Better Embedded Design Tools with Automated Reasoning

By

Rohit Ramesh

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:

Prabal Dutta, Chair
Alice Agogino
Bjoern Hartmann
Sanjit Seshia

Fall 2023

Better Embedded Design Tools with Automated Reasoning

Copyright 2023
By
Rohit Ramesh

Abstract

Better Embedded Design Tools with Automated Reasoning

By

Rohit Ramesh

Doctor of Philosophy in Computer Science

University of California, Berkeley

Prabal Dutta, Chair

Contemporary tools for the design of embedded systems, task-specific electronic devices, are built on a paradigm that has not fundamentally changed since the era of pen-and-paper drafting despite new computational tools that enable new, better workflows for designers and engineers. Embedded systems are the glue we use to connect the digital and physical worlds, letting us leverage the connectivity and automation of computers to solve problems from home automation to battlefield awareness. More user friendly tooling would reduce skill requirements, speed up design cycles, and allow more people to solve their problems with embedded systems. Contemporary Electronic Design Automation (EDA) tools are designed around a single step in the design process, board layout, where an electrical schematic is turned into a design for a printed circuit board, the copper and fiberglass base that connects all the other components in a system. However, engineers go through a series of phases before they reach board layout: exploring the problem they are trying to solve, sketching out a high-level system architecture, and refining that into a well-defined electrical circuit. Better EDA tools would fit more naturally into this workflow, existing to support users in earlier phases, presenting them with information as it becomes relevant, and automating routine or repetitive work.

This dissertation describes two such tools, both built by formulating the design process in mathematical terms and using algorithms to reason about our formalisms, all while wrapped in user-friendly interfaces. The first tool, Embedded Design Generation (EDG) is a proof-of-concept system meant to push the limits of automation in the embedded design process. Given a high-level specification for a device it uses satisfiability solvers to synthesize, from whole cloth, a design meeting that spec. The second, Polymorphic Blocks, uses block diagrams to represent designs in arbitrary stages of construction and propagator semantics for error checking, predictive suggestions, and other features.

Contents

Contents **i**

1 Introduction **1**
1.1 The Modern Embedded Development Process . . . 3
1.1.1 Specification Finding . . . 4
1.1.2 System Architecture Development . . . 5
1.1.3 Schematic Capture . . . 7
1.1.4 Board Layout . . . 10
1.2 Evaluating Embedded Development Tools . . . 11
1.3 Using Automated Reasoning . . . 12
1.4 Summary . . . 13
1.5 Statement of Prior Publication . . . 14

2 Embedded Design Generation **15**
2.1 Introduction . . . 15
2.2 Related Work . . . 17
2.2.1 General EDA . . . 17
2.2.2 PBD and Domain-Specific Tools . . . 18
2.3 Methodology . . . 19
2.4 EDG Prototype Architecture . . . 22
2.4.1 Blocks, Links, and Ports . . . 23
2.4.2 Type Signatures . . . 23
2.4.3 Design Space Model . . . 25
2.4.4 SMT encoding, solving, and decoding . . . 26
2.5 Type System Design . . . 26
2.5.1 Software and Hardware Modeling . . . 26
2.5.2 Ports and Links . . . 27
2.5.3 Components . . . 28
2.6 Evaluation . . . 29
2.6.1 Basic Synthesis Tasks . . . 31
2.6.2 Inferring Missing Design Elements . . . 32
2.6.3 Preservation of Function . . . 32

2.6.4 Performance . . . 32
2.7 Discussion . . . 33
2.7.1 Performance and Optimization . . . 33
2.7.2 Type System Fidelity . . . 34
2.7.3 Usability . . . 35
2.8 Conclusion . . . 35

3 Polymorphic Blocks **37**
3.1 Introduction . . . 37
3.1.1 Statement on Author Contributions . . . 39
3.2 Related Work . . . 39
3.2.1 Electronics and HCI . . . 39
3.2.2 PCB Design Tools . . . 39
3.2.3 Chip Design and Hardware Description Languages . . . 40
3.3 System Design . . . 41
3.3.1 Block Diagram Model . . . 42
3.3.2 Electronics Model and Libraries . . . 44
3.3.3 Hardware Description Language . . . 45
3.3.4 Visualization and Refinement Interface . . . 48
3.3.5 Board Generation . . . 48
3.4 System Implementation . . . 48
3.4.1 Compiler Structure . . . 48
3.4.2 Block Diagram Layout . . . 49
3.5 Example Applications . . . 49
3.5.1 Simon . . . 49
3.5.2 Datalogger . . . 50
3.6 User Study: Methodology . . . 51
3.6.1 Participants . . . 51
3.6.2 Structure . . . 51
3.7 User Study: Results . . . 52
3.7.1 Project: Power Meter . . . 52
3.7.2 Project: Thermistor Reader . . . 52
3.7.3 Project: Multifunction Instrument . . . 53
3.7.4 Advantages . . . 53
3.7.5 Limitations . . . 55
3.7.6 Part Building . . . 56
3.7.7 Graphical Interfaces . . . 56
3.7.8 Design Time . . . 56
3.8 Limitations and Future Work . . . 57
3.8.1 Library-Based Approach . . . 57
3.8.2 Electronics Model . . . 57
3.8.3 Users and User Study . . . 57

3.8.4 Graphical Interfaces 58
3.9 Conclusion 58

4 Conclusion **60**

Bibliography **62**

Acknowledgments

I would like to start by thanking Prabal Dutta, my advisor for the whole of my nine years as a PhD student. His support and advice has been invaluable. I would not have made it this far without his insight into research topics, engineering decisions, and making my work presentable to an audience.

Professors Sanjit Seshia and Edward A. Lee also have my thanks for taking the time to speak to me about the more formal, mathematical side of my work. Their efforts shaped my approach to modeling messy, real-world problems in a rigorous, efficient way.

Of my fellow students, I would like start by singling out Richard Lin. He has focused on the same problems as me but approached them from a very different, yet highly complementary, direction. His dedication to usability and user testing has meant our joint work is more approachable to a wider audience. Without his influence I would likely be building technically interesting tools that nobody outside of a small circle would be able to use. Instead, I've been able to contribute to a platform that has the potential to help a wide audience.

Lastly, I'd like to thank my fellow members of Lab11. The lab has helped keep me grounded and pulled me out of my shell when I've needed it most. In particular Meghan Clark, Pat Panutto, and Branden Ghena took it upon themselves to mentor and support me whenever I was having a difficult time. I wouldn't have made it here without them all and I am immensely grateful.

This work was supported in part by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; the National Science Foundation under grant(s) CPS-1239031; the NSF/Intel Partnership on Cyber-Physical System Security and Privacy under award proposal title "Synergy: End-to-End Security for the Internet of Things: NSF Proposal No. 1505684"; DARPA CRAFT; ExCAPE: Expeditions in Computer Augmented Program Engineering (NSF grant CCF-1139138); NSF CNS 1505773, CNS 1822332, 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; and in part with funds from the Paul and Judy Gray Alumni Presidential Chair in Engineering Excellence.

Chapter 1

Introduction

Embedded systems are a critical part of the modern world because they are the interface between digital systems and the physical environment, appearing in a massive array of forms and with many diverse uses. Programmatic observation and control of the real world is now a near-ubiquitous fact of life. Even the simplest modern microwave is an embedded system; it contains electronic components that sense and manipulate the environment, those components are controlled by a processor, and the device as a whole exists to do a small set of tasks. Unlike general purpose systems like computers and smartphones, which are built to run arbitrary applications, embedded systems are built to perform a narrow range of tasks. This focus means that an embedded system for home automation, while still interacting with the environment and containing a processor, will look nothing like a drone or a robot.

Embedded development tools have to handle not just diversity in system design, but the corresponding diversity in approaches to design. What matters to a developer can vary wildly based on their project. Common metrics like system cost and time-to-market exist alongside metrics that appear less frequently, like aesthetic beauty. Developers also choose to prioritize different decisions; where one developer might start by dividing their design into high-level building blocks, another may zoom in on one critical component and expand outwards from there. Even critical elements like testing vary based on factors such as required turn-around time, budget, or availability of simulation tools.

Modern embedded development tools are built around a single, easily-modeled phase in the design process, ignoring the creative core of embedded development. Tools like KiCad [29], Eagle [3], and Altium [1] exist to support the design of printed circuit boards (PCBs), while most of the creative decision making happens as users design circuits. Contemporary embedded tooling provides only bare-bones assistance with parts selection, error checking, design organization, and other aspects of circuit design. This lack of meaningful support means that users require more knowledge and skill to design embedded systems than is otherwise neccesary.

Embedded development tools built using techniques from automated reasoning can better serve designers by intelligently presenting useful information, and automating rote work while preserving their freedom in how to approach the design process. We propose a methodology

that accomplishes this by modeling system designs mathematically, such that an incomplete or in-progress design simply corresponds to an unfinished model. Automated reasoning, built to analyze general mathematical models, can be incorporated into a backend that sees no difference between designs in any state of completion and produces inferences regardless. Embedded development tools can then manage the correspondence between an easy-to-use, user-facing frontend and the underlying mathematical model. This allows us to present inferences from the backend in the form of auto-completion suggestions, context-aware search, robust error checking, and other desirable features without artificially limiting how users interact with our tools or forcing them into a narrow workflow.

We evaluate our methodology along three core axes: automation, salience and ergonomics. Automation is the degree to which the user has to do unnecessary or repetitive work, with more wasted effort meaning a less automated tool. Salience considers what information is presented to the user and how useful it is. Tools with good salience proactively show the user information as it is needed, without polluting their environment with noise. Ergonomics looks at how well the tool fits a user's workflow and mental models, specifically whether they have to divert from a "natural" process in order to accommodate a tool's idiosyncrasies.

Our methodology can improve on contemporary practice because automated reasoning tools can work with a design flexibly, without introducing artificial constraints that would be reflected onto users. The field of automated reasoning broadly covers algorithms that can perform mathematical reasoning tasks without human intervention. This runs the gamut from tools that evaluate and simplify systems of equations to more complex algorithms that can search for proofs or solve constraint satisfaction problems. Our work relies on the fact that, if a complete design can be expressed as a set of mathematical statements, incomplete designs can be seen as sets missing some of those statements. Automation comes from tools that can look at a set of statements and generate new statements, pushing an incomplete design closer to completion. Certain inferences, like the existence of an error, are produced only when salient because the very fact that you can derive an error is evidence that the error is relevant. Different approaches to the design process present as different ways to assemble a set of statements; ergonomic tools are then a byproduct of how any set is a valid target for automated reasoning. Automated reasoning is a fundamental part of how our methodology produces usable design tools.

This dissertation covers two tools built with this philosophy. The first, Embedded Design Generation (EDG), is a proof-of-concept tool meant to push the boundaries of automation by translating high-level requirements and a space of possible designs into a constraint satisfaction problem that a satisfiability solver can solve. The second, Polymorphic Blocks (PolyBlocks), builds a design support tool around block diagrams, translating those diagrams into equational relationships that a propagator network can evaluate in order to provide features like auto-complete and robust error checking.

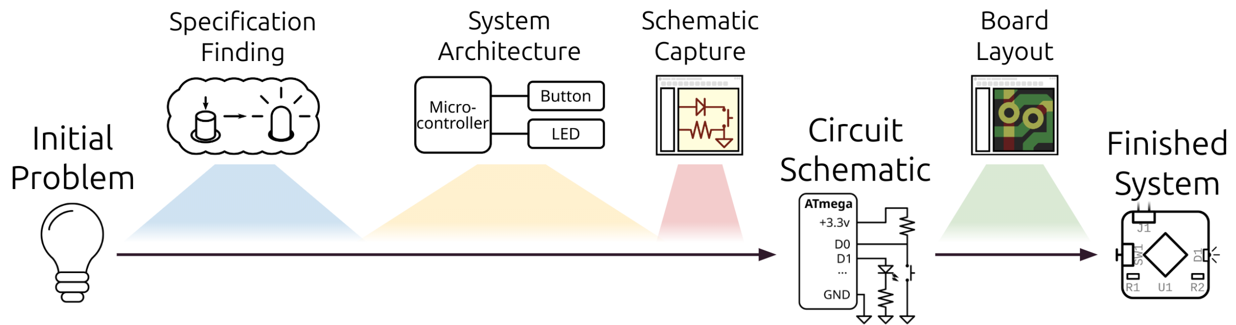


Figure 1.1: **The contemporary embedded development process consists of four major steps.** *Specification Finding* explores an initial problem statement, making the project goals more explicit and evaluating possible implementation strategies. *System Architecture* is the meat of the design process where a high-level sketch of a design is refined into a concrete circuit schematic. *Schematic Capture* has designers performing data-entry, digitizing the schematic for later use. Finally, *Board Layout* is when the schematic is turned into a design for a Printed Circuit Board (PCB), suitable for manufacture. In practice there is some flexibility in the process, with backtracking being common and board layout sometimes dropped entirely when no PCB is needed.

1.1 The Modern Embedded Development Process

The existing paradigm for circuit design follows the same flow as any other design process, moving from a highly-abstract initial conception of a device through ever more concrete steps until the device is sufficiently well defined that it can be fabricated. The process, as shown in figure fig. 1.1, flows from an initial problem statement or specification, usually rendered in non-formal, human-readable language, that defines what the designer is trying to achieve. This problem statement usually is not sufficiently well defined to serve as a specification for the rest of the work. During specification finding designers gather information and explore possible solutions, producing requirements that define a high-level strategy for future engineering efforts. Commonly this leads to a system architecture design phase where a high-level, abstract sketch of the target system, usually in the form of a block diagram, is refined into a circuit diagram that explicitly defines all the components used and how they are electrically connected. This is where much of the creative and skilled work in the embedded design process lies; engineers draw on information from datasheets, component libraries, and other sources, as well as their own skill and experience, to ensure they produce a functioning design that satisfies their requirements. Schematic capture is where Electronic Design Automation (EDA) tool suites like Eagle and KiCad enter the development process. These initial tools act an entry point, allowing designers to digitize their circuits for future steps in the process. Such suites provide tools to design Printed Circuit Boards (PCBs): fiberglass panels which components can be soldered into, with copper traces connecting those

components as specified by the schematic. PCB designs (also called board layouts) can then be manufactured, providing the physical medium to which components are soldered, creating a finished embedded system.

Our model of the design process is based on Lin et al. [35], where we interviewed 15 participants in a semi-structured manner. Starting with background information, including motivations, designs, and views on the EDA process, we questioned them on their process as they moved from idea to final device. Based on those responses, we then went into depth on each step of that process, 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 [6], 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 or lost files; we asked them to either visualize their designs or consider 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 [58]. 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, distilling them into the model in fig. 1.1.

1.1.1 Specification Finding

A project's initial problem statement will rarely be well-defined enough for implementation work to begin. Instead designers start by gathering preliminary information in a process we call *Specification Finding*. The goal is to go from a broad idea—for example, wanting to measure crop growth in a farm—to a more concrete set of requirements—say, measuring stalk height and water retention at least once every day. Better defined requirements guide future steps in the development process and shape the implementation of the target device.

Projects go about choosing requirements and specifications in myriad ways, as no one technique or class of techniques, is universally appropriate. Someone designing a cash register may build cardboard mockups and test how comfortable they are to use. Someone else, designing industrial automation systems, might spend time taking measurements in a factory to know what extremes of temperature and vibration their device has to survive. The requirements important to a project vary wildly, leading to a diversity of techniques for discovering them.

No one tool or descriptive format can capture a useful range of high-level specifications while also connecting to lower-level tools that are more aware of the electronic domain. Instead, we find that participants use a wide variety of representations that don't directly connect to lower level tools, like lists, whiteboard drawings, and slide decks. Participants often used multiple representations, spanning multiple abstraction levels, in concert; one project's

specifications included, for instance, checklists in shared documents alongside datasheets for parts that needed to be in the final design.

Importantly, specification finding is not a static process and the requirements themselves change over time. Participants noted that specifications should be treated as living documents that are updated as the design process unearths new complications. Some changes, like those triggered when designers discover a requirement is impractical, can result in backtracking that touches every part of a project. In other situations, participants described realizing that a requirement they chose didn't actually solve their ultimate problem, forcing them to rethink what needed to be in their specification.

Our participants were skeptical of domain specific tools for specification finding because of its fluid nature and how it touches every other step in the design process. Such a tool would have to be able to encode broad requirements for a system, reason about how an implementation affects those requirements, and deal with a constant process of back-and-forth as user observations, the current system's design, and a living specification influence each other. While this might be possible for a carefully chosen subset of requirements, it is not possible for the general case. This becomes obvious when you consider some of the more novel requirements our participants discussed, like aesthetic beauty and ease-of-use. Allowing designers to handle these concerns themselves is better than trying to force specification finding into the more rigid framework a support tool would require.

1.1.2 System Architecture Development

System architecture is the meat of the embedded development process; going from a sketch of a system to a finalized circuit requires repeatedly evaluating design alternatives, incorporating external information, ensuring system correctness, and making creative choices. Engineers start this phase with a vague idea of how to implement their device. Most of the details are left undefined and what is known can vary wildly based on the designer and project. System architecture starts a process of gradual refinement where details of implementation are incrementally filled in: splitting a design into separate regions, choosing components, connecting interfaces, and so on. By the end, they will have a complete, unambiguous circuit design, which serves as a bottleneck for the rest of the embedded design process. During system architecture, designers are constrained by their specification, available resources, the results of testing, and a wide variety of other concerns they must balance. After the system architecture step, designers are constrained by their circuit design, with other factors playing comparatively small roles.

Despite the multifaceted balancing act that designers must perform, they represented designs in the same way across projects with system architecture diagrams. Architecture diagrams are, fundamentally, block diagrams. Blocks themselves can range from capturing a single component, such as a resistor or LED, to representing large portions of a design, like the power circuitry or all of the sensors in a system. Links between blocks represent connected interfaces, flows of some resource between the connected blocks, like power through some

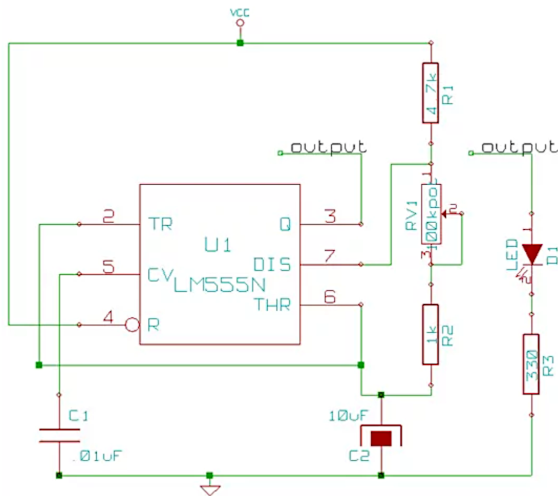
wires or data on a bus. As with blocks, links can be precise, capturing single wires in the circuit, or more general, as with an interface for some "data" that has been left unspecified.

Architecture diagrams, and the elements within them, exist on a spectrum from abstract and ambiguous, with critical information missing, to complete and concrete, unambiguous, with all the information needed for future steps. At their most abstract one finds blocks for "power" and links carrying "signals". This is common during the early system architecture phase, where ambiguous parts of a diagram serve as placeholders for later work. Over the course of the design process, architecture diagrams become more concrete. Abstract blocks are replaced with better defined versions, until they represent individual electronic components. The same applies for links, which become individual wires connecting pins. A finished design contains a schematic within it, like the one in fig. 1.2a, that can be extracted for use elsewhere.

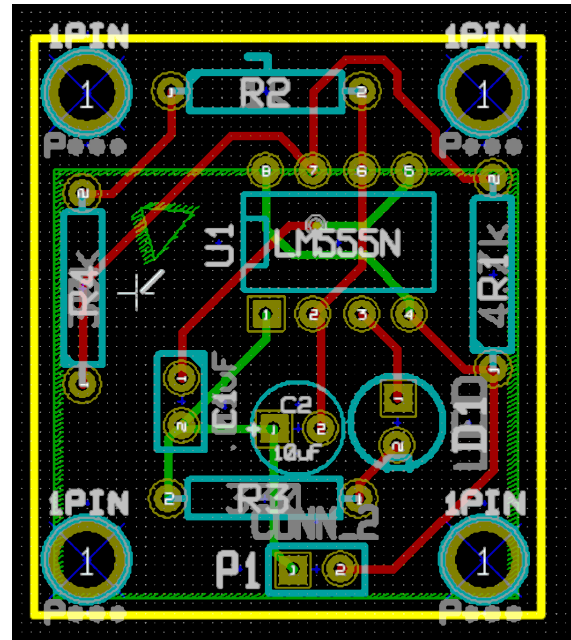
For many of our participants, the process of designing a circuit *is* the process of changing and refining the architecture diagram. Each of the creative choices made by a designer are captured in incremental changes to the diagram. Designing a power architecture is the same as going into the corresponding block and populating it with sub-blocks for battery management, voltage regulation, or anything else needed to handle the parent power block's responsibilities. Choosing a communications bus is the same as annotating a connection for "sensor data" with a specific protocol, possibly even splitting it into the individual wires that protocol uses. Picking parts is the same as changing a block's name from something more abstract, like "microcontroller" or "camera", to the part number for a specific component. Over time, an architecture diagram will slowly change from a gross overview of a design to a fully-specified representation of the final circuit with all the information needed for board layout.

As with specification finding, participants worked with the system architecture using unspecialized tools like whiteboards, graphics software, PowerPoint, or Vizio. Even those participants who used schematic capture tools here mentioned that they did so in a non-functional capacity, as block diagram drawing tools, rather than tools that could assist in any domain specific way.

However, unlike specification finding, which requires generic tools because of the diversity of forms a specification can take, participants near-universally represented the system architecture as a block diagram. The example diagrams provided by our study participants tended to be very similar both structurally and representationally. Participants tended to divide designs into similar functional components; blocks for power supplies, processing, sensors, and actuators were extremely common. Likewise, participants chose to capture similar information in their diagrams; the data moving over a bus was usually preserved, while details like the bus' full protocol and connections to ground were often dropped. Given this relative homogeneity, at least when compared to a project's specifications, it is surprising that assistive tools for system architecture are not in common use.



(a) Circuit Schematic



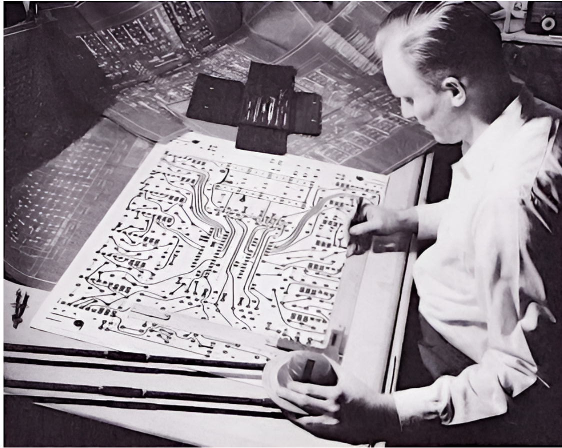
(b) Board Layout

Figure 1.2: **A blinking LED circuit** [50]. The schematic (a) is depicted with standard symbols for the components (red, blue labels); their interfaces, here called pads (red); and the connections between them, here wires (green). The corresponding board (b) also has components (blue, grey labels); interfaces, here pads (gold); and connections, here traces (red and green).

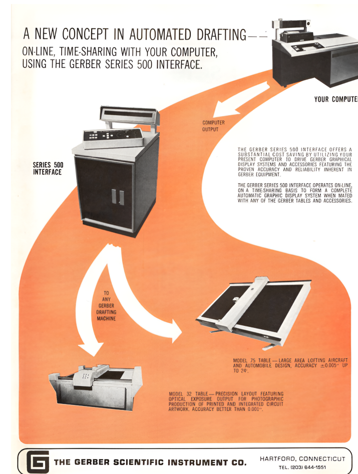
1.1.3 Schematic Capture

Once a circuit has been designed in the system architecture phase, it needs to be digitized for later steps with schematic capture software. Schematic editors are usually the first EDA specific tool a designer encounters in any project. During *schematic capture* developers extract the electrical circuit from their architecture diagram, retaining only basic part choice and connectivity information devoid of higher order structure, as in fig. 1.2a. Our participants considered it a “neccesary evil”, an unavoidable step stopping further progress until completed.

This focus on the lowest layer of abstraction is a product of the historical development of EDA tools. In the 1950s, most PCBs were drafted by hand, as in fig. 1.3a. By the ’60s electrical engineers were moving away from pen-and-paper drafting for electrical circuits and printed circuit boards, adopting a workflow that used partially transparent photomasks in the board lithography process. These photomasks were created in photoplotters that printed images onto trasparent sheets of plastic. These images, the board layout, were specified as a series of commands: draw a line of X thickness from coordinate Y to coordinate Z, expose an annulus centered at A with inner radius B and outer radius C, and so on. Added together, these commands allowed engineers to define the structure of circuit boards in a repeatable,



(a) Hand Drawn Board Layout



(b) Gerber Interface Ad

Figure 1.3: **Early techniques for board layout.** Before photoplotters and computer aided design tools most printed circuit boards (PCBs) were drawn by hand, as seen in subfigure (a) [55]. Photoplotters gave engineers a less error-prone, more repeatable way to create layouts, one amenable to computer control. Subfigure (b) shows an advertisment from the Gerber Scientific Company showcasing a photoplotter designed for PCB manufacturing along with an interface for computer control [56].

modifiable way, one amenable to computer automation and tooling.

Figure 1.3b shows an advertisement for a series of products that connect a computer to a photoplotter, something that was becoming more common by the 1970s. Modern board layout tools have their start in the software of this time, but circuit design, and schematic drafting, were largely still pen-and-paper tasks. Schematic capture tools were designed as an entry point to the digital design process, where one would take their hand drawn circuits and, in a largely data-entry task, convert them to a digital form for use in board layout. Modern schematic capture tools like Eagle, Altium, and KiCad fill the same role, allowing the design work that has happened elsewhere to be imported into the board layout process. They have incorporated new features, like copy-and-paste, but the fundamental paradigm has remained the same, shunning interaction with the conceptually richer system architecture phase of design.

Schematic capture tools, being contrained to a thin slice of the development process, have other problems. A major one is enabling reuse of previous designer work in new projects without introducing significant manual effort or necessitating error prone integration of old subsystems into new designs. When a net of electrical connections is copied into a project it often needs to be tweaked to ensure compatibility with a different set of sibling systems or design requirements. It is easy to miss a neccesary change or incorrectly integrate components because this process is largely manual, with the designer having to keep track of all relevant

concerns themselves. Modern tools can only provide limited assistance when most of the relevant context from the system architecture phase was never digitized in order to minimize data-entry effort. This means that parametric designs, which have internal logic to change the underlying circuit based on the environment in which they're used, can't meaningfully exist because there's insufficient knowledge about the environment. A better approach would need to capture this missing information without disrupting designers' workflow so more advanced features, like parametric design elements, can exist.

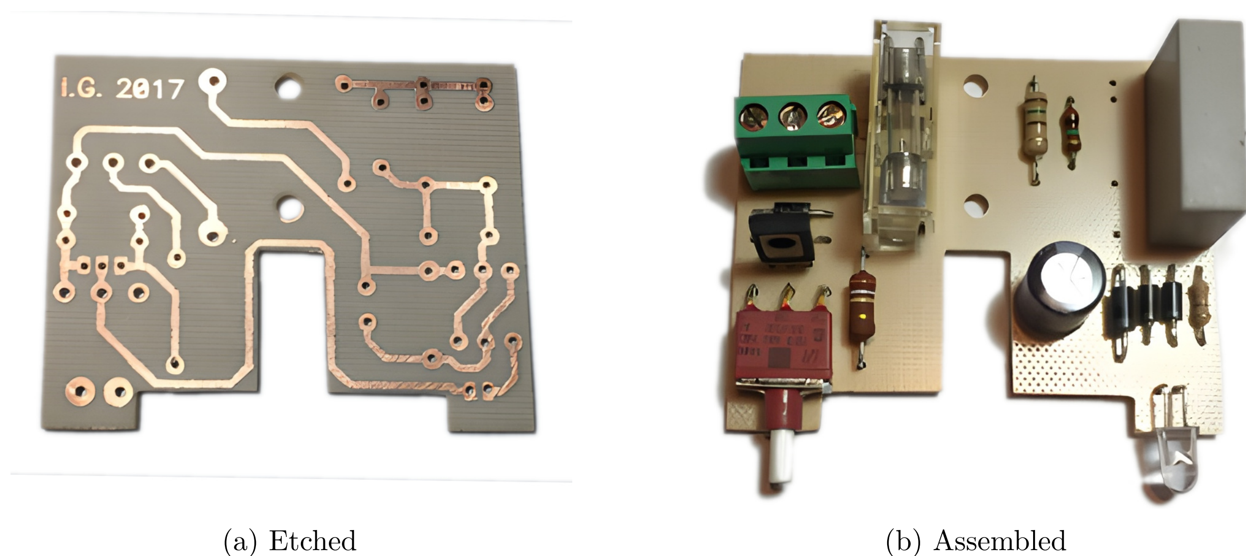
Figure 1.4: **An electrical rules checking (ERC) pin compatibility matrix.** Schematic capture tools try to provide error checking by assigning schematic pins to a small set of classes and using a matrix, like the one shown, to determine whether each connection is valid. Schematic editors lack the information needed for more substantive checks. This simplified model creates many false positives and false negatives.

One place where modern EDA tools do try to move beyond the pen-and-paper model is with their implementations of Electrical Rules Checking (ERC). ERC attempts to provide a series of automated checks that catch design mistakes early, before they cause problems in prototypes or finished devices. Sadly, a large proportion of our respondents simply turn ERC off or enable it only intermittently. Electrical rules checks, as they exist in major EDA suites, produce too many false positive and false negative alerts to keep active during the majority of schematic capture. Figure 1.4 shows why, ERC is usually defined by a matrix of

pin types that only check the compatibility of point-to-point connections based on a small list of possible classes. With only an electrical connectivity network it is hard to do more. The higher-order context that would be required to validate voltage compatibility, power consumption, bus protocols, and other properties spanning one or more wires isn't available to modern EDA tools.

In the current design paradigm, schematic capture is a thin data-entry layer only meant to get information into a format suitable for board layout and physical design. To make this feasible, it strips away the rich context of the system architecture phase, leaving designers to encode only the *much* smaller electrical connectivity of their design. This focus on the electrical net, while reasonable given its historical origins, leaves schematic capture software with too little information for more transformative changes to the embedded design process.

1.1.4 Board Layout



(a) Etched (b) Assembled

Figure 1.5: **Printed Circuit Boards (PCBs) before and after assembly** [17]. Processes for manufacturing PCBs use board layouts, like the one in fig. 1.2b, to specify where copper should remain on the final product. Traces on the PCB, visible in subfig. (a), are electrical connections that correspond to wires in the circuit diagram. When populated with components, as in subfig. (b), the device is complete.

During board layout engineers convert their schematics into designs for printed circuit boards (PCBs), which physically and electrically connect components in the final device. Engineers begin by importing a schematic into board layout tools. They assign each schematic symbol a footprint, the copper designs on the final PCB that components can be soldered to. Footprints are annotated with pads, the specific locations on the footprint that correspond to

pins so the layout tool will know how to map parts of the schematic to locations on the board. Each pad is an electrical interface. When the device is manufactured, it will be soldered to a component, connecting that component to the rest of the system. Because they have access to the schematic, board layout tools can keep track of which pads need to be connected. Engineers draw traces between pads, matching them to wires in the schematic. When every component and wire is represented by a footprint or trace, the layout is complete and can be used to manufacture a PCB. Figure 1.5 shows the end result, both after immediately manufacturing and once the PCB is populated with components.

Board layout is well suited for computerization. Both bookkeeping and error checking are computationally simple but tedious to perform manually. Ensuring that each part of the schematic has a corresponding element in the board layout amounts to checking items off a list. Error checking mostly focuses on the physical properties of the board; for example, ensuring traces don't intersect and pads have sufficient empty space surrounding them. These measurements are easy for a layout tools to perform when they are built to keep track of shapes and locations.

The effect of Computer Aided Design on board layout is immediately visible when comparing old boards with new. Old, manually-drafted designs, like the one in fig. 1.3a, tend to have traces with wide sweeping curves. This is because traces were drawn with tape, to ensure a consistent width, and sharp turns would make the tape bunch up, keeping the sheet from lying flat. The modern style, visible in figs. 1.2b and 1.5a is markedly different. Instead of smooth curves, traces are drawn with straight lines and a distinct preference for 45 degree angles, a byproduct of how early layout tools snapped traces to a grid, trading artistic freedom for computational simplicity.

We do not focus on board layout in this dissertation, taking the existing tools as given and targeting them directly. Instead of rewriting them for marginal gains, our work produces netlists, stripped schematics that layout tools can use as inputs. The relative robustness of existing software means that we can focus on underserved portions of the embedded design process.

1.2 Evaluating Embedded Development Tools

Evaluating our methodology requires judging how our tools affect the development process, something we do by looking at the different types of costs designers pay to design something. We can see the design process as one of sequential decision making; the user makes one choice, then another, then another, until they have a complete design. This framing gives us a way of looking at the costs inherent in the design process, three of which we use to evaluate design tools: time, attention, and momentum. Tools shape which decisions a user has to make and each decision takes some *time*, so a good tool allows a user to accomplish a task with fewer, quicker decisions. Likewise, users need information in order to make each decision. This information can be managed by either the user, spending their *attention*, or by their tools, minimizing their burden. Finally, users will want to make these decisions in some

order that feels natural to them. A bad tool won't accommodate this and forces the user to make decisions in some other order, sapping their *momentum*. These costs correspond directly to the three criteria we use to judge design tools, respectively: automation, salience, and ergonomics.

Automation compares the decisions users have to make when using a tool. Less automated tools forces more decision points, especially ones which are more complex or redundant. More automated tools minimize redundancy, complexity, and wasted work. Automation saves users time, which is usually directly valuable. An extreme example of automation is design synthesis, where all of the decisions needed to move from a specification to a design are made by the tool without human intervention. Less extreme cases include features like parameterized designs which allow the user to perform a task once and reuse that work elsewhere.

Salience assesses the information presented to the user. Tools with worse salience require the user to dig for necessary information. Tools with better salience are proactive, presenting the user with information that is immediately relevant without overwhelming their attention. Features which notify the user can make or break salience. For instance, a tool's error alerts and auto-complete suggestions can be timely and useful or they could be noise the user has to filter out.

Ergonomics looks at how flexible the tool is and whether it requires change in how users think. Less ergonomic tools force their users to make decisions in a certain order. They have a significant gap between the user's mental model and the tool's model of the problem. More ergonomic tools present abstractions in the ways that a user naturally reaches for, and are not tied to any one order of operations. A very ergonomic tool 'disappears' in the user's experience. It just does what the user wants it to; without needing them to spend time or effort on figuring out how to wrestle it into a useful shape.

1.3 Using Automated Reasoning

Our methodology relies on how automated reasoning tools can make inferences about mathematical models in a generic and flexible way, independent of the domain being modeled. Automated reasoning covers a variety of different algorithms from logic programming, theorem proving, and constraint solving. These algorithms work on sets of mathematical statements, examining them to produce inferences, new statements that are consistent with the input set. Depending on choice of logic, the axioms a tool understands. Different tools can be used to reason about different logics: boolean logic, linear algebra, predicate logic, and others. Different tools also produce different types of inferences; one might produce proofs that some statement is true, while another might rewrite statements into more elegant forms. This dissertation focuses on two such tools, satisfiability-modulo-theory solvers [5] (SMT solvers) and propagators, along with a logic of real numbers and boolean variables.

Our work builds on this framework by translating architecture diagrams into sets of statements within a chosen logic. Every property a design has, from the clock speed of a

microcontroller to the color of an LED, is translated into a variable in our model. Constraints within a block or relationships represented by links become statements relating those variables to each other. These systems of equations are models of the design that tools can reason over. Much of our methodology's power comes from how automated reasoning tools simply see a bag of statements. It does not matter if some statements are missing—for instance due to an abstract or incomplete design—the reasoning tools will produce inferences regardless.

SMT Solvers, used in EDG, try to solve systems of constraints, i.e. find assignments to every variable such that every constraint is satisfied. They do not support multi-step interactions; transactions are all or nothing. If you can encode a complex problem in a system of constraints, then an SMT solver can find a solution in a single step. As they search for a solution, SMT solvers will, internally, speculate. They will try making choices one way, and if that does not work, try another. EDG leans on this decision making capacity, using the solver for choices that would normally fall to the user.

Propagators, used in PolyBlocks, view a system of equations as a network and push values around as they're discovered. When a variable is given a fixed value, all the equations it appears in are updated and, where applicable, simplified. When equations are simplified enough to be solvable, they are and the answers substituted into other equations. So, when a new piece of information is added it can trigger waves of updates that propagate through the model. If there is a conflict, then the propagation engine can detect that and raise an error. This allows for easy automation to ensure a design stays consistent as the user makes changes, and inconsistencies caught just as they appear. While not as powerful as a SAT solver, propagators are sufficient for implementing many designer support features.

1.4 Summary

We seek to show that automated reasoning tools working with mathematical models of embedded systems can be used to build development tools that better support users. Embedded Design Generation (EDG) is a proof-of-concept meant to show that our methodology is powerful enough to cover the entire embedded design process and can make every decision needed to go from specification to final design. Polymorphic Blocks (PolyBlocks) shows that the techniques from EDG can, with some modification, be used in a practical embedded development tool suitable for a general audience of embedded developers. By extending EDG's electronics model, PolyBlocks shows that its core approach of representing embedded systems as constraints can work on the types of problems designers encounter in the real world. PolyBlocks also shows how our methodology can be made tractable enough for an interactive development tool by replacing EDG's powerful but slow backend with a simpler form of automated reasoning that is significantly faster. Together EDG and PolyBlocks serve as evidence for our thesis and as a foundation for future design tools.

1.5 Statement of Prior Publication

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

- Turning Coders into Makers: The Promise of Embedded Design Generation [46] (SCF '17), co-authored with Richard Lin, Antonio Iannopollo, Alberto Sangiovanni Vincentelli, Prabal Dutta, and Björn Hartmann
- Polymorphic Blocks: Unifying High-level Specification and Low-level Control for Circuit Board Design [34] (UIST '20), primarily authored by Richard Lin with other co-authors Connie Chi, Nikhil Jain, Ryan Nuqui, Prabal Dutta, and Björn Hartmann. Also used in Richard Lin's PhD thesis [33].

None of this work would have been possible without the effort of all the co-authors, including my advisor Prabal Dutta, graduate student colleagues Richard Lin and Antonio Iannopollo, and all the undergraduates who have worked with us over the years.

Chapter 2

Embedded Design Generation

2.1 Introduction

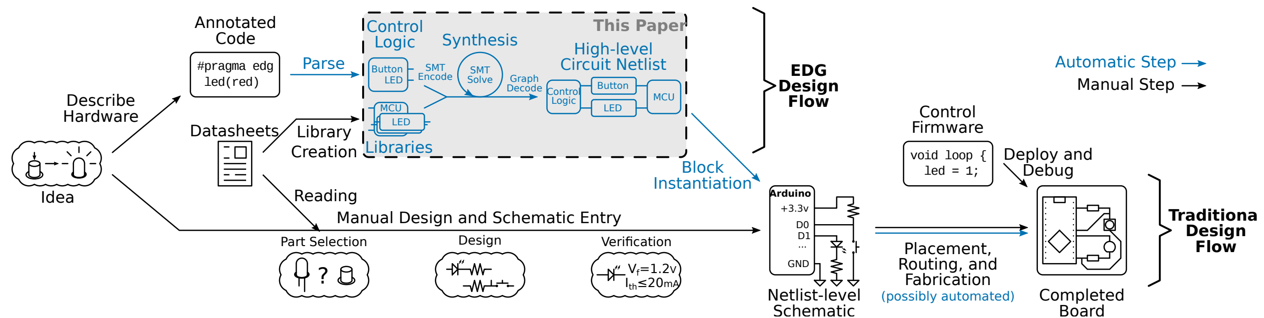


Figure 2.1: **The Embedded Design Generation (EDG) Methodology.** In contrast with traditional embedded development methods, which rely on significant user skill, EDG only requires a high-level specification to generate an electrically correct circuit that satisfies user requirements. Whole-cloth synthesis serves as a test of this dissertation's thesis that an approach centered around automated reasoning is capable of building powerful and usable embedded development tools. By completely automating the design process, with no human in the loop, we show that automated reasoning tools are capable of capturing and reasoning about the design process in all its complexity. As a proof-of-concept our prototype only implements those portions of the system that directly interact with the synthesis step. We both manually generate the specification needed, instead of extracting it from existing code, and manually create a circuit netlist from the output block diagram.

Embedded design is a process of sequential decision making that requires skill, knowledge, and judgment. At each step, the designer is weighing their specification against the current state of the project, gathered information, and the options in front of them. These judgment calls slowly add up until they have a finished schematic.

A good support tool has to understand and assist with these decisions in whole or in part. To know whether an error is salient the tool must know about the parts involved, have an understanding of how "complete" the design is, and be aware of other factors that feed into whether a user should be shown some error. An ergonomic work surface requires understanding how well defined the design is at any point in time. Knowing when to trigger automation requires differentiating between decisions requiring user input and those completely implied by other decisions.

Synthesis, the process of going from a specification to a finished design without user interaction, is an extreme test of understanding. A tool capable of synthesis must have a powerful enough model and powerful enough reasoning to handle the smaller choices. A model that can determine whether a full design works can also be used to detect smaller errors. A synthesis tool that can use minimal or partial designs as a specification can reason about designs no matter how the user approaches them. Synthesis is an example of the maximum possible automation, with most other automation operations existing within.

Synthesis is also a valuable feature in its own right, especially for novices lacking in background knowledge or skills. This is shown by the widespread interest in small-scale fabrication of embedded devices by non-professionals. We see a proliferation in other tools designed to make embedded development more approachable by using encapsulated hardware modules. Arduino, Raspberry Pi, Gumstix and others provide off-the-shelf modules that trade flexibility of design for ease of use. A synthesis tool would allow novices to subvert that trade-off by making it easier for them to develop systems optimized for their exact goals.

To that end, we propose a novel methodology, *Embedded Design Generation* (EDG), which exploits advances in constraint solvers to allow the automated generation of functionally correct-by-construction[1] board-level designs from a high level specification. EDG serves as a test of the applicability of automated reasoning tools to embedded design, as successful synthesis implies EDA tools can perform any of its sub-tasks with the same approach.

Tools based on EDG would only require that the user annotate their embedded software with simple requirements and, from that specification, synthesize the final circuit diagram, bill of materials, and firmware. Software APIs, electrical properties of circuits (e.g. Kirchhoff's current and voltage laws), and other low level details are combined with the user input into a system of constraints.

Existing constraint solvers can then generate designs which are functionally correct, electrically sound, and satisfy the user specification. To show that this both works and is computationally feasible, we build a prototype tool and test it with a variety of examples from different domains.

Figure 2.1 provides an overview of our proposed design flow and compares it to current embedded design practices. EDG abstracts away, through automation, much of the electronics expertise needed for tasks like parts selection, circuit design, and verification. In addition, figs. 2.2 to 2.6, 2.9 and 2.10 are all a part of a running example where we describe the

[1]We do not consider timing or other performance constraints in this paper as the focus is to empower designers who do not have to produce industrial strength boards.

construction of a simple device with a single LED and button.

2.2 Related Work

EDG builds on prior work in "electronic design automation" (EDA) by specializing the Platform-Based Design (PBD) methodology [48] for maker-scale embedded development. PBD is a methodology which has been successfully used to create synthesis tools in a number of domains, including integrated circuit (IC) development and automotive engineering. EDA community has incrementally raised the abstraction level of many embedded development tasks and by using insights from PBD and synthesis tools in other domains, we contribute to that progress.

2.2.1 General EDA

General-purpose board-level circuit design tools have largely not moved beyond graphical schematic capture, where users place electronic components and connect their pins together. In mainstream tools, hierarchical blocks allow some degree of abstraction by grouping low-level components together, but their lack of parameterization limits re-use. Additionally, while electronic design automation (EDA) tools feature electrical verification checks, these are of limited utility to makers. Matrix-based connection legality checks (for example, checking input-output directionality), though ubiquitous in design suites, are rarely used, non-extensible, catch only small classes of bugs, and have a high false-positive rate. Higher-end design suites often feature technologically advanced checks, like electromagnetic compatibility (EMC) or radio frequency interference (RFI), but these generally require significant skill to operate.

There has been some work towards building board design tools better suited for makers. For instance, PHDL [44] is a Verilog-like language for describing netlists that allows some parameterization of blocks and better designentry. However, like Verilog, it is only a static description of a circuit.

JITPCB [4] takes the concept further and embeds a hardware construction language in a general purpose programming language, allowing circuit generators instead of simple parameterized blocks. However, like PHDL, it does not have a model of the underlying design space, preventing it from catching many errors. JITPCB also does not reason over voltage, current, bandwidth, or other properties needed to perform useful verification of a design, something our tool does.

EDASolver [15] aims to be a synthesis tool for microcontroller based embedded systems. When given a tree that describes the basic structure of an embedded device, EDASolver can choose specific components to generate a circuit fitting that broad structure. Unlike JITPCB, it does have some understanding of the electrical properties of an embedded system, and can use that to choose valid components from a pool of parts. As EDASolver has neither published source code nor a technical paper, we are unable to fully characterize its capabilities

and limitations, but its modeling of electronics does not appear to be extensible beyond voltage and current limits.

While both JITPCB and EDASolver have some ability to choose specific components from vague specifications and automate the assignment of individual pins, these features are constrained by their inability to reason over the *topology* of a circuit. Our tool, and likely any tool that follows the EDG methodology, is capable of not only choosing components as needed but also adding elements to the topology of a circuit. This means our tool can create new power domains, insert amplifiers and buffers, and infer the need for IO expanders whenever required to create a valid design. Fundamentally, we reason over the space of possible designs and the requirements without the need to tightly constrain the topology of possible solutions. As a result, even our rudimentary tool can compensate for limitations in parts or complexities in a specification in much the same way that an engineer might.

2.2.2 PBD and Domain-Specific Tools

In Platform-Based Design's (PBD) terminology, our methodology maps user input to a set of library components according to well-defined composition rules that can be verified statically. PBD-based tools solve the synthesis problem by opportunistically *composing* elements from a library to generate systems of constraints which can be solved by external solvers. For instance, METRO II [12] allows for general model integration and architecture exploration, where the mapping process between specification and platform is validated through simulation. Likewise, PYCO [23] synthesizes a complete specification for a system from a partial set of Linear Temporal Logic (LTL) constraints and a library of components with LTL contracts. Although reminiscent of these techniques, the approach taken for EDG does not require the use of LTL or other logic languages.

Some techniques related to our approach have been also used in program synthesis. Brahma [26, 18] synthesizes loop-free programs over bitvectors out of a library of simpler functions. This allows Brahma to generate software from a sparse specification of boolean logic constraints. [20] propose the use of types in a program to synthesize valid expressions which are then suggested to the programmer.

Robotics-oriented design tools like EMLab [7] and ROSLab [37, 38] solve similar problems to EDG in that domain. EMLab is a block-level development tool for robotic electronics that uses an SMT based verification mechanism similar to our own, however it does not extend that to provide synthesis. ROSLab provides a similar pathway from code to circuitry, however unlike our tool, it is limited to custom-made hardware elements that support their custom chaining protocol. In contrast, EDG works with off-the-shelf electronics in order to reduce the cost of creating a library of parts and enable the fast integration of new components.

Finally, tools aiding interactive device design largely also follow the pattern of automatically figuring out details from a high-level design, albeit in more constrained domains. For instance Midas [49] automatically designs capacitive touch layouts given user-specified sensor type, shape, and position. Likewise, PaperPulse [45] adds interactive electronics elements to paper crafts from a library of widgets.

2.3 Methodology

The goal of Embedded Design Generation is to create better abstractions for developing embedded devices and tools that can perform robust verification of device designs. However, better verification requires our tool to reason about the relationship between hardware and software. Verifying the electrical properties of a thermometer does no good if that thermometer has no way to send its data to the designer's software. Our key insight is that many of the design's hardware requirements are reflected in the code, for example in required libraries and pin assignments. Yet the fundamental logic of the device, how it functions at runtime, is rarely reflected in the hardware. This asymmetry suggests that higher level abstractions for embedded development should be similar to embedded code.

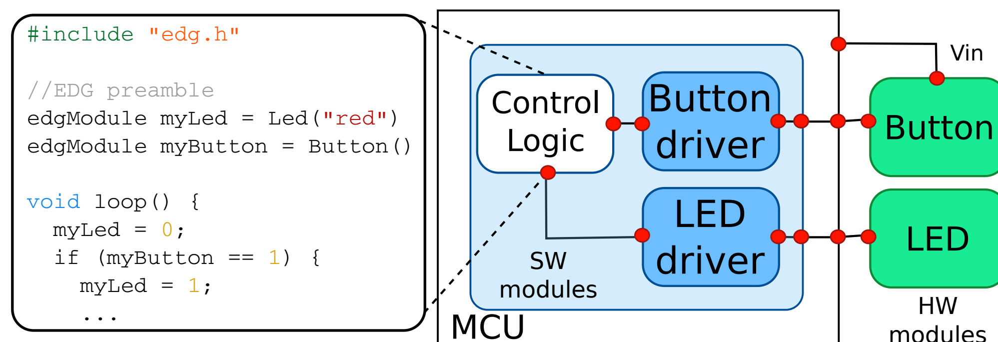


Figure 2.2: **Code can be a specification for a device.** The code to the left, the control logic, specifies a device in which a light blinks when a button is pressed. It is also the software that is eventually run on that device, the annotations in the `EDGPreamble` specify the hardware infrastructure needed to make the software function as intended. The block diagram to the right describes one possible device that matches that specification, by meeting all the hardware requirements in and being able to run the control logic. Due to our focus on the synthesis process, the code shown here is a mockup that shows one possible structure for a specification.

If we want to be able to describe the device at a higher level, we can capture the most important parts of its function and construction in its code. We can specify how the device acts at runtime, as well as the hardware infrastructure needed for the device to function. Figure 2.2 shows a stylized example of this, where many of the implicit hardware requirements that are expressed in user code are rendered as explicit declarations for a design generation tool to use. The software in fig. 2.2 is a specification for the hardware *and* the runtime operation of the device.

To make this kind of high-level abstraction useful, we must be able to compile it into the firmware and circuitry needed to construct an embedded device. However, firmware and circuit diagrams are too low-level for efficient synthesis. Instead we represent the result as a *block diagram*, like the one in fig. 2.2, which can be easily turned into a final design. Likewise, we need to be able to tell if those block diagrams actually describe correct devices, so that

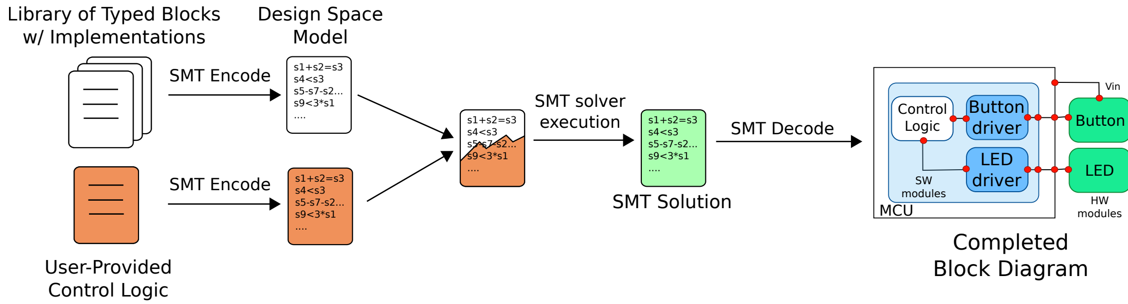


Figure 2.3: **Design Generation at a High Level.** EDG tools use existing constraint solvers to perform synthesis. The tools convert knowledge about the design space and control logic specification into constraint satisfaction problems whose solutions are block diagrams describing valid device designs. These block diagrams completely specify the design of an embedded device and can be easily converted to more useful formats.

we do not generate broken or invalid designs. A *type system* gives us a way to construct the blocks for real-world parts and an algorithm to decide whether any given block diagram is correct. Finally, we need to choose a single valid device from the space of possible devices. We do this by constructing a *design space model*, which captures a wide range of possible designs in ways that existing constraint solvers can reason about.

The Embedded Design Generation methodology is built around these three core concepts:

Block Diagrams capture the conceptual structure of a device across both hardware and software boundaries, by taking elements of the final design and representing them as blocks with connectivity information. These diagrams are an intuitive yet powerful model for working with systems, and can capture device structure, resources, and many of the other relationships found between elements of a design.

The Type System defines rules for how we transcribe the real world properties of circuits and software into blocks and their *type signatures*. It also gives us *type checking*, a process that determines whether any block diagram describes a valid device.

The Design Space Model is a system of constraints suitable for general-purpose solvers, built from a library of blocks and their type signatures. This model can then be used to generate a complete, working, block diagram for a device from a specification.

Figure 2.3 shows our methodology for design generation, which exploits the growing speed and increasing expressive power of constraint satisfaction problem (CSP) solvers. We convert a library of blocks, with corresponding type signatures, into a monolithic set of constraints that model the space of potential designs made up of those blocks. These constraints are then composed with constraints derived from the control logic to produce a CSP whose space of valid solutions is the space of valid block diagrams that meet our specification. We then pass this CSP to the solver and decode the result into a block diagram that will successfully typecheck.

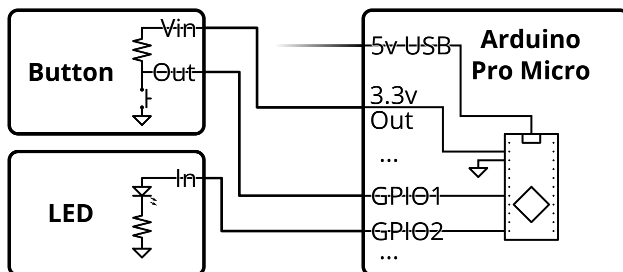


Figure 2.4: **Convert a block diagram into circuitry by linking implementations together.** Instantiating the block diagram from fig. 2.2 requires taking implementation details associated with each block, in this case a relevant sub-circuit, and connecting them together based on the links between their blocks.

```
#include "edg.h"


//EDG Preamble
edgModule myLed = Led("red")
edgModule myButton = Button()

void loop() {
  myLed = 1;
    ...
}
```

(a) Original Control Logic

```
#include "led.h"
#include "button.h"

//EDG Preamble
var myLed = initLED(GPIO2);
var myButton = initButton(GPIO1);

void loop() {
  myLed = 1;
    ...
}
```

(b) Instantiated Firmware

Figure 2.5: **Convert a block diagram to firmware by filling in templates.** Instantiating software is a simple template replacement operation. Figure 2.5a is a mockup of user-provided control logic for the device. Figure 2.5b is the code after we replace the EDG-provided template elements with the concrete implementations provided by other blocks. Note that the code outside of these templates is unmodified.

Block diagrams are ideal representations because they are easy to convert into the design files needed to fabricate a device. Figure 2.4 shows how the final circuit can be created by connecting individual block implementations along the links between them. Similarly, fig. 2.5 shows how the firmware can be instantiated with template replacement operations that pull from code snippets provided by connected blocks.

Block diagrams also work at many levels of fidelity. In general, blocks can be composed of smaller blocks until one recurses down to single instructions or individual circuit elements. Our current tool works with relatively large blocks made up of entire libraries or breakout boards. This allows us to abstract away questions of timing delay, electromagnetic interference, and many other phenomena that become evident at smaller scales. Large blocks also mean there is a smaller space of possible configurations for solvers to reason over, making their

immediate use more feasible. As solvers grow faster and more expressive, EDG tools can move to using finer granularity models with smaller blocks.

We structure each block diagram around the notions of blocks, ports, and links. As we have seen, *blocks* represent realizable elements of our final design and each has a number of *ports* which represent specific capabilities, relationships, or resources a block may have. *Links* are then the connections between ports that represent the transfer of resources, usage of capabilities, or other relationships between blocks. For instance, the connection of a serial line or the use of a software API.

A block diagram must have all the information needed to instantiate a device but many parts have properties and settings that are not solely defined by their connections. Consider the LED in fig. 2.4, which could be annotated with information about its color. To allow the block diagram to represent this information, blocks, ports, and links all have *concrete types*, which are structures made up of named primitives—like integers, boolean values, and strings—or nested substructures. Concrete types allow each block to specify the information needed to instantiate it as well as additional properties useful in other phases of design generation.

The block diagram alone is not enough for synthesis, since we require a way to determine whether any given block diagram will result in a valid device. The *type system* gives us a way to generate blocks and their *type signatures*, constraints over those blocks, so that we can check the correctness of an entire block diagram. As in fig. 2.6, type signatures annotate blocks, ports and links with constraints that limit the concrete types they may have within in a block diagram. Then, type checking ensures that each element of the design satisfies its type signature. In section 2.4.2 we explore how we constructed a type system for our prototype tool that accurately detects and rejects invalid designs with this procedure.

Finally, Embedded Design Generation requires that we are able to turn a library of blocks and type signatures into a *design space model* that our tools can reason over. In practice, we expect this to take the form of a monolithic constraint satisfaction problem to which we can add the specifications, usually in the form of a control logic block, for any particular synthesis task. This single model can then be optimized or added to, as new parts become available or new limitations in the design space are found. We build the Design Space Model by generating a CSP for each block that might be included in an output design, and then adding variables that determine whether any pair of ports is connected. The solver can then choose which connections exist and give us the final block diagram, with valid concrete types for each block, link, and port. While optimizations can be layered over this, we believe that any design space model will have this core structure.

2.4 EDG Prototype Architecture

Our prototype tool implements the EDG methodology described in the previous section, with a focus on synthesizing devices from relatively large blocks at a level high above individual

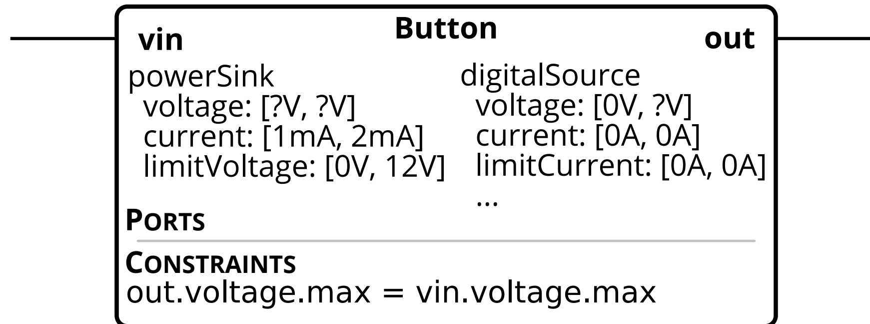


Figure 2.6: **Type signatures are constraints on their elements.** While elements of a design are given concrete types, the blocks on their own are usable in a variety of settings. Type signatures are simply the conditions under which a block will work as intended, presented as constraints over the concrete type of a block. A type system is the set of rules for how to map real-world properties into types and type signatures, such that a block diagram which typechecks can be instantiated into a working device. In this case we constrain the expected input and output voltages of a button to be equal, a limitation on the possible concrete types that button may have.

resistors and ICs. We implemented our tool in Haskell and used Z3 [13] as the underlying constraint solver.[2]

2.4.1 Blocks, Links, and Ports

As in the general methodology, our prototype uses blocks, links, and ports to represent possible designs for embedded devices. Figures 2.7 and 2.8 illustrate the principal data structures used in our tool. The user provides their input in the form of control logic which we manually encode as a block that must appear in the final design.

2.4.2 Type Signatures

Each block, link, and port in our library contains a type signature, i.e. a set of bounds on the concrete type an element may have in a valid block diagram. In a block diagram concrete types are data structures composed of named fields, each linked to a value. These values can be boolean, integers, floats, strings, UIDs or another nested set of field-value pairs.

Our tool's internal representation for type signatures is shown in figs. 2.7 and 2.8. These structures capture all the pieces of information needed to generate the SMT representation of a design element, with the majority of the constraints simply being stored as expressions that translate directly into SMT constraints. As in fig. 2.6 each constraint provides a way

[2]Our code and the results of our experiments are available at https://lab11.github.io/edg-sat-prototype/appendix/scf2017.

```
PORT p:
  used :: Bool:              Indicates whether the port is used in the final design
  connected :: Bool:         Indicates whether the port is connected to another port
  class :: String:           Identifier used to constrain which ports can connect to each other
  type :: [fields]:          List of fields to be translated to SMT variables
  constraints::[expr]:       List of formulas over the type of the port.
                             Must be true for system to typecheck
IMPLICIT CONSTRAINTS:
  connected => used:
   Ensures that a port is part of the design if connected to any other port
  forall c in constraints, used => c:
   Only requires the solver to satisfy the constraints if the port is being used
```

Figure 2.7: **Ports use implicit constraints to capture connectivity.** The implicit constraints in each port allow us to relax the constraints on the SMT solver. The first constraint says that if the port is connected to another then the port must be used in the output block diagram. Along with the corresponding constraints from fig. 2.8, this ensures that every element has a flag to show whether it is used in the final design. The second implicit constraint tells the SMT solver that none of the type signature's constraints need to be satisfied if the element is not used , minimizing its work.

```
BLOCK/LINK b:
  UID :: String:             Unique identifier for the block
  used :: Bool:              Indicates whether the block is in the final design
  ports :: [port]:           List of ports attached to this block
  type :: [fields]:          List of fields to be translated as SMT variables
  constraints::[expr]:       List of formulas over the type of the block and its ports.
                             Must be true for system to typecheck
IMPLICIT CONSTRAINTS:
  forall p in Ports, p.used => used:
   Ensures that a block is part of the design if any of its ports are part of the design.
  forall c in constraints, used => c:
   Only requires the solver to satisfy the constraints if the block is being used
```

Figure 2.8: **Blocks and links have identical representations in our tool.** Despite their stylistic differences, both blocks and links capture relationships between ports, along with some internal data and constraints. This allows us to turn connections between modules via links into one-to-one relationships between ports on modules and ports on links, simplifying the process of constructing an SMT problem.

to express the ambiguity in a type signature, since each block has many valid concrete types and can therefore work in a variety of different designs. The constraints are arbitrary expressions consisting of boolean expressions, ordering operators, linear arithmetic operators, and references to values found in the concrete type of that element.

In other cases the constraints can be used to specify that a value falls in some range, that there exists an equality which must be preserved, or any other condition that is representable as an expression in our solver. These expressions can capture many complex behaviors, like the assignment of pins to functions on a microcontroller, ranges of voltages and currents, and even the nesting of interfaces where our tool has to infer additional parts.

We choose to limit constraints on numerical values to linear arithmetic because non-linear relationships that cannot be conservatively approximated by linear ones are relatively rare given the fidelity of our tool. Since Z3 and other SMT solvers are much slower when working with non-linear constraints, we choose to limit ourselves to the faster option.

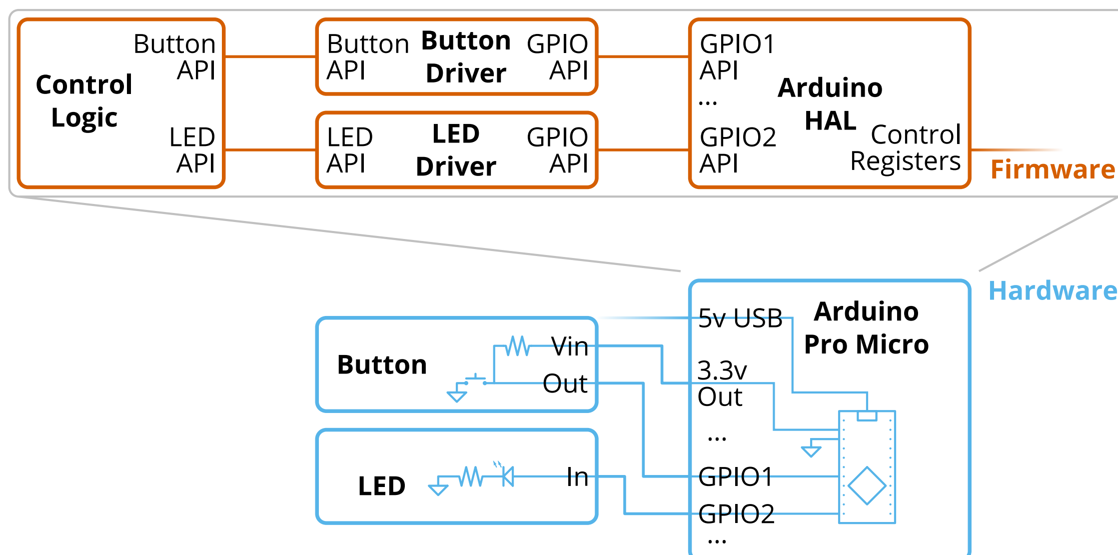


Figure 2.9: **There are symmetries between the hardware and firmware elements of a design.** Many of the hardware elements of a design are paired with corresponding firmware elements, as in the case of an LED and its driver code. These pairs tend to have identical structure in both domains, even if there are other components that only exist in one domain.

2.4.3 Design Space Model

Our prototype naively constructs a design space model from its library of blocks and links. We rely on the fact that constraints in type signatures are almost identical to the equivalent SMT expression.

All the type signature fields in figs. 2.7 and 2.8 are transformed into sets of constraints. Each flag in the element and value in the type becomes a variable the SMT solver is capable of assigning. Then we add constraints between those variables to match those in the type signatures. From this state, we generate a large adjacency matrix where each cell is an unassigned boolean value that determines whether a particular pair of ports is linked. If ports are linked, their types are set equal and they are marked as being connected. This lets us simulate a one-to-one connection between ports on a block and ports on a link. We then extract this adjacency matrix from the SMT solver's solution and use it to construct the block diagram by walking the resulting graph and recovering each block's concrete type.

2.4.4 SMT encoding, solving, and decoding

Working from the control logic and the design space model, our tool encodes the complete synthesis problem as a system of boolean and linear arithmetic constraints which are then solved by an SMT solver. Blocks, links and ports are all translated to equivalent SMT constraints as described in the previous subsection. The control logic is treated like any other block and added to the CSP, though with the additional requirement that it be used in the final design.

Once the encoding is complete, our tool generates an SMT-LIB v2.0 compatible file which is then passed to the SMT solver. If the solver is able to find a solution to the system of constraints, it is decoded into data structures where all the type signatures have been resolved to concrete types. Finally, the resulting network of blocks is presented to the user as a block diagram describing a device that can run the control logic.

2.5 Type System Design

As connection legality essentially drives circuit synthesis, the properties and constraints captured in the type signatures are especially important. In our prototype, we attempt to model the parameters needed to ensure that the circuit is electrically valid, programatically valid, and meets user requirements.

2.5.1 Software and Hardware Modeling

As a complete embedded design tool, our prototype must model both the hardware (circuits) and software (user code and drivers). While it is common to think of them as completely separate domains, as in fig. 2.9, they are usually heavily intertwined in practice.

Most peripherals ultimately expose a firmware API and most electrical components are controlled to some degree by the firmware. As in fig. 2.10, our representation combines the hardware and firmware domains in ports and blocks when appropriate.

Compared to separate representations, this reduces the number of ports and blocks that the solver needs to search through, improving performance. This combined model accurately

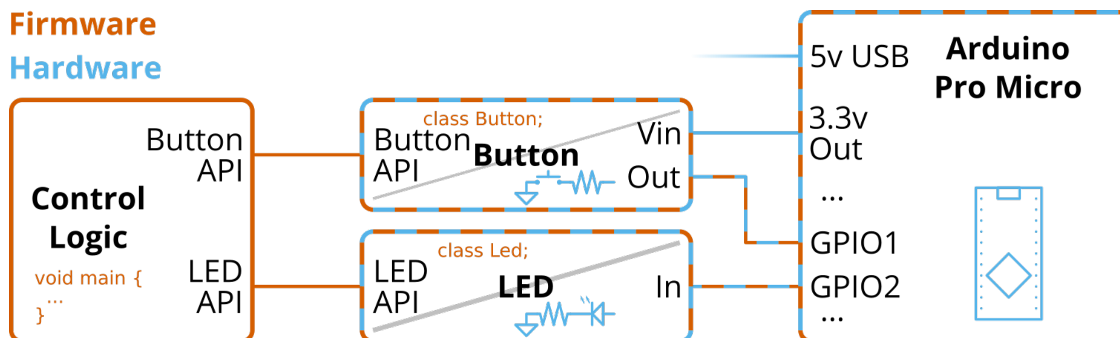


Figure 2.10: **Our representation with an integrated view of hardware and firmware.** This combines the circuit design and firmware drivers of each peripheral from fig. 2.9 into a single block. In this case the firmware and hardware blocks for the Arduino, button, and LED are combined into a single block representing each component. Note that pure hardware and firmware elements still do exist.

represents how many APIs control electrical connections and drivers are usually associated with a device, without additional complicated constraints to tie domains together.

2.5.2 Ports and Links

As synthesis is interface driven, components are almost completely defined by their ports. Our type system models common electrical ports, including several digital communications networks, as well as arbitrary firmware APIs.

2.5.2.1 Firmware Ports

Firmware ports define pure firmware interfaces, APIs. They are modeled as either producers or consumers with a type, like LEDs or temperature sensors, and optional data, like sensor resolution. Ports on the control logic are the starting point for generating a design. Additional constraints prevent hardware referenced by one piece of code from being split between different controllers.

2.5.2.2 Electrical Ports

Electrical ports define a pure electrical interface, which does not interact with the firmware domain. Our type system only has power ports, which define either an always-on voltage source or device power input.

Power ports capture voltage levels and current flows through a port. Both are modeled as ranges to capture device tolerances, as in the output of a wall wart, and runtime variation, like when an LED is on or off. These represent the full spectrum of expected circuit states during operation.

We also specify voltage and current limits as ranges, where the expected operating range must be contained within the tolerable range. While upper limits are useful for absolute maximum ratings, ranges capture lower limits, like minimum operating voltage or minimum current draws. Despite being a highly conservative model, this encodes the most important information needed for power compatibility checking.

Our current type system gives all components a common ground, so that power ports are single-ended voltage sources referenced to an implicit universal ground. This limitation is mostly for simplicity, but captures most beginner and intermediate designs. Advanced features like isolation domains require additions to our type system.

2.5.2.3 Controlled Ports

Controlled ports define an electrical port controlled with a firmware API. The simplest example is the microcontroller-driven GPIO, which is described as a digital signal.

Digital ports have all the properties of power ports including the ability to supply power. This models the common usage of microcontroller GPIOs to switch small loads, like LEDs, while generalizing to any controlled load. We also capture voltage thresholds that check both signal level compatibility and thresholds on switched loads.

2.5.2.4 Digital Communications Ports

Digital communications ports are a variant of the bidirectional digital port for common communications protocols. Many digital communications protocols require multiple wires, which we bundle as a single port. This is for efficiency reasons: all the wires travel together, and modeling each pin as a separate port creates extra connections that increase the search space and hurt synthesis performance. Our type system models ports for several communications buses including I^2C, SPI, and UART. Each bus checks for signal level compatibility as well as bus-specific properties like I^2C address uniqueness.

2.5.3 Components

2.5.3.1 Peripherals

Most components representing peripheral devices are structured as adapters that provide one interface and require another in order to function correctly.

One such example is the controlled LED, whose hardware is just the standard LED circuit with a ballasting resistor. We model this as a two-port element: an LED API producer port, and a GPIO consumer port. The GPIO port also models important electrical characteristics like current and voltage limits.

As an adapter-style component, ports on both sides are required to be connected. While a LED without an electrical input is useless, the requirement for an API port prevents synthesis from placing extraneous, unrequested LEDs. Most other peripheral components, like buttons, temperature sensors, or LCD displays, are similar.

True adapter components also exist. The GPIO expander requires a I2C slave connection and provides extra GPIOs. Likewise, a digital amplifier requires a power supply and low-power digital output and produces a new digital output at power supply voltages.

2.5.3.2 Firmware

Our type system also models pure firmware blocks in the same way. For example, a FAT32 library provides a file system API and consumes a low-level nonvolatile memory API.

2.5.3.3 Microcontrollers

Microcontrollers are structured differently because they serve as the control source for devices they provide interfaces but do not have requirements aside from power. Otherwise, they are modeled like every other component and are largely defined by their ports.

2.6 Evaluation

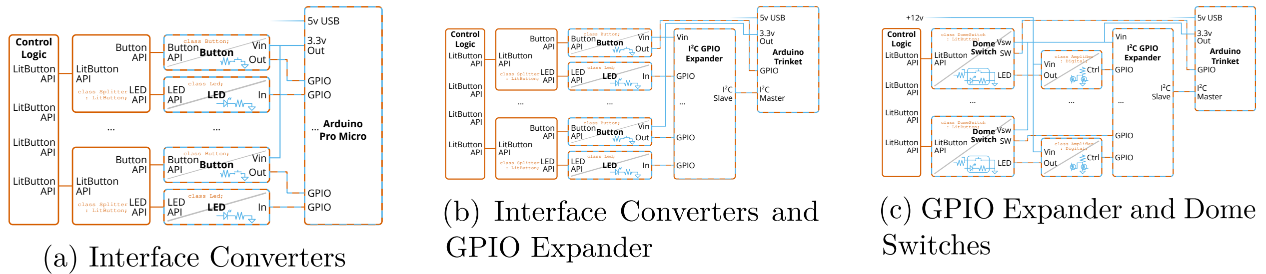
We create a number of embedded devices by manually generating the control logic block's type signature, using our prototype tool to synthesize a design, and manually instantiating each design to test its correctness. Each device was synthesized with three separate libraries of varying size.

Our full library is used for all examples, except where noted, and consists of these components:

- *Microcontrollers*: Arduino Pro Micro 3.3,V, Arduino Trinket 3.3,V
- *Basic peripherals*: tactile switch with pull-up, LED with resist-or, 12V lit dome switch with pull-up, 12V fan
- *Device peripherals*: Sparkfun 16x2 serial LCDs (3.3,V and 5,V versions), SD card with SPI interface, Sparkfun OpenLog
- *Sensors*: TMP102 I^2C temperature sensor, QRE1113 reflectance sensor with output resistor
- *Interfaces*: I^2C GPIO expander, high-side digital amplifier, TB6612-FNG dual motor driver, L7805 voltage regulator
- *Software*: FAT32 filesystem driver

The library contains a total of 73 blocks and links. This is a highly constrained set that is likely not very representative of the libraries any production system would use. However, it should suffice to gain a broad idea of the performance characteristics of our tool and accurately capture how our tool responds to limitations in the library of available blocks.

We first examine a number of simple test cases. Then we look at how our tool responds to restrictions on available parts, both in terms of small changes and instances where the device is radically changed. Finally, we look at the performance of our tool as it synthesized all the designs described.



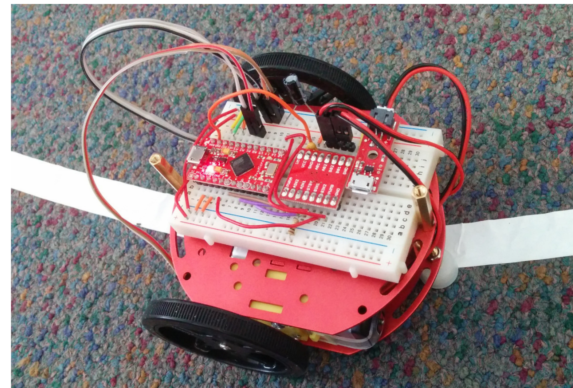
(a) Interface Converters

(b) Interface Converters and GPIO Expander

(c) GPIO Expander and Dome Switches

Figure 2.11: **Three Versions of Simon.** These three designs are generated from identical control logic blocks. Each design could be generated from code similar to the EDG preamble found in fig. 2.2, where each required module is turned into a port connecting to that peripheral. Despite their radically different construction these designs are functionally identical, differing in only the size of the buttons and minor timing variations.

(a) Simon Matching Game



(b) Line-Following Robot

Figure 2.12: **Physical realization of Simon and line-following robot.** Figure 2.12a shows the three Simon variants on a large breadboard. The left section is the initial synthesis result (2.11a), using a microcontroller and discrete LEDs and switches. The center section is the second result (2.11b), where the system is forced to use a pin-constrained microcontroller and must infer a GPIO expander. The right section is the final result (2.11c), using inferred digital amplifiers to drive external dome switches. Figure 2.12b shows the line-following robot, which successfully generates a circuit that correctly integrates the pre-selected chassis and motors.

2.6.1 Basic Synthesis Tasks

We synthesize a number of simple devices to show the range of domains EDG may be useful for and to analyze the results.

2.6.1.1 Blinking LED

We synthesize the simple light and button combination that we have been using as a running example. This device has a single LED and a single button that blinks the LED when pressed. The blinking light has long been the “Hello, World” of embedded hardware design and seemed a fitting place to start.

We use this design as an initial test case as we developed our tool and type system because it captures many of the most common constraints in embedded development. Any synthesized design has to keep a coherent chain of control from the control logic to each peripheral, a chain that captures software relationships, hardware relationships, and relationships that jump between those domains. This example also captures basic power management as the tool has to ensure that voltage levels are correct and that current limits are met. Our tools have synthesized many working versions of this device, and those that we constructed functioned correctly.

2.6.1.2 Temperature Controller

This device is a basic control system where a thermometer reads the local temperature, displays it on a small LCD screen, and runs a fan when it is too hot. Here we capture bus topologies like I^2C, as well as a more complex power system that could supply the 12,V fan, 3.3,V microcontroller, and 3.3,V sensor. Our tools are able to synthesize this device while avoiding pitfalls like mismatched power sources.

2.6.1.3 Line Following Robot

Our final basic design is a line-following robot designed to stand on a specific chassis with pre-mounted motors. Here we verify that we synthesize a device from a more complete partial design. In this case, we knew both the code we wished to run and the motors we wanted to use, so we specified that all three components must be included in the design. Our tool managed to correctly connect the motors to the microcontroller, including adding motor drivers and the split-level power system needed to use them.

Our tool does not distinguish between being asked to design a device with just the control logic as a specification, three separate blocks that must all be in the design, or some manually-designed critical portion of a device that needs non-critical surrounding infrastructure. This versatility means that in addition to the design process we focus on in this paper, EDG-based tools can support many other forms of interaction.

2.6.2 Inferring Missing Design Elements

One of the most useful features of the EDG methodology is that it can infer additions to the topology of a device when the available set of parts is limited.

To test this we design a simple datalogger that reads a temperature sensor and writes the result to an SD card. Our first synthesis of this device produced a design that used the OpenLog breakout board, a combination of SD card socket and preprogrammed chip with a simple serial interface for SD card filesystem access

Then we run the synthesis process again after removing the OpenLog from the library of available parts. This time, our tool adds an SD card holder to the device and used a software FAT32 driver to provide a filesystem, showing that our tool can adapt to accommodate constraints in the available pool of parts or non-obvious interactions between interfaces.

2.6.3 Preservation of Function

The final design we synthesize is our own version of the Simon electronic game. This device flashes four lights in a random order and asks the player to press the corresponding buttons in that same order. Our library supports two major options for synthesizing this design: large buttons with built in LEDs or smaller discrete LEDs and buttons in matching pairs.

Our initial attempt to synthesize this design resulted in a mix of these two options, likely not what a designer would want. We had to add a constraint to ensure that all the buttons used by device had a similar type. The resulting design, shown in fig. 2.11a, consists of four pairs of similar buttons and LEDs and a microcontroller with sufficient IO pins to directly connect all peripherals.

After removing the large microcontroller from the library and leaving only a pin-limited microcontroller, our tool created the design in fig. 2.11b. This design adds a GPIO expander to provide enough pins to control all the peripherals.

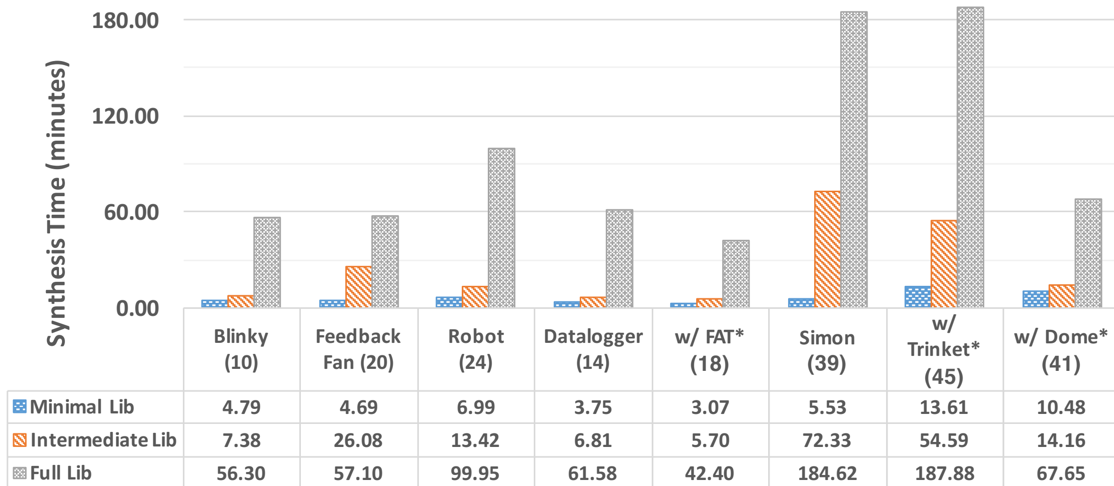
Our final change is removing the driver that allowed us to use a discrete LED and button pair as a single lit button. Figure 2.11c shows the result, and we note that this device shares no parts in common with our original version of Simon—instead accomplishing the same task with a completely different implementation.

We constructed all three versions (see fig. 2.12a) and they functioned identically barring the difference in parts and some drift in the timing. Our synthesis process preserves the key details of our control logic, no matter the components used.

2.6.4 Performance

We provide synthesis runtime data for our designs to show both the feasibility of our methodology and the scaling behavior.

Experiments were run on a server with dual-socket Intel E5-2667 CPUs (3.3 GHz, 8 physical cores per socket) and 192 GB of RAM. Note that the computational time was dominated by Z3 which is single-threaded. All experiments used under 2 GB of RAM.



	Blinky (10)	Feedback Fan (20)	Robot (24)	Datalogger (14)	w/ FAT* (18)	Simon (39)	w/ Trinket* (45)	w/ Dome* (41)
Minimal Lib	4.79	4.69	6.99	3.75	3.07	5.53	13.61	10.48
Intermediate Lib	7.38	26.08	13.42	6.81	5.70	72.33	54.59	14.16
Full Lib	56.30	57.10	99.95	61.58	42.40	184.62	187.88	67.65

Figure 2.13: **EDG synthesis time for the discussed examples.** Each minimal library contains only the blocks necessary to synthesize the design. The full libraries include all 73 of the encoded blocks, except for Datalogger FAT*, Simon Trinket*, and Simon Dome* for which the full library was constrained to make simpler designs impossible. Each intermediate library is approximately half the size of the full library. Times are expressed in minutes and the parentheses next to each name contain the number of blocks and links in each design.

Figure 2.13 shows the synthesis time for all the experiments. Every design was synthesized using three libraries with different sizes. As expected, our tool performance depends both on the size of the library and the complexity of the solution, represented by the number of blocks used for each design. Larger libraries or solutions usually resulted in longer runtimes. The biggest designs required several tens of components and a runtime up to three hours which is reasonable compared to the time required to perform the same tasks manually. However, we believe that our tool's performance can be drastically improved, as we discuss in the next section.

2.7 Discussion

We believe that future work will further support our hypothesis that the EDG methodology is a powerful and feasible way to improve embedded development tools.

2.7.1 Performance and Optimization

Ultimately, the tool we implemented is just a prototype to demonstrate that the EDG methodology is fundamentally feasible. While runtimes for even our limited library of components are not as fast as we would like, there are many possible optimizations.

Performance of existing solvers has improved over the years through advancements in the basic boolean SAT techniques [22], and SMT solver theories [10]. Recent years have also shown major improvements in the expressive power of constraint solvers, including techniques like counterexample-guided inductive synthesis [27] which allow solvers to incorporate new domains of reasoning through a feedback process. Communities in these fields are active and we do not believe this trend will stop anytime soon.

Additionally, we believe we can greatly improve our tool's performance by exploiting a more efficient SMT encoding of the design space model. Currently the design space model is naively translated to an SMT equivalent, without leveraging symmetries, pre-computed solutions, or user insight. in addition to the type system we introduced here. Among other solutions, we plan to do this by including the ability to reason over circuit equations, satisfy temporal requirements, and support distributed computation.–>

Even if, ultimately, full synthesis against a large library is infeasible we believe the EDG methodology still holds promise. The integrated representation of electronic components with firmware drivers eliminates the often-manual step of mapping pins to firmware, while the rich type system allows automated electrical verification to a greater degree than existing matrix-based connectivity checks. Synthesis at a smaller scale is still important, whether for ensuring the thoroughness of the type system, or for automating design within a constrained environment.

2.7.2 Type System Fidelity

The type system determines connection legality and its thoroughness determines the correctness of EDG's output. While the model for our prototype is largely based on our experiences as embedded designers, a more formal treatment of which properties are relevant is desirable. In particular we would like to develop a formal composable ontology for elements of an embedded system that integrates well with EDG.

Additionally, information from datasheets is insufficient for synthesis, occasionally even contradictory. Strict compatibility checks using datasheet-provided specifications often produces false positive errors. For instance, logic voltage thresholds are usually given as a single value under arbitrary test conditions. Using that value directly would make many reasonable designs unsynthesizable, as the specification conditions are excessive for the digital signaling methods our tool models. Instead, we use less conservative bounds that are accurate given the low frequency conditions our model assumes. However, higher accuracy models would be possible with more precise datasheets, especially if specifications were given as simple mathematical functions.

Our limited library also obviates the need to encode the physical details of components. For example, both a weak indicator LED and a lighting-grade power LED would satisfy a user requirement for a LED. Additional constraints like brightness, power draw, or form factor are necessary to fully capture user intent.

2.7.3 Usability

EDG proposes an input very different from the traditional electronics and embedded design flow. While it requires less electronics expertise to build a functional device, a larger question is where best to draw the line between automated processes and user input. Alternatively, hybrid interactive approaches may be desirable. For example, an EDG based assistive tool might ask a designer, "I see your parts are not voltage-level compatible, would you like me to insert a regulator?" Future user studies can illuminate the trade-offs between these different strategies in the electronics design domain.

Embedded hardware development also does not end with board fabrication, and debugging poses significant challenges [39]. EDG's richer data model, containing information like peripheral topology and expected voltages, can support novel assistive debugging strategies. Approaches include automatically generated self-test routines for peripherals or interactive guided debugging.

Finally, a comprehensive, complete set of libraries ultimately forms the basis for EDG. Building such libraries is far from painless, currently involving the manual, time-consuming translation of datasheet specifications to part constraints. Better languages for encoding part data could increase accessibility, while formalisms (like better type systems) can reduce the likelihood of mistakes.

2.8 Conclusion

EDG is a proof of concept tool that automates much of the embedded design process. It is fundamentally different from the standard EDA pipeline, replacing the system architecture and schematic capture phases of development with a hands-free, fully-automatic process. Instead, the user provides a high-level specification in the form of control logic and skips directly to a finished schematic.

EDG was designed to test how effective automated reasoning could be. It successfully minimizes user effort by making *all* the decisions needed in the embedded development process. It shows that it is possible to encode all the necessary information mathematically and practicably reason with it. That robustness will be exploited in Polymorphic Blocks, EDG's successor, to build a development tool suitable for the general public.

EDG is extremely flexible in terms of both suitable input and potential output because of its use of automated reasoning. It can handle inputs that range between minimal specifications for synthesis, partial designs for completion, and complete designs for validation. Its outputs are robust to changes in available parts while accurately preserving function. Both of these are possible because automated reasoning tools see the constraint satisfaction problems (CSPs) as a context-less pile of constraints, with no limitations on evaluation or reasoning not implied by the structure of the problem. This means that a user can invoke EDG at any point in the design process in order to short circuit to the finish.

This tool, despite its proof of concept status, serves as a starting point for other tools

because it proves how powerful our methodology is. Polymorphic Blocks chooses to make the electronics model more robust, simplify the use of automated reasoning, and act as a support tool that exists alongside the standard embedded workflow. Extensions to EDG could use the same synthesis capability to generate a space of satisfying designs, bringing automation to specification finding by letting a user traverse the trade-off space for their project from the start. Alternately, the model is open to extension with other reasoning logics including temporal logic and analog signals. Finally, the integration of software in the control logic could allow for simulation and modeling of devices even without synthesis, allowing for designers to gain salient information about device function before any implementation considerations.

EDG exists to show how powerful automated reasoning is when used as part of the embedded design process, proving that an incredibly high degree of automation is possible while remaining ergonomic. It serves as a foundation for future tools by allowing us to infer what level of modeling is needed for user interactions and what can be accomplished with other automated reasoning tools.

Chapter 3

Polymorphic Blocks

3.1 Introduction

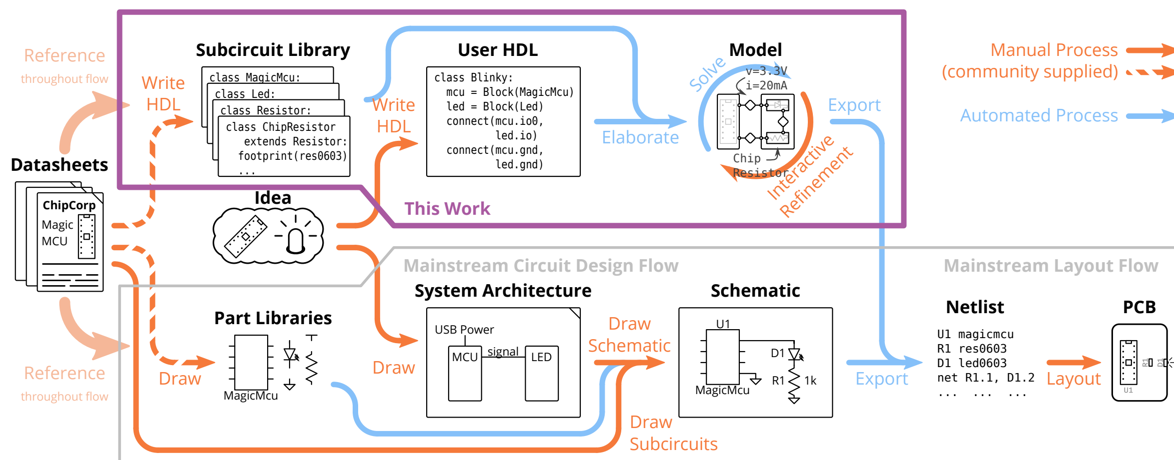


Figure 3.1: In the Polymorphic Blocks approach (purple box), circuit designers start by writing their system architecture in a hardware description language (HDL), which is then elaborated into a hierarchy block graph model and expanded using community libraries. That graph is then refined through interactive 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, direct re-use of sub-circuit files is difficult and uncommon outside limited contexts, and schematics are typically manually entered from the bottom-up using reference circuit diagrams from datasheets.

With Embedded Design Generation (EDG) serving as a proof-of-concept, we next distill

the lessons learned into a user-facing tool. While synthesis is powerful, it is not how most engineers want to work. Synthesis is too slow for on-line use, instead only being useful as a batch process. The user has little control over the output. Generally, it disempowers designers and ignores subtle factors that are not representable within its model.

An ideal tool would act in partnership with the designer, sharing control over the process, while still having a model as powerful as EDG. This is possible with a propagation engine, an automated reasoning tool which cannot find solutions out of whole cloth but can make smaller inferences based on known information. Determining salience for auto-complete and error checking, automating basic bookkeeping, and ensuring design consistency are all possible without a powerful solver. With less work being done, such a tool would be more responsive and less disruptive to designer workflows.

In this work, we strive to build tools that can support board-level design, from the first high-level system diagram sketch all the way to a layout-ready circuit. In particular, we note that hierarchy block diagrams naturally span multiple abstraction levels while being familiar to users due to their support in mainstream tools. We hypothesize that extending EDG's model with the software concepts of polymorphism and generators can raise the level of design and increase tool automation without sacrificing low-level control. In addition, we use a simpler form of automated reasoning within our system's compiler to allow useful real-time interactions with users.

Much 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. Writing these blocks as generators – executable code to translate high-level specifications into an implementation, e.g. an LED-resistor subcircuit that calculates resistance from input voltage – 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.

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 [47].

We implement this new model of circuit design in Polymorphic Blocks, an end-to-end system for authoring block diagrams. As summarized in fig. 3.1, users write designs in a hardware description language (HDL) 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 HDL, electronics model, and user interface distinguishes our work from related work on purely textual PCB HDL efforts [4, 44] and high-level design tools that don't also allow lower-level control [2].

Overall, we contribute a novel generator HDL for board-level circuit design, supporting tooling, and an accompanying evaluation. In the rest of this paper, we expand on our hierarchy block diagram model, its expression in our HDL, 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.

3.1.1 Statement on Author Contributions

This chapter is taken from the paper "Polymorphic Blocks: Unifying High-level Specification and Low-level Control for Circuit Board Design" [34] and was primarily written by Richard Lin. The contents also appear as a primary component of the thesis "Human-Centered Circuit Board Design With Flexible Levels of Abstraction and Ambiguity" [33].

Rohit Ramesh, this dissertation's author, primarily contributed to the internal model and electronics model used by PolyBlocks (sections 3.3.1 and 3.3.2) with lesser contributions to the HDL (3.3.3) and the compiler design (3.4.1). These sections comprise the core uses of automated reasoning and formal methods within the PolyBlocks tool and serve as a case study for their applicability within the context of embedded development.

3.2 Related Work

Our work relates to recent HCI research in supporting the broader electronics design lifecycle, and to specific projects that reimagine PCB and chip design tools.

3.2.1 Electronics and HCI

The HCI research community has recently seen a growth of interest in tools for electronics that cover all phases of project conceptualization, design, debugging, fabrication, and mass production. A number of projects have worked on augmented breadboards that help with physical circuit construction [14, 60, 59]. Other tools focus on introducing software programmable components, e.g. for designing analog circuits [52] or using augmented reality [30]. Other projects support constructing circuits through step-by-step tutorials~ [57] or debugging fabricated PCBs [53]. While many tools focus on enabling novices, some projects also consider how to enable scaling from electronic prototypes to mass production [28]. Our research fits into this larger landscape but focuses specifically on the task of translating ideas from system architecture diagrams into printed circuit boards.

3.2.2 PCB Design Tools

Our recent study on PCB design practices [35] revealed that while the interesting hardware design tends to happen across levels of abstraction, mainstream PCB suites such as KiCad

[29] and higher-end commercial suites like Altium [1] and Xpedition [42] operate mainly at the level of individual components. Much of the development of these tools seems to have focused on board layout, with features like interactive and sketch auto-routing, and signal integrity and power analysis. Circuit verification is typically limited to Electrical Rules Check (ERC) in the form of pin-type compatibility checks, but the coarse types (e.g., passive, input, output, power) limit usefulness. Although the circuit entry side has seen advances like hierarchical support, these are still first and foremost schematic drawing tools, not circuit design tools.

While part libraries [32] are used in mainstream tools, these are less capable than subcircuit generator libraries. Organizations may also re-use internal schematic files [41], but re-use of community schematic files is difficult and uncommon [35].

Current schematic verification revolves primarily around peer review [40], but recent commercial tools like Valydate [43] automate some schematic checks with a static model of parts. While aspects of their electrical model appear similar to ours, these are still verification, not design, tools.

Some recent academic work on PCB design tools has focused on novices. Fritzing [31] provides a breadboard view of a circuit as a conceptual bridge to the schematic view, but is still fundamentally a schematic drawing tool. AutoFritz [36] extends this with circuit autocomplete suggestions, but does not change the fundamental design abstraction. While its connection-oriented data-driven approach allows it to leverage a large corpus of existing designs, the resulting correctness guarantees are weaker than a model-based approach.

Recent work has also examined tools operating at a higher level of design. These include Trigger-Action-Circuits [2], where designs are specified at a behavioral level; Geppetto [19], where designs are specified at a block-diagram level; and circuito.io [9] and EDASolver [15], where designs are a collection of parts attached to a central microcontroller. However, lack of support for user-defined parts limits designs to a single level of abstraction, fixed by the tool. Furthermore, while these systems model electronics to some degree to synthesize working circuits, those details have not been published.

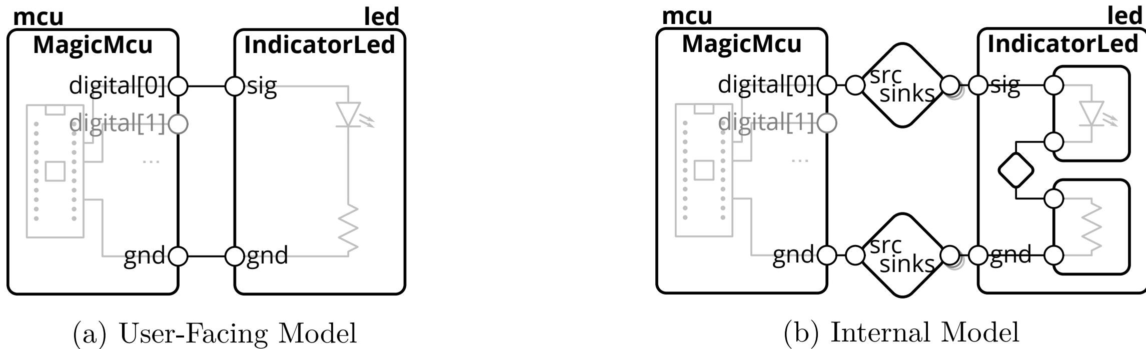
Our prior work on EDG [46] focused on the underlying blocks and links problem structure, electronics model, and synthesis algorithms, but fell short of a complete design system. This work extends EDG's model with hierarchy blocks, and combines it with an user-facing HDL and tooling to produce an end-to-end tool with an accompanying user study and analysis.

3.2.3 Chip Design and Hardware Description Languages

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. PCB HDLs like PHDL [44] are 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.

Verilog-AMS [24] and VHDL-AMS [8] 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.

Generators are an evolution on the basic HDL, encoding the rules to generate a family of similar modules instead of describing a single instance. Chip-level generators include Chisel [25] for digital hardware, and OASYS [21] and BAG [11] for analog hardware. JITPCB [4] brings generators to the PCB space by embedding circuit construction primitives in a general purpose programming language. Our system also uses subcircuit generators as a key component, but augments it with electronics modeling to enable design support features like parts selection and correctness checks.



(a) User-Facing Model

(b) Internal Model

Figure 3.2: **An example of a simple blinky led circuit in our user-facing model (a) and internal model (b).** 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).

3.3 System Design

In the Polymorphic Blocks workflow, as summarized in fig. 3.1, users start with an idea and a high-level system architecture in mind. They then translate that architecture into code written in our HDL, 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, an abstract step-down converter that can be refined into buck converter subcircuits based on particular controller chips. Our visualization and refinement interface 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 complete the layout of a PCB.

In the following sections, we will use a running example of a simple blinking LED circuit, shown in fig. 3.2, 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 fig. 3.3.

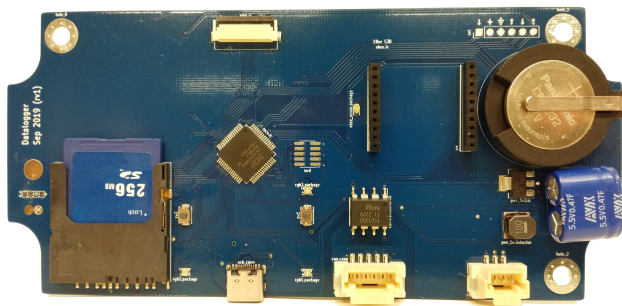


Figure 3.3: A more complex example: the datalogger PCB produced with our system, further explained in section 3.5.2.

3.3.1 Block Diagram Model

Figure 3.2 shows our model's basic 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 fig. 3.2, 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.

3.3.1.1 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 fig. 3.2 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 fig. 3.2b. In the more complex case, where multiple sub-block ports connect to a containing block port, a bridge is necessary. For example, 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.

3.3.1.2 Class Hierarchy

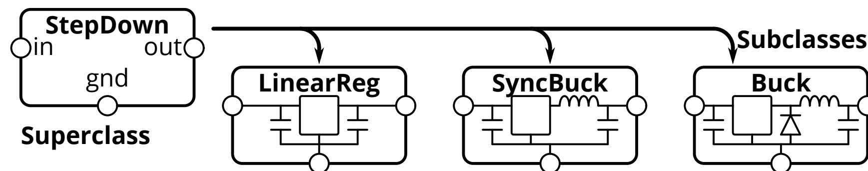


Figure 3.4: **Class hierarchy example with step-down voltage converters.** The abstract step-down converter has three subclasses, a linear regulator, a synchronous buck converter, and a buck converter. All these fulfill the step-down converter interface and functionality, and can be used in its place. This mechanism provides support for abstraction in our model.

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. Prior work [35] found that embedded designers 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 a subclass has a is-a relationship with its superclass but

can be more specific and concrete. For example, in fig. 3.4, 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.

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.

3.3.2 Electronics Model and Libraries

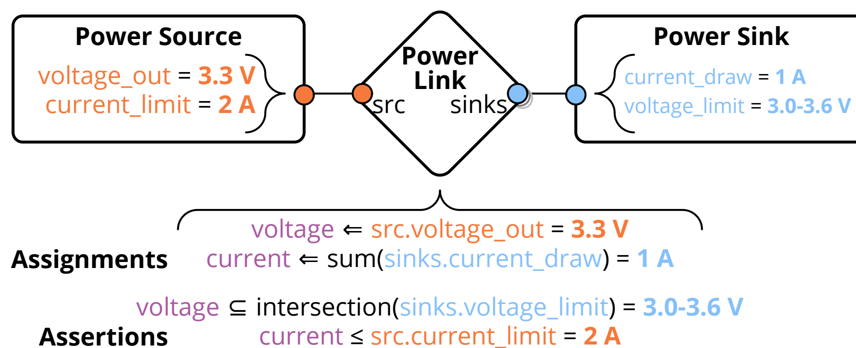


Figure 3.5: **An example of parameter propagation and checking in our model, with a simplified constant-voltage link.** Ports are defined with their physical properties: voltage output and current limits for sources, and current draw and voltage limits for sinks. These parameters “flow” through connected 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.

We built an electronics layer on top of this basic structure that 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 fig. 3.5, 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 would calculate the resistance and capacitance based on a cutoff frequency and impedance specification, while a resistive divider block would find 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 HDL.

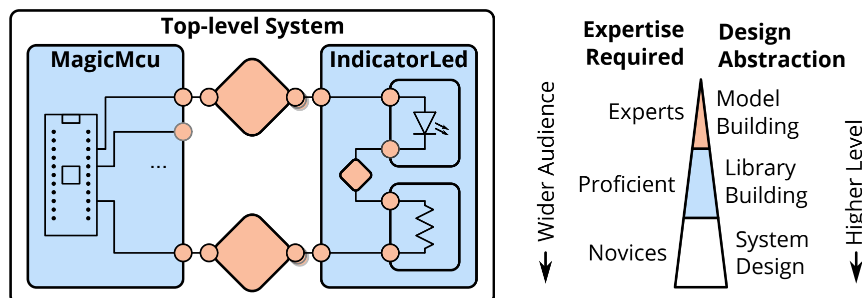


Figure 3.6: The scalable levels of design is intended to be accessible and useful to novices who can compose system-level designs using libraries, while the relatively fewer but more experienced electronics experts build those libraries of blocks and underlying port and link models.

Our overall vision of the layers of our system and how different users interact with it is summarized in fig. 3.6.

3.3.3 Hardware Description Language

Taking inspiration from recent work on chip generators [25], we provide a generator HDL 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 fig. 3.7 (which describes the diagram in fig. 3.2a, the HDL is a Python-embedded domain specific language, making use of its object-oriented features. Classes represent a re-usable block template, while objects represent individual instances. Generators defining a block's contents are written as a member function which can instantiate and connect sub-blocks, ports, and parameters.

```
class Blinky(Block):
  def contents(self):
    super().contents()
    self.mcu = self.Block(Nucleo_F303k8())
    self.led = self.Block(IndicatorLed())
    self.connect(self.mcu.gnd, self.led.gnd)
    self.connect(self.mcu.digital[0], self.led.io)
```

Figure 3.7: 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.

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

Figure 3.8: Example of an alternative structure for instantiating the Blinky circuit using implicit connect and chain. Line 2 defines the ports (microcontroller ground) that hierarchy blocks in the code block should connect to, and the tags to match. The `IndicatorLed` instantiated on line 5 defines a 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.

We also provide syntactic sugar constructs for frequent use cases as shown in fig. 3.8. 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 HDL. These constructs can be mixed with each other, as also shown in fig. 3.8, 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 fig. 3.9. The same also mostly holds true for links, given their block-like structure.

```python
class IndicatorLed(GeneratorBlock):
  def __init__(self) -> None:
    super().__init__()
    self.io = self.Port(DigitalSink())
    self.gnd = self.Port(Ground())

  def generate(self):
    super().generate()
    voltage = self.get(self.io.output_high_voltage)
    self.led = self.Block(Led())
    self.res = self.Block(Resistor(
        resistance=(voltage / 0.010,    # max current, 10 mAmp
                    voltage / 0.001)))  # min current, 1 mAmp
```

Figure 3.9: 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. We omit the internal connections for brevity.

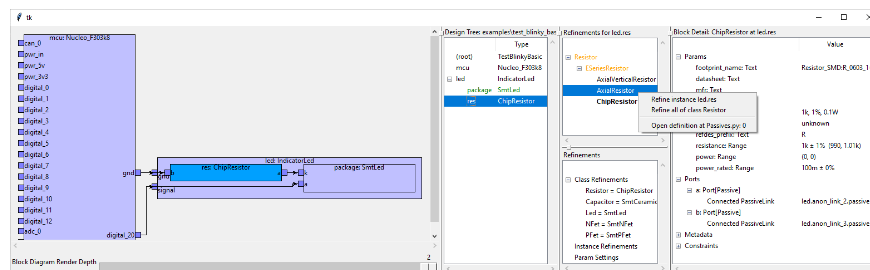


Figure 3.10: Visualization and refinement GUI with the Blinky example from fig. 3.2 open. An automatically laid out block diagram is on the left side, while tree view of the design is immediately to the right. In the design tree, abstract blocks (needing refinement) would be shown in yellow, refined 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.

3.3.4 Visualization and Refinement Interface

As prior work [35] has highlighted the need to balance control and transparency with automation, we also provide a visualization and refinement GUI. This user interface, shown in fig. 3.10, visualizes the HDL 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 HDL 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 allow automated compatibility checks.

3.3.5 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 [29] board layout tool. Otherwise, we currently 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 HDL variable names. However, this does require those names to be stable, so additional techniques will be needed to support user HDL refactoring.

3.4 System Implementation

The user-facing HDL is implemented as a library of base classes in Python, with mypy static type annotations allowing the user HDL to be type checked. The HDL compiler, netlister, and visualization interface were also written in Python with the TkInter GUI toolkit.

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

3.4.1 Compiler Structure

The hardware compiler takes the "high-level" model, as in fig. 3.2a, and incrementally "lowers" the model by adding detail and expanding sub-elements until getting to the lowest form, as in fig. 3.2b. 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.

3.4.2 Block Diagram Layout

We use ELK [16] (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.

We run a series of simplification transforms to hide internal details like bridge and adapter pseudo-blocks by collapsing them and merging their input and output edges. High-fanout links (containing over 3 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.

3.5 Example Applications

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

3.5.1 Simon

We extend the Blinky example into the Simon memory game, shown in fig. 3.11 and consisting of four colored light-up buttons and an accompanying audio tone for each color.

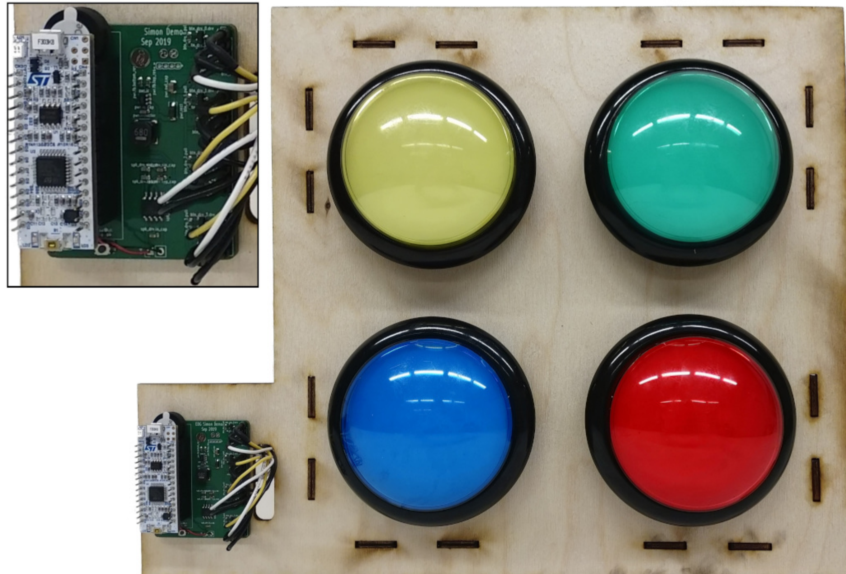


Figure 3.11: The Simon PCB (with detail view) and connected buttons.

We use a socketed Nucleo board as both a power source and microcontroller. Since the lights in the dome buttons require 12 volts while the Nucleo only supplies 5 volts, we use a boost converter to generate the necessary voltage and a MOSFET circuit to drive 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 HDL 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.

3.5.2 Datalogger

A more complex design is the datalogger, shown in fig. 3.3, 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.

3.6 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 [51]. 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".

3.6.1 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.

3.6.2 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. We did not record these sessions, but took field 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 tutorial session, in which participants worked through a tutorial document which involved building the blinky design from fig. 3.2, then extending it with a switch, LED array, discrete microcontroller, and temperature sensor. This tutorial introduced all the HDL 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 HDL.

Once participants were satisfied with their HDL, 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 HDL, 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.

3.7 User Study: Results

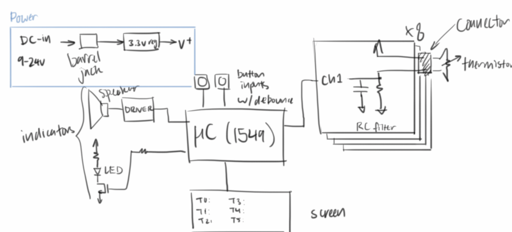
Overall, participants spent an average of 1 hour 5 minutes completing the tutorial, and 5 hours 15 minutes working on their HDL, including 2 hours building subcircuit and part libraries, and including untracked time understanding the circuits being built and becoming familiar with the system. By the 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 P02's project shown in fig. 3.12 and HDL in fig. 3.13. Further figures for all projects, including block diagram visualizations, are included in the supplemental materials.

3.7.1 Project: Power Meter

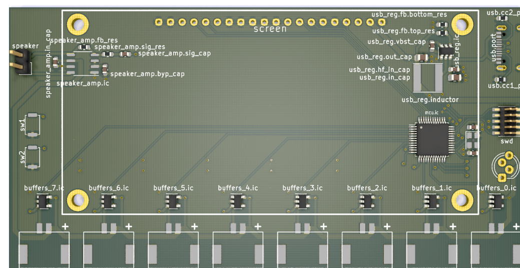
P01's project was an inline power meter that measures the voltage and current passing through it. P01 started by modeling the INA190 current sense amplifier chip, then building the top-level system with stub sub-blocks for the current and voltage sense chains, and finally implementing those sub-blocks 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 HDL writing and based on available library parts.

P01 wrote 112 lines of system-level HDL (including signal chain sub-blocks), 20 lines of generator libraries, and 95 lines of part definitions. The layout had 66 individual components.

3.7.2 Project: Thermistor Reader



(a) System Diagram



(b) Board Render

Figure 3.12: P02's initial system diagram for the thermistor reader, and the resulting PCB (rendering) they produced using our system.

P02's project was a thermistor reader that displays readings from a bank of 8 thermistors and plays an audio alert if bounds are exceeded. P02 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.

P02 wrote 52 lines of system-level HDL, 40 lines of generator libraries, and 15 lines of part definitions. The layout had 90 individual components.

3.7.3 Project: Multifunction Instrument

P03's project was an USB oscilloscope, function generator, logic analyzer, and power supply combination device, all driven from a microcontroller. P03 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.

P03 wrote 48 lines of system-level HDL, 24 lines of generator libraries, and 90 lines of part definitions. The layout had 53 individual components.

3.7.4 Advantages

Overall, participants were happy with the system architecture and level of design, with P01 noting that it matched the ideal.

```
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 3.13: The system-level HDL for P02's thermistor board, simplified for brevity.

Participants also liked the pre-built blocks and the encapsulation they provide. P02 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. P03 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 P01 considering it the best part of the system. P02 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.

3.7.5 Limitations

While participants generally felt the electrical checks were reasonable without being excessive, P01 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, P02 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, P01 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. P01 also preferred checks to be non-fatal and not prevent netlist generation where possible, though P02 preferred to not waive checks and instead use more targeted and explicit mechanisms like tightening the worst-case current draws.

P03 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 HDL may differ from the scripting aspects used by hardware designers. P01 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 P01 noted with the refinement process is that this data are stored separately from the HDL, so the HDL alone would be insufficient for a design review. Suggestions include having refinements generate code back into the HDL, or having refinements be part of review. In general, P01 and P03 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 HDL syntax issues like excessive verbosity reducing the signal-to-noise ratio.

3.7.6 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. P02 found writing the math for the RC filter calculation to be easy, and P03 noted that having an existing generator as a starting was very helpful. On the other hand, P01 pushed for an untyped port, which would in effect waive model checks for when one just wants things to connect.

3.7.7 Graphical Interfaces

All participants also made use of the visualization and refinement interface to explore the compiled designs. P01 noted that circuit reading usually relies on visual pattern matching on schematics, and it was harder to see the connectivity structure from the HDL, though P02 believed the HDL to be reasonably clear. P01 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 HDL and block diagram update loop, perhaps by integrating the visualization into an IDE. One use case suggested by P02 was to highlight block pins that still need to be connected.

Participants did have differing opinions on the HDL as a design entry interface. P02 thought the HDL with its for loop and textual entry was faster, though modern schematic tools somewhat close the gap with support for hierarchy replication. P01 noted more generally that HDLs and graphical schematic editors were suitable for different purposes, preferring schematics for analog designs with high connectivity between a few components, and preferring HDLs when the equivalent schematic sheets would be very complex and cluttered.

3.7.8 Design Time

All participants mentioned design time as a metric when comparing this system to mainstream flows, with P03 also mentioning design pain. While acknowledging that it was difficult to fairly compare time for such different flows, P02 and P03 estimated their projects would have taken about as long in a traditional flow (give or take depending on assumptions), while P01 was more wary about comparing new tools to familiar tools. P03 further noted that the end results were more “portable”, including time invested in reusable components. However, P02 was unsure about benefits when dealing with specialized, one-off components, and P01 noted the flexibility in mainstream flows to defer component sizing to quickly proceed to layout.

3.8 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.

3.8.1 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 [54] 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.

3.8.2 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.

3.8.3 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.

3.8.4 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 HDL 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 HDL, better automatic block diagram layouts (possibly with user-specified hints), and better tools for tracing and sense-making of constraint errors.

Furthermore, while an HDL 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.

3.9 Conclusion

Unlike Embedded Design Generation (EDG), Polymorphic Blocks (PolyBlocks) exists alongside the user throughout the system architecture process. Within PolyBlocks, a design exists as a mixed-abstraction block diagram that can be highly abstract, with many details left undefined; concrete, with a fully known implementation; or an arbitrary combination of the two. The user has the freedom to approach the design process, the process of going from an abstract to a concrete design, however they wish without losing PolyBlocks' support, due to its ergonomic design. As they do, PolyBlocks will provide suggestions, check for errors, and proactively present salient information. Features like parametricity and generators encourage reuse, and automate much of the work a user would have to do. When a user is finished with a design, PolyBlocks can export a netlist to board layout tools just like a more conventional schematic capture tool. This is all captured within a user-facing IDE that mimics a software development environment.

PolyBlocks, like EDG, uses a block diagram model that can be rendered into a system of constraints. Unlike EDG, it uses a simpler propagation engine to reason about those constraints. An electronics model defines how the real-world physical properties of a device are mapped to properties and constraints within a block diagram. PolyBlocks' propagation engine can view the set of constraints within a design in order to auto-complete missing elements, check for correctness, and query various hypotheticals. More abstract components simply generate fewer constraints than more concrete elements of a design, but this does not fundamentally change how the back-end sees them or what operations it can perform.

PolyBlocks' design, based on a constraint model and evaluated using algorithms from automated reasoning, is why it is a supportive design tool. Automated reasoning tools free it from having to privilege a particular abstraction level, allowing it to have the mixed-abstraction levels needed to ergonomically fit into users' existing workflows. It is possible to have high-level design information, like bus protocols, existing alongside low-level information,

like individual pin voltages, with interactions between them that are evaluated as needed. This coexistence allows PolyBlocks to proactively detect errors even in partial designs, presenting them as early as possible without compromising accuracy. Many of the other features of PolyBlocks are made possible by the abstraction agnostic nature of the propagation engine.

PolyBlocks' core approach shows that even a less powerful automated reasoning tool, a propagation engine, can provide significant benefits to an embedded design tool. Many useful tasks for a design tool are simple enough that a constraint solver would be overkill. Trading off power for speed is worthwhile for an interactive tool that must respond quickly to user actions. The core benefits of automated reasoning are still present even with less inferential power and can still be used to make tools that are more ergonomic, salient, and automated.

Chapter 4

Conclusion

Modern Electronic Design Automation (EDA) tools have stagnated due to a focus on board layout, leaving the earlier, more-important parts of the design process languishing without support. These tools lack the holistic perspective over the design process needed to effectively assist designer decision making.

This dissertation has proposed a methodology for building powerful, assistive embedded development tools through the use of techniques from automated reasoning and validated that approach in two new tools. The first, Embedded Design Generation (EDG), is a proof-of-concept system for design synthesis that demonstrates how our technique is capable of handling the process of system architecture development without human intervention. Its successor, Polymorphic Blocks (PolyBlocks), uses the lessons learned from EDG in a more conventional tool that supports the user as they work. Together they sketch a new model for embedded development tools that complements how designers approach the process.

There are a number of paths to extending and improving upon our work:

Extending the Model: The logic we use in both EDG and PolyBlocks cannot cover the full complexity of embedded design. In an immediate sense it does not need to; a logic covering basic constraints with real numbers is sufficient for reasoning about most systems. However, new types of logical statements would allow us to reason accurately about aspects of a design that we must currently treat conservatively. For instance, both tools assume that all parts in a system are always on. With a logic that includes functions over time, like linear temporal logic, we could better model the effects of power consumption when elements of a design are turned on and off. A logic for analog signal processing could replace our current model, allowing us to reason about filters and impedance. Lastly a tractable notion of paths and interfaces would allow EDG to handle code spanning multiple processors, allocating drivers in a heterogenous system.

Alternate Reasoning Methods: One-shot satisfiability solving and variable propagation are far from the only way to interact with automated reasoning tools. Solvers like Z3 [13] support more incremental interaction modes, potentially allowing clients to find multiple solutions for similar problems more efficiently than starting from scratch each time. The additional speed could make it feasible for EDG to find a Pareto frontier of designs satisfying

a specification, so users could directly examine the tradeoff space for their problem.

Broadening our Domain: It should be possible to extend our work to cover more than just the system architecture phase. Extending PolyBlocks to the software domain could lead to an extended workflow which deeply integrates cross domain simulations, creating a fast path for tests that would otherwise require prototyping. For example, a common mistake by novice designers make is placing filter capacitors and crystals too far from the components that use them. Board layout tools could check for errors like that if they knew how those components are being used, something a tool aware of context from system architecture phase could do.

Improving Usability: Multiple users have commented on how PolyBlocks would be improved with better user interfaces. Instead of a unidirectional flow from HDL to architecture diagram, they asked for bidirectional interactions where editing the diagram changed the code. Other potential improvements include a less cluttered HDL and changes to the electronics model to make extending it easier.

One of the broader lessons from this work is that automated reasoning can be useful in design tools targeting novices. Other fields use automated reasoning within development tools, but they use it to ensure correctness and prevent errors. Formal methods increase the difficulty of integrated circuit design and software development, at least until the task grows so large or costly that the lower error rate wins out. This doesn't have to be the only paradigm for its use. Instead, other types of development tools should consider using automated reasoning to improve the entry level experience, rather than focusing on the hardest tasks.

EDG and PolyBlocks serve to show what is possible when the embedded design process is augmented with automated reasoning. Between functional demonstrations and tests with outside users, our methodology has proven its worth. There is the potential to both improve our existing systems and bringing these ideas somewhere new. Ultimately, the true test of this work is in the hands of designers; seeing whether tools based on these ideas are enthusiastically adopted by a community of users.

Bibliography

[1] Altium. *Altium Designer*. 2018. URL: https://www.altium.com/altium-designer/.

[2] 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. New York, NY, USA: ACM, 2017, pp. 331–342. URL: http://doi.acm.org/10.1145/3126594.3126637.

[3] Autodesk. *EAGLE | PCB Design Software*. 2018. URL: https://www.autodesk.com/products/eagle/overview.

[4] Jonathan Bachrach, David Biancolin, Austin Buchan, Duncan W Haldane, and Richard Lin. "JITPCB". In: *Intelligent robots and systems (IROS), 2016 IEEE/RSJ international conference on*. IEEE, 2016, pp. 2230–2236.

[5] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Second. IOS Press, 2021, pp. 1267–1329.

[6] H. Beyer and K. Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. Interactive technologies series. Morgan Kaufmann, 1998. URL: https://books.google.com/books?id=T8pcH4QjATkC.

[7] Nicola Bezzo, Peter Gebhard, Insup Lee, Matthew Piccoli, Vijay Kumar, and Mark Yim. "Rapid Co-Design of Electro-Mechanical Specifications for Robotic Systems". In: *ASME 2015 international design engineering technical conferences and computers and information in engineering conference*. American Society of Mechanical Engineers, 2015, V009T07A009–V009T07A009.

[8] Ernst Christen, Kenneth Bakalar, Allen M Dewey, and Eduard Moser. "Analog and Mixed-Signal Modeling Using the VHDL-AMS Language". In: *36th design automation conference*. 1999, pp. 21–25.

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

[10] Sylvain Conchon, David Déharbe, David M. Heizmann, and Tjark Weber. *SMT-COMP 2016:* Mar. 2017. URL: http://smtcomp.sourceforge.net/2016/index.shtml.

[11] 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.

[12] Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L. Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. “metroII: A Design Environment for Cyber-Physical Systems”. In: *ACM Trans. Embed. Comput. Syst.* 12.1s (Mar. 2013), 49:1–49:31. URL: http://doi.acm.org/10.1145/2435227.2435245.

[13] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the construction and analysis of systems*. TACAS’08/ETAPS’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[14] Daniel Drew, Julie L Newcomb, William McGrath, Filip Maksimovic, David Mellis, and Björn Hartmann. “The Toastboard: Ubiquitous Instrumentation and Automated Checking of Breadboarded Circuits”. In: *Proceedings of the 29th annual symposium on user interface software and technology*. 2016, pp. 677–686.

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

[16] Eclipse Foundation. *Eclipse Layout Kernel*. 2020. URL: https://www.eclipse.org/elk/.

[17] Iacopo Giangrandi. *Etching Printed Circuits Boards at Home*. [Accessed 13-12-2023]. 2017. URL: https://www.giangrandi.org/electronics/pcb/pcb.shtml.

[18] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. “Synthesis of Loop-Free Programs”. In: *SIGPLAN Not.* 46.6 (June 2011), pp. 62–73. URL: http://doi.acm.org/10.1145/1993316.1993506.

[19] Gumstix. *Geppetto*. 2018. URL: www.gumstix.com/geppetto/.

[20] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. “Complete Completion Using Types and Weights”. In: *SIGPLAN Not.* 48.6 (June 2013), pp. 27–38. URL: http://doi.acm.org/10.1145/2499370.2462192.

[21] 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.

[22] Marijin Heule, Matti Järvisalo, and Tomář Balyo. *The International Sat Competition Wepage*. July 2016. URL: http://www.satcompetition.org/.

[23] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. “Constrained Synthesis from Component Libraries”. In: *13th International Conference on Formal Aspects of Component Software (FACS)*. Besancon, France, Oct. 2016.

[24] Accellera System Initiative. “Verilog-AMS Language Reference Manual”. In: (2014).

[25] 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.

[26] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. "Oracle-Guided Component-Based Program Synthesis". In: *Proceedings of the 32nd ACM/IEEE international conference on software engineering (ICSE)*. May 2010, pp. 215–224.

[27] Susmit Jha and Sanjit A. Seshia. "A Theory of Formal Synthesis via Inductive Learning". In: *Acta Informatica* (2017), pp. 1–34. URL: http://dx.doi.org/10.1007/s00236-017-029 4-5.

[28] 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.

[29] KiCad. *KiCad EDA*. 2018. URL: http://kicad-pcb.org/.

[30] Yoonji Kim, Youngkyung Choi, Hyein Lee, Geehyuk Lee, and Andrea Bianchi. "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.

[31] 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. New York, NY, USA: Association for Computing Machinery, 2009, pp. 351–358. URL: https://doi.org/10.1145/1517664.15177 35.

[32] Ultra Librarian. 2020. URL: https://www.ultralibrarian.com/.

[33] Richard Lin. "Human-Centered Circuit Board Design with Flexible Levels of Abstraction and Ambiguity". PhD thesis. Dec. 2021. URL: http://www2.eecs.berkeley.edu/Pubs/Te chRpts/2021/EECS-2021-259.html.

[34] Richard Lin, Rohit Ramesh, Connie Chi, Nikhil Jain, Ryan Nuqui, Prabal Dutta, and Björn Hartmann. "Polymorphic Blocks: Unifying High-level Specification and Low-level Control for Circuit Board Design". In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Virtual Event USA: ACM, Oct. 2020, pp. 529–540. URL: 10.1145/3379337.3415860.

[35] Richard Lin, Rohit Ramesh, Antonio Iannopollo, Alberto Sangiovanni Vincentelli, Prabal Dutta, Elad Alon, and Björn Hartmann. "Beyond Schematic Capture: Meaningful Abstractions for Better Electronics Design Tools". In: *Proceedings of the 2019 CHI conference on human factors in computing systems*. CHI '19. New York, NY, USA: Association for Computing Machinery, 2019. URL: https://doi.org/10.1145/3290605.33 00513.

[36] Jo-Yu Lo, Da-Yuan Huang, Tzu-Sheng Kuo, Chen-Kuo Sun, Jun Gong, Teddy Seyed, Xing-Dong Yang, and Bing-Yu Chen. "AutoFritz: Autocomplete for Prototyping Virtual Breadboard Circuits". In: *Proceedings of the 2019 CHI conference on human factors in computing systems*. CHI '19. New York, NY, USA: Association for Computing Machinery, 2019. URL: https://doi.org/10.1145/3290605.3300633.

[37] Ankur Mehta, Nicola Bezzo, Peter Gebhard, Byoungkwon An, Vijay Kumar, Insup Lee, and Daniela Rus. "A Design Environment for the Rapid Specification and Fabrication of Printable Robots". In: *Experimental robotics: The 14th international symposium on experimental robotics*. Ed. by M. Ani Hsieh, Oussama Khatib, and Vijay Kumar. Cham: Springer International Publishing, 2016, pp. 435–449. URL: http://dx.doi.org/10.1007/978-3-319-23778-7_29.

[38] Ankur M Mehta, Joseph DelPreto, Benjamin Shaya, and Daniela Rus. "Cogeneration of Mechanical, Electrical, and Software Designs for Printable Robots from Structural Specifications". In: *Intelligent robots and systems (IROS 2014), 2014 IEEE/RSJ international conference on*. IEEE, 2014, pp. 2892–2897.

[39] David A. Mellis, Leah Buechley, Mitchel Resnick, and Björn Hartmann. "Engaging Amateurs in the Design, Fabrication, and Assembly of Electronic Devices". In: *Proceedings of the 2016 ACM conference on designing interactive systems*. DIS '16. New York, NY, USA: ACM, 2016, pp. 1270–1281. URL: http://doi.acm.org/10.1145/2901790.2901833.

[40] 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.

[41] 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.

[42] Mentor. *Xpedition Enterprise*. 2018. URL: https://www.mentor.com/pcb/xpedition/.

[43] Mentor. *Xpedition Valydate Schematic Analysis*. 2020. URL: https://www.mentor.com/pcb/xpedition/schematic-analysis/.

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

[45] Raf Ramakers, Kashyap Todi, and Kris Luyten. "PaperPulse: An Integrated Approach for Embedding Electronics in Paper Designs". In: *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. ACM, 2015, pp. 2457–2466.

[46] Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. "Turning Coders into Makers: The Promise of Embedded Design Generation". In: *Proceedings of the 1st annual ACM symposium on computational fabrication*. SCF '17. New York, NY, USA: ACM, 2017, 4:1–4:10. URL: http://doi.acm.org/10.1145/3083157.3083159.

[47] Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, Ted Selker, and Mike Eisenberg. “Design Principles for Tools to Support Creative Thinking”. In: (2005).

[48] Alberto Sangiovanni-Vincentelli. “Quo Vadis, SLD? Reasoning about the Trends and Challenges of System Level Design”. In: *Proceedings of the IEEE* 95.3 (2007), pp. 467–506.

[49] Valkyrie Savage, Xiaohan Zhang, and Björn Hartmann. “Midas: Fabricating Custom Capacitive Touch Sensors to Prototype Interactive Objects”. In: *Proceedings of the 25th annual ACM symposium on user interface software and technology*. ACM, 2012, pp. 579–588.

[50] Scott. *Tutorials for KiCad - A Free Open Source Schematic and PCB Layout Editor*. [Accessed 12-12-2023]. 0. URL: https://store.curiousinventor.com/guides/kicad/.

[51] Ben Shneiderman. “Creativity Support Tools: Accelerating Discovery and Innovation”. In: *Commun. ACM* 50.12 (Dec. 2007), pp. 20–32. URL: https://doi.org/10.1145/1323688.1323689.

[52] 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.

[53] 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.

[54] Tabula. *Tabula*. 2020. URL: https://tabula.technology/.

[55] *The History of Printed Circuit Board PCB 1880 - Present*. [Accessed 11-12-2023]. 0. URL: https://how2electronics.com/history-printed-circuit-board-pcb-nextpcb/.

[56] Tube Time. *The Original Gerber Photoplotter. Yes, the Same Gerber Format We Use for PCBS Today. Pic.twitter.com/4tyWsox9yl*. July 2020. URL: https://twitter.com/TubeTimeUS/status/1280669013220012032.

[57] Jeremy Warner, Ben Lafreniere, George Fitzmaurice, and Tovi Grossman. “ElectroTutor: Test-driven Physical Computing Tutorials”. In: *Proceedings of the 31st annual ACM symposium on user interface software and technology*. 2018, pp. 435–446.

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

[59] Te-Yen Wu, Hao-Ping Shen, Yu-Chian Wu, Yu-An Chen, Pin-Sung Ku, Ming-Wei Hsu, Jun-You Liu, Yu-Chih Lin, and Mike Y Chen. “CurrentViz: Sensing and Visualizing Electric Current Flows of Breadboarded Circuits”. In: *Proceedings of the 30th annual ACM symposium on user interface software and technology*. 2017, pp. 343–349.

[60] 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*. 2017, pp. 311–319.