Supporting Circuit Design with a Block-Based, Generator Language

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

Copyright held by the owner/author(s).
CHI'20,, April 25–30, 2020, Honolulu, HI, USA
ACM 978-1-4503-6819-3/20/04.
https://doi.org/10.1145/3334480.XXXXXXX

Abstract
Printed circuit boards (PCBs) are an essential part of modern electronics, yet current tools provide limited assistance in designing those circuits. While prior work has experimented with approaches like circuit synthesis, generators, and modeling; recent work has identified hierarchical block diagrams as a promising user-facing model that can extend the capability of tools without sacrificing generality. We contribute a hardware construction language and associated user interfaces which enable non-expert hardware designers to build boards by working at a high level of abstraction. The underlying hierarchy block diagram model further enables skilled engineers to build libraries of subcircuit generator blocks, encapsulating their knowledge. Extending this block model with parameters and constraints further enables an electronics model that provides meaningful automation, such as selecting parts and ensuring certain correctness properties. We discuss the design of our system, detailing both fundamental abstractions and usability trade-offs, and present a preliminary evaluation through the design of example electronics projects.

Author Keywords
printed circuit board (PCB) design; circuit design, hardware construction language (HCL).

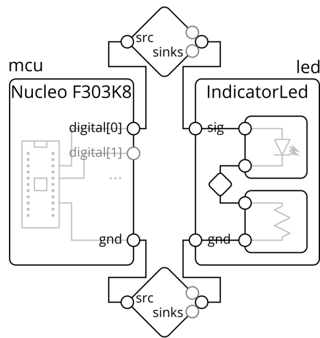

Figure 1: An example of a simple blinky LED circuit in our hierarchical block diagram model. Blocks (rectangles) have ports (circles), which are connected together through links (diamonds) Our model of electrical systems largely follows from the block diagram representations commonly found in existing design workflows. Importantly, the hierarchical nature allows spanning multiple levels of abstraction.

CCS Concepts

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

Introduction

Printed circuit boards (PCBs) are at the core of modern electronic devices across a broad range of industries. Given the ubiquity and importance of PCBs in electronics, it is worth examining the PCB design process. In the past, it was difficult to design an embedded system due to complexity and costs. Today, thanks to a wide range of online tools and resources, the barrier of entry for creating electronics has never been lower. However, many online resources still lack the capability to help guide users through their design process.

Yet, board designers still face many challenges. Primarily, a significant body of knowledge spanning many subdomains, like analog circuits, power systems, and digital logic, is required to design any nontrivial system. While electronics design automation (EDA) suites exist to support the design process, mainstream schematic design tools leaves much to be desired [7]. Their capabilities largely end at being able to draw schematics, and error checks are rudimentary with low accuracy. Often, design involves manually synthesizing the output of many separate tools.

In this work, we build upon previous work on synthesizing circuits from a high level specifications [9] by providing an user-facing hardware construction language (HCL) frontend built upon abstractions familiar to modern hardware designers. Furthermore, extension to a hierarchy block diagram model continues the support for high-level design while also allowing experienced engineers to provide implementations for those blocks and build out reusable libraries. This separation of interface from implementation enables relative novices to leverage the knowledge of experts.

In the rest of this paper, we first detail our underlying hierarchy block diagram model, then present the HCL and associated graphical user interface (GUI) for refining and exploring designs, and finally conduct a preliminary evaluate by building two electronic devices.

Related Work

Our prior work examining modern practices in board design [7] revealed that while the interesting hardware design happens at a high (system architecture) level of abstraction, mainstream schematic tools operate at a much lower (individual components) level. Thankfully, both approaches are fundamentally block diagrams, and composing them hierarchically (where a block is defined by a sub-block-diagram) allows a common model through system-level design. Here, we extend those concepts into a working system that paves the way towards an evaluation of those ideas.

As for recent work on novel electronics design tools, one approach has been hardware description languages (HDLs). The simplest is PHDL [8], which gives a textual representation of schematics and allows limited re-use. JITPCB [2] further extends the concept by embedding the HDL in a programming language and enabling circuit generators, though example applications remained fairly basic, such as arraying components. In both systems, design support automation, such as parts selection and correctness checks, is limited by the lack of an electronics model beyond connected pins. An inability to model operating conditions such as voltages and currents could mean parts are operated outside rated conditions.

Recent work has also seen high-level design tools, including Trigger-Action-Circuits [1], where designs are specified at a behavioral level, and Geppetto [4], where designs are

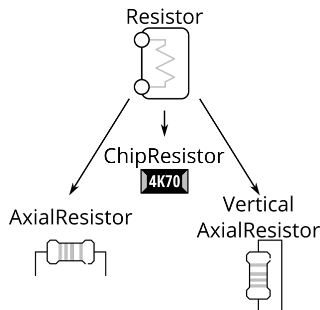

Figure 2: Type hierarchy example with resistors. Resistor has three subtypes, ChipResistor, AxialResistor, and VerticalAxialResistor, which all fulfill the resistor interface and functionality, and can be used in its place. This mechanism provides support for abstraction and ambiguity in our model.

specified at a block-diagram level. Both likely have an more advanced electronics model, as they are able to generate working circuits, but neither describes these models, nor how library components are built.

EDG [9] is more unique in detailing the underlying blocks and links problem structure, electronics model, and circuit synthesis algorithm, but with much less focus on the user interface. Our system extends that fundamental model with hierarchy blocks and combines it with generators to produce an end-to-end circuit design tool capable of high-level design.

System Design

Model and Abstractions

Our foundational models and abstractions are designed to work well for users without compromising on expressiveness. At the most primitive level we provide users with three things: first, a block diagram model for system designs; second, a type or constraint system that validates whether any given block diagram represents a functional embedded design; and finally, a specification that describes how to encode concrete properties of design components, like acceptable voltage range or pin type within the type system.

In Figure 1 we show the three main components of our block diagram model: blocks, links, and ports. Blocks represent portions of a design that can be connected together via some Links. Likewise Ports, describe specific interfaces between blocks and links. At this level we don't need to tie any of those definitions to an abstraction or abstraction level. In fact fixing a specific abstraction level, like modern schematic editors do, hinders the user by forcing them to use different tooling for different portions of their workflow.

There are two notions of hierarchy that our model uses. The first is structural hierarchy where each block or link can contain some internal structure at a lower lever of abstraction. For instance an abstract LED block, something with interfaces like "Power" and "Input Signal", can itself be made up of a sub-circuit containing the diodes, transistors, and resistors that describe components at a schematic level. This holds for Links and Ports as well, with high level interfaces like "Data Bus" containing internal links that each represent distinct electrical connections for data and clocking.

The second notion of hierarchy in our model, the type hierarchy show in Figure 2, integrates tightly with the notion of structural hierarchy. Blocks, Links, and Ports all have type signatures that we can use to check compatibility, and verify the correctness of a system design. The key property of our type system is that any particular specific implementation of an element, like a power system, is a subtype of the more general class. Altogether, this means that superclasses and hierarchy blocks provide a safe parametric abstraction for both the user and our underlying tooling.

Hardware Construction Language

As for a user-facing interface into this graph model, recent work in the chip space [5] has demonstrated the effectiveness of generator languages. Generators can not only describe a single instance of a design, but also encode the methodology to construct a class of designs. For example, an LED-resistor subcircuit generator might automatically calculate the resistance needed given the input voltage.

We follow a similar approach, providing block diagram construction primitives as functions in Python and enabling programmatic generation of hardware. Python's ease-of-use and popularity among even non software engineers make it a good candidate for host language.

As shown by the Blinky example in Figure 3, the hardware construction interface revolves around object-oriented pro-

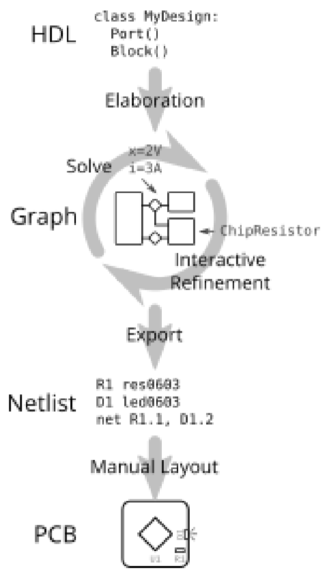

Figure 4: Overall system flow. Designers start by writing the design HDL, which is then elaborated into the hierarchy block graph model. That graph is refined through interactive choices in the GUI and solving constraints in the blocks. The result is then exported to a netlist, imported into a board design tool, and ready for manual layout.

```
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 3: 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.

gramming. Classes represent a hierarchy block template that can be re-used, while object represent individual instances. The generator defining the block's contents are written as member functions, and can call methods to instantiate sub-blocks, ports, and parameters.

Subcircuits and generators are defined similarly, as shown in Figure 5. The same also mostly holds true for links, given their block-like structure.

Links and Inference

Link types are automatically inferred based on the types of ports being connected, freeing the user from manually specify this information. Strongly typed links can detect and prevent mistakes like nonsensical connections, while also providing specific rules for parameter propagation. For example, as hinted on the LED example in Figure 5, a Digital link would propagate logic threshold voltages.

GUI

Prior work [7] has highlighted the need for control and transparency when automating circuit design, so our system features a GUI to allow interaction with a generated design.

```
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.link().out_threshold.upper())
10     self.led = self.Block(Led())
11     self.res = self.Block(Resistor(
12         resistance=(voltage / 0.010, voltage / 0.001)))
```

Figure 5: Simplified code for the indicator LED subcircuit. Lines 4 and 5 define the external ports by their types, while lines 10 and 11 define the internal blocks. Notably, as shown on line 9, generators can access solved values like input digital logic thresholds, and use those to automatically size internal blocks like the resistor. We omit the internal connections for brevity.

The current prototype, shown in Figure 6, illustrates the core required functionality of providing visibility into the system's reasoning through displaying solved values. Furthermore, the ability to set value constraints and select block refinements allows the HDL design to stay at a high level while specifics can be set interactively.

Continued work is required to make this interface more usable, such as by supplementing the tree view with an automatically laid out hierarchy block diagram [3].

Board Generation

As subcircuits are fully defined at lower levels of the hierarchy block diagram, the overall design is equivalent to a schematic. While we do not address the problem of physical board design, the system can produce a *netlist* describing components and their connectivity, which can be imported into KiCad's [6] board layout tool. The overall flow

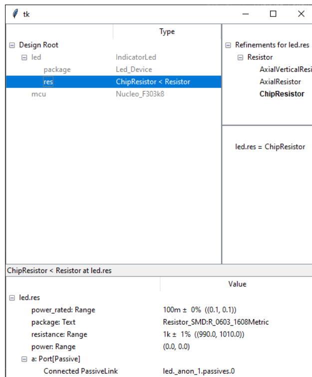

Figure 6: Prototype design explorer GUI. The tree view on the top left provides navigation through the design, while the tree view on the top right lists refinements for the currently selected element. Here, we see the options for the resistor, with the surface-mount `ChipResistor` selected and the resulting constraint listed in the center right. The bottom box displays details of the selected element.

is summarized in Figure 4.

As the overall hardware design flow involves a back-and-forth between schematic and layout, we use name stability to allow updates without losing a work-in-progress layout. However, additional strategies are needed when name changes are necessary, such as when refactoring.

Preliminary Evaluation

We conduct a preliminary evaluation of our system by constructing a few example designs, then validating the functionality of the resulting hardware.

Simon

An extension of the above Blinky example is the Simon memory game, which consists of four colored light-up buttons and an accompanying audio tone for each color.

We continue to use the Nucleo board as both a power source and the central microcontroller. Since the dome buttons require 12 volts while the Nucleo can only supply 5 volts, we use a boost converter to generate the necessary voltage, and a MOSFET circuit to switch the buttons from the 3.3 volt capable microcontroller. We further added a speaker driver, speaker connector, and debugging red-green-blue LED. In terms of structure, each of these was a library sub-block.

Overall, the top-level HDL for Simon is 58 lines of code. Of note is that the boost converter instantiation was only one line to specify the controller chip and desired output voltage. The boost converter generator library encapsulates the details and process of component sizing.

Datalogger

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

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

Libraries

As shown in the above examples, libraries are what ultimately enables significant design automation. Though we have built a library including 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.

Mixed-initiative approaches can help alleviate this process, allowing users to scan datasheets and select individual tables which can then be automatically parsed. While archaic encoding or formatting in some datasheets complicates the process, using external tools like Tabula [10] and DocParser is a potential solution.

Overall, collaboration from a large community may be key to building a critical mass of parts to support the needs of users.

Conclusion

Building upon recent work examining how electronics designers work and proposing a hierarchy block diagram abstraction, we implemented a circuit design tool based on those principles and which is capable of providing mean-

Figure 7: The Simon PCB with connected buttons. Our system is able to generate the 5v to 12v boost converter subcircuit to drive the LEDs in the dome buttons.

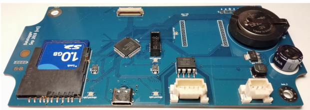

Figure 8: The datalogger PCB. Our system supports complex subcircuits such as microcontrollers application circuits and analog power generators such as the current-limited supercapacitor backup.

ingful design automation. System designers can compose systems using high-level blocks, while experienced engineers can provide the implementation of those blocks as reusable generators, encapsulating their design methodology in executable code. We demonstrate the capability of this system though example designs, where complex subcircuits are generated from high-level specifications.

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

REFERENCES

[1] 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`

[2] 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`

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

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

[5] 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`

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

[7] 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`

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

[9] 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`

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