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

CHI '20 Extended Abstracts, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6819-3/20/04.
https://doi.org/10.1145/3334480.3382887

Abstract
Modern electronic design automation (EDA) tooling tends to focus on either the system-level design or the low-level electrical connectivity between physical components on a printed circuit board (PCB). We believe that a usable and functional system for circuit design needs to be able to interleave both levels of abstraction seamlessly and allow designers to transition between them freely. Existing work has experimented with approaches like circuit synthesis, functional characterization, or fine grained physical modeling. Each of these approaches augment the design process as it exists today, with its fundamental split between various levels of abstraction. We notice that hierarchical block diagrams can capture both high-level system structure as well as fine grained physical connectivity, and use that symmetry to construct a model for electronic circuits that can span the entire design process. Additionally, we construct user interfaces for our model that can support users of different skill levels throughout a design task. We discuss the design of our system, detailing both fundamental abstractions and usability trade-offs, and demonstrate its current capabilities through the design of example electronics projects.

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

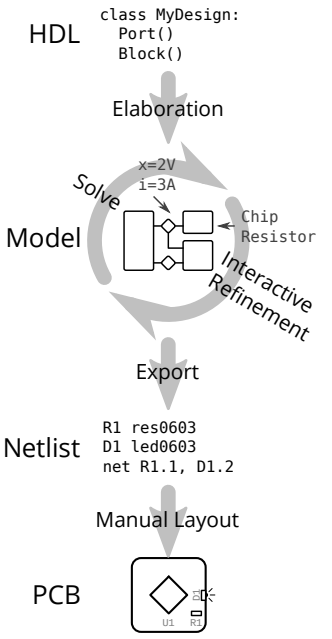

Figure 1: Overall system flow. Designers start by writing the design HDL, which is then elaborated into the hierarchy block graph model. That graph is refined through interactive choices in the GUI and automatically solved constraints in the blocks. The result is then exported to a netlist file, which can be imported into a board design tool for manual layout.

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 integral part of most electronic device design. One common design workflow today starts with a high level system diagram that captures all the major functional blocks in a device (such as processing, power, or IO) without defining most of the details needed to implement those portions of a design [10]. From there, designers recursively refine each block in until they can create a circuit diagram, a low-level representation of a design, and enter it into the tooling needed for the physical design process. This refinement process tends to require a significant body of knowledge spanning many subdomains such as analog circuits, power systems, and digital logic.

Modern electronic design automation (EDA) largely focuses on that last step, after the actual electrical design problem is solved, and where the major remaining work is data entry required to progress to physical design. EDA tools enter the design process too late to provide more fundamental design assistance, and are further limited by their weak correctness checks.

In this work we strive to build tools that can support electronics design from the first high-level systems diagram through to the creation of a netlist, the map of connections needed to layout the physical lines of copper on a PCB. In particular, we note that hierarchical block diagrams serve as a natural structure for the design process that spans across abstraction levels. Furthermore, the addition of some parametricity continues the support for high-level design while allowing experienced engineers to provide implementations for those blocks and build reusable libraries. This separation of interface from implementation enables relative novices to leverage the knowledge of experts. We foresee an open-source community of engineers and designers, similar to that in the software world, where open collaboration and communication lowers the barrier of entry into electronics design even further.

In the rest of this paper, we first detail our underlying hierarchy block diagram model, present a user-facing hardware description language (HDL) for authoring block diagrams, describe an associated graphical interface for refining and exploring designs, and finally demonstrate our system's capabilities by building two devices.

Related Work

Our prior work examining modern practices in board design revealed that while the interesting hardware design tends to happen across levels of abstraction, mainstream schematic tools operate at the level of individual components [10]. Hierarchical block diagrams were identified as a promising model that can support both high level system design and automate lower level subcircuit design.

Some recent work on novel electronics design tools has focused on novices. Fritzing [9], for example, provides a breadboard view of a circuit as a conceptual bridge to the schematic view. However, it does not offer any more design assistance than mainstream schematic tools.

Another approach has been hardware description languages (HDLs). The simplest is PHDL [12], which gives a textual representation of schematics and allows limited reuse. JITPCB [2] extends the concept by embedding circuit construction functionality into a programming language and enabling circuit generators, such as arraying components. In both systems, design support automation, such as parts

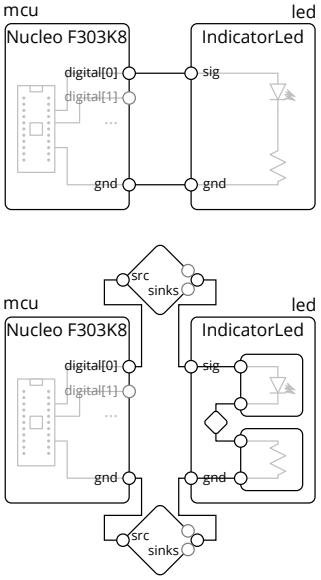

Figure 2: An example of a simple blinky LED circuit in our conceptual user-facing model (top) and internal model (bottom). The simplified user-facing model is presented at a single level of hierarchy, and contains just blocks (rectangles) with ports (circles) that can be connected. This largely follows 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).

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; Geppetto [6], where designs are specified at a block-diagram level; and circuito.io [3] and EDA-Solver [4], where designs are a collection of parts attached to a central microcontroller. As these systems are able to generate working circuits, they likely do some electronics modelling, but their details have not been published. Lack of support for user-defined parts further limits designs to a single level of abstraction, fixed by the tool.

AutoFritz [11], on the other hand, supports designers by providing circuit autocomplete suggestions. However, it, too, is limited to a single level of abstraction, that of individual components. Its connection-oriented, data-driven approach also provides weaker correctness guarantees than a model-based approach.

While EDG [13] focuses on the underlying blocks and links problem structure, electronics model, and circuit synthesis algorithm, less attention is paid to 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

The overall workflow of our system is summarized in Figure 1. Designers start by writing HDL code, which is elaborated down into a hierarchical block diagram model. Details in this model, such as electrical parameters and block sub-types, may start unknown, but are refined through a combination of user input in an interactive GUI and a solver. When fully elaborated, the flattened hierarchical block diagram encodes a schematic, and can be exported via a netlist to an external board layout tool.

In the rest of this section, we first detail the underlying model, then examine the user-facing HDL and GUI, and finally discuss integration with downstream tools.

Model and Abstractions

Our foundational model is 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 2 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 implicitly inferred links. Likewise, ports describe specific interfaces between blocks and links.

While modern schematic tools require specification of particular parts, our system enables designers to, for example, instantiate and connect an LED at that level of specificity. In fact, fixing a specific abstraction level, like modern tools do, hinders the user by requiring different tooling for different portions of their workflow.

This flexibility of abstraction layer is enabled by the 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"

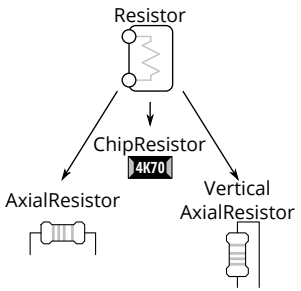

Figure 3: Type hierarchy example with resistors. Resistor has three subtypes, ChipResistor, AxialResistor, and VerticalAxialResistor, which all fulfill the resistor interface and functionality, and can be used in its place. This mechanism provides support for abstraction and ambiguity in our model.

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 busses 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 3, 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 [7] 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 HDL example in Figure 4 (which essentially describes the simplified model in Figure 2), the hardware construction interface revolves around object-oriented programming. Classes represent a hierarchy block

```
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 4: 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.

```
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      TARGET_CURRENT_MIN = 0.001;  # 1 mAmp
10     TARGET_CURRENT_MAX = 0.010;  # 10 mAmp
11     voltage = self.get(self.io.output_high_voltage)
12     self.led = self.Block(Led())
13     self.res = self.Block(Resistor(
14         resistance=(voltage / TARGET_CURRENT_MAX,
15                     voltage / TARGET_CURRENT_MIN)))
```

Figure 5: Simplified code for the indicator LED subcircuit. Lines 4 and 5 define the external ports by their types, while lines 12 and 13 define the internal blocks. Notably, as shown on line 11, 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.

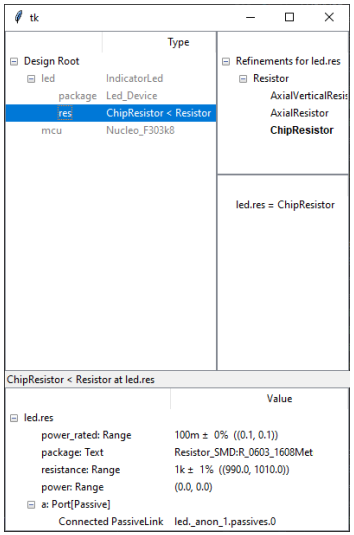

Figure 6: Prototype design explorer GUI with the Blinky example from Figure 2 open. The tree view on the top left lists the elements of the hierarch block diagram and allows navigation through the design. The tree view on the top right lists refinements for the currently selected `Resistor`, with the surface-mount `ChipResistor` selected and the resulting constraint listed in the center right. The bottom box displays details of the selected element.

template that can be re-used, while objects 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 needing to manually specify this information. Strongly typed links can detect and prevent mistakes like nonsensical electrical connections, such as between power and data wires. Furthermore, links can also provide rules for parameter propagation and limits, for example for output voltages and rated maximums, though verification remains a work-in-progress.

We do caution that an electrically correct circuit may not be functionally useful. While connecting the UART data ports of two GPSes together is electrically valid and allowed in our model, the result is nonsensical. Future work could model higher domains, such as firmware and dataflow.

GUI
Prior work [10] has highlighted the need to balance control and transparency with automation, so our system features a GUI to allow interaction with a generated design.

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 subtypes allows the HDL design to stay at a high level while specifics can be set interactively. For example, in a design that calls for a generic resistor, a user can select a particular sub-type like a surface-mount resistor.

Ongoing work includes refining the interface to be more usable, such as by supplementing the tree view with an automatically laid out hierarchy block diagram [5].

Board Generation
As subcircuits are fully defined at lower levels of the hierarchy block diagram, the overall design is equivalent to a schematic. Our system can export this netlist file describing components and their connectivity, which can then be imported into KiCad's [8] 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 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.

Example Applications

We demonstrate the capabilities 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 use a socketed Nucleo board as both a power source and microcontroller. Since the LEDs in 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 drive the illumination from a 3.3 volt pin. We further added a speaker driver, speaker con-

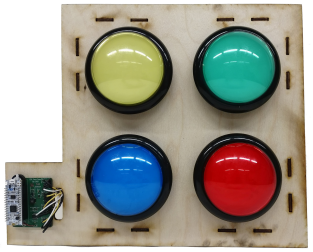

Figure 7: The Simon PCB (bottom-left) with connected buttons (right). Our system is able to generate the 5v to 12v boost converter subcircuit to drive the LEDs in the dome buttons.

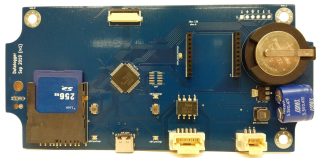

Figure 8: The datalogger PCB. Our system supports complex subcircuits such as microcontrollers application circuits and analog power generators such as the current-limited supercapacitor backup.

nector, and debugging tricolor 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 – 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.

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 directly on the board.

In addition to the mandatory 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 [14] 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 meaningful 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.

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

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

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

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

[7] 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

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

[9] 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

[10] 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

[11] 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

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

[13] 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

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