Human-Centered Circuit Board Design With Flexible Levels of Abstraction and Ambiguity

Richard Lin

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-259
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-259.html

December 17, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Human-Centered Circuit Board Design With Flexible Levels of Abstraction and Ambiguity

by

Richard Cheng Lin

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Björn Hartmann, Co-chair
Professor Elad Alon, Co-chair
Professor Kimiko Ryokai

Fall 2021

Human-Centered Circuit Board Design With Flexible Levels of Abstraction and Ambiguity

Copyright 2021
by
Richard Cheng Lin

Abstract

Human-Centered Circuit Board Design With Flexible Levels of Abstraction and Ambiguity

by

Richard Cheng Lin

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Björn Hartmann, Co-chair

Professor Elad Alon, Co-chair

Printed Circuit Board (PCB) design tools are critical in enabling users to build non-trivial electronics devices. However, mainstream tools work at the lowest level of design — schematics and individual components — which can make design difficult for beginners and tedious for experts.

In this work, we start by examining current board design practices to learn how a variety of users approach the process, what tools they use, what works well, and where things are lacking. From those results, we focus on building tools to support the system architecture level of design — a representation that we believe captures the most important design decisions without getting mired in the details.

Taking inspiration from software engineering and chip design languages, we implement a hardware construction language (HCL) that supports users at multiple levels of abstraction (from system to subcircuits), enables mixed-ambiguity design through abstract components (such as a generic resistor instead of a particular part number), and encodes the methodology for subcircuit design instead of static instances (for example, automatically sizing a resistor in a LED circuit). To support users working with this very different interface to board design, we also build an integrated development environment (IDE) plugin that adds a linked graphical view and editor to the standard textual code editor.

Small but in-depth user studies on these tools indicate that this approach can provide benefits over mainstream flows, but also suggests avenues for continued work.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, my deepest thanks to my advisors Björn Hartmann and Elad Alon for supporting my journey through grad school. Your input – whether in designing and implementing systems, writing clear and concise papers and presentations, or in general being a researcher – has been invaluable in helping me get where I am today.

Thanks additionally to committee members Kimiko Ryokai and Prabal Dutta for providing additional perspectives and feedback that have helped strengthen this work. Furthermore, I'm deeply grateful to both Björn and Prabal for taking me on, adopting this board HDL project in the early years, and giving me the opportunity to build all this out.

I'd also like to thank James Demmel for both taking me on as an undergraduate researcher and for serving as my first year advisor – and critically, giving me the freedom to explore in that first year, which led me to hardware design, hardware description languages, and ultimately this project.

This project has been interdisciplinary and collaborative, and I've been fortunate to work with fellow grad student Rohit Ramesh, who shares a similar interest in electronics design and has a complementary formal methods and theory focus. His work on the core models provided the foundation that ultimately enables the design tools and interactions in this work. Continuing discussions will hopefully improve the power and expressiveness of these core models while keeping everything consistent, usable, and amenable to synthesis.

Throughout the years, many undergraduate researchers have also contributed to various parts of this project according to their interests. Their contributions have been helpful in getting systems built and evaluated, and many are still in the modern codebase. In particular, thanks to Sharan Satish Kumar for investigating Python GUI frameworks, Yulie Park for building the KiCad netlister, Connie Chi for investigating datasheet parsing (which turned out to be fundamentally hard), Nikhil Jain and Ryan Nuqui for exploring the underlying model and implementing IDE features, Josephine Koe and Tengis Dashmunkh for building parts libraries and generators, and Leena Elzeiny for making core compiler improvements. And a huge thanks to all who pre-tested the systems and user study protocols, and found bugs so that our participants didn't.

None of this would have been possible without all the user study and interview participants who have provided their time and feedback. And additional thanks to the anonymous peer reviewers of the conference papers that made up this dissertation for their suggestions that have helped make this work better.

Beyond this project, I'd like to thank b-crew labmates Andrew Head, Forrest Huang, Bala Kumaravel, Eldon Schoop, James Smith, Jeremy Warner, and J.D. Zamfirescu, who have provided feedback, insight, support, and fun times. Further thanks to Paulos lab members Molly Nicholas, Sarah Sterman, Kevin Tian, and Cesar Torres, especially for organizing reading parties where we could find (and fix) deficiencies in our papers so that the conference peer reviewers didn't. And an additional shout-out to administrative staff Aleta Martinez

for making all kinds of logistics painless, including the dreaded scheduling of the qualifying exam.

And while my dissertation focuses on a board HDL rather than a chip HDL, I would like to thank all those who I've worked with as part of the ASPIRE and ADEPT labs. Chisel has been a major source of inspiration for this project, in the high level generator HDL concept, the lower-level implementation choices, and how to engineer large systems that don't topple. Shout-outs to fellow grad students Adam Izraelevitz, Jack Koenig, Vignesh Iyer, and Kevin Laeufer who I've worked the closest with as part of the Chisel and ChiselTest projects, but also many others on what was a huge effort. Additional shout-outs to the lab's amazing administrative staff Roxana Infante, Tami Chouteau, and Ria Briggs who have kept the lab running and planned events; our systems administrator Kostadin Ilov who kept our infrastructure running; and staff engineers Jim Lawson and Chick Markley for dealing with the less-glamorous-yet-absolutely-critical day-to-day operations of the Chisel codebase.

No less important are the departmental staff Shirley Salanio and Jean Nguyen, who have been invaluable in helping navigate the administrative side of grad school as well as supporting me through more difficult times.

Beyond grad school, the solar car project team (CalSol) has been a major part of both my undergraduate and graduate years. Over my twelve years on the team, I've been able to engage in hands-on and interdisciplinary work, race solar cars across both the United States and Australia, and have the fortune of working with many amazing people. While there's too many to list comprehensively, I'd like to acknowledge fellow grad students members Derek Chou and Jean-Étienne Tremblay, and our milestone project of building an FPGA-based active battery balancing system within a month in 2018. Additional shout-outs to longtime members Tristan Lall, Byron Hopps, and Devan Lai, who have stayed involved past their undergraduate years and have been an invaluable source of expertise.

The COVID years were particularly challenging to our team, and I'd like to give special recognition to co-program directors Andrew Wang and Anthony Zhou for pulling us through while providing the most humanistic leadership in recent memory, and engineering director Erik Francis for his tireless dedication which was crucial to getting cars done and onto the racetrack. Additional special thanks to my COVID-era housemates and teammates Geoffrey Ding and Alexander Zerkle for being awesome people, making lockdown not all that bad, and putting up with (and contributing to) all the inanimate ducks, plush or otherwise.

CalSol's electrical sub-team (and by extension, some example boards in the dissertation like the datalogger in Figure 5.3) has heavily relied on the Supernode makerspace in Cory Hall, and I'd like to thank our departmental staff contacts David Au and Jeremy Kramer for keeping the place available as an open and student-run makerspace. I'd also like acknowledge my fellow student admins over the years Derek Chou, Calvin Li, Nick Firmani, Roy Tu, Allan Zhao, Lam Nguyen, Alicia Auduong, Avinash Jois, Harrison Zheng, Olivia Hsu, Dinesh Parimi, Kevin Zheng, Eric Deck, and Sylvia Jin, all of whom have helped keep the place running by contributing to new user training, staffing, equipment deployment and maintenance, and all the mundane-but-necessary upkeep like cleaning.

It's been an amazing eight years in grad school and twelve years in Berkeley, and I'm sure I've probably forgotten someone above - but thanks to you as well for being part of my time here.

This work was supported in part by NSF awards CNS 1505728, IIS 1149799, CNS 1505773, and CNS 1822332; DARPA contract FA8750-20-C-0156 (SDCPS); Synergy: Collaborative: CPS-Security: End-to-End Security for the Internet of Things; the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; the Paul and Judy Gray Alumni Presidential Chair in Engineering Excellence; ADEPT Lab industrial sponsor Intel; ADEPT Lab affiliates Google, Siemens, and SK Hynix; Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849; and the Camozzi Group via the iCyPhy consortium. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Chapter 1

Introduction

Electronics devices are all around us and play an increasingly large part in our lives - from the things that we directly interact with like smartphones and game pads, to those that quietly operate in the background like thermostats and internet-connected sensors. Almost universally, electronics are built on printed circuit boards (PCBs), which in turn are designed with electronics design automation (EDA) tools. Yet, the circuit design aspect supported by these tools still revolve around drawing schematics – largely unchanged over the past several decades, despite significant gains in computational power and the shifting context of design as do-it-yourself and making have become more popular and accessible.

Although schematics provide a very direct mapping to the components on the board and allows a fine degree of control, this low level of design is not without tradeoffs. For novices and hobbyists, these tools require an understanding of electrical engineering details in order to build a working circuit and board. For experts and professionals, the low-level representation can be tedious and inefficient, requiring every degree of freedom to be resolved even when there is a large acceptable design space.

1.1 Overview and Thesis

Overall, this dissertation aims to examine how to re-think circuit design tools to support modern expectations of automation, while minimizing tradeoffs in flexibility and control that higher level design tools often make. This work takes an interdisciplinary approach: not only looking at the electrical engineering aspect of the problem, but also taking inspiration from modern programming languages that enable a wide variety of users to build custom automations, and with a human-computer interaction framing to understand underlying user needs and ensure solutions remain usable.

As such, the thesis at the core of this work is:

> a hardware description language approach to board design can better support user design goals by providing a higher level of design and abstracting away lower-level tasks, while remaining usable and minimizing trade-offs in flexibility.

1.2 Contributions

Supporting that thesis, this dissertation makes the following contributions:

- A formative interview study with 15 participants to assess current practices and problems in board-level design, taking a holistic end-to-end ideas-to-device perspective without being limited to current EDA flows (Chapter 4). We found that a significant amount of design work happens before EDA tools come into play - and often without the help of software at all. By the time users get to schematic capture, much of the interesting and creative work has already been done, and schematic capture itself involves a lot of rote transcription.

- A concept for a hardware construction language (HCL) based system that address major issues with current flows revealed in the above formative study, and preliminary user feedback on mockups of this language and supporting tooling (Chapter 4). Though users indicated enthusiasm around the system architecture level of abstraction, they also expressed caution around issues such as dependence on and trust in libraries. We further discuss relevant design principles for future board tools based on our results.

- An implementation of the above system as a Python-embedded HCL, based around a hierarchy block model to scale across multiple levels of abstraction and supporting design re-use through user-definable libraries of blocks (Chapter 5). Blocks can be generators, containing executable code that constructs the block's sub-circuit from higher-level parameters (for example, calculating resistance given a voltage and current specification), while a type system and electronics model supports substitution and refinement (for example, whether a particular step-down converter subcircuit can be used in place of a generic voltage converter).

  We evaluate this system by designing several example boards in this system and with a small user study where participants built devices of their choice. Users were able to complete designs in our system and provided a wide range of feedback including suggestions for better tooling and areas of concern such as the higher learning curve.

- Supporting tooling for the above HCL in the form of an integrated development environment (IDE) to bridge the novel code approach with familiar schematic views, by augmenting the standard text editor with a block diagram visualization supporting schematic editor like actions (Chapter 6). As both views are linked – the visualization can be updated from code edits, and code can be generated from schematic like actions – users can easily move between interfaces based on their own preferences and the task at hand.

  We evaluate this interface with a small user study in which participants complete a pre-defined project, and make generalized recommendations for similar tools based on the results. Although users differed in whether they preferred to produce HCL by

writing text or through graphical actions, the visualization still provides something for everyone, such as by acting as an intuitive reference for writing HCL.

1.3 Roadmap

These chapters form the rest of this dissertation:

Chapter 2: Background provides an overview of the workflow for modern board-level EDA tools, along with the historical context and the overarching class of creativity support tools.

Chapter 3: Related Work looks at related work, both in novel board-level design tools and empirical studies, as well as in adjacent fields such as general electronics debugging, chip-level digital logic design, and programming tools. While recent work on board-level design tools also aim to raise the level of design with libraries of blocks, they often neither support user-defined libraries nor address how libraries are created, which we believe is a crucial step for generalizability and practical usage. To address this, we take inspiration from novel chip-level hardware description languages, which successfully span multiple levels of abstraction from library-level modules to entire chips while offering a high degree of automation.

Chapters 4, 5, and 6 describe the the novel work in this dissertation, and are summarized in more detail above in Section 1.2.

Chapter 7: Conclusion concludes the dissertation with a summary of contributions and findings and ideas for future work.

Chapter 8: Glossary provides a definition of selected terms and acronyms.

1.4 Statement of Prior Publication

This thesis is based on, and incorporates material from, these prior published works:

- Beyond Schematic Capture: Meaningful Abstractions for Better Electronics Design Tools [42] (CHI '19), co-authored with Rohit Ramesh, Antonio Iannopollo, Alberto Sangiovanni Vincentelli, Prabal Dutta, Elad Alon, and Björn Hartmann
- Polymorphic Blocks: Unifying High-level Specification and Low-level Control for Circuit Board Design [43] (UIST '20), co-authored with Rohit Ramesh, Connie Chi, Nikhil Jain, Ryan Nuqui, Prabal Dutta, and Björn Hartmann
- Opportunities and Challenges for Circuit Board Level Hardware Description Languages [41] (HATRA '20), co-authored with Björn Hartmann
- Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages [44] (UIST '21), co-authored with Rohit Ramesh, Nikhil Jain, Josephine Koe, Ryan Nuqui, Prabal Dutta, and Björn Hartmann

I am the first author on all these papers, but none of this would have been possible without the effort of all the co-authors, including my advisors Björn Hartmann and Elad Alon, graduate student colleagues Rohit Ramesh and Antonio Iannopollo, and all the undergraduates who have worked with us over the years.

Chapter 2

Background

Today, a wide range of users design circuit boards for a variety of reasons - spanning hobbyists and makers working on personal projects, to commercial teams working on complex products. The falling cost of production (especially for one-off prototypes), an active maker community, and beginner-friendly platforms like Arduino [4] have helped make electronics and board design more accessible. In mainstream practice, EDA tools for board design generally have two parts: *schematic capture*, where users draw the circuit, and *board layout*, where users place and route components from the schematic on a virtual board. This rest of this chapter provides a brief overview of mainstream board design tools, as well as principles for the larger class of creativity support tools.

A variety of board-level EDA tools exist for different user groups: for example KiCad [35] is open source; EAGLE [6] is largely geared towards hobbyists; while Altium [1], Cadence Allegro [12], and Mentor Xpedition [55] are geared towards professionals working on complex projects. This chapter only covers high-level concepts and notable features of these tools, but more in-depth tutorials and user guides are readily available online.

2.1 Schematic Capture

In graphical schematic capture tools such as the one shown in Figure 2.1, users create the schematic by placing schematic symbols (representing components - for example, the squiggly lines for a resistor) onto the schematic sheet, then drawing wires to connect their pins. Symbols for common parts are often available in libraries [40], such as bundled with the tool, provided by the component manufacturer, or created by the community. The circuits themselves can be designed by referencing well-known circuits (such as RC low-pass filters and various opamp circuits) and application schematics in manufacturer-provided datasheets.

While computerized schematics can theoretically be infinitely large, schematics are often drawn to fit on printable sheets. More complex designs often require multiple sheets, and there are two common organizational techniques:

- **A flat schematic** simply treats the contents of multiple sheets as if they were the

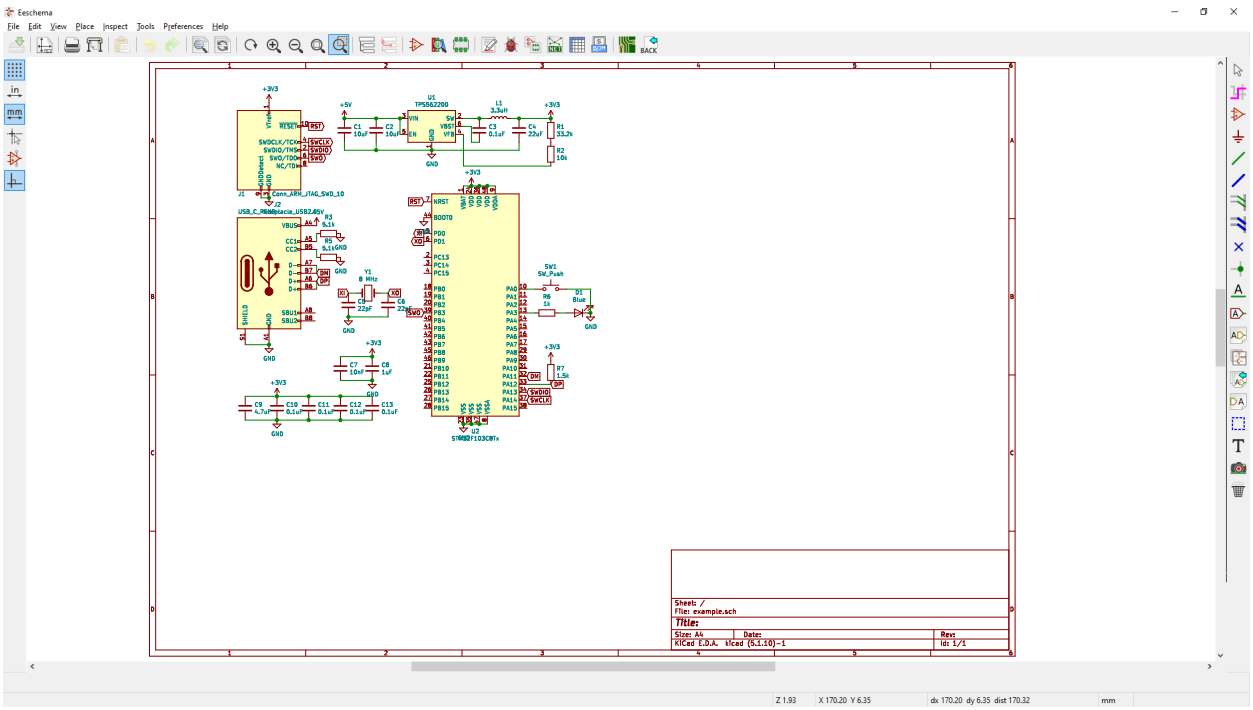

Figure 2.1: Example schematic in KiCad.

contents of one big schematic. Connections between sheets are handled with tunnels, where tunnels with the same name are connected together.

- **A hierarchical schematic** abstracts sheets of sub-circuits into a symbol that is treated like a component. This provides for abstraction and re-use in design, for example the same sub-circuit can be instantiated several times simply by duplicating the block.

While sub-sheets in hierarchical schematics can theoretically be shared, practical barriers to large scale direct re-use of design files include different file formats and tailoring subcircuits (for example, tuning component values) for each design. However, organizations may re-use internal schematic files [54].

Tools often support automated schematic checks with Electrical Rule Checks (ERC). Typically, pins are assigned some type (like Power Input and Power Output), and rules check for illegal connections between pins (for example, multiple Power Outputs driving a single wire, or pin that is not connected). This tends to be a pretty coarse system: for example the Power Output pin type does not further define voltages. Otherwise, schematic verification tends to be done by human peer review [53].

Tools also provide varying degrees of automation features, to ease repetitive tasks. The simplest example are various wizards, such as for creating a chip symbol with rows of pins on both sides. Slightly more complex is Altium's Smart Paste [2], which allows a copied selection to be pasted as different objects - for example, converting a group of selected pins to tunnels.

More complete tools may also offer some kind of programming interface for end-users, such as in the form of a scripting language or plugin API. While coding expertise is required to write custom scripts, pre-made scripts are often shared on the internet and can be used directly. More advanced plugin can integrate directly into the tool's GUI and only require installation.

Overall, however, these are still first and foremost schematic drawing tools and very limited in what they do to actually support system and circuit design.

2.2 Layout and Assembly

As schematics are an abstract representation of a circuit, they cannot be directly made into circuit boards. The core connectivity data from the schematic, such as in the form of a *netlist* consisting of a list of components and connections between their pins, can be imported into a board layout tool. An example netlist is shown in Figure 2.3. As in Figure 2.2, components are dumped on the virtual board and must then be placed and routed by the user. The layout can typically be incrementally updated from a changed schematic, without needing to restart layout.

Placement and routing can be fully manual, tool-assisted (such as with *online DRC*, which stops dragging a wire when it gets too close to another one), or fully automated (such as with *autorouters*). Some tools may support features for replicating layouts of similar subcircuits, such as Altium's rooms feature [63] or KiCad's Action Plugins [59]. There has also been much work on autorouting algorithms [21], but quality of results varies widely with factors such as choice of tool and design complexity.

Tools commonly perform two kinds of basic correctness checks. *Design rules check (DRC)* checks for physical manufacturability, such as minimum wire width, minimum spacing between wires, and minimum hole sizes. *Layout vs. schematic (LVS)* checks that the connectivity on the layout is the same as the schematic, catching unintentionally crossed wires or unconnected pins. Some tools also support more advanced layout-sensitive analyses, such as for signal integrity (mainly for high-speed signals) and power analysis (for voltage drop and power loss over the copper lines).

2.3 Historical Context

While modern tools are undoubtedly easier to use, the fundamental paradigms have been around for almost half a century. Computerized graphical schematic appeared in the liter-

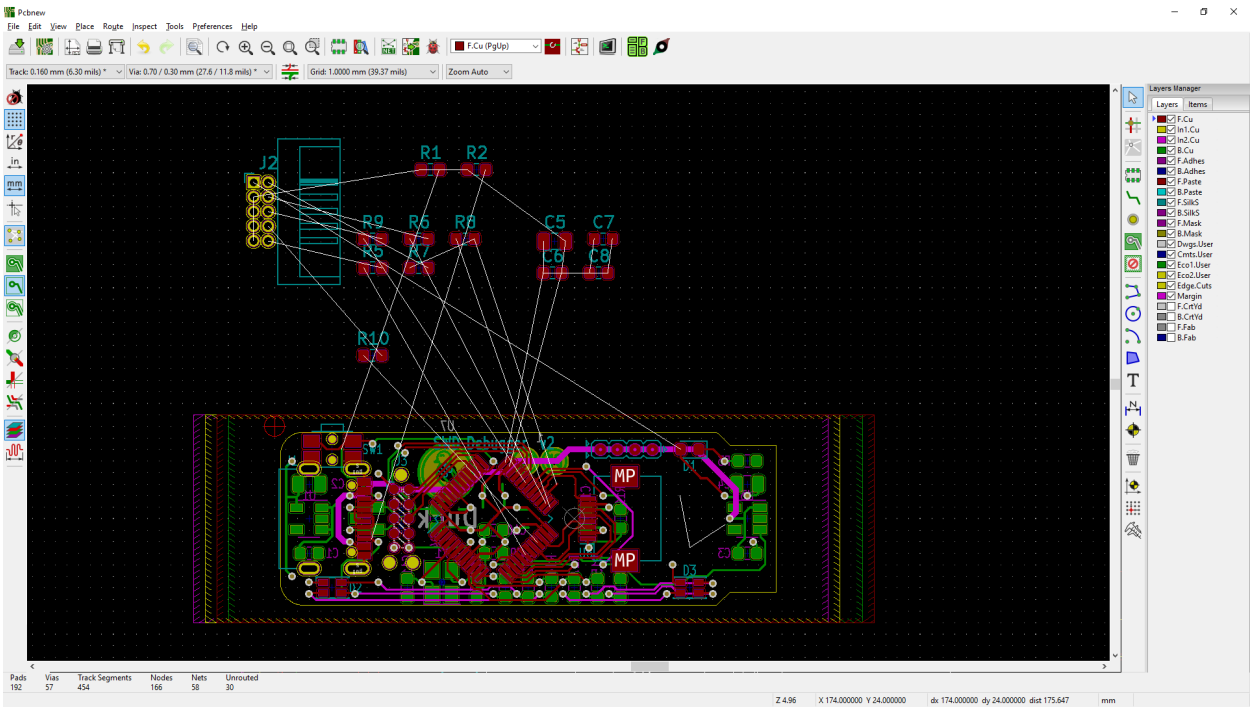

Figure 2.2: **Example layout in KiCad**. The components on the top have been freshly imported from a netlist generated from a schematic, while the components on the bottom have already been placed and routed.

ature as early as the 1970s, as a response to the awkwardness of punch card or text based schematic entry. Matthews [48] criticized tools of the time as "operator-aided computers", and describes a system similar to today's mainstream schematic tools. Despite the requirement for more user time on then-expensive computing resources, this system both reduced the overall design time and eliminated an error-prone punch card step.

Shiraishi regarded interactive schematic drawing as time-consuming, and his ICAD/PCB system [69] instead digitizes hand-drawn schematics using pattern recognition techniques. However, it focused on logic circuits and required a standard schematic style.

Another common theme of EDA systems from this era was integration between the different steps that we take for granted in modern EDA tools, such schematic entry, layout, and simulation. Reasons included the time, cost, and potential for errors from manual translation [48, 8, 39].

Systems also explored methods for component placement and board routing. Varying degrees of automatic placement were part of many tools, while other systems provided interactive feedback on manual placements [48, 69]. Autorouting was also a research theme

```
(components
  (comp (ref U2)
    (value STM32F103C8Tx)
    (footprint Package_QFP:LQFP-48_7x7mm_P0.5mm)
    ...)
  (comp (ref J2)
    (value USB_C_Receptacle_USB2.0)
    (footprint Connector_USB:USB_C_Receptacle_BSC_A40-00119)
    ...)
  (comp (ref U1)
    (value TPS562200)
    (footprint Package_TO_SOT_SMD:SOT-23-6)
    ...)
  ...)
(nets
  (net (code 6) (name +3V3)
    (node (ref U2) (pin 24))
    (node (ref U2) (pin 48))
    (node (ref U2) (pin 9))
    (node (ref U2) (pin 36))
    (node (ref U2) (pin 1))
    ...)
  (net (code 10) (name DM)
    (node (ref J2) (pin B7))
    (node (ref J2) (pin A7))
    (node (ref U2) (pin 32)))
  (net (code 11) (name DP)
    (node (ref J2) (pin B6))
    (node (ref J2) (pin A6))
    (node (ref U2) (pin 33))
    ...)
  ...)
```

Figure 2.3: **Example netlist snippet generated from a KiCad schematic**, showing the components, their properties (value or part number and footprint), and the nets (connections) their pins are a part of.

[39, 8, 48, 47], typically in conjunction with interactive manual routing.

Rager and Weiner [64] did an in-depth study on dense board layouts, recommending an interactive system where a human guides automated processes, but is provided with powerful assistive tools when lower-level manual intervention is needed.

2.4 Creativity Support Tools

EDA tools are part of the larger class of creativity support tools, which have received attention in the HCI literature with topics ranging from theoretical foundation, observations, and suggestions. In particular, Shneiderman et al. [70, 72, 71] and Resnick et al.[67] identifies several design principles, including supporting exploratory search, collaboration, and "low thresholds, high ceilings, and wide walls" (easy for novices while powerful for experts). Furthermore, meaningful evaluation strategies for creativity support tools are less clear than for traditional productivity tools with their controlled studies and standardized measures. This dissertation combines these high-level principles with domain knowledge and user feedback to build tools that aim to better support users in building electronics.

Chapter 3

Related Work

In recent years, there has been a significant body of work on electronics, especially for non-professionals such as students and makers, and spanning from empirical studies to tools with a variety of underlying technical approaches. Although this dissertation mainly focuses on board-level electronics, this chapter also briefly reviews relevant work in chip design and programming tools that inspires some of the approaches taken.

3.1 Empirical Studies

The rise of the maker movement has spurred some recent work on examining how people actually work with electronics and the practical issues that arise. For example, Crossed Wires [11] examines mistakes by novices in prototyping a physical computing system onto a breadboard, observing mistakes in both the hardware construction, software development, and the interface between the two. Mellis [52] observed novices over the course of a six week workshop as they were taught embedded design and ultimately built printed circuit boards. While it is possibly the most comprehensive examination involving modern EDA tools and practices so far, the focus on novices sets a low complexity ceiling.

However, there does not seem to be much work on the more professional and advanced users side. While tool vendors likely conduct usability studies, the results generally are not available to the academic community. This poses challenges for reimagining design tools that can be useful throughout the design complexity spectrum, instead of narrowly tailored for novices. As such, we start by conducting a study of current design practices for a range of users.

3.2 Electronics Design

While mainstream design practice has largely settled on schematics, novel work provides more automation such as by synthesizing the circuit from a specification, or by raising the level of design with pre-made modules.

Synthesis Approaches

Synthesis approaches take a variety of inputs. For example, Trigger-Action-Circuits [3], takes input specified at a behavioral and dataflow level – what the circuit is supposed to do – and produces a breadboarding diagram. On the other hand, circuito.io [16] and EDASolver [19] allows users to specify the design as a list of peripherals connected to a microcontroller. However, technical details are scarce: only Trigger-Action-Circuits had a corresponding publication, which focused more on interactions and an user study. None emphasize user-defined parts, which limits users to the parts library supplied by the tool vendor.

Embedded Design Generation (EDG) [65] and Echidna [57, 58] are similar, allowing users to specify a partial design such as sensors and motors, then automatically completes the design through interface-driven synthesis, for example by adding power systems and compute elements. While functionally similar, these systems have different underlying technical approaches: EDG translates the circuit design problem and passes it to a SAT solver, while Echidna is based on heuristic-assisted tree search.

Synthesis approaches have also been applied to other domains, such as for aircraft power systems [62]. However, it structures the problem as a collection of equations specified by the user, rather than the more user-friendly domain-specific abstractions provided by EDG and Echidna. While this dissertation does not address synthesis, it does continue work on domain-specific formal models for electronics that may support synthesis tools in the future.

Model-driven synthesis aside, AutoFritz [45] extends a schematic-like design system with data-driven circuit autocomplete suggestions. Such an approach makes different trade-offs: while model-driven tools require the (potentially tedious) creation of libraries of parts, data-driven tools can mine existing schematics. However, without an understanding of electronics, they may be more liable to make mistakes and may not be able to generalize past their dataset. While suitable for a recommendation system where the human still ultimately controls low-level details, this may be a significant limitation for higher-level design.

Although highly automated schematic tools do not appear to be in common use today, there are many existing tools that can synthesize sub-circuits from specifications. For example, TI's WEBENCH [32] automatically selects power conversion chips from their product line given high-level parameters (such as input and output voltage) and provides a full subcircuit around it. Online calculators are also available for well-known circuits, such as resistor pickers for resistive dividers [31]. While very useful, these are not integrated into schematic tools and still require the user to stitch generated outputs into the final designs and ensure consistency in the face of changes. This dissertation looks at an integrated design system which can automatically invoke arbitrary sub-circuit generators.

Module-based Design

In contrast to synthesis tools which automate decision-making, module-based design keeps users in control and instead speed up design by working at the level of modules instead of components. A module contains some subcircuit, much like hierarchy blocks in mod-

ern schematic tools, but is otherwise a black box and do not allow the user to modify its implementation.

Recent commercial tools combine module-based circuit design with a layout view: instead of placing abstract component on a schematic sheet, users place module bounding boxes on a board view. Connections between components are still abstract (they do not physically correlate to circuit board trace layouts), so board layout and routing still happen separately. Examples include Sparkfun À La Carte [75] and Geppetto [25], but details have not been published, so their internal models, library creation process, and algorithms are not known.

MorphSensor [88] also makes use of module-based design with mixed schematic and physical views, but focuses more on the novel 3D design view for flexible PCBs. Circuit design is out of scope of that tool, and it instead imports connectivity data from files produced by mainstream schematic editors.

Electronics Models

Synthesis tools generally need some kind of electronics model to check parts for functionality and compatibility to produce correct designs. For example, this may include encoding voltage limits of parts, then checking it against the actual voltage on the power line it is connected to.

EDG [65] defines a mixed electronics model (modeling voltages and their limits, currents and their limits, and digital logic threshold voltages) and software model (modeling programming interfaces like buttons and LEDs) on ports and their connections. This dissertation builds upon the electronics model. Additionally, both Sparkfun À La Carte [75] and Geppetto [25] appear to have some model that limits incorrect connections, but again comprehensive technical details have not been published.

Design tools aside, Valydate [56] provides automated schematic checking. As it is also commercial work, technical details have not been published, but it appears to have some kind of electronics model and parts library that includes voltage and current limits.

Summary

Overall, one common characteristic of recent work, both commercial and academic, synthesis or otherwise, is the use of libraries of higher-level components. At the very least, this helps speed up design through design re-use, instead of needing the user to re-draw schematics from scratch. Many also incorporate some kind of electronics model more detailed than the coarse pin types in mainstream ERC, to automate checks or guard against user error. However, none of these projects focus on library creation by end users, a step necessary to achieve scalability and generality beyond what a team of tool developers could feasibly do. This dissertation attempts to combine a module-based design flow with an electronics model, in a tool that allows users to create both libraries and top-level boards within an unified interface.

3.3 Electronics Prototyping and Beyond

While boards are a relatively permanent and stable form for building electronics, prototyping may be preferred on more reconfigurable platforms like breadboards.

Breadboards and Prototyping

One thread of research has been to augment breadboards, especially as breadboards are commonly a starting point for students. These include measuring and visualizing voltages [18] and current flows [86], and detection of inserted components through active probing [85]. Bifröst [51] further combines code instrumentation and logic analyzer circuit tracing for examining the hardware-software boundary. CircuitStack [81], on the other hand, adds a printed connectivity layer to breadboards to avoid the time-consuming and error-prone process of wiring up breadboards.

A wide range of other work exists to support prototyping. One example is the use of software programmable components, such as for designing analog circuits [76] or through an augmented reality interface [36]. Augmented reality has also been used to overlay simulation data onto a physical device built with parts from a specialized kit [14]. Other projects support constructing circuits through step-by-step tutorials [83].

PCBs for Novices and Debugging

PCBs may still be useful for novices who have a design they want in a more stable form. Fritzing [37] helps by providing a breadboard view of a circuit as a conceptual bridge to the schematic view, but is still fundamentally a schematic drawing tool. The previously mentioned AutoFritz [45] builds on top of that with data-driven circuit autocomplete suggestions.

As PCBs don't always work the first time around, some work exists to help in the debugging process. For example, Pinpoint [77] automatically inserts test points onto a board and generates a corresponding bed-of-nails testharness. This can be used to probe signals and dynamically isolate subcircuits. BoardLab [23] takes a different approach, instead using a magnetic positioning system to link a probed point on a physical board to the schematic.

Scaling

While many tools focus on enabling novices, some work also examines challenges of scaling from one-of prototypes to production [34]. While the work in this dissertation does not specifically address production, aspects of the system that specify a design space such as substitutable parts may make it easier to optimize a design for production.

3.4 Hardware Description Languages

Although graphical schematics are mainstream for board design, hardware description languages (HDLs) dominate digital logic design for chips and can provide useful inspiration.

Chip HDLs

HDLs like Verilog and VHDL are common in the chip design space for defining digital logic. Generally, digital logic HDLs combine a *structural component*, which specifies hardware in terms of modules and connections, and a *behavioral component*, which specifies arithmetic and logic flows. Both languages are organized in terms of modules with ports, and like hierarchical schematics for boards, digital logic modules may be composed of other modules.

Verilog-AMS [29] and VHDL-AMS [15] provide analog and mixed-signal extensions on top of their base digital languages. Though they allow for modeling and simulation of circuit behavior, they are neither design nor synthesis languages.

Hardware construction languages (HCLs) that support *generators* are an evolution on the basic HDL. Instead of merely providing a textual format for describing a module, these encode the rules for constructing a family of modules from high-level parameters. Chip-level HCLs include Chisel [33] for digital hardware, and OASYS [27] and BAG [17] for analog hardware. One implementation strategy, as taken by Chisel, is providing hardware construction primitives (for example, instantiate module and connect ports) embedded in a general purpose programming language.

There is also a thread of work addressing the lengthy, often hours-long latency for chip tools to recompile from a design change – a bottleneck on productivity. Strategies include improving synthesis performance [82] and simulation performance [74, 68] through partitioning and incremental compilation. While board-level designs are orders of magnitude less complex than chip designs, similar techniques can also speed up board-level compilation.

Board HDLs

HDLs also exist for boards, with an early one being PHDL [61] which effectively is a schematic in textual format, but with modules and limited parameterization to promote design reuse. Unlike chip HDLs, PCB HDLs are purely structural, as it is unclear what behavioral abstractions can suit the wide space of PCB electronics. However, an HDL interface to the same schematic abstractions provides little more design automation than a graphical editor.

More recently, JITPCB [7] and SKiDL [73] brings generators to the PCB space by embedding circuit construction primitives in a general purpose programming language. This dissertation builds on top of that core idea, but augments it with more detailed electronics modeling to enable design support features like parts selection and correctness checks.

3.5 Programming Tools

Hardware description languages, and especially ones that enable generator constructs, are essentially programming languages — for which there is a large body of work including supporting tooling. For HDLs, and especially for PCB design where there is currently less familiarity with textual design methods, tools can help bridge the gap between current practice and powerful-but-novel techniques. This section reviews some of the most relevant work, which while not directly related to building PCBs, still serves as inspiration.

Live Programming

Live programming tools continuously run code as it is written by the user, and display the results to provide immediate feedback. These tools can help connect the more abstract representation of code to concrete examples [20], which users might find easier to work with or to help navigate execution contexts [50]. Prior empirical research [38] further indicates that simpler liveness tools were frequently used, and even small amounts of liveness can have disproportionate impact.

One critical aspect of live programming systems is the latency between code change and visible effects [66]. Prior work has discussed possible techniques to reduce latency [50, 79], such as predictively starting to compute results or returning speculative results. Compilation of HDLs can require non-negligible amounts of time, and tricks like speculative results can preserve a smooth interaction flow.

Graphical Tools

Beyond one-way visualization, work has also examined *output direct manipulation* – how these visual representations could be manipulated to write code in specific domains. One recent example is Sketch-n-Sketch [28], which provides a graphical editor for a custom language for 2D graphics. Other examples include such editors for string manipulation and diagramming [49].

However, perhaps the most often used similar class of tools are graphical user interface (GUI) builders, where users can define their GUI graphically through direct manipulation interactions such as drag-and-drop. These tools similarly aim to provide an interface that is closer to the domain than the equivalent GUI construction code they generate. Yet, the code generated often is not stylistically clean or even meant to be directly edited [87] – perhaps fine for a GUI with defined integration points and static structure, but less so for an HDL where programming the structure is the point.

Separately from graphical interfaces for writing textual code, there are also *visual programming languages* where the language itself is graphical. Examples include educational languages such as Scratch [46] which provides a blocks interface to an imperative programming language, and studies of its usage have shown some benefits [5]. Beyond education, visual languages have also seen practical use, most notably with LabVIEW [30] which fea-

tures a different, dataflow abstraction. While ideas from visual language approach could provide the capabilities of a generator HDL without any code, textual interfaces are currently the dominant and familiar way to program.

Chapter 4

Formative Studies

4.1 Introduction

Given the lack of recent research into how people actually design boards, diving straight into designing and building tools runs the risk of not actually understanding user needs and missing pain points. Therefore, we start by learning about current design flows, with the focus of discovering what approaches to better design tools are fruitful, and why.

We take a broadly-scoped and holistic approach of understanding the design flows from idea to physical device. This end-to-end investigation avoids being limited to assumptions baked into current tools, and tries to get at the fundamental goal of building devices. Crucially, this can also reveal steps that are under-served and where novel tools be impactful.

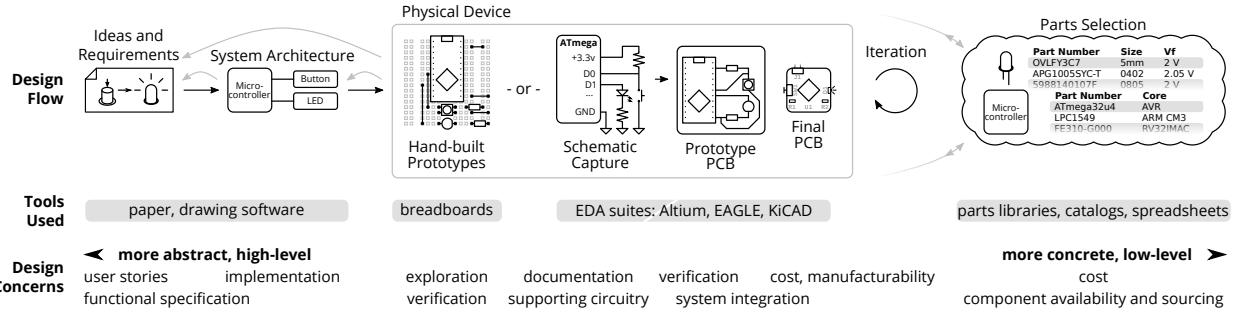

Figure 4.1: **The electronics design flow, as described by our participants.** Users start with an idea, refine that into a system architecture, and then iterate physical prototypes. Parts selection happens throughout the process. While certain steps require linear progression, iteration and revision of earlier stages also happen. Overall, EDA tools only support a small part of this process, and moving between steps was a major source of friction.

	Age	Motivations	EDA tool	Design discussed	Mockup study
P01	late 20s	school, hobby	EAGLE	LED board	Yes
P02	early 20s	school, hobby	EAGLE	analog feedback-controlled heater	Yes
P03	early 20s	school, job (startup)	EAGLE	Arduino motor controller	No
P04	early 20s	school, research, hobby	EAGLE	music recording system	Yes
P05	late 20s	research, side jobs, hobby	EAGLE	robotics	Yes
P06	early 30s	research, hobby	EAGLE	electrical muscle stimulation	No
P07	early 20s	school, hobby, job (industry engineer)	KiCad	IO controller	Yes
P08	early 30s	job (industry engineer), hobby	KiCad	educational kits	Yes
P09	mid 30s	research, school	EAGLE	educational blocks kit	Yes
P10	mid 30s	job (industry engineer), hobby	EAGLE	breakout board	Yes
P11	late 20s	job (research engineer), hobby	Altium	motherboards for chip tapeouts	Yes
P12	early 20s	job (industry engineer), hobby	Altium	power converter	Yes
P13	late 20s	research	EAGLE	embedded development board	No
P14	late 20s	job (industry engineer), hobby	DipTrace	debug adapter	No
P15	late 30s	job (industry engineer), hobby	Altium	general consumer electronics	No

Table 4.1: Summary of study participants.

For each step in the flow, we also delve into which tools (if any) are used, whether they work well, and where the pain points are. This deeper look reveals details of designers' thought processes, intermediate goals within their workflows, and their interactions and frustrations with existing tools.

We use what we learn to envision and mock up plausible, alternate tools that can better support these design flows. Then, by examining participant feedback of these prospective designs, we look more closely at what designers find valuable, and where future tool builders should focus.

The overall contributions consist of a formative interview study with 15 participants to assess current practices and problems (in Sections 4 and 5), followed by the design of a tool concept that addresses major issues (in Section 6), and ending with user feedback on a mockup of the tool concept (in Sections 7 and 8). This overall methodology follows the example of prior papers that combine formative studies with concept designs for better tools in other domains [60, 26].

4.2 Participants

We conducted an interview study with 15 participants (14 male), of which 10 returned for the follow-up mockup study. While small, this group covers a wide variety of skill levels, design types, and EDA tools used. Critically, both professional and hobby users are included. A summary of participants' backgrounds is shown in Table 4.1.

All participants are familiar with the design process from idea to PCB, and all but one have completed at least one full project consisting of all those steps.

We recruited participants using two methods: personal referrals (7 participants), and relevant email lists (8 participants) such as those of a local makerspace, university design courses, and student groups. While the only criteria was some experience building PCBs in EDA tools, we did not recruit those working on highly complex designs to avoid a long tail of specialized issues. Participants were compensated with a $20 gift card for each of the interview study and mockup study.

4.3 Interview Study: Methodology

Interviews were semi-structured and start with background information, including motivations, designs, and views of flow from idea to final device. Based on those responses, we then go into depth on each step in the flow, examining tools used, pain points, references used, and general suggestions or comments. Interviews averaged 90 minutes with a standard deviation of 29 minutes, and were conducted either in-person at the participant's workplace or through videoconference.

Utilizing the principles of contextual inquiry [9], we asked for an example design to ground discussions when possible. A majority of participants were able to do so, but some could not because of confidentiality and lost files. Instead, we asked them to either visualize their designs or bring up stock schematic and board layout images.

Interviews were conducted by one interviewer and audiotaped with the participant's consent. One researcher, experienced with board design and familiar with most of the tools discussed, then conducted an open coding phase over the transcriptions, and further grouped codes into related topics [84]. From these, we looked for themes that both had design implications for EDA tools and either had support among multiple participants or were notable outliers.

4.4 Interview Study: Findings

Participants provided rich data on their design flows, and how tools both did and did not support steps in those flows.

Design Flows

As shown in Figure 4.1, we broadly divide the design flow into these steps, in order: specification finding, system architecture development, and physical device iterations on a variety of media (including breadboards, milled PCBs, and commercially produced PCBs). Overall, each step incrementally refines the design to be more concrete, until finally a PCB can be produced. While there is a strict chain of dependencies between steps, designers regularly iterated and backtracked, especially in response to new information from testing and design.

Specification Finding

Determining the requirements and specifications for a device is a varied process that differed from user to user and from project to project. Specifications could capture a whole host of design goals including technical and functional requirements, user interactions, and aesthetic goals. These could be captured as drawings on a whiteboard, lists on documents and slide decks, or even a chip design that the system is built around. In many cases, these were living documents, with requirements and project scoping being a back-and-forth process where each edit forces many other changes down the line.

System Architecture Development

Specifications were then refined into a system architecture, represented as a block diagram. This serves as an intermediate step, translating from requirements into an implementation strategy.

The key distinguishing feature of this step is support for varying and mixed levels of abstraction.

> Each engineer will have what feels right for them. (P15)

Blocks in participants' architecture diagrams ranged from the generic ("accelerometer", "trigger circuit", or even just "sensors") to the specific (part numbers and subcircuit schematics). Three participants had examples that mixed abstractions on the same document, with some blocks being generic and others having part numbers. Some diagrams also indicated types of information flow between design elements such as communication buses or protocol information.

Drawings were overwhelmingly the most common representation: ten participants used either paper, whiteboards, or graphics software like PowerPoint and Visio. Schematic capture tools could also be used to produce nonfunctional diagrams, and two mentioned occasionally using EDA tools for this step. While digital tools gave designers powerful advantages including hyperlinking, cloud sharing, and backup, the unconstrained nature of drawings was most important:

> I feel very free to sketch in whatever language I want and whatever higher level I want. (P06)

Overall, participants generally enjoyed this step:

> I kind of like it. [...] It's a very creative area where somebody gives you requirements and you have the freedom to meet them however you see fit. [...] There's the creative freedom that you don't have once you get to the schematic and the layout. (P14)

Prototyping

Ten participants talked about a prototyping phase, which could be done with solderless breadboards, soldered protoboards, milled PCBs, or development boards and kits. Agility was a goal, which rapid prototyping machines could help with:

> I'm fortunate enough to have an LPKF [PCB mill] to mill the boards with. And that's been great. Usually the board goes through three or four revisions after soldering, so it's not just that, oh, I made one board and then it's done. (P03)

More generally, others also iterated on PCBs for their projects, with earlier boards acting as prototypes of the final design.

Prototypes were generally intended to validate some aspect of the design, though one participant also noted their value for exploring concepts and implementations. Validation was not limited to electrical functionality: mechanical characteristics, user feedback, and firmware development were also goals.

Schematic Capture

Schematic entry is where PCB design suites typically enter the design process.

Concerns here tended to be much lower-level, to the point where issues of schematic layout and readability were as common as those of circuit design and functionality. Participants noted the value of the schematic as a reference for later debugging or a document that should be shared with others. Aesthetics aside, messy designs could also conceal schematic errors or lead to bugs.

Mentions of manual transcription as part of the process were common – from either physical prototypes, or combining block diagrams with vendor-supplied reference schematics. While circuit designs in the abstract saw re-use, the inability to import data resulted in a time-consuming, tedious process. Yet, this was not completely devoid of designer input: reference designs may need to be adapted for the specific application through parts selection and component sizing. Quality and trust were also barriers to direct re-use: for example, worries about the quality of random Internet parts libraries or quirks in unofficial organization-wide reference designs.

Overall, attitudes about schematic entry were less positive:

> It's more of a necessarily evil. I wouldn't say it's a bad thing or a good thing, it's just like, I need to do this because otherwise I can't get my board. (P03)

Board Layout

Participant concerns during this phase were also low level and often related to the physical design and the final product: mechanical integration, signal integrity, manufacturability, and cost.

Despite both schematic capture and layout being part of the same EDA suite and schematic import being a common feature in layout tools, moving between schematic and layout was a notable source of friction. Five participants complained about the initial placement of components in layout:

> Altium kind of just barfs it out in a, not stacked on top of each other, but there's really not a lot of rhyme or reason. [...] It all seems pretty random. (P11)

Updating a layout after a schematic modification was also noted as problematic.

Participants also frequently consulted datasheets, placement rules, and routing guidelines during this step. While parts libraries and design guidelines could be shared between projects, layout re-use was rare. This was a result of limited tool support and projects needing customized layouts.

Parts Selection

Parts selection happened throughout the other stages of the design process. For example, critical parts may be specified on the block diagram, while common parts like resistors may not be picked until just before ordering.

Concerns varied widely. Eight participants mentioned optimizing for cost, while anther worked in a price-insensitive industry. Three also preferred parts that were immediately available in their makerspace or research lab. Otherwise, there was a long tail of other concerns, including hand-solderability, stocking, RoHS compliance, or avoiding vendors in organization-wide blacklists.

Overall, this phase could require significant manual work and was deceptively difficult:

> It's something that I find to be challenging and I think that people underestimate, [...] everyone's like, "eh, whatever, you're just buying stuff" and then they realize like "oh, actually, just buying stuff is not super easy". (P05)

Iteration

As alluded to throughout, many concerns do not fit purely within one design phase. For example, participants mentioned going back and forth between layout and schematic to optimize pin assignments for routing, or redesigning the schematic to work around unavailable parts.

In general, while later steps are dependent on the results of earlier steps, those results are not always locked down. As an extreme example, one participant recalls being told:

> Hey, you made this great device to guarantee these specs, but we really need this new part and it kind of breaks the spec that we gave you before. Deal with it. (P12)

One strategy participants used to deal with this was defensive design. This included defending against mistakes and errors, such as by inserting optional jumpers between subcircuits to allow modification or removal of connections, and defending against specification changes, such as by picking a microcontroller with a wide peripheral set for flexibility.

Use of Automation

Participants talked about their experiences using automation features provided by their tools. These features aim to minimize errors and ease tedious tasks, and fell into the broad categories of design verification and routing assistance.

Design Verification

EDA suites generally include electrical rules check (ERC), which checks schematics for common issues, and design rules check (DRC), which checks layouts for manufacturability.

ERC is commonly implemented by assigning pin classes (for example: input, output, or bidirectional) and defining a matrix of legal connections. Opinions were varied: six mentioned using this feature (all with caveats), while five specifically mentioned not using it. While electrical rules checking has utility in catching some simple mistakes like unconnected wires, the limitations were significant:

> A lot of false negatives. And false positives. Very few true positives. (P07)

On the other hand, no one mentioned skipping layout-versus-schematic or DRC, both of which are generally very accurate. Complaints were limited to bugs, like not catching split ground planes.

Routing Assistance

Only two participants reported using autorouting, all limited to simple designs. The general view was that the benefits were not worth the time costs of setting up the job properly or fixing poor results.

However, mixed-initiative, assistive routing features were well-received. These include online or interactive DRC, which does manufacturability checks on traces as they are placed, and smart routing tools like push-and-shove, which allow the trace being placed to intelligently displace existing traces.

> The auto routers are [terrible], the auto placements are [terrible]. It's a highly manual process. I like push and shove routing, those are great. (P12)

Tool Selection

We also asked participants about why they chose their particular EDA suite. Community effects dominated: their choices were influenced by the tools used by their friends or teams,

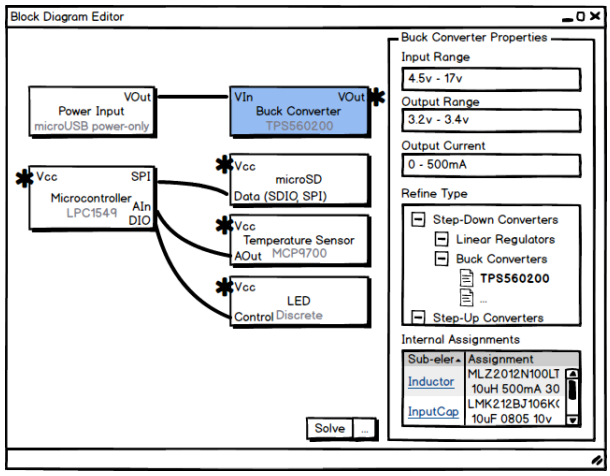

Figure 4.2: **Mockup of the block diagram interface**, showing the system architecture of the datalogger example design. The details pane on the right shows information on the selected buck converter: modeled operating parameters, selected and alternate implementations, and parts selections internal to the block. Showing operating parameters demonstrates how the system ensures design correctness.

the tools taught in class, and the existence of an ecosystem of tutorials and libraries. For those in industry, widespread usage was also important for compatibility with contractors and ease of hiring.

Those using or switching to KiCad noted the benefits of open-source software, such as lack of vendor lock-in, ease of sharing, and perpetual access to designs.

Summary

Overall, our main takeaway is that much of the interesting and creative work happens through a combination of system architecture and parts selection. Past that, schematic capture tends towards elaborating the system architecture by mashing in reference circuits, but the lack of design re-use results in a tedious and time-consuming process.

Links across steps are also major sources of friction. While converting a paper system architecture into a digital schematic is burdensome but unavoidable, moving between schematic, layout, and parts selection was just frustrating.

4.5 Concept Design

The core insight from the interviews, then, is for designers to work at the system architecture level. This higher level of abstraction captures the essential design intent without being mired

in implementation details. This section presents the high-level design and motivation, and the actual implementation is described in Chapters 5 and 6.

Note that this strategy has support in the creativity support tools literature [67], satisfying the principles of *supporting exploration* by reducing the design effort and *designing for designers* by being grounded in actual workflows.

Ambiguity in Block Diagrams

As schematics are fundamentally also block diagrams, the interactions and interfaces from today's schematic capture tools provide a solid and familiar starting point. In our use case, these block diagrams would also need to scale between multiple levels of abstraction. At the lowest level, blocks would still represent individual components, but at higher levels, blocks would be sub-circuits. While many tools already support this with the notion of *hierarchy blocks*, additional features are necessary to support the system architecture level of design.

Primarily, we need support for ambiguity. While current schematics must be fully defined down to the last part, system architecture diagrams in our interviews tended to encode minimalist design intent, leaving many decisions open. An example would be labeling a block generically as "accelerometer" instead of with a specific part number.

This ambiguity further provides opportunities for tools to automate the currently-manual and sometimes-tedious parts selection process. Recent work in synthesizing schematic from high-level specifications, including EDG [65] and Trigger-Action-Circuits [3] demonstrate the technological feasibility of this approach. As participants generally optimized for some criteria (commonly, but not always, cost) during their parts selection process, tools should also optimize for an user-defined objective function. Alternatively, the system could generate and display a shortlist of alternatives as in Trigger-Action-Circuits, though they reported mixed results with their novice participants.

An underlying constraint-based data model, as described in EDG, works well here. Types of components, like "accelerometer", would be just one aspect that could be constrained. More powerfully, such a system allows users to directly enter functional specifications, such as the minimum required bandwidth of said accelerometer. This also gracefully handles nonuniform ambiguity, which we observed from diagrams containing a mix of generic blocks as well as specific part numbers.

Supporting Libraries

Even an unambiguous high-level design still must be combined with implementations of used blocks to form a layout-ready schematic. However, our interviews show this to be a major issue: there usually aren't libraries of block implementations, and designers generally have to transcribe from datasheet reference circuits. Practical solutions must also incentivize the creation and sharing of re-usable libraries, either by making the process easier or by providing additional value for designers.

In any tool responsible for parts selection, libraries would need to model parts to a sufficient degree to check correctness of the entire system. As with EDG, electrical characteristics like absolute maximum ratings could be encoded in a block's constraints. This would automate some of the currently-manual checks mentioned by our participants, such as voltage and current compatibility. Furthermore, these checks could address one of the primary drawbacks of ERC, being more accurate than current pin-type based schemes.

One roadblock is that reference circuits often need to be customized for each application. In these cases, static hierarchy blocks would preclude any meaningful re-use. However, a generator methodology may be the solution: encoding the rules for generating a block implementation instead of fixed, static instances. As a simple example, a generator for a LED circuit would contain the logic to size the resistor given the LED current and voltages.

Generators built using hardware construction languages (HCLs) have been used for both chips [33] and PCBs [7]. Despite the limited exploration of their usability, HCL based generators show significant promise as an abstraction. We further note that, as is common in the chip industry, these textual representations can be applied to top-level designs as well. This may be highly advantageous in some cases: for example, instantiating large arrays of LEDs becomes trivial.

Allowing recursive ambiguity, where block implementations can contain further ambiguous blocks, can also be helpful. Reference schematics may be ambiguous: for example, even given a specific accelerometer, its reference circuit may include capacitors that do not have part numbers. This also aligns well with some observed design practices, where common passive components like resistors and capacitors are not chosen until ordering.

Interface Mockups

An example of an interface around these augmented block diagrams is shown in Figure 4.2. This shows a potential system architecture for a datalogger that records temperature data to a microSD card. Designers would be able to specify blocks that can range from the generic, such as the temperature sensor, to the specific, such as the microSD card socket. When a specific part number is needed for a generic block, the user could either allow the tool to automatically choose, or refine individual blocks from a list of compatible parts.

An example of the HCL approach is shown in Figure 4.3. The example design for a buck converter generator illustrates how current barriers to reuse in schematic tools are addressed. The parameters in the block interface specify what the subcircuit needs to do, while the constraints ensure design consistency by defining limits and how parameters propagate. The arbitrary code in the generator can then build customized subcircuits applicable in many different designs, for example by encoding the component sizing equations taken from the datasheet.

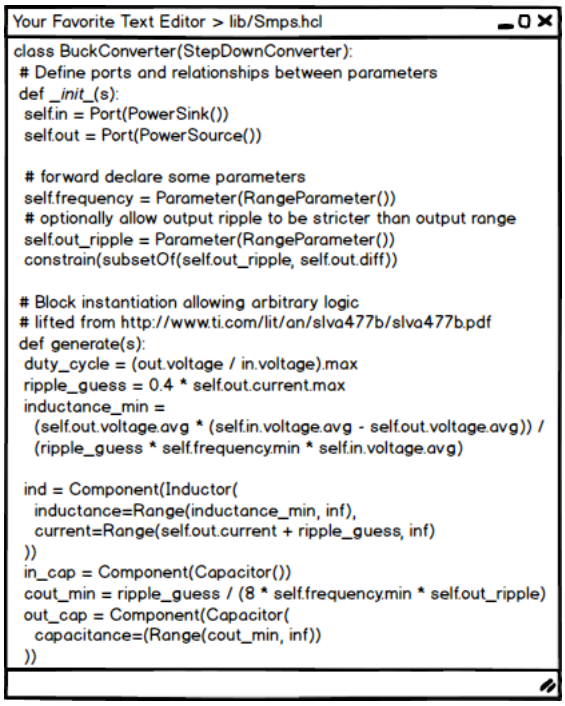

Figure 4.3: **Mockup of the hardware construction language interface**, showing the implementation of a buck converter generator. The first block of code, in `__init__`, defines the block interface: ports and constraints between parameters on those ports. The second block of code, in `generate`, contains the logic for instantiating sub-components once the block interface has been fully resolved. Here, this consists of equations transcribed from a buck converter datasheet.

4.6 Mockup Study: Methodology

As building the proposed design tool is a nontrivial engineering task, we felt it important to validate and refine the design first. In particular, we want to understand whether users would find this abstraction useful, and more importantly, their underlying reasoning and any perceived limitations.

To do so, we built clickthrough mockups of design flows through an example project spanning the two interfaces described above. These mockups allowed us to talk concretely with a visual aid that conveys similarities to conventional EDA tools, but without requiring the full system that any meaningful interactivity would require. We choose a datalogger as our example project because they have real-world applications, and balance easy participant comprehension with being complex enough for better tooling to be meaningful. The example system architecture, including the choice of blocks, are modeled off of observed diagrams.

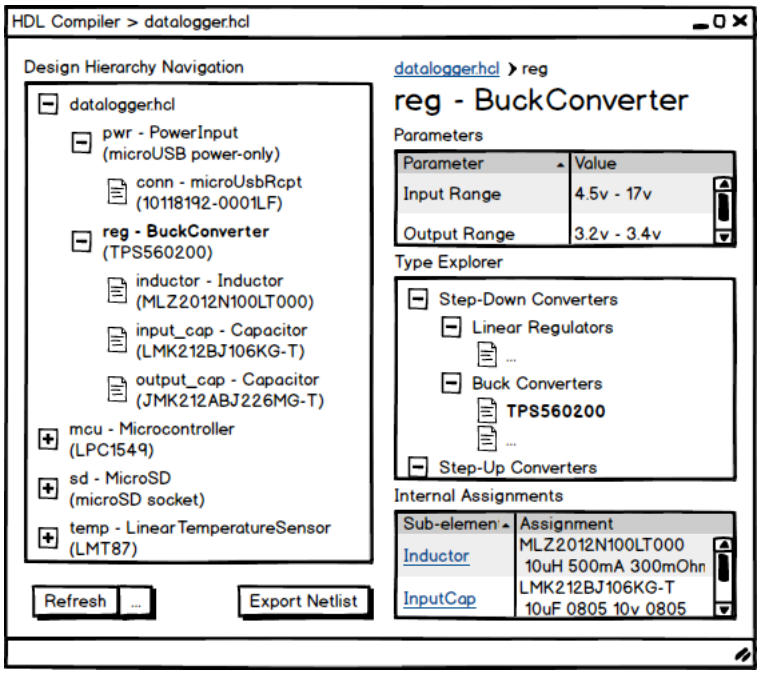

Figure 4.4: **Mockup of the hardware construction language compiler interface**. This provides information similar to the detail pane in the block diagram interface, but using a tree view for navigation in absence of block positioning data.

After some preliminaries, we presented our participants with an empty canvas in the block diagram interface. From there we walked participants through instantiating the high-level architecture from parts libraries before asking the tool to fully solve the design as shown Figure 4.2. The finalized design is equivalent to a full schematic, which we explain as being directly exportable to a layout tool like KiCad.

We then move into the HCL mockups, first showing a one-to-one transcription of the datalogger high-level design in code as a conceptual bridge. Further examples demonstrate the power of HCLs, first showing array instantiation of temperature sensors using a for loop, then showing the buck converter generator in Figure 4.3. We provide inspection into the solved design through the compiler interface in Figure 4.4. A tree view replaces the block diagram view as the HCL does not encode block placement and layout information, and the block properties pane becomes read-only.

We asked open-ended questions about advantages, disadvantages, and applicability, particularly compared against each other or conventional tools. We also asked about acceptable solver runtimes. For the HCL, we further asked about the utility of a hypothetical schematic visualization view and what kind of additional verification users would perform. The latter gets at notions of trust in the tool and libraries.

We purposefully used a sketch-like art style to key participants to focus more on design abstractions instead of UI specifics [24]. Additionally, by having participants compare-and-contrast between two interfaces, and asking for the rationale behind answers, we hope to reduce the effects of acquiescence bias [80]. This is especially relevant for those recruited through personal referrals.

Otherwise, the interview and analysis procedure were the same as the initial interview study. Ten of the original interview participants, as described in Table 4.1, returned for this follow-up study. Interviews averaged 46 minutes with a standard deviation of 14 minutes.

4.7 Mockup Study: Findings

Participants were generally enthusiastic about the system architecture level of abstraction for its ability to reduce manual work, but noted concerns about increased design automation.

Advantages

Automated design verification, essentially a more powerful ERC, was the most common advantage, mentioned by 5 participants. These automated checks reduce the chance of an uncaught error making it to fabrication while the encapsulation of datasheet parameters allowed replacement of the tedious manual verification process.

A related advantage, mentioned by three participants, was the integration of parts data into the main design flow:

> It does all of the parameter searching, and comes up with an appropriate part, which is what I do anyway just on Digi-Key, which doesn't have a very friendly user interface that is not tied closely into the design. (P11)

Designing at the system architecture level also provided advantages. Three participants noted the similarity to their existing processes, that this was part of their existing flow:

> I'm already generating some visual representation that's generated in software. If that can connect me to my other things, then I would really value that. (P05)

Not only would automated linkages from system architecture to schematic to layout save time, but it could eliminate mistakes during manual transcription. Errors where something is forgotten entirely during transcription were especially insidious, compared to design correctness issues which were more likely to be caught during inspection.

Finally, some participants brought up additional benefits with the HCL interface. Two talked about automated consistency throughout the design even as other parts change, and another noted that the methodology used for manual calculations was often not kept and must be rediscovered if needed later.

Limitations

The most common concern, mentioned by six participants, was a requirement for or dependence on quality libraries. Missing parts could either be invisible, especially for users solely relying on the system, or difficult to build.

All participants were inclined to share their libraries, but some noted limitations like concerns about competitiveness (especially if sharing uncommon parts), employer policy, and quality bars. Reasoning for this general attitude ranged from open-source philosophy to the practical benefits of community contributions. However, one participant expressed doubt about whether part manufacturers would contribute to a system that interoperates with competitors' parts.

Correctness was also a commonly cited criteria, especially since the tool introduces generative features:

> You're automating design here. That is, it's hard to do and it requires a lot of trust. (P07)

Discussions of trust in the overall tool were generally implicit: all participants mentioned doing some kind of verification on the generated output, from connectivity-based spot checks to comparing against datasheet specs. Sometimes, these statements would be qualified: one mentioned being thorough the first time, while two others suggested building trust by having the system show its work by generating report including the data sources and rules behind checks.

Trust in the libraries themselves was also a key part of trusting the tool. Of the five who talked about this, four mentioned trusting libraries from the part manufacturer or reputable organizations like Digi-Key and Adafruit. Trust in community libraries was mixed and based on a variety of heuristics, such as attention to detail and spot checks against datasheets. Community feedback was another aspect, including rating systems and indications of successful builds.

Finally, even the higher level of abstraction still requires nontrivial engineering knowledge:

> Beginners don't understand the difference between buck and boost and current and max and minimum footprint space versus component cost. (P10)

Blocks vs. HCL

While all participants were able to follow and understand the HCL examples, they also talked about trade-offs with the block diagram interface.

When asked about use cases, there were (predictably) mentions of parameterization and repetitive designs for the HCL. However, there were also mentions of its unsuitability for designs where its capabilities are not required, such as connectivity-driven or straightforward designs. One participant made the observation that:

> [The HCL] feels less kind of exploratory. It feels more like something I'd do if I already have sketched out something on paper, and then I need to figure out the components. [...] [The block diagram interface] feels almost like, to be a little bit abstract, it feels less serious, right? Because you're kind of working with these graphical representations, whereas this is code. (P04)

Participants were more critical of the HCL, with five mentioning the learning curve as a disadvantage. Four also mentioned the code representation as more difficult to work with, instead preferring a visual schematic. One described the HCL as completely unusable, though could still see value for large repetitive operations.

Participants had mixed feelings about textual interfaces. Two believed it would be faster, though one thought that even writing a for loop would be slower than operations in the block diagram interface. Another noted benefits of compatibility with version control tools and text editors.

Finally, one participant recognized that it is not an either-or situation, correctly noticing the possibility of using a GUI to define constraints. As both the block diagram interface and the HCL are built on top of the same data model, both could support constraints with the appropriate interface elements.

Running Time

Thoughts about acceptable solver running times largely fell into two broad groups: interactive, generally on the order of seconds, and batch, which spanned minutes to hours. An equal number of participants were in each group.

Those who wanted interactive runtimes pointed to the responsiveness of existing board design tools and modern websites as justification. They also suggested a modified version of the mockup flow to achieve these speeds, such as solving a subset of the design, or incrementally solving for design changes.

Those participants who were comfortable with batch processes cited both the time savings of automation as well as avoiding manual tedious work. Three mentioned benchmarking against manual processes, such as parts search. Another talked about the idea of active time and background time: while manual verification of a schematic requires active attention to the problem, one could attend to other tasks while waiting for the solver to complete in the background.

Summary

First, our results suggest that designers have two primary concerns when evaluating new tools: correctness and design effort. However, both must be evaluated holistically, across the entire design flow.

While the integration of parts data from datasheets provides a correctness advantage from a more powerful ERC, designing at the system architecture level also eliminates an

error-prone manual transcription step from paper. Both also provide an important speed advantage: the higher level of abstraction also frees users from worrying about details that a computer could solve, and block re-use reduces time spent reinventing the wheel. However, trust in both the system and libraries was a major concern, but could be earned through visibility into automated processes and community feedback mechanisms.

Second, reliable partial automation seems to be preferable to unreliable full automation. The initial interviews hint at this, with participants preferring assistive push-and-shove routing to fully automated routing. We see a similar trend in the mockup study, where participants were happy with the incrementally higher level of abstraction instead of pushing for, say, full synthesis from system requirements.

It may be useful to view the balance of user effort and system effort as a multi-dimensional trade-off, in terms of factors such as user time required, tediousness of tasks, expressiveness of abstractions, and feasibility of automation. For example, asking the user to further constrain a design to reduce the search space for runtime reasons may be a reasonable strategy.

Finally, based on the feedback from the mockup user study, we believe that our concept design is a good starting point for the designers of future tools.

4.8 Future Work

While this concept system address what we think are the highest-impact and lowest-hanging fruit in electronics design tools with our concept, both our interviews and principles for creativity support tools [67] suggest that these considerations are worth further investigation:

Iteration

Designers tended to iterate, both within the EDA suite, such as optimizing between schematic and layout, and through physical prototypes. Our concept only tangentially addresses iteration through refining constraints of the example design.

Open Interchange

We observed two design flows involving significant use of external tools (Inkscape and chip design suites), and there are likely to be more highly custom workflows. However, supporting these may be more of an implementation detail, by documenting file formats or exposing programming interfaces.

Community and Collaboration

Community effects were a large factor throughout both the initial and mockup interviews. Library quality and availability were emphasized in the mockup responses, but both may ultimately depend on the existence of a vibrant community. How to encourage the formation of, and sharing within, such a community may be as important as the tool design itself.

Enabling Library Creation

Tooling may also encourage creating libraries by partially automating turning datasheets into machine-readable data. For example, uConfig [13] is able to extract pinout data from PDFs for datasheets from certain vendors. Furthermore, tools might also parse the highly-regular electrical characteristics tables, and populate block model parameters.

Beyond the Schematic

While the concept primarily addresses schematic capture, the interviews also suggest improvements to other stages like layout. Additionally, there may be value in persisting ambiguity past schematic capture, such as to optimize for layout area.

Beyond Electronics

While this study was conducted in the context of PCB design tools, the findings and recommendations may be applicable to other domains. The ideas of incrementally raising the level of abstraction, specification and utilization of ambiguity in design, and eliminating tedious transcription work through better integration can generalize to any design domain. Knowing the limitations and requirements of these approaches, such as the need for trustworthy automation, will be important to building practical systems.

Our exploration comparing and combining visual interfaces and programming languages can also inform other design domains. One application may be in mechanical CAD, where parameterized parts could be defined in a powerful generator language, akin to OpenSCAD, and instantiated in a visual assembly-level interface.

4.9 Limitations

Ultimately, electronics design is a very broad field with many specialties. While we chose to address PCB design in general because of its ubiquity, we also realize that tools tailored for a particular subdomain may be more powerful.

Our goal of looking at entire PCB design flows also trades depth for breadth. An interview study meant a fairly high-level investigation and would be subject to participant recall limitations. However, our findings could form the basis for more targeted observational studies of particular steps.

Participants also tended to be younger (early 20s to late 30s), likely as a result of a majority of the mailing lists being university-affiliated. However, given the observation of similar design flows and repeated themes, we believe that we have at least found some interesting directions for future work. The data here could also form the basis for wider studies that use more scalable methods such as surveys.

Finally, while the feedback on our tool concept was largely positive, it is far from proof that it is useful, usable, or even feasible. Though we try to maintain an internal consistency

in our mockups and construct a representative design flow, they are only our best effort at imagining what such a tool would look like without actually building it. However, now knowing that we are not horribly off the mark, we move on to building and evaluating the system in the next chapter.

Chapter 5

Hardware Description Language

5.1 Introduction

In the previous chapter, we proposed a sketch for a tool that had the following features:

- supports the system architecture level of abstraction, typically block diagrams of high-level components
- also supports defining those high-level blocks, as re-usable libraries of subcircuits
- enables generators that automatically customize the implementation of those libraries, for example sizing components by higher-level parameters
- allows ambiguity in those block diagrams, for example with a generic resistor instead of a specific part number, at both the system architecture and library level

While we mocked up a hardware construction language (HCL) approach which automatically produces customized implementations of blocks based on higher-level specifications, in general programming concepts could support all of the features. Concepts from type hierarchies like polymorphism can encode the relationships between generic and specific parts to support ambiguity, while modern programming languages provide good support for abstraction.

Furthermore, like interfaces, classes, and inheritance in object-oriented programming, constructing electronics from blocks allows a division of labor: system designers can focus on high-level architecture while experienced engineers can build reusable libraries of blocks. Defining blocks as generators – executable code that translates high-level specifications into an implementation, e.g. a LED-resistor subcircuit that calculates resistance from input voltage – also helps separates interface from implementation and enables relative novices to leverage the knowledge of experts. Furthermore, block-level polymorphism – refining blocks with compatible subtypes, e.g., substituting a specific buck converter in place of an abstract voltage converter – balances high-level design with fine-grained control.

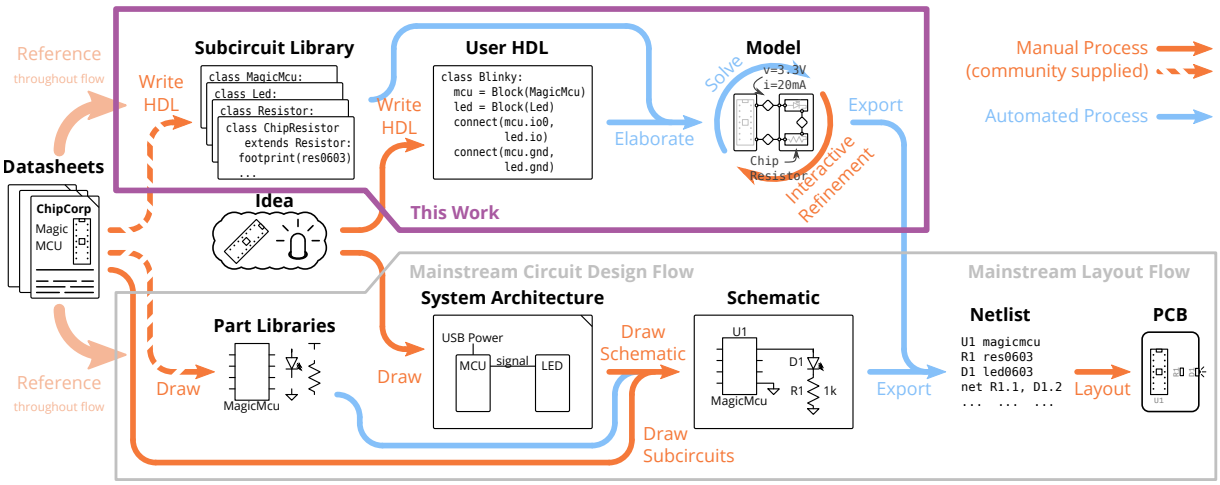

Figure 5.1: **The Polymorphic Circuit Blocks HCL design flow**. In the Polymorphic Circuit Blocks approach (purple box), board designers start by writing their system architecture in a hardware description language (HCL), which is then elaborated into a hierarchy block graph model and expanded using community libraries. That graph is then refined through interactive choices in a GUI and automatically propagated parameters are checked to ensure system correctness. The result can be exported to a netlist file, which can then be imported into a board design tool for layout. In contrast, mainstream tools (gray box) generally do not support system architecture level design, so such diagrams are often done with pen and paper. Furthermore, schematics are typically manually entered from the bottom-up using reference circuit diagrams from datasheets: while part libraries are available in mainstream tools, these are limited to single components and direct re-use of sub-circuit files is difficult and uncommon outside limited contexts.

We foresee an open-source community of engineers and designers, similar to that in the software world, where open collaboration and communication *lowers the threshold* of entry into electronics design even further, while preserving a *high ceiling* of complex designs, and offering *wide walls* of rapid exploration of design alternatives [67].

We implement this new model of circuit design in Polymorphic Circuit Blocks, an end-to-end system for authoring block diagrams. As summarized in Figure 5.1, users write designs in an HCL with the aid of subcircuit generator libraries, then interactively explore refinements to obtain a layout-ready circuit. An underlying electronics model checks designs using constraints such as operating voltages and currents. Supporting tooling in the form of a graphical *visualization and refinement interface* enables users to view their designs as block diagrams and specify refinements. This combination of HCL, electronics model, and user interface distinguishes our work from related work on purely textual PCB HCL efforts [7, 61] and high-level design tools that don't also allow lower-level control [3].

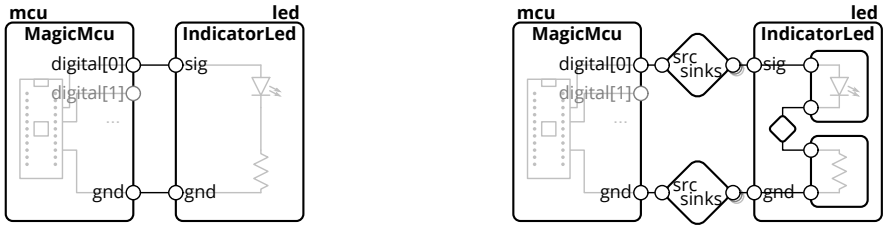

Figure 5.2: **Example of a blinky LED circuit in the Polymorphic Circuit Blocks HCL user-facing model (left) and internal model (right)**. The simplified user-facing model is presented at a single level of hierarchy, and contains just blocks (rectangles) with ports (circles) that can be connected. This largely follows representations in system architecture diagrams. The more detailed internal model spans multiple levels of abstraction by including internal hierarchy, and connections are described through links (diamonds).

Overall, we contribute a novel generator HCL for board-level circuit design, supporting tooling, and an accompanying evaluation. In the rest of this chapter, we expand on our hierarchy block diagram model, its expression in our HCL, the visualization and refinement interface, and important implementation choices. We then demonstrate our system's capabilities by building and testing two example embedded devices, and report on a remote study with three electrical engineers who designed PCBs of their own choice with our system.

5.2 System Design

In the Polymorphic Circuit Blocks workflow, as summarized in Figure 5.1, users start with an idea and a high-level system architecture in mind. They then translate that architecture into code written in our HCL, which is fundamentally organized around hierarchy block diagrams extended with generators, a type system, and an electronics model. Block level polymorphism and a class hierarchy allows the use of abstract blocks which can be refined later – for example, a generic voltage step-down block can be refined into a buck converter subcircuit based on a particular controller chip. Our visualization and refinement interface then allows the user to inspect their design and review these refinement choices. Finally, the user can export a netlist which can then be used to lay out the PCB in mainstream tools.

In the following sections, we will use a running example of a simple blinking LED circuit, shown in Figure 5.2, to introduce our model and the design workflow. However, it is important to emphasize that the system is designed to handle and produce more complex designs such as the data logger in Figure 5.3.

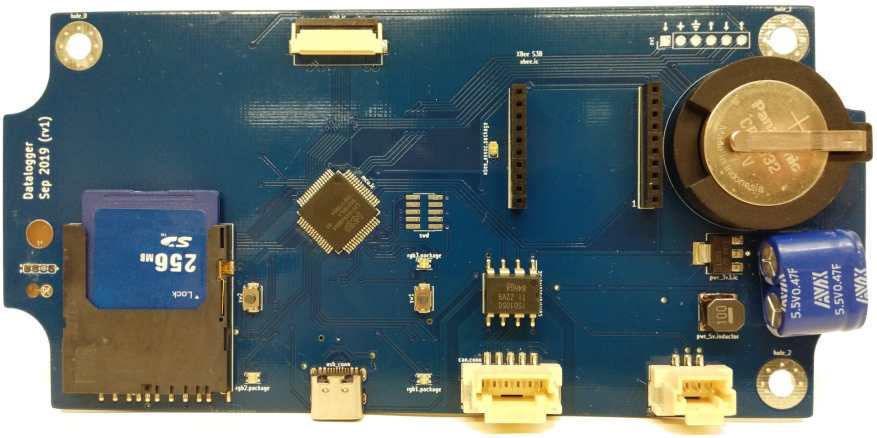

Figure 5.3: **A more complex example: the datalogger PCB designed in Polymorphic Circuit Blocks**, further explained in Section 5.4.

Block Diagram Model

Figure 5.2 shows our model's structure, extending the basic block diagram and consisting of blocks, ports, and links.

Blocks, shown as rectangles in figures, are elements of the circuit and the main construct users will interact with. They represent structures from single components like resistors and chips, to subcircuits like buck converters, to abstract functional blocks like voltage converters. Internally, they can have a set of parameters that define operating conditions along with constraints on those parameters.

Ports, shown as small circles in figures, represent the interface of blocks like power pins, GPIO pins, and signal busses. They can also contain parameters that describe properties of the interface, like maximum voltage ratings.

Links, shown as diamonds in figures, represent connections between ports, defining how ports connect and how parameters can propagate. They are structured much like blocks, containing ports, parameters, and constraints, however, block ports can only connect to link ports (and vice versa). As shown in Figure 5.2, links are simplified in the user-facing model as a connection between ports, and inferred into explicit objects in the internal model based on the types of connected ports.

This model improves on mainstream schematics by enabling electronics modeling and additional automated checks. However, more advanced automation and design support requires two notions of hierarchy: a structural hierarchy for encapsulation and a class hierarchy for abstraction.

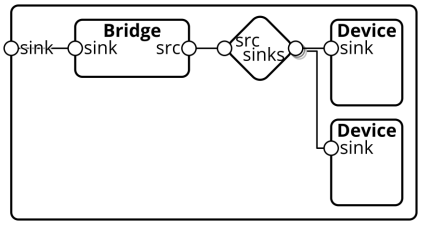

Figure 5.4: **Bridge example which adapts an externally facing port to an internally facing link**. The enclosing block has one externally facing sink port, which must feed two internal devices. A bridge "block" provides the internal version (in this case, the flipped version) of the externally facing port, with one port directly connected to the external port and its flipped version exposed to an internally facing link which multiple ports can then connect to.

Structural Hierarchy

Modern schematic editors already support a form of structural hierarchy via hierarchy blocks, which can be placed on the schematic like ordinary components but represent a sub-"sheet" or sub-circuit instead of a single component. This serves two purposes: as an organizational tool to make large schematics comprehensible, and as a re-use tool for replicating the same circuit block. We support the same concept, as shown in Figure 5.2 right where the `IndicatorLed` nests internal LED and resistor sub-blocks. Generators, discussed later, further increase the encapsulation power of these hierarchy blocks.

Hierarchy support requires cross-hierarchy additions to the block model. In the simplest case, a sub-block port can be directly exported to a containing block port, as shown with the `IndicatorLed`'s ports in Figure 5.2 right. In the more complex case, where multiple sub-block ports connect to a containing block port, a bridge is necessary. For example, as in Figure 5.4, a block might have a single power input feeding two sub-blocks, but a connection of only power inputs is nonsensical. A bridge would take the external facing port, a power input, and present a flipped internal version, a power source, to feed the sub-blocks. Bridges are structured as two-ported blocks, with one port being directly exported, and the other connecting to the internal link. We note that this structure preserves parameters and constraints of the internal blocks, allowing automatic management of lower-level invariants.

This hierarchy also extends to ports, which can be bundles of sub-ports, and links, which can be composed from sub-links. For example, the UART port is comprised of two digital ports TX and RX, and the UART link contains two digital links.

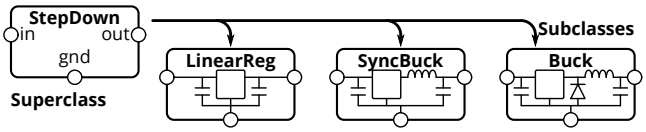

Figure 5.5: **Class hierarchy example with step-down voltage converters**. The abstract step-down converter has three subclasses, a linear regulator, a synchronous buck converter, and a buck converter. All these fulfill the step-down converter interface and functionality, and can be used in its place. This mechanism provides support for abstraction in our model.

Class Hierarchy

The main differentiator from mainstream schematic tools is the notion of a class hierarchy for blocks. While modern schematic tools require blocks to be specific parts, we would like designers to be able to, for example, instantiate and connect a "generic" LED at that (ambiguous) level of specificity. This is grounded in observations from our interviews in Chapter 4, where participants tend to start with high-level and weakly specified versions of designs, using general modules like power, sensing, and processing.

Our class hierarchy, borrowing inheritance concepts from object-oriented programming, defines how parts are functionally similar and can be used in place of one another. Superclasses provide higher-level interfaces, while subclasses have a is-a relationship with their superclass but can be more specific and concrete. For example, in Figure 5.5, a buck converter is a type of (and can be used in place of a) generic step down converter. This allows using blocks that are abstract – generic and without implementation – and delaying the precise specification until later.

These abstract parts also enable more generalizable library blocks, by allowing system designers control over elements nested within the structural hierarchy. For example, libraries can use the generic resistor block, preserving the choice of whether to use surface-mount or through-hole parts for the system designer who would have a better idea of application requirements.

We note that, differently from object-oriented programming, replacing a block with a subclass is not always safe. For example, a generic and abstract buck converter would not have current limits, but a concrete one made of physical components would. Block constraints enable automated checks to catch compatibility issues with selected refinements, but designer expertise is generally helpful in making high-level trade-offs.

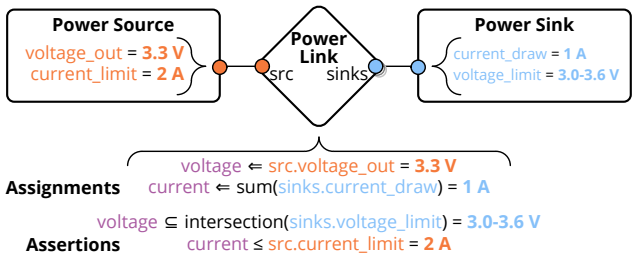

Figure 5.6: **Example of parameter propagation and checking in the Polymorphic Circuit Blocks model**, with a simplified constant-voltage link. Ports are defined with their physical properties: voltage output and current limits for sources, and current draw and voltage limits for sinks. These parameters "flow" through connected ports to links, which create aggregate parameters of connected ports such as the total current drawn and acceptable voltage range. Links also define assertions to check correctness properties.

Electronics Model and Libraries

An electronics layer on top of this basic structure models common pin types and part ratings. This consists of common links and their associated ports, such as a power link representing a constant-voltage power net, and power source and sink ports encoding output voltages, input currents, and their limits. We also define signal types, including digital ports modeling high and low voltage thresholds and analog ports modeling input and output impedances. Multi-wire protocols like SPI, USB, and CAN are modeled as bundles composed of the above single-wire primitives. As shown in Figure 5.6, we structured the model so that parameters on ports define properties of the device (e.g., voltage limits and current draw for a power sink), while links define properties of the net as derived from connected devices (e.g., voltage on a wire).

With this electronics model, we built a library of common blocks. Primitives include a resistor generator using the E24 series of preferred numbers, and inductor, capacitor, diode, and transistor generators created from parts tables. These primitives are defined with untyped passive ports, and are wrapped in higher-level library blocks (e.g., pull-up resistors for digital lines and decoupling capacitors for power lines) that translate port parameters to component parameters (e.g., pin voltage to rated voltage on a decoupling capacitor).

These library blocks provide significant design automation and integration. For example, a low-pass resistor-capacitor (RC) filter block calculates the resistance and capacitance based on a cutoff frequency and impedance specification, while a resistive divider block finds a pair of resistor values in the E24 series meeting the target ratio and output impedance. The library also includes application circuits of more specialized devices like microcontrollers, displays, and protocol converters, all of which can be directly dropped into the system architecture level HCL.

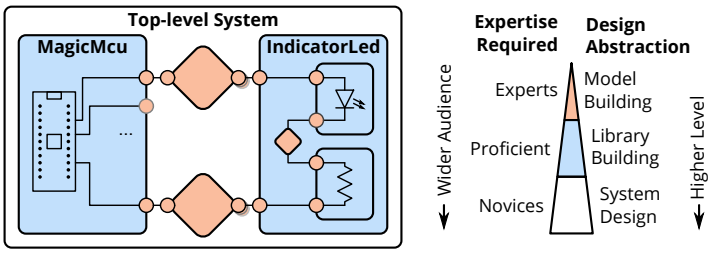

Figure 5.7: **Blinky circuit showing the different levels of design and where people of varying skill levels can fit in**. The overall system is intended to be accessible and useful to novices who can compose system-level designs using libraries, while the relatively fewer but more experienced electronics experts build those libraries of blocks and underlying port and link models.

```
1  class Blinky(Block):
2    def contents(self):
3      super().contents()
4      self.mcu = self.Block(Nucleo_F303k8())
5      self.led = self.Block(IndicatorLed())
6      self.connect(self.mcu.gnd, self.led.gnd)
7      self.connect(self.mcu.digital[0], self.led.io)
```

Figure 5.8: **Example code defining the Blinky circuit `Block`**. Within the block's `contents`, lines 4 and 5 instantiate the sub-blocks for the Nucleo microcontroller board and a discrete LED. Lines 6 and 7 then make the signal and ground connections.

The overall vision of the layers of our system and how different users interact with it is summarized in Figure 5.7.

Hardware Description Language

Taking inspiration from recent work on chip generators [33], we provide a generator HCL interface for authoring blocks. This programmatic construction of blocks captures the design methodology to construct a family of subcircuits, and separates interface from implementation by translating high-level inputs into internal parameters. For example, the LED-resistor generator calculates the resistor value given the input voltage.

As shown by the Blinky code example in Figure 5.8 (which describes the diagram in Figure 5.2 left), the HCL is a Python-embedded domain specific language, making use of its object-oriented features. Classes represent re-usable block templates, while objects represent

```
1  with self.implicit_connect(
2      ImplicitConnect(self.mcu.gnd, [Common]),
3  ) as imp:
4    (self.led, ), _ = self.chain(self.mcu.digital[0],
5        imp.Block(IndicatorLed()))
```

Figure 5.9: **Example of an alternative structure for instantiating the Blinky circuit using implicit connect and chain**. Line 2 defines the ports (microcontroller ground) that inner blocks instantiated in the `with` block should connect to, by tag matching. The `IndicatorLed` instantiated on line 5 has its ground port tagged with `Common`, so it is automatically to the microcontroller's ground. The `chain` statement on line 4 then connects the microcontroller's digital pin to the LED's `Input`-tagged signal pin.

individual instances. Generators defining a block's contents are written as a member function which can instantiate and connect sub-blocks, ports, and parameters.

We also provide syntactic sugar constructs for frequent use cases as shown in Figure 5.9. The first, implicit connect, is motivated by the large number of common connections like power and ground. This is structured as a code block, in which internal sub-blocks will have connections made by tag matching. The second, chain, is motivated by the frequent appearance of connections through blocks, in one port and out another. Syntactically, this allows block declaration and connection to happen on one line, and also makes linear connection topologies more obvious in HCL. These constructs can be mixed with each other, as also shown in Figure 5.9, where the implicit connect provides the ground and the chain provides the signal.

Subcircuits and generators are defined in the same way, as shown in Figure 5.10. The same also mostly holds true for links, given their block-like structure.

Visualization and Refinement Interface

As our formative studies in Chapter 4 highlighted the need to balance control and transparency with automation, we also provide a visualization and refinement GUI. This user interface, shown in Figure 5.11, visualizes the resulting design with an automatically laid out block diagram and provides insight into the system's reasoning though inspection of solved values.

Furthermore, users can select block subclass refinements in the interface, allowing the HCL to remain high-level while specifics can be dealt with interactively. The resulting subcircuit is then automatically generated, and model checks catch mistakes. For example, a user could refine an abstract resistor into a concrete surface-mount chip resistor, and its modeled power rating allows automated compatibility checks.

```
1  class IndicatorLed(GeneratorBlock):
2    def __init__(self) -> None:
3      super().__init__()
4      self.io = self.Port(DigitalSink())
5      self.gnd = self.Port(Ground())
6
7    def generate(self):
8      super().generate()
9      voltage = self.get(self.io.output_high_voltage)
10     self.led = self.Block(Led())
11     self.res = self.Block(Resistor(
12         resistance=(voltage / 0.010,    # max current, 10 mAmp
13                     voltage / 0.001)))  # min current, 1 mAmp
```

Figure 5.10: **Simplified code for the indicator LED subcircuit**. Lines 4 and 5 define the external ports by their types, while lines 10-13 define the internal blocks. Notably, as on line 9, generators can access solved values like digital logic thresholds, and use those to automatically size internal blocks like the resistor. Internal connections omitted for brevity.

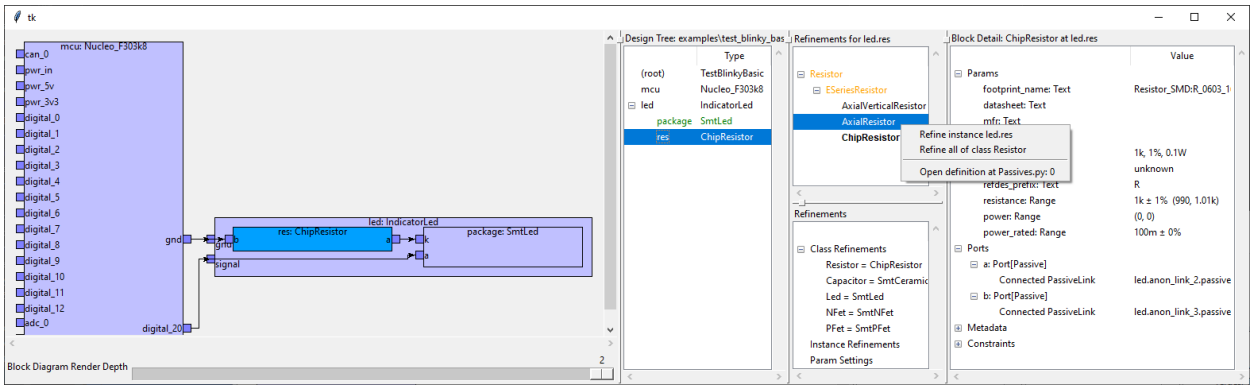

Figure 5.11: **Visualization and refinement GUI with the Blinky example** from Figure 5.2 open. An automatically laid out block diagram is on the left side, while tree view of the design is immediately to the right. In the design tree, abstract blocks (needing refinement) would be shown in yellow, refined blocks in green, and error blocks in red. The top of the vertically split pane shows the available refinements for the currently selected block, and users can apply block-specific or type-wide refinements through a context menu. The bottom pane shows all the chosen refinements. The rightmost pane displays details of the selected block, including parameters and connected ports.

Board Generation

As subcircuits are fully defined at lowest level of the hierarchy block diagram, the overall design is equivalent to a schematic. Our system can export this as a netlist file describing components and their connectivity, which can then be imported into KiCad's [35] board layout tool. Otherwise, we do not address board layout.

As the overall hardware design flow involves a back-and-forth between schematic and layout, we enable netlist updates to a work-in-progress layout by generating deterministic component names using HCL variable names. However, this does require those names to be stable, so additional techniques will be needed to support user HCL refactoring.

5.3 System Implementation

The user-facing HCL is implemented as a library of base classes in Python, with mypy static type annotations allowing user code to be type checked. The compiler, netlister, and visualization interface were also written in Python with the TkInter GUI toolkit. The entire project is open-source at `https://github.com/BerkeleyHCI/PolymorphicBlocks`.

The user HCL code invokes hardware construction methods (like `Block` and `Port`) which builds up the hierarchy block model as a tree data structure.

Compiler Structure

The hardware compiler takes the "high-level" model, as in Figure 5.2 left, and incrementally "lowers" the model by adding detail and expanding sub-elements until getting to the lowest form, as in Figure 5.2 right. This is structured as a tree walk, from blocks to its internal ports, sub-blocks, and links, recursively. Each visited block is transformed as follows:

Refinement: if there is a refinement selected for the type or particular block, the block is replaced with the refinement.

Generation: if the block is a generator, the generator is provided with the concrete values of any accessible parameters, then invoked to define the block's internal elements.

Generators run once and not in any specific order, so all referenced parameters must have at least worst-case bounds, and the generator must be written to produce a working implementation for that entire range. Similarly, generators must specify pre-execution worst-case bounds for parameter values. For example, voltage converter generators define a worst-case current draw before a tighter one is available post-generation. This is an implementation limitation, and future work could explore better approaches like inferring an order from the constraint graph and allowing interactive updates.

Constraint graph update: constraints between parameters are parsed into a directed graph. Constraints of the form "`a == something`" are recorded as assignments to `a`, and constraints of the form "`a subset-of something`" are recorded as bounds to `a`. Parameter values are evaluated by walking the constraint graph, and only when needed (lazily). A value may have any number of subset bounds, but only one assigned value (as long as it satisfies

all subset bounds). Constraints not matching either form do not affect evaluation, and are instead recorded as assertions that are checked at the end.

Netlisting is handled as a compiler phase after the design has been fully lowered, and is also a tree walk that builds up and writes out the index of footprints, pins, and connections.

Block Diagram Layout

We use ELK [22] (through py4j) as the block diagram layout engine, specifically its "layered" algorithm which supports hierarchy blocks and ports. As this algorithm relies on directed edges to provide a reasonable layout, we infer directionality primarily from the link port. For example, a voltage source would be the tail, and a voltage sink would be the head. Bidirectional ports are treated as sinks, except for when the link has no sources, the first bidirectional port is treated as a source.

A series of simplification transforms hides internal details like bridge and adapter pseudo-blocks by collapsing them and merging their input and output edges. High-fanout links (containing over thee sinks) have their edges replaced with stubs for simplicity, analogous to power rail and ground symbols in schematics. Overall, while these approximations are not perfect, they appear to produce usable block diagrams.

5.4 Example Applications

We demonstrate the capabilities of our system by designing, physically building, and testing two example systems.

Simon

We extend the Blinky example into the Simon memory game, shown in Figure 5.12 and consisting of four colored light-up buttons and an accompanying audio tone for each color.

We use a socketed Nucleo microcontroller development board as both a power source and compute element. Since the lights in the dome buttons require 12 volts while the Nucleo only supplies 5 volts, a boost converter generates the necessary voltage and a MOSFET circuit drives the lights from a 3.3 volt pin. We further added a speaker driver, speaker connector, and debugging tricolor LED. In terms of structure, each of these is a library sub-block.

Overall, the top-level HCL for Simon is 58 lines. Of note is that the boost converter instantiation requires only one line of code including the desired output voltage, minimizing design effort for an element where we do not care about the specific implementation. The boost converter generator library encapsulates the details and process of component sizing.

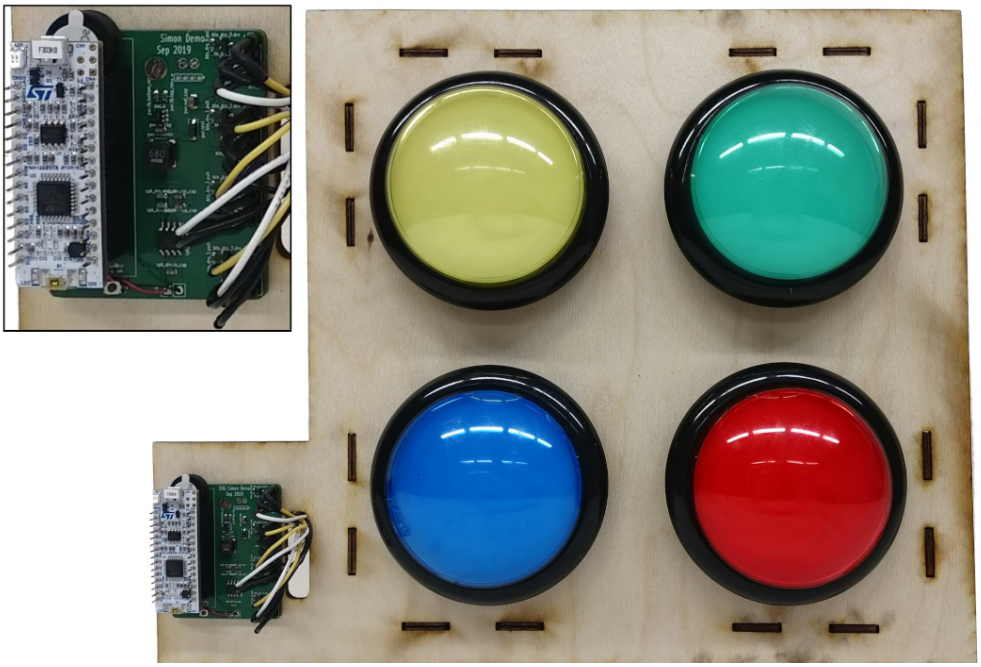

Figure 5.12: The Simon PCB (with detail view) and connected buttons.

Datalogger

A more complex design is the datalogger, shown in Figure 5.3, which records data from a Controller Area Network (CAN) interface onto an SD card. In contrast to Simon's socketed microcontroller board, this drops a microcontroller chip and its supporting components directly on the board.

In addition to the necessary CAN interface, SD card socket, microcontroller, and power conditioning blocks, this design also includes a supercapacitor-based backup power supply. Similar to the boost converter generator, this block generates a current-limited charger and automatically sizes internal elements like the transistor and reference voltage divider.

5.5 User Study: Methodology

While the preceding examples demonstrate that our system can produce working boards, usability is also an important practical consideration. We ran a small user study, in which participants designed an electronics project of their choice.

Overall, our study design prioritizes ecological validity (realism) with open-ended tasks and participants' choice of projects, important aspects for creativity support tools [72]. Furthermore, we focused on qualitative feedback: as a concept significantly different from current

practice, we felt that answers to "where and why does it work" which could drive future work were more interesting than a binary "does it work".

Participants

We recruited 3 local participants through personal referrals, including two professional engineers and one electrical engineering undergraduate. All participants had at least intermediate familiarity with PCB design and Python.

Participants were compensated with gift cards at $50 an hour for the data collection interviews, and given a budget of up to $300 for parts and boards to build their projects.

Structure

We set up a fresh virtual machine (VM) for each participant, which they would remote-desktop into using X2go. Each VM ran Ubuntu 18.04 with XFCE and IntelliJ Community Edition (which all participants used) pre-configured to work with our system. Participants did not have issues navigating the remote desktop interface, and everything was reasonably responsive.

We asked participants to share their VM window over video conference so we could watch their progress and provide help. While we did not record these sessions, we did take notes. As documentation and error messages were specifically not under evaluation, we would answer any questions participants had, including giving pointers to example code where appropriate.

The study started with a introductory session where participants worked through a tutorial document which involved building the blinky design from Figure 5.2, then extending it with a switch, LED array, discrete microcontroller, and temperature sensor. This tutorial introduced all the HCL constructs, from basic model and abstractions to the implicit-connect and chain syntactic sugar constructs, and ended with a simple part definition exercise for the temperature sensor.

Afterwards, we worked with participants to define a project of appropriate complexity and scope. In particular, we wanted a system architecture which neatly decomposes into blocks and could re-use common library elements, but also involved building a generator and modeling a few parts. We felt that building a single generator would help in understanding how automation features (like low-pass RC generators) work, while remaining considerate of participants' time. Furthermore, as the effectiveness of our tool depends on extensive libraries which normally would be provided by a community in mature projects, we also built library parts needed for participants' projects for parts we deemed common. This phase was conducted with a mix of video conference and instant messaging, as a back-and-forth process which spanned several days. We then scheduled time for participants to actually write HCL.

Once participants were satisfied with their HCL, we conducted a semi-structured interview. Topics included their overall thoughts about working in the system and comparisons with mainstream flows, as well as specific thoughts on the HCL, abstractions, electronics model, and supporting tooling. We attempted to reduce the effects of acquiescence bias by

encouraging participants to be frank and by framing the interview as constructive feedback rather than evaluation. Interviews were audio recorded (with participants' consent), and lasted an average of 2 hours and 19 minutes.

Afterwards, participants had the option of continuing to a board layout, which was primarily independent and on their own computer, unless they needed to make netlist changes. Because of COVID-19, we were unable to physically fabricate, assemble, and test the final devices.

5.6 User Study: Results

Overall, participants spent an average of 1 hour 5 minutes completing the tutorial, and 5 hours 15 minutes working on their HCL, including 2 hours building subcircuit and part libraries, and including untracked time understanding the circuits being built and becoming familiar with the system. By the end, participants were able to work effectively with the system, got designs to a point they were satisfied with, and continued to layout. All three projects are detailed below, with PH02's project shown in Figure 5.13 and HCL in Figure 5.14.

Project: Power Meter

PH01's project was an inline power meter that measures the voltage and current passing through it. PH01 started by modeling the INA190 current sense amplifier chip, then building the top-level system with stub sub-blocks for the current and voltage sense chains, and finally implementing those sub-blocks including writing the differential RC filter generator. The initial design idea came as a sketch of the analog signal chain in KiCad, while the rest of the system came together during HCL writing and based on available library parts.

PH01 wrote 112 lines of system-level HCL (including signal chain sub-blocks), 20 lines of generator libraries, and 95 lines of part definitions. The layout had 66 individual components.

Project: Thermistor Reader

PH02's project was a thermistor reader that displays readings from a bank of 8 thermistors and plays an audio alert if bounds are exceeded. PH02 chose to start by writing the thermistor and RC filter combination generator, which would calculate the series resistor and parallel capacitor values given the nominal thermistor resistance. Of note is the use of a for loop to generate the repeated thermistors and signal chains. This was also the only case requiring a model override: the OLED and speaker worst-case current draw exceeded capabilities of the USB port, so an inline pseudo-block (using 3 lines of code) was used to lower the modeled current, effectively telling the system that these parts would not be run at full power.

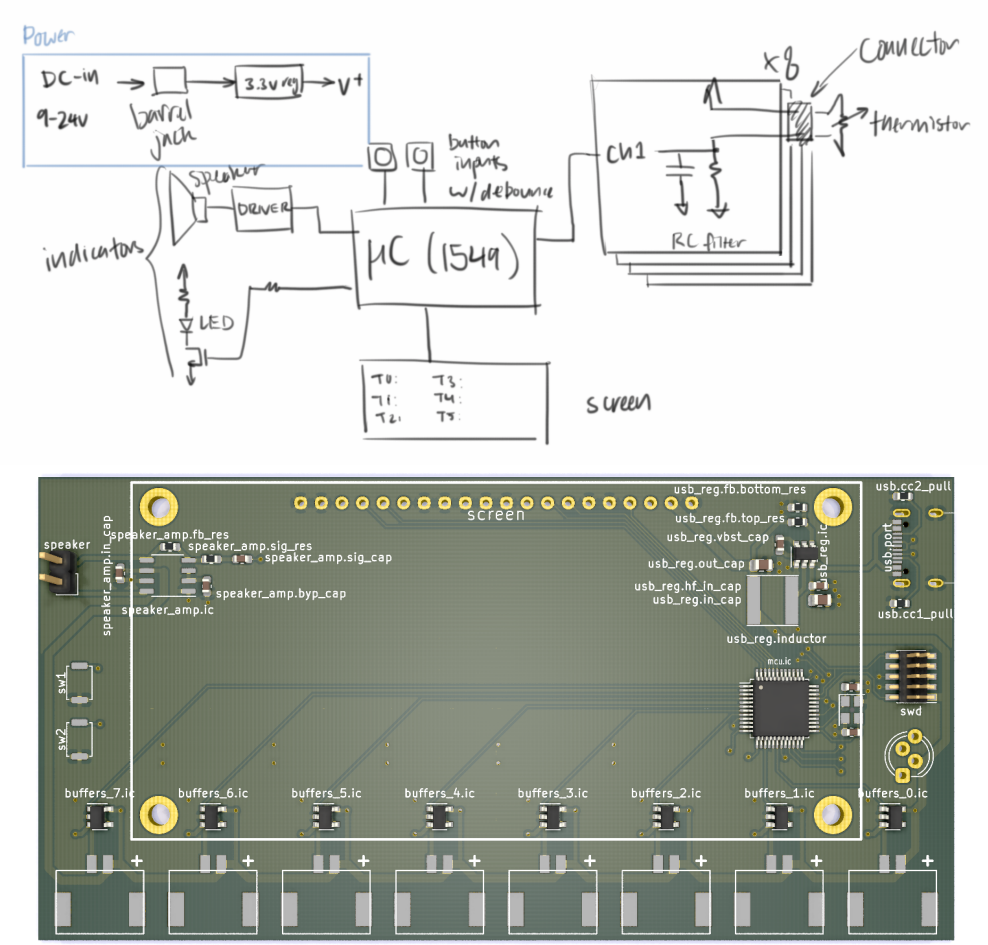

Figure 5.13: **PH02's initial system diagram for the thermistor reader**, and the resulting PCB (rendering) they produced using our system.

PH02 wrote 52 lines of system-level HCL, 40 lines of generator libraries, and 15 lines of part definitions. The layout had 90 individual components.

Project: Multifunction Instrument

PH03's project was an USB oscilloscope, function generator, logic analyzer, and power supply combination device, all driven from a microcontroller. PH03 chose to start by writing the variable-output buck converter generator, modifying the existing feedback controller chip based buck converter by adding a PWM input, MOSFET switch, and diode. This process turned out to be tricky, requiring deeper circuits knowledge to size switches and diodes compared to the typical process of choosing an off-the-shelf chip and using reference schematics and part selections. However, once completed, the top-level system architecture, including hooking up the converter, signal buffers, LCD, and USB, progressed smoothly.

PH03 wrote 48 lines of system-level HCL, 24 lines of generator libraries, and 90 lines of part definitions. The layout had 53 individual components.

Advantages

Overall, participants were happy with the system architecture and level of design, with PH01 noting that it matched the ideal.

Participants also liked the pre-built blocks and the encapsulation they provide. PH02 noted that library blocks could reduce the need to read through datasheets and make it more difficult to miss non-obvious elements like the pull-down resistors on the Type-C receptacle. PH03 also compared the cleaner and integrated generator library approach of our system with their painful existing flow of building buck converters by searching on chip vendor sites, using Excel calculators, and downloading and importing footprints.

All participants found the more detailed automated checks to be useful, with PH01 considering it the best part of the system. PH02 felt the system could be particularly useful for novices, making it more difficult to get an obviously bad schematic compared to the weaker ERC in existing tools. Furthermore, in combination with previous hardware-proven designs built in this system, the block diagram visualization, and familiarity with the circuit from doing the layout, all participants had between medium and high confidence that their design would work. However, participants were more skeptical of community libraries, for example saying that they would do spot checks or want quality indicators.

Limitations

While participants generally felt the electrical checks were reasonable without being excessive, PH01 cautioned that the checks were better described as sanity checks as the modeled values were based on datasheets which might assume certain conditions, context that is lost in our model. Furthermore, PH02 noted that the modeling and encapsulation of generators

might not be comprehensive: for example, a user instantiating a thermistor block would need to know whether the signal rises or falls with increasing temperature.

All participants encountered failed checks, often due to tolerances set too strict for parts like resistive dividers. Though participants recognized these as true-positives and solved these by loosening tolerances, this tolerance specification with stackup differs from design practices around nominal values. Furthermore, PH01 found the common tolerance debugging process of loosen, re-compile, and iterate to be annoying, suggesting either tighter iteration loops or presenting the best achievable value. PH01 also preferred checks to be non-fatal and not prevent netlist generation where possible, though PH02 preferred to not waive checks and instead use more targeted and explicit mechanisms like tightening the worst-case current draws.

PH03 felt that the learning curve was steeper than a GUI, and that the system does require familiarity with Python. Furthermore, the object-oriented Python in our HCL may differ from the scripting aspects used by hardware designers. PH01 also noted mismatches between terminology and class names presented in our system and existing schematic capture concepts, and viewed intuitive names as essential to easy learning.

One issue PH01 noted with the refinement process is that this data are stored separately from the HCL, so the HCL alone would be insufficient for a design review. Suggestions include having refinements generate code back into the HCL, or having refinements be part of review. In general, PH01 and PH03 also noted good tool support for code diffs, though also acknowledged the existence of schematic diff tools.

Finally, participants brought up a slew of less-fundamental usability issues with the system. This ranged from poor automatic net naming, to HCL syntax issues like excessive verbosity reducing the signal-to-noise ratio.

Part Building

Though all participants agreed that modeling parts and writing generators was worth the cost if it was likely to be re-used and shared, they differed in the details. PH02 found writing the math for the RC filter calculation to be easy, and PH03 noted that having an existing generator as a starting was very helpful. On the other hand, PH01 pushed for an untyped port, which would in effect waive model checks for when one just wants things to connect.

Graphical Interfaces

All participants also made use of the visualization and refinement interface to explore the compiled designs. PH01 noted that circuit reading usually relies on visual pattern matching on schematics, and it was harder to see the connectivity structure from the HCL, though PH02 believed the HCL to be reasonably clear. PH01 also thought that while the automatically generated block diagram was reasonable for the top level, deeper levels showing individual components significantly deviated from schematic convention. However, that was

tempered with the hope that adding a few more simple rules, like ordering ports by voltage, could produce significant improvements.

All participants also independently suggested tightening the HCL and block diagram update loop, perhaps by integrating the visualization into an IDE. One use case suggested by PH02 was to highlight block pins that still need to be connected.

Participants did have differing opinions on the HCL as a design entry interface. PH02 thought the HCL with its for loop and textual entry was faster, though modern schematic tools somewhat close the gap with support for hierarchy replication. PH01 noted more generally that HCLs and graphical schematic editors were suitable for different purposes, preferring schematics for analog designs with high connectivity between a few components, and preferring HCLs when the equivalent schematic sheets would be very complex and cluttered.

Design Time

All participants mentioned design time as a metric when comparing this system to mainstream flows, with PH03 also mentioning design pain. While acknowledging that it was difficult to fairly compare time for such different flows, PH02 and PH03 estimated their projects would have taken about as long in a traditional flow (give or take depending on assumptions), while PH01 was more wary about comparing new tools to familiar tools. PH03 further noted that the end results were more "portable", including time invested in reusable components. However, PH02 was unsure about benefits when dealing with specialized, one-off components, and PH01 noted the flexibility in mainstream flows to defer component sizing to quickly proceed to layout.

5.7 Limitations and Future Work

While we have presented a system that ultimately produces working boards and conducted user trials with an emphasis on simulating realistic conditions, there are both important limitations and open avenues for continued work.

Graphical Interfaces

Based on user feedback, perhaps the most important usability improvement would be better integration with graphical block diagram or schematic representations. The most ambitious idea would be a fully linked, hybrid HCL and block diagram editor, allowing users to freely move between whichever representation suits their current task best. Less ambitious would be tighter updating of block diagrams from HCL, better automatic block diagram layouts (possibly with user-specified hints), and better tools for tracing and sense-making of constraint errors.

Furthermore, while an HCL is necessary to write generators, the resulting blocks and the rest of our design model can be used from within a graphical, schematic-like interface. This would eliminate the need for programming experience and provide a more familiar interface and graceful transition.

We implement and evaluate an integrated development environment (IDE) plugin based on these ideas and supporting mixed text and graphical editing next, in Chapter 6.

Library-Based Approach

Our approach relies on having good and complete libraries to maximize re-use. Though our current library includes many common parts and subcircuits, it is far from complete. While a database of simple parts might be easily parse-able from a parametric product table, complete details for more complex parts are often only available in PDF datasheets. Future research on extracting data from datasheets with tools such as Tabula [78] and DocParser could accelerate this effort.

Overall, collaboration from a large community may be key to building a critical mass of parts and subcircuit generators to support the needs of users. However, as noted by participants, this must be balanced with quality indicators to enable confidence in re-use.

Electronics Model

The foundational abstractions of hierarchy blocks, links, and parameters appeared useful to and was understood by users. While the electronics model proved suitable for our intermediate-level example designs and user projects, it has many limitations, for example defining only a few signal interfaces and lacking support for multiple grounds. We do caution that continued work extending the model must balance functionality with usability and usefulness.

Users and User Study

In building our system and libraries, we focused on supporting intermediate-level designers and projects. In particular, sufficient circuits background enables effective use of library blocks, while less complex projects avoid needing a long tail of specialized parts. However, we believe that with additional work – such as on-demand documentation for novices, or an expanded library and model for experts – our approach will scale up and down both the skill and complexity hierarchy.

That being said, we do caution against generalizing the user study results, given the small participant pool and the selection for circuits knowledge and programming experience. We position our results as a first step, leaving larger and more robust studies – and the need for a more polished and scalable system – as future work.

```
self.usb = self.Block(UsbDeviceCReceptacle())
with self.implicit_connect( ImplicitConnect(self.usb.pwr, [Power]),
                            ImplicitConnect(self.usb.gnd, [Common]) ) as imp:
  self.usb_reg = imp.Block(BuckConverter(output_voltage=(3.0, 3.3)))

with self.implicit_connect( ImplicitConnect(self.usb_reg.pwr_out, [Power]),
                            ImplicitConnect(self.usb.gnd, [Common]) ) as imp:
  self.mcu = imp.Block(Lpc1549_48())
  (self.swd, ), _ = self.chain(imp.Block(SwdCortexTargetHeader()), self.mcu.swd)
  (self.crystal, ), _ = self.chain(self.mcu.xtal, imp.Block(
      OscillatorCrystal(frequency=12 * MHertz(tol=0.005))))
  (self.usb_esd, ), _ = self.chain(self.usb.usb, imp.Block(UsbEsdDiode()), self.mcu.usb_0)

  self.thermistors = ElementDict[ThermistorLowPassRc]()  # Thermistor array and buffers
  self.buffers = ElementDict[OpampFollower]()
  for i in range(8):
    (self.thermistors[i], self.buffers[i]), _ = self.chain(
      imp.Block(ThermistorLowPassRc(47*kOhm(tol=0.05), 0.5*kHertz(tol=0.2), True)),
      imp.Block(OpampFollower()), self.mcu.new_io(AnalogSink))

  self.screen = imp.Block(Nhd_312_25664uc())  # Screen
  self.connect(self.mcu.new_io(DigitalBidir), self.screen.cs)
  self.connect(self.mcu.new_io(DigitalBidir), self.screen.reset)
  self.connect(self.mcu.new_io(DigitalBidir), self.screen.dc)
  self.connect(self.mcu.new_io(SpiMaster), self.screen.spi)

  self.sw1 = imp.Block(DigitalSwitch())  # Switches
  self.connect(self.sw1.out, self.mcu.new_io(DigitalBidir))
  self.sw2 = imp.Block(DigitalSwitch())
  self.connect(self.sw2.out, self.mcu.new_io(DigitalBidir))
  self.rgb_led = imp.Block(IndicatorSinkRgbLed())  # Indicator light
  self.connect(self.mcu.new_io(DigitalBidir), self.rgb_led.red)
  self.connect(self.mcu.new_io(DigitalBidir), self.rgb_led.green)
  self.connect(self.mcu.new_io(DigitalBidir), self.rgb_led.blue)

self.forced_current = self.Block(ForcedCurrentDraw( (0, 0.1*Amp) ))
self.speaker_amp = self.Block(Lm4871())
self.speaker = self.Block(Speaker())
self.connect(self.forced_current.pwr_in, self.usb_reg.pwr_out)
self.connect(self.forced_current.pwr_out, self.speaker_amp.pwr)
self.connect(self.speaker_amp.spk, self.speaker.input)
self.connect(self.speaker_amp.gnd, self.usb.gnd)
self.connect(self.speaker_amp.sig, self.mcu.new_io(AnalogSource))
```

Figure 5.14: **The system-level HCL for PH02's thermistor board**, simplified for brevity.

Chapter 6

Integrated Development Environment

6.1 Introduction

While the last chapter described an HCL approach to board design and demonstrated that it can produce boards with more automation and design assistance than mainstream schematic flows, HCLs are also a very different interface. As the user study showed, the Visualization and Refinement Interface which provided a schematic-like block diagram view of the compiled HCL was useful, yet a major issue was the lengthy compilation delay on each update. All participants suggested tightening the visualization with the HCL writing process.

We note that HCLs and block diagrams are ultimately views on similar underlying data, but each representation emphasizes different information and provides different affordances for understanding and manipulating a design [10]. For example, while an HCL might enable loops to quickly generate arrays of components, the circuit connectivity is much more obvious in a block diagram. However, more practically, mainstream schematic tools are likely more familiar to existing electronics designers and familiar interfaces can lower the barrier to entry for more powerful approaches.

In this chapter, we present a novel integrated development environment (IDE) approach to bridging these two views. Our approach is centered on interactive block diagrams that are synchronized with HCL code: code edits are reflected in diagram updates, and diagrams can be edited to modify code. We hope this schematic-like representation can provide a complementary view for editing the HCL, and furthermore a graphical interface may make HCLs more accessible to a broader community of hardware engineers, device designers, and hobbyists with more limited programming expertise. Our work relates to visual editors for other domains such as GUI editors with code generation. However, our approach differs in that code remains the primary design input, and modifications from the GUI and direct text edits can be arbitrarily interleaved. This preserves the flexibility of the underlying HCL.

We contribute an IDE implementing these techniques, both as a tool to concretely support board design, and more generally as another design point in the space of similar tools [28, 49] that bring programming power to domains beyond software. Furthermore, we contribute

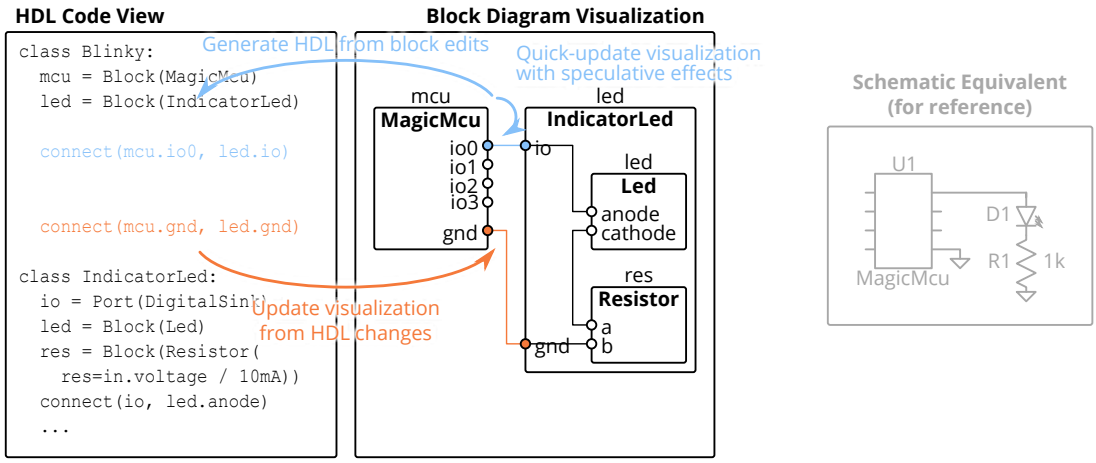

Figure 6.1: **Left: an overview of our integrated development environment (IDE) approach for tooling to support working with circuit board-level hardware description languages (HCLs)**. This consists of the traditional text editor (left half of the IDE), and a block diagram visualization of the compiled HCL (right half of the IDE). Edit actions on the block diagram, such as creating connections between ports shown in blue, generate the corresponding lines of code in the HCL and update the visualization without incurring the latency of a full recompile. The HCL code, including that inserted from block diagram edit actions, can also be freely edited to preserve the full power and flexibility of the base HCL. User-triggered updates recompile the HCL, and changes such as the connection shown in orange, are visible in the block diagram and available for editing. **Right: As a reference, the equivalent design in a mainstream schematic editor**. Comparatively, schematics often require the system designer to work at the lowest level of abstraction (instead of re-using library components like `IndicatorLed`) and manually handle component calculations (like the resistor) which are both tedious and do not preserve design intent.

an evaluation in the form of a qualitative and remote user study with four participants, from which we draw takeaways and recommendations for similar tools. We found that even though some participants preferred to write HCL by direct text edits instead of through the block diagram interface, the tight visualization loop was beneficial to all users. Furthermore, participants were able to blend use of the block diagram interface with textual HCL edits, suggesting that this mixed IDE approach is viable and even if not all possible code edits are supported from the GUI.

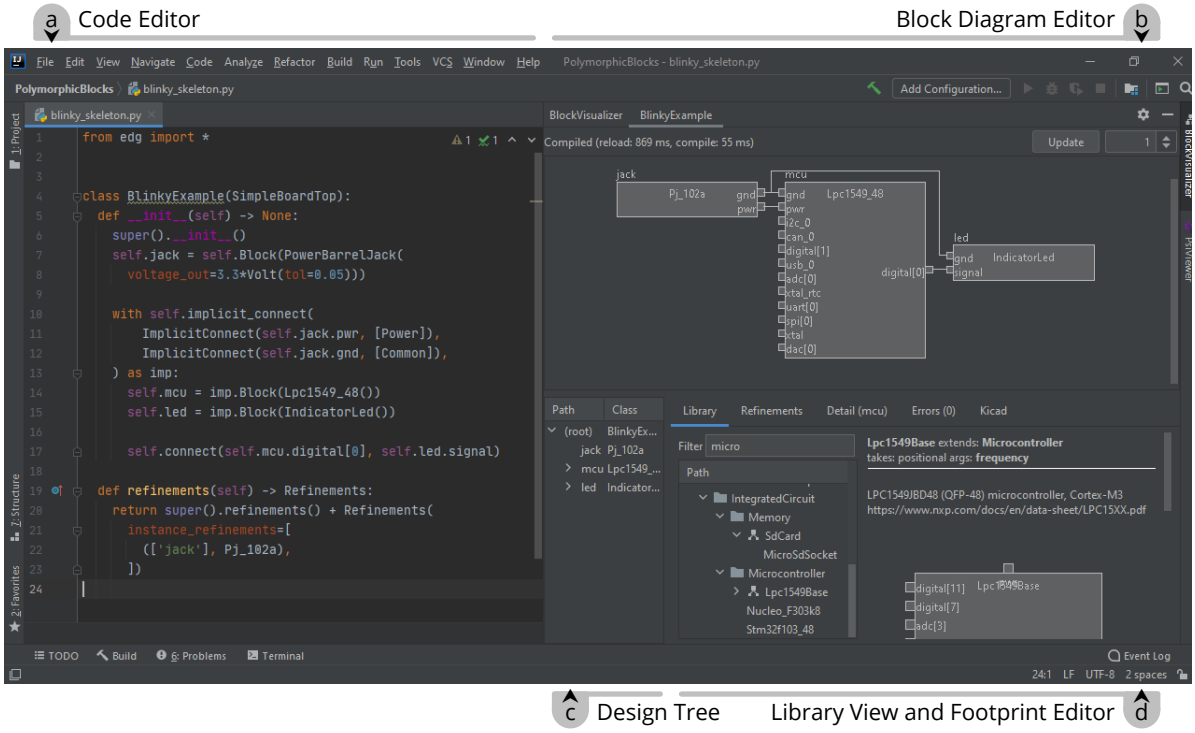

Figure 6.2: **Overview of the IDE components**. The IDE, implemented as an extension for IntelliJ, has several major views: (a) a code editor for the hardware description language, (b) an interactive block diagram editor that corresponds to the compiled HCL, (c) a tree view of the same compiled HCL, and (d) a library view for browsing available blocks.

6.2 System Description

Although there are many possible ways to bridge HCL code and schematics, our system is motivated by two underlying principles:

1. Preserve the full power of the underlying HCL, while noting that from the formative studies in Chapter 4 that board design tends to be highly iterative. This precludes approaches that do not allow users to freely move between code and schematic workflows, such as single-shot code generation from schematics (like GUI builders) in which continued editing from the schematic view may be difficult after modification of the generated HCL. We further note that as there are likely programming constructs for which there is no (useful) corresponding graphical representation, the visualization will necessarily be a subset of the HCL.

2. Interfaces based in current practices. While this means supporting a schematic-like

interface for HCL editing, it also means boundaries on what be may less useful in a graphical editor. For instance, although a LabVIEW-style [30] interface may support programming constructs such as arithmetic and control flow as blocks in a purely graphical environment, code is overwhelmingly the more common way to express this kind of logic.

From these principles, we built a system that takes an "HCL first" approach, where the HCL code is the primary and authoritative design artifact. Tooling then provides support for understanding, navigating, and producing this code through an IDE metaphor built around a schematic-like view of the compiled output design as shown in Figure 6.2. More concretely, the block diagram view provides users with a visual representation of the HCL, and, combined with the library view, provides for schematic-like GUI edit actions that generate into corresponding HCL. However, we explicitly do not support the more code-like parameterization operations. Furthermore, as a prototype, we chose to focus on the basics and do not support syntactic sugar constructs like implicit connects or refactoring operations like delete and rename.

In the rest of this section, we will introduce our system in more detail through how a hypothetical user might build the example blinky LED system in Figure 6.3 and magnetic field sensor component in Figure 6.4, starting from an empty top-level design.

Block Diagram Edit Actions

Like with schematic capture tools, the user starts with adding parts to the design by searching libraries. In our system, the library view on the bottom right lists available components as a tree, organized by the type hierarchy which encodes both categorical (e.g., power converters, sensors) and structural (e.g., three ported DC-DC converter) dimensions. The textbox above provides search and filtering by simple string matching.

Since each edit occurs within the HCL, our user first starts by moving the caret to where code should be inserted, at the beginning of `def __init__(...):`. Then, to add a microcontroller, they would first enter `microcontroller` in the library filter textbox, which brings up the microcontroller category in the library tree. Within that category, our user chooses the `Lpc1549_48`, double-clicks it to insert the block instantiation line at the caret, and provides a name in the pop-up prompt.

Alternatively, right-clicking (instead of double-clicking) brings up the context menu which provides suggested locations for code insertion independent of the caret position. For block inserts, one such option is appending to the end of the `__init__` method.

Simultaneously, the block diagram updates with the visual representation of the newly added block. To preserve flow with GUI operations, this is not the result of a full recompile, but is instead a *speculative effect*: the system assumes (regardless of context, such as if the caret was within an `if` block or `for` loop) that exactly one instance of the block would have been created. These blocks are indicated with a hatched fill, and additionally do not contain

```
1  class BlinkyExample(SimpleBoardTop):
2    def __init__(self) -> None:
3      super().__init__()
4      self.jack = self.Block(PowerBarrelJack(voltage_out=3.3*Volt(tol=0.05)))
5
6      with self.implicit_connect(
7              ImplicitConnect(self.jack.pwr, [Power]),
8              ImplicitConnect(self.jack.gnd, [Common]),
9      ) as imp:
10       self.mcu = imp.Block(Lpc1549_48())
11       self.led = imp.Block(IndicatorLed())
12
13       self.connect(self.mcu.digital[0], self.led.signal)
14
15   def refinements(self) -> Refinements:
16     return super().refinements() + Refinements(
17       instance_refinements=[
18         (['jack'], Pj_102a),
19       ])
```

Figure 6.3: **Example system-level design HCL**. Lines 4, 10, and 11 respectively instantiate and name `Block` objects for a barrel jack, microcontroller, and indicator LED. Blocks can have assignable parameters, such as the target output voltage of the barrel jack in line 4. Lines 15 - 19 further refine the abstract `PowerBarrelJack` instantiated in line 4 to be the specific and concrete subtype, the PJ-102A.

an internal implementation. However, their ports are valid, which allows connections to be made to them without re-compiling.

If our user instantiated a block with a required parameter, such as the `voltage_out` specification for a barrel jack, the unfilled keyword argument appears on the instantiation line. As with parameterization in general, the user must write this in HCL, here by providing an output voltage target of `3.3*Volt(tol=0.05)` as in line 4 of the example in Figure 6.3.

After repeating the instantiation flow with the LED, our user can then start connecting blocks. As shown in Figure 6.5, unconnected-but-required ports are marked with a red fill, and the user starts with one such port, the LED's signal input pin. Double-clicking the pin starts the connect tool, which dims the rest of the schematic except for ports that have compatible types, as also shown in Figure 6.5. The user chooses the only such available pin on the microcontroller, a digital IO line, then double-clicks again to commit the connection. For connects, names are optional and may be left blank.

Similarly with the block insert action, the `connect` statement is inserted at the caret, and the connection is immediately though speculatively made on the block diagram. As required ports are connected, the red error fill also goes away.

Our user then initiates a recompile through the Update button, and a second or two

```
1  class Lf21215tmr_Device(FootprintBlock):
2    def __init__(self) -> None:
3      super().__init__()
4      self.vcc = self.Port(
5        VoltageSink(voltage_limits=(1.8, 5.5)*Volt, current_draw=(0, 1.5)*uAmp),
6        [Power]
7      )
8
9      self.gnd = self.Port(Ground(), [Common])
10
11     self.vout = self.Port(DigitalSource.from_supply(
12       self.gnd, self.vcc, output_threshold_offset=(0.2, -0.3)
13     ))
14
15     self.footprint(
16       'U', 'Package_TO_SOT_SMD:SOT-23',
17       {
18         '1': self.vcc,
19         '2': self.vout,
20         '3': self.gnd,
21       },
22       mfr='Littelfuse', part='LF21215TMR',
23     )
```

Figure 6.4: **Example part definition of a Lf21215TMR digital magnetic field sensor**. Lines 4 - 13 defines the interface by instantiating ports `vcc` of type `VoltageSink` (voltage input), `gnd` of type `Ground`, and `vout` of type `DigitalSource` (digital output) with defined voltage and current parameters. Lines 15 - 23 define the associated footprint and pinning.

later, the recompilation completes and diagram updates. The hatched fill disappears, but the diagram is otherwise unchanged as the speculative effects matched the HCL.

All edit actions are guarded by basic checks, such as for name legality and insert position. Connect insertion further checks that the referenced blocks and ports are declared before the insertion point. These checks are performed before the action is invoked, so an invalid caret location would result in a greyed out context menu item. For double-click actions, an error message pops up instead.

We currently do not support deletion actions in the graphical editor, as accurate static analysis of Python code is difficult. However, the right-click context menu for block diagram objects has options to navigate to the line of code where a block is instantiated or a port is connected to assist in textual edits.

Code-to-block diagram navigation is also supported. Where a line of code may correspond to several objects in the block diagram (such as within a block instantiated multiple times), a disambiguation list pops up for the user to choose from.

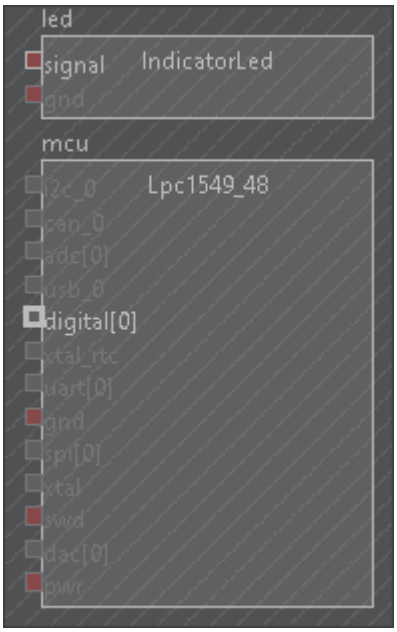

Figure 6.5: **The connection interface, showing legal connections (by type) from the microcontroller's digital IO**, with other pins dimmed out. Additionally, unconnected-but-required pins are highlighted in red, while the hatched fill indicates the preview status of both the LED and LPC1549 blocks inserted in the GUI as well as the modified status of the block they were inserted in.

Design-wide Edits

After inserting and connecting the rest of the circuit, the design now includes an abstract barrel jack block that requires a refinement.

Our user starts by selecting the abstract block in the diagram view. As with the block insertion flow, they search for barrel jacks in the library browser, and choose a `Pj_102a` barrel jack receptacle. The right-click context menu provides options to refine either the selected block instance, or all blocks of its class.

Because refinements are written in the top-level design's class and commonly as a single return statement like in lines 15-19 of Figure 6.3, there is no need for caret positioning. Since the code does not have a refinements function yet, the entire code block is generated, including the selected refinement. However, if a refinements block already existed, the selected refinement would be appended at the end of the list.

This feature expects refinements to be written with this specific structure in order to insert new refinements. While meta-programming refinements using arbitrary code may have advantages in some cases, we believe that the required refinements structure will suit

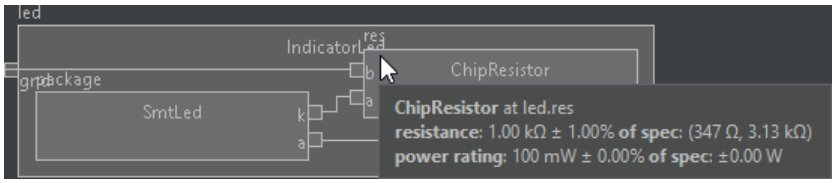

Figure 6.6: **Inspection of the resistor in the `IndicatorLed` block**, showing both the actual resistance of the selected part and its specification or requested resistance, which in turn was derived from the connected voltage source.

most applications.

Speculative effects do not apply to refinements. While refinements can change parameters throughout the entire design as well as the implementation of the refined block, refinements do not change the ports on the refined block itself and so do not impact following edit operations.

Inspection

At this point, there are several blocks in the design, and our user may be interested in the details of the generated design. For example, to understand what was inside the `IndicatorLed`, our user double-clicks into it. This allows navigating a potentially complex design by viewing one level of hierarchy at a time. Otherwise, standard mousewheel-to-zoom and drag-to-pan interfaces support moving around the diagram.

Our user may be curious about the value of the resistor in this LED-resistor circuit, especially since it was automatically chosen. They mouse-over the part to show additional details, producing the popup shown in Figure 6.6 containing summary data of the object in question.

In general, summary views exist for common blocks (such as showing component values for resistors, capacitors, and inductors, or showing ratios for resistive dividers) and common connections (such as showing voltage thresholds between digital ports, or impedances between analog ports). While the same information is also available through a tree view of the entire design showing all parameters and constraints, this avoids information overload with a human-curated description at an appropriate level of detail.

Library Creation and Edits

Where existing libraries are insufficient, end-users will need to create custom block definitions. To model the magnetic sensor component in Figure 6.4, our user starts by choosing a base class from the library browser, in this case the `FootprintBlock` class that allows an

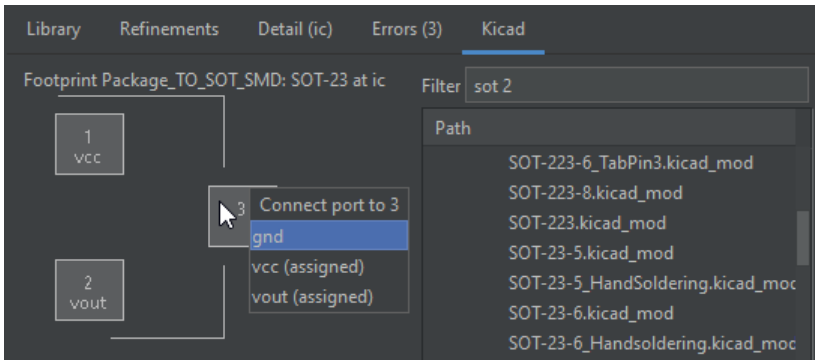

Figure 6.7: **The IDE's footprint assignment and pinning interface**. The footprint browser on the right shows the available KiCad footprints, while the footprint preview on the left allows assignment of mappings between footprint pads and block ports. All edit actions generate HCL.

associated PCB footprint. The context menu provides an option to create a new subclass, which inserts the class template at the caret.

As blocks are written similarly to top-level designs, the same block diagram based sub-block instantiation and connection interactions also apply here. However, non-top-level blocks additionally support ports, which are inserted similarly to blocks: by positioning the caret, searching for a port type in the library browser, and giving it a name.

To associate the PCB footprint, our user switches over to the KiCad tab shown in Figure 6.7. Similarly to the library browser, they start by searching for an SOT-23 device using the filter box, then double-click the specific footprint from the list to insert the `footprint` statement at the caret.

The footprint itself also appears on the left side of the tab. From here, our user could double-click on a pad to bring up a list of connect-able ports, then choose one to update the `footprint` statement with the selected port to pad mapping. Similarly to block insertion and connect, effects of each action shown speculatively on the footprint interface. The final footprint code looks like lines 15-23 of Figure 6.4.

6.3 System Implementation

The block diagram view and library browser and footprint tabs are built as a tool window plugin for the IntelliJ Community Edition IDE with the PyCharm Community plugin. The plugin itself is written in Scala and uses the Swing GUI toolkit to work with IntelliJ. The entire project is open sourced at `https://github.com/BerkeleyHCI/edg-ide`.

The IDE operates in part on Polymorphic Circuit Blocks' compiled designs which are

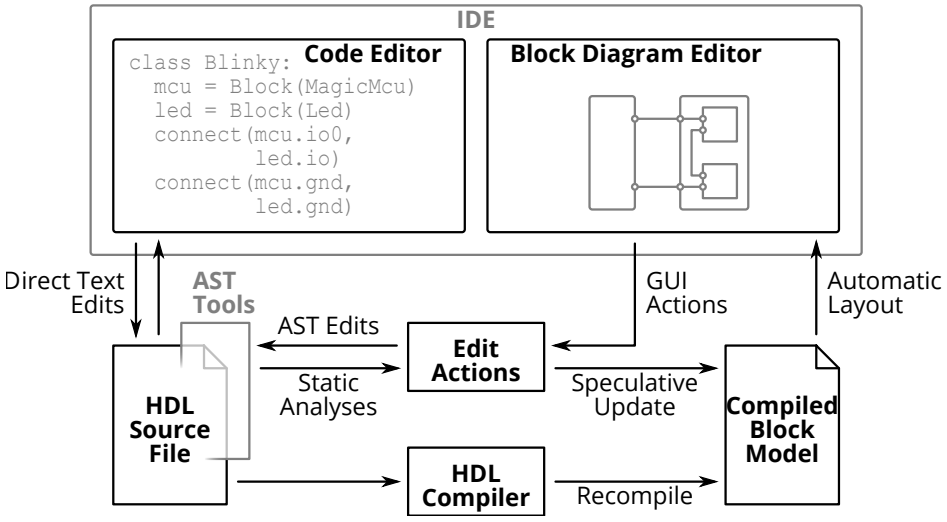

Figure 6.8: **Overall architecture of the IDE**. HCL source code is compiled into a internal block model, which creates the user-facing block diagram through automatic layout. Edit actions and some static analyses are done through an AST-like view (IntelliJ's Project Structure Interface, or PSI) of the underlying HCL and speculatively update the compiled block model.

defined in (and serialized with) Protocol Buffers. A Python stub "server" program handles HCL re-compilation requests from the IDE and communicates through gRPC, built on top of Protocol Buffers. This overall architecture is shown in Figure 6.8.

To speed up re-compilation, the IDE keeps a cached version of compiled library blocks used in the current design and listens for changes to the code of these classes. On a change, it invalidates the cached versions. For simplicity and to keep the system responsive, analysis only considers the class hierarchy and would miss other dependencies, such as changes to global functions. As a last resort, the user can also manually clear caches.

Code Analysis and Edits

For functionality dealing with code, the IDE uses IntelliJ's Project Structure Interface (PSI), which is a combination of parse tree (AST) like data structure combined with navigation and analysis tools.

As an example of analysis functionality, the invalidate-cache-on-modify invalidates the edited class and all derived subclasses by using the PSI's find-subclasses-of function. Parameters for block and port classes are found by inspecting the class's `__init__` arguments (including varargs `*args` and `**kwargs` arguments) and tracing those through `super().__init__`

calls if such a call is the first statement.

Code edits are similarly performed using PSI write operations, by building up the PSI tree representation of the text to be inserted, then either inserting it as a new node or replacing an existing node. A code formatter automatically manages stylistic aspects of the inserted code (for example, inserting line breaks on long lines).

Block Diagram Visualization

The block diagram layout algorithm is functionally similar to the one for the Visualization and Refinement Interface described in Chapter 5: start with the design as a set of hierarchy blocks, ports, and connections, infer the connection directions by port type (for example, a voltage source is an edge tail while a voltage sink is an edge head), simplify internal pseudo-blocks (like port type adapters) into direct connections, and replace high-fanout connections (for example, a voltage source connected to four voltage sinks) with tunnels or named nets. To simplify editing, we also remove disconnected array ports except for one to allocate a new port. This is structured as a series of transformations on a hierarchy block data structure, then finally passed to ELK's [22] "layered" algorithm for layout to obtain the final positioning and size data of graphical elements.

6.4 User Study: Methodology

As our system's workflow is a different way of working with code and a very different way of constructing hardware (compared to mainstream schematic flows), we felt it important to get user feedback to understand how these tools might actually be used. To that end, we ran a small user study in which participants complete a pre-defined project with our IDE. This structure tries to balance realism (ecological validity), decoupling observations about the underlying HCL from the IDE interface, and use of participants' time.

Because we are examining a prototype IDE and novel concepts which may not have the degree of interface and interaction polish as a final product, our study's goals lean more towards qualitative feedback and usage observations to drive ideas for iteration, rather than a more quantitative and evaluative approach that we do not believe is appropriate at this stage.

Participants

We recruited four participants through personal referrals, with the goal of sampling for range by having participants of different skill levels and motivations for PCB design. As a baseline, we required intermediate familiarity with PCB design and Python – supporting complete novices is currently out of scope. Two participants built PCBs primarily for personal and hobby reasons (one electrical engineering undergraduate, one professional software

engineer), while the other two participants designed PCBs professionally (one graduate student researcher and one industry engineer). Two participants had prior experience with the Polymorphic Circuit Blocks HCL, and all participants had at least some familiarity with HCLs in general such as Verilog.

Participants were compensated with gift cards at $50 an hour.

Structure

This study was conducted entirely by videoconference, similarly to the HCL user study in Chapter 5 Each participant accessed a fresh virtual machine (VM) running our IDE through the remote-desktop application X2go.

We encouraged participants to share their VM window over videoconference so we could watch their progress (which all did). While these sessions were not recorded, we took notes on participants' flows, specifically where they chose to use the IDE or not. Additionally, as coding practically often relies on community references like StackOverflow that do not yet exist for this HCL, we would answer any questions from participants.

The first part of the study consisted of a tutorial session to familiarize participants with the IDE and HCL. Participants worked through a tutorial document which involved building a slightly expanded version of the `BlinkyExample` design from Figures 6.3 and 6.4. While the document hand-held participants through individual GUI edit actions, it also described the resulting code so participants could understand the generated HCL. The tutorial also described some common but code-only syntactic sugar constructs like implicit connects and included refactoring exercises using them.

In the second part of the study, participants used the IDE to build a predetermined mini-project from a specification document. This project was an USB-powered ambient light sensor with a visual readout (such as through an LCD). The required USB connector, display, microcontroller, and power converter blocks were included in the library, and multiple options existed for each (such as having a choice between micro-B and Type-C USB connectors).

After the system-level design was completed, participants then modeled the BH1620FVC analog light sensor and application subcircuit before integrating it into their overall system. This task included automatically calculating the current-to-voltage load resistor value based on the high-level input parameters of maximum illuminance and maximum output voltage.

Participants were free to use (or not use) the IDE to build this project, based on their preferences or what felt natural. This gave us the opportunity to observe what features were useful, and how these features worked in the larger picture of circuit design.

The study ended with a semi-structured interview, covering overall thoughts of the HCL and IDE, comparisons against schematic capture flows, and specific thoughts on the block diagram edit actions, visualization updates, automatic layout, and inspection interface. Interviews were framed as constructive feedback and we encouraged participants to be frank about the system's strengths and shortcomings to reduce effects of acquiescence bias. Interviews were audio recorded (with participants' consent) and lasted an average of 1 hour and 34 minutes.

6.5 User Study: Results

Participants spent an average of 60 minutes to complete the tutorial, 19 minutes to build the system-level design of their project, and 52 minutes to build the light sensor part model and subcircuit. We note that we expected the subcircuit modeling to take significantly longer because of the additional need to understand the component datasheet, the system's electronics model at a deeper level, and parameterization functions – things outside the scope of the IDE. Furthermore, as library parts are designed to fully encapsulate details, it is much simpler to instantiate and connect them.

Individual Flows

Overall, participants had diverse flows as a result of individual preferences, and while they all made use of the IDE in writing HCL, they differed in what features they used and preferred. Furthermore, all participants also directly edited textual HCL, such as to make use of syntactic sugar operations unavailable in IDE (all participants), to refactor and rearrange GUI-generated block instantiations and connects (also all participants), or as the primary way to produce HCL (PW03 and PW04). In the rest of this section, we report on individual flows and preferences.

PW01 started by sketching out the system architecture block diagram, then instantiated the blocks with GUI actions and connected them with a mix of text editing and GUI actions. Overall, PW01 felt the IDE approach was a good in-between for schematics and HCL, but still requires users to have a baseline competence with both code and circuits.

PW02 similarly instantiated blocks and made connections from GUI actions, but as a more iterative process including refactoring inserted blocks. PW02 specifically noted the helpfulness of the connect interface view filtering by legal connections, and also felt the IDE was a good coupling of code and diagrams.

PW03, on the other hand, used text edits to instantiate some blocks and all connects, noting a preference for copy-paste coding. Uniquely, PW03 made the connections to the display by writing a for-loop iterating through a list of ports. While PW03 did not feel the block diagram generated edits were useful, the library browser was noted for its discoverability of parts and the block visualization was noted for its discoverability of connections.

PW04 similarly also preferred text edits for block instantiation and connection, noting "not being a code snippets person". However, PW04 also mentioned the tightly-integrated block diagram as an invaluable reference of what needed to be connected.

Edit Actions

Where participants used the GUI to write code, they almost always used the caret-based actions instead of the other insertion points suggested in the context menu. P04 noted that this list of alternatives was confusing.

Participants did criticize our interface as being clunky (PW02), such as by requiring multiple selections (both a caret position and navigating to the edited block in the block diagram, PW03) or being very sensitive to caret location (PW04). In a broader sense, both PW01 and PW03 mentioned preserving flow by staying in one interface instead of jumping between text and block diagram. However, these may be more issues of our specific prototype implementation, and it may be possible that tweaks to the interface specifics could produce a more usable flow.

On the other hand, PW04 called out the footprint and pinning interface as something done well, because it directly matches the visual presentation often provided by component documentation. Participants additionally suggested different interfaces for edit actions, such as drag-and-drop for block instantiation (PW01), click-and-drag for making connections (PW04), or improving the existing IDE autocomplete with awareness of the HCL (for example, presenting only connectable ports; PW03).

As a completely different interface, PW01 also suggested an import flow from existing schematic tools, both to use existing libraries written in mainstream tools, and to support users who are more familiar and comfortable working with mainstream tools.

Block Diagram Visualization

For the automatically generated block diagrams, the overall consensus was that it was very usable for writing HCL including reasonable adherence to convention, but still had many rough edges. All participants suggested more usage of symbols, such as ground symbols for the ground connections or part symbols instead of the simple rectangles for blocks. PW03 additionally suggested symbols as a way to manage complexity when zoomed out: dense text could fade out and the entire block could be replaced with a symbol.

PW01, PW02, PW03 all discussed ideas for manual layout constraints, such as grouping blocks together, but also acknowledged it is a hard problem without clear solutions. PW04, on the other hand, felt that a tool for creating presentation-grade diagrams may be out of scope here.

Finally, PW01 and PW02 mentioned layout considerations: PW01's schematic capture flow takes into account rough layout floorplanning when placing symbols, while PW02 felt disconnected from the physical (footprint) view though also believed that more stuff on the block diagram could be cluttering.

Refresh and Speculative Effects

Participants had varied opinions on the manual (user-initiated) recompilation and block diagram update. At one extreme, PW01 preferred continuous compilation to minimize the feedback loop between HCL edits and visual presentation. On the other hand, PW03 felt that the current user-initiated scheme makes sense, crucially preserving diagram stability as text is being edited. PW04 was in-between, feeling that automatic updates could save a few

keystrokes, but the system should be smart about detecting when the user is at a stopping point.

As for speculative effects from block diagram insert or connect operations, participants generally did not notice the details and felt things seemed synchronized.

6.6 Discussion

While we have built a functional but prototype IDE and obtained user feedback, there are important limitations of the user study to keep in mind when interpreting results. Yet, even if qualified, this data can help focus practically useful directions for future work.

Study Limitations

While we believe we accomplished our goal of sampling for range given the diverse observed flows and feedback, the small participant pool does mean the feedback is not exhaustive. However, we do believe it is appropriate in terms of early usability testing for a novel concept and for informing future work.

Additionally, because the entire study was one session per participant, learning effects may still be in play. If participants had used this system for longer, they may have tried and adopted different workflows. This may be especially important for a tool intended to support long-term projects and include professional users.

Overall Takeaways

Overall, we believe a major generalizable takeaway is that graphical editing operations do not need to cover all conceivable code edits – here, participants were able to effectively blend use of GUI tooling with textual HCL edits for operations not supported by the GUI. Consistent with prior work [38], the simplest tools of visualization and library browsing were the ones that were most consistently used. Beyond PCB design, tools working on similar concepts may find it valuable to focus on supporting a few common workflows well before being bogged down by how to support trickier operations in a GUI.

While not all participants preferred the code generation features, others aspects of the IDE were still useful in helping manually write HCL. Perhaps unsurprisingly for a prototype tool, the interactions were not perfect and this may have affected some participants' decisions type out HCL instead, but it also does appear to be partially rooted in personal preference. We believe that tools working with HCLs will need to acknowledge and support diverse and free-form flows, and these user observations and feedback can provide starting points for further iteration.

Furthermore, imperfect techniques that improve responsiveness, like speculative effects and caching, can provide the smooth interface expected of direct manipulation system even with slower compilers. Despite both techniques having unsupported edge cases, which may

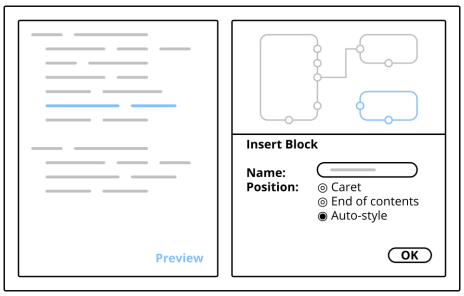

Figure 6.9: **Potential flow-preserving IDE interface concept with insertion options and live preview**, which provides options for code insertion positions as part of the default block insertion process and includes a live preview and an automatic position that takes stylistic factors into consideration.

be fundamentally difficult to resolve especially with a dynamic language like Python, they seemed to have worked well enough in practice for our (albeit somewhat limited) user study. More generally, it may be useful for future work to explore this time-accuracy trade-off in more detail, as well as investigate compromise strategies that provide more accurate results as they become available.

Flow for Edit Actions

While our intent with caret editing was to disambiguate degrees of freedom for edit actions in a mainstream language like Python, in practice participants raised issues with our implementation. However, we believe that the feedback of needing a consistent flow, instead of jumping between HCL and GUI, provides a useful guiding principle for future output-directed manipulation tools in these mainstream but messy languages.

While participants did not use the edit locations in the context menu, avenues for future work may include how those choices can be unobtrusively made part of the default workflow. Perhaps users could be offered the option alongside other parameters like name and provided with a reasonable default option, such as in Figure 6.9. Smarter defaults might be inferred according to style rules, perhaps selected by the user. Furthermore, a live preview of the code to be inserted may help users understand the choice. Alternatively, as in Figure 6.10, edits may also be tracked (or buffered) by the IDE to support smooth sequences of GUI actions, while powerful refactoring tools can help users clean up their code afterward.

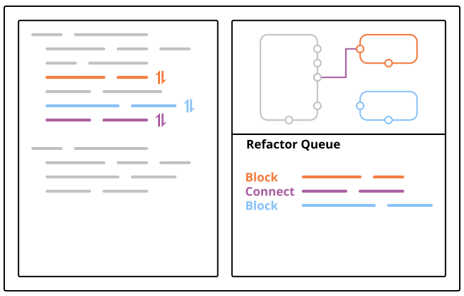

Figure 6.10: **Potential flow-preserving IDE interface concept that tracks inserted blocks for later refactoring**, allowing users to complete a sequence of block insertions without worrying about what the code looks like, then return later to refactor and rearrange the generated code with the help of tools.

Layout

Although participants found our automatically generated block diagram visualization sufficient, there is still much room for improvement. The user feedback provides ideas for specific features, though the ultimate goal would be automated schematic layout from a netlist. While our implementation uses a stock layout algorithm for general hierarchical block diagrams, custom algorithms tailored around electronics conventions may do much better.

Performant Recompilation

Finally, from a more engineering standpoint, speculative effects may require code that is similar to what would be in the compiler, but just different enough to require re-implementation. While the architecturally elegant solution would be to fully recompile after a GUI-inserted code edit, this may not always be performant enough to sustain a sequence of interactive GUI edits. While speculative effects will likely always remain a solution in general, it is worth exploring how far we can get with optimizations such as incremental re-compilation to avoid the complexity of speculative effects.

Chapter 7

Conclusion

7.1 Summary of Contributions

Reiterating the introduction, this dissertation makes the following contributions:

- A formative interview study with 15 participants to assess current practices and problems in board-level design, taking a larger end-to-end ideas-to-device perspective without being limited to current EDA flows (Chapter 4). We found that a significant amount of design work happens before EDA tools come into play - and often without the help of software at all. By the time users get to schematic capture, much of the interesting and creative work has already been done, and schematic capture itself involves a lot of rote transcription.

- A concept for a hardware construction language (HCL) based system that address major issues with current flows revealed in the above formative study, and preliminary user feedback on mockups of this language and supporting tooling (Chapter 4). Though users indicated enthusiasm around the system architecture level of abstraction, they also expressed caution around issues such as dependence on and trust in libraries. We further discuss relevant design principles for future board tools based on our results.

- An implementation of the above system as a Python-embedded HCL, based around a hierarchy block model to scale across multiple levels of abstraction and supporting design re-use through user-definable libraries of blocks (Chapter 5). Blocks can be generators, containing executable code that constructs the block's sub-circuit from higher-level parameters (for example, calculating resistance given a voltage and current specification), while a type system and electronics model supports substitution and refinement (for example, whether a particular step-down converter subcircuit can be used in place of a generic voltage converter).

  We evaluate this system by designing several example boards in this system and with a small user study where participants built boards of their choice. Users were able

to complete designs in our system, and provided a wide range of feedback including suggestions for better tooling and areas of concern such as the higher learning curve.

- Supporting tooling for the above HCL in the form of an integrated development environment (IDE) to bridge the novel code approach with familiar schematic views, by augmenting the standard text editor with a block diagram visualization supporting schematic editor like actions (Chapter 6). As both views are linked – the visualization can be updated from code edits, and code can be generated from schematic like actions – users can easily move between interfaces based on their own preferences and the task at hand.

  We evaluate this interface with a small user study in which participants complete a pre-defined project, and make generalized recommendations for similar tools based on the results. Although users differed in whether they preferred to produce HCL by writing text or through graphical actions, the visualization still provides something for everyone, such as by acting as an intuitive reference for writing HCL.

7.2 Future Work

In addition to the chapter-specific future work, a few overarching directions for future work exist:

- **Model extensions**: the models, both problem structure and electronics model, sit at the core of this HCL, and largely determines how expressive it is. While generators supporting arbitrary code mean that there is almost no limit to what any one component can do, the models set the rules for interactions between components.

  However, proposed model extensions must also be balanced with the costs: these introduce additional complexity into the core compiler, and must play nicely and integrate with other core compiler features.

  - **Core Model**: Although the core model of blocks, ports, links, parameters and constraints on parameters has enabled the example designs shown in the previous chapters, improvements to the core model could make the system much more expressive and further the goal of design re-use. One idea is higher-level blocks, or blocks that take a block type as a parameter. These can enable generic subcircuits, for example a discrete boost converter parameterized by the controller chip, instead of an unique subcircuit class for each converter chip. Other ideas from type systems in programming languages may also be useful: for example, could it make sense to have union and intersection types for hardware design?
  - **Electronics Model**: The current electronics model encodes some of the most common static component ratings and parameters specified with modern devices and automates their checks, but there are many more in a modern datasheet that

can be useful. A large part of this will likely be driven by what specifications are provided by manufacturers, but tools may also drive manufacturers to provide additional data optimized for these automated reasoning systems.

One issue here is balancing what to include in the electronics model that all parts must adhere to, versus what is part of the long tail and not worth the effort. A middle ground may be to have some sort of optional model extensions which parts can implement on an opt-in basis, but can also fail gracefully for parts that are unaware of this extension. How this can work is an open question. Inspiration from programming languages may be helpful, where some languages support community-defined extensions through a compiler plug-in architecture.

- **Cross-Domain Support**: While EDG [65] had a mixed software and electronics model, this dissertation focuses on only the electronics side. Adding (back) in support for software models could help integration into firmware flows, for example generating header files containing pin mappings, linking in interface shims and adapters, or even entire software components.

  More ambitiously would be integrating and linking models from multiple domains, for example mechanical and behavioral, so automated tools can automatically check and enforce consistency — a pain point in current workflows. Furthermore, going back to EDG, a design in one domain (for example, behavioral description) can provide a specification for another domain (for example, the electronics design on a PCB), providing a framework for synthesis.

- **User-defined Extensibility**: While a standardized model is a powerful tool to enable interoperability, it is also likely to be limited to the lowest common denominator. As hinted in the electronics model section, user-extensible model extensions may be a way to support both a base model and custom extensions. This does not need to be limited to electronics, but could be extensions to the core model or additional domains. However, this would need to be done in a way that interoperates with the base model, other user-defined extensions, and ideally supports a path to be added to the standard model.

- **IDE extensions**: while Chapter 6 describing the IDE provides some ideas for future work, there are also cross-cutting concerns:

  - **Visualizations for Extended Models**: While users found the existing automatically generated visualizations acceptable for design and better layout algorithms that adhere to schematic conventions can be an improvement, it is much less clear how to define an intuitive visualization that works across multiple domains. How can multiple domains be shown on the same screen, and how can cross-domain connections and blocks be made apparent? Might some kind of layering approach help - either for always displaying multiple domains, or merely as a visual aid for moving between domains?

  - **Support for Advanced Constructs**: Although we made a conscious decision to restrict the IDE graphical editor features to analogues in modern schematic editors, it may be interesting to explore ideas for supporting the more programmatic hardware construction features. Furthermore, perhaps this could be integrated with improved visualization: for example, perhaps an action could be to array-replicate a component, which would generate a `for`-loop in the code and show stacked elements on the visualization.

    That being said, probably not everything needs a corresponding graphical interface. For example, equations for parameter propagation should probably continue to be defined in code, as opposed to a LabVIEW-style [30] graphical dataflow interface.
  - **User-defined Extensions**: The IDE should also support user-defined model extensions. While anything built on top of the standard core model is already supported (such as custom link and port types), any modifications of the core model may require modifications to the IDE. Conventionally, this may be some kind of plugin architecture, but more lightweight methods that don't require knowledge of IDE internals can help provide user-defined extensions the same degree of tooling support as the base models.
  - **Design Space Exploration**: Currently, all design details still must be resolved by the user, which requires knowledge of the trade-offs to make when choosing between parts. Since the higher level of design captures a design space, a computer could do a much more thorough and rigorous search of the design space. To be useful, this requires the data for making the trade-offs to be encoded in the model and the optimization criteria to be formalized, but ideally in a form that is still accessible to novices. Questions include what data would be important to support these optimizations (for example, component cost? total board area?) and what canned optimization criteria could be useful to novices?

- **Building Community**: Finally, as much of the vision of this system relies on libraries, someone will have to build out those libraries. In the software world, there is a significant open-source community that writes and shares code that is used by others. While there shouldn't be any fundamental barriers to the same philosophy in the board design domain, there are some challenges:

  - **Trust and Quality**: Users in our studies repeatedly mentioned trust and quality issues as a barrier to using community-supplied libraries. While library author is one proxy to assess trust and quality, are there other indicators and mechanisms? For example, automated tests are common in the software and chip domain — what might the same look like for boards, especially when simulation models are not available for the vast majority of parts? It is also important to note that even in the software domain, trust is not a solved problem (even if it is often ignored) and software supply chain is an emerging area of research.

- **Balancing Scale and Agility**: While having comprehensive libraries is necessary for a generalizable and useful tool, it also has the drawback of inertia, especially on the underlying system. For example, with a huge library of electronics parts, it will probably be difficult to make breaking changes to the core electronics model. Yet, iteration is likely key to converging on good solutions, so locking down models early may be undesirable. Are there solutions that both allow scaling and agility? For example, how might optional extensions work with older parts that don't support the extension, and what mechanisms could help and incentive users to update older libraries?

7.3 Conclusion

Ultimately, the goal of tools is to enable people to do more, and it is the hope that these tools for electronics design will provide something for everyone: for existing engineers, a way to work more efficiently by building upon and re-using existing designs without the tedious transcription, and for novices and makers, the ability to build more complex and customized devices that otherwise may not have been feasible.

Chapter 8

Glossary

This section defines some domain-specific terms used throughout the dissertation with the relevant context.

- **Direct Manipulation Interface**: an interface where users are presented with and can manipulate representations of objects, such as moving and connecting electronic components on a screen in a schematic capture program.
- **EDA (Electronics Design Automation)**: the currently mainstream tools for designing electronics, which in general can refer to both chips and boards. For boards, this consists of two parts:
  - **Schematic Capture**: the first step of a board EDA workflow, where users draw the circuit schematic by placing components and connecting their pins. Example in Figure 2.1. This dissertation proposes a replacement for schematic capture as a hardware construction language (HCL).
  - **Board Layout**: the second step of a board EDA workflow, where components from the schematic are placed on a virtual board, and conductive copper traces between their pins are routed. Example in Figure 2.2. This step is out of scope of this dissertation.
- **ERC (Electrical Rules Check)**: an automated check on a schematic, typically based on a coarse system of types on components' pins (for example, Power Input and Power Output), and rules defining illegal connections between them (for example, multiple Power Outputs connected together, or pin that is not connected).
- **DRC (Design Rules Check)**: an automated check on a board layout for physical manufacturability, such as minimum copper width, minimum spacing between copper, and minimum hole sizes.
- **HCL (Hardware Construction Language)**: a subset of hardware description languages (HDLs) which additionally supports user-defined generators where modules

can be defined as a program that generates its implementation based on higher-level parameters. For example, a board HDL might enable an LED module to define the resistor's resistance as a function of the voltage, as opposed to a vanilla HDL which requires a fixed numeric resistance.

- **HDL (Hardware Description Language)**: an approach to electronics design where users define the hardware in a textual format, as opposed to graphical schematics. Currently mainstream practice for chip design, but not board design. Typically supports encapsulating a sub-circuit as a module, and making connections between modules.
- **GUI (Graphical User Interface)**: an interface where users are presented with graphical objects, as opposed to (for example) text interfaces such as for writing code. In this dissertation, this typically refers to visualizations (and interactive tools around those visualizations) generated from a compiled design.
- **IDE (Integrated Development Environment)**: a software tool for software development, commonly augmenting the standard code text editor with additional features such as syntax highlighting, refactoring tools (for example, rename a variable and all its usages), and visualizations (for example, of a class hierarchy).
- **PCB (Printed Circuit Board)**: a common way of building electronics, consisting of a board to which electronic components are soldered down, and containing conductive copper traces to connect those components' pins. Example in Figure 5.3.

Bibliography

[1] Altium. *Altium Designer*. 2018. URL: https://www.altium.com/altium-designer/ (visited on 09/20/2018).

[2] Altium. *Smart Paste*. 2021. URL: https://www.altium.com/documentation/altium-designer/sch-dlg-smartpasteformsmart-paste-ad (visited on 10/20/2021).

[3] 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. Québec City, QC, Canada: ACM, 2017, pp. 331–342. ISBN: 978-1-4503-4981-9. DOI: 10.1145/3126594.3126637. URL: http://doi.acm.org/10.1145/3126594.3126637.

[4] Arduino. *Arduino - Home*. 2018. URL: https://www.arduino.cc (visited on 09/20/2018).

[5] Michal Armoni, Orni Meerbaum-Salant, and Mordechai Ben-Ari. "From Scratch to "Real" Programming". In: *ACM Trans. Comput. Educ.* 14.4 (Feb. 2015). DOI: 10.1145/2677087. URL: https://doi.org/10.1145/2677087.

[6] Autodesk. *EAGLE | PCB Design Software*. 2018. URL: https://www.autodesk.com/products/eagle/overview (visited on 09/20/2018).

[7] Jonathan Bachrach et al. "JITPCB". In: *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE. 2016, pp. 2230–2236. DOI: 10.1109/IROS.2016.7759349.

[8] Hedayat Markus Bayegan and Einar Aas. "An Integrated System for Interactive Editing of Schematics, Logic Simulation and PCB Layout Design". In: *Proceedings of the 15th Design Automation Conference*. DAC '78. Las Vegas, Nevada, USA: IEEE Press, 1978, pp. 1–8. URL: http://dl.acm.org/citation.cfm?id=800095.803058.

[9] H. Beyer and K. Holtzblatt. *Contextual Design: Defining Customer-centered Systems*. Interactive Technologies Series. Morgan Kaufmann, 1998. ISBN: 9781558604117. URL: https://books.google.com/books?id=T8pcH4QjATkC.

[10] Alan Blackwell and Thomas Green. "Notational systems–the cognitive dimensions of notations framework". In: *HCI models, theories, and frameworks: toward an interdisciplinary science. Morgan Kaufmann* (2003).

[11] Tracey Booth et al. "Crossed Wires: Investigating the Problems of End-User Developers in a Physical Computing Task". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI '16. San Jose, California, USA: ACM, 2016, pp. 3485–3497. ISBN: 978-1-4503-3362-7. DOI: 10.1145/2858036.2858533. URL: http://doi.acm.org/10.1145/2858036.2858533.

[12] Cadence. *Allegro PCB Designer*. 2018. URL: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/pcb-design-and-analysis/pcb-layout/allegro-pcb-designer.html (visited on 09/20/2018).

[13] Sébastien Caux. *uConfig*. https://github.com/Robotips/uConfig. 2018.

[14] Joshua Chan, Tarun Pondicherry, and Paulo Blikstein. "LightUp: An Augmented, Learning Platform for Electronics". In: *Proceedings of the 12th International Conference on Interaction Design and Children*. IDC '13. New York, New York, USA: ACM, 2013, pp. 491–494. ISBN: 978-1-4503-1918-8. DOI: 10.1145/2485760.2485812. URL: http://doi.acm.org/10.1145/2485760.2485812.

[15] Ernst Christen et al. "Analog and mixed-signal modeling using the VHDL-AMS language". In: *36th Design Automation Conference*. 1999, pp. 21–25.

[16] circuito.io. *Circuit Design App for Makers- circuito.io*. Feb. 2020. URL: https://www.circuito.io/.

[17] J. Crossley et al. "BAG: A Designer-Oriented Integrated Framework for the Development of AMS Circuit Generators". In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '13. San Jose, California: IEEE Press, 2013, pp. 74–81. ISBN: 9781479910694.

[18] Daniel Drew et al. "The Toastboard: Ubiquitous Instrumentation and Automated Checking of Breadboarded Circuits". In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. UIST '16. Tokyo, Japan: ACM, 2016, pp. 677–686. ISBN: 978-1-4503-4189-9. DOI: 10.1145/2984511.2984566. URL: http://doi.acm.org/10.1145/2984511.2984566.

[19] EDASolver. *EDASolver: Welcome to Functional EDA*. Jan. 2016. URL: https://edasolver.com.

[20] Jonathan Edwards. "Example Centric Programming". In: *SIGPLAN Not.* 39.12 (Dec. 2004), pp. 84–91. ISSN: 0362-1340. DOI: 10.1145/1052883.1052894. URL: https://doi.org/10.1145/1052883.1052894.

[21] A. C. Finch et al. "A Method for Gridless Routing of Printed Circuit Boards". In: *Proceedings of the 22Nd ACM/IEEE Design Automation Conference*. DAC '85. Las Vegas, Nevada, USA: IEEE Press, 1985, pp. 509–515. ISBN: 0-8186-0635-5. URL: http://dl.acm.org/citation.cfm?id=317825.317937.

[22] Eclipse Foundation. *Eclipse Layout Kernel*. 2020. URL: https://www.eclipse.org/elk/ (visited on 01/05/2020).

[23] Pragun Goyal et al. "BoardLab: PCB As an Interface to EDA Software". In: *Proceedings of the Adjunct Publication of the 26th Annual ACM Symposium on User Interface Software and Technology*. UIST '13 Adjunct. St. Andrews, Scotland, United Kingdom: ACM, 2013, pp. 19–20. ISBN: 978-1-4503-2406-9. DOI: 10.1145/2508468.2514936. URL: http://doi.acm.org/10.1145/2508468.2514936.

[24] Saul Greenberg and Bill Buxton. "Usability Evaluation Considered Harmful (Some of the Time)". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '08. Florence, Italy: ACM, 2008, pp. 111–120. ISBN: 978-1-60558-011-1. DOI: 10.1145/1357054.1357074. URL: http://doi.acm.org/10.1145/1357054.1357074.

[25] Gumstix. *Geppetto*. 2018. URL: www.gumstix.com/geppetto/ (visited on 01/02/2020).

[26] Kotaro Hara, Christine Chan, and Jon E. Froehlich. "The Design of Assistive Location-based Technologies for People with Ambulatory Disabilities: A Formative Study". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI '16. San Jose, California, USA: ACM, 2016, pp. 1757–1768. ISBN: 978-1-4503-3362-7. DOI: 10.1145/2858036.2858315. URL: http://doi.acm.org/10.1145/2858036.2858315.

[27] R. Harjani, R. A. Rutenbar, and L. R. Carley. "OASYS: a framework for analog circuit synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8.12 (1989), pp. 1247–1266.

[28] Brian Hempel, Justin Lubin, and Ravi Chugh. "Sketch-n-Sketch: Output-Directed Programming for SVG". In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. New Orleans, LA, USA: Association for Computing Machinery, 2019, pp. 281–292. ISBN: 9781450368162. DOI: 10.1145/3332165.3347925. URL: https://doi.org/10.1145/3332165.3347925.

[29] Accellera System Initiative. "Verilog-AMS Language Reference Manual". In: (2014).

[30] National Instruments. *LabVIEW*. 2020. URL: http://www.ni.com/labview (visited on 07/28/2021).

[31] Texas Instruments. *Voltage Divider Calculator*. 2021. URL: https://www.ti.com/download/kbase/volt/volt_div3.htm (visited on 10/26/2021).

[32] Texas Instruments. *WEBENCH® Power Designer*. 2021. URL: https://www.ti.com/design-resources/design-tools-simulation/webench-power-designer.html (visited on 10/26/2021).

[33] A. Izraelevitz et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[34] Rushil Khurana and Steve Hodges. "Beyond the Prototype: Understanding the Challenge of Scaling Hardware Device Production". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–11.

[35] KiCad. *KiCad EDA*. 2018. URL: http://kicad-pcb.org/ (visited on 09/20/2018).

[36] Yoonji Kim et al. "VirtualComponent: a Mixed-Reality Tool for Designing and Tuning Breadboarded Circuits". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–13.

[37] André Knörig, Reto Wettach, and Jonathan Cohen. "Fritzing: A Tool for Advancing Electronic Prototyping for Designers". In: *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*. TEI '09. Cambridge, United Kingdom: Association for Computing Machinery, 2009, pp. 351–358. ISBN: 9781605584935. DOI: 10.1145/1517664.1517735. URL: https://doi.org/10.1145/1517664.1517735.

[38] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. "The Road to Live Programming: Insights from the Practice". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 1090–1101. ISBN: 9781450356381. DOI: 10.1145/3180155.3180200. URL: https://doi.org/10.1145/3180155.3180200.

[39] F. B. Lavering. "AUTO CARD Automated Printed Circuit Board Design". In: *Proceedings of the SHARE Design Automation Workshop*. DAC '64. New York, NY, USA: ACM, 1964, pp. 9.1–9.29. DOI: 10.1145/800265.810745. URL: http://doi.acm.org/10.1145/800265.810745.

[40] Ultra Librarian. 2020. URL: https://www.ultralibrarian.com/ (visited on 07/10/2020).

[41] Richard Lin and Bjoern Hartmann. "Opportunities and Challenges for Circuit Board Level Hardware Description Languages". In: *Human Aspects of Types and Reasoning Assistants (HATRA 2020) Workshop*. 2020. URL: https://arxiv.org/abs/2011.08242.

[42] Richard Lin et al. "Beyond Schematic Capture: Meaningful Abstractions for Better Electronics Design Tools". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019. ISBN: 9781450359702. DOI: 10.1145/3290605.3300513. URL: https://doi.org/10.1145/3290605.3300513.

[43] Richard Lin et al. "Polymorphic Blocks: Unifying High-Level Specification and Low-Level Control for Circuit Board Design". In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 529–540. ISBN: 9781450375146. DOI: 10.1145/3379337.3415860. URL: https://doi.org/10.1145/3379337.3415860.

[44] Richard Lin et al. "Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages". In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. UIST '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1039–1049. ISBN: 9781450386357. DOI: 10.1145/3472749.3474804. URL: https://doi.org/10.1145/3472749.3474804.

[45] Jo-Yu Lo et al. "AutoFritz: Autocomplete for Prototyping Virtual Breadboard Circuits". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019. ISBN: 9781450359702. DOI: 10.1145/3290605.3300633. URL: https://doi.org/10.1145/3290605.3300633.

[46] John Maloney et al. "The Scratch Programming Language and Environment". In: *ACM Trans. Comput. Educ.* 10.4 (Nov. 2010). DOI: 10.1145/1868358.1868363. URL: https://doi.org/10.1145/1868358.1868363.

[47] Ola A. Marvik. "An Interactive Routing Program with On-line Clean-up of Sketched Routes". In: *Proceedings of the 16th Design Automation Conference*. DAC '79. San Diego, CA, USA: IEEE Press, 1979, pp. 500–505. URL: http://dl.acm.org/citation.cfm?id=800292.811760.

[48] Andrew J. Matthews. "A Human Engineered PCB Design System". In: *Proceedings of the 14th Design Automation Conference*. DAC '77. Piscataway, NJ, USA: IEEE Press, 1977, pp. 182–186. URL: http://dl.acm.org/citation.cfm?id=800262.809124.

[49] Sean McDirmid. "The Future of Programming will be Live". In: *Curry On!* Curry On! 2016. Rome, 2016. URL: https://www.youtube.com/watch?v=bnqkglrSqrg.

[50] Sean McDirmid. "Usable Live Programming". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 53–62. ISBN: 9781450324724. DOI: 10.1145/2509578.2509585. URL: https://doi.org/10.1145/2509578.2509585.

[51] Will McGrath et al. "Bifröst: Visualizing and Checking Behavior of Embedded Systems Across Hardware and Software". In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17. Québec City, QC, Canada: ACM, 2017, pp. 299–310. ISBN: 978-1-4503-4981-9. DOI: 10.1145/3126594.3126658. URL: http://doi.acm.org/10.1145/3126594.3126658.

[52] David A. Mellis et al. "Engaging Amateurs in the Design, Fabrication, and Assembly of Electronic Devices". In: *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*. DIS '16. Brisbane, QLD, Australia: ACM, 2016, pp. 1270–1281. ISBN: 978-1-4503-4031-1. DOI: 10.1145/2901790.2901833. URL: http://doi.acm.org/10.1145/2901790.2901833.

[53] Mentor. *Error Reduction in the Design Definition Phase*. 2020. URL: https://www.mentor.com/pcb/multimedia/player/error-reduction-in-the-design-definition-phase-0db48520-5d96-43ba-a208-d10513b742c6 (visited on 07/04/2020).

[54] Mentor. *Get to Market Fast and First with Reusable Circuit Blocks*. 2020. URL: https://www.mentor.com/pcb/resources/overview/get-to-market-fast-and-first-with-reusable-circuit-blocks-981762c9-485a-416f-877c-b6dbf7622c45 (visited on 07/10/2020).

[55] Mentor. *Xpedition Enterprise*. 2018. URL: https://www.mentor.com/pcb/xpedition/ (visited on 09/20/2018).

[56] Mentor. *Xpedition Valydate Schematic Analysis*. 2020. URL: https://www.mentor.com/pcb/xpedition/schematic-analysis/ (visited on 07/03/2020).

[57] Devon J. Merrill, Jorge Garza, and Steven Swanson. "Echidna: Mixed-Domain Computational Implementation via Decision Trees". In: *Proceedings of the ACM Symposium on Computational Fabrication*. SCF '19. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2019. ISBN: 9781450367950. DOI: 10.1145/3328939.3329004. URL: https://doi.org/10.1145/3328939.3329004.

[58] Devon J. Merrill and Steven Swanson. "Reducing Instructor Workload in an Introductory Robotics Course via Computational Design". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 592–598. ISBN: 9781450358903. DOI: 10.1145/3287324.3287506. URL: https://doi.org/10.1145/3287324.3287506.

[59] MitjaNemec. *KiCad Action Plugins*. 2021. URL: https://github.com/MitjaNemec/Kicad_action_plugins (visited on 10/20/2021).

[60] Martez E. Mott et al. "Understanding the Accessibility of Smartphone Photography for People with Motor Impairments". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 520:1–520:12. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3174094. URL: http://doi.acm.org/10.1145/3173574.3174094.

[61] Brant Nelson, Brad Riching, and Josh Mangelson. *Using a Custom-Built HDL for Printed Circuit Board Design Capture*. PCB West 2012 Presentation. 2012.

[62] Pierluigi Nuzzo et al. "A contract-based methodology for aircraft electric power system design". In: *IEEE Access* 2 (2013), pp. 1–25.

[63] Zachariah Peterson. *Quickly Replicate Circuits with a Connection Room in Your PCB Layout*. 2021. URL: https://resources.altium.com/p/connection-room (visited on 10/20/2021).

[64] David Rager and Herb Weiner. "The Design of a Dense PCB Using an Interactive DA System". In: *SIGDA Newsl.* 8.2 (June 1978), pp. 50–58. ISSN: 0163-5743. DOI: 10.1145/1061458.1061464. URL: http://doi.acm.org/10.1145/1061458.1061464.

[65] Rohit Ramesh et al. "Turning Coders into Makers: The Promise of Embedded Design Generation". In: *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication*. SCF '17. Cambridge, Massachusetts: ACM, 2017, 4:1–4:10. ISBN: 978-1-4503-4999-4. DOI: 10.1145/3083157.3083159. URL: http://doi.acm.org/10.1145/3083157.3083159.

[66] Patrick Rein et al. "How Live Are Live Programming Systems? Benchmarking the Response Times of Live Programming Environments". In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. PX/16. Rome, Italy: Association for Computing Machinery, 2016, pp. 1–8. ISBN: 9781450347761. DOI: 10.1145/2984380.2984381. URL: https://doi.org/10.1145/2984380.2984381.

[67] Mitchel Resnick et al. "Design principles for tools to support creative thinking". In: (2005).

[68] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. "Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 271–286. ISBN: 9781450362405. DOI: 10.1145/3297858.3304010. URL: https://doi.org/10.1145/3297858.3304010.

[69] Hiroshi Shiraishi et al. "ICAD/PCB: Integrated Computer Aided Design System for Printed Circuit Boards". In: *Proceedings of the 19th Design Automation Conference*. DAC '82. Piscataway, NJ, USA: IEEE Press, 1982, pp. 727–732. ISBN: 0-89791-020-6. URL: http://dl.acm.org/citation.cfm?id=800263.809282.

[70] Ben Shneiderman. "Creativity Support Tools". In: *Commun. ACM* 45.10 (Oct. 2002), pp. 116–120. ISSN: 0001-0782. DOI: 10.1145/570907.570945. URL: http://doi.acm.org/10.1145/570907.570945.

[71] Ben Shneiderman. "Creativity support tools: A grand challenge for HCI researchers". In: *Engineering the User Interface* (2009), pp. 1–9.

[72] Ben Shneiderman. "Creativity Support Tools: Accelerating Discovery and Innovation". In: *Commun. ACM* 50.12 (Dec. 2007), pp. 20–32. ISSN: 0001-0782. DOI: 10.1145/1323688.1323689. URL: http://doi.acm.org/10.1145/1323688.1323689.

[73] *SKiDL*. 2021. URL: https://github.com/devbisme/skidl (visited on 10/26/2021).

[74] Haven Skinner et al. "LiveSim: A Fast Hot Reload Simulator for HDLs". In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2020, pp. 126–135. DOI: 10.1109/ISPASS48437.2020.00028.

[75] Sparkfun. *À La Carte*. 2020. URL: https://alc.sparkfun.com/ (visited on 12/16/2020).

[76] Evan Strasnick, Maneesh Agrawala, and Sean Follmer. "Scanalog: Interactive design and debugging of analog circuits with programmable hardware". In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 2017, pp. 321–330.

[77] Evan Strasnick, Sean Follmer, and Maneesh Agrawala. "Pinpoint: A PCB Debugging Pipeline Using Interruptible Routing and Instrumentation". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–11.

[78] Tabula. *Tabula*. 2020. URL: https://tabula.technology/ (visited on 01/05/2020).

[79] Steven L. Tanimoto. "A perspective on the evolution of live programming". In: *2013 1st International Workshop on Live Programming (LIVE)*. 2013, pp. 31–34. DOI: 10.1109/LIVE.2013.6617346.

[80] Maryam Tohidi et al. "Getting the Right Design and the Design Right". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. Montréal, Québec, Canada: ACM, 2006, pp. 1243–1252. ISBN: 1-59593-372-7. DOI: 10.1145/1124772.1124960. URL: http://doi.acm.org/10.1145/1124772.1124960.

[81] Chiuan Wang et al. "CircuitStack: Supporting Rapid Prototyping and Evolution of Electronic Circuits". In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. UIST '16. Tokyo, Japan: ACM, 2016, pp. 687–695. ISBN: 978-1-4503-4189-9. DOI: 10.1145/2984511.2984527. URL: http://doi.acm.org/10.1145/2984511.2984527.

[82] Sheng-Hong Wang et al. "LiveHD: A Productive Live Hardware Development Flow". In: *IEEE Micro* 40.4 (2020), pp. 67–75. DOI: 10.1109/MM.2020.2996508.

[83] Jeremy Warner et al. "ElectroTutor: Test-Driven Physical Computing Tutorials". In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 2018, pp. 435–446.

[84] R.S. Weiss. *Learning From Strangers: The Art and Method of Qualitative Interview Studies*. Free Press, 1995. ISBN: 9781439106983. URL: https://books.google.com/books?id=i2RzQbiEiD4C.

[85] Te-Yen Wu et al. "CircuitSense: Automatic Sensing of Physical Circuits and Generation of Virtual Circuits to Support Software Tools." In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17. Québec City, QC, Canada: ACM, 2017, pp. 311–319. ISBN: 978-1-4503-4981-9. DOI: 10.1145/3126594.3126634. URL: http://doi.acm.org/10.1145/3126594.3126634.

[86] Te-Yen Wu et al. "CurrentViz: Sensing and Visualizing Electric Current Flows of Breadboarded Circuits". In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17. Québec City, QC, Canada: ACM, 2017, pp. 343–349. ISBN: 978-1-4503-4981-9. DOI: 10.1145/3126594.3126646. URL: http://doi.acm.org/10.1145/3126594.3126646.

[87] Apostolos V. Zarras et al. "And the Tool Created a GUI That Was Impure and Without Form: Anti-Patterns in Automatically Generated GUIs". In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. EuroPLoP '18. Irsee, Germany: Association for Computing Machinery, 2018. ISBN: 9781450363877. DOI: 10.1145/3282308.3282333. URL: https://doi.org/10.1145/3282308.3282333.

[88] Junyi Zhu et al. "MorphSensor: A 3D Electronic Design Tool for Reforming Sensor Modules". In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. UIST '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 541–553. ISBN: 9781450375146. DOI: 10.1145/3379337.3415898. URL: https://doi.org/10.1145/3379337.3415898.