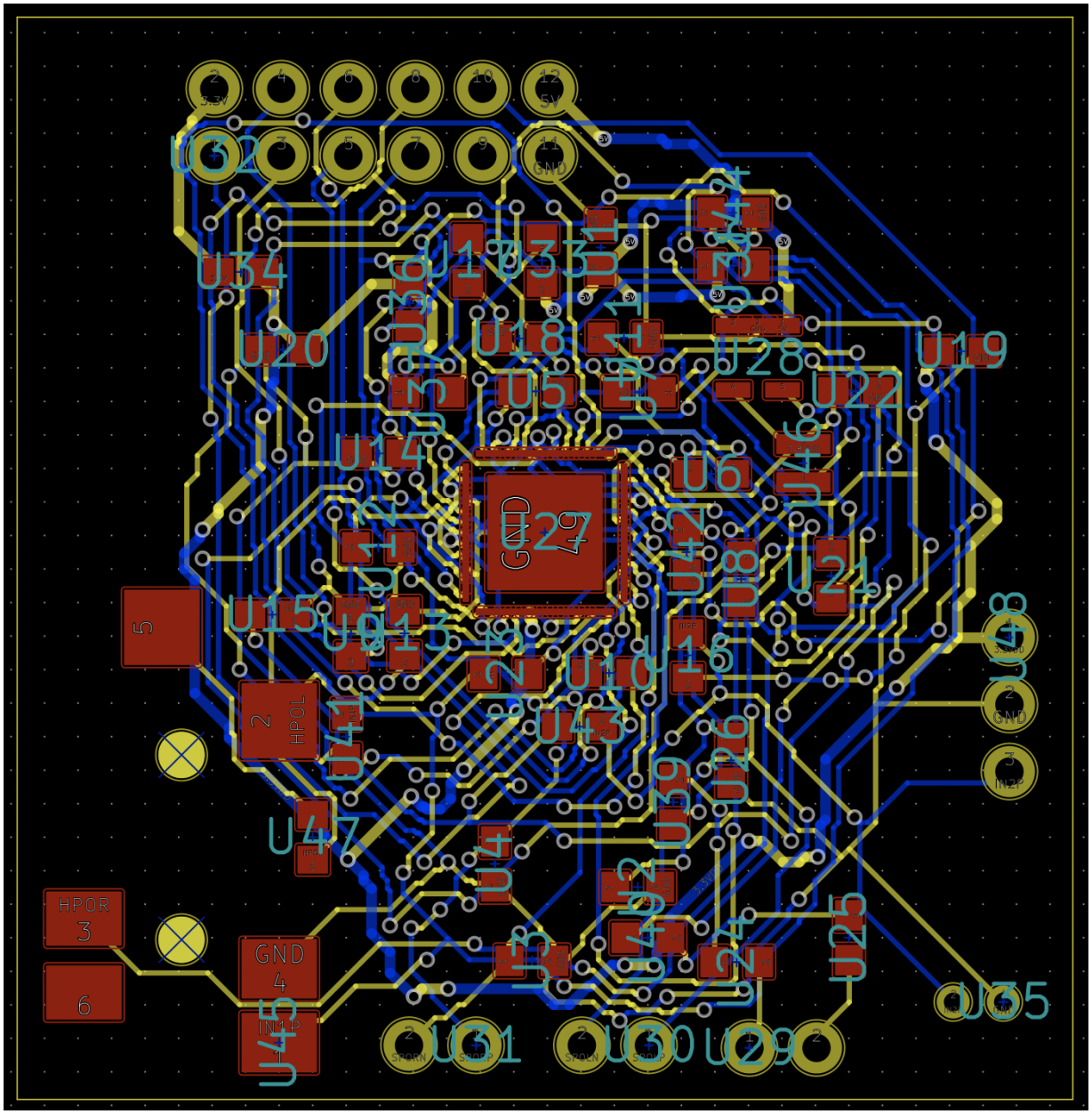

Figure 4.3: Layout of benchmark design bm4 using our placer and router. This figure shows a design of medium complexity and a layout result generated without any human input.

4.10 Layout results

We present comparative PCB layout results for each benchmark design in Tables 4.2 (via counts), 4.3 (total trace length), and 4.4 (CPU time) using the metrics for five different combinations of placers and routers:

1. manual placement and routing,
2. manual placement with FreeRouting (FR) autorouter,
3. our placer with FreeRouting autorouter,
4. manual placement with our autorouter, and
5. a fully automatic layout using our framework.

Figure 4.2 shows bm7 routed with each combination, for visual comparison. Figure 4.3 shows a larger design (bm4) with fully automated layout.

We modified the basic implementation of FreeRouting to allow FreeRouting to be run in "batch mode" without a GUI. Even so, evaluating the output of FreeRouting requires manually integrating the output with the original design using the KiCad GUI. We used an automated script to collect via and trace-length information after manual conversion.

Via count

Via counts for each placer and router combination are presented in Table 4.2. Manual routing uses through-hole vias; our router and FreeRouting use laser vias. FreeRouting outperforms the manual result for most designs, perhaps because the human designers were trying to optimize for legibility over via count. Our router uses 11.2 times more vias when routing bm10. In this layout, the configuration of the two high pin count components causes our router to take many layer-changing detours.

Our router's solution for bm8's automatic placement uses 41 times more vias than FreeRouting. In this placement the large through-hole MCU and headers are placed to cut across the middle of the PCB, causing a routing blockage. Our placer also compacts the components in this design, leaving too-little room for routing between them; this is a serious issue for through-hole components because their pins block routing on all layers.

Our router uses 6.5 times more vias to route, the automatic placement of bm11. This layout also contains approximately 120 DRVs. The high pin count components and small space make bm11 very challenging to place.

As our router uses A* search, via count is heavily dependent on a "layer change cost" parameter that penalizes expanding search to other layers while routing. Tuning the layer change cost may improve via count at the expense of run time. We did not experiment with tuning routing parameter in this work.

Total trace length

Routed wirelength for each placer and router combination are presented in Table 4.3. Manual layouts generally result in less total trace length compared to automatic placement. The automatic placements bm8 and bm11 have especially high trace length compared to the manual placement due to the layout issues that also affected via count. For bm8, the trade off between trace length and via count is evident. FreeRouting prefers to take more detours on the same layer, while our router prefers vias. This behavior is dependant on tuning parameters for both routers.

CPU time

CPU routing time for each router on each placement is presented in Table 4.4. The manual placement of bm9 has comparatively long pin-to-pin distances. This challenges our routers grid-based search algorithm. The topological routing algorithm of FreeRouting is able to find a solution comparatively quickly.

Table 4.2: Layout results by via count for each benchmark.

Router:	manual	FR	FR	ours	ours
Placer:	manual	manual	ours	manual	ours
bm1	129	*	*	280	331
bm2	4	4	6	11	18
bm3	44	*	*	73	97
bm4	37	*	*	49	61
bm5	33	*	*	27	37
bm6	30	6	13	47	68
bm7	0	0	2	10	11
bm8	0	0	1	3	41
bm9	166	*	*	128	128
bm10	37	25	27	280	36
bm11	75	*	*	48	310

Our router ran with 20 iterations of rip-up and reroute. More iterations may improve metrics at the cost of CPU time. We terminated FreeRouting if it did not complete in less than 5 hours, but more routing time may have allowed FreeRouting to find a viable solution. FreeRouting often becomes "stuck" if it cannot find a DRV-free solution. Both FreeRouting and our router run single-threaded. Routing was run on Google Cloud Platform virtual instances (type n1-standard-16, Intel Haswell).

DRVs

Layouts generated with FreeRouting do not contain DRVs, but FreeRouting may not terminate if a DRV-free solution cannot be found. Layouts generated with our autorouter have complete LvS connectivity but contain clearance-related DRC errors (except for bm7, which has none).

We suspect that DRVs can be reduced by tuning algorithm parameters for each design but a full evaluation of these effects is beyond the scope of this work.

Table 4.3: Layout results by total trace length (mm) for each benchmark.

Router:	manual	FR	FR	ours	ours
Placer:	manual	manual	ours	manual	ours
bm1	5034	*	*	5710	5326
bm2	395	323	503	432	591
bm3	1300	*	*	1380	1421
bm4	1043	*	*	1117	1054
bm5	681	*	*	578	600
bm6	778	834	1009	1021	1239
bm7	128	110	110	97	87
bm8	1483	1110	3960	1529	2633
bm9	3721	*	*	4043	3548
bm10	1743	1527	1396	2135	1980
bm11	1120	*	*	1216	2158

Table 4.4: Layout results by CPU routing time (s) for each benchmark.

Router:	FR	FR	ours	ours
Placer:	manual	ours	manual	ours
bm1	*	*	1275	1196
bm2	47	51	71	129
bm3	*	*	654	688
bm4	*	*	291	412
bm5	*	*	336	305
bm6	174	224	200	167
bm7	7	14	17	28
bm8	50	130	117	373
bm9	*	*	3574	66
bm10	537	620	2720	2601
bm11	*	*	251	315

* entries did not complete in less than 5 hours.

4.11 Future and Continuing Work

Overall, our framework demonstrates significant progress in developing a completely automated end-to-end PCB layout flow. However, producing a placer and router that can rival manual layout is still a significant challenge for the future.

We have also identified several areas that no automated PCB layout tools (commercial or otherwise) address: copper fills, laser via layer count optimization, automated placement-routing iteration, design area optimization, 3D aware placement, and routing-aware placement. Each of these aspects of PCB layout is present in commercial PCBs but must be specified manually by human engineers, increasing cost and turn-around time. Our framework provides an entry point for research into automating solutions to commercially-relevant problems.

Our framework can also help demonstrate the value of the ongoing academic research into escape routing, differential pair routing, and bus length matching, by providing the context of a full layout flow and manufacturable result.

We will continue to augment our PCB benchmark suite with real, open-source designs, with a focus on designs that require advanced constraints. We hope that our work motivates others to release open benchmarks and tools for PCB layout.

4.12 Acknowledgement

Chapter 4, in part, is currently being prepared for submission for publication of the material. Merrill, Devon; Lin, Ting-Chou; Holtz, Chester; Wu, Yen-Yi; Garza, Jorge; Swanson, Steven; and Cheng, Chung-Kuan. The dissertation author was the primary author of this chapter.

Chapter 5

Conclusion

We have attacked the problem of computational design for embedded systems from three angles: a class for novices; a computational design tool for more complex devices; and a framework for automated PCB layout.

From the results of these projects, we can draw several conclusions. But, primarily, these projects show that it is possible to create end-to-end novice-friendly computational design tools in the embedded systems space that are fast for moderately complex devices and capable of creating dense PCBs with advanced constraints. Several specific concluding remarks on each project follow:

Conclusions from the Robot Parade course

We pilot tested our computational design tool approach in an introductory physical computing course (which we called Robot Parade) for 196 CS1 students. The students designed, assembled, and programmed a custom robot with minimal instructor assistance. The robots are Arduino-based, and each included a student-designed PCB. The students assembled the robots from off-the-shelf electronic components according to automatically generated assembly instructions. Students programmed their robots using simple APIs that were automatically

generated and customized for each unique robot. A minimum of two quarters after the completion of the course, the grade point average (GPA) for the students who completed the course, was found to be 0.15 higher than a comparison group of similar students ($n = 498$, $p < 0.05$).

The Robot Parade course, from the perspective of its creators, had a dual purpose. First, to give students hands-on engineering experience. Specifically, the end-to-end experience of designing, building, and programming an embedded device with a custom PCB. Second, to demonstrate our computational design tool, a tool that facilitates each phase of the creation of small robots from the PCB, to the assembly, to the programming.

Robot Parade was successful in these goals, but Robot Factory, the design tool, did not produce optimal designs in terms of PCB size, design cost, or manufacturability. Still, the tool enabled hundreds of students to create functional embedded devices without the intensive instructional support that would have been necessary in a traditional lab setting. This relatively simple computational design tools gave hands-on experience to students and appeared to have positively impacted the students' GPA. The success of our approach to the computational design tool, including the functional design style of computational design, lead us to expand the tool with Echidna.

Conclusions from Echidna

Compared to the Robot Parade tool, Echidna allows more complex devices to be created by using a heuristic search algorithm. Echidna still enables non-electrical engineers to move from conception to implementation with their mechatronic ideas by generating and searching through a design space that automatically fills in supporting requirements, such as PCB placement and wiring, around their functional specification. We model the space as a decision tree whose root is the user's list of lights, motors, sensors, and other functional components that need to be connected, powered, and controlled. Once found, a complete and valid design can be used to synthesize geometry for 3D printing, circuits, and firmware resulting in a set of "ready-to-go"

files for creating their device.

Echidna contributes an algorithm that efficiently designs mix-domain (electronic, software, and mechanical) embedded systems for computational design tools. Fast design algorithms are necessary for computational design tools for electronics to move out of the "toy" stage towards devices that are comparable to those designed by human engineers. Components facilitate solving the requirement satisfaction problem. Library creation and maintenance are the difficult part of a component-biased computational design system for embedded electronics. Synthesizing electronic devices using component libraries is well-suited to heuristic search and can be fast.

There are two factors that limit the complexity and capability of the devices Echidna can design. The first is library support. Creating a large library of validated and compatible components is not necessarily a research problem but does require a large number of engineer-hours.

The second is the limited capability of contemporary PCB layout tools, especially open-source tools that are available to researchers. Contemporary commercial PCB layout tools are reasonably capable of routing moderately-complex PCBs (for example, boards with DDR3) if an engineer provides a compatible manual placement and manually programs the routing constraints. However, commercial tools are not designed to be fully automated, they cannot automatically place components, and they are closed source, so they cannot be modified by researchers. Current open-source PCB layout programs are modifiable but are still not designed to be fully-automated and are much less capable of complex routing with advanced constraints.

Conclusions from our PCB layout framework

The wide gap between PCB layout research and commercial practice limits the usefulness of the computational design community's research contributions to industry. PCB layout researchers can emulate the steps taken by the VLSI community to solve this problem: specifically, the steps of creating open benchmarks and open and automated workflows. VLSI workflows are

available that can take a design from a netlist to layout with minimal human interaction. These workflows enable the evaluation of a contribution in context, for example, in evaluating whether a change in layout methodology improves routability. They also facilitate research contributions, for example, allowing experimentation with routing heuristics without coding a router from scratch.

The PCB layout framework, detailed in Chapter 4, has the unique combination of being fully-automated and open-source. Ours is the first PCB layout framework that can provide the same benefits to PCB layout research as existing VLSI workflows provide in that domain. We also propose a method for objectively evaluating PCB layouts: a set of benchmarks, a proposal for adopting a standard open benchmark format, and a list of evaluation metrics. The PCB benchmarks are based on real designs, ranging from 19 to 57 components and from 15 to 99 nets. We believe that these contributions will help close the gap between PCB layout research and commercial practice, supporting the production and evaluation of industry-relevant PCB layout research.

The future of fully-automated design for embedded systems

Our vision for the future of automated design for embedded systems is a single PCB synthesis toolchain that assists the user with functional specification and totally automates schematic-capture and layout for complex devices with human-level results. Echidna, as discussed in Chapter 3, encompasses much of this vision but cannot meet the goals of human-level results on complex devices. A high-level road map for the future of automated embedded system design can be imagined:

The first step is moving away from application notes; that is, human-language notes that specify circuit constraints. Commercial tools have the ability to document many electrical constraints, but these capabilities are often not used and are rarely fully used in industry. Our discussions with PCB CAD engineers have exposed a push towards requiring machine-readable

specifications for the electrical constraints in a PCB design.

Once circuit constraints are commonly expressed in a machine-readable format, circuit design can move away from graphical schematic capture. PCB hardware description languages (HDLs) are poised to replace graphical schematic design. For example, the recent proposal of Python-based hierarchical circuit blocks [LRC+20].

In parallel to HDL-based schematic capture will be the maturation of no-human-in-the-loop PCB layout. Automated PCB layout requires machine-readable circuit specifications, such as expressed in an HDL. However, the layout is a very different problem that requires optimization in a non-differentiable space. We believe a focused open-source effort is required to realize this goal in a way that is accessible to the research community. Chapter 4 happens to present the beginnings of such an effort.

The user interface to a speculated future PCB-synthesis tool chain may be the PCB HDL directly, at least for experts. Although, there is a resistance among contemporary PCB engineers to changing their current design practice. For non-experts, graphical tools, such as those discussed in Chapter 2, may be more appropriate. Unfortunately, no tools in the academic space have been demonstrated as an effective front-end for embedded systems beyond low complexity; demonstration designs are usually at the level of toys, such as pattern matching games. In the commercial space, Gepetto [gum] can create higher-complexity designs but does use pre-routed PCB circuit blocks and human-driven routing behind the scenes.

If the challenges discussed in this section continue to be given serious attention by the research community, there will be a revolution in PCB design over the next several years.

Bibliography

[adv] Advanced circuits. www.4pcb.com/.

[AGF17a] Fraser Anderson, Tovi Grossman, and George Fitzmaurice. Trigger-action-circuits. *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology - UIST 17*, 2017.

[AGF17b] 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*, UIST '17, pages 331–342, New York, NY, USA, 2017. ACM.

[alt] Benchmarking practices and process for pcb designers. Accessed: 2020-05-28.

[ANP+19] Tutu Ajayi, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, and Lutong Wang. Toward an open-source digital flow: First learnings from the openroad project. *Proceedings of the 56th Annual Design Automation Conference 2019 on - DAC 19*, 2019.

[BBB+16] 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)*. IEEE, 2016.

[BCC17] James M. Bern, Kai-Hung Chang, and Stelian Coros. Interactive design of animated plushies. *ACM Transactions on Graphics*, 36(4):1–11, 2017.

[BCT+15] Gaurav Bharaj, Stelian Coros, Bernhard Thomaszewski, James Tompkin, Bernd Bickel, and Hanspeter Pfister. Computational design of walking automata. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, SCA '15, pages 93–100, New York, NY, USA, 2015. ACM.

[CC00] Michael E. Caspersen and Henrik Baerbak Christensen. Here, there and everywhere - on the recurring use of turtle graphics in cs1. *Proceedings of the Australasian conference on Computing education - ACSE 00*, 2000.

[cer] Kicad software gets the cern treatment. Accessed: 2020-05-28.

[CKKW18] C. Cheng, A. B. Kahng, I. Kang, and L. Wang. Replace: Advancing solution quality and routability validation in global placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018.

[Cor13] Stelian Coros. Computational design and motion control for characters in the real world. *Proceedings of Motion on Games*, 2013.

[CSF13] Arquimedes Canedo, Eric Schwarzenbach, and Mohammad Abdullah Al Faruque. Context-sensitive synthesis of executable functional models of cyber-physical systems. *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems - ICCPS '13*, 2013.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DSZ+16] Tao Du, Adriana Schulz, Bo Zhu, Bernd Bickel, and Wojciech Matusik. Computational multicopter design. *ACM Transactions on Graphics*, 35(6):1–10, Nov 2016.

[eda14] Edasolver - automatic component selection and pin matching, 2014. Accessed: 2018-04-05.

[FI65] C. J. Fisk and D. D. Isett. "accel" automated circuit card etching layout. *Proceedings of the SHARE design automation project on - DAC 65*, 1965.

[GDG+17] Gaoyang Guan, Wei Dong, Yi Gao, Kaibo Fu, and Zhihao Cheng. Tinylink: A holistic system for rapid development of iot applications. *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking - MobiCom 17*, 2017.

[Gey71] J Geyer. Connection routing algorithm for printed circuit boards. *IEEE Transactions on Circuit Theory*, 1971.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dreal: An smt solver for nonlinear theories over the reals. In *Proceedings of the 24th International Conference on Automated Deduction*, CADE'13, pages 208–214, Berlin, Heidelberg, 2013. Springer-Verlag.

[gum] Meet geppetto, iot embedded hardware design and production. Accessed: 2020-05-28.

[HLGW19] Tsung-Wei Huang, Chun Xun Lin, Guannan Guo, and Martin D F Wong. Essential building blocks for creating an open-source eda project. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019*, Proceedings - Design

Automation Conference, United States, 6 2019. Institute of Electrical and Electronics Engineers Inc.

[HP91] Idit Harel and Seymour Papert. *Constructionism*. Ablex Publishing Corporation, 1991.

[Ind] Adafruit Industries. Pro trinket 5v 16mhz. https://www.adafruit.com/product/2000.

[IoTSV17] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. *Constrained Synthesis from Component Libraries*, pages 92–110. Springer International Publishing, Cham, 2017.

[JG96] Sakait Jain and Hae Chang Gea. PCB Layout Design Using a Genetic Algorithm. *Journal of Electronic Packaging*, 118(1):11–15, 03 1996.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KL16] Seong-Won Kim and Youngjun Lee. The effect of robot programming education on attitudes towards robots. *Indian Journal of Science and Technology*, 9(24), 2016.

[KLL14] Andrew B. Kahng, Hyein Lee, and Jiajia Li. Horizontal benchmark extension for improved assessment of physical cad research. *Proceedings of the 24th edition of the great lakes symposium on VLSI - GLSVLSI 14*, 2014.

[LCC+15] Jingwei Lu, Pengwen Chen, Chin-Chih Chang, Lu Sha, Dennis Jen-Hsin Huang, Chin-Chi Teng, and Chung-Kuan Cheng. eplace: Electrostatics-based placement using fast fourier transform and nesterov's method. *ACM Trans. Des. Autom. Electron. Syst.*, 20(2):17:1–17:34, March 2015.

[LHAZ15] Honghua Li, Ruizhen Hu, Ibraheem Alhashim, and Hao Zhang. Foldabilizing furniture. *ACM Transactions on Graphics*, 34(4), 2015.

[LRC+20] 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. *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020.

[LV09] Yoad Lustig and Moshe Y. Vardi. Synthesis from component libraries. *Foundations of Software Science and Computational Structures Lecture Notes in Computer Science*, page 395–409, 2009.

[MGS19] Devon J. Merrill, Jorge Garza, and Steven Swanson. Echidna: Mixed-domain computational implementation via decision trees. *Proceedings of the ACM Symposium on Computational Fabrication - SCF 19*, 2019.

[MS19] Devon J. Merrill and Steven Swanson. Reducing instructor workload in an introductory robotics course via computational design. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education - SIGCSE 19*, 2019.

[MSP+17] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[MTN+15] Vittorio Megaro, Bernhard Thomaszewski, Maurizio Nitti, Otmar Hilliges, Markus Gross, and Stelian Coros. Interactive design of 3d-printable robotic creatures. *ACM Transactions on Graphics*, 34(6):1–9, 2015.

[OW06] M. M. Ozdal and M. D. F. Wong. Algorithmic study of single-layer bus routing for high-speed boards. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):490–503, March 2006.

[PR90] A. Pneuli and R. Rosner. Distributed reactive systems are hard to synthesize. *Proceedings 31st Annual Symposium on Foundations of Computer Science*, 1990.

[RC00] M. Rosenblatt and H. Choset. Designing and implementing hands-on robotics labs. *IEEE Intelligent Systems*, 15(6):32–39, 2000.

[RLI+17a] 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.

[RLI+17b] Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. Turning coders into makers. *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication - SCF 17*, 2017.

[RLI+17c] Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. Turning coders into makers: The promise of embedded design generation. *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication - SCF 17*, 2017.

[SFJ+17] Peng Song, Chi-Wing Fu, Yueming Jin, Hongfei Xu, Ligang Liu, Pheng-Ann Heng, and Daniel Cohen-Or. Reconfigurable interlocking furniture. *ACM Transactions on Graphics*, 36(6):1–14, 2017.

[SJ07] P. Spindler and F. M. Johannes. Fast and accurate routing demand estimation for efficient routability-driven placement. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, April 2007.

[SJDD93] D. Staepelaere, J. Jue, T. Dayan, and W. W. . Dai. Surf: rubber-band routing system for multichip modules. *IEEE Design Test of Computers*, 10(4):18–26, Dec 1993.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM SIGPLAN Notices*, 41(11):404, 2006.

[Sou78] J. Soukup. Fast maze router. *15th Design Automation Conference*, 1978.

[SSL+14] Adriana Schulz, Ariel Shamir, David IW Levin, Pitchaya Sitthi-Amorn, and Wojciech Matusik. Design and fabrication by example. *ACM Transactions on Graphics (TOG)*, 33(4):62, 2014.

[SYN+15] Daniel Saakes, Hui-Shyong Yeo, Seung-Tak Noh, Gyeol Han, and Woontack Woo. Mirror mirror: an on-body clothing design system. *SIGGRAPH 2015*, 2015.

[WHHF18] David Weintrop, Alexandria K. Hansen, Danielle B. Harlow, and Diana Franklin. Starting from scratch: Outcomes of early computer science learning experiences and implications for what comes next. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, pages 142–150, New York, NY, USA, 2018. ACM.

[YW10] Tan Yan and Martin D. F. Wong. Recent research development in pcb layout. *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2010.

[ZAC+17] Ran Zhang, Thomas Auzinger, Duygu Ceylan, Wilmot Li, and Bernd Bickel. Functionality-aware retargeting of mechanisms to 3d shapes. *ACM Transactions on Graphics*, 36(4):1–13, 2017.