Echidna: Mixed-domain Computational Implementation via Decision Trees

Devon J. Merrill
Computer Science and Engineering
UC San Diego
devon@ucsd.edu

Jorge Garza
Computer Science and Engineering
UC San Diego
jgarzagu@eng.ucsd.edu

Steven Swanson
Computer Science and Engineering
UC San Diego
swanson@ucsd.edu

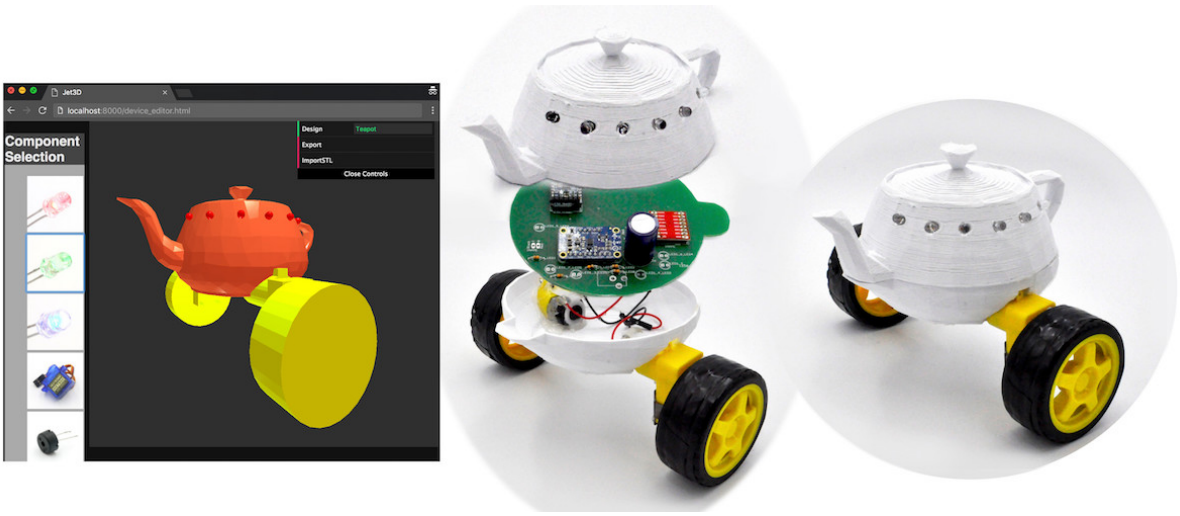

Figure 1: Overview of Echidna.
Users start with a functional, graphical specification (left). Echidna creates ready to fabricate manufacturing files. The user then assembles the fabricated components via generated assembly instructions into a complete device (right).

ABSTRACT

Custom mechatronic devices offer personalized functionality, but also come with many non-functional requirements that are unfamiliar to those inexperienced with electronics such as current draw and servo power. The Echidna prototype system enables non-electrical engineers to move from conception to implementation with their mechatronic ideas by generating and searching through a design space that automatically fills in supporting requirements, such as PCB placement and wiring, around their functional specification. The space is modeled as a decision tree whose root is the user's list of lights, motors, sensors, and other functional components that need to be connected, powered, and controlled. Once found, a complete and valid design can be used to synthesize geometry for 3D printing, circuits, and firmware resulting in a set of "plug and fabricate" files for creating their device. We demonstrate how Echidna realizes several designs and discuss how it can be further customized to task-specific applications.

SCF '19, June 16–18, 2019, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-6795-0/19/06.
https://doi.org/10.1145/3328939.3329004

CCS CONCEPTS

• **Applied computing** → **Computer-aided design**; • **Hardware** → **PCB design and layout**.

KEYWORDS

Fabrication, synthesis, computational design

ACM Reference Format:
Devon J. Merrill, Jorge Garza, and Steven Swanson. 2019. Echidna: Mixed-domain Computational Implementation via Decision Trees. In *SCF '19: Symposium on Computational Fabrication (SCF '19), June 16–18, 2019, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3328939.3329004

1 COMPUTATIONAL IMPLEMENTATION

The "Maker" movement has made it fashionable to design and build your own gadgets, contraptions, clothing, *objets d'art*, and household articles. Often these gadgets and devices combine computational, electronic, and mechanical aspects. Hobbyist makers must coordinate mutually dependent design problems across these three domains. This can be prohibitive for inexperienced makers who do not possess a broad set of technical skills.

For experienced designers, the process of "filling in the gaps" between a functional specification and a working device employs well-known implementation techniques. The tedium of applying procedural knowledge can be a distraction from the end goal of the system and introduce unnecessary human error.

1.1 Computerized tools for design and engineering

Conventional computer-aided design (CAD) tools focus on the "how" of a design. That is, how will a device fulfill its functions? The answer to this question is a set of *technical constraints*. Users of CAD tools are expected to know how a device will work and specify it exactly. Conventional CAD tools manage, analyze, validate, and accelerate specifying these technical constraints.

On the other hand, computational design tools facilitate specifying and exploring the "what" of a design. That is, what will the device do? What will be the functions of the device? These are *functional constraints* to contrast the technical constraints of a design.

In our view, computational implementation tools are a different class of computerized tools. Computational implementation tools take an incomplete set of constraints – functional, technical, or both – and outputs a more complete set of technical constraints. Ideally, the output constraints are complete enough that the device can be created without further user input.

For example, consider the functional specification given by the 3D model in the left of Figure 1. The designer wants to build a two-wheeled teapot-shaped device. The functional constraints are the shape of the robot and the placement of some functional components. The specification details how the mechatronic device should function: light up and wheel around. It does not provide the information required for the assembly and control of the device.

The user wants the device on the right, a fabricated, functional artifact. A traditional CAD workflow would have an engineer (or several) choose and connect the internal components, specify the circuit board placement and routing, prepare the 3D printing files for a case of the correct shape, and code an API and firmware. With this complete specification, the parts of the device can be assembled, printed, and programmed to create the physical artifact.

Our computational implementation system does the work of that engineer by accepting, as input, the functional specification on the left of Figure 1 and outputting ready-to-go manufacturing and programming files.

We do not consider our system to be a computational fabrication system because the fabrication happens offsite, using conventional methods. However, many of the generated manufacturing files are loaded directly into computer-controlled machines (CNC machines) without further human intervention.

How our system relates to several computational design, fabrication, and implementation systems are discussed in Section 7.

1.2 Project Overview

As discussed, our system is an automated implementation system. It takes a high-level specification and outputs files that are ready to be sent to a manufacturer. If we consider the manufacturer to be part of the system (the files can be sent directly by the system), the output of our system is finished – or at least ready-to-be-programmed – devices.

This automated implementation system is modular. These are the main components: a specification interface, a high-level synthesis engine, and low-level domain-specific processors. These components can be swapped out, removed, or replaced to produce devices of different levels of completion from specifications with different levels of abstractness. This paper focuses on a specific embodiment of the system which takes a graphical device specification and outputs a complete electro-mechanical device implementation with example firmware.

Specification interface. The intent of this paper is to focus on the computational implementation side of our toolchain. However, we will briefly discuss two graphical constraint specification frontends to give context to the implementation backend.

We have two graphical interfaces that we have used with our system, one two-dimensional and one three-dimensional. Each graphical interface creates a functional specification with two types of constraints: geometric (shape) constraints, and functional component placements.

The two-dimensional interface specifies a circuit board shape and the location of components on that circuit board. This interface has some elements of a computational design system in that it gives guidance to the user with regards to invalid placement and the like. Figure 6 show some devices designed with this interface. This interface has been used with our system by first-quarter university students to create approximately 200 electro-mechanical devices over several years in an introductory electronics course. Please see our previously published experience report for further details on this course and use of this GUI in an educational context [Merrill and Swanson 2019].

The three-dimensional interface is similar. Except, instead of a 2D PCB shape, a 3D "shell" is specified and components are placed on and around it in three dimensions. Unlike the 2D GUI, the 3D interface has not yet been tested "in the wild." Figures 1, 4, and 5 show the 3D GUI.

The output of these interfaces is a design specification which includes a 2D or 3D geometric shape specification and a set of functional components (possibly with parameters, like color or speed) arranged in space.

High-level synthesis engine. This paper focuses on our system as a computational implementation system, not as computational design. So while the specification interface must be present, the main focus of this paper will be on the process of transforming the functional specification provided by the user interface into a set of technical constraints that are ready to manufacture.

The first step of this process is to iteratively satisfy the constraints required by each element of the user's functional specification. This is done by abstracting the functional elements as atomic components with interfaces. Each interface either must or may be connected to another type of interface. Finding a set of connections such that each required interface is connected in a valid way is the first (and most difficult) step of automated implementation.

Section 2 formalizes our approach to this problem and Section 3 describes our approach to solving this problem as search over decision trees.

Low-level domain-specific processors. After the high-level connections are made and the component list has been finalized, there are several different domain-specific processes that produce the final, low-level technical specifications of the design.

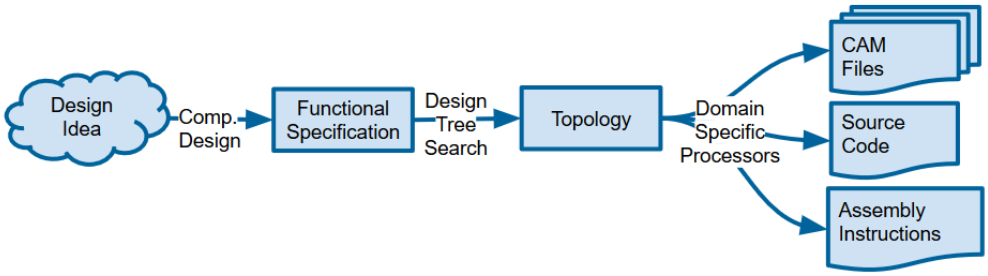

Figure 2: Computational design and implementation flow.
Using computational design, users create a specification from their design idea. Our automated system uses design tree search to generate a topology for the design. Then domain specific processors create a complete technical specification.

These include PCB place and route, API generation, and 3D printing file preparation. Our system uses either very simple, commercial, or open source solutions for these processes and they are discussed briefly in Section 4.4. Special consideration is given to the problem of API generation for arbitrary mixed-domain topologies in Section 4.5.

2 MULTI-DOMAIN DESIGN AS SEARCH

Echidna models computational design as a search over a tree of partial designs. Search starts at the root – a design that embodies the designers intent but lacks the internal parts and connections required to function.

Child nodes are designs that are the product of *transformations* – connecting interfaces of components that define a relationship between the components ranging from electrical connectivity to physical attachment. Echidna expands and searches the tree until it arrives at a completed realizable design.

The following section first describes Echidna's approach and then formalizes our notion of a design, the components it contains, the connections between those components, and the transformations that operate on them. We begin with an overview of how these parts fit together and an example to illustrate the formalism we will use.

2.1 Overview

Echidna designs are collections of components with connections between them. We consider mechanical components such as a leg, wheel, drive train, or button; electrical components such as microcontrollers, batteries, motor controllers ICs; and software components such as control APIs and software libraries.

The designer describes the device they wish to build by providing a list of functional components of the device and describing its physical shape. For instance, the functional components of an alarm clock would include the display, a speaker, several buttons, and shape for the clock.

Figure 4 shows Echidna's graphical user interface (GUI) for specifying a design. For simplicity, the interface assumes there is a single 3D "body" that represents the shape of the design. The designer can then attach components to the body. We refer to these end components as functional components.

After each desired functional component is connected to the body then extra components need to be added, such as a microcontroller or drivers. These extra components are the internal components that enable correct operation for each of the functional components added to the body.

Echidna specifies the relations between components as connections between component interfaces. The complete design requires connections between the interfaces of both functional components and internal components. For instance, the output interface of the power supply would connect to the power input interface of the microcontroller. Likewise, the display's "mounted-to" interface might connect to the "mounted-on" interface of the clock's case. The designer specifies some of these relationships using the GUI (such as the location of the display), but many are left for Echidna to create as it searches the space of designs.

We distinguish between interfaces that must be connected for the component to function (the *required interfaces*), and others. We refer to the optional interfaces as *provided interfaces* because they typically provide a capability that another component needs. The microcontroller's power input interface is required, but its general purpose input/output (GPIO) interfaces are provided.

Our search-based design system creates a complete design by starting with an incomplete design and then adding components and connections until no required interfaces remain unconnected.

2.2 Designs

Our algorithm operates on trees of *designs*. A design is a graph of components and the connections between the components. Each edge in the graph represents the connection of two components via specific interfaces. Two components may have more than one edge if the more than one pair of interfaces connects the components.

Our model considers several special types of designs:

Initial design specifications. An initial design specification is the entry point into an exploration of the design space. All other designs are derived from the initial design. Initial designs contain only user added components.

The range of possible initial design specifications defines the design space available to the user.

Incomplete designs. Incomplete designs have at least one component with an unconnected required interface. Echidna may further explore these partial designs by applying transformations. After

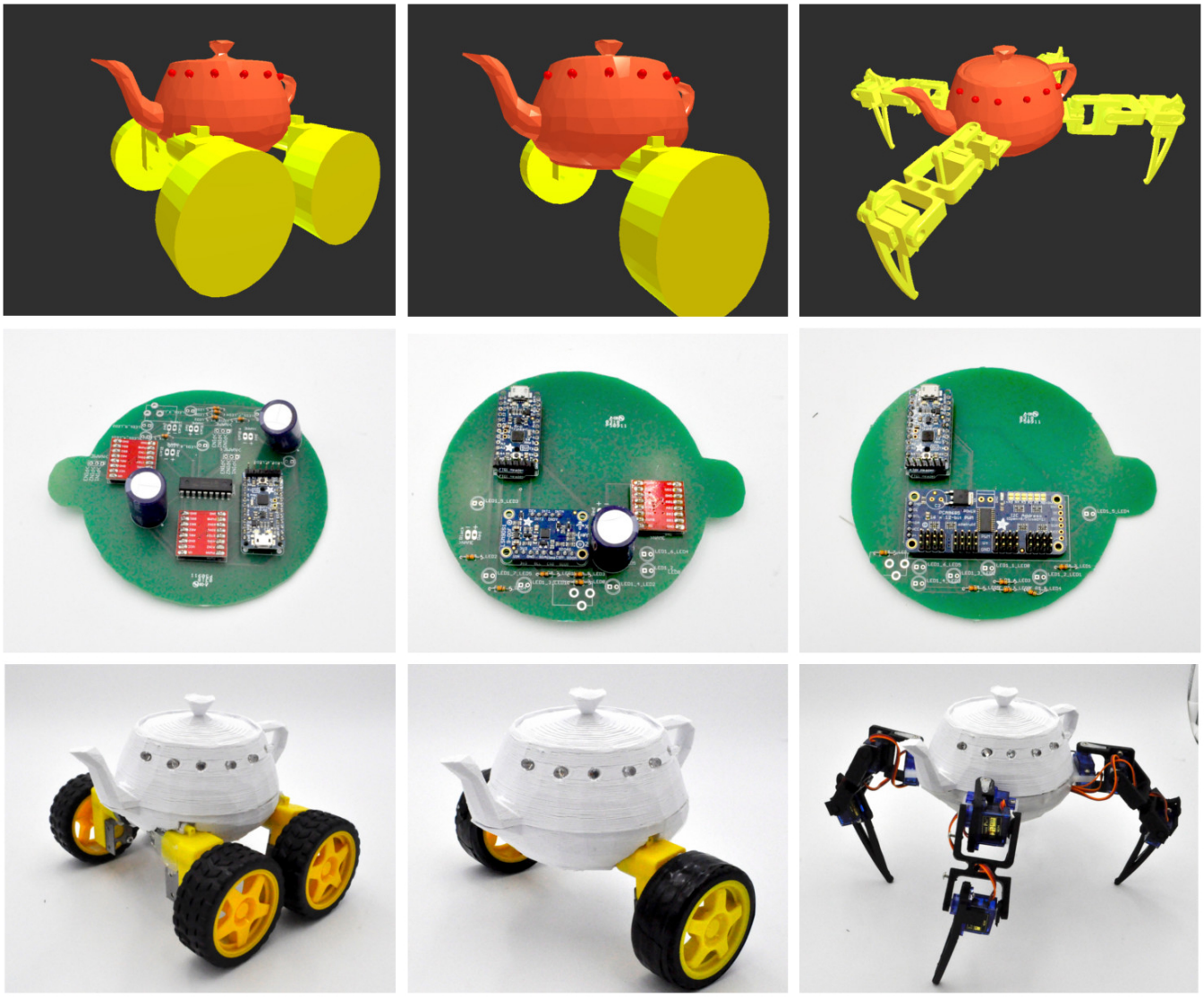

Figure 3: Synthesized robots.
These graphical initial design specifications (top row) are presented with their corresponding synthesized circuit boards (center row) and fabricated designs (bottom row).

a series of composed transformations, an incomplete design may become a complete design.

Complete designs. Complete designs do not contain any components with unconnected required interfaces. When a complete design is found, the search algorithm can terminate.

Dead-end designs. Dead-end designs are incomplete designs that Echidna cannot transform into a complete design through any allowed transformation. Dead-end designs root a sub-tree in the design tree that does not contain any complete designs.

If no transformations are possible on a design, then it is a leaf of the design tree and it is *trivially dead-end*.

2.3 Components

Components represent atomic units of a design.

For example, the Echidna library offers components such as buttons, lights, speakers, wheels, legs, knobs, sensors, microcontrollers, motor drivers, power supplies, resistors, capacitors, software libraries, and control APIs.

Some components add functionality to a design; a motorized wheel component adds mobility. Other components support the functional components; a motor driver component is necessary for the motorized wheel to function.

Users add functional components to an initial design specification to define their design intention. Echidna adds internal components to a design to support the functional components.

A component is represented by a set of internal parameters, a set of provided interfaces, and a set of required interfaces.

Echidna components contain mechanical, electronic, and software aspects. For example, a robotic end effector component incorporates a mechanical model of the physical structure, a kinematic model of the actuator, an electronic interface for the actuator, and a positioning API. These aspects of the component are exposed to Echidna as the interfaces and parameters.

2.4 Interfaces

Interfaces connect components. They represent the requirement or ability to provide electronic signals, mechanical attachment, software function arguments, and the other ways that the components interact. Echidna expresses interfaces by a set of parameters, a satisfaction predicate, and a component transformation.

The parameters of an interface holds information needed to determine compatibility with other interfaces. This information may include size and shape, voltage and current ranges, input/output direction, location, and other necessary information.

The satisfaction predicate is a function that evaluates to true when a valid connection is possible between the interface and another interface in the design. When the satisfaction predicate is true for two interfaces, the interfaces satisfy each other. For example, the satisfaction predicate of a 5 V, 100 mA power input will evaluate to true for a 5 V, 150 mA power supply. But, it will evaluate to false for any 4 V power supply, any power supply with capacity less than 100 mA, or any interface that is not a power supply at all.

An interface's component transformation alters the component that contains it. Component transformations are triggered when an interface is connected. For instance, connecting a serial interface may mean that a GPIO interface is unavailable. Or, connecting a mechanical interface may change the needed actuation torque parameter of a motor.

Specifically, two things may change in a component through component transformation. First, the number and type of unconnected interfaces may change. Second, the parameters of the component may change.

We require that previously connected interfaces remain connected in the component, but any unconnected interfaces may be removed by the component transformation. In addition, the transformation may add new, unconnected interfaces to the component.

Hierarchical interfaces. We have found that composing interfaces provides a useful method for creating new interfaces for new components. For example, many interfaces (such as a full-duplex serial port) are made of a set of single wire digital I/O connections. We specify the higher-order interface (the serial port) as a collection of lower-order interfaces (the digital I/O wires). This can lead to the situation where the same low-order interface is used by more than one higher-order interface. If any of an interfaces sub-interfaces are unavailable for connection (perhaps due to previously being connected) then the whole higher-order interface is also unavailable.

2.5 Design transformations

A design transformation returns an output design which contains the same functional elements as the input design. The output design also contains exactly one more connection than the input design and at most one more component.

With this type of transformation it may be possible to find a series of transformations that, when composed, transform an initial design specification into a complete design that retains all the function components of the initial design specification.

A design transformation is the application of two component transformations when making a connection. We allow one of the two components to be a new instance of a component from the library.

3 DESIGN TREE SEARCH ALGORITHM

Given an initial design comprised of functional components, Echidna's goal is to find a corresponding complete design. It achieves this starting at the root design (the initial design specification), and expanding the tree by adding components and making connections.

Below, we discuss the design tree in more detail, describe the basic search algorithm we use, and the heuristics that allow it to complete in a reasonable amount of time.

3.1 Design Tree

A *design tree* is a decision tree. Each node in the tree represents a (potentially incomplete) design, and the decision made at each node is how to transform the node's design to produce a new (child) design. Section 2.5 describes the two transformations we allow: connecting interfaces on two components or adding a new component and then connecting one of its interfaces to an existing component.

The leaves of the tree are designs that cannot be further transformed. There are two reasons this occurs. First, if the design's components have no unconnected, required interfaces; the design is complete. Second, if we cannot connect any interfaces between the design's components *and* adding an additional component from the library cannot change this property, then the design is a dead end.

The large number of possible transformations make design trees extremely large. Consider a design with r required interfaces from its included components, p provided interfaces from its included components, and l provided interfaces from library components. This design may have $r(p+l)$ children in the worst case. As r and p are roughly proportional to the number of included components, $|C|$, and l is roughly proportional to the number of components in the library $|L|$, we can also say the number of children is on the order of $|C|^2 + |C||L|$. For our examples (Section 5), Echidna finds complete designs at depths around 400 that have up to 400 components. At that depth, the expected fanout leads to an effectively infinite number of designs.

3.2 Search algorithm constraints

The properties of design trees mean that many conventional tree search algorithms will not work. Depth-first search fails when it descends down a fruitless subtree (for example, one that includes an ill-advised transformation early on). The tree's high fanout makes

enumerating even the second level of the tree intractable. This prevents naive breadth-first search.

Instead, an efficent computational implementation system needs an algorithm that meets two requirements. First, it should avoid the need to enumerate all the children of any design. Second, it must detect and avoid fruitless subtrees.

To achieve these goals, Echidna adopts a frontier-based, heuristic search algorithm. Echidna updates the value of the heuristic for each design as the search progresses.

The heuristic is a weighted sum that combines estimates of the design's cost, the expected difficulty in completing the design, and the estimated likelihood that the design is the root of a fruitless subtree.

We describe each of these components in detail below.

Cost. The cost factor models the designer's preference for cheaper, simpler designs. Echidna calculates design cost, cost(D), as the sum of the monetary costs of the components in the design. The cost function may also encode hand annotated preferences of the designer. For example, a lower cost for components that are spatially located close to each other; this would represent a preference for compact designs.

Completion. Our computational implementation system measures the difficulty to complete a design using the quantity and types of unconnected interfaces. The value accounts for required and provided interfaces differently.

Intuitively, more unconnected required interfaces means the design is far from completion. Therefore, designs with more unconnected required interfaces will be more difficult to complete. Furthermore, some required interfaces are more difficult to satisfy than others. For instance, an I2C bus connection is harder to satisfy than a general-purpose IO (GPIO) connection because the I2C bus must connect to particular pins on a microcontroller, while any pin will work for GPIO. In this case Echidna assigns a higher *complexity* to the I2C interface.

Unconnected provided interfaces, on the other hand, make a design easier to complete because they represent unused "features" of the design's components. The higher the complexity of the unconnected, provided interfaces, the more valuable they are.

To compute the net impact of unconnected interfaces, Echidna computes the sum of the complexity of for all unconnected interfaces of a design D as

$$\text{completion}(D) = \sum_{p \in P} \text{complexity}(p) - \sum_{r \in R} \text{complexity}(r) \quad (1)$$

where P is the set of provided interfaces of the components of D and R is the set of required interface of the components of D.

The complexity of the interfaces is determined partially by humans. Lower-order interfaces are hand-annotated in the library based on the presumed rarity of compatible interfaces. The complexity of higher-order interfaces is automatically determined by summing the complexities of lower-order interfaces. Intuitively this makes sense, more complex (hierarchically) interfaces have higher complexity ratings.

In regards to electrical interfaces, we have seen that a default value of 1 for base interfaces works well, with a value of 2 for all other interfaces. Because most pins are dual function, multi-function pins are automatically assigned an appropriate complexity value due to hierarchical specification.

Dead-end heuristic. The final factor in Echidna's design evaluation estimates whether a design is in a fruitless sub-tree. The heuristic assumes that the more trivially dead-end descendents a node has, the more likely it is that the node is a dead-end node. Our estimate for a parent design, D, is updated each time a new design is found in the sub-tree with D at its root. To calculate the dead-end heuristic, we keep track of the proportion of trivially dead-end descendents of each node. The dead-end heuristic, dead(D), is the ratio of the number of trivially dead-end descendents of D to the total number of descendents of D visited so far.

Here, the value of dead(D) influences the exploration of the children of D which, in turn, influences dead(D). In practice, this has the effect of encouraging the exploration of designs without switching, until a dead-end child is found. Then, that design is avoided, until all other sibling designs have a similar number of explored dead-end children.

Combined design heuristic. The full selection heuristic is

$$h(D) = \alpha\text{cost}(D) - \beta\text{completion}(D) + \gamma\text{dead}(D) \quad (2)$$

where α, β and γ are weights for the sub-heuristics.

We find that these weights should be set such that $\gamma >> \beta >> \alpha$. This allows search to abandon bad sub-trees quickly. Ties are broken in favor of the deepest design (that is, the ones farther from the root) and then by cost.

3.3 Search algorithm

Echidna's search algorithm is a frontier-based search algorithm similar to A*. It maintains a frontier of candidate designs. Initially, the frontier contains only the root.

At each step, we choose the best design, D according to Equation 2, and try to connect an unsatisfied required interface on one of its components to an unused provided interface on another. When making this connection, we prefer to connect the required interface with higher complexity, and provided interfaces with lower complexity.

If we cannot make a connection among the existing components, we expand the set of candidate provided interfaces to include the interfaces to components in the library. If we find a component that provides an interface that can satisfy another interface in the design, we add the component and make the connection. If several components provide the interface, we choose the cheapest one. We then add the new design to the frontier.

If the new design is complete, we return that design.

If no connections can be made, D is a dead end, and we remove it from the frontier and update the count of dead descendents for each of its ancestors.

If the frontier is not empty, we select the new best design, and repeat the process.

3.4 Evaluation

We do not claim that this algorithm is necessarily optimal. However, using heuristics, it is able to quickly and automatically create

topologies for a variety of the class of devices we are interested in. Also topologies are created in a way that guarantees that they are correct by construction (assuming a bug free library). These properties make decision-tree search a good choice for our computational implementation system.

Our experiences have indicated that this algorithm generated reasonable topologies; the topologies generated by this search algorithm are very similar to those generated by a human. For example, the three devices shown in Figure 6 have the same topologies as those generated by a human, given the same constraints (and allowing for variation from swapping equivalent pins). Further evaluation of algorithm performance is given in Section 5.

4 IMPLEMENTATION

We implemented design tree search as the back end of our full system that allows novice users to create a variety of mechanical and electronic devices without design experience in these domains.

4.1 System architecture

Our system is composed of several, quite different, parts.

In production, our web-based GUI runs on Google's App Engine and is written in Python and Typescript.

While the GUI is in use, geometric information passed to servers to check overlap and other quick-to-computer feedback.

When a use submits a design for implementation, the specification (geometric constraints and component locations) are pushed to a job queue. A bank of servers poll this queue, waiting to start the more computationally intensive task of implementation. These servers require several minutes of setup time so must be aggressively autoscaled if demand is expected..

When a job is pulled, design tree search (implemented in Python) is used to finalize the component list and the connections between the components. This creates the device topology. After design tree search, the device topology is handed off to file converters that reformat the topology into forms which can be fed through the diverse set of domain specific processors used. These include PCB files (EAGLE sch and brd files), Jinja template files for source code compilation, svg graphics and html files for assembly instructions, and other. The glue code that ties these processors together is implemented in Python.

The domain specific processors then work to create the final technical specification of the device. The most computationally intensive of these processes is PCB place and route. Once all the final technical files are created, they are compressed and uploaded to a cloud database and a link is given to the user to download the implementation. At each stage of the process the user is updated on the current stage of implementation. If a stage fails, that stage can send a human-readable error message back to the user interface.

4.2 Graphical user interface

Our system begins with a high-level graphical specification created through a drag-and-drop interface.

Three-dimensional devices. Users begin with a basic three dimensional model that serves as the body of the design. From a pallet, users add functional components such as lights, motors, legs, buzzers, sensors, and so on. Some components must be directly attached to the exterior of the body. Others may be placed in space around the body. Still others, such as internal sensors, are not spatially placed by the user but are added by selection only.

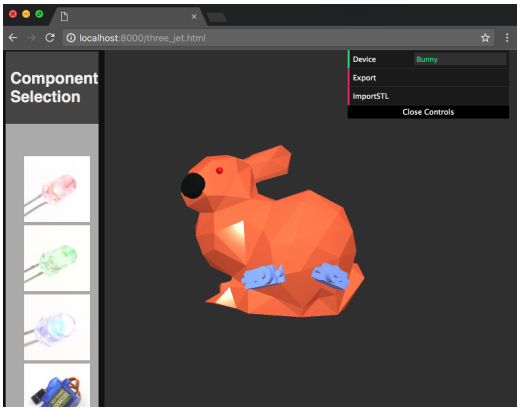

Figure 4: GUI for specifying 3D devices.

The user also selects a cross section of the body that forms the location and shape of the internal circuit board.

The graphical user interface can also act in a circuit board only mode which enables the user to graphically place functional components directly on a circuit board to create devices (Figure 6).

Two-dimensional devices. Similarly to the three-dimensional version, users drag and drop components from a library onto their device. However, instead of a cross-section of a 3D form providing the PCB shape, the PCB geometry is specified directly. Designs created with this GUI do not require a 3D printed component. The devices shown in Figure 6 were designed with the 2D GUI. This version of the GUI has been used by novices in a classroom setting to create simple electronic devices and robots [Merrill and Swanson 2019]. The 2D GUI and computational implementation system as used in the course is available at `http://robots.gadgetron.build`.

4.3 Component library

We use a component library with over 300 different components. These components include mechanical components such as robotic legs, sections of robotic arms, motorized wheels, and buttons; electrical components such as lights, microcontrollers, and sensors; and software components such as control APIs.

To use these components in designs, our system needs a variety of information about each component. This information typically includes a subset of the following: 3D models, mechanical drawing, electrical schematics, source code snippets, weight, assembly instructions, and price information. In addition, each component entry contains detailed interface information that implements the interface objects defined in Section 2.4.

Limitations our of library are discussed in Section 6.

4.4 Domain specific implementation

Once Echidna creates a multi-domain design topology, the topology is processed by several domain specific procedures. These procedures involve completing templates with information from the design topology, collating and converting files, and processing with commercial software.

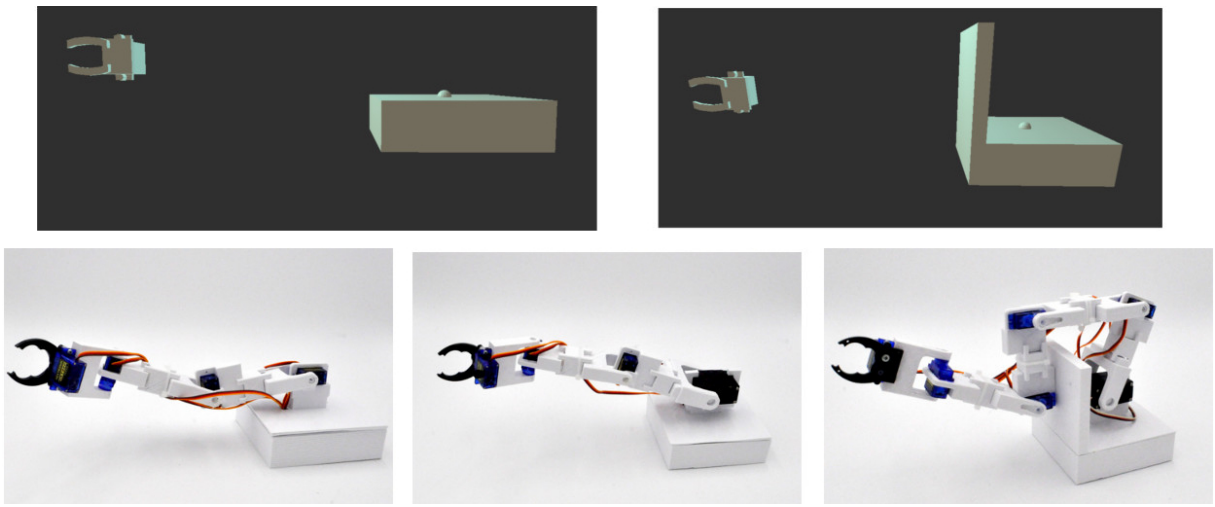

Figure 5: Synthesized robotic arms.
Initial design specification for base configuration and heavy-lifting configuration (top left), initial design specification for obstacle configuration (top right), fabricate base configuration (bottom left), fabricate heavy configuration (bottom center), and fabricate obstacle configuration (bottom right).

3D printed components. Echidna modifies the 3D printable shell with mounting holes and brackets that are specific to the type and location of each components. Echidna creates mounting holes by removing material in the shape of the component. For some components, especially actuators, Echidna adds material to act as mounting substrate. 3D printing files are prepared and modified using OpenSCAD and Ultimaker Cura, both open source software.

Printed circuit boards. Echidna creates a circuit schematic for the electronic components via the design topology. From the schematic, Echidna creates a board layout using a simple packing algorithm (greedy linear search). The outline of the circuit board is the shape of the internal wall of the shell at the user-selected cross-section (less any interfering mounted components). The electrical connections on the circuit board are routed using Autodesk EAGLE, a commercial autorouter. If placement or routing fails, a warning is displayed to the user. This warning makes the suggestion to increase the empty space on the board by increasing the size of the device. We have found that very few (<3%) of student created design are not able to be automatically placed and routed [Merrill and Swanson 2019]. These failures could be address by integration with superior autorouters, but EAGLE was chosen due to its ubiquity and open file format.

Source code. Echidna generates firmware source code through parameterized templates. Each controllable pin of the microcontroller or single-board computer is bound to a variable in a template. These variables then parameterize software object constructors based on information from the design topology. These objects form a user-friendly API for writing custom programs. Echidna also creates an example program which cycles through the basic functions of each component present in the design topology. The example program may be used as a basic test of the device or as a starting point for a custom program.

4.5 Generating APIs for complex topologies

Echidna generates control APIs for components that require the coordination of several other components. Some component interfaces specify API calls and their parameters. During synthesis, Echidna tracks these parameter symbolically. These components also specify how the API functions they require relate to the API functions they provide, defining a transfer function.

For example, a microcontroller's transfer function may take a `pin_1(voltage=5)` constraint and transform it into a `digitalWrite(pin_1, HIGH)` API call. This transform may seem trivial, but these simple transfer function can be automatically combined to create API for arbitrary device topologies.

When Echidna connects a component, Echidna parameterizes that component's transfer function with the constraints of the previous components. This creates a new system of (more complex) constraints that Echidna propagates further.

If the final constraint system is purely geometric, specialized inverse kinematic solvers are optimized for the problem. Mixed geometric and logical constraint systems can be solved by general constraint satisfaction tools such as dReal [Gao et al. 2013] to provide API implementations.

In Echidna, we have successfully used both the symbolic computation package SymPy [Meurer et al. 2017] and dReal for API implementations and connection type checking. Echidna stores both variables and composed constraints symbolically in the components' parameters (Section 2.3). The component transformations can then access these symbols during design topology synthesis.

The key feature of our approach is that it is domain and topology agnostic. While, for example, digital control and kinematic control are well studied, and systematic methods have been developed to solve problems in each domain, our system of composing constraints allows each domain to be solved in the same process,

with arbitrary topologies, without human intervention. This obviates the need to code specialized control and solvers for each new component and domain.

Kinematic control example. As an example, a robot arm's end effector (Figure 5), could require a `move_to(x,y)` API where `x` and `y` are absolute coordinates. If the end effector is attached to a series of arm segments, then each segment adds mechanical constraints to the inverse kinematic system that the movement API must solve.

Again, Echidna generates the API by composing the transfer functions of each component.

A single segment's transfer function relates the inputs, x^{start}, y^{start}, and r to the outputs x, y by the equation:

$$(x, y) \in \{(a, b) | a = x^{start} + l \times \sin(r), b = y^{start} + l \times \cos(r)\} \quad (3)$$

where r is the segment actuator's rotation, l is the segments length, and (x^{start}, y^{start}) is the position of the segment's start.

For a chain of segments, let $(x_k^{start}, y_k^{start})$ be the location of the start of a segment k and (x_k^{end}, y_k^{end}) be the location of the end of segment k, and r_k be the actuator rotation for segment k and l_k be the length of segment k.

The constraints in an n-segment chain are described by this non-linear system (equations 4 through 6).

$$(\mathrm{x}, \mathrm{y}) = (x_1^{end}, y_1^{end}) \quad (4)$$

$$\bigwedge_{i=2}^{n} (x_{i-1}^{start}, y_{i-1}^{start}) = (x_i^{end}, y_i^{end}) \quad (5)$$

$$\bigwedge_{i=1}^{n} \left[(x_i^{end}, y_i^{end}) \in \{(a, b) | a = {x_i}^{start} + l_i \times \sin(r_i), b = {y_i}^{start} + l_i \times \cos(r_i)\} \right] \quad (6)$$

Here the transfer function of each mechanical component transforms the propagated constraints in two ways. First, it adds a spatial constraint to the starting end of the attached component (Equation 5). It also appends the trigonometric constraints to the system with a conjunction (Equation 6).

When it comes time to make an API call, `move(x,y)`'s parameters are set and the rest of the free variables can be solved for via inverse kinematics (or an error is raised if the system cannot be solved).

Digital control example. As another example, an LED component may require a `blink_once()` API. If the LED is connected to the microcontroller (MCU) through an IO expanders, the LED cannot be manipulated directly by the microcontroller. Echidna generates this API implementation by composing the functions that relate the inputs and outputs of the microcontroller, the IO expander, and the signal inputs of the LED.

This example follows the generation of the `blink_once()` API during synthesis as an ordered list of constraints.

The value of the symbolic constraints in each partial design are formatted as:

```
1: component.interface(key=value, ...)
```

In the initial design specification, the LED's API constraint is unmodified. The initial ordered list of symbolic constraints is:

```
1: LED.anode(state=DIGITAL_HIGH)
2: generic.API(method=delay, parameter=onPeriod)
3: LED.anode(state=DIGITAL_LOW)
4: generic.API(method=delay, parameter=offPeriod)
```

Echidna adds a GPIO expander and connects the expander's `out_1` interface to the LED's `anode` interface:

```
1: expander.out_1(state=DIGITAL_HIGH)
2: generic.API(method=delay parameters=onPeriod)
3: expander.out_1(state=DIGITAL_LOW)
4: generic.API(method=delay parameters=offPeriod)
```

The IO expander's transfer function determines how to expand constraints 1 and 4, where `0x7E` is `LED_IC`'s I2C address, `0x01 0xFF` is the command to bring `out_1` high, and `0x01 0x00` is the command to bring `out_1` low:

```
1:  expander.I2C_data(type="addr", data=0x7E)
2:  expander.I2C_data(type="data", data=0x01)
3:  expander.I2C_data(type="data", data=0xFF)
4:  expander.I2C_data(type="end")
5:  generic.API(method=delay, parameter=onPeriod)
6:  expander.I2C_data(type="addr", data=0x7E)
7:  expander.I2C_data(type="data", data=0x01)
8:  expander.I2C_data(type="data", data=0x00)
9:  expander.I2C_data(type="end")
10: generic.API(method=delay, parameter=offPeriod)
```

Echidna adds a microcontroller (MCU) and connects the I2C bus interface to the expander.

```
1:  MCU.I2C_data(type="addr", data=0x7E)
2:  MCU.I2C_data(type="data", data=0x01)
3:  MCU.I2C_data(type="data", data=0xFF)
4:  MCU.I2C_data(type="end")
5:  generic.API(method=delay, parameter=onPeriod)
6:  MCU.I2C_data(type="addr", data=0x7E)
7:  MCU.I2C_data(type="data", data=0x01)
8:  MCU.I2C_data(type="data", data=0x00)
9:  MCU.I2C_data(type="end")
10: generic.API(method=delay, parameter=offPeriod)
```

The MCU's transfer function transforms the symbolic constraints to call to the MCU's API interface:

```
1:  MCU.API(method=beginTransmissionI2C, parameter=0x7E)
2:  MCU.API(method=writeI2C, parameter=0x01)
3:  MCU.API(method=writeI2C, parameter=0xFF)
4:  MCU.API(method=endTransmissionI2C()
5:  MCU.API(method=delay, parameter=onPeriod)
6:  MCU.API(method=beginTransmissionI2C, parameter=0x7E)
7:  MCU.API(method=writeI2C, parameter=0x01)
8:  MCU.API(method=writeI2C, parameter=0x00)
9:  MCU.API(method=endTransmissionI2C)
10: MCU.API(method=delay, parameter=offPeriod)
```

The API constraints are transformed into source code by a parameterized template:

```
blinkOnce(onPeriod, offPeriod) {
    beginTransmissionI2C(0x7E);
    writeI2C(0x01);
    writeI2C(0xFF);
    endTransmissionI2C();
    delay(onPeriod);
    beginTransmissionI2C(0x7E);
    writeI2C(0x01);
    writeI2C(0x00);
    endTransmissionI2C();
    delay(offPeriod);
}
```

5 RESULTS

To demonstrate our computational implementation system we have used it to design and fabricate several devices: a small teapot-shaped

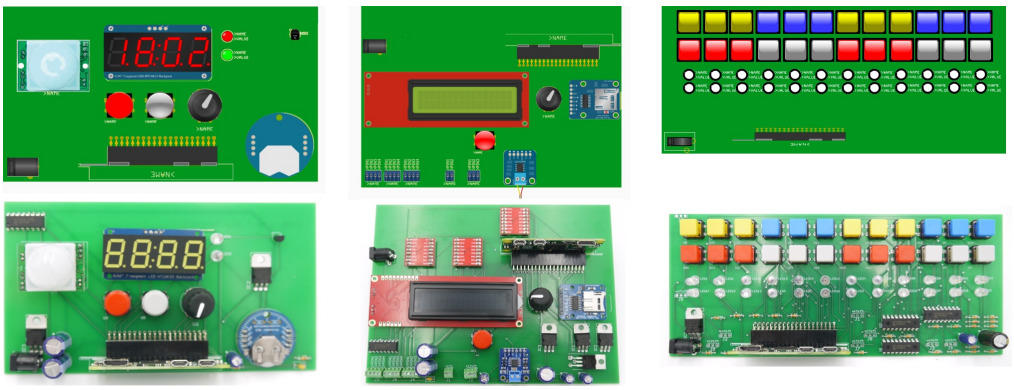

Figure 6: Synthesized circuit boards.
These fabricated circuit boards (bottom) are paired with how they appear in the 2D version of our GUI (top). They include a thermostat (left), a 3D printer control board (center), and a 2x12 music sequencer (right). Sequencers 4x12 and 8x12 not shown.

robot to demonstrate how changes to the mechanical design components can affect the electronic and software internal components; a robotic arm to demonstrate how design tree search can find solutions to mechanical constraints; finally, a selection of embedded device control boards to demonstrate how design tree search scales to hundreds of components. We implemented Echidna's search algorithm in Python.

5.1 Teapot robots

We have designed three configurations of a self-contained, motive robot (Figure 3). Each configuration has a different method of motion specified in its initial design specification. The initial specification also includes six LED lights for each configuration.

These robots have in common several internal components added through design tree search. These are a microcontroller unit, a battery, and six current limiting resistors for the LEDs. In addition, Echidna adds several other internal components to each configuration, including control APIs.

Wheeled configurations. Two of the configurations use motorized wheels. For each pair of wheels, Echidna adds a motor driver board and a large capacitor.

The configuration with four wheels is short on GPIO pins from the microcontroller. In this configuration, Echidna automatically adds and configures an IO expander component.

The configuration with two wheels does not need an IO expander. However, a design with only two wheels cannot satisfy the four wheel control API. Instead, Echidna adds a self-balancing two-wheel API. The self-balancing API component has a tilt sensing required interface, so Echidna adds an IMU.

Legged configuration. This configuration does not require motor driver boards. Instead, Echidna selects a servo control board and gait control API to satisfy the leg components.

5.2 Robotic arm

This example of a robotic arm and gripper demonstrates how geometric and mechanical constraints can be satisfied by design tree search (Figure 5). In each of the three configuration, a base and gripper are added as design components. The gripper is not mounted to the base, and our system must complete the robotic arm. The synthesized mechanical components differ in each configuration, but the synthesized internal electronics are similar to the legged teapot robot.

Base configuration. In this configuration, design tree search first finds the required interface on the gripper and matches a robotic arm segment. This new segment introduces another similar required interface. The final link (to the base) is completed by one last segment.

Heavy lifting configuration. In this configuration, we increased the weight of the end effector as a parameter of the component specification. When a new arm segment is added, the segments weight adds to the arm's total weight. This combined weight is propagated to the required interface of each new segment.

When we increased the weight of the end effector, Echidna selects a more powerful motor for the final end segment to lift the increased propagated weight.

Designed around obstacles. In this configuration, we modify the base such that the direct path from the end effector to the attachment point on the base is blocked. Here, Echidna designs the arm around the obstacle. The weight of the additional arm segments causes the last segment to again use a more powerful motor.

5.3 Embedded circuit boards

When run in circuit-board-only mode, our system allows users to place design components directly on a circuit board. This is useful to demonstrate our system's scaling properties. Using this mode,

Table 1: Topology synthesis characteristics of example circuits.

	Thermo-stat	3D Printer	Sequencer 2x12	Sequencer 4x12	Sequencer 8x12
Design elements	12	18	50	98	194
Internal elements	9	19	39	99	189
Connections	50	101	97	200	406
Search time	**0.3s**	**0.9s**	**2.9s**	**8.9s**	**43.7s**

we designed five devices. These are: a thermostat, a 3D printer control board, and three configurations of a musical sequencer board (Figure 6). The three configurations of the music sequencer use different numbers of rows of 12 buttons and lights.

The thermostat, printer controller, and two row sequencers were fabricated and appear in Figure 6 along with how they appear in the 2D version on our GUI.

Table 1 lists the design characteristics of the five designs. We obtained these synthesis results on a system with a 2.7 GHz Intel Core i5 processor. As the number of design elements increases, so does the required synthesis time.

Thermostat. The thermostat's user added components include a real-time clock, an occupancy sensor, and a temperature sensor. Echidna adds and connects several power regulation components, signal filtering components, current limiting resistors, and an analog to digital converter (ADC). In this design, Echidna's heuristics select the correct path through the design tree without backtracking and completes in less than one second. Here, synthesis time is dominated by start up overhead.

3D printer control board. The 3D printer control board's user added components include several motor control hookups, a thermocouple hookup, several high-current control hookups, and an SD card reader. Echidna adds and connects several motor drivers, high current transistors, power regulation and filtering components, current limiting resistors, and an ADC.

Sequencers. The sequencers include a large number of input components (buttons) and output components (LEDs). These necessitate the addition of several digital IO expander ICs.

As the number of buttons and lights increases, more IO expanders are required. This necessitates exploring and discarding many dead-end subtrees, as our search algorithm prefers designs with fewer components. The smaller designs complete in several seconds. For the largest design with 383 components, the synthesis time is 44 seconds.

6 LIMITATIONS OF THE LIBRARY

To develop, use, and evaluate our tool, we created a library of components and interfaces. As we have described, this library includes electronic, mechanical, and software components and interfaces. Our experiences with the tool have shown that our component library is sufficient to create interesting devices. Still, there are many limitations to the current library. In the current system, the library is the primary limiter of the system's usefulness.

For example, the teapot robot: if a user were to place four wheels in a row, then the current system would not know to add a balancing system to the robot. The four-wheel API and control system (as currently implemented) does not require this constraint. We believe that such a constraint *could* be implemented using our interface system; the robotic arm example shows that the implementation system is able to handle moderately complex geometric constrains.

As an other example, the currently the user cannot specify degrees of freedom of the robotic arm. If a suitable component existed in the library (it does not), then the implementation system might connect the pincer directly to the base using a component with zero degrees of freedom (a solid beam). We believe that a degree-of-freedom constraint *could* be added using our interface system; such a constraint is very similar to those used by electrical components that require specific ranges of voltage and current.

There are a large number of such example. We believe that we have demonstrated that our computational implementation system reasonably general and fast for our use cases (small embedded devices). However, our tree-based implementation algorithm (specifically the heuristics) were developed and tested using our library. If the library were expanded, extended to other domains, or if the components were made to be significantly more complex, then further heuristics and connection algorithms may need to be develop.

Anecdotally, we have found that library creation, maintenance, and debugging quickly require the largest portion of effort as we add capabilities to Echidna and other computational implementation systems that we have developed. We feel that this issue poses the largest barrier to the creation and adoption of similar computational implementation tools, especially in an academic context.

7 RELATED WORK

Our approach to computational design and implementation spans computational geometry, robotics, electronic design automation (EDA), and programming languages. In the following section we detail our contribution to computational design and fabrication by comparing it to related works.

Library-based synthesis. Component-based synthesis is the construction of a network of elementary components, given a library of components such that the constructed network behaves in the intended way. The set of components chosen, and the connections between them constitute a *topology*.

The problem of synthesis from a component library has been studied in the context of software [Lustig and Vardi 2009; Pneuli and Rosner 1990; Solar-Lezama et al. 2006] in particular, with research emerging in the mixed hardware-software domains [Iannopollo et al. 2017; Ramesh et al. 2017a]. The complexity of the components in the library can make it difficult to reuse the computational techniques provided by a design tool. For example, Schulz et al. [2014] system in *Design and Fabrication by Example* requires CAD models that are detailed down to the screw. Our algorithm uses a library of individual components that are described using Python.

Design for Fabrication. Computational design and implementation systems exist in many domains. Given a design, they fill in the gaps that stand between the specification and an artifact. Implementation is often framed as a task-specific problem and as a result, many systems in this genre target a narrow domain *e.g.* plush toys [Bern et al. 2017], furniture [Li et al. 2015; Song et al. 2017], clothing [Saakes et al. 2015], mechanical objects [Bharaj et al. 2015; Zhang et al. 2017], or trigger-action circuits [Anderson et al. 2017].

Echidna brings a domain-agnostic approach to implementation that enables the creation of complex design topologies from components that span the mechanical, electrical, and software domains.

Embedded electronics domain. EDASolver [eda 2014] is a tool that connects complex components and interfaces, but cannot infer

when additional components are needed to satisfy the requirements of a system. Echidna is able to add many components if required to complete a design.

Just in Time Printed Circuit Board (JITPCB) [Bachrach et al. 2016] synthesizes high-level specifications into circuit netlists. Like our system, the JITPCB uses EAGLE for PCB routing. Integral to JITPCB is number of slave microcontroller that make up a required chain-of-stars topology. Our system instead allows any topology that can be generated by connecting interfaces.

Embedded Design Generation (EDG) [Ramesh et al. 2017b] tackles synthesizing small embedded circuits. EDG solves for the components needed in a circuit from constraint descriptions. Solving the constraints for even small designs can take hours or days. EDG uses the Z3 [De Moura and Bjørner 2008] constraint solver. Our system, using design tree search, significantly out performs EDG without requiring library pruning for acceptable runtimes.

Trigger-Action Circuits (TAC) [Anderson et al. 2017] synthesizes both small circuits and firmware with an eye towards novice prototyping. This work enables electronics novices to specify an "if this then that" circuit that senses a stimulus from the environment and executes an action. For example, "if the temperature is above 72 degrees, then turn on the fan". The system lets the user specify just the functionality using a graphical programming interface and then synthesizes an implementation along with instructions. TAC is important in that they incorporate behavior into the computational design process; this work does not address many computational implementation questions. We see TAC as a possible design frontend for Echidna's fast implementation backend. TAC uses breadth-first search which does not scale to large designs and libraries.

Mixed-Domain Computational Design. Recent computational design and implementation projects focus on creating devices which incorporate mechanical, electronic, and software elements. Some of these focus on a narrow class of devices such as walking robots [Coros 2013; Megaro et al. 2015] or multicopters [Du et al. 2016].

These systems maintain disparate topologies for the mechanical, electrical and software components of a design. There are strong constraints in the mechanical domain, no topology for the electrical components, and a fixed software topology. In contrast, Echidna generates multiple- and mixed-domain topologies from nil.

Canedo et al. [2013] propose a method for inferring energy transfer components in physical modeling languages. This technique is able to infer abstract design topologies (such as an abstract pipe to transfer hot air or a wire to transfer current). It focuses on generating comprehensible descriptions, not fabricatable implementations. In contrast, Echidna generates concrete topologies and fabricatable implementations from a much larger library of components.

REFERENCES

2014. EDAsolver - Automatic component selection and pin matching. (2014). http://edasolver.com/ Accessed: 2018-04-05.

Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2017. Trigger-Action-Circuits: Leveraging Generative Design to Enable Novices to Design and Build Circuitry. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 331–342. https://doi.org/10.1145/3126594.3126637

Jonathan Bachrach et al. 2016. Jitpcb. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE.

James M. Bern, Kai-Hung Chang, and Stelian Coros. 2017. Interactive design of animated plushies. *ACM Transactions on Graphics* 36, 4 (2017), 1âĂŞ11. https://doi.org/10.1145/3072959.3073700

Gaurav Bharaj, Stelian Coros, Bernhard Thomaszewski, James Tompkin, Bernd Bickel, and Hanspeter Pfister. 2015. Computational Design of Walking Automata. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation (SCA '15)*. ACM, New York, NY, USA, 93–100. https://doi.org/10.1145/2786784.2786803

Arquimedes Canedo, Eric Schwarzenbach, and Mohammad Abdullah Al Faruque. 2013. Context-sensitive synthesis of executable functional models of cyber-physical systems. *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems - ICCPS '13* (2013). https://doi.org/10.1145/2502524.2502539

Stelian Coros. 2013. Computational Design and Motion Control for Characters in the Real World. *Proceedings of Motion on Games* (2013). https://doi.org/10.1145/2522628.2541251

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Tao Du, Adriana Schulz, Bo Zhu, Bernd Bickel, and Wojciech Matusik. 2016. Computational multicopter design. *ACM Transactions on Graphics* 35, 6 (Nov 2016), 1âĂŞ10. https://doi.org/10.1145/2980179.2982427

Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2013. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *Proceedings of the 24th International Conference on Automated Deduction (CADE'13)*. Springer-Verlag, Berlin, Heidelberg, 208–214. https://doi.org/10.1007/978-3-642-38574-2_14

Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. 2017. *Constrained Synthesis from Component Libraries*. Springer International Publishing, Cham, 92–110. https://doi.org/10.1007/978-3-319-57666-4_7

Honghua Li, Ruizhen Hu, Ibraheem Alhashim, and Hao Zhang. 2015. Foldabilizing furniture. *ACM Transactions on Graphics* 34, 4 (2015). https://doi.org/10.1145/2766912

Yoad Lustig and Moshe Y. Vardi. 2009. Synthesis from Component Libraries. *Foundations of Software Science and Computational Structures Lecture Notes in Computer Science* (2009), 395âĂŞ409. https://doi.org/10.1007/978-3-642-00596-1_28

Vittorio Megaro, Bernhard Thomaszewski, Maurizio Nitti, Otmar Hilliges, Markus Gross, and Stelian Coros. 2015. Interactive design of 3D-printable robotic creatures. *ACM Transactions on Graphics* 34, 6 (2015), 1âĂŞ9. https://doi.org/10.1145/2816795.2818137

Devon J. Merrill and Steven Swanson. 2019. Reducing Instructor Workload in an Introductory Robotics Course via Computational Design. *Proceedings of the 50th ACM Technical Symposium on Computer Science Education - SIGCSE 19* (2019). https://doi.org/10.1145/3287324.3287506

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103

A. Pneuli and R. Rosner. 1990. Distributed reactive systems are hard to synthesize. *Proceedings 31st Annual Symposium on Foundations of Computer Science* (1990). https://doi.org/10.1109/fscs.1990.89597

Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, BjÃűrn Hartmann, and Prabal Dutta. 2017a. Turning coders into makers. *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication - SCF 17* (2017). https://doi.org/10.1145/3083157.3083159

Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. 2017b. Turning coders into makers: the promise of embedded design generation. In *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication*. ACM, 4.

Daniel Saakes, Hui-Shyong Yeo, Seung-Tak Noh, Gyeol Han, and Woontack Woo. 2015. Mirror mirror: an on-body clothing design system. *SIGGRAPH 2015* (2015). https://doi.org/10.1145/2785585.2792961

Adriana Schulz, Ariel Shamir, David IW Levin, Pitchaya Sitthi-Amorn, and Wojciech Matusik. 2014. Design and fabrication by example. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 62.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *ACM SIGPLAN Notices* 41, 11 (2006), 404. https://doi.org/10.1145/1168918.1168907

Peng Song, Chi-Wing Fu, Yueming Jin, Hongfei Xu, Ligang Liu, Pheng-Ann Heng, and Daniel Cohen-Or. 2017. Reconfigurable interlocking furniture. *ACM Transactions on Graphics* 36, 6 (2017), 1âĂŞ14. https://doi.org/10.1145/3130800.3130803

Ran Zhang, Thomas Auzinger, Duygu Ceylan, Wilmot Li, and Bernd Bickel. 2017. Functionality-aware retargeting of mechanisms to 3D shapes. *ACM Transactions on Graphics* 36, 4 (2017), 1âĂŞ13. https://doi.org/10.1145/3072959.3073710