Echidna: Programmable Schematics to Simplify PCB Design

Shengye Wang, Devon Merrill, Chris Taylor, Steven Swanson
University of California, San Diego
9500 Gilman Dr. MC 0404
La Jolla, California
{shengye, djmerrill, c4nelson, swanson}@ucsd.edu

ABSTRACT

In this paper we introduce Echidna, a hybrid schematic/ text-based language for describing PCB circuit schematics. Echidna allows designers to use high-level programming constructs to describe schematics, supports modular, reusable design components with well-defined interfaces, and provides for complex parameterization of those modules. Echidna deeply integrates a high-level programming language into a schematic-based design flow. The designer can describe schematics in code, as a schematic, or as a seamless combination of the two. We demonstrate its usefulness with several case studies.

1. INTRODUCTION

Recent years have seen great progress in raising the abstraction level of designing electronic systems. The rise of user-friendly platforms such as Arduino [1] and Raspberry Pi [2] have opened up embedded systems to a huge audience. High-level synthesis tools have made it easier than ever to program FPGAs and design integrated circuits. These tools hide complexities through familiar, programming-based interfaces and allow novice designers to produce reliable designs quickly.

The design methodology for printed circuit boards (PCBs) has lagged behind these other areas, however. Most designers use schematic capture to specify the functionality a PCB implements, and while schematics offer an intuitive means for describing electrical connections, they suffer from significant draw backs. To name just three: Even modestly-sized schematics are hard to navigate, verify, and modify; the lack of sophisticated tools for parameterizing design modules, such as IP blocks, limits designers' ability to reuse or generalize frequently-used components; and there is no general mechanism for specifying (or verifying) complex relationships between different components.

As a result, designing circuit schematics is complex, time consuming, and error-prone. This discourages all but the most motivated novice designers, students, and "makers" from building custom PCBs, and wastes the time of experienced circuit designers, since they must re-implement similar circuits over and over.

This paper presents Echidna, a hybrid schematic/text-based language for specifying PCB circuits. Echidna supports hierarchical, reusable design by allowing the designer to break the circuit into parameterized modules that have well-defined interfaces. Designers can specify the structure of their circuits using schematics and use Python code to specify and enforce complex relationships between components, or they can specify circuits in pure Python. These *programmable schematics* enable the construction of libraries of reusable components such as power supplies, amplifiers, and microcontrollers that can significantly reduce design complexity.

To evaluate Echidna we use it to implement a range of circuits and circuit components including an ARM-FPGA hybrid system, an Arduino-class microcontroller, a family of multi-rotor aircraft (i.e., "quadcopters"), and a library of discrete components (i.e., resistors, capacitors, etc.) that automates part selection based on circuit requirements. We find that adding a high-level programming language makes schematics more expressive and flexible while improving their readability. We also find that a library of Echidna modules can replace up to 91% of the discrete components in some designs.

The remainder of this paper is organized as follows. Section 2 discusses current PCB circuit design methodology and its shortcomings. Section 3 provide an overview of Echidna. Section 4 discusses implementation details and trade-offs of Echidna Section 5 provides case studies of using Echidna. Section 6 places Echidna in context with related tools and projects, and Section 7 concludes.

2. SCHEMATIC SHORTCOMINGS

Schematics excel at specifying the connections and structure of electrical circuits. However, they fall short in expressing the complex relationships between components which are often the most challenging aspects of a design. They also do not provide a robust mechanism to parameterize or reuse portions of a circuit.

Figure 1 illustrates the kinds of relationships PCB designers must enforce between components and the challenges they create. The figure reproduces three partial schematics for the datasheet for a lithium-ion polymer (LiPo) battery charger [3]. The data sheet also contains extensive documentation on how the designer should select the discrete components that surround the central integrated circuit (IC). For instance:

- Resistor R_{ISET} sets the charging current, I_{CH}, according to this formula: $\mathrm{I_{CH}} = 1800\ V/\mathrm{R_{ISET}}$.
- $\mathrm{R_{ISET}}$ should be a precisely calibrated (i.e., 1%) metal film resistor.
- Resistor $\mathrm{R_x}$ sets the charging voltage, $\mathrm{V_{bat}}$ to $4.2 + 3.04 \times 10^{-6} * \mathrm{R_x}$.

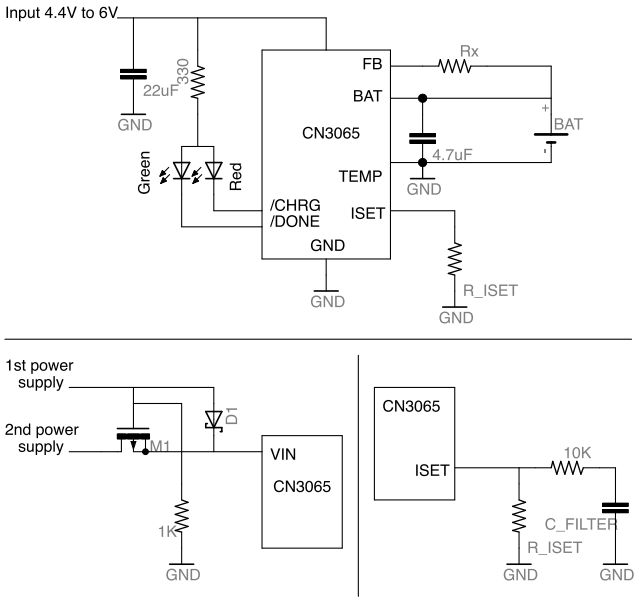


Figure 1: The reference schematics from the datasheet for a lithium polymer battery charger [3] provide only partial documentation for configuring the device and leave the designer to perform tedious and error-prone calculations.

- The capacitive load on the ISET pin can limit the acceptable value of R_{ISET}: $R_{ISET} < 1/(6.28 \times 2 \times 10^5 \times C)$. An RC filter (the 10 K resistor and C_{filter} it lower right) can shield the R_{ISET} pin from this interference.

The values of other components (e.g., current carrying capabilities and M1 and D1) are not discussed. Likewise, the datasheet provides no guidance on the value of C_{filter}.

Faced with this fragmentary and incomplete documentation, the designer is left with a series of puzzles they must solve. While none of them are particularly difficult for an experienced designer, they all take time and open the door for errors. For a novice engineer, they present a serious obstacle.

Figure 2 illustrates how the lack of modularity and support for design reuse complicate schematic design. The figure shows the schematic for a simple Arduino-based, remote controlled "quadcopter." The circuit comprises several mostly-independent components: the microcontroller, the accelerometer and gyroscope, the power supply, the four motor controllers, and various programming and debugging interfaces.

Several aspects of this schematic make it more difficult than necessary to create, modify, and maintain. None of the components in the design are novel – many other designs contain similar or identical circuits. Indeed, several parts of this schematic were copied from open-source hardware designs in an ad hoc manner, since there is no way to cleanly reuse parts of other designs. Others were transcribed from data sheet schematics similar to those in Figure 1. In both cases, the design inherits all the bugs from the original, is subject to errors during manual copying, and does not benefit from any improvements (or bug fixes) made to the originals.

Second, the design contains four identical motor drivers. If the designer wishes to make a change to this part of the design, she will need to apply those changes four times, increasing the opportunity for error.

Echidna attempts to solve these problems by extending schematic-based PCB circuit design to support modularity, parameterization, and reuse and to allow the designer to specify complex relationships between components.

3. Echidna

Echidna is a hybrid schematic/text-based design representation for describing PCB circuits that solves the problems described in Section 2. Echidna lets designers use schematics to specify a circuit's structure and Python code to specify complex relationships between components. It encourages and enables design reuse by providing first-class support for hierarchical design, modularity, and parameterized, programmable modules.

Below we describe the functionality and key features of Echidna with two simple examples: An LED circuit and a band-pass filter.

3.1 Programmable Schematics

The core building block of a Echidna design is a *module*. Echidna modules are self-contained, reusable, programmable circuit fragments. The designer can describe a module using schematic, the Echidna Python library, or a combination of the two. Modules have well-defined interfaces that allow designers to assemble them into more complex designs without needing to understand their internal implementation.

Figures 3 and 4 illustrate the correspondence between schematics and code in Echidna. Both figures describe `MyLED`, a module that contains an LED and a current-limiting resistor. In the schematic version (Figure 3) the code in pink gives the module's definition. This includes the module's name, its parameters, and its electrical interfaces. Parameters let the designer control how Echidna will instantiate the module. `MyLED` has one parameter named `voltage` that specifies the supply voltage the circuit will use. The default value of `voltage` is 5 V. The module definition also describes its electrical interfaces. `MyLED` has two electrical interfaces: `POS` and `NEG` connect to the nets `NET_P` and `NET_N` in the circuit.

The code in blue configures the module according to the value of `voltage`. In particular, the code calculates the appropriate resistance for the resistor, depending on `voltage` and given the voltage drop of the LED (2.2 V) and its typical current draw (40 mA). More generally, designers can use arbitrary Python code to specify the module's behavior.

Figure 4 describes the same circuit using the Echidna Python library. The code in the figure illustrates three key features of the Echidna library. First, Echidna modules correspond to subclasses of the Python class `Echidna.Module`. Second, the `instantiate_schematic()` builds Echidna modules and connects with nets using `add_part()` and `add_net()`. And `configure()` sets the resistor value. The Echidna library also includes facilities for removing components and nets from the module. For schematic-based modules (e.g., Figure 3), Echidna generates this code internally from the schematic.

3.2 Hierarchical Design and Parameterization

Echidna describes a circuit as a tree of modules. The

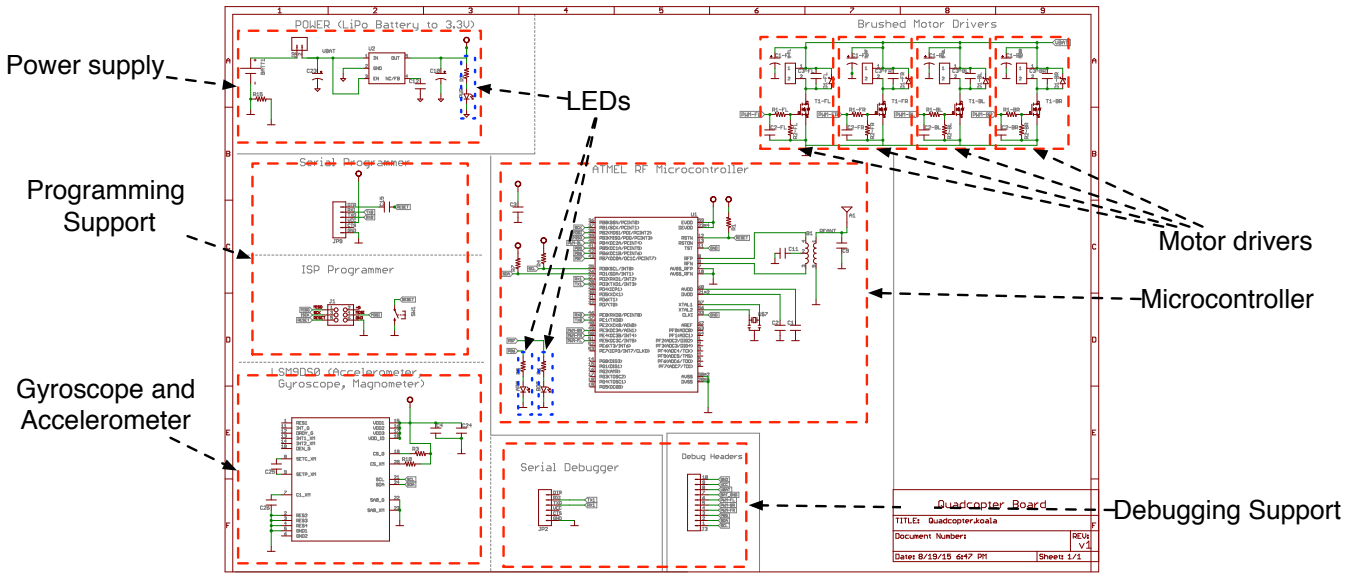
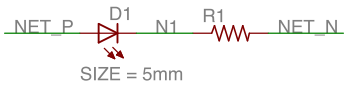


Figure 2: This schematic for a remote controlled "quadcopter" illustrates how traditional schematics excel at describing the structure of as circuit. However, they cannot express complex relationships between components or break the circuit into independent, parameterizable modules.

```
module_name = "MyLED"
parameters  = {"voltage":5.0}
interfaces  = {"POS":"NET_P", "NEG":"NET_N"}
```




```
def configure(self):
  self.R1.value = (self.voltage - 2.2) / 0.04
```

Figure 3: Echidna allows the designer to parameterized circuits and define relationships between components. In this case, the code in blue sets the resistance of `R1` based on the `voltage` parameter.

```
import Echidna
from GenericParts import LED, Resistor

class MyLED(Echidna.Module):

  def __init__(self):
    super(MyLED, self).__init__()
    self.add_interface(["POS", "NEG"])
    self.add_parameter({"voltage":5.0})

  def instantiate_schematic(self):
    self.add_part("D1", LED())
    self.D1.SIZE = "5mm"
    self.add_part("R1", Resistor())
    self.add_net("NET_P").connect(self.POS, self.D1.anode)
    self.add_net("N1").connect(self.D1.cathode, self.R1.t1)
    self.add_net("NET_N").connect(self.NEG, self.R2.t2)

  def configure(self):
    self.R1.value = (self.voltage - 2.2) / 0.04
```

Figure 4: The Echidna Python library can create circuits without a schematic. This code is equivalent to the programmable schematic in Figure 3.

module at the root of the tree represents the entire design. Parent modules connect their children using nets and can control the behavior of a child module by setting the child's parameters (e.g., `voltage` in `MyLED`).

When Echidna processes a design, it first instantiates the root module, executing `instantiate_schematic()` to translate the schematic into Echidna objects and `configure()` to execute any code embedded in the schematic. Then, it recursively instantiates each of its children using the parameters those two functions set. As it traverses the tree, Echidna builds an representation of the entire circuit. When it is done, it generates a schematic file that the designer can load into an existing PCB design tool flow to complete the design process. Currently, Echidna uses the Eagle [4] file format to read and write schematics.

Figures 5 and 6 illustrate how a designer can combine Echidna's module hierarchy with programmability. Figure 5 shows a first-order low-pass filter module. The parameters `gain` and `freq` specify the gain and cut-off frequency of the filter, respectively, and the module's code uses them to compute the values for the resistor and capacitor. Figure 6 combines the low pass filter with a high pass filter (schematic not shown) to build a band pass filter using a combination of schematic and code. Figure 7 builds the same bandpass filter using pure Python code. The hybrid version is clearer because it uses schematic elements to specify structure and code to perform the configuration. The code-only version is more difficult to follow because specifying connectivity in code is cumbersome.

This example also illustrates the benefits of modular and hierarchical design. First, modules allow the designer to decompose their design into smaller, independent, more manageable pieces. This makes the design easier to understand, especially if another designer needs to modify the design in the future.

Second, well-engineered module interfaces can facilitate design reuse and hide implementation details from the de-

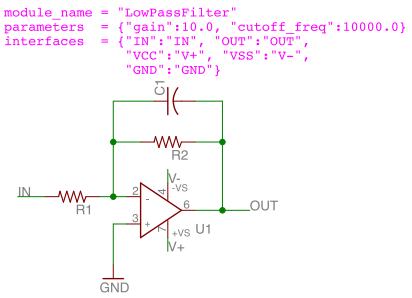


Figure 5: The Echidna schematic for a first-order, low-pass filter uses Python code to calculate circuit values based on the `gain` and `freq` parameters.

```
import CircuitsByCode
import AnalogFilters

class BandPassFilter(CircuitsByCode.Module):

  def __init__(self):
    super(BandPassFilter, self).__init__()
    self.add_interface(["IN", "OUT", "VCC", "VSS", "GND"])
    self.add_parameter({"gain":10.0, "cutoff_low":10000.0,
                                     "cutoff_high":20000.0})

  def configure(self):
    self.add_part("M1", AnalogFilters.LowPassFilter())
    self.add_part("M2", AnalogFilters.HighPassFilter())
    self.M1.cutoff_freq = self.cutoff_low
    self.M2.cutoff_freq = self.cutoff_high
    self.M1.gain = self.M2.gain = self.gain ** 0.5
    self.net("IN").connect([self.IN, self.M1.IN])
    self.net("N1").connect([self.M1.OUT, self.M2.IN])
    self.net("OUT").connect([self.M2.OUT, self.OUT])
    self.net("VCC").connect([self.VCC, self.M1.VCC, self.M2.VCC])
    self.net("VSS").connect([self.VSS, self.M1.VSS, self.M2.VSS])
    self.net("GND").connect([self.GND, self.M1.GND, self.M2.GND])
```

Figure 7: Example code of combining a low-pass filter and a high-pass filter to get a band-pass filter.

signer. For instance, to build a higher-quality band-pass filter, the designer could reuse the schematic in Figure 6, and just replace the first-order filter modules with a second- or third-order filter modules.

Third, modularity allows the construction of a library of reusable, parameterized modules that designers can use to quickly assemble complex designs. The high-pass and low-pass filters are good examples, but they are only the beginning. Section 5 illustrates how such a library can reduce design complexity across many designs.

4. Echidna IMPLEMENTATION

Echidna's implementation includes a set of Python classes that represent Echidna modules and the connections between them as well as a suite of tools that operate on those data structures.

Below, we describe the key Echidna classes and the Echidna tool flow.

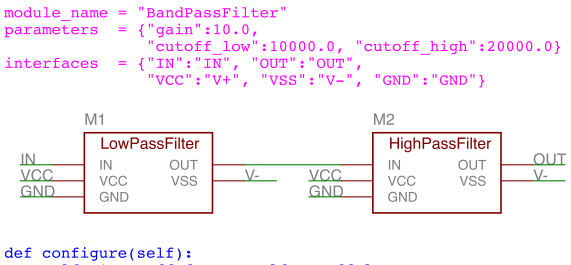


Figure 6: Example schematic of combining a low-pass filter and a high-pass filter to get a band-pass filter.

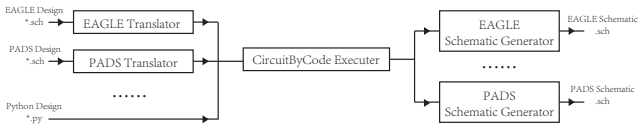


Figure 8: Echidna toolchain contains a tool-independent executer, as well as tool-dependent translator and schematic generator that converts between Echidna Python representation and schematic used for schematic capturing tools.

4.1 Echidna Execution Model

Echidna represents a design as a tree of modules that contain both schematic fragments and code that can modify and configure those fragments to meet the needs of the design.

After the designer has assembled the modules her design requires, Echidna processes the design convert the hierarchical design into a complete schematic. This processing proceeds in three stages.

Translation.

Echidna converts the hand-drawn schematics into Python code that produces the equivalent Python objects. This code constitutes the module's `instantiate_schematic()` method.

Execution.

To assemble the complete design, Echidna performs a depth-first traversal of the module tree starting at its root. Algorithm 1 contains pseudo-code for the Echidna execution model. The execution stage effectively executes `finalize()` on the root of the module tree.

The execution model involves four methods that each module provides. The first is the module's constructor. The constructor for a module creates the module's parameters and interfaces, but does not create the module's sub-modules or the connections between them. A module in this states is *instantiated*.

The second function, `instantiate_schematic()`, instantiates the module's sub-modules included in the module's schematic and makes the connections that the schematic specifies. The translation phase automatically creates this

Algorithm 1 The Echidna execution model

```
1: procedure FINALIZE(CurrentModule)
2:     CurrentModule.instantiate_schematic()
3:     CurrentModule.configure()
4:     for each submodule m do
5:         if not m.isFinalized() then
6:             FINALIZE(m)
7:         end if
8:     end for
9: end procedure
```

method, so Echidna developers do not need to implement or modify it.

The third function, `configure()`, is the most important. It is responsible for configuring the module to behave as the designer intends it to. Depending on the needs of the module, this method can instantiate modules, change parameters on modules, connect modules to one another, delete module, etc.

The last function is `finalize()` (shown in Algorithm 1). `finalize()` converts an instantiated module into a *finalized* module. By definition, all of the sub-modules in a finalized module are also finalized. Finalized modules are also immutable.

By default, `finalize()` finalizes all of the module's sub-modules after configuration. However, `configure()` can call `finalize()` explicitly. This is necessary instances where one sub-module's configuration depends on aspects of another sub-module that are not known until after the sub-module is finalized. The example in Section 5 will illustrate how this can become necessary.

Schematic generation.

Echidna flattens the finalized tree of modules into a single schematic so that the designer can use other design tools to complete the design (e.g., board layout and manufacturing). To the extent possible, the schematic generator reuses the hand-drawn schematics that are part of the modules. However, `configure()` may have added or removed sub-modules and/or connections. The generator uses some simple heuristics to layout new portions of the schematic.

4.2 Echidna Modules

Echidna modules are the basic building blocks of the Echidna designs. A module can include sub-modules and the electrical connections between (i.e., a schematic) along with Python code (i.e., in its `configure()` method) that can add and remove sub-modules, modify the connections between them, and configure sub-modules as needed.

During execution, modules move through a well-defined two-stage life-cycle. First, the module is *instantiated* as a child of its parent module. The parent module can connect instantiated modules together and it can set set parameters on instantiated modules.

The interface for an instantiated module includes electrical *interfaces* and *parameters*. Electrical interfaces represent the electrical connections the module can make to other modules. Each interface has a unique name. The schematic for a Echidna module defines electrical connections between sub-modules by connecting the interfaces on two modules.

Parameters are input data values that affect how a module will configure itself. Parameters can be arbitrary Python objects and are available to `configure()`. For example, an generic power supply module might take parameters to control its output voltage and maximum output current. The LED example in Figure 3 takes a voltage parameter.

The `configure()` can set *attributes* on the module. Attributes are only available to the parent module after finalization and they are immutable once the module is finalized. Attributes can be arbitrary Python objects.

Electrical interfaces, parameters, and attributes provide a flexible interface to modules, while hiding most details of their implementation. In particular, the only information about a finalized module that is visible to the parent are the attribute values. This provides an effective information hiding mechanism that makes it easier to replace one sub-module with another.

Well-engineered modules use these three mechanisms to facilitate design partitioning, enable module reuse, and hide implementation details from the designer. For example, a generic 5 V power regulation module might have three electrical interfaces: power input, power output, and ground. It might takes a minimal output current requirement and safe operating input range as parameters. After configuring itself, it will report the actual operating range, maximum current, and output ripple as attributes. A designer using such a module will not need to worry about the internal implementation of the power supply and, if multiple power supply modules (e.g., one optimized for cost and one suitable for adverse environmental conditions) provided the same interface, replacing one with the other would be very easy.

After the `finalize` process, Echidna will exploit its extension mechanism perform customized tasks. At the end, Echidna executer will dump the result to Echidna Python representation.

4.3 Echidna and Other PCB Design Tools

Echidna does focuses on simplifying the creation of PCB schematics. It is intended to leverage existing design tools for schematic capture, creation and authoring of PCB part libraries, board design and layout, and CAM file generation.

The only portions of Echidna that depend on these other tools are the translation and schematic synthesis stages. The execution stage and the code embedded in the design (i.e., `configure()`) all operate on tool-agnostic data structures.

Echidna currently inter-operates with Cadsoft EAGLE [4]. EAGLE is freely available, has an active user community, and has an open, XML-based file format that facilitates easy translation and schematic creation. However, there is nothing EAGLE-specific in Echidna's design and, if we had access to the documentation for their file formats Echidna would work just as well with PADS [5], Altium [6], OrCAD [7]. Echidna uses the Swoop [8] library to read and write EAGLE files.

4.4 Echidna and Python

The Echidna translator can convert both schematics and PCB libraries into Echidna Python code. By default, a schematic corresponds to a single Echidna module with the same name. However, the user can define multiple modules in a single schematic by placing the modules on separate schematic sheets and providing an interface declaration (as we saw in Figure 3) on each sheet.

For PCB libraries, Echidna creates a Python library that contains modules that correspond to each of the parts the PCB library. Which libraries Echidna uses depends on the

underlying PCB tools it is interacting with. For EAGLE, Echidna searches the paths in EAGLE's "libraries" configuration variable and automatically converts the libraries (i.e., ".lbr" files) into Python libraries.

This convention allows designers to refer to parts in the library using familiar Python syntax. For example, in Figure 7 the module's `configure()` method creates an instance of `AnalogFilters.LowPassFilter` that corresponds to `LowPassFilter` device in `AnalogFilters.lbr` (the EAGLE library).

4.5 Extending Echidna

Echidna provides a simple extension mechanism that allows designers to add new features. The extension mechanism uses the visitor design pattern. To use it, the designer provide call back functions that the framework invokes for each module during a depth-first traversal of the module tree.

Echidna includes two extensions as examples. The first generates a bill of materials (BOM) by collecting the names and attributes of all leaf modules (which correspond to physical electrical components).

The second is an I2C validation pass that performs checks to ensure that I2C busses are wired correctly (i.e., that all I2C devices connect the `SCL` and `SDA` lines to the correct interfaces and that there is only one master per bus). To participate, modules with I2C connections must provide a function called `report_i2c()` that provides information about the module's I2C connection, including whether it is a master or slave device.

5. CASE STUDIES AND EXAMPLES

We have found Echidna to be useful in designing schematics for many different PCBs and that it makes the PCB design process easier and less error prone. This section describes five different use cases for Echidna that illustrate its power and flexibility. The designs include a library of modules to automate part selection, a generic, flexible LED module, a configurable multi-rotor design, a re-implementation of the Arduino Uno in Echidna, and a complex design that combines an FPGA with an ARM-based microcontroller.

5.1 Automatic Part Selection

Selecting which components to use in a PCB can confront the designer with a vast number of choices. For instance, if a design calls for a 10 KOhm resistor or a 22 μF capacitor Digikey provides 473 and 1467 options, respectively. Alternatives include package types, different sizes, and temperature ranges.

Echidna provides a library of modules, called `GenericParts`, that represent common electrical components, including resistors, capacitors, diodes, LEDs, N-type MOSFETs, P-type MOSFETs, and resonators. The module for each type of device has parameters that let the designer specify requirements for a particular instance of the module. For instance, the resistor's parameters include resistance, power dissipation, how precise the resistance value needs to be, and package type (e.g., through-hole, 0805, or 0603). The designer can also specify upper and/or lower bounds for the parameters and specify part selection priorities. For instance, the designer can specify that the module should select a resistor with a resistance within 10% of 332.4 Ohms with a preference for cheaper, through-hole components.

Electrical Interfaces	
`PWR`	Power source
`GND`	Ground
`CTRL`	Control pin
Parameters	
`color`	LED color
`size`	LED size
`brightness`	LED brightness
`ctrl_vh`	`CTRL` pin high voltage
`ctrl_polarity`	`CTRL` pin polarity
`ctrl_max_i`	`CTRL` pin max current
`pwr_v`	`PWR` pin voltage
`pwr_max_i`	`PWR` pin max current

Table 1: Interface and parameters of `LED`.

When a design instantiates a module from `GenericParts`, the code in each module searches through a database of available parts and finds the best match based on the requirements and priorities. That part will appear in the final schematic. The module also stores additional information about the part (e.g., manufacture, part number, distributor, price, and technical information about the part) it chose and exposes it as attributes.

`GenericParts` leverages several of Echidna's advanced features like parameterization and feedback, and relies on several of Python's standard libraries to load and query its part databases. It comprises 7300 lines of Python.

We use `GenericParts` in all of the designs we build with Echidna, and we consider it part of the Echidna "standard library."

5.2 A Generic LED Module

LEDs are common on PCBs and different designs require LEDs in different colors and of different brightnesses. It is often possible to drive the LED directly from a microcontroller's digital IO pin, but if the current requirement of the LED exceeds the capability of the IO pin, a more complex design is necessary.

We built a generic, reusable LED module called `LED` that automatically shifts from one design to the other as needed. `LED` lets the designer specify the color and brightness of the LED along with information about the available power supplies. The schematic then selects the correct implementation strategy and uses `GenericParts` to select the necessary components.

Table 1 summarizes `LED`'s electrical interfaces and parameters. In its `configure()` function, `LED` sets the `color`, `size`, and `brightness` parameters on the LED and finalizes it, triggering `GenericParts` part selection code. Once finalized, the LED's attributes provide its maximum current draw and its forward voltage drop.

Based on those values, the module determines whether the `CTRL` pin can provide sufficient power by examining `ctrl_vh` and `ctrl_max_i`. If `CTRL` can provide enough power, `LED` computes the resistance necessary to limit the current through the LED and sets the parameters on the resistor, and `GenericParts` will choose a suitable resistor. It then connects the LED to the resistor and the `CTRL` net.

If `CTRL` cannot power the LED, `LED` examines `pwr_v` and `pwr_max_i` to see if `PWR` will work instead. If it will, `LED` instantiates a P-MOSFET or N-MOSFET (depending on

	Micro	Nano	Pro	Uno	Mega	Due	Gemma	LilyPad
ICSP Header	1	1	1	2	2	2		
DC Input Socket			4	1	1	1	1	
LED Circuit	3	1	2	4	4	6	2	4
Oscillator	1	1	1	2	2	3		1
Power Convertor	2	1	1	2	2	1	1	1
USB Interface	1	1		1	1	2	1	1
Arduino Shield Connector	1	1	1	1	1	1		
Microcontroller	1	1	1	1	1	1	1	1
AVR USB Programmer				1	1	1		
Other Modules	1	1	2	2	8		1	
Reusbale Module Coverage (%)	90	83	69	91	91	90	83	88
Part Count Reduction (%)	76	77	52	81	84	82	68	71

Table 2: There are many opportunities for reuse across Arduino designs. The high values for "Reusable Module Coverage" and "Part Count Reduction" show that re-implementing these designs in Echidna would significantly reduce design complexity.

the value of `ctrl_polarity`), attaches `CTRL` to the gate and uses the MOSFET to power the LED. If `PWR` is insufficient to power the LED, the module signals failure and notifies the designer.

`LED`'s implementation requires just 77 lines of Python and, for most designs, makes adding an LED as easy as specifying its color and brightness.

5.3 A Configurable Multirotor Aircraft

Figure 9(a) shows a generalized Echidna version of the Arduino-based quadcopter design in Figure 2. The design replaces most of the discrete components with Echidna modules that are common to many Arduino designs (for example, the power supply, the programming pins, and the debugging interface), and uses `GenericParts` for the rest. It also factors out the motor driver circuit into a separate module, and then uses Python code to instantiate a variable number of `BrushMotor` (b) modules, depending on the value of the `n_rotor`. As a result, the same Echidna schematic can serve as the basis for 4-, 6-, and 8-rotor aircraft.

5.4 Redesigning Arduinos

Arduino [1] is a family of microcontroller-based development boards that use a common set of libraries that facilitate embedded software development. There are many variants (for example, Arduino Uno, Arduino Due, Arduino Pro, Arduino Mini, Arduino Nano) but they share many parts of their design in common: For instance, they all uses similar power supply circuits, and many of them include the same programming and debugging interfaces.

Figure 10 shows the module hierarchy of the Echidna version of Arduino Uno. The root modules contains just four modules: the power supply subsystem, the USB programmer subsystem, the microcontroller, and the USB connector. The simplicity of the top-level design protects the designer from significant complexity: In all, the designer only explictly instantiates seven sub-modules. All of these are common across multiple Arduino designs and many of them are good candidates for inclusion in a standard library of Echidna modules.

To quantify the opportunities for reuse with Echidna, we surveyed eight different Arduino variants to identify similar or identical components across multiple designs. Table 2 summarizes the results and identifies nine candidate modules. They range from simple to complex. One of the simplest is the in-circuit serial programming (ICSP) programming header. It only contains a 6-pin header. While this does not reduce the complexity of the top-level schematic, it would eliminate potential bugs. An ICSP module could have meaningful names for each electrical connection rather than relying on the designer to organize the nets correctly on the header's pins. The most complex reusable component is the USB programmer. It contains a dedicated AVR microcontroller, 18 electronic components, and 20 nets. This circuit appears in the Arduino Uno, the Arduino Mega, and the Arduino Due schematics and accounts for much of the complexity in those designs. Factoring it into a module reduces complexity and allows all three of those designs to share in improvements and bug fixes.

The bottom two rows of Table 2 quantify how much Echidna can reduce the complexity of the designs. The left hand column lists the modules. The "Other Modules" row includes several modules used in one or two designs. The numbers in the table are the number of instances of that module in a particular design. "Coverage" is the fraction of total electrical components in the final design that are part of a reusable component. "Reduction" measures the reduction in modules or parts in the top-level schematic. Both are proxies for the reduction in complexity that Echidna allows. Coverage values range from 69% to 91% and reduction values range from 52% to 84%.

5.5 ARM-FPGA Development Board

Echidna is useful on larger and more complicated designs as well. In this section we use Echidna to re-implement an open source development board called the iCore3 [9] that hosts an STMicro ARM Cortex-M4 processor (STM32F407IG) and a Altera Cyclone IV FPGA (EP4CE10F17C8N).

Figure 11 shows the organization of iCore3. The ARM processor connects to the Cyclone IV FPGA via a flexible static memory controller (FSMC) interface that allows the ARM processor can access FPGA directly via load and store instructions. The ARM processor connects to several external peripherals, including a USB controller, an Ethernet controller, and an SD card reader. A synchronous DRAM (SDRAM) connects to the FPGA.

Figuer 12 shows the key modules in the Echidna version of the same design. The "Top" module (at left) clearly shows the overall structure of the design (i.e., that it is an FPGA connected to an ARM combined with some headers). The FPGA and microcontroller subsystems (center and right) likewise make the structure of those portions of the circuit clear by hiding the internals of the peripherals.

The reduction in complexity is striking. At the top level, the number of connection between components is drastically reduced: The original iCore3 schematic described 359 nets connecting 1193 pins on 183 components. Top level schematic and the schematics for the FPGA and ARM subsystems contain just 16 modules, 33 nets, and 82 connections (only counting schematic connections, not counting the connections described in the CSV file). Seven groups, 516 connections in total are created with Echidna code (see below).

We describe the key modules of the design in more detail below.

5.5.1 The FPGA Module

Our FPGA Echidna module simplifies the task of inte-

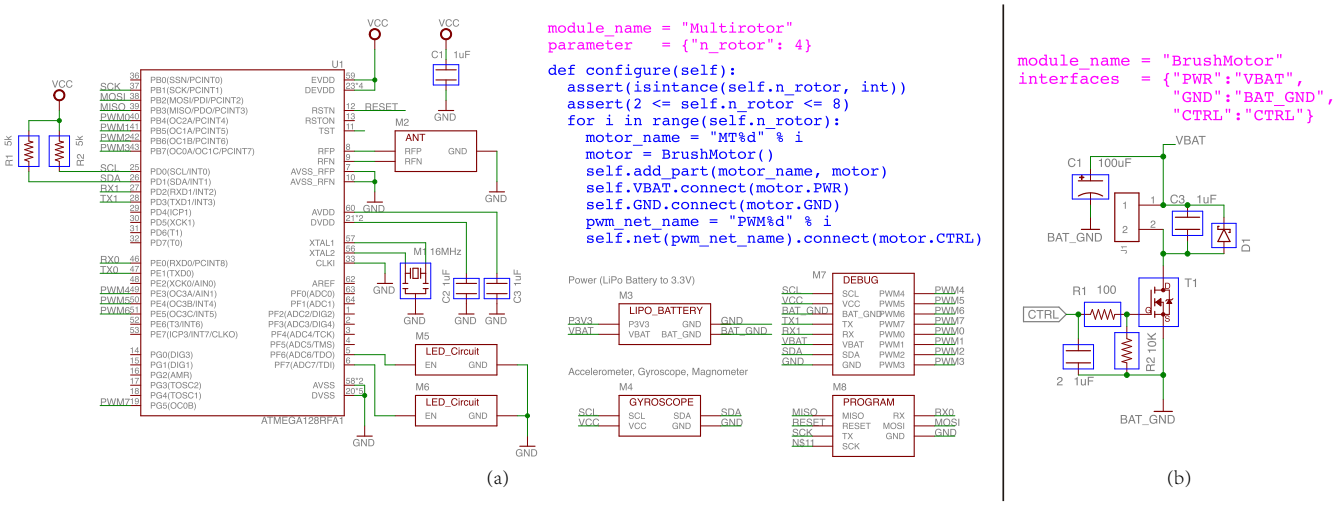


Figure 9: In the Echidna schematic for configurable multirotor aircraft, the top-level module (a) instantiates `n_rotor` copies of the `BrushMotor` module (b). The whole design contains over 60 discrete components, but only a fraction are visible at the top level, providing a clearer picture of the circuit's structure.

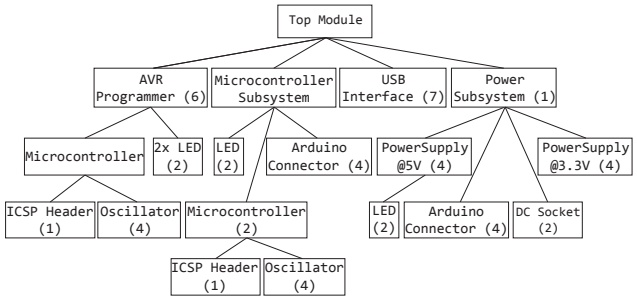


Figure 10: Breaking the design of the Arduino Uno into Echidna modules hides many details of the design from the top-level. The number in parenthesis indicates number of terminal submodules inside the corresponding module.

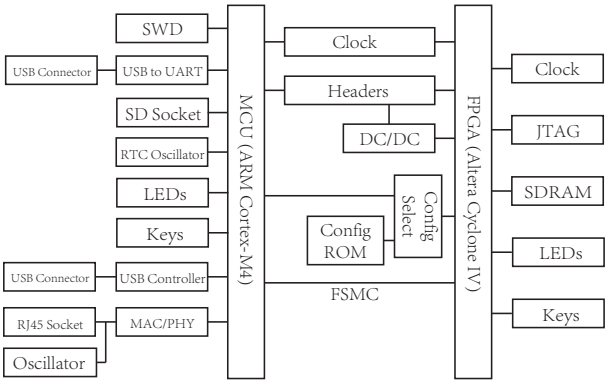


Figure 11: Original system organization of the ARM-FPGA hybrid system.

grating it into the design. The FPGA has 256 pins. Of these, 172 are IO pins that will connect to other devices (i.e., ARM processor, the SDRAM, etc.), 65 are power and ground pins, and 19 are devoted to functions such as configuration, a JTAG interface, and clock generation.

In most cases the designer is most concerned with the IO pins, since these (or the configurable fabric they connect to) are why she decided to include FPGA in the design. She will want control over how those pins connect to other devices and the voltage level that each bank of IO pins operates at. The other pins (i.e., power, ground, and configuration) and the components they connect to represent design effort that is only indirectly related to what the designer wants to achieve.

Our FPGA module (Figure 13) reflects this distinction. It hides the programming and power supply circuitry inside the module and exposes the IO pins. This allows the designer to focus on the interesting parts of the design.

Since all those components are hidden inside the module, the module's interface is much simpler. It includes eight banks of IO lines, clocks, configuration interface, power, ground, and status lines. It exposes the configuration interface so that its user can choose to implement an external configuration mechanism, or use use the embedded default configuration circuit. The figure also lists the module's parameters. They include the voltage levels for each of the IO banks, the target frequency for the FPGA's clock input, and whether to enable internal configuration circuit. Based on those parameters, the module instantiates the required clock generation, JTAG, and configuration circuits.

The module is also able to make extensive optimizations to the power supply components. Each IO bank can operate at different voltage, so in the worst case the FPGA module would require eight separate voltage regulators. However, if some of the banks operate at the same voltage, they could share a regulator, but that regulator would need to provide more current. Based on the IO level parameters for each bank, module creates the minimum number of voltage regulators required. Then, based on the efficiency attributes for those regulators, it calculates its input current and voltage

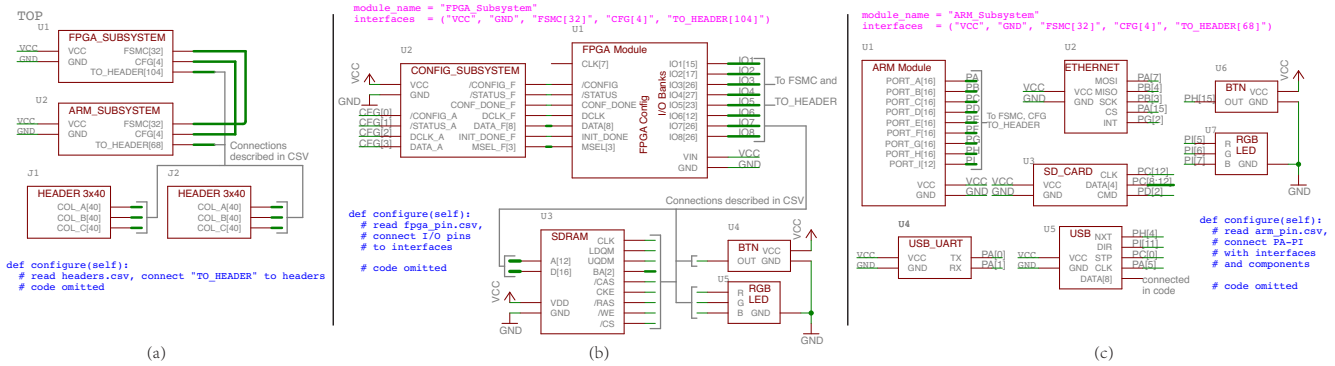


Figure 12: The Echidnaversion of the iCore3 [9] development board. The top-level design (a) contains the FPGA subsystem (b) and the ARM subsystem (c). The FPGA subsystem uses the FPGA module (detailed in Figure 13) to instantiate the FPGA and its supporting circuitry. The ARM subsystems uses a Echidna module for the ARM microcontroller for the same purpose. Echidna simplifies the design further by loading connection information from CSV file using the Python standard library's CSV API.

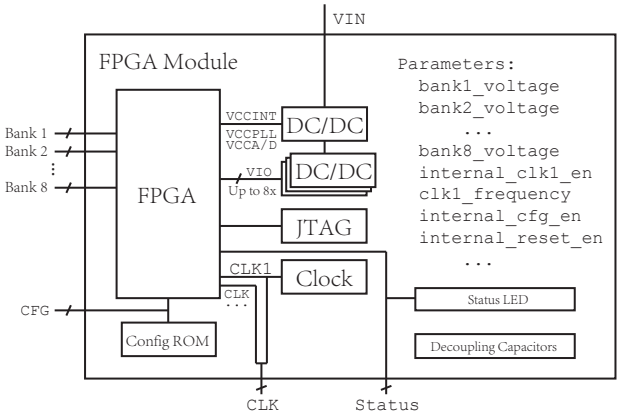


Figure 13: The FPGA module. We embed common supporting circuit of FPGA inside a wrapper and use parameters to customize them. This FPGA module hides trivial but necessary circuit from its user. It has much clearer interfaces, and is reusable in various designs.

requirements and, in turn exposes those as attributes on itself, so that the parent module can connect the power input of the FPGA module correctly.

5.5.2 *The ARM Module*

The ARM module (not shown) is simpler than FPGA module, since the ARM processor is less configurable. Like the FPGA, the microcontroller has some electrical interfaces that likely of great importance to the designer (e.g., the FSMC bus, the ethernet, USB, and other peripheral interfaces) and some that are required for the operation of microcontroller (e.g., clock generation, programming, and debug interfaces). The microcontroller module exposes former and hides the latter by integrate the components that connect to them.

5.5.3 *The ARM and FPGA Subsystems*

Figure 12 (b) and (c) combine the ARM and FPGA modules with peripheral modules. For the ARM subsystem this includes the IO interfaces for the board (i.e., Ethernet, USB, and a card reader). For the FPGA, it includes the memory and the configuration logic.

5.5.4 *Connecting the Modules*

Both the ARM and FPGA have numerous of pins. Besides the 32-wire FSMC bus connecting ARM and FPGA, most of the other pins from these two chips are connected to headers so users of the board can use them as they like. Connecting these pins correctly and documenting the connections is tedious, error prone, and makes reading and modifying the schematic difficult.

Echidna allows the designer to solve this problem by creating these connections in software. One solution would be to store the interface connection information in a CSV (comma separated values) file and use Python's built-in CSV parsing support to load the file and make the connections. Building such tables to describe pin assignments is a common design practice, and using the table directly eliminates the possibility of transcription errors.

6. RELATED WORK

Several other systems have tried to improve circuit design for PCBs, but none of them resolve all the challenges that Echidna addresses. Some PCB tool flows allow for design reuse, but none of their mechanisms provides parameterization or programmability. Specialized languages for describing PCB schematics showed effectiveness of expressing circuits in text-based programming, but they sacrifice the intuitive aspects of schematic capture. Hardware description languages for digital design (i.e., FPGA and ASIC design) apply similar ideas, but they do not address the specific challenges of PCB design.

Some PCB tool flows allow for design reuse or hierarchical design. For instance, Altium Designer [6] supports "snippets" which allow sub-circuit reuse and Cadsoft EAGLE [4] supports "modules," but neither of these mechanisms provides parameterization or programmability.

In addition, some PCB design software provides scripting functionality. For example, PADS [5] provides "automa-

tion server" that allows scripts to emulate GUI operations to manipulate the design, Altium Designer offers program functionality through "Altium Designer scripting system," EAGLE user can extends its functionality using its proprietary "User Language Program" (ULP) and scripting mechanisms. All these programming interfaces suffer from serious deficiencies. PADS' automation server, Altium's scripting system, and EAGLE's scripts are tied to GUI operations rather than treating the design as first-class data structure. EAGLE ULP interface is read-only. For example, in PADS a designer must write `ActiveDocument.ActiveSheet.Components("U1").Pins.Count` to access the pin counts of the component `U1` in current sheet rather than refer to the component itself. This makes the scripts brittle, interferes with composition and reuse, and makes it difficult to describe well-defined interfaces for sub-components. On the other hand, Echidna embeds the scripts (user functions) in the module, so that the scripts becomes part of the module's design. It also creates a private namespace for the module, exposes abstract electrical and programming interface to its user.

PHDL [10] is an specialized hardware description language models text-based schematics for PCB circuits, whose source can be compiled into a netlist that can be imported into layout tools. PHDL offers hierarchical and reusable design, but it sacrifices the intuitiveness of schematics, does not offer parameterization, and lacks native programming support.

Several other tools provide APIs for modifying schematics. PyEagle [11] wraps EAGLE's ULP functions in Python, but ULP cannot modify schematics. Swoop [8] and eaglepy [12] offer Python class bindings for EAGLE file structure. Both of them allows manipulation on low-level elements in EAGLE format such as line segments, but they do not offer hierarchical design, first-class module, and design reuse. Echidna uses Swoop to manipulate EAGLE files.

EDAsolver [13] is a declarative language that lets users specify the types of components in a device and will then automatically select component and connect pins. It hides components and connections decision from the designer, but it is not programmable beyond a limited predefined set of operations.

Hardware description languages (HDLs) like Verilog and VHDL describe the structure, behavior, and design of digital circuits and are commonplace digital IC design. Widely-used HDLs provide limited (and cumbersome) parameterization support. Some recent research projects attempt to remedy and combine general programming languages with hardware description languages to improve programmability and parameterization. Chisel [14] brings in object orientation, functional programming, parameterized types and type inference to digital logic design. MyHDL [15] attempts to model digital logic systems with Python functions. Genesis 2 [16] uses SystemVerilog as an underlying hardware description language, but integrate Perl scripting to provide more flexible parameterization and elaboration. Echidna is similar in some respects to these tools, but focuses specifically on PCB design. In particular it provide support for part selection and more detailed control over circuit implementation, tasks usually handled by the "backend" of conventional digital logic design tools.

7. CONCLUSION

We have described Echidna, a hybrid schematic/code-based language for specifying PCB schematics. Echidna allows designers to use schematic capture to specify the structure of the circuit and then augment it with programmability to enable sophisticated parameterization and design reuse. We find that Echidna has the potential to reduce the complexity of PCB circuit designs, accelerate the design process, and prevent many common bugs.

8. REFERENCES

[1] "Arduino." `http://www.arduino.cc/`, 2015. Accessed: 2015-11-24.

[2] "Raspberry pi." `https://www.raspberrypi.org/`, 2015. Accessed: 2015-11-24.

[3] "Lithium ion battery charger for solar-powered systems cn3065, rev 1.0."

[4] "Eagle, the easy applicable graphical layout editor." `http://www.cadsoftusa.com/`, 2016. Accessed: 2016-4-1.

[5] "Pads pcb design software: Mentor graphics." `https://www.pads.com/`, 2016. Accessed: 2016-4-1.

[6] "Altium: Pcb design tools." `http://www.altium.com/`, 2016. Accessed: 2016-4-1.

[7] "Orcad pcb solutions." `http://www.orcad.com/`, 2016. Accessed: 2016-4-1.

[8] "Swoop is a python library for working with cadsoft eagle files." `https://pypi.python.org/pypi/Swoop`, 2015. Accessed: 2016-4-1.

[9] "icore3 dual core control system - schematic." `http://www.eeschool.org/forum.php?mod=viewthread&tid=1901`, 2015. Accessed: 2016-4-1.

[10] "Phdl: An hdl alternative to pcb graphical schematic capture tools." `http://sourceforge.net/p/phdl/wiki/Home/`, 2015. Accessed: 2015-11-24.

[11] "Pyeagle - use most of the eagle ulp api from python." `https://github.com/rmawatson/PyEagle`, 2014. Accessed: 2016-4-1.

[12] "eaglepy: Read, modify, and create cadsoft eagle files." `https://pypi.python.org/pypi/eaglepy`, 2014. Accessed: 2016-4-1.

[13] "Edasolver." `http://edasolver.com/`, 2015. Accessed: 2015-11-24.

[14] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), pp. 1216–1225, ACM, 2012.

[15] J. Villar, J. Juan, M. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe, "Python as a hardware description language: A case study," in *Programmable Logic (SPL), 2011 VII Southern Conference on*, pp. 117–122, April 2011.

[16] O. Shacham, M. Wachs, A. Danowitz, S. Galal, J. Brunhaver, W. Qadeer, S. Sankaranarayanan, A. Vassiliev, S. Richardson, and M. Horowitz, "Avoiding game over: Bringing design to the next level," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, (New York, NY, USA), pp. 623–629, ACM, 2012.