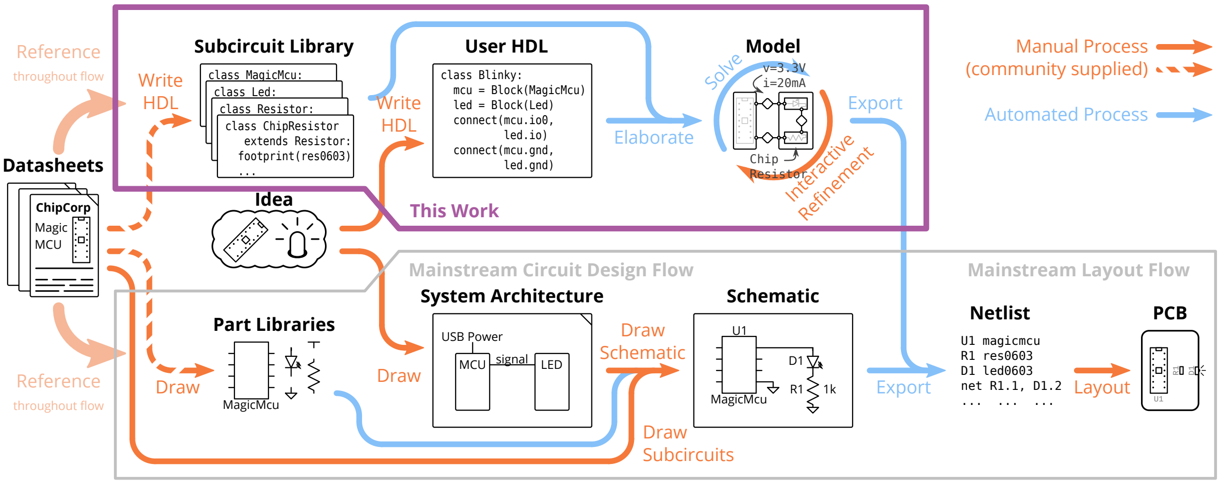
Polymorphic Blocks: Unifying High-level Specification and Low-level Control for Circuit Board Design

Richard Lin, Rohit Ramesh, Connie Chi, Nikhil Jain, Ryan Nuqui, Prabal Dutta, Björn Hartmann
University of California, Berkeley
{richard.lin, rkr, conniejchi, nikhil.jain, ryannuqui, prabal, bjoern}@berkeley.edu



```
class MagicMcu:
 class Led:
  class Resistor:
   class ChipResistor
     extends Resistor:
    footprint(res0603)
    ...
```

```
class Blinky:
  mcu = Block(MagicMcu)
  led = Block(Led)
  connect(mcu.io0,
          led.io)
  connect(mcu.gnd,
          led.gnd)
```

```
U1 magicmcu
R1 res0603
D1 led0603
net R1.1, D1.2
...  ...  ...
```

Figure 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.

ABSTRACT

Mainstream board-level circuit design tools work at the lowest level of design — schematics and individual components. While novel tools experiment with higher levels of design, abstraction often comes at the expense of the fine-grained control afforded by low-level tools. In this work, we propose a hardware description language (HDL) approach that supports users at multiple levels of abstraction from broad system architecture to subcircuits and component selection. We extend the familiar hierarchical block diagram with polymorphism to include abstract-typed blocks (e.g., generic resistor supertype) and electronics modeling (i.e., currents and voltages). Such an approach brings the advantages of reusability and encapsulation from object-oriented programming, while addressing the unique needs of electronics designers such as physical correctness verification. We discuss the system design, including fundamental abstractions, the block diagram construction HDL, and user interfaces to inspect and fine-tune the design; demonstrate example designs built with our system; and present feedback from intermediate-level engineers who have worked with our system.

UIST '20, October 20–23, 2020, Virtual Event, USA
© 2020 Copyright is held by the author/owner(s).
ACM ISBN 978-1-4503-7514-6/20/10.
http://dx.doi.org/10.1145/3379337.3415860

Author Keywords
printed circuit board (PCB) design; circuit design; hardware description language (HDL).

CCS Concepts
•Hardware → PCB design and layout; •Software and its engineering → Domain specific languages;

INTRODUCTION

Circuit design, especially at the printed circuit board (PCB) level, is an integral part of most electronic device design. A typical workflow starts with a high level system diagram capturing all major functional blocks in a device (such as processing, power, or IO) without necessarily defining their implementation [19]. From there, designers iteratively refine blocks to get the low-level circuit schematic necessary to continue to board layout. This step tends to require a significant body of knowledge spanning many sub-domains such as analog circuits, power systems, and digital logic.

Modern electronic design automation (EDA) tools largely focus on schematic capture and layout – the data entry after the actual system architecture and circuit design problem is solved. They come into play too late in the design process to provide more fundamental design assistance.

In our 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 this basic model with the software concepts of polymorphism and generators can raise the level of design and increase tool automation without sacrificing low-level control.

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. a 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 [27].

We implement this new model of circuit design in Polymorphic Blocks, an end-to-end system for authoring block diagrams. As summarized in Figure 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 [3, 25] 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.

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.

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 [7, 34, 33]. Other tools focus on introducing software programmable components, e.g. for designing analog circuits [29] or using augmented reality [16]. Other projects support constructing circuits through step-by-step tutorials [32] or debugging fabricated PCBs [30]. While many tools focus on enabling novices, some projects also consider how to enable scaling from electronic prototypes to mass production [14]. Our research fits into this larger landscape but focuses specifically on the task of translating ideas from system architecture diagrams into printed circuit boards.

PCB Design Tools

Our recent study on PCB design practices [19] revealed that while the interesting hardware design tends to happen across levels of abstraction, mainstream PCB suites such as KiCad [15] and higher-end commercial suites like Altium [1] and Xpedition [23] 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 [18] are used in mainstream tools, these are less capable than subcircuit generator libraries. Organizations may also re-use internal schematic files [22], but re-use of community schematic files is difficult and uncommon [19].

Current schematic verification revolves primarily around peer review [21], but recent commercial tools like Valydate [24] 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 [17] provides a breadboard view of a circuit as a conceptual bridge to the schematic view, but is still fundamentally a schematic drawing tool. AutoFritz [20] 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 [10], where designs are specified at a block-diagram level; and circuito.io [5] and EDASolver [8], 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 [26] 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.

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 [25] 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 [12] and VHDL-AMS [4] 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 [13] for digital hardware, and OASYS [11] and BAG [6] for analog hardware. JITPCB [3] 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.

SYSTEM DESIGN

In the Polymorphic Blocks workflow, as summarized in Figure 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

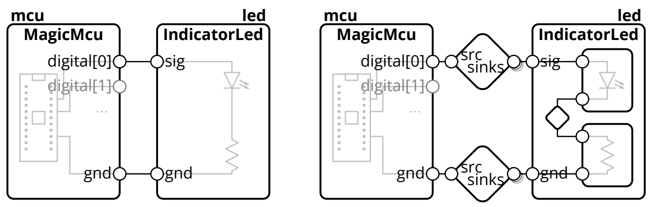


Figure 2. An example of a simple blinky LED circuit in our user-facing model (left) and internal model (right). The simplified user-facing model is presented at a single level of hierarchy, and contains just blocks (rectangles) with ports (circles) that can be connected. This largely follows representations in system architecture diagrams. The more detailed internal model spans multiple levels of abstraction by including internal hierarchy, and connections are described through links (diamonds).

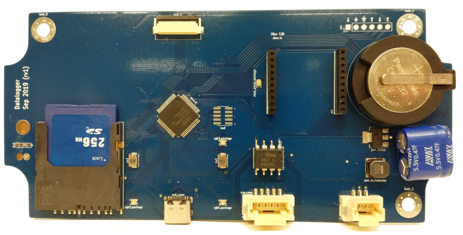


Figure 3. A more complex example: the datalogger PCB produced with our system, further explained in Section 5.2.

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 Figure 2, to introduce our model and the design workflow. However, it is important to emphasize that the system is designed to handle and produce more complex designs such as the data logger in Figure 3.

Block Diagram Model

Figure 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 Figure 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.

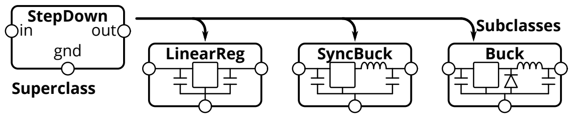


Figure 4. Class hierarchy example with step-down voltage converters. The abstract step-down converter has three subclasses, a linear regulator, a synchronous buck converter, and a buck converter. All these fulfill the step-down converter interface and functionality, and can be used in its place. This mechanism provides support for abstraction in our model.

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.

Structural Hierarchy

Modern schematic editors already support a form of structural hierarchy via hierarchy blocks, which can be placed on the schematic like ordinary components but represent a sub-“sheet” or sub-circuit instead of a single component. This serves two purposes: as an organizational tool to make large schematics comprehensible, and as a re-use tool for replicating the same circuit block. We support the same concept, as shown in Figure 2 right where the `IndicatorLed` nests internal LED and resistor sub-blocks. Generators, discussed later, further increase the encapsulation power of these hierarchy blocks.

Hierarchy support requires cross-hierarchy additions to the block model. In the simplest case, a sub-block port can be directly exported to a containing block port, as shown with the `IndicatorLed`’s ports in Figure 2 right. 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.

Class Hierarchy

The main differentiator from mainstream schematic tools is the notion of a class hierarchy for blocks. While modern schematic tools require blocks to be specific parts, we would like designers to be able to, for example, instantiate and connect a “generic” LED at that (ambiguous) level of specificity. Prior work [19] 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 Figure 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.

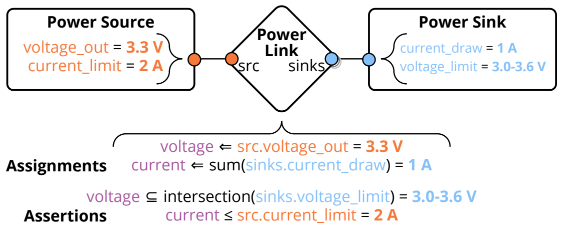

Power Source
voltage_out = 3.3 V
current_limit = 2 A
Power Link
src
sinks
Power Sink
current_draw = 1 A
voltage_limit = 3.0-3.6 V
Assignments
voltage ⇐ src.voltage_out = 3.3 V
current ⇐ sum(sinks.current_draw) = 1 A
Assertions
voltage ⊆ intersection(sinks.voltage_limit) = 3.0-3.6 V
current ≤ src.current_limit = 2 A

Figure 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.

Electronics Model and Libraries

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 Figure 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).

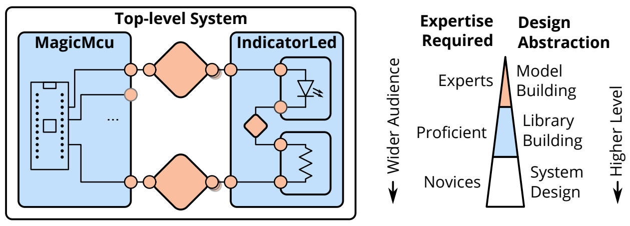


Figure 6. The scalable levels of design is intended to be accessible and useful to novices who can compose system-level designs using libraries, while the relatively fewer but more experienced electronics experts build those libraries of blocks and underlying port and link models.

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

Figure 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.

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.

Our overall vision of the layers of our system and how different users interact with it is summarized in Figure 6.

Hardware Description Language

Taking inspiration from recent work on chip generators [13], 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.

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

Figure 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.

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

Figure 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.

As shown by the Blinky code example in Figure 7 (which describes the diagram in Figure 2 left), 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.

We also provide syntactic sugar constructs for frequent use cases as shown in Figure 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 Figure 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 Figure 9. The same also mostly holds true for links, given their block-like structure.

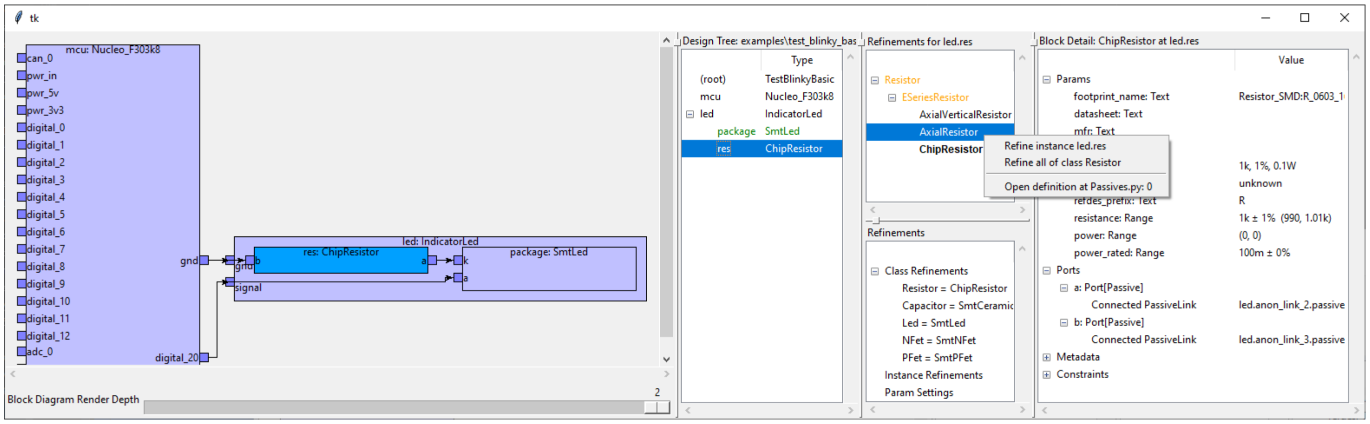
Visualization and Refinement Interface

As prior work [19] has highlighted the need to balance control and transparency with automation, we also provide a visualization and refinement GUI. This user interface, shown in Figure 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.

Board Generation

As subcircuits are fully defined at lowest level of the hierarchy block diagram, the overall design is equivalent to a schematic.



Design Tree: examples\test_blinky_bas

	Type
(root)	TestBlinkyBasic
mcu	Nucleo_F303k8
led	IndicatorLed
package	SmtLed
res	ChipResistor

Refinements for led.res

Resistor
ESeriesResistor
AxialVerticalResistor
AxialResistor
ChipResistor

Refinements

Class Refinements
Resistor = ChipResistor
Capacitor = SmtCeramic
Led = SmtLed
NFet = SmtNFet
PFet = SmtPFet
Instance Refinements
Param Settings

Block Detail: ChipResistor at led.res

	Value
Params	
footprint_name: Text	Resistor_SMD:R_0603_1
datasheet: Text	
mfr: Text	
	1k, 1%, 0.1W
	unknown
refdes_prefix: Text	R
resistance: Range	1k ± 1% (990, 1.01k)
power: Range	(0, 0)
power_rated: Range	100m ± 0%
Ports	
a: Port[Passive]	
Connected PassiveLink	led.anon_link_2.passive
b: Port[Passive]	
Connected PassiveLink	led.anon_link_3.passive
Metadata	
Constraints	

Figure 10. Visualization and refinement GUI with the Blinky example from Figure 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.

Our system can export this as a netlist file describing components and their connectivity, which can then be imported into KiCad's [15] 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.

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 entire project is open-source at `https://github.com/BerkeleyHCI/PolymorphicBlocks`.

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

Compiler Structure

The hardware compiler takes the "high-level" model, as in Figure 2 left, and incrementally "lowers" the model by adding detail and expanding sub-elements until getting to the lowest form, as in Figure 2 right. This is structured as a tree walk, from blocks to its internal ports, sub-blocks, and links, recursively. Each visited block is transformed as follows:

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

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

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

Constraint graph update: constraints between parameters are parsed into a directed graph. Constraints of the form "`a == something`" are recorded as assignments to `a`, and constraints of the form "`a subset-of something`" are recorded as bounds to `a`. Parameter values are evaluated by walking the constraint graph, and only when needed (lazily). A value may have any number of subset bounds, but only one assigned value (as long as it satisfies all subset bounds). Constraints not matching either form do not affect evaluation, and are instead recorded as assertions that are checked at the end.

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

Block Diagram Layout

We use ELK [9] (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.

EXAMPLE APPLICATIONS

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

Simon

We extend the Blinky example into the Simon memory game, shown in Figure 11 and consisting of four colored light-up buttons and an accompanying audio tone for each color.

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.

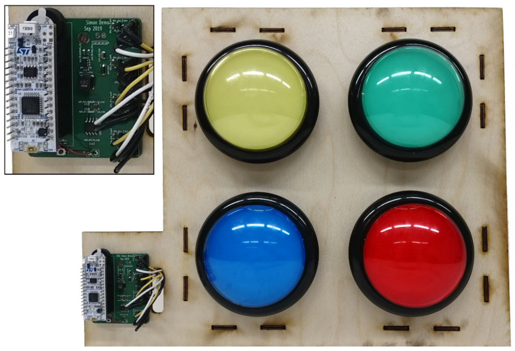


Figure 11. The Simon PCB (with detail view) and connected buttons.

Datalogger

A more complex design is the datalogger, shown in Figure 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.

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 [28]. Furthermore, we focused on qualitative feedback: as a concept significantly different from current practice, we felt that answers to "where and why does it work" which could drive future work were more interesting than a binary "does it work".

Participants

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

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

Structure

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

We asked participants to share their VM window over video conference so we could watch their progress and provide help. 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 Figure 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.

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 Figure 12 and HDL in Figure 13. Further figures for all projects, including block diagram visualizations, are included in the supplemental materials.

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.

Project: Thermistor Reader

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.

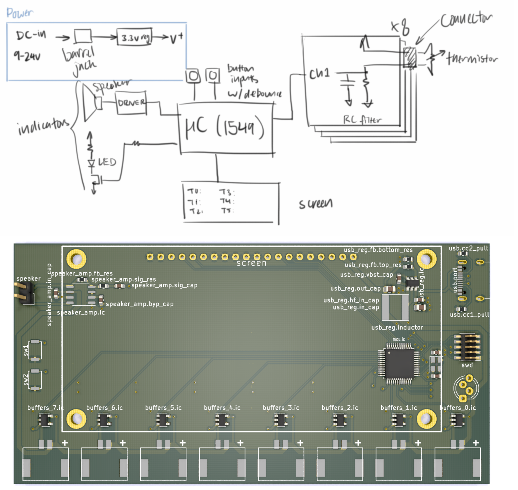


Figure 12. P02's initial system diagram for the thermistor reader, and the resulting PCB (rendering) they produced using our system.

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.

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.

Advantages

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

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,

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

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

Limitations

While participants generally felt the electrical checks were reasonable without being excessive, 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.

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.

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.

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.

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.

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 [31] and DocParser could accelerate this effort.

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

Electronics Model

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

Users and User Study

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

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

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.

CONCLUSION

Building upon recent work examining how electronics designers work and proposing a hierarchy block diagram abstraction, we implemented an HDL and compiler based on those principles and which is capable of providing meaningful design automation. System designers can compose systems using high-level blocks, while experienced engineers can provide the implementation of those blocks as re-usable generators, encapsulating design methodology in executable code.

We demonstrate the capability of this system though hardware-proven example designs, where complex subcircuits are generated from high-level specifications. Furthermore, a small-scale but realistic user study indicates that the overall abstractions are usable and useful for intermediate-level projects, and participants' feedback provided important data on limitations as well as ideas for future work.

Ultimately, we hope our work enables existing engineers to design more efficiently, and extends the reach of novices in building custom, personalized devices.

ACKNOWLEDGMENTS

This work was supported in part by NSF CNS 1505773 and CNS 1822332, Synergy: Collaborative: CPS-Security: End-to-End Security for the Internet of Things, in part by 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.

REFERENCES

[1] Altium. 2020. Altium Designer. (2020). https://www.altium.com/altium-designer/

[2] Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2017. 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)*. ACM, New York, NY, USA, 331–342. DOI: http://dx.doi.org/10.1145/3126594.3126637

[3] Jonathan Bachrach, David Biancolin, Austin Buchan, Duncan W Haldane, and Richard Lin. 2016. JITPCB. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE, 2230–2236. DOI: http://dx.doi.org/10.1109/IROS.2016.7759349

[4] Ernst Christen, Kenneth Bakalar, Allen M Dewey, and Eduard Moser. 1999. Analog and mixed-signal modeling using the VHDL-AMS language. In *36th Design Automation Conference*. 21–25.

[5] circuito.io. 2020. Circuit Design App for Makers-circuito.io. (Feb. 2020). https://www.circuito.io/

[6] J. Crossley, A. Puggelli, H.-P. Le, B. Yang, R. Nancollas, K. Jung, L. Kong, N. Narevsky, Y. Lu, N. Sutardja, E. J. An, A. L. Sangiovanni-Vincentelli, and E. Alon. 2013. 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)*. IEEE Press, 74–81.

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

[8] EDASolver. 2020. EDASolver - Automatic component selection and pin matching. (2020). https://edasolver.com

[9] Eclipse Foundation. 2020. Eclipse Layout Kernel. (2020). https://www.eclipse.org/elk/

[10] Gumstix. 2018. Geppetto. (2018). www.gumstix.com/geppetto/

[11] R. Harjani, R. A. Rutenbar, and L. R. Carley. 1989. OASYS: a framework for analog circuit synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 8, 12 (1989), 1247–1266.

[12] Accellera System Initiative. 2014. Verilog-AMS Language Reference Manual. (2014).

[13] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. DOI: http://dx.doi.org/10.1109/ICCAD.2017.8203780

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

[15] KiCad. 2020. KiCad EDA. (2020). http://kicad-pcb.org/

[16] Yoonji Kim, Youngkyung Choi, Hyein Lee, Geehyuk Lee, and Andrea Bianchi. 2019. VirtualComponent: a Mixed-Reality Tool for Designing and Tuning Breadboarded Circuits. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–13.

[17] André Knörig, Reto Wettach, and Jonathan Cohen. 2009. Fritzing: A Tool for Advancing Electronic Prototyping for Designers. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction (TEI '09)*. Association for Computing Machinery, New York, NY, USA, 351–358. DOI: http://dx.doi.org/10.1145/1517664.1517735

[18] Ultra Librarian. 2020. (2020). https://www.ultralibrarian.com/

[19] Richard Lin, Rohit Ramesh, Antonio Iannopollo, Alberto Sangiovanni Vincentelli, Prabal Dutta, Elad Alon, and Björn Hartmann. 2019. 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)*. Association for Computing Machinery, New York, NY, USA, Article Paper 283, 13 pages. DOI: http://dx.doi.org/10.1145/3290605.3300513

[20] Jo-Yu Lo, Da-Yuan Huang, Tzu-Sheng Kuo, Chen-Kuo Sun, Jun Gong, Teddy Seyed, Xing-Dong Yang, and Bing-Yu Chen. 2019. AutoFritz: Autocomplete for Prototyping Virtual Breadboard Circuits. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, Article Paper 403, 13 pages. DOI: http://dx.doi.org/10.1145/3290605.3300633

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

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

[23] Mentor. 2020c. Xpedition Enterprise. (2020). https://www.mentor.com/pcb/xpedition/

[24] Mentor. 2020d. Xpedition Valydate Schematic Analysis. (2020). https://www.mentor.com/pcb/xpedition/schematic-analysis/

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

[26] Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. 2017. Turning Coders into Makers: The Promise of Embedded Design Generation. In *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication (SCF '17)*. ACM, New York, NY, USA, Article 4, 10 pages. DOI: http://dx.doi.org/10.1145/3083157.3083159

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

[28] Ben Shneiderman. 2007. Creativity Support Tools: Accelerating Discovery and Innovation. *Commun. ACM* 50, 12 (Dec. 2007), 20–32. DOI: http://dx.doi.org/10.1145/1323688.1323689

[29] Evan Strasnick, Maneesh Agrawala, and Sean Follmer. 2017. 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*. 321–330.

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

[31] Tabula. 2020. Tabula. (2020). https://tabula.technology/

[32] Jeremy Warner, Ben Lafreniere, George Fitzmaurice, and Tovi Grossman. 2018. ElectroTutor: Test-Driven Physical Computing Tutorials. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 435–446.

[33] 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. 2017a. CurrentViz: Sensing and Visualizing Electric Current Flows of Breadboarded Circuits. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 343–349.

[34] Te-Yen Wu, Bryan Wang, Jiun-Yu Lee, Hao-Ping Shen, Yu-Chian Wu, Yu-An Chen, Pin-Sung Ku, Ming-Wei Hsu, Yu-Chih Lin, and Mike Y Chen. 2017b. 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*. 311–319.