Turning *Coders* into *Makers*: The Promise of Embedded Design Generation

Rohit Ramesh
University of Michigan
Ann Arbor, MI, USA
rohitram@umich.edu

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

Antonio Iannopollo
University of California at Berkeley
Berkeley, CA, USA
antonio@berkeley.edu

Alberto Sangiovanni-Vincentelli
University of California at Berkeley
Berkeley, CA, USA
alberto@berkeley.edu

Björn Hartmann
University of California at Berkeley
Berkeley, CA, USA
bjoern@eecs.berkeley.edu

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

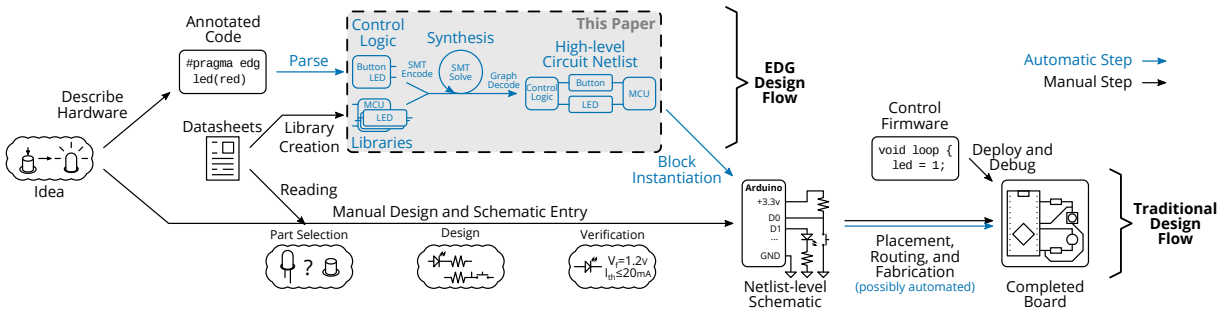


Figure 1: ***The Embedded Design Generation (EDG) Methodology.*** *In contrast, with traditional embedded development methods, which rely on significant user skill, EDG only requires a high-level specification to generate an electrically correct circuit that satisfies user requirements. While EDG is a larger methodology, this paper focuses on the synthesis of high-level block diagrams from user specifications. In particular our prototype only implements those portions of the system that directly interact with the synthesis step. We both manually generate the specification needed, instead of extracting it from existing code, and manually create a circuit netlist from the output block diagram.*

ABSTRACT

As personal fabrication becomes increasingly accessible and popular, a larger number of makers, many without formal training, are dabbling in embedded and electronics design. However, existing general-purpose, board-level circuit design techniques do not share desirable properties of modern software development, like rich abstraction layers and automated compiler checks, which facilitate powerful tools that ultimately lower the barrier to entry for programming, by allowing a higher level of design—separating specification from implementation—and providing automated guidance and feedback. In this paper, we present a novel methodology for embedded design generation that allows the generation of complete designs from high-level specifications. We present an implementation capable of synthesizing a variety of examples to show that our approach is viable. Starting from user-specified requirements and a library of available components, our tool encodes the design space as a system of constraints. Off-the-shelf solvers then reason over these constraints to produce a block diagram containing sufficient information to generate the finalized device firmware, bill of materials, and circuit netlist.

KEYWORDS

Type System, Synthesis, Embedded Design, Makers, Satisfiability Modulo Theorem, Software Defined Hardware

SCF Í7, June 12-13, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4999-4/17/06…$15.0
https://doi.org/10.1145/3083157.3083159

ACM Reference format:
Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. 2017. Turning *Coders* into *Makers*: The Promise of Embedded Design Generation. In *Proceedings of Symposium on Computational Fabrication, Cambridge, MA, USA, June 12-13, 2017 (SCF Í7),* 10 pages.
https://doi.org/10.1145/3083157.3083159

1 INTRODUCTION

Modern software development practices and tools have allowed programmers to become more productive with less effort and less knowledge of low-level details, largely thanks to high-level abstractions, expressive programming languages, and compile-time correctness checks. Similarly, automated layout tools, logic and high-level synthesis, and hardware construction languages have led to analogous improvements in integrated circuit design. However, while board-level design tools have come a long way since the days of pen-and-paper schematics and literal tape-outs, significant knowledge of electronics design, circuit theory, and tool operation is still required to realize a full embedded project.

As physical fabrication costs today are already affordable and continue to fall, we argue that the most significant barrier to the creation of embedded devices is now in the design phase. Students, creative designers, and generally those without electronics training or practice have a difficult path ahead of them when they need to build custom prototypes and proof-of-concept hardware.

At the same time, the maker movement has demonstrated widespread interest in many kinds of small-scale fabrication by non-professionals, including that of embedded devices. Though Arduino, Raspberry Pi, and similar projects have shown that embedded design is approachable by novices [Gibb 2010], they do so only by trading off the flexibility afforded from using discrete components for the encapsulation provided by hardware modules and isolated, non-interoperable ecosystems.

Drawing on inspiration from the software and integrated circuit design communities, similar strategies—namely, higher levels of abstraction, correct-by-construction generation, and automated correctness checks—can be applied to the hardware design space to reduce the skill floor required for embedded design [Mellis et al. 2016]. Furthermore, we claim that tools driven by a computational approach to design will be able to reason in a way that encapsulates low-level detail while retaining the flexibility provided by a wide selection of discrete components.

To that end, we propose a novel methodology, *Embedded Design Generation* (EDG) which exploits advances in constraint solvers to allow the automated generation of functionally correct-by-construction[1] board-level designs from a high level specification.

Tools based on EDG would only require that the user annotate their embedded software with simple requirements and, from that specification, synthesize the final circuit diagram, bill of materials, and firmware. Software APIs, electrical properties of circuits (e.g. Kirchhoff's current and voltage laws), and other low level details are combined with the user input into a system of constraints.

Existing constraint solvers can then generate designs which are functionally correct, electrically sound, and satisfy the user specification. To show that this both works and is computationally feasible, we build a prototype tool and test it with a variety of examples from different domains.

Figure 1 provides an overview of our proposed design flow and compares it to current embedded design practices. EDG abstracts away, through automation, much of the electronics expertise needed for tasks like parts selection, circuit design, and verification. In addition, Figures 2 to 6, 9 and 10 are all a part of a running example where we describe the construction of a simple device with a single LED and button.

[1]We do not consider timing or other performance constraints in this paper as the focus is to empower designers who do not have to produce industrial strength boards.

2 RELATED WORK

EDG builds on prior work in "electronic design automation" (EDA) by specializing the Platform-Based Design (PBD) methodology [Sangiovanni-Vincentelli 2007] for maker-scale embedded development. PBD is a methodology which has been successfully used to create synthesis tools in a number of domains, including integrated circuit (IC) development and automotive engineering. EDA community has incrementally raised the abstraction level of many embedded development tasks and by using insights from PBD and synthesis tools in other domains, we contribute to that progress.

2.1 General EDA

General-purpose board-level circuit design tools have largely not moved beyond graphical schematic capture, where users place electronic components and connect their pins together. In mainstream tools, hierarchical blocks allow some degree of abstraction by grouping low-level components together, but their lack of parameterization limits re-use. Additionally, while electronic design automation (EDA) tools feature electrical verification checks, these are of limited utility to makers. Matrix-based connection legality checks (for example, checking input-output directionality), though ubiquitous in design suites, are rarely used, non-extensible, catch only small classes of bugs, and have a high false-positive rate. Higher-end design suites often feature technologically advanced checks, like electromagnetic compatibility (EMC) or radio frequency interference (RFI), but these generally require significant skill to operate.

There has been some work towards building board design tools better suited for makers. For instance, PHDL [Nelson et al. 2012] is a Verilog-like language for describing netlists that allows some parameterization of blocks and better design entry. However, like Verilog, it is only a static description of a circuit.

JITPCB [Bachrach et al. 2016] takes the concept further and embeds a hardware construction language in a general purpose programming language, allowing circuit generators instead of simple parameterized blocks. However, like PHDL, it does not have a model of the underlying design space, preventing it from catching many errors. JITPCB also does not reason over voltage, current, bandwidth, or other properties needed to perform useful verification of a design, something our tool does.

EDASolver [EDASolver 2016] aims to be a synthesis tool for microcontroller based embedded systems. When given a tree that describes the basic structure of an embedded device, EDASolver can choose specific components to generate a circuit fitting that broad structure. Unlike JITPCB, it does have some understanding of the electrical properties of an embedded system, and can use that to choose valid components from a pool of parts. As EDASolver has neither published source code nor a technical paper, we are unable to fully characterize its capabilities and limitations, but its modeling of electronics does not appear to be extensible beyond voltage and current limits.

While both JITPCB and EDASolver have some ability to choose specific components from vague specifications and automate the

assignment of individual pins, these features are constrained by their inability to reason over the *topology* of a circuit. Our tool, and likely any tool that follows the EDG methodology, is capable of not only choosing components as needed but also adding elements to the topology of a circuit. This means our tool can create new power domains, insert amplifiers and buffers, and infer the need for IO expanders whenever required to create a valid design. Fundamentally, we reason over the space of possible designs and the requirements without the need to tightly constrain the topology of possible solutions. As a result, even our rudimentary tool can compensate for limitations in parts or complexities in a specification in much the same way that an engineer might.

2.2 PBD and Domain-Specific Tools

In Platform-Based Design's (PBD) terminology, our methodology maps user input to a set of library components according to well-defined composition rules that can be verified statically. PBD-based tools solve the synthesis problem by opportunistically *composing* elements from a library to generate systems of constraints which can be solved by external solvers. For instance, METRO II [Davare et al. 2013] allows for general model integration and architecture exploration, where the mapping process between specification and platform is validated through simulation. Likewise, PYCO [Iannopollo et al. 2016] synthesizes a complete specification for a system from a partial set of Linear Temporal Logic (LTL) constraints and a library of components with LTL contracts. Although reminiscent of these techniques, the approach taken for EDG does not require the use of LTL or other logic languages.

Some techniques related to our approach have been also used in program synthesis. Brahma [Gulwani et al. 2011; Jha et al. 2010] synthesizes loop-free programs over bitvectors out of a library of simpler functions. This allows Brahma to generate software from a sparse specification of boolean logic constraints. Gvero et al. [2013] propose the use of types in a program to synthesize valid expressions which are then suggested to the programmer.

Robotics-oriented design tools like EMLab [Bezzo et al. 2015] and ROSLab [Mehta et al. 2016, 2014] solve similar problems to EDG in that domain. EMLab is a block-level development tool for robotic electronics that uses an SMT based verification mechanism similar to our own, however it does not extend that to provide synthesis. ROSLab provides a similar pathway from code to circuitry, however unlike our tool, it is limited to custom-made hardware elements that support their custom chaining protocol. In contrast, EDG works with off-the-shelf electronics in order to reduce the cost of creating a library of parts and enable the fast integration of new components.

Finally, tools aiding interactive device design largely also follow the pattern of automatically figuring out details from a high-level design, albeit in more constrained domains. For instance Midas [Savage et al. 2012] automatically designs capacitive touch layouts given user-specified sensor type, shape, and position. Likewise, PaperPulse [Ramakers et al. 2015] adds interactive electronics elements to paper crafts from a library of widgets.

3 METHODOLOGY

The goal of Embedded Design Generation is to create better abstractions for developing embedded devices and tools that can perform

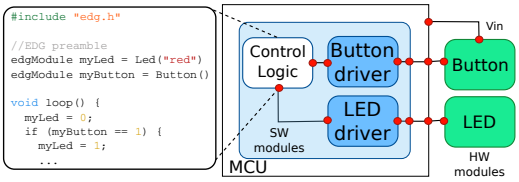


Figure 2: *Code can be a specification for a device.* *The code to the left, the control logic, specifies a device in which a light blinks when a button is pressed. It is also the software that is eventually run on that device, the annotations in the* `EDGPreamble` *specify the hardware infrastructure needed to make the software function as intended. The block diagram to the right describes one possible device that matches that specification, by meeting all the hardware requirements in* and *being able to run the control logic. Due to our focus on the synthesis process, the code shown here is a mockup that shows one possible structure for a specification.*

robust verification of device designs. However, better verification requires our tool to reason about the relationship between hardware and software. Verifying the electrical properties of a thermometer does no good if that thermometer has no way to send its data to the designer's software. Our key insight is that many of the design's hardware requirements are reflected in the code, for example in required libraries and pin assignments. Yet the fundamental logic of the device, how it functions at runtime, is rarely reflected in the hardware. This asymmetry suggests that higher level abstractions for embedded development should be similar to embedded code.

If we want to be able to describe the device at a higher level, we can capture the most important parts of its function and construction in its code. We can specify how the device acts at runtime, as well as the hardware infrastructure needed for the device to function. Figure 2 shows a stylized example of this, where many of the implicit hardware requirements that are expressed in user code are rendered as explicit declarations for a design generation tool to use. The software in figure 2 is a specification for the hardware *and* the runtime operation of the device.

To make this kind of high-level abstraction useful, we must be able to compile it into the firmware and circuitry needed to construct an embedded device. However, firmware and circuit diagrams are too low-level for efficient synthesis. Instead we represent the result as a *block diagram*, like the one in figure 2, which can be easily turned into a final design. Likewise, we need to be able to tell if those block diagrams actually describe correct devices, so that we do not generate broken or invalid designs. A *type system* gives us a way to construct the blocks for real-world parts and an algorithm to decide whether any given block diagram is correct. Finally, we need to choose a single valid device from the space of possible devices. We do this by constructing a *design space model*, which captures a wide range of possible designs in ways that existing constraint solvers can reason about.

The Embedded Design Generation methodology is built around these three core concepts:

Block Diagrams capture the conceptual structure of a device across both hardware and software boundaries, by taking elements of the final design and representing them as blocks with connectivity information. These diagrams are an intuitive yet powerful

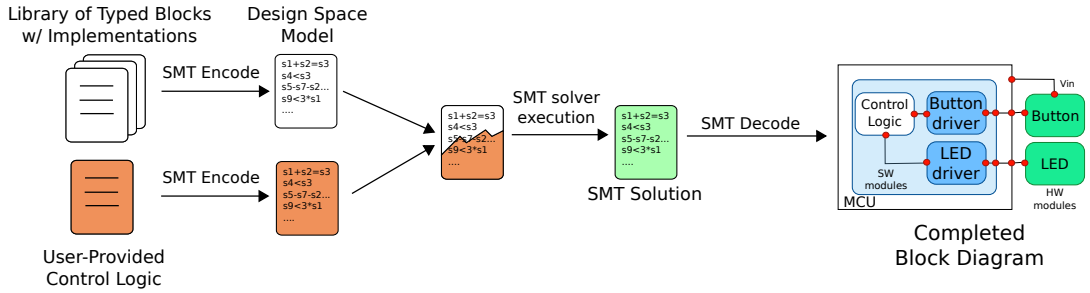


Figure 3: ***Design Generation at a High Level.*** *EDG tools use existing constraint solvers to perform synthesis. The tools convert knowledge about the design space and control logic specification into constraint satisfaction problems whose solutions are block diagrams describing valid device designs. These block diagrams completely specify the design of an embedded device and can be easily converted to more useful formats.*

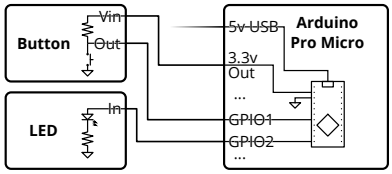


Figure 4: ***Convert a block diagram into circuitry by linking implementations together.*** *Instantiating the block diagram from figure 2 requires taking implementation details associated with each block, in this case a relevant sub-circuit, and connecting them together based on the links between their blocks.*

```
#include "edg.h"

//EDG Preamble
edgModule myLed = Led("red")
edgModule myButton = Button()

void loop() {
  myLed = 1;
    ...
}
```

(a) *Original Control Logic*

```
#include "led.h"
#include "button.h"

//EDG Preamble
var myLed = initLED(GPIO2);
var myButton = initButton(GPIO1);

void loop() {
  myLed = 1;
    ...
}
```

(b) *Instantiated Firmware*

Figure 5: ***Convert a block diagram to firmware by filling in templates.*** *Instantiating software is a simple template replacement operation. Figure 5a is a mockup of user-provided control logic for the device. Figure 5b is the code after we replace the EDG-provided template elements with the concrete implementations provided by other blocks. Note that the code outside of these templates is unmodified.*

model for working with systems, and can capture device structure, resources, and many of the other relationships found between elements of a design.

The Type System defines rules for how we transcribe the real world properties of circuits and software into blocks and their *type signatures*. It also gives us *type checking*, a process that determines whether any block diagram describes a valid device.

The Design Space Model is a system of constraints suitable for general-purpose solvers, built from a library of blocks and their type signatures. This model can then be used to generate a complete, working, block diagram for a device from a specification.

Figure 3 shows our methodology for design generation, which exploits the growing speed and increasing expressive power of constraint satisfaction problem (CSP) solvers. We convert a library of blocks, with corresponding type signatures, into a monolithic set of constraints that model the space of potential designs made up of those blocks. These constraints are then composed with constraints derived from the control logic to produce a CSP whose space of valid solutions is the space of valid block diagrams that meet our specification. We then pass this CSP to the solver and decode the result into a block diagram that will successfully typecheck.

Block diagrams are ideal representations because they are easy to convert into the design files needed to fabricate a device. Figure 4 shows how the final circuit can be created by connecting individual block implementations along the links between them. Similarly, figure 5 shows how the firmware can be instantiated with template replacement operations that pull from code snippets provided by connected blocks.

Block diagrams also work at many levels of fidelity. In general, blocks can be composed of smaller blocks until one recurses down to single instructions or individual circuit elements. Our current tool works with relatively large blocks made up of entire libraries or breakout boards. This allows us to abstract away questions of timing delay, electromagnetic interference, and many other phenomena that become evident at smaller scales. Large blocks also mean there is a smaller space of possible configurations for solvers to reason over, making their immediate use more feasible. As solvers grow faster and more expressive, EDG tools can move to using finer granularity models with smaller blocks.

We structure each block diagram around the notions of blocks, ports, and links. As we have seen, *blocks* represent realizable elements of our final design and each has a number of *ports* which represent specific capabilities, relationships, or resources a block may have. *Links* are then the connections between ports that represent the transfer of resources, usage of capabilities, or other relationships between blocks. For instance, the connection of a serial line or the use of a software API.

A block diagram must have all the information needed to instantiate a device but many parts have properties and settings that are not solely defined by their connections. Consider the LED in figure 4, which could be annotated with information about its color. To allow the block diagram to represent this information, blocks, ports, and links all have *concrete types*, which are structures made up of

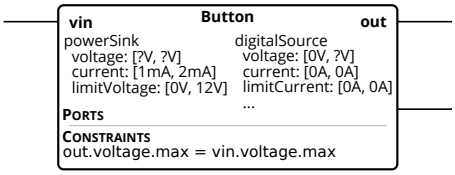


Figure 6: *Type signatures are constraints on their elements.* *While elements of a design are given concrete types, the blocks on their own are usable in a variety of settings. Type signatures are simply the conditions under which a block will work as intended, presented as constraints over the concrete type of a block. A type system is the set of rules for how to map real-world properties into types and type signatures, such that a block diagram which typechecks can be instantiated into a working device. In this case we constrain the expected input and output voltages of a button to be equal, a limitation on the possible concrete types that button may have.*

named primitives—like integers, boolean values, and strings—or nested substructures. Concrete types allow each block to specify the information needed to instantiate it as well as additional properties useful in other phases of design generation.

The block diagram alone is not enough for synthesis, since we require a way to determine whether any given block diagram will result in a valid device. The *type system* gives us a way to generate blocks and their *type signatures*, constraints over those blocks, so that we can check the correctness of an entire block diagram. As in figure 6, type signatures annotate blocks, ports and links with constraints that limit the concrete types they may have within in a block diagram. Then, type checking ensures that each element of the design satisfies its type signature. In section 4.2 we explore how we constructed a type system for our prototype tool that accurately detects and rejects invalid designs with this procedure.

Finally, Embedded Design Generation requires that we are able to turn a library of blocks and type signatures into a *design space model* that our tools can reason over. In practice, we expect this to take the form of a monolithic constraint satisfaction problem to which we can add the specifications, usually in the form of a control logic block, for any particular synthesis task. This single model can then be optimized or added to, as new parts become available or new limitations in the design space are found. We build the Design Space Model by generating a CSP for each block that might be included in an output design, and then adding variables that determine whether any pair of ports is connected. The solver can then choose which connections exist and give us the final block diagram, with valid concrete types for each block, link, and port. While optimizations can be layered over this, we believe that any design space model will have this core structure.

4 EDG PROTOTYPE ARCHITECTURE

Our prototype tool implements the EDG methodology described in the previous section, with a focus on synthesizing devices from relatively large blocks at a level high above individual resistors and ICs. We implemented our tool in Haskell and used Z3 [De Moura and Bjørner 2008] as the underlying constraint solver.[2]

[2]Our code and the results of our experiments are available at https://lab11.github.io/edg-sat-prototype/appendix/scf2017.

```
PORT p:
  used :: Bool:          Indicates whether the port is used in the final design
  connected :: Bool:     Indicates whether the port is connected to another port
  class :: String:       Identifier used to constrain which ports can connect to each other
  type :: [fields]:      List of fields to be translated to SMT variables
  constraints::[expr]:   List of formulas over the type of the port.
                         Must be true for system to typecheck
IMPLICIT CONSTRAINTS:
  connected => used:
   Ensures that a port is part of the design if connected to any other port
  forall c in constraints, used => c:
   Only requires the solver to satisfy the constraints if the port is being used
```

Figure 7: *Ports use implicit constraints to capture connectivity.* *The implicit constraints in each port allow us to relax the constraints on the SMT solver. The first constraint says that if the port is connected to another then the port must be used in the output block diagram. Along with the corresponding constraints from figure 8, this ensures that every element has a flag to show whether it is used in the final design. The second implicit constraint tells the SMT solver that none of the type signature's constraints need to be satisfied if the element is not used , minimizing its work.*

```
BLOCK/LINK b:
  UID :: String:         Unique identifier for the block
  used :: Bool:          Indicates whether the block is in the final design
  ports :: [port]:       List of ports attached to this block
  type :: [fields]:      List of fields to be translated as SMT variables
  constraints::[expr]:   List of formulas over the type of the block and its ports.
                         Must be true for system to typecheck
IMPLICIT CONSTRAINTS:
  forall p in Ports, p.used => used:
   Ensures that a block is part of the design if any of its ports are part of the design.
  forall c in constraints, used => c:
   Only requires the solver to satisfy the constraints if the block is being used
```

Figure 8: *Blocks and links have identical representations in our tool.* *Despite their stylistic differences, both blocks and links capture relationships between ports, along with some internal data and constraints. This allows us to turn connections between modules via links into one-to-one relationships between ports on modules and ports on links, simplifying the process of constructing an SMT problem.*

4.1 Blocks, Links, and Ports

As in the general methodology, our prototype uses blocks, links, and ports to represent possible designs for embedded devices. Figures 7 and 8 illustrate the principal data structures used in our tool. The user provides their input in the form of control logic which we manually encode as a block that must appear in the final design.

4.2 Type Signatures

Each block, link, and port in our library contains a type signature, i.e. a set of bounds on the concrete type an element may have in a valid block diagram. In a block diagram concrete types are data structures composed of named fields, each linked to a value. These values can be boolean, integers, floats, strings, UIDs or another nested set of field-value pairs.

Our tool's internal representation for type signatures is shown in figures 7 and 8. These structures capture all the pieces of information needed to generate the SMT representation of a design element, with the majority of the constraints simply being stored as expressions that translate directly into SMT constraints. As in figure 6 each constraint provides a way to express the ambiguity in a type signature, since each block has many valid concrete types and can therefore work in a variety of different designs. The constraints are arbitrary expressions consisting of boolean expressions,

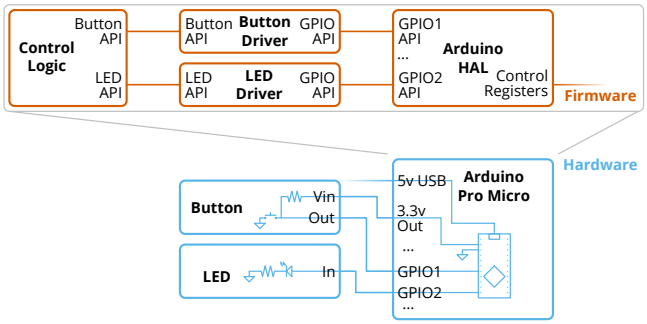


Figure 9: *There are symmetries between the hardware and firmware elements of a design.* *Many of the hardware elements of a design are paired with corresponding firmware elements, as in the case of an LED and its driver code. These pairs tend to have identical structure in both domains, even if there are other components that only exist in one domain.*

ordering operators, linear arithmetic operators, and references to values found in the concrete type of that element.

In other cases the constraints can be used to specify that a value falls in some range, that there exists an equality which must be preserved, or any other condition that is representable as an expression in our solver. These expressions can capture many complex behaviors, like the assignment of pins to functions on a microcontroller, ranges of voltages and currents, and even the nesting of interfaces where our tool has to infer additional parts.

We choose to limit constraints on numerical values to linear arithmetic because non-linear relationships that cannot be conservatively approximated by linear ones are relatively rare given the fidelity of our tool. Since Z3 and other SMT solvers are much slower when working with non-linear constraints, we choose to limit ourselves to the faster option.

4.2.1 Design Space Model. Our prototype naively constructs a design space model from its library of blocks and links. We rely on the fact that constraints in type signatures are almost identical to the equivalent SMT expression.

All the type signature fields in figures 7 and 8 are transformed into sets of constraints. Each flag in the element and value in the type becomes a variable the SMT solver is capable of assigning. Then we add constraints between those variables to match those in the type signatures. From this state, we generate a large adjacency matrix where each cell is an unassigned boolean value that determines whether a particular pair of ports is linked. If ports are linked, their types are set equal and they are marked as being connected. This lets us simulate a one-to-one connection between ports on a block and ports on a link. We then extract this adjacency matrix from the SMT solver's solution and use it to construct the block diagram by walking the resulting graph and recovering each block's concrete type.

4.3 SMT encoding, solving, and decoding

Working from the control logic and the design space model, our tool encodes the complete synthesis problem as a system of boolean and linear arithmetic constraints which are then solved by an SMT

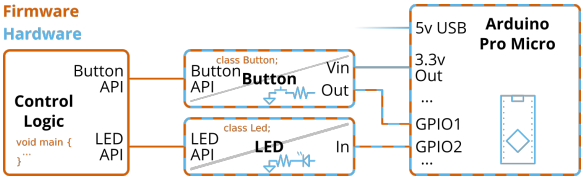


Figure 10: *Our representation with an integrated view of hardware and firmware.* *This combines the circuit design and firmware drivers of each peripheral from figure 9 into a single block. In this case the firmware and hardware blocks for the Arduino, button, and LED are combined into a single block representing each component. Note that pure hardware and firmware elements still do exist.*

solver. Blocks, links and ports are all translated to equivalent SMT constraints as described in the previous subsection. The control logic is treated like any other block and added to the CSP, though with the additional requirement that it be used in the final design. Once the encoding is complete, our tool generates an SMT-LIB v2.0 compatible file which is then passed to the SMT solver. If the solver is able to find a solution to the system of constraints, it is decoded into data structures where all the type signatures have been resolved to concrete types. Finally, the resulting network of blocks is presented to the user as a block diagram describing a device that can run the control logic.

5 TYPE SYSTEM DESIGN

As connection legality essentially drives circuit synthesis, the properties and constraints captured in the type signatures are especially important. In our prototype, we attempt to model the parameters needed to ensure that the circuit is electrically valid, programatically valid, and meets user requirements.

5.1 Software and Hardware Modeling

As a complete embedded design tool, our prototype must model both the hardware (circuits) and software (user code and drivers). While it is common to think of them as completely separate domains, as in figure 9, they are usually heavily intertwined in practice.

Most peripherals ultimately expose a firmware API and most electrical components are controlled to some degree by the firmware. As in figure 10, our representation combines the hardware and firmware domains in ports and blocks when appropriate.

Compared to separate representations, this reduces the number of ports and blocks that the solver needs to search through, improving performance. This combined model accurately represents how many APIs control electrical connections and drivers are usually associated with a device, without additional complicated constraints to tie domains together.

5.2 Ports and Links

As synthesis is interface driven, components are almost completely defined by their ports. Our type system models common electrical ports, including several digital communications networks, as well as arbitrary firmware APIs.

5.2.1 Firmware Ports. Firmware ports define pure firmware interfaces, APIs. They are modeled as either producers or consumers

with a type, like LEDs or temperature sensors, and optional data, like sensor resolution. Ports on the control logic are the starting point for generating a design. Additional constraints prevent hardware referenced by one piece of code from being split between different controllers.

5.2.2 Electrical Ports. Electrical ports define a pure electrical interface, which does not interact with the firmware domain. Our type system only has power ports, which define either an always-on voltage source or device power input.

Power ports capture voltage levels and current flows through a port. Both are modeled as ranges to capture device tolerances, as in the output of a wall wart, and runtime variation, like when an LED is on or off. These represent the full spectrum of expected circuit states during operation.

We also specify voltage and current limits as ranges, where the expected operating range must be contained within the tolerable range. While upper limits are useful for absolute maximum ratings, ranges capture lower limits, like minimum operating voltage or minimum current draws. Despite being a highly conservative model, this encodes the most important information needed for power compatibility checking.

Our current type system gives all components a common ground, so that power ports are single-ended voltage sources referenced to an implicit universal ground. This limitation is mostly for simplicity, but captures most beginner and intermediate designs. Advanced features like isolation domains require additions to our type system.

5.2.3 Controlled Ports. Controlled ports define an electrical port controlled with a firmware API. The simplest example is the microcontroller-driven GPIO, which is described as a digital signal.

Digital ports have all the properties of power ports including the ability to supply power. This models the common usage of microcontroller GPIOs to switch small loads, like LEDs, while generalizing to any controlled load. We also capture voltage thresholds that check both signal level compatibility and thresholds on switched loads.

5.2.4 Digital Communications Ports. Digital communications ports are a variant of the bidirectional digital port for common communications protocols. Many digital communications protocols require multiple wires, which we bundle as a single port. This is for efficiency reasons: all the wires travel together, and modeling each pin as a separate port creates extra connections that increase the search space and hurt synthesis performance. Our type system models ports for several communications buses including I^2C, SPI, and UART. Each bus checks for signal level compatibility as well as bus-specific properties like I^2C address uniqueness.

5.3 Components

5.3.1 Peripherals. Most components representing peripheral devices are structured as adapters that provide one interface and require another in order to function correctly.

One such example is the controlled LED, whose hardware is just the standard LED circuit with a ballasting resistor. We model this as a two-port element: an LED API producer port, and a GPIO consumer port. The GPIO port also models important electrical characteristics like current and voltage limits.

As an adapter-style component, ports on both sides are required to be connected. While a LED without an electrical input is useless, the requirement for an API port prevents synthesis from placing extraneous, unrequested LEDs. Most other peripheral components, like buttons, temperature sensors, or LCD displays, are similar.

True adapter components also exist. The GPIO expander requires a I2C slave connection and provides extra GPIOs. Likewise, a digital amplifier requires a power supply and low-power digital output and produces a new digital output at power supply voltages.

5.3.2 Firmware. Our type system also models pure firmware blocks in the same way. For example, a FAT32 library provides a file system API and consumes a low-level nonvolatile memory API.

5.3.3 Microcontrollers. Microcontrollers are structured differently because they serve as the control source for devices they provide interfaces but do not have requirements aside from power. Otherwise, they are modeled like every other component and are largely defined by their ports.

6 EVALUATION

We create a number of embedded devices by manually generating the control logic block's type signature, using our prototype tool to synthesize a design, and manually instantiating each design to test its correctness. Each device was synthesized with three separate libraries of varying size.

Our full library is used for all examples, except where noted, and consists of these components:

- *Microcontrollers*: Arduino Pro Micro 3.3 V, Arduino Trinket 3.3 V
- *Basic peripherals*: tactile switch with pull-up, LED with resist-or, 12V lit dome switch with pull-up, 12V fan
- *Device peripherals*: Sparkfun 16x2 serial LCDs (3.3 V and 5 V versions), SD card with SPI interface, Sparkfun OpenLog
- *Sensors*: TMP102 I^2C temperature sensor, QRE1113 reflectance sensor with output resistor
- *Interfaces*: I^2C GPIO expander, high-side digital amplifier, TB6612-FNG dual motor driver, L7805 voltage regulator
- *Software*: FAT32 filesystem driver

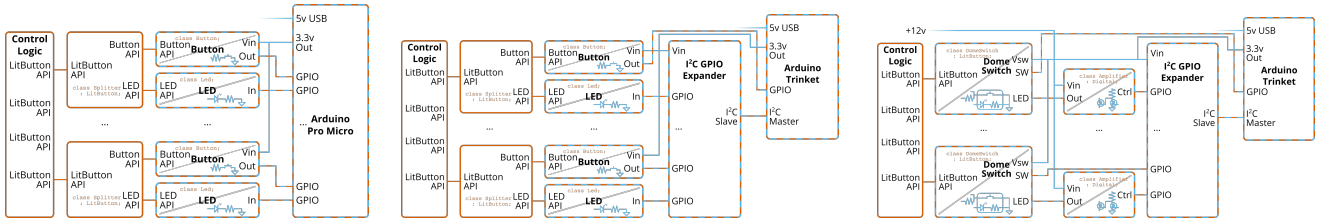
The library contains a total of 73 blocks and links. This is a highly constrained set that is likely not very representative of the libraries any production system would use. However, it should suffice to gain a broad idea of the performance characteristics of our tool and accurately capture how our tool responds to limitations in the library of available blocks.

We first examine a number of simple test cases. Then we look at how our tool responds to restrictions on available parts, both in terms of small changes and instances where the device is radically changed. Finally, we look at the performance of our tool as it synthesized all the designs described.

6.1 Basic Synthesis Tasks

We synthesize a number of simple devices to show the range of domains EDG may be useful for and to analyze the results.

6.1.1 Blinking LED. We synthesize the simple light and button combination that we have been using as a running example. This device has a single LED and a single button that blinks the LED



(a) *Interface Converters* **(b)** *Interface Converters and GPIO Expander* **(c)** *GPIO Expander and Dome Switches*

Figure 11: *Three Versions of Simon.* *These three designs are generated from identical control logic blocks. Each design could be generated from code similar to the EDG preamble found in figure 2, where each required module is turned into a port connecting to that peripheral. Despite their radically different construction these designs are functionally identical, differing in only the size of the buttons and minor timing variations.*

(a) *Simon* **(b)** *Line-Follower*

Figure 12: *Physical realization of Simon and line-following robot.* *The left figure shows the three Simon variants on a large breadboard. The left section is the initial synthesis result (11a), using a microcontroller and discrete LEDs and switches. The center section is the second result (11b), where the system is forced to use a pin-constrained microcontroller and must infer a GPIO expander. The right section is the final result (11c), using inferred digital amplifiers to drive external dome switches. The right figure shows the line-following robot, which successfully generates a circuit that correctly integrates the pre-selected chassis and motors.*

when pressed. The blinking light has long been the "Hello, World" of embedded hardware design and seemed a fitting place to start.

We use this design as an initial test case as we developed our tool and type system because it captures many of the most common constraints in embedded development. Any synthesized design has to keep a coherent chain of control from the control logic to each peripheral, a chain that captures software relationships, hardware relationships, and relationships that jump between those domains. This example also captures basic power management as the tool has to ensure that voltage levels are correct and that current limits are met. Our tools have synthesized many working versions of this device, and those that we constructed functioned correctly.

6.1.2 Temperature Controller. This device is a basic control system where a thermometer reads the local temperature, displays it on a small LCD screen, and runs a fan when it is too hot. Here we capture bus topologies like I^2C, as well as a more complex power system that could supply the 12 V fan, 3.3 V microcontroller, and 3.3 V sensor. Our tools are able to synthesize this device while avoiding pitfalls like mismatched power sources.

6.1.3 Line Following Robot. Our final basic design is a line-following robot designed to stand on a specific chassis with pre-mounted motors. Here we verify that we synthesize a device from a more complete partial design. In this case, we knew both the code we wished to run and the motors we wanted to use, so we specified that all three components must be included in the design. Our tool managed to correctly connect the motors to the microcontroller, including adding motor drivers and the split-level power system needed to use them.

Our tool does not distinguish between being asked to design a device with just the control logic as a specification, three separate blocks that must all be in the design, or some manually-designed critical portion of a device that needs non-critical surrounding infrastructure. This versatility means that in addition to the design process we focus on in this paper, EDG-based tools can support many other forms of interaction.

6.2 Inferring Missing Design Elements

One of the most useful features of the EDG methodology is that it can infer additions to the topology of a device when the available set of parts is limited.

To test this we design a simple datalogger that reads a temperature sensor and writes the result to an SD card. Our first synthesis of this device produced a design that used the OpenLog breakout board, a combination of SD card socket and preprogrammed chip with a simple serial interface for SD card filesystem access

Then we run the synthesis process again after removing the OpenLog from the library of available parts. This time, our tool adds an SD card holder to the device and used a software FAT32 driver to provide a filesystem, showing that our tool can adapt to accommodate constraints in the available pool of parts or non-obvious interactions between interfaces.

6.3 Preservation of Function

The final design we synthesize is our own version of the Simon electronic game. This device flashes four lights in a random order and asks the player to press the corresponding buttons in that same order. Our library supports two major options for synthesizing this design: large buttons with built in LEDs or smaller discrete LEDs and buttons in matching pairs.

Our initial attempt to synthesize this design resulted in a mix of these two options, likely not what a designer would want. We had to add a constraint to ensure that all the buttons used by device had

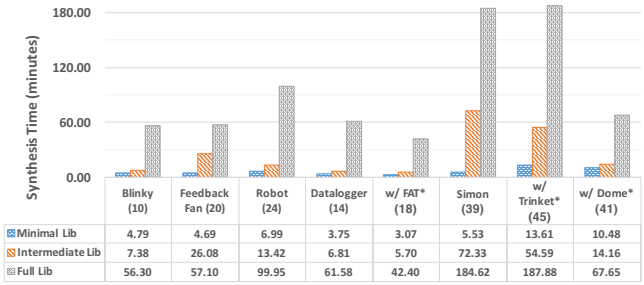
a similar type. The resulting design, shown in figure 11a, consists of four pairs of similar buttons and LEDs and a microcontroller with sufficient IO pins to directly connect all peripherals.

After removing the large microcontroller from the library and leaving only a pin-limited microcontroller, our tool created the design in figure 11b. This design adds a GPIO expander to provide enough pins to control all the peripherals.

Our final change is removing the driver that allowed us to use a discrete LED and button pair as a single lit button. Figure 11c shows the result, and we note that this device shares no parts in common with our original version of Simon—instead accomplishing the same task with a completely different implementation.

We constructed all three versions (see figure 12a) and they functioned identically barring the difference in parts and some drift in the timing. Our synthesis process preserves the key details of our control logic, no matter the components used.

6.4 Performance



	Blinky (10)	Feedback Fan (20)	Robot (24)	Datalogger (14)	w/ FAT* (18)	Simon (39)	w/ Trinket* (45)	w/ Dome* (41)
Minimal Lib	4.79	4.69	6.99	3.75	3.07	5.53	13.61	10.48
Intermediate Lib	7.38	26.08	13.42	6.81	5.70	72.33	54.59	14.16
Full Lib	56.30	57.10	99.95	61.58	42.40	184.62	187.88	67.65

Figure 13: *EDG synthesis time for the discussed examples.* *Each minimal library contains only the blocks necessary to synthesize the design. The full libraries include all 73 of the encoded blocks, except for Datalogger FAT*, Simon Trinket*, and Simon Dome* for which the full library was constrained to make simpler designs impossible. Each intermediate library is approximately half the size of the full library. Times are expressed in minutes and the parentheses next to each name contain the number of blocks and links in each design.*

We provide synthesis runtime data for our designs to show both the feasibility of our methodology and the scaling behavior.

Experiments were run on a server with dual-socket Intel E5-2667 CPUs (3.3 GHz, 8 physical cores per socket) and 192 GB of RAM. Note that the computational time was dominated by Z3 which is single-threaded. All experiments used under 2 GB of RAM. Figure 13 shows the synthesis time for all the experiments. Every design was synthesized using three libraries with different sizes. As expected, our tool performance depends both on the size of the library and the complexity of the solution, represented by the number of blocks used for each design. Larger libraries or solutions usually resulted in longer runtimes. The biggest designs required several tens of components and a runtime up to three hours which is reasonable compared to the time required to perform the same tasks manually. However, we believe that our tool's performance can be drastically improved, as we discuss in the next section.

7 DISCUSSION

We believe that future work will further support our hypothesis that the EDG methodology is a powerful and feasible way to improve embedded development tools.

7.1 Performance and Optimization

Ultimately, the tool we implemented is just a prototype to demonstrate that the EDG methodology is fundamentally feasible. While runtimes for even our limited library of components are not as fast as we would like, there are many possible optimizations.

Performance of existing solvers has improved over the years through advancements in the basic boolean SAT techniques [Heule et al. 2016], and SMT solver theories [Conchon et al. 2017]. Recent years have also shown major improvements in the expressive power of constraint solvers, including techniques like counterexample-guided inductive synthesis [Jha and Seshia 2017] which allow solvers to incorporate new domains of reasoning through a feedback process. Communities in these fields are active and we do not believe this trend will stop anytime soon.

Additionally, we believe we can greatly improve our tool's performance by exploiting a more efficient SMT encoding of the design space model. Currently the design space model is naively translated to an SMT equivalent, without leveraging symmetries, precomputed solutions, or user insight.

Even if, ultimately, full synthesis against a large library is infeasible we believe the EDG methodology still holds promise. The integrated representation of electronic components with firmware drivers eliminates the often-manual step of mapping pins to firmware, while the rich type system allows automated electrical verification to a greater degree than existing matrix-based connectivity checks. Synthesis at a smaller scale is still important, whether for ensuring the thoroughness of the type system, or for automating design within a constrained environment.

7.2 Type System Fidelity

The type system determines connection legality and its thoroughness determines the correctness of EDG's output. While the model for our prototype is largely based on our experiences as embedded designers, a more formal treatment of which properties are relevant is desirable. In particular we would like to develop a formal composable ontology for elements of an embedded system that integrates well with EDG.

Additionally, information from datasheets is insufficient for synthesis, occasionally even contradictory. Strict compatibility checks using datasheet-provided specifications often produces false positive errors. For instance, logic voltage thresholds are usually given as a single value under arbitrary test conditions. Using that value directly would make many reasonable designs unsynthesizable, as the specification conditions are excessive for the digital signaling methods our tool models. Instead, we use less conservative bounds that are accurate given the low frequency conditions our model assumes. However, higher accuracy models would be possible with more precise datasheets, especially if specifications were given as simple mathematical functions.

Our limited library also obviates the need to encode the physical details of components. For example, both a weak indicator LED and

a lighting-grade power LED would satisfy a user requirement for a LED. Additional constraints like brightness, power draw, or form factor are necessary to fully capture user intent.

7.3 Usability

EDG proposes an input very different from the traditional electronics and embedded design flow. While it requires less electronics expertise to build a functional device, a larger question is where best to draw the line between automated processes and user input. Alternatively, hybrid interactive approaches may be desirable. For example, an EDG based assistive tool might ask a designer, "I see your parts are not voltage-level compatible, would you like me to insert a regulator?" Future user studies can illuminate the trade-offs between these different strategies in the electronics design domain.

Embedded hardware development also does not end with board fabrication, and debugging poses significant challenges [Mellis et al. 2016]. EDG's richer data model, containing information like peripheral topology and expected voltages, can support novel assistive debugging strategies. Approaches include automatically generated self-test routines for peripherals or interactive guided debugging.

Finally, a comprehensive, complete set of libraries ultimately forms the basis for EDG. Building such libraries is far from painless, currently involving the manual, time-consuming translation of datasheet specifications to part constraints. Better languages for encoding part data could increase accessibility, while formalisms (like better type systems) can reduce the likelihood of mistakes.

8 CONCLUSION

This paper describes Embedded Design Generation, a methodology for developing usable, maker oriented tools for embedded development. EDG focuses on using existing constraint solvers to search a space of possible designs for systems matching a given specification. We described how this model facilitates the development of tools that can convert software specifications of devices into realizable designs. We also developed a prototype tool built on EDG and showed that it is capable of synthesizing simple devices more effectively than many tools of similar scale. This includes being able to infer complex circuit topologies to manage power, interface expansion, and signaling. Our tool is also able to infer the changes to software structure needed to provide interfaces the user asks for. We synthesized multiple concrete devices and physically built each of them to show the correctness of the design. Our most representative tests took a few hours to synthesize, less time than many novices would take to complete the same task.

While our prototype is not yet a broadly usable embedded development tool, we believe that it sheds light on the capabilities of Embedded Design Generation, reinforcing our hypothesis that existing constraint solvers are approaching the speed and expressive power needed to act as the foundation for a new space of maker-oriented embedded-design tools.

9 ACKNOWLEDGEMENTS

We would like to thank Sanjit Seshia and Mark Horowitz for their insightful advice and feedback over the course of this project, as well Daniel Fremont for his advice on encoding problems for SMT solvers. This work was supported in part by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; the National Science Foundation under grant(s) CPS-1239031; the NSF/Intel Partnership on Cyber-Physical System Security and Privacy under award proposal title "Synergy: End-to-End Security for the Internet of Things: NSF Proposal No. 1505684"; DARPA CRAFT; and ExCAPE: Expeditions in Computer Augmented Program Engineering (NSF grant CCF-1139138).

REFERENCES

Jonathan Bachrach, David Biancolin, Austin Buchan, Duncan W Haldane, and Richard Lin. 2016. JITPCB. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE, 2230–2236.

Nicola Bezzo, Peter Gebhard, Insup Lee, Matthew Piccoli, Vijay Kumar, and Mark Yim. 2015. Rapid co-design of electro-mechanical specifications for robotic systems. In *ASME 2015 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers, V009T07A009–V009T07A009.

Sylvain Conchon, David Déharbe, David M. Heizmann, and Tjark Weber. 2017. SMT-COMP 2016:. (March 2017). http://smtcomp.sourceforge.net/2016/index.shtml

Abhijit Davare, Douglas Densmore, Liangpeng Guo, Roberto Passerone, Alberto L. Sangiovanni-Vincentelli, Alena Simalatsar, and Qi Zhu. 2013. metroII: A Design Environment for Cyber-physical Systems. *ACM Trans. Embed. Comput. Syst.* 12, 1s, Article 49 (March 2013), 31 pages. https://doi.org/10.1145/2435227.2435245

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

EDASolver. 2016. EDASolver: Welcome to Functional EDA. (Jan. 2016). https://edasolver.com

Alicia M Gibb. 2010. *New media art, design, and the Arduino microcontroller: A malleable tool*. Ph.D. Dissertation. Pratt Institute.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. *SIGPLAN Not.* 46, 6 (June 2011), 62–73. https://doi.org/10.1145/1993316.1993506

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. *SIGPLAN Not.* 48, 6 (June 2013), 27–38. https://doi.org/10.1145/2499370.2462192

Marijin Heule, Matti Järvisalo, and Tomář Balyo. 2016. The International Sat Competition Wepage. (July 2016). http://www.satcompetition.org/

Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. 2016. Constrained Synthesis from Component Libraries. In *13th International Conference on Formal Aspects of Component Software (FACS)*. Besancon, France.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. 215–224.

Susmit Jha and Sanjit A. Seshia. 2017. A theory of formal synthesis via inductive learning. *Acta Informatica* (2017), 1–34. https://doi.org/10.1007/s00236-017-0294-5

Ankur Mehta, Nicola Bezzo, Peter Gebhard, Byoungkwon An, Vijay Kumar, Insup Lee, and Daniela Rus. 2016. *A Design Environment for the Rapid Specification and Fabrication of Printable Robots*. Springer International Publishing, Cham, 435–449. https://doi.org/10.1007/978-3-319-23778-7_29

Ankur M Mehta, Joseph DelPreto, Benjamin Shaya, and Daniela Rus. 2014. Cogeneration of mechanical, electrical, and software designs for printable robots from structural specifications. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE, 2892–2897.

David A. Mellis, Leah Buechley, Mitchel Resnick, and Björn Hartmann. 2016. Engaging Amateurs in the Design, Fabrication, and Assembly of Electronic Devices. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems (DIS '16)*. ACM, New York, NY, USA, 1270–1281. https://doi.org/10.1145/2901790.2901833

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

Raf Ramakers, Kashyap Todi, and Kris Luyten. 2015. PaperPulse: an integrated approach for embedding electronics in paper designs. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2457–2466.

Alberto Sangiovanni-Vincentelli. 2007. Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. *Proc. IEEE* 95, 3 (March 2007), 467–506. https://doi.org/10.1109/JPROC.2006.890107

Valkyrie Savage, Xiaohan Zhang, and Björn Hartmann. 2012. Midas: fabricating custom capacitive touch sensors to prototype interactive objects. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 579–588.