Embedded Device Generation: Turning Software into Hardware

Rohit Ramesh
EECS Department
University of Michigan
rohitram@umich.edu

Prabal Dutta
EECS Department
University of Michigan
prabal@eecs.umich.edu

This work was presented at the CrossFAB workshop at CHI 2016.

Abstract
This paper introduces *embedded device generation*, a design and synthesis process that allows anyone who can write code to develop embedded hardware. The key insight is that a simple program can serve as a complete design specification for the embedded device that runs it. We envision future device generation tools that can enable new workflows for makers and provide building blocks that personal fabrication researchers can integrate into their own work. We are currently developing a prototype version of these tools, and we soon hope to solicit feedback from the personal fabrication community.

Author Keywords
Hardware Synthesis; Embedded Systems; Personal Fabrication

ACM Classification Keywords
B.1.m [Hardware]: Miscellaneous; J.6 [Computer Aided Engineering]; 3 [Special Purpose and Application-Based Systems]: Real-time and Embedded Systems

Introduction
Embedded systems are an integral part of personal fabrication, as well as an important component of 3D printing, the Internet of Things, smart infrastructure, robotics, and numerous other technologies. Yet despite their increasing

Figure 1: We envision that our embedded device generation tools will be capable of generating a wide variety of devices including sensors, simple robots, musical instruments, and even handheld games.

ubiquity, many issues compound to make developing embedded hardware a difficult and tedious process. Choosing components for a device requires wading through reams of datasheets and documentation, design tools only provide meager forms of correctness checking, and embedded software development often requires users to reimplement large blocks of code to make tiny hardware changes. These issues increase cost, create barriers to learning, and waste time on tasks that could be automated.

We believe that developing embedded device generation tools will alleviate these issues and allow anyone who can write simple software to design embedded hardware. Our work focuses on automatically generating embedded systems, like those in Figure 1, from programs that describe the their functionality. At their simplest, these programs look like the software written for an Arduino, yet they capture enough information to allow our tools to automatically choose components, wire them together, and generate a complete design for an embedded device. We envision an ecosystem that will allow any programmer to write a program describing a device they need, send the generated design to a fabrication service, and receive a finished device in the mail the next day. There are 18.2 million professional software developers in the world [2] and the number of students learning computer science is growing rapidly, yet these are only the most visible groups with the skills needed to use device generation. We aim to empower students, professionals, hobbyists, the creative, and the curious to make electronic art, automate their homes, experiment with robotics, and design sensors that help them gather information about their world.

Our design for embedded device generation follows the familiar pattern of transforming a hard problem into an easier one, solving the easier problem, and transforming that solution back into a solution for the original problem. To do this, we develop a formal model that represents embedded devices as a network of components, which allows us to tractably reason about the relationships between those components. Figure 2 presents an overview of our system. We first *analyze* a user-supplied program to generate a specification within the formal model, then incrementally *synthesize* a schematic by adding components which bring it closer to satisfying the specification, and finally *reify* that schematic into a complete design for an embedded system.

An Outline for Device Generation

Embedded device generation will enable new workflows for makers, lower the bar for building hardware, and give a vast audience of programmers the ability to immediately jump into embedded hardware development. Imagine a maker who is building a custom setup for home-brewing beer. They have a fermentation barrel and water bath, but need to carefully control the temperature of the water during the brewing process. Instead of buying a costly control system that may not be tailored to their needs, the maker can write a program describing what they want their temperature controller to do. This program—like the one in Figure 3—describes the components the device must have, as well as how those components act as the device runs. Once the program is complete, embedded device generation tools take over, choose parts, verify that they are compatible, and generate a design for the temperature controller. Depending on the maker's exact needs, they can tell the device generation tools to optimize for price, power draw, or some other user-defined criterion. When the generation process is complete, the maker can send the design to be fabricated and will soon receive their custom home-brewing temperature controller in the mail.

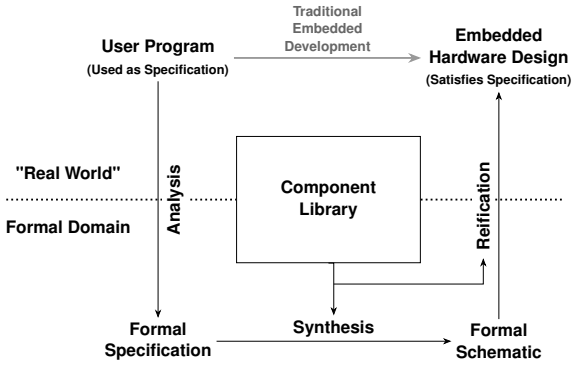

Figure 2: Embedded device generation is a three phase process. First, a user program is *analyzed* to generate a specification. Second, a schematic for the embedded device is *synthesized* using parts from a component library. Finally, that schematic is *reified* into a complete design for an embedded device by replacing components in the schematic with implementations taken from the component library.

Here, we examine the process outlined in Figure 2 in more depth. The initial step, *analysis*, analyzes the program to gather information about the specifications of the device. In its most basic form, this parses the program to retrieve those components which must appear in the final device and converts the execution logic into a component that is inserted into the synthesis process. Eventually tools from program analysis will also extract information about timing and energy consumption in order to more aggressively optimize the design of generated devices. The second step, *synthesis*, searches the space of possible designs for a valid device schematic that satisfies the specification. If performed naively, such a search would be intractable, but we design the formalism to have a number of symmetries that significantly prune the search space. The final step, *reification*, completes the process by converting the device schematic into a complete design for an embedded device. The resulting software and hardware designs will then be passed to existing tools that compile the generated device firmware and automatically create a PCB.

```
// Component Declarations
component thermometer = new Thermometer(
  immersion, min-temp <= 0c,
  max-temp >= 100c);
component heater = new Heater(immersion,
  power > 10w);
component cooler = new Cooler(immersion);
component status = new RGB-LED();

// Activity Description
fn main(){
  while(true){
    if(thermometer.temp() < 16c){
      // Water temp too low
      cooler.off();
      heater.on();
      status.setColor(Blue);
    } else if(thermometer.temp() > 20c){
      // Water temp too high
      heater.off();
      cooler.on();
      status.setColor(Red);
    }else{
      // Water temp just right
      heater.off();
      cooler.off();
      status.setColor(Green);
    }
  }
}
```

Figure 3: This program acts as a self-contained specification and implementation for a temparature controller. The initial section acts as a declaration of the components the controller must contain, as well as the properties those components must have. The second section is a decription of how the device should act, written as if the declared components had a well-defined generic software API. The program, as a whole, acts as a specification that embedded device generation tools attempt to satisfy when generating a device.

Design Considerations

In order to be successful, embedded device generation tools need be easy for novices to learn and powerful enough for advanced users. Thankfully, the computer science research community has significant experience designing and using programming languages. Embedded microcontrollers like the Arduino have spurred the development of easy-to-use programming idioms and design patterns. These conventions allow those without formal training to learn embedded development skills quickly while continuing to make progress on the projects that impelled them to start working with embedded devices [1]. Tools from programming language design can allow device generation tools to incorporate new technologies as they become available by creating a robust framework for change over time [3]. Tools from type theory can allow expert users to express complex relationships between software and hardware while being invisible to novices [4]. Embedded device generation synthesizes these concepts to provide an accessible way to describe the function of embedded systems, and then provide the tools to turn those descriptions into reality.

Our design for embedded device generation allows it to become an important research tool, providing faster and easier ways to develop hardware, and providing components that integrate into other research efforts. Device generation can be coupled with new tools for circuit printing and multi-material 3D printing [5], to create systems with complex electronic and mechanical properties. Other systems could incorporate device generation as a component, generating bespoke embedded devices that are specialized to the task at hand. Existing tools could incorporate the component library and formalisms to provide engineers stronger forms of design verification. The modularity of embedded device generation will allow researchers to incorporate its components into their work, opening new avenues for inquiry.

Conclusion

Embedded hardware development is still an arduous process, requiring time, money, and skill that is often unreasonable. Device generation will change that by providing users an accessible method for describing embedded hardware and tools that can automate turning those descriptions into actual hardware. The components of embedded device generation can serve as building blocks for future researcher efforts, creating powerful new methods for reasoning about embedded systems. Device generation tools will provide large, new communities the ability to build embedded devices and serve as a catalyst for innovation within the personal fabrication community.

Acknowledgements

This work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. This material is based upon work partially supported by the National Science Foundation under Grant No. 1505684.

References

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

[2] Al Hilwa. 2013. *2014 Wordwide Software and ICT-skilled Worker Estimates*. Technical Report. IDC.

[3] Nathaniel Nystrom, Michael R Clarkson, and others. 2003. Polyglot: An extensible compiler framework for Java. In *Compiler Construction*. Springer, 138–152.

[4] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA.

[5] Pitchaya Sitthi-Amorn, Javier E Ramos, and others. 2015. MultiFab: a machine vision assisted platform for multi-material 3D printing. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 129.