BHDL: A Lucid, Expressive, and Embedded Programming Language and System for PCB Designs

Hebi Li[†], Youbiao He[†], Qi Xiao[§], Jin Tian[†] and Forrest Sheng Bao[†]
[†]Dept. of Computer Science, [§] Dept. of Electrical and Computer Engineering
Iowa State University, Ames, IA, 50011, United States
Email: {hebi, yh54, qxiao, jtian}@iastate.edu, forrest.bao@gmail.com

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586086

***Abstract*—Graphical PCB design tools like KiCAD lack support for high-level abstraction such as functions and loops. To improve PCB design productivity, we hereby present BHDL, a programming framework for PCB designs. In its compact and declarative syntax, schematics and layouts can be modeled effectively and expressed concisely. Treating all circuits, even a resistor, as functions, BHDL naturally supports modularized development that builds a complex design up from smaller designs hierarchically. As an embedded Domain Specific Language (eDSL), BHDL allows users to leverage the full feature of the host language for customization and extension. Our Jupyter kernel supports web-based, REPL-style development and generates auto-placed PCBs.**

***Index Terms*—Electronic Design Automation (EDA), Computer-Aided Design (CAD), Printed Circuit Board (PCB), Hardware Description Language (HDL), Programming Language (PL), Automatic Placement**

I. Introduction

Printed Circuit Boards (PCBs) have been dominantly designed with graphical Electronic Design Automation (EDA) tools such as KiCAD and EAGLE. Although graphical EDA tools have the advantage of intuitive interface and rich visualization, they lack abstractive and efficient representation of circuits. As a result, PCB designs are prone to be hard to maintain, share, and reuse. A recent study [1] finds that PCB engineers often complain about the low abstraction level of current EDA tools, which makes PCB design significantly tedious.

Unlike graphical EDA tools, Hardware Description Languages (HDLs) allow engineers to define circuits using computer programs. Such abstraction improves development efficiency. The popularity of the VHDL [2] and Verilog [3] languages in digital IC and FPGA development have inspired the study into HDLs for PCBs. PHDL [4] is a description language that compiles to a netlist of connected component. JITPCB [5] introduces a declarative language embedded in a general-purpose language Stanza [6]. Pursued as proprietary software [7], documentation about the language syntax and semantics is very limited. SkiDL [8] and PCBDL [9] are two Python modules for PCB designs.

However, these existing HDLs for PCBs all fall short in certain aspects in the PCB design workflow. PHDL and JITPCB only allows expressing non-bus connections as netlists, which can be verbose and hard to verify and update. Limited by Python, SkiDL, and PCBDL programs cannot be declarative, making them inconvenient and verbose for describing circuits. This also makes it challenging to modularize, especially when hierarchically, a complex design into smaller user-designed circuits in SkiDL and PCBDL. No existing systems, except JITPCB, support layout co-design where the user specifies constraints for placing circuit components on a board. Without layout support, the PCB design stays at schematics.

Li, He, and Bao's work in this paper is partially supported by NSF grants MCB-1821828 and CNS-1817089. Corresponding author: Forrest Sheng Bao.

To facilitate a more effective PCB design workflow, we propose BHDL (standing for Board HDL), a declarative, simple, modular, and expressive HDL and system, in this paper. BHDL is designed as an *embedded Domain Specific Language (eDSL)* inside the modern and highly-regarded programming language Racket [10]–[12]. Domain specific for PCBs, BHDL is declarative to focus on only the necessary components to define a circuit so that programs can be concise and effortlessly readable. BHDL introduces only a handful of structures and keywords to smooth the learning curve for new users.

To support modularized design, all circuits, even the simplest discrete parts like a resistor, are modeled as functions and can be easily parameterized and called to form bigger circuits. BHDL is expressive despite being a simple DSL, because it has access to the rich language features of the host language like functions and loops. For example, we can create an array of pushbuttons using a for-loop or create different test versions of a circuit using an if-statement easily instead of drawing several, highly-overlapping schematics (Fig. 9 and Section II-E). Users also have native access to the enormous software libraries of the host language ecosystem.

BHDL provides a simple and intuitive syntax for expressing circuit connections naturally. In addition to the `net` syntax for expressing netlist, BHDL also provides the `series`, `parallel`, and `bus` syntax for making series/parallel wires and to connect a bus of pins. All connection syntax in BHDL can be mixed and nested in arbitrary depth to express complex circuits in a concise and hierarchical manner. Besides, BHDL supports in-place anonymous part creation inside the connection syntax, instead of referring to a previously defined part instance. We present the complete and formal syntax and their well-defined formal semantics.

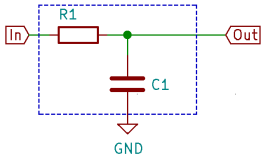
PCBs are tightly coupled with physical layouts. Without layout support, the PCB design stays at schematics. BHDL supports layout design in two complementary ways. First, BHDL supports the co-design of the physical layout directly in the language. In particular, BHDL provides simple and relative layout combinators namely `hstack`, `vstack`, `rotate` and `at`. These layout combinators are local, facilitating hierarchical designs where smaller circuit layouts can be directly used in composing larger circuits. In addition to programmatic layout co-design, we also adapt state-of-the-art VLSI auto-placers [13], [14] to find appropriate locations for free parts, with simple and effective improvements for PCBs.

The development model of BHDL is interactive and incremental, follows the Read-Eval-Print Loop (REPL) [15] or interactive scripting model. BHDL is shipped with the Jupyter [16] notebook environment for easy coding, running, and visualization, as shown in the screen-shot in Fig. 11. Our framework generates placed PCB board. The board can then be routed using the open-source FreeRouting [17] tool to produce the final PCB for manufacturing. Our framework is open source at https://bhdl.org

II. BHDL Syntax and Semantics

We begin introducing the BHDL language using a simple RC circuit in Fig. 1. The formal syntax is given later in Fig. 3 using BNF [18] notation, while the formal semantics are given in Fig. 7 using operational semantics notation [19], [20].

Because BHDL is based on Racket which roots back to LISP, BHDL uses nested pairs of brackets to control the structure. A circuit design begins with the keyword `circuit` which, optionally, follows the circuit name. The circuit name can be used later as a function to instantiate an instance of a circuit, like a constructor for a class in OOP. A circuit definition consists of 4 clauses: `pin`, `part`, `wire`, `layout`. One may consider the `pin` clause as declaring the arguments and returns, the `part` clause as calling other functions, and the `wire` clause as other code in the body of the function.



(a) Schematic

```
(circuit RC
  (pin In Out GND)
  (part [R1 (R In Out)]
        [C1 (C Out GND)]
        )
) ; end circuit
```

(b) BHDL code

Fig. 1: An example RC circuit

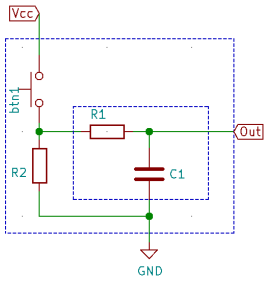
A. The `pin` syntax

The `pin` syntax defines the interface signals, such as I/Os, power, and ground, of a circuit so it can be hooked with other circuits to form a bigger design. Such signal names are listed after the keyword `pin`. A pin of a circuit can be referred later in a `wire` clause using the dot operation, e.g., `R1.2` means the 2nd pin of `R1`. In Fig. 1, the simple RC circuit has three I/O signals, the `In`, `Out`, and `GND`.

B. The `part` syntax and circuit instantiation

In BHDL, every circuit, even the simplest parts like a resistor, is a function. To instantiate a circuit instance, call the circuit function with or without arguments and parameters, and then bind it with a variable. In Racket or other LISP dialects, a function call is a prefix expression `( function-name argument-1 argument-2 ... )`

BHDL extends it by allowing parametrizations with keywords and default values for tasks unique to PCB design such as selecting footprints and set resistances (see Sections II-E and III-A).



(a) Schematic

```
(circuit RC_button
  (pin Vcc Out GND)
  (part [RC1   (RC   _   Out GND)]
        [R2    (C  _  GND)]
        [btn1  (SW VCC _)]
        )
  (wire
     (net R2.1 btn1.2 R1.1)
  )
) ; end circuit
```

(b) BHDL code

Fig. 2: A BHDL circuit created for the RC circuit.

If arguments are used when calling a circuit as a function, the first few shall match those defined in the `pin` clause. Hence function calls with arguments mean connecting signals of the circuit under construction (the arguments) to pins of the circuits begin called. In Fig. 1, the two pins of `R1` are connected to `In` and `Out` while those of `C1` are connected to `In` and `GND`.

In Fig. 2, `RC1` is an instance of the circuit `RC` defined in Fig. 1, whose last two pins are connected to the pins `Out` and `GND` of the new circuit `RC_button`. The underscore `_` in the first argument of `RC` is a placeholder variable meaning that the `In` pin of `RC1` is connected to nowhere, for now. The similar usage is applied for `R2` and `btn1`. In order to connect the three hanging-in-the-air pins, we will introduce a new syntax `wire`.

syntax	notes
Part 1: host language	
$prog ::= e \ \ldots$	(expression)
$e ::= a$	(constants)
$\mid x$	(var)
$\mid \lambda x.e$	(function definition)
$\mid (f\ e)$	(apply)
$\mid$ let $x = e$ in e	(let-binding)
$\mid$ (for ($[x\ e]$ ...) e_{body})	(loop)
$\mid$ (if $e_{cond}\ e_{true}\ e_{false}$)	(conditional)
Part 2: `circuit` clause	
$\mid$ (circuit $X_{opt}\ pa_{opt}\ c$ ...)	(circuit X)
$\mid$ (X pin_{opt} ... v_{opt} ...)	(circuit instantiation)
$pa ::=$ (param x ... $[x\ e]$ ...)	(circuit parameters)
$c ::=$ (pin $p \ldots$)	
$\mid$ (part $[x\ e] \ldots$)	
$\mid$ (wire $w \ldots$)	
$\mid$ (layout $l \ldots$)	
Part 3: `wire` clause	
$w ::=$ (net $e.pin$ ...)	(the `net` syntax)
$\mid$ (series $w \ldots$)	(the `series` syntax)
$\mid$ (parallel $w \ldots$)	(the `parallel` syntax)
$\mid$ (bus $b \ldots$)	(the `bus` syntax)
$\mid e$	
$\mid e.pin$	
$b ::=$ ($[e.pin$ ...$]$)	(`bus` clause variant 1)
$\mid$ (e $[pin$ ...$]$)	(`bus` clause variant 2)
Part 4: `layout` clause	
$l ::=$ (hstack $al_{opt}\ l$ ...)	(horizontal layout)
$\mid$ (vstack $al_{opt}\ l$ ...)	(vertical layout)
$\mid$ (rotate l $degree$)	(rotation)
$\mid$ (at $l_1\ x\ y\ l_2$)	(overlapping)
$\mid e$	
$al ::=$ center	(default alignment)
$\mid$ top $\mid$ bottom $\mid$ left $\mid$ right	

Fig. 3: Formal BHDL syntax. From top to bottom: λ-calculus syntax, BHDL `circuit` syntax, BHDL `wire` connection syntax.

C. The `wire` syntax

The `wire` syntax in BHDL supports four kinds of connection clauses: `net`, `series`, `parallel`, and `bus`, whose examples are given in Fig. 4.

1) `net`: The very basic one, the `net` syntax, takes a list of pins to declare a conventional "netlist" connecting all the given pins together. In Fig. 4a, the BHDL code connects pin 1 of three resistors together. Formally, in Fig. 7a, we define the value of nets$(p_1 \ \ldots \ p_n)$ to be a circuit y where the nets of y is a single netlist connecting all the given pins.

2) `series`: The `series` syntax is used to specify parts connected in series, such as the two resistors and a capacitor in Fig. 4b.

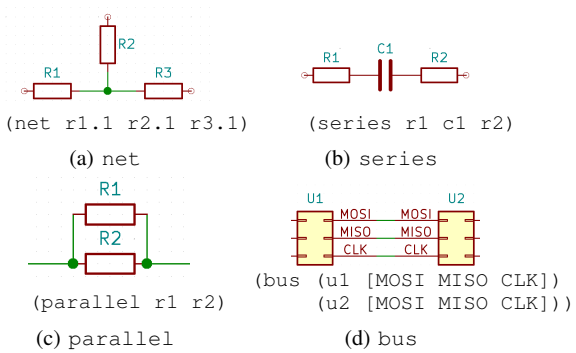


Fig. 4: Example schematics and BHDL code of four connection types.

The `series` syntax expects a list of parts, and the semantics is to connect the `left` and `right` pins of each component in a series. For a two-pin non-polarized device, like a resistor, BHDL will randomly assign a pin as `left` and the other as `right`. For polarized devices and devices with more than two pins, users and library authors can choose which pins are treated as `left` and `right`. Dot operations referring to pins can be used in series connections, e.g.,

```
(series r1 d1.cathode d1.anode r2)
```

where `d1` is a diode while `r1` and `r2` are two resistors. The formal operational semantic of the `series` is shown in Fig. 7b.

3) `parallel`*:* The `parallel` syntax is used to specify parts connected in parallel, such as the two resistors in Fig. 4c. The `parallel` syntax expects a list of parts, and the semantic is to connect the `left` and `right` of each component in a in parallel. The `left` and `right` here are handled similarly to the case for `series`. The formal operational semantic of the `parallel` connection is shown in Fig. 7c.

4) `bus`*:* The `bus` syntax is used to connect corresponding pins of two or more entities such as the MOSI, MISO, and CLK pins of two SPI ports in Fig. 4d. Using the `bus` syntax, this can be very convenient and less error-prone compared with using `series` and `net` syntax. As shown in Fig. 3 (the b ::= line), the BUS syntax has two variants. The code in Fig. 4d is variant 2 where part name is specified once because pins share the part name. It can be rewritten into the more verbose variant 1 as

```
(bus ([u1.MOSI u1.MISO u1.CLK])
     ([u2.MOSI u2.MISO u2.CLK]))
```

where the part name is specified for every pin. The formal `bus` semantic is shown in Fig. 7d.

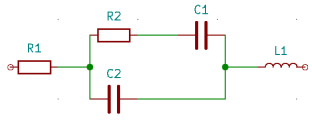
The `bus` connection can be more useful when concurrently connecting many parts. For example, in Fig. 5, the `bus` syntax is used to connect corresponding pins of Arduino boards of different form factors. Correspondence between pins of different Arduino boards is listed clearly in our `bus` syntax. The underscores are used as placeholders for pins that do not exist for certain circuits.

5) Mixing and nesting `wire` *syntax:* The idea of utilizing `left` and `right` pins in the wire semantics enables developers to mix and nest the connection syntax in arbitrary depth. This allows BHDL to easily declare relatively "long" wires for more complex circuits. An example of nesting `series` and `parallel` is given in Fig. 6. If using only the `net` syntax, the code is verbose and not readily as pins and connections need to be specified individually. By nesting `series` and `parallel` syntax, we can get a much more natural and simple code in Fig. 6c where every part is used only once.

6) In-place part creation: In combination with the `series` and `parallel` syntax, BHDL supports *in-place anonymous component creation*. This allows developers to create parts like resistors and capacitors in-place when they are used, without the relatively tedious workflow of creating parts beforehand, binding to a variable, and referring to the parts in the field. In Fig. 6d, we extend the nesting code to use in-place parts creation so that developers do not need to refer to a previously created part. Because such components are anonymous, we borrow the term "lambda functions" to call them "lambda circuits."

```
(circuit ultimate_arduino_adapter
 (part ...)
 (wire
  (bus
   (uno   [A0 A1 A2 A3 A4 A5 -  -  -  D2 ... D13 GND 3V3])
   (micro [A0 A1 A2 A3 -  -  -  -  -  ... -   GND -   ])
   (mini  [A0 A1 A2 A3 -  -  A6 -  -  D2 ... D13 GND -   ])
   (nano  [A0 A1 A2 A3 -  -  A6 D0 D1 D2 ... D13 GND 3V3])
   (mkr   [A0 A1 A2 A3 A4 A5 A6 D0 D1 D2 ... D13 GND 3V3]))
 (layout ...)))
```

Fig. 5: A `bus` connection example for an adapter to bridge all Arduino form factors. Ellipses are used for omission and do not mean ranging.



(a) The circuit

```
(net r1.2 r2.1 c2.1)
(net r2.2 c1.1)
(net c1.2 c2.2 l1.1)
```

(b) It's verbose in netlists

```
(series
  r1
  (parallel
    (series r2 c1)
    c2)
  l1)
```

(c) Nest `series` and `parallel`

```
(series
  (R 10)
  (parallel
    (series (R 20) (C 100))
    (C 100))
  (L 10))
```

(d) In-place part creation

Fig. 6: Convenience of using nested `series` and `parallel`, and in-place part creation.

D. The `layout` syntax

PCBs are tightly coupled with physical layouts because it is close to the final physical product. It can be tedious and inaccurate to design the physical layout in the existing EDA workflow, e.g., repeatedly move parts around, or compute coordinates using external tools and then import. Users could specify absolute global coordinates in existing EDAS tools. However, it makes hierarchically compose higher-level designs extremely difficult.

Inspired by JITPCB [5] and the functional picture library Pict [21], BHDL provides several simple but powerful relative layout combinators, enabling a hierarchical co-design of physical layout. The idea is to specify the relative location among components, such as "left of" and "above". The layout is hierarchical: developers can lay out the sub-circuit first, and the placed circuits can be directly used to layout a bigger circuit, using the same set of layout primitives.

Formally, the syntax of layout is shown in Fig. 3. We provide four simple layout combinators, namely `hstack` (horizontal stack, left-to-right), `vstack` (vertical stack, top-to-down), `rotate`, and `at`. For `hstack` and `vstack`, an optional alignment argument can be specified, which defaults to center alignment, and can be chosen also from `top`, `bottom`, `left`, or `right`. The spacing between the components can also be specified pragmatically, and is omitted here for simplicity. `rotate` turns the circuit by a degree,

$$\frac{y.nets \rightsquigarrow \text{nets}(p_1, ..., p_n) \quad y.L \rightsquigarrow p_1 \quad y.R \rightsquigarrow p_1}{(\texttt{net}\ \ p_1 ... p_n) \rightsquigarrow y}$$

(a) `net` semantics

$$\frac{N_{i,j} \rightsquigarrow \text{nets}(x_i.R, x_j.L) \quad y.nets \rightsquigarrow N_{1,2} \cup ... \cup N_{n-1,n} \quad y.L \rightsquigarrow x_1.L \quad y.R \rightsquigarrow x_n.R}{(\texttt{series}\ x_1 \ldots x_n) \rightsquigarrow y}$$

(b) `series` semantics

$$\frac{N_1 \rightsquigarrow \text{nets}(x_1.L, ..., x_n.L) \quad N_2 \rightsquigarrow \text{nets}(x_1.R, ..., x_n.R) \quad y.nets \rightsquigarrow N_1 \cup N_2 \quad y.L \rightsquigarrow x_1.L \quad y.R \rightsquigarrow x_1.R}{(\texttt{parallel}\ x_1 \ldots x_n) \rightsquigarrow y}$$

(c) `parallel` semantics

$$\frac{N_i \rightsquigarrow \text{nets}(x_1.p_i, \ldots, x_n.p_i) \quad y.nets \rightsquigarrow N_1 \cup ... \cup N_m \quad y.L \rightsquigarrow \text{None} \quad y.R \rightsquigarrow \text{None}}{(\texttt{bus}\ (x_1[p_1 \ldots p_m]) \ \ldots \ (x_n[p_1 \ldots p_m])) \rightsquigarrow y}$$

(d) `bus` semantics

$$\frac{X.pin \rightsquigarrow \{q_1, ..., q_n\} \quad N_i \rightsquigarrow \text{nets}(x.q_i, p_i) \quad y.nets \rightsquigarrow N_1 \cup ... \cup N_m \quad y.L \rightsquigarrow \text{None} \quad y.R \rightsquigarrow \text{None}}{(\texttt{circuit}\ (\texttt{part}\ [x\ (X\ p_1\ ...\ p_n)]\ ...)\ ...) \rightsquigarrow y}$$

(e) Circuit Instantiation semantics

Fig. 7: Formal BHDL semantics. In the notation, the operator $\rightsquigarrow$ should be read as "reduce to". Below the long horizontal line, the code to the left of $\rightsquigarrow$ is resolved to y which is further defined above the line. The nets$(x_i.R, x_j.L)$ means a netlist consisting of two pins, the "right" pin of x_i and the "left" pin of x_j.

```
(layout
  (hstack
    (vstack btn1 btn2 btn3)
   esp01))
```

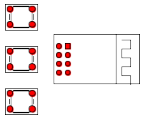


Fig. 8: A `layout` example.

and `at` defines the offset (x, y) between two parts. With these layout combinators, users can compose the layout of a circuit hierarchically and functionally, together with the design of the circuits.

We show a layout example in Fig. 8. The `vstack` clause stacks three pushbuttons (`btn1 btn2 btn3`) vertically, default aligned in the middle. The `hstack` clause further places the 3-button group and an ESP01 module (`esp01`) from left to right, default aligned in the center.

E. Embedding BHDL into the host language

In this section, we illustrate how BHDL can be used together with host language constructs. In Fig. 3 Part 1, we show the language constructs available in the host languages, including variable binding, function definition and application, and control flows such as conditionals and loops.

Our `circuit` DSL embeds inside the host language naturally, and can interact with the host language in two complementary ways. Firstly, the BHDL circuit is a first-class value inside the host language, and it can appear anywhere a value is expected. A circuit can be bound to a variable, passed as an argument to a function, as well as an expression constituting a function body or let-expression body. Secondly, we can use host language constructs inside the BHDL circuit clauses, including `part`, `wire`, as well as `layout`.

We show an example circuit in Fig. 9. In the `part` clause, we define a list of buttons using a *for*-loop. Each of the buttons is connected with a pull-up resistor in series across the battery. The joint between the button and the pull-up resistor is connected to a

```
(circuit TripleButton
    (param [mode "Release"] ) ; keyword parameters
    (part [buttons (for ([i 3]) (SW) ) ]
          ; create an array of 3 switches by
          ; calling the circuit/function SW thrice
          [bat1 (Battery '3V)]
          [u1 (ESP01)]
    ); end part
    (wire
      (net bat1.pos u1.VCC)
      (net bat1.neg u1.GND)
      (for ([btn buttons]; Python:  for btn in buttons:
            [ctrl_pin (list (if (= mode "Release")
                                u1.CH_PD
                                u1.RST ; else case
                            ); end if
                            u1.GPIO0 u1.GPIO2
                    ); end list
            ])
        (series bat1.pos (R '10k) btn bat1.neg)
        (net ctrl_pin btn.1)
      ) ; end for
    ); end wire
)
```

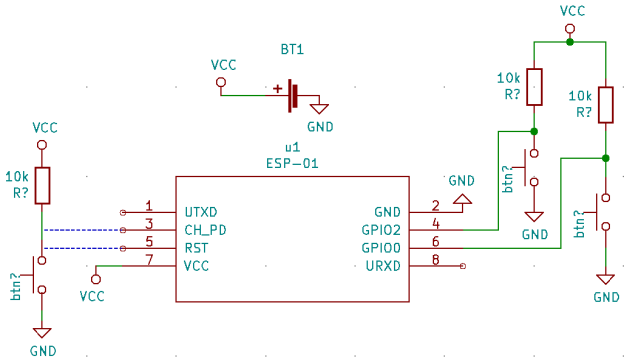


Fig. 9: An example of using function, branch, and loops with BHDL

pin of the ESP-01. By using a for-loop in the `wire` clause, we can create three copies of the resistor-button series circuit, and connect each of them with one pin of `u1`. Instead of writing three `series` clauses and three `net` clauses, we only need to write the template for one iteration. Further, we create two versions of the circuit, controlled by the parameter `mode`. The two versions differ in the dash-lines in Fig. 9 that only one of them should become an actual wire depending on the `mode`. We use an `if` statement to make the selection. When `mode` is `Debug`, `u1`'s `CH_PD` is hooked with the left most button. When `mode` is `Release`, it is `u1`'s `RST`. This circuit can then be called to generate two versions with different values for the parameter `mode`:

```
(TripleButton #:mode "Debug"); code for version 1
(TripleButton #:mode "Release"); code for version 2
```

III. System and Implementation

A. Part Library Management

BHDL uses an additional `footprint` syntax in a `circuit` definition to simplify the effort to associate the circuit with a footprint. Beginning with the keyword `footprint`, a `footprint` clause is immediately followed by a package name, and then the pin names in the order defined in the footprint file. For example, a resistor can be defined in BHDL as simple as

```
(circuit R [value '10k]
          ; default value for resistors is 10kOhm
    (import "https://github.com/KiCad/kicad-footprints/blob/
   master/Resistor_SMD.pretty/R_0603_1608Metric.kicad_mod"
```

```
as 0603)
(pin 1 2)
(footprint 0603 1 2))
```

where pads 1 and 2 are defined in a footprint library. The `import` clause allows users to rename a footprint name, usually lengthy, to a short name. The first argument of the `import` clause can be a footprint in the current search path, an absolute or relative file path, or an Internet URL. The ability to import from an Internet URL allows BHDL to have instant access to a large collection of existing high-quality open-source footprint libraries. Currently BHDL supports the KiCAD-format footprint library but can be extended easily later.

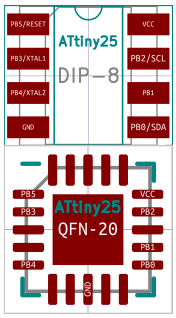
A circuit can have multiple footprints for users to select. For example, the code

```
[r (R _ _ '100k #:fp 0603) ]
```

selects 0603 package for a resistor and sets the value to `100k` from default `10k` (the first argument after pins is reserved for `value` for convenience). BHDL assumes default values for parameters. For footprints, the first footprint defined in a circuit is the default footprint. Thus, there will be no difference if the footprint parameter is not selected in Fig. 1.

A more complex example is given in Fig. 10 where the 8 pins of an ATtiny25 are associated with DIP-8 and QFN-20 packages. Because ATtiny25 occupies only 8 out of the 20 pins that QFN-20 provides, unused pins are indicated by the special placeholder symbol underscore. In addition, to make pin referencing easier, here we use an extended `pin` syntax that each pin can be a list of names, for example, `PB0` can also be called `SDA`. The example also introduces the `prefix` syntax so parts can be named and/or numbered starting from the prefix.

```
(circuit ATtiny25
  (pin
    [PB0 SDA]
    [PB2 SCL]
    [PB3 XTAL1]
    [PB4 XTAL2]
    [PB5 RESET])
  (footprint DIP-8
    PB5 PB3 PB4 GND
    PB0 PB1 PB2 VCC)
  (footprint QFN-20
    PB5 PB3 - - PB4
    - - GND - -
    PB0 PB1 - PB2 VCC
    - - - - -)
  (PREFIX 'U))
```



(a) BHDL code (b) Footprint rendered with pin names

Fig. 10: An example of footprint management

B. Auto-Placement

An autoplacer finds locations for parts to meet layout constraints specified by users in BHDL, as well as parts that are free from constraints. Automatic placement has been intensively studied in the VLSI domain [13], [22], [23]. But much less attention has been paid to the placement problem for PCBs. We initially tried to port some publically available autoplacers [13], [23] for VLSIs to PCB placement. But it was not successful, probably because such autoplacers target the VLSI problems, and are not suitable for PCBs.

We thus present our implementation of auto-placement engine for PCBs. We implement the state-of-the-art analytical global placer RePlace [13], [24]–[26]. The RePlace algorithm optimizes the total wirelength (HPWL). However, directly applying the algorithm may place components too close to each other, reducing the routability. Tuning the density penalty of RePlace formulation is tricky to achieve desired component separation. We apply a simple and effective approach to easily control the density at a higher-level. In particular, we introduce an padding parameter that enlarge components' footprints by the padding before feeding into the placement engine. It has shown promising and effective control over the placement density, and is easy to tune.

We also implement a simulated annealing (SA) detailed placer [14] to legalize the local overlapping. One special placement constraint for PCBs is that PCB components are typically much larger and more irregular than VLSI cells, and rotation is desired to fit the components in small space. Thus in the SA-based detailed placer, in addition to the random walk in horizontal and vertical directions, we introduce a third dimension to rotate the components by random angles. This enables our placer to successfully rotate components in small space and generate valid placement result when non-rotating placer fails to solve.

C. Implementation Details and User Interface

We implement our system in the Racket [10], [11] language framework, because it has a rich set of tools for building syntax abstractions and DSLs. Our auto-placer implementation is done in a higher-level Julia [27] programming language, for Julia is easier to debug and extend, with efficient and straight-forward GPU acceleration [28].

The development model of BHDL follows the Read-Eval-Print Loop (REPL) [15] inherited from Racket. In particular, the BHDL code is not just compiled to the output KiCAD board every time the design changes. Instead, it is an incremental process that the circuit definitions are read and evaluated, and user can interact with the running process and incrementally edit the code to tune their circuit design. In addition, we use the Jupyter [16] notebook as the preferred development environment because it is easy to run code blocks and get results and board visualization directly besides the code blocks.

A screenshot of a BHDL Jupyter session is given in Fig. 11 showing different stages of using our system: incremental development of a circuit on the left (left column), rendering of auto-placement result (top-center and top-right), and exporting files in KiCAD PCB format (bottom-center, exportable to Gerber for manufacturing) and Bill of Material (BOM, bottom-right). In particular, the top-center circuit is an agronomic keyboard. Our `layout` syntax makes specifying the relative positions and orientations between keys very convenient.

IV. Conclusion

In this paper, we propose BHDL, a declarative, simple, modular, and expressive HDL and system for efficient PCB design. BHDL expresses circuit connections naturally by providing simple and intuitive syntax that can be mixed and nested, with well-defined formal semantics. Our system also features programmatic hierarchical co-design of physical layout as well as adapted state-of-the-art auto-placers to turn the designs into physically placed PCB boards. BHDL is open-source and shipped with REPL-driven Jupyter environment for interactive and incremental development.

We hope our framework can be a platform to facilitate the research to revolutionize PCB development. Our language and system will make the creation and analysis of PCBs and tasks on PCBs, such as verification, simulation, placement, and routing, much more systematic, as a code-based representation of PCBs are much more easier to be analyzed. Techniques developed in software engineering can be ported to help PCB designs. With the declarative and compact representation of PCB design problems, there are also opportunities to incorporate deep learning and reinforcement learning automate or optimize tasks in PCB designs.

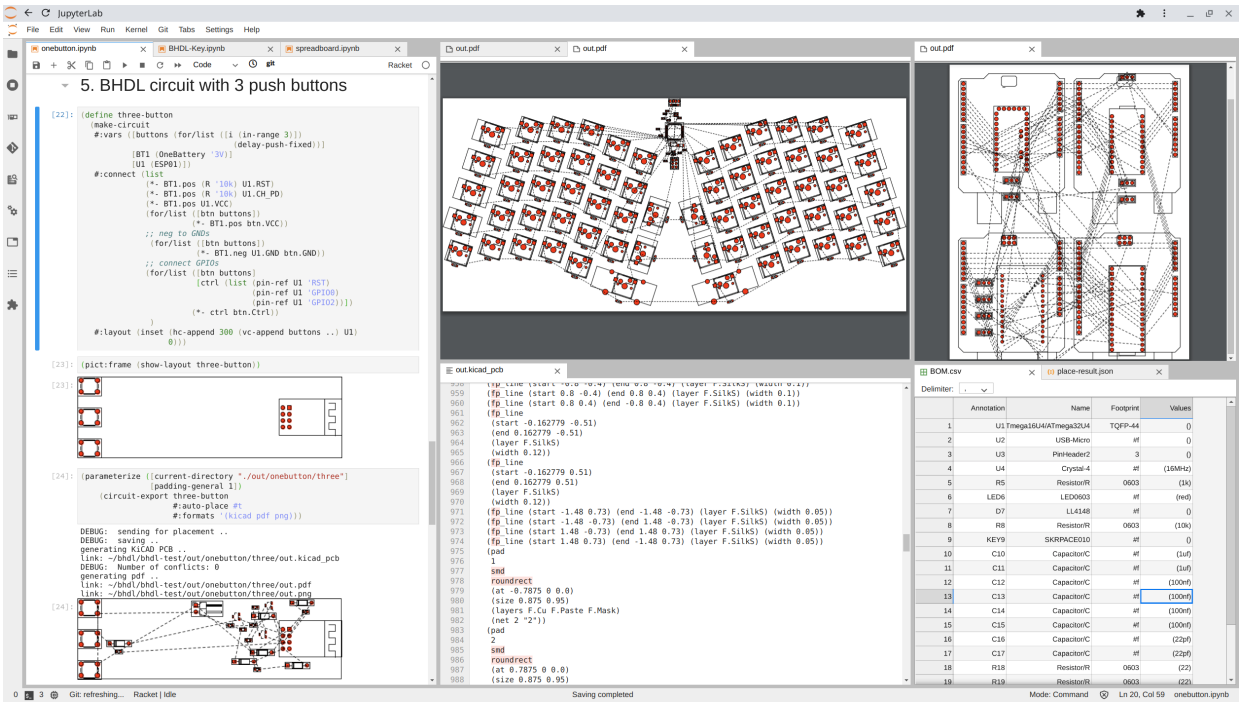


Fig. 11: Jupyter Development Environment for BHDL, showing example code, layout renderings, BOMs, and KiCAD-format PCB file.

References

[1] R. Lin, R. Ramesh, A. Iannopollo, A. Sangiovanni Vincentelli, P. Dutta, E. Alon, and B. Hartmann, "Beyond schematic capture: Meaningful abstractions for better electronics design tools," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, 2019, p. 283.

[2] "IEEE standard for VHDL language reference manual," *IEEE Std 1076-2019*, 2019.

[3] "IEEE standard for SystemVerilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, 2018.

[4] B. Nelson, B. Riching, and R. Black, "Using a custom-built HDL for printed circuit board design capture." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2012.

[5] J. Bachrach, D. Biancolin, A. Buchan, D. W. Haldane, and R. Lin, "JITPCB," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 2230–2236.

[6] "Stanza language," http://lbstanza.org/, accessed: 2020/09/30.

[7] "JITX.com," https://www.jitx.com/, accessed: 2020-09-30.

[8] "SkiDL," https://github.com/xesscorp/skidl, accessed: 2020-09-30.

[9] "PCBDL," https://github.com/google/pcbdl, accessed: 2020-09-30.

[10] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "The racket manifesto," in *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[11] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin Hochstadt, "A programmable programming language," *Communications of the ACM*, vol. 61, no. 3, pp. 62–71, 2018.

[12] "Racket programming language," https://racket-lang.org/, accessed: 2020/09/30.

[13] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, "Replace: Advancing solution quality and routability validation in global placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 9, pp. 1717–1730, 2018.

[14] C. Sechen, *VLSI placement and global routing using simulated annealing*. Springer Science & Business Media, 2012, vol. 54.

[15] H. Abelson and G. J. Sussman, *Structure and interpretation of computer programs*. The MIT Press, 1996.

[16] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows." in *ELPUB*, 2016, pp. 87–90.

[17] "FreeRouting," https://github.com/freerouting/freerouting, accessed: 2020-09-30.

[18] D. Crocker and P. Overell, "Augmented BNF for syntax specifications: ABNF," RFC 2234, November, Tech. Rep., 1997.

[19] G. D. Plotkin, "A structural approach to operational semantics," Computer Science Department, Aarhus University Denmark, Tech. Rep., 1981.

[20] G. Winskel, *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[21] "Functional pictures," https://docs.racket-lang.org/pict/, accessed: 2020-09-30.

[22] T. F. Chan, J. Cong, T. Kong, and J. R. Shinnerl, "Multilevel optimization for large-scale circuit placement," in *IEEE/ACM International Conference on Computer Aided Design. ICCAD-2000. IEEE/ACM Digest of Technical Papers (Cat. No. 00CH37140)*. IEEE, 2000, pp. 171–176.

[23] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1228–1240, 2008.

[24] J. Lu, P. Chen, C.-C. Chang, L. Sha, J. Dennis, H. Huang, C.-C. Teng, and C.-K. Cheng, "ePlace: Electrostatics based placement using Nesterov's method," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2014, pp. 1–6.

[25] J. Lu, H. Zhuang, P. Chen, H. Chang, C.-C. Chang, Y.-C. Wong, L. Sha, D. Huang, Y. Luo, C.-C. Teng *et al.*, "ePlace-MS: Electrostatics-based placement for mixed-size circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 5, pp. 685–698, 2015.

[26] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[27] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[28] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2018.