Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages

Richard Lin
richard.lin@berkeley.edu
University of California, Berkeley

Rohit Ramesh
rkr@berkeley.edu
University of California, Berkeley

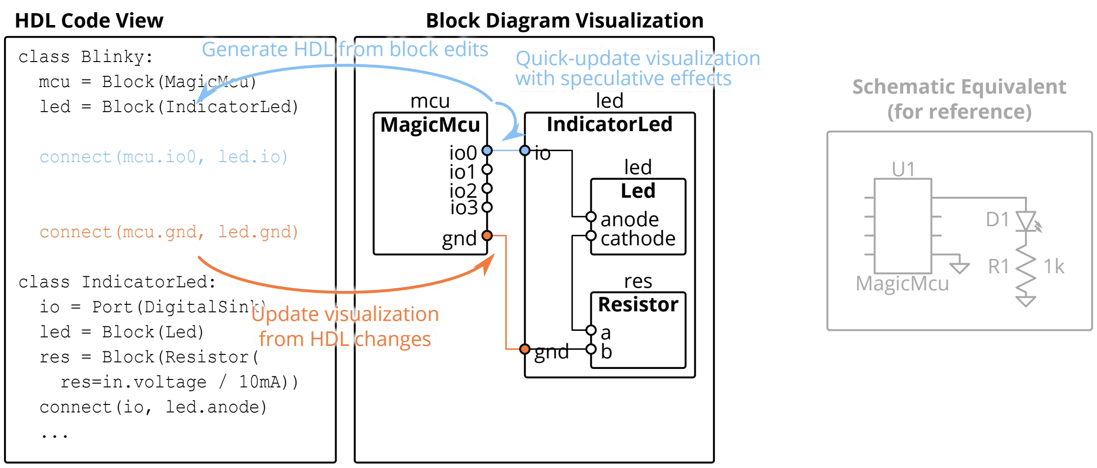
Nikhil Jain
nikhil.jain@berkeley.edu
University of California, Berkeley

Josephine Koe
koe@berkeley.edu
University of California, Berkeley

Ryan Nuqui
ryannuqui@berkeley.edu
University of California, Berkeley

Prabal Dutta
prabal@berkeley.edu
University of California, Berkeley

Björn Hartmann
bjoern@berkeley.edu
University of California, Berkeley


HDL Code View
Block Diagram Visualization
Generate HDL from block edits
Quick-update visualization with speculative effects
Update visualization from HDL changes
mcu
MagicMcu
io0
io1
io2
io3
gnd
led
IndicatorLed
io
led
Led
anode
cathode
res
Resistor
a
b
gnd
Schematic Equivalent (for reference)
U1
MagicMcu
D1
R1
1k

```
class Blinky:
  mcu = Block(MagicMcu)
  led = Block(IndicatorLed)

  connect(mcu.io0, led.io)


  connect(mcu.gnd, led.gnd)

class IndicatorLed:
  io = Port(DigitalSink)
  led = Block(Led)
  res = Block(Resistor(
    res=in.voltage / 10mA))
  connect(io, led.anode)
  ...
```

Figure 1: Left: an overview of our integrated development environment (IDE) approach for tooling to support working with circuit board-level hardware description languages (HDLs). This consists of the traditional text editor (left half of the IDE), and a block diagram visualization of the compiled HDL (right half of the IDE). Edit actions on the block diagram, such as creating connections between ports shown in blue, generate the corresponding lines of code in the HDL and update the visualization without incurring the latency of a full recompile. The HDL code, including that inserted from block diagram edit actions, can also be freely edited to preserve the full power and flexibility of the base HDL. User-triggered updates recompile the HDL, and changes such as the connection shown in orange, are visible in the block diagram and available for editing. **Right: As a reference, the equivalent design in a mainstream schematic editor.** Comparatively, schematics often require the system designer to work at the lowest level of abstraction (instead of re-using library components like `IndicatorLed`) and manually handle component calculations (like the resistor) which are both tedious and do not preserve design intent.

ABSTRACT

In many engineering disciplines such as circuit board, chip, and mechanical design, a hardware description language (HDL) approach provides important benefits over direct manipulation interfaces by supporting concepts like abstraction and generator metaprogramming. While several such HDLs have emerged recently and promised power and flexibility, they also present challenges – especially to designers familiar with current graphical workflows. In this work, we investigate an IDE approach to provide a graphical editor for a board-level circuit design HDL. Unlike GUI builders which convert an entire diagram to code, we instead propose generating

UIST '21, October 10–14, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8635-7/21/10.
https://doi.org/10.1145/3472749.3474804

equivalent HDL from individual graphical edit actions. By keeping code as the primary design input, we preserve the full power of the underlying HDL, while remaining useful even to advanced users. We discuss our concept, design considerations such as performance, system implementation, and report on the results of an exploratory remote user study with four experienced hardware designers.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**; • **Hardware** → *PCB design and layout*; *Hardware description languages and compilation*; • **Human-centered computing** → *Graphical user interfaces.*

KEYWORDS

integrated development environment (IDE), hardware description language (HDL), printed circuit board (PCB) design

ACM Reference Format:
Richard Lin, Rohit Ramesh, Nikhil Jain, Josephine Koe, Ryan Nuqui, Prabal Dutta, and Björn Hartmann. 2021. Weaving Schematics and Code: Interactive Visual Editing for Hardware Description Languages. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3472749.3474804

1 INTRODUCTION

Printed circuit boards (PCBs) are foundational to modern electronics, and are usually designed with electronics design automation (EDA) tools. In current mainstream practices [18], these tools have two main components: schematic capture for designing the circuit, and board layout for transforming the abstract circuit to a physical design. While both of these steps are necessary for transforming an idea into reality, here we will focus on the circuit design problem.

One issue with modern schematic tools is that they present a very low-level design abstraction, that of individual components and wires. While this provides high flexibility in design, it also creates problems for users. For novices and hobbyists, these tools require them to have an understanding of low-level electrical engineering details in order to build a working board. For experts and professionals, the low-level abstractions can be tedious and inefficient, requiring every degree of freedom to be resolved even when there is a large acceptable design space.

One recent approach to address these shortcomings are board-level hardware description languages (HDLs) [17], such as shown on the left half of the IDE concept in Figure 1. Much like software engineering, these languages enable a tiered approach to design through the use of libraries and abstraction: system-level designs can be made up of high-level library components which do not require deep electrical engineering knowledge to use, with those components being defined by more expert users who can encapsulate their knowledge and design process as *generators* with executable code. An example of the latter is the code for the `IndicatorLed` block in Figure 1, which calculates the needed resistance and automates a process that is otherwise manual in schematic design.

Yet, an HDL approach also comes with a different set of issues, primarily in the added requirement for programming in a specialized language. Furthermore, while both HDLs and schematics can ultimately be views on similar underlying data, each representation emphasizes different information and provides different affordances for understanding and manipulating a design [3]. For example, while an HDL might enable loops to quickly generate arrays of components, the circuit connectivity is much more obvious in a block diagram. Finally, as HDLs depend on libraries that encapsulate expert knowledge, it will be essential to make contribution as easy as possible for the experts who are typically familiar with schematic-based tools.

In this paper, we present a novel approach to bridging schematic- and HDL-based workflows. Our approach is centered on interactive block diagrams that are synchronized with HDL code: code edits are reflected in diagram updates, and diagrams can be edited to modify code. Our work relates to visual editors for other domains such as GUI editors with code generation. However, our approach differs in that code remains the primary design input, and modifications from the GUI and direct text edits can be arbitrarily interleaved. This preserves the flexibility of the underlying HDL.

Overall, we contribute an IDE implementing these techniques, both as a tool to concretely support board design, and more generally as another design point in the space of similar tools [9, 22] that bring programming power to domains beyond software. The IDE is designed to work with Polymorphic Blocks [17], our open-source board-level HDL. Furthermore, we contribute an evaluation in the form of a qualitative and remote user study with four participants, from which we draw takeaways and recommendations for similar tools. We found that even though some participants preferred to write HDL by direct text edits instead of through the block diagram interface, the tight visualization loop was beneficial to all users. Furthermore, participants were able to blend use of the block diagram interface with textual HDL edits, suggesting that this mixed IDE approach is viable and even if not all possible code edits are supported from the GUI.

2 RELATED WORK

As a project covering electronics and programming environments, we relate to a large body of prior work on electronics design, hardware description languages in general, and live programming.

2.1 Electronics Design

Academically, there has been much work on board-level electronics in the HCI community. Broadly speaking, the same level of component-and-wire based circuit design is often prototyped on solderless breadboards, and a thread of work exists on augmented breadboards [4, 14, 36, 37]. For boards specifically, there is another thread of work on debugging and prototyping assist technology [32, 33] as well as examining how to scale from prototype to production [12].

As for design tools, much prior work revolves around synthesizing devices from a high-level specification. Both Embedded Design Generation [27] and Echidna [25, 26] start with user-defined functional components like sensors and motors, and perform interface-driven synthesis to complete the rest of the device. Trigger Action Circuits [2] works at a higher, behavioral and dataflow level, and produces a breadboarding diagram. EDASolver [5] also synthesizes circuits from specifications, though details have not been published.

Related to both breadboarding and circuit boards is Fritzing [15], which provides both views of a circuit to bridge concepts for novices. AutoFritz [19] builds on top of that with data-driven circuit autocomplete suggestions.

The current dominant paradigm [18] of interactive schematic capture can be seen in both open-source [13] and commercial tools [1, 23], and has been around for almost half a century [20] with little fundamental modifications. Here, users draw schematics on a virtual canvas by placing components and connecting their pins together, offering both benefits and drawbacks of very low-level design. While much continuing commercial work appears to focus on layout-related issues like autorouting and signal integrity [23], there has also been some work related to the circuit itself, such as Valydate [24] which automates parts of schematic review.

A recent paradigm is module-based board design, as with Sparkfun À La Carte [31] and Geppetto [8]. These provide a mixed schematic and layout view, with users working with high-level modules that represent a subcircuit, and placing its bounding box on a physical board view. However, details have not been published, so their internal models, library creation process, and algorithms are not known. Related is MorphSensor [39], which provides a 3D design view for flexible PCBs. It, too, has a module-based flow with mixed schematic and physical views, though avoids the circuit design problem by importing schematics from mainstream tools.

Overall, one trend of recent tools is that they are library-based and model components at a more detailed level than mainstream tools. Yet, these projects do not really address library creation by end users, a step necessary to achieve scalability and generality beyond what a team of tool developers could feasibly do.

2.2 Hardware Description Languages

Polymorphic Blocks [17] attempts to address this problem by providing a hardware description language (HDL) interface suitable for writing system-level designs and library components. Library reusability is further supported by adapting programming language concepts like type hierarchies and polymorphism, such as by having an abstract buck converter component that can be implemented by concrete sub-types like subcircuits around specific chips. Furthermore, the code aspect of an HDL approach enables writing generators that can encode a design methodology for a family of designs instead of a static instance, such as by sizing the buck converter given higher-level parameters.

Though powerful, this textual interface is very different than mainstream schematic capture tools. While Polymorphic Blocks included a Visualization and Refinement Interface that provided a block diagram visualization of the compiled HDL, this was not integrated into the text editor, required a lengthy compilation delay on each update, and could not generate edits to the HDL. In particular, we note that all participants from the user study asked for a tighter HDL editing and block diagram visualization loop, which is what we address in this work.

In the broader context, Polymorphic Blocks was inspired by work on Chisel [11], a generator HDL for digital logic for chip design. We note that HDLs like Verilog and VHDL are currently the dominant approach to digital logic design in the chip industry. Although visualizers exist for digital logic designs and diagrams are often used to express microarchitectural designs, visualizations are not commonly part of digital logic HDL flows. Yet, aspects of this work may be applicable to the digital logic domain.

There is also a thread of work addressing the lengthy, often hours-long latency for chip tools to produce results from a HDL change – a bottleneck on productivity. Strategies include improving synthesis performance [35] and simulation performance [29, 30], such as through partitioning and incremental compilation. While board-level designs are orders of magnitude less complex than chip designs, we do adopt similar techniques to reduce latency.

2.3 Live Programming

Our work is related to *live programming* in programming languages, where code being written is continuously run with results displayed to the user. These tools can help connect the more abstract representation of code to concrete examples [6], which users might find easier to work with or to help navigate execution contexts [21]. Prior empirical research [16] further indicates that simpler liveness tools were frequently used, and even small amounts of liveness can have disproportionate impact — which is what we attempt to do here.

One critical aspect of live programming systems is the latency between code change and visible effects [28]. Prior work has discussed possible techniques to reduce latency [21, 34], such as predictively starting to compute results or returning speculative results. As our underlying HDL's compiler is not performant enough to fully recompile every change at interactive rates, our system relies on speculative results to preserve a smooth interaction flow.

Beyond one-way visualization, work has also examined *output direct manipulation* – how these visual representations could be manipulated to write code in specific domains. One recent example is Sketch-n-Sketch [9], which provides a graphical editor for a custom language for 2D graphics. Other examples include such editors for string manipulation and diagramming [22].

However, perhaps the most often used similar class of tools are graphical user interface (GUI) builders, where users can define their GUI graphically through direct manipulation interactions such as drag-and-drop. These tools similarly aim to provide an interface that is closer to the domain than the equivalent GUI construction code they generate. Yet, the code generated often is not stylistically clean or even meant to be directly edited [38] – perhaps fine for a GUI with defined integration points and static structure, but less so for an HDL where programming the structure is the point.

In this larger context, we aim to develop tools that can harness the power and flexibility of an HDL, but with an interface that is closer to how circuits are designed today: schematics and block diagrams. We hope this schematic-like representation can provide a complementary view for editing the HDL, and furthermore a graphical interface may make HDLs more accessible to a broader community of hardware engineers, device designers, and hobbyists with more limited programming expertise. We further differentiate our work through the emphasis on practicality and flexibility by supporting a Python-embedded HDL instead of a fully custom programming language and co-existing with free-form textual edits to the HDL.

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

Figure 2: Example project code which consists of a PJ-102A barrel jack, a LPC1549JBD48 microcontroller, and an indicator LED.

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

Figure 3: Example part definition of a Lf21215TMR digital magnetic field sensor.

3 BACKGROUND

As our IDE is designed to work with Polymorphic Blocks [17], a novel HDL that has not yet seen widespread adoption, we will briefly recap the HDL in this section to provide readers with a more concrete understanding of what our IDE does. While our IDE integrates ongoing development on Polymorphic Blocks, the fundamental concept remains true to the original paper. The main user-facing changes are moving the refinement data into the user HDL (from the separate Visualization and Refinement Interface) and changing to a directed constraints model.

Overall, Polymorphic Blocks is a Python-embedded domain specific language with object-oriented features. Classes represent reusable block templates and objects are individual instances of those blocks. The abstraction capabilities provided by this framework allow the HDL to be used for both constructing top-level designs and writing part definitions. Examples of both are presented in the rest of this section.

3.1 Top-Level Design

The example code in Figure 2 (and its corresponding block diagram visualization shown in the IDE screenshot in Figure 4) demonstrate how blocks and connections are instantiated in the HDL in a top-level design that powers and controls an LED light. Lines 4, 10, and 11 respectively instantiate and name `Block` objects for a barrel jack, microcontroller, and indicator LED. Some component blocks additionally have assignable parameters, such as the target output voltage of the barrel jack in line 4, allowing for more powerful design correctness checks.

Line 13 demonstrates use of the `connect(...)` function to connect a digital IO port on the `mcu` block to the signal port of `led`. Lines 6 - 13 demonstrate `implicit_connect`, a syntactic sugar construct to simplify certain types of common connections, like for power and ground. In short, this works by matching the tags such as `Power` and `Common` (ground) in lines 7 - 8 with the tags specified in components' ports (not shown)

Lines 15 - 19 further refine the abstract `PowerBarrelJack` instantiated in line 4 to be the specific and concrete subtype, the PJ-102A. In this manner, users can initially instantiate `Blocks` of an abstract type to preserve potential alternatives, then later refine them to a more specific subtype depending on design requirements.

3.2 Defining Library Blocks

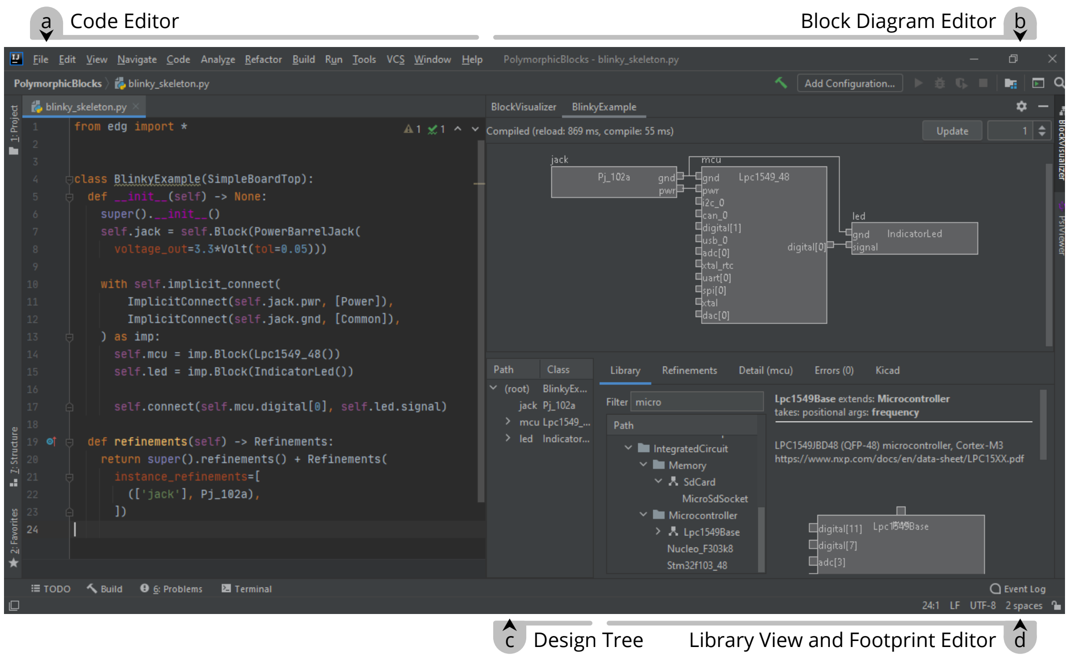
Figure 3 is an example part definition of a LF21215TMR digital magnetic field sensor device. Parts are defined similarly to top-level designs, and could have internal blocks and connections in the same way. In this case, however, there is only an associated footprint and pinning as shown on lines 15 - 23.

The `Lf21215tmr_Device` class also defines its exterior ports. Lines 4 - 13 instantiates the ports `vcc` of type `VoltageSink` (voltage input), `gnd` of type `Ground`, and `vout` of type `DigitalSource` (digital output) with defined voltage and current parameters. These forms the external interface of this device, abstracting away other elements as internal details. Lines 4 - 9 additionally show the component-side usage of implicit connection tags `Power` and `Common` on their respective ports.

4 SYSTEM DESCRIPTION

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

(1) Preserve the full power of the underlying HDL, while recognizing that board design tends to be highly iterative [18]. This precludes approaches that do not allow users to freely move between code and schematic workflows, such as single-shot code generation from schematics (like GUI builders) in which continued editing from the schematic view may be difficult after modification of the generated HDL. We further note that as there are likely programming constructs for which there is no (useful) corresponding graphical representation, the visualization will necessarily be a subset of the HDL.
(2) Interfaces based in current practices. While this means supporting a schematic-like interface for HDL editing, it also means boundaries on what be may less useful in a graphical


a Code Editor
Block Diagram Editor b
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
PolymorphicBlocks - blinky_skeleton.py
PolymorphicBlocks blinky_skeleton.py
Add Configuration...
BlockVisualizer BlinkyExample
Update
jack Pj_102a gnd pwr
mcu Lpc1549_48 gnd pwr i2c_0 can_0 digital[1] usb_0 adc[0] xtal_rtc uart[0] spi[0] xtal dac[0] digital[0]
led gnd IndicatorLed signal
Path Class
(root) BlinkyEx...
jack Pj_102a
mcu Lpc1549_...
led Indicator...
Library Refinements Detail (mcu) Errors (0) Kicad
Filter micro
Path
IntegratedCircuit
Memory
SdCard
MicroSdSocket
Microcontroller
Lpc1549Base
Nucleo_F303k8
Stm32f103_48
digital[11] digital[7] adc[3] Lpc1549Base
TODO Build 6: Problems Terminal
Event Log
24:1 LF UTF-8 2 spaces
c Design Tree
Library View and Footprint Editor d

```
from edg import *


class BlinkyExample(SimpleBoardTop):
  def __init__(self) -> None:
    super().__init__()
    self.jack = self.Block(PowerBarrelJack(
      voltage_out=3.3*Volt(tol=0.05)))

    with self.implicit_connect(
        ImplicitConnect(self.jack.pwr, [Power]),
        ImplicitConnect(self.jack.gnd, [Common]),
    ) as imp:
      self.mcu = imp.Block(Lpc1549_48())
      self.led = imp.Block(IndicatorLed())

      self.connect(self.mcu.digital[0], self.led.signal)

  def refinements(self) -> Refinements:
    return super().refinements() + Refinements(
      instance_refinements=[
        (['jack'], Pj_102a),
      ])
```

Compiled (reload: 869 ms, compile: 55 ms)

Lpc1549Base extends: **Microcontroller**
takes: positional args: **frequency**

LPC1549JBD48 (QFP-48) microcontroller, Cortex-M3
https://www.nxp.com/docs/en/data-sheet/LPC15XX.pdf

Figure 4: The IDE, implemented as an extension for IntelliJ, has several major views: (a) a code editor for the hardware description language, (b) an interactive block diagram editor that corresponds to the compiled HDL, (c) a tree view of the same compiled HDL, and (d) a library view for browsing available blocks.

editor. For instance, although a LabVIEW-style [10] interface may support programming constructs such as arithmetic and control flow as blocks in a purely graphical environment, code is overwhelmingly the more common way to express this kind of logic.

From these principles, we built a system that takes an "HDL first" approach, where the HDL code is the primary and authoritative design artifact. Tooling then provides support for understanding, navigating, and producing this code through an IDE metaphor built around a schematic-like view of the compiled output design as shown in Figure 4. More concretely, the block diagram view provides users with a visual representation of the HDL, and, combined with the library view, provides for schematic-like GUI edit actions that generate into corresponding HDL. However, we explicitly do not support the more code-like parameterization operations. Furthermore, as a prototype, we chose to focus on the basics and do not support syntactic sugar constructs like implicit connects or refactoring operations like delete and rename.

In the rest of this section, we will introduce our system in more detail through how a hypothetical user might build the example device in the prior section, starting from an empty top-level design.

4.1 Block Diagram Edit Actions

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

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

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

Simultaneously, the block diagram updates with the visual representation of the newly added block. To preserve flow with GUI operations, this is not the result of a full recompile, but is instead a *speculative effect*: the system assumes (regardless of context, such as if the caret was within an `if` block or `for` loop) that exactly one instance of the block would have been created. These blocks are indicated with a hatched fill, and additionally do not contain an internal implementation. However, their ports are valid, which allows connections to be made to them without re-compiling.

If our user instantiated a block with a required parameter, such as the `voltage_out` specification for a barrel jack, the unfilled keyword argument appears on the instantiation line. As with parameterization in general, the user must write this in HDL, here by

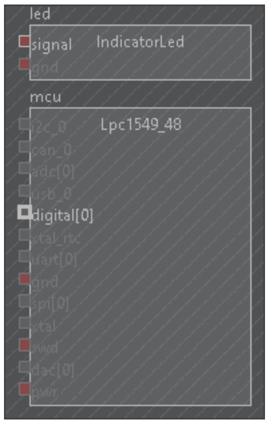


Figure 5: The connection interface, showing legal connections (by type) from the microcontroller's digital IO, with other pins dimmed out. Additionally, unconnected-but-required pins are highlighted in red, while the hatched fill indicates the preview status of both the LED and LPC1549 blocks inserted in the GUI as well as the modified status of the block they were inserted in.

providing an output voltage target of `3.3*Volt(tol=0.05)` as in line 4 of the example in Figure 2.

After repeating the instantiation flow with the LED, our user can then start connecting blocks. As shown in Figure 5, unconnected-but-required ports are marked with a red fill, and the user starts with one such port, the LED's signal input pin. Double-clicking the pin starts the connect tool, which dims the rest of the schematic except for ports that have compatible types, as also shown in Figure 5. The user chooses the only such available pin on the microcontroller, a digital IO line, then double-clicks again to commit the connection. For connects, names are optional and may be left blank.

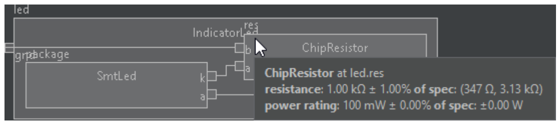
Similarly with the block insert action, the `connect` statement is inserted at the caret, and the connection is immediately though speculatively made on the block diagram. As required ports are connected, the red error fill also goes away.

Our user then initiates a recompile through the Update button, and a second or two later, the recompilation completes and diagram updates. The hatched fill disappears, but the diagram is otherwise unchanged as the speculative effects matched the HDL.

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

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

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



ChipResistor at led.res
resistance: 1.00 kΩ ± 1.00% **of spec:** (347 Ω, 3.13 kΩ)
power rating: 100 mW ± 0.00% **of spec:** ±0.00 W

Figure 6: Inspection of the resistor in the `IndicatorLed` block, showing both the actual resistance of the selected part and its specification or requested resistance, which in turn was derived from the connected voltage source.

4.2 Design-wide Edits

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

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

Because refinements are written in the top-level design's class and commonly as a single return statement like in lines 15-19 of Figure 2, there is no need for caret positioning. Since the code does not have a refinements function yet, the entire code block is generated, including the selected refinement. However, if a refinements block already existed, the selected refinement would be appended at the end of the list.

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

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

4.3 Inspection

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

Our user may be curious about the value of the resistor in this LED-resistor circuit, especially since it was automatically chosen. They mouse-over the part to show additional details, producing the popup shown in Figure 6 containing summary data of the object in question.

In general, summary views exist for common blocks (such as showing component values for resistors, capacitors, and inductors, or showing ratios for resistive dividers) and common connections (such as showing voltage thresholds between digital ports, or impedances between analog ports). While the same information is also available through a tree view of the entire design showing all

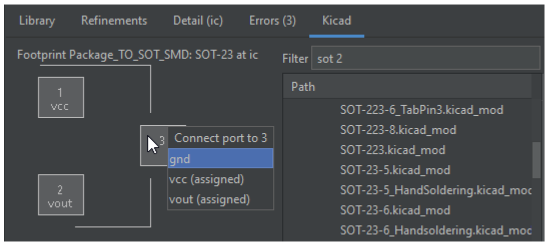


Figure 7: The IDE's footprint assignment and pinning interface. The footprint browser on the right shows the available KiCad footprints, while the footprint preview on the left allows assignment of mappings between footprint pads and block ports. All edit actions generate HDL.

parameters and constraints, this avoids information overload with a human-curated description at an appropriate level of detail.

4.4 Library Creation and Edits

Where existing libraries are insufficient, end-users need to create custom block definitions. To model the magnetic sensor component in Figure 3, our user starts by choosing a base class from the library browser, in this case the `FootprintBlock` class that allows an associated PCB footprint. The context menu provides an option to create a new subclass, which inserts the class template at the caret.

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

To associate the PCB footprint, our user switches over to the KiCad tab shown in Figure 7. Similarly to the library browser, they start by searching for an SOT-23 device using the filter box, then double-click the specific footprint from the list to insert the `footprint` statement at the caret.

The footprint itself also appears on the left side of the tab. From here, our user could double-click on a pad to bring up a list of connect-able ports, then choose one to update the `footprint` statement with the selected port to pad mapping. Similarly to block insertion and connect, effects of each action are shown speculatively on the footprint interface. The final footprint code looks like lines 15-23 of Figure 3.

5 SYSTEM IMPLEMENTATION

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

The IDE operates in part on Polymorphic Blocks' compiled designs which are defined in (and serialized with) Protocol Buffers. A Python stub "server" program handles HDL re-compilation requests from the IDE and communicates through gRPC, built on top of Protocol Buffers. This overall architecture is shown in Figure 8.

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

5.1 Code Analysis and Edits

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

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

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

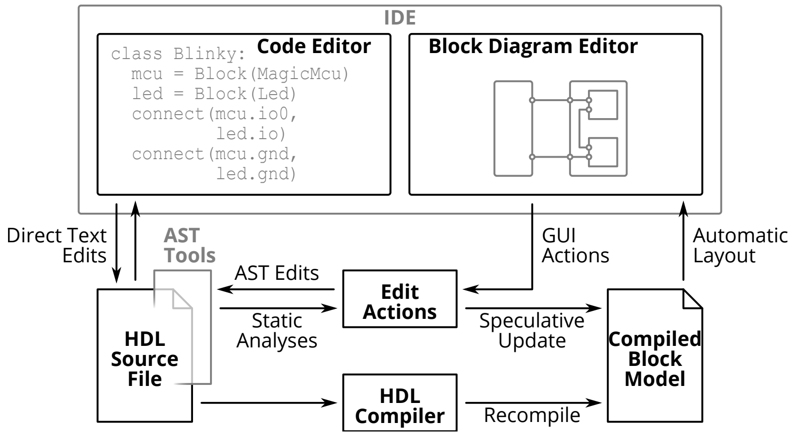
5.2 Block Diagram Visualization

The block diagram layout algorithm is functionally similar to the structure used by Polymorphic Blocks: start with the design as a set of hierarchy blocks, ports, and connections, infer the connection directions by port type (for example, a voltage source is an edge tail while a voltage sink is an edge head), simplify internal pseudo-blocks (like port type adapters) into direct connections, and replace high-fanout connections (for example, a voltage source connected to four voltage sinks) with tunnels or named nets. To simplify editing, we also remove disconnected array ports except for one to allocate a new port. This is structured as a series of transformations on a hierarchy block data structure, which is finally passed to ELK's [7] "layered" algorithm for layout to obtain the final positioning and size data of graphical elements.

6 USER STUDY: METHODOLOGY

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

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



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

Figure 8: Overall architecture of our system. HDL source code is compiled into a internal block model, which creates the user-facing block diagram through automatic layout. Edit actions and some static analyses are done through an AST-like view (IntelliJ's Project Structure Interface, or PSI) of the underlying HDL and speculatively update the compiled block model.

6.1 Participants

We recruited four participants through personal referrals, with the goal of sampling for range by having participants of different skill levels and motivations for PCB design. As a baseline, we required intermediate familiarity with PCB design and Python – supporting complete novices is currently out of scope. Two participants built PCBs primarily for personal and hobby reasons (one electrical engineering undergraduate, one professional software engineer), while the other two participants designed PCBs professionally (one graduate student researcher and one industry engineer). Two participants had prior experience with the Polymorphic Blocks HDL, and all participants had at least some familiarity with HDLs in general such as Verilog.

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

6.2 Structure

This study was conducted entirely by videoconference. Each participant accessed a fresh virtual machine (VM) running our IDE through the remote-desktop application X2go.

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

The first part of the study consisted of a tutorial session to familiarize participants with the IDE and HDL. Participants worked through a tutorial document which involved building a slightly expanded version of the `BlinkyExample` design from Figures 2 and 3. While the document hand-held participants through individual GUI edit actions, it also described the resulting code so participants could understand the generated HDL. The tutorial also described some common but code-only syntactic sugar constructs like implicit connects and included refactoring exercises using them.

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

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

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

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

Our user study procedure guide, the tutorial document, and the project specification document are included in the supplemental materials.

7 USER STUDY: RESULTS

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

7.1 Individual Flows

Overall, participants had diverse flows as a result of individual preferences, and while they all made use of the IDE in writing HDL, they differed in what features they used and preferred. Furthermore, all participants also directly edited textual HDL, such as to make use of syntactic sugar operations unavailable in IDE (all participants), to refactor and rearrange GUI-generated block instantiations and connects (also all participants), or as the primary way to produce HDL (P03 and P04). In the rest of this section, we report on individual flows and preferences.

P01 started by sketching out the system architecture block diagram, then instantiated the blocks with GUI actions and connected them with a mix of text editing and GUI actions. Overall, P01 felt the IDE approach was a good in-between for schematics and HDL, but still requires users to have a baseline competence with both code and circuits.

P02 similarly instantiated blocks and made connections from GUI actions, but as a more iterative process including refactoring inserted blocks. P02 specifically noted the helpfulness of the connect interface view filtering by legal connections, and also felt the IDE was a good coupling of code and diagrams.

P03, on the other hand, used text edits to instantiate some blocks and all connects, noting a preference for copy-paste coding. Uniquely, P03 made the connections to the display by writing a for-loop iterating through a list of ports. While P03 did not feel the block diagram generated edits were useful, the library browser was noted for its discoverability of parts and the block visualization was noted for its discoverability of connections.

P04 similarly also preferred text edits for block instantiation and connection, noting "not being a code snippets person". However, P04 also mentioned the tightly-integrated block diagram as an invaluable reference of what needed to be connected.

7.2 Edit Actions

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

Participants did criticize our interface as being clunky (P02), such as by requiring multiple selections (both a caret position and navigating to the edited block in the block diagram, P03) or being very sensitive to caret location (P04). In a broader sense, both P01 and P03 mentioned preserving flow by staying in one interface instead of jumping between text and block diagram. However, these may be more issues of our specific prototype implementation, and it may be possible that tweaks to the interface specifics could produce a more usable flow.

On the other hand, P04 called out the footprint and pinning interface as something done well, because it directly matches the visual presentation often provided by component documentation. Participants additionally suggested different interfaces for edit actions, such as drag-and-drop for block instantiation (P01), click-and-drag for making connections (P04), or improving the existing IDE autocomplete with awareness of the HDL (for example, presenting only connectable ports; P03).

As a completely different interface, P01 also suggested an import flow from existing schematic tools, both to use existing libraries written in mainstream tools, and to support users who are more familiar and comfortable working with mainstream tools.

7.3 Block Diagram Visualization

For the automatically generated block diagrams, the overall consensus was that it was very usable for writing HDL including reasonable adherence to convention, but still had many rough edges. All participants suggested more usage of symbols, such as ground symbols for the ground connections or part symbols instead of the simple rectangles for blocks. P03 additionally suggested symbols as a way to manage complexity when zoomed out: dense text could fade out and the entire block could be replaced with a symbol.

P01, P02, P03 all discussed ideas for manual layout constraints, such as grouping blocks together, but also acknowledged it is a hard problem without clear solutions. P04, on the other hand, felt that a tool for creating presentation-grade diagrams may be out of scope here.

Finally, P01 and P02 mentioned layout considerations: P01's schematic capture flow takes into account rough layout floorplanning when placing symbols, while P02 felt disconnected from the physical (footprint) view though also believed that more stuff on the block diagram could be cluttering.

7.4 Refresh and Speculative Effects

Participants had varied opinions on the manual (user-initiated) recompilation and block diagram update. At one extreme, P01 preferred continuous compilation to minimize the feedback loop between HDL edits and visual presentation. On the other hand, P03 felt that the current user-initiated scheme makes sense, crucially preserving diagram stability as text is being edited. P04 was in-between, feeling that automatic updates could save a few keystrokes, but the system should be smart about detecting when the user is at a stopping point.

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

8 DISCUSSION

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

8.1 Study Limitations

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

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

8.2 Overall Takeaways

Overall, we believe a major generalizable takeaway is that graphical editing operations do not need to cover all conceivable code edits – here, participants were able to effectively blend use of GUI tooling with textual HDL edits for operations not supported by the GUI. Consistent with prior work [16], the simplest tools of visualization and library browsing were the ones that were most consistently used. Beyond PCB design, tools working on similar concepts may find it valuable to focus on supporting a few common workflows well before being bogged down by how to support trickier operations in a GUI.

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

Furthermore, imperfect techniques that improve responsiveness, like speculative effects and caching, can provide the smooth interface expected of direct manipulation system even with slower compilers. Despite both techniques having unsupported edge cases, which may be fundamentally difficult to resolve especially with a dynamic language like Python, they seemed to have worked well enough in practice for our (albeit somewhat limited) user study. More generally, it may be useful for future work to explore this time-accuracy trade-off in more detail, as well as investigate compromise strategies that provide more accurate results as they become available.

8.3 Flow for Edit Actions

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

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

8.4 Layout

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

8.5 Performant Recompilation

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

9 CONCLUSION

While recent HDL approaches for PCB design have been promising in terms of design power, the textual code interface is a departure from currently mainstream schematics. In this work, we implemented and obtained exploratory user feedback for an IDE approach that aims to bridge the familiar schematics with the power and expressiveness of an HDL. Within the larger picture of programming tools, and in contrast to similar work on GUI builders and structure editors, our approach emphasizes flexibility in working with and moving between code and block diagram views.

We hope that flexible tooling like this can be be a best-of-both-worlds approach to building hardware with HDLs. We believe that improved tooling can play a crucial part in encouraging adoption of HDLs, and ultimately in making easy and efficient hardware design accessible to everyone.

ACKNOWLEDGMENTS

We would like to thank our user study participants for their time and feedback, as well as the anonymous reviews for their suggestions.

This work was supported in part by DARPA contract FA8750-20-C-0156 (SDCPS); the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and the Paul and Judy Gray Alumni Presidential Chair in Engineering Excellence. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

[1] Altium. 2020. *Altium Designer*. 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 (Qu&#233;bec City, QC, Canada) *(UIST '17)*. ACM, New York, NY, USA, 331–342. https://doi.org/10.1145/3126594.3126637

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

[4] 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* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 677–686. https://doi.org/10.1145/2984511.2984566

[5] EDASolver. 2020. *EDASolver - Automatic component selection and pin matching.* https://edasolver.com

[6] Jonathan Edwards. 2004. Example Centric Programming. *SIGPLAN Not.* 39, 12 (Dec. 2004), 84–91. https://doi.org/10.1145/1052883.1052894

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

[8] Gumstix. 2018. *Geppetto.* www.gumstix.com/geppetto/

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

[10] National Instruments. 2020. *LabVIEW.* http://www.ni.com/labview

[11] 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. https://doi.org/10.1109/ICCAD.2017.8203780

[12] 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* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3313831.3376761

[13] KiCad. 2020. *KiCad EDA.* http://kicad-pcb.org/

[14] 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* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300407

[15] 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* (Cambridge, United Kingdom) *(TEI '09)*. Association for Computing Machinery, New York, NY, USA, 351–358. https://doi.org/10.1145/1517664.1517735

[16] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The Road to Live Programming: Insights from the Practice. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1090–1101. https://doi.org/10.1145/3180155.3180200

[17] Richard Lin, Rohit Ramesh, Connie Chi, Nikhil Jain, Ryan Nuqui, Prabal Dutta, and Björn Hartmann. 2020. Polymorphic Blocks: Unifying High-Level Specification and Low-Level Control for Circuit Board Design. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/3379337.3415860

[18] 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* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, Article 283, 13 pages. https://doi.org/10.1145/3290605.3300513

[19] 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* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, Article 403, 13 pages. https://doi.org/10.1145/3290605.3300633

[20] Andrew J. Matthews. 1977. A Human Engineered PCB Design System. In *Proceedings of the 14th Design Automation Conference (DAC '77)*. IEEE Press, 182–186.

[21] Sean McDirmid. 2013. Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 53–62. https://doi.org/10.1145/2509578.2509585

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

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

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

[25] Devon J. Merrill, Jorge Garza, and Steven Swanson. 2019. Echidna: Mixed-Domain Computational Implementation via Decision Trees. In *Proceedings of the ACM Symposium on Computational Fabrication* (Pittsburgh, Pennsylvania) *(SCF '19)*. Association for Computing Machinery, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3328939.3329004

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

[27] 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* (Cambridge, Massachusetts) *(SCF '17)*. ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3083157.3083159

[28] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. 2016. How Live Are Live Programming Systems? Benchmarking the Response Times of Live Programming Environments. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop* (Rome, Italy) *(PX/16)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2984380.2984381

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

[30] Haven Skinner, Rafael Trapani Possignolo, Sheng-Hong Wang, and Jose Renau. 2020. LiveSim: A Fast Hot Reload Simulator for HDLs. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 126–135. https://doi.org/10.1109/ISPASS48437.2020.00028

[31] Sparkfun. 2020. *À La Carte.* https://alc.sparkfun.com/

[32] 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* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 321–330. https://doi.org/10.1145/3126594.3126618

[33] 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* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3290605.3300278

[34] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*.

[35] Sheng-Hong Wang, Rafael Trapani Possignolo, Haven Blake Skinner, and Jose Renau. 2020. LiveHD: A Productive Live Hardware Development Flow. *IEEE Micro* 40, 4 (2020), 67–75. https://doi.org/10.1109/MM.2020.2996508

[36] 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. 2017. CurrentViz: Sensing and Visualizing Electric Current Flows of Breadboarded Circuits. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 343–349. https://doi.org/10.1145/3126594.3126646

[37] 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. 2017. 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* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 311–319. https://doi.org/10.1145/3126594.3126634

[38] Apostolos V. Zarras, Georgios Mamalis, Aggelos Papamichail, Panagiotis Kollias, and Panos Vassiliadis. 2018. And the Tool Created a GUI That Was Impure and Without Form: Anti-Patterns in Automatically Generated GUIs. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLoP '18)*. Association for Computing Machinery, New York, NY, USA, Article 24, 8 pages. https://doi.org/10.1145/3282308.3282333

[39] Junyi Zhu, Yunyi Zhu, Jiaming Cui, Leon Cheng, Jackson Snowden, Mark Chounlakone, Michael Wessely, and Stefanie Mueller. 2020. MorphSensor: A 3D Electronic Design Tool for Reforming Sensor Modules. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '20)*. Association for Computing Machinery, New York, NY, USA, 541–553. https://doi.org/10.1145/3379337.3415898