Let's Make Robots!
Automated Co-Generation of Electromechanical Devices from User Specifications Using a Modular Robot Library

by

Joseph DelPreto

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2016

Author .....
Department of Electrical Engineering and Computer Science
January 29, 2016

Certified by .....
Daniela Rus
Professor and Director of CSAIL
Thesis Supervisor

Accepted by .....
Leslie Kolodziejski
Chair of the Committee on Graduate Students

Let's Make Robots!

Automated Co-Generation of Electromechanical Devices from User Specifications Using a Modular Robot Library

by

Joseph DelPreto

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2016 in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Personalized on-demand robots have a vast potential to impact daily life and change how people interact with technology, but so far this potential has remained largely untapped. Building robots is typically restricted to experts due to the extensive knowledge, experience, and resources required. This thesis aims to remove these barriers with an end-to-end system for intuitively designing robots from high-level specifications. By describing an envisioned structure or behavior, casual users can immediately build and use a robot for their task.

The presented work encourages users to treat robots for physical tasks as they would treat software for computational tasks. By simplifying the design process and fostering an iterative approach, it moves towards the proliferation of on-demand custom robots that can address applications including education, healthcare, disaster aid, and everyday life.

Users can intuitively compose modular components from an integrated library into complex electromechanical devices. The system provides design feedback, performs verification, makes any required modifications, and then co-designs the underlying subsystems to generate wiring instructions, mechanical drawings, microcontroller code for autonomous behavior, and user interface software. The current work features printable origami-inspired foldable robots as well as general electromechanical devices, and is extensible to many fabrication techniques. Building upon this foundation, tools are provided that allow users to describe functionality rather than structure, simulate robot systems, and explore design spaces to achieve behavioral guarantees.

The presented system allows non-engineering users to rapidly fabricate customized robots, facilitating the proliferation of robots in everyday life. It thereby marks an important step towards the realization of personal robots that have captured imaginations for decades.

Thesis Supervisor: Daniela Rus, Professor and Director of CSAIL

Acknowledgments

There have been many adventures, challenges, questions, and experiments along the road to this thesis. I would like to thank Daniela for her tireless guidance throughout the last few years, providing motivation and support with infectious enthusiasm and energy. Her advice and insight always help to light the way forward, revealing paths towards exciting research that can benefit society.

This work has been performed in collaboration with numerous people at MIT and beyond. Ankur helped to lay the foundations for this project, and I am grateful for his numerous contributions, his help with writing papers and performing demonstrations, the presentations he has given, and his overall knack for making robots foldable. The efforts of our collaborators at Cornell's Autonomous Systems Lab and MIT's Programming Languages and Software Engineering Lab have also helped to extend the capabilities of our robot compiler and make it accessible to a wider audience. In addition, the ideas and related work of the entire Printable Programmable Robot team have helped to inspire and enhance the pursuits presented in this thesis. This work was also funded in part by NSF grants 1240383 and 1138967, as well as the NSF Graduate Research Fellowship 1122374.

I would also like to thank all of the members of the Distributed Robotics Lab for creating a welcoming and supportive community. Their camaraderie, collaboration, friendship, and antics have helped to fill the years with fun times and fond memories.

Last but not least, I extend my appreciation and gratitude to family and friends who have made all of this possible. Their appreciation, excitement, and tolerance for my projects provide the fuel that drives my motivation and imagination. Robots can be entertaining by themselves, but the wonder that they inspire in other people is truly remarkable.

Table of Contents

Table of Contents - Continued

Chapter 1

Introduction

If you can dream it, you can do it.

– Walt Disney

Contents

1.1 Vision

Robots have long represented the cutting-edge of technology, weaving their way through our books, theaters, movies, and above all our imaginations. The idea of customized personal robots has embedded itself in the heart of many visions of technology and hopes for the future. It is at the forefront of the intersection between what we know should be possible and what has not yet been realized.

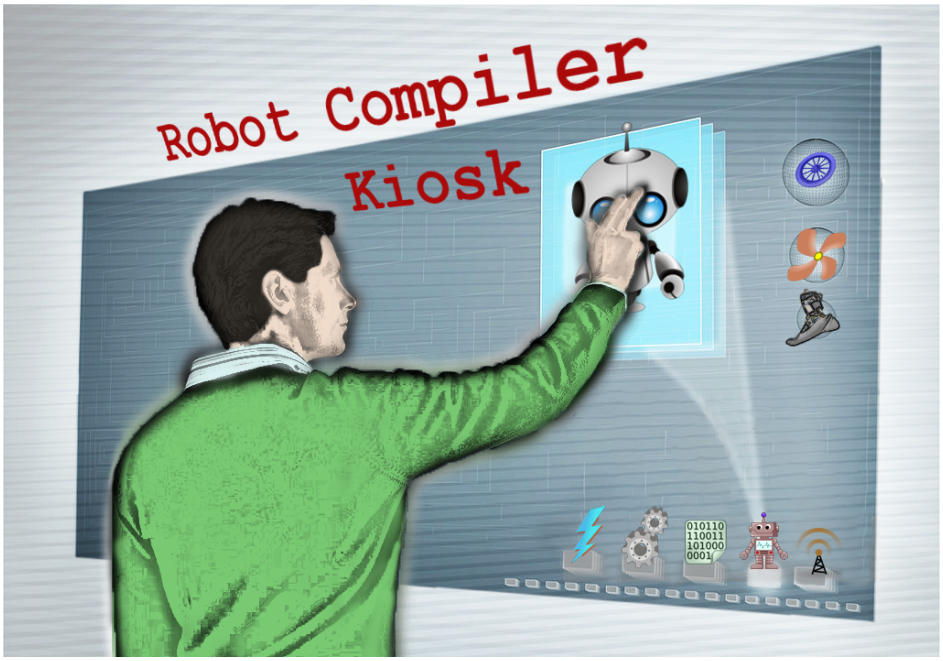


Figure 1.1: Making personal robots should be as easy as downloading and customizing a piece of software. Illustrated here is a conception of a novice user intuitively designing a robot from customizable building blocks.

This gap has mostly been due to a large knowledge barrier imposed by modern processes. Currently, intricate low-level knowledge and experience regarding the interactions between electrical, mechanical, and software systems must be leveraged to design and fabricate robots. This process often involves multiple software packages and computer-assisted technologies, adding to the list of domain-specific training required to build complex electromechanical systems. Furthermore, the entire time-intensive iterative design flow frequently begins anew for each robot as the implementations become specialized for certain tasks. As a result, robots have become increasingly commonplace in research and industrial settings, where the necessary knowledge exists and where special-purpose robots are very useful, but they have not infiltrated everyday life for mainstream society.

To address this issue, a system is needed that automates the design and fabrication of robots. Non-engineering users should be able to describe what they want a robot do at a very high level, and with minimal additional interaction obtain a completed robot that achieves the task. The presented system works towards this goal by allowing users to design robots by connecting customizable building blocks from a library – this could lead to robot

kiosks such as the one envisioned in Figure 1.1, where users can describe and immediately obtain fully operational robots. On-demand robots could then address a wide variety of applications including education, disaster search and recovery, healthcare, everyday tasks, and beyond. If making a robot to solve a physical problem is as easy as downloading an app to solve a computational problem, then robots can begin to proliferate and improve people's lives throughout society in ways only limited by creativity.

1.2 Contributions

The presented work brings this future of pervasive on-demand robots closer by enabling the co-design of electrical, mechanical, and software systems from high-level user specifications. It creates a framework for experts to design parameterized low-level building blocks encapsulating all necessary design information, and for algorithms to hierarchically compose these blocks into collections of ever-increasing complexity. This results in a library of robotic components that casual users can browse, interconnect, and customize.

Using this system, a user describes what the robot should look like and how it should interact with the environment while the implementation details are autonomously managed behind the scenes. The system verifies the design, makes adjustments and completions to ensure functionality, and ultimately produces a fabricable set of outputs including the robot structure, electrical layouts, bills of materials, low-level driver code, high-level behavioral software, smartphone user interfaces, and assembly instructions.

In particular, this work describes the following:

- an end-to-end software system leveraging a modular parameterized element that can encapsulate all necessary design information such as electrical, mechanical, and software data
- algorithms for hierarchically composing these modules into new components of ever-increasing complexity while retaining and appropriately modifying the encapsulated design information
- a library of modular robotic components designed in this manner
- algorithms to allow casual users to select parts from this library and intuitively drag them together to design a robot
- an intuitive graphical design process that merges structure and behavior via information flow among modular components
- algorithms for analyzing user designs to verify functionality, and to address issues by autonomously inserting or removing components and connections

- algorithms for autonomously compiling a finalized robot design, specified as a collection of modules, into fabricable outputs that include:
  - electrical layouts and bills of materials
  - mechanical files such as 2D drawings for origami-inspired foldable robots or solid object files for 3D printing
  - assembly instructions to guide the user through the mechanical fabrication and electrical wiring
  - a customized graphical user interface to wirelessly control each new robot from a smartphone
  - software that can be immediately programmed onto the robot's microcontrollers, containing low-level driver code as well as high-level control software so the robot performs the desired autonomous behavior
- a custom serial communication protocol to enable reliable information exchange throughout mesh networks of interconnected microcontrollers
- higher-level tools that allow the user to start by describing desired robot behaviors instead of structures
- higher-level tools that provide behavioral simulations and design space explorations for systems of personalized robots to achieve behavioral guarantees
- sample robots and electromechanical devices designed and fabricated using this system

The core system, the robot compiler, provides an intuitive design interface for robot systems by encouraging hierarchical compositions of customizable base components that are drawn from a modularized library. The system then compiles these design specifications into integrated fabricable outputs, co-designing the electrical, mechanical, and software subsystems. Various algorithms are implemented to efficiently achieve this synthesis and to provide feedback, while a custom serial communication protocol ensures reliable communication among microcontrollers on the final robots. Higher level extensions have then been added to this core system; these allow novice users to start by describing envisioned behavior, and provide tools for simulating systems of designed robots and for modifying parameters to ensure that behavioral guarantees are satisfied. As a whole, the presented work moves towards the vision of pervasive customized robots by allowing casual users to intuitively rapidly prototype novel inexpensive robots.

1.3 Software-Defined Hardware Using a Modular Library

If casual users are to feel comfortable designing and building a robot, then the interface should mirror those of common services that they already use in other aspects of their lives. From browsing home decoration catalogs or customizing a new car, to preparing a gourmet meal or ordering a laptop, people are accustomed to selecting from libraries of options in order to achieve an overall vision, consulting an expert if necessary. They may also hire a professional to independently make choices based on a description of the overall vision. The same paradigm can be applied to creating robots; components of varying complexity, such as sensors, motors, grippers, or arms, can be chosen from a library to assemble a robot design while the tedious details are processed behind the scenes. If a user is unsure of what components to choose or how to connect them, the system may act as the expert by autonomously making decisions and recommendations. If the system also accepts behavioral or task descriptions, then the user can simply provide a high-level vision and allow the system to independently choose components from the library. In this way, a robot can be intuitively designed without worrying about the underlying engineering challenges. A conception of a robot kiosk implementing this paradigm can be seen in Figure 1.1.

The long-term objective is therefore to develop a hardware compiler that can automatically design and fabricate a robot to accomplish desired goals from a description of the desired tasks. Towards this end, the current system modularizes electrical, mechanical, software, and user interface elements to create a database of parameterized blocks encapsulated in a common abstraction suitable for hierarchical composition. It allows electrical, mechanical, and software components to be coupled at the lowest level by experts, and then abstracted into functionally defined blocks usable by novices. From compositions of these blocks, the system automatically generates complete robot designs including electrical layouts, fabrication files, firmware, software for autonomous behavior, and user interfaces. Higher-level algorithms are also written on top of this foundation, allowing for intuitive user inputs such as functional requirements and behavioral descriptions.

1.4 Designing Robot Structures Using Information Flow

In order to allow users to intuitively design robots, the system should encourage a design flow that leverages how people already approach design tasks. Towards this end, the presented model focuses on describing the flow and manipulation of information among chosen components. For example, a designer may focus on how the motion of one component should influence the motion of another component, or on how interacting with a graphical interface affects the robot's behavior. An illustration of this is shown in Figure 1.2; red mechanical arrows indicate physical connections among parts, thus illustrating the final structure, and green arrows indicate data flow among parts, thereby illustrating communication and behavior. A concrete example of information flow using implemented components from the current system can also be seen in Figure 3.14. By visualizing the flow of information and actions in this way, the behavior of the robot as well as the relationships between components can be made readily apparent without exposing implementation details.

Internally, each component of the database is an encapsulated module that can be conceptually replaced by a parameterized "black box" mapping a set of inputs to outputs. These ports conceptually transmit information related to the behavior defined by that component. Ports can take on a number of different types, depending on the nature of the information transmitted therein. Electrical ports transmit electrical signals, and their connections are realized by wires or other communication channels. Data ports represent the flow of conceptual information such as software values, and their connections are realized by code functions or variables. Mechanical ports transmit information in the form of spatial position and orientation, and their connections are realized by physical joints. By providing ports of different types on the same component, these various subsystems are integrated and designed simultaneously.

Connecting library components along their interfaces traces a path of information from a set of inputs to a set of outputs through various transformations. The overall input/output relationship defines the functionality of the designed mechanism, while the specific path describes an implementation. Different implementations can achieve the same target func-

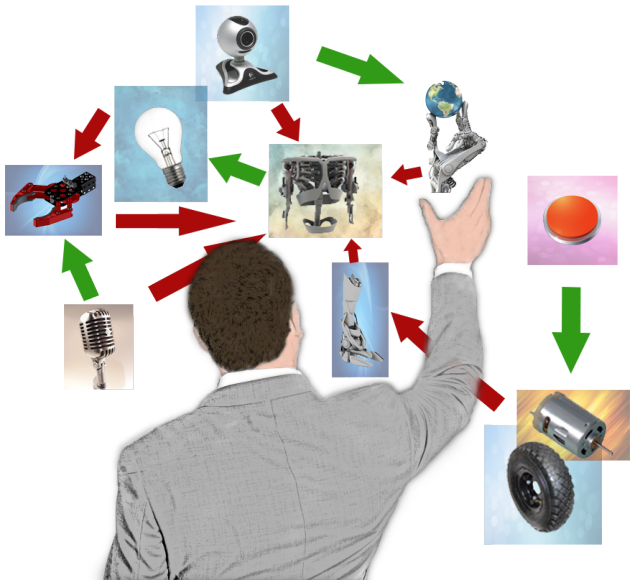


Figure 1.2: This conceptual illustration indicates a design based on information flow using modules from a robot library. Red arrows indicate mechanical connections, reflecting the final structure; for example, the motor is connected to the leg which is mounted on the torso. Green arrows indicate information flow, and reflect the communication paths and behavior; for example, the hand receives object detection information from a camera, a gripper closes in response to noise, and pressing a button activates a motor.

tionality; for example, pressing a user interface button may move the robot forward regardless of whether the robot uses legs or wheels. Mapping the flow of both conceptual and physical information provides a means of visually understanding the overall behavior at a high level, while directly reflecting the underlying hierarchical composition.

1.5 Designing Robot Behaviors

Using information flow among modules to design a robot marks a compromise between high-level abstractions and low-level details, providing a middle ground for novice and expert users. Novice users can go a level higher and design from behavioral descriptions, allowing the system to make autonomous decisions about constructing the information flow, while expert users can go a level lower and inspect or create individual components.

To facilitate higher-level abstractions, two extensions have been added to the robot compiler. The first builds upon the foundation of modular components and algorithms to allow users to specify desired behaviors. The system generates a finite state machine controller from the behavioral description, then aids the user to choose appropriate components and shape the information flow. The result is a system that starts with a functional task specification and ends with a complete autonomous robot.

A second extension to the robot compiler provides tools for simulating and verifying behavioral properties of robot systems, as well as for parameter exploration. It leverages a new programming language designed for controlling robots to grant intermediate users the ability to program high-level behaviors in a more intuitive manner. It also interfaces with the robot compiler's library and algorithms, allowing for the creation of robots that achieve the behavior. The robot compiler also generates kinematic models of these robots that enable the new extension to simulate their interaction with the environment and with other robots. In this way, it can verify global behavioral properties for systems of robots, explore different component parameters to adjust the robot designs, and provide robot controller logic that will achieve the desired goals.

1.6 Thesis Outline

An outline of the thesis is depicted in Figure 1.3. It begins with a discussion of related work in Chapter 2. Chapter 3 then provides an overview of the robot compiler software system, describing the class infrastructure and the subsystems implemented for robotics applications. Chapter 4 then expands on the issue of automated software generation within this framework, including how component connections, data networks, user interfaces, and robot behaviors are realized for each new robot. A novel serial communication protocol is then presented in Chapter 5, which enables reliable communication between microcontrollers in mesh networks. Chapter 6 uses all of these tools to synthesize user-specified designs, perform automated design verification and modification, and produce complete mechanical, electrical, and software outputs.

Sample robots and other electromechanical devices generated using this system are presented in Chapter 7. These include origami-inspired printable robots as well as a distributed robot garden and more general electromechanical devices. Each of these has an associated user interface, and most of them also demonstrate autonomous behavior.

Extensions to the robot compiler foundation that enable the abstraction of user input to a higher level are then described. Chapter 8 allows users to start with a functional task description instead of structural specifications. Chapter 9 enables simulation and parameter determination for multi-robot systems to achieve desired behavioral guarantees.

Finally, Chapter 10 presents potential areas for future work as well as concluding remarks. Together, these chapters describe an end-to-end system that facilitates the rapid prototyping of customized personal robots. With it, an important step is taken towards changing how people view robots and how robots shape everyday life.

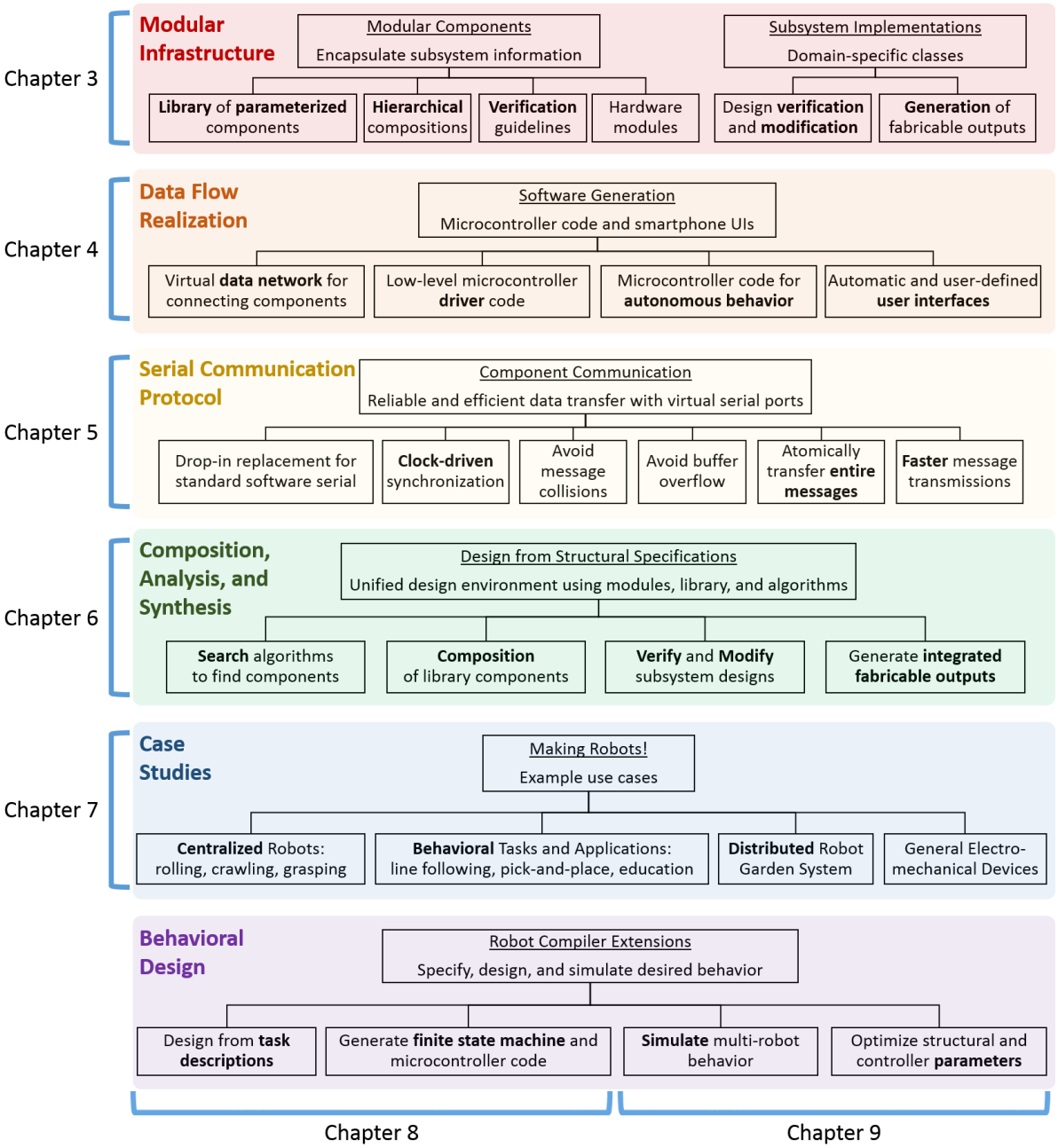


Figure 1.3: The thesis is divided into chapters that describe the core robot compiler as well as higher-level extensions implemented on top of its library and algorithms. It also presents a wide range of robots and devices created using the end-to-end system.

Chapter 2

Related Work

The more that you read, the more things you will know.
The more that you learn, the more places you'll go.

– Dr. Seuss

Contents

The system presented in this thesis builds upon work from the realms of rapid fabrication technologies, modular design methods, functional specifications, programming frameworks for robotic embedded systems, and robotic system specification. Most directly, it joins and extends previously published work regarding the robot compiler foundation [1–3], the distributed robot garden system [4], and the robot compiler's integrations with Cornell's `LTLMoP` [5] and MIT's `React` [6].

2.1 Modular Robotics and Design Automation

The use of modular methods can greatly simplify, clarify, and speed up system design. There has been research regarding how to decompose systems into modules [7], and this paradigm has been widely adopted throughout the software development communities. Modular design can also be applied to robot creation to achieve similar benefits over custom design [8, 9] and work has even been done to try to automate such a design process [10].

Various commercial products have also been developed that work to make robots modular and accessible to people of widely varying levels of experience. For example, littleBits [11], MOSS [12], and VEX [13] offer a variety of electronic modules that can snap together to form basic functional products. The user does not need to write any code in order to use these blocks, but the lack of an option for intuitive programming can be a hindrance when creating complex systems. Similarly, while the blocks offer a wide range of functionality, the electronics are linked to the mechanical structures to an extent that may make it difficult to quickly create an arbitrary robot. Another widely used and highly developed modular system is Lego Mindstorms [14]; this provides a flexible platform with a suite of devices that all connect directly to a central processor, and a relatively simple programming interface to allow for more complex designs and increased control.

These research and commercial systems all call upon the use of a discrete set of specially designed physical modular building blocks, adding expense and limiting the configuration space. The system presented in this thesis adapts the modular design method to use off-the-shelf electronics and a 2D cut-and-fold fabrication process, enabling a much broader range of customizability from cheaper raw materials. Furthermore, it helps to automate the design process by, for example, recommending components and performing basic verifications and modifications. The mechanical, electrical, and software designs are synthesized automatically for each new robot from high-level user descriptions.

While physical structures are often designed in an interactive graphical environment using CAD software, this has not been readily extended to complex integrated electromechanical

products. Towards this end, there has been work on creating domain specific programming languages to specify hardware designs using software for electrical circuits [15] and rigid bodies [16]. This software-defined-hardware paradigm has also been used to define robotic designs for simulations using a scripted modular language [10, 17]. Finally, recent tools such as the robot compiler presented in this work and `ROSLab` [18] make use of the democratization of rapid prototyping technologies to unify the development of manufacturable robots. The system presented in this thesis employs a similar design method for user input, while focusing on physical device creation and directly compiling user designs into fabricable outputs.

2.2 Rapid Prototyping of Robot Structures

There has been a growing body of research related to the rapid fabrication of mechanical structures using a variety of techniques.

Arbitrary on-demand structures are generally achievable by additive manufacturing using 3D printers, and advances in printer technology have made desktop printers such as [19, 20] available to the general public. However, while complex solid geometries are easily manufactured with 3D printing, achieving the required compliance and mobility necessary for general robotic systems is difficult using most common techniques [21]. Limited workarounds have been explored [22, 23], but these often lack robustness or reliability. In addition, 3D printers are typically plagued by long fabrication times – though quicker than conventional manufacturing processes, parts still take on the order of hours to build.

Mechanical structures can alternatively be realized in an origami-inspired process by folding patterned 2D sheets into the desired geometry. A variety of substrates are possible, including cardboard laminates [24], single layer plastic film [25], or more exotic materials such as for small exoskeletons [26] and metallic structures [27]. These designs can be manually folded by hand, folded by embedded or external active stimuli, or passively folded by controlled environmental conditions [28, 29]. These processes have been used to create passive 3D structures [28,30] as well as active programmable robots such as insect-like micro-

robots [26], hexapedal robots [31], and worm robots [32]. The system presented in this paper employs these origami-inspired fabrication processes to rapidly create inexpensive robots.

Although such 2D fabrication methods have been employed for rapid prototyping, being able to manufacture devices in a time frame of minutes, creating the fabrication drawings typically requires careful hand design by experienced engineers. The aforementioned works used sophisticated 2D CAD programs, generating drawings that were difficult to visualize as 3D objects, and custom electronic circuitry and software to drive the actuators. The robots were created as monolithic integrated designs, and so these issues compound as designs grow in size and complexity. Progress has been made towards simplifying this process, for example by using manually scripted elements to simplify design iteration [33] or by using a component-based mechanical design [34]. These works remain within the realm of designing desired structures, however, rather than abstracting the design input to a task-based level or integrating electrical and software systems.

There have also been attempts to automate the decomposition of 3D shapes into 2D fold patterns [35–37]. These tools and algorithms, however, focus mostly on solid objects, employing various heuristics to generate polyhedra obeying certain rules; compliant and kinematic structures are not addressed.

2.3 Robot Controllers and Simulation

There has been an increasing interest in the robot community regarding the automatic construction of provably correct robot controllers from high-level or temporal logic task specifications. These controllers, if successfully synthesized, will behave as specified in the mission statements.

The synthesis and execution of these controllers from temporal logic specifications have been demonstrated [38, 39]. Work has also been done to address a variety of challenges regarding the controllers, such as a changing workspace during controller execution [40,

41], motion planning for robots given temporal goals [42], and generating optimized robot trajectories from temporal logic task specifications [43]. These controllers can also be used to control multiple robots [44].

The artificial intelligence and planning communities also have languages to create functional specifications, such as STRIPS [45] or PDDL [46]. Using these planning languages, functional specification is defined and solved with the composition of various actions that each consist of pre-conditions or post-conditions. As an extension to the robot compiler, high-level task specifications are used to generate provably correct robot controllers for customized behaviors. The functional specification system for this work stems mostly from [47]. Given a robot model and its environment, controllers that satisfy high-level task specifications are automatically composed; this controller is then converted to microcontroller code by the robot compiler and the robot library is used to instantiate a physical robot that achieves the desired behavior.

There has also been substantial research regarding the behavior of controllers that interact with the external world, as in the case of cyber-physical systems. A paramount question for this area is how to handle time. From a theoretical perspective, timed systems are well studied [48, 49] and the semantics of timed languages such as `Giotto` [50] have successfully transferred to embedded systems in industry [51]. However, due to the complexity of the hardware and software stack, even determining the worst case execution time of a program can be difficult [52]. Therefore, few programming languages feature time in their primitives.

Considering more general interactions between a discrete software controller and the continuous physical world leads to hybrid systems [53–55]. A hybrid system contains both a discrete part such as a software controller and a continuous part such as a robot subject to the laws of physics. In the hybrid automata formalism [54], a hybrid system is a finite automaton where each state has associated continuous dynamics described by differential equations. The analysis of hybrid systems is a hard problem that is only decidable for simple subclasses [53, 56–58]. Most of the work on verification of hybrid systems, and more generally numerical programs, has focused on linear systems or uses linear abstractions.

Non-linear systems are still an open challenge [59], and the body of work on them is more limited and more recent [60–63].

From a practical perspective, there exists fully integrated modeling and programming environments such as Simulink [64] and LabView [51] that provide both simulation and code generation. Modelica [65] is a language purely for modelling and simulation that can be seen as a generalization of bond graphs [66] for multi-domain systems such as those incorporating mechanical, electrical, and hydraulic components. For programming, the Robotic Operating System (ROS) [67] aims at streamlining the development of high-level software for robots and has found applications in both academia and industry. For low-level programming, Wiring [68], based on the concepts of Processing [69], is popular for programming simple microcontrollers such as Arduinos.

The programming language `React` works towards addressing many of these concerns regarding robot controllers and behavioral programming. Integrated with the robot compiler, it helps to provide an integrated system within which robots can be simulated, designed, programmed, and fabricated.

Chapter 3

Robot Compiler Architecture

It is always wise to look ahead, but difficult to look further than you can see.

– Winston Churchill

Contents

The robot compiler is a software system with a versatile framework of classes and algorithms for creating and composing components, as well as a growing library of integrated modules. It enables rapid fabrication of complex devices by automating the co-design of electrical, mechanical, and software subsystems. The following sections outline the architecture of this system, and describe the robotic subsystems currently implemented.

Some of the concepts presented in this chapter are also discussed in [1–3]
The complete code for the robot compiler can be found at http://people.csail.mit.edu/delpreto

3.1 Compiler Overview

The design system is implemented as a Python package, with the designs themselves defined and generated by Python scripts. This software-defined-hardware paradigm allows for general cross-platform compatibility, and inherits many of the benefits inherent in software development. It provides a great deal of flexibility and adaptability, allowing disparate electromechanical mechanisms utilizing various fabrication methods to be readily generated from the same design interface.

The software package implementing the integrated co-design environment is divided into a few main categories, which are illustrated in Figure 3.1 and listed below:

- classes that define the underlying code architecture of the software-defined-hardware, such as parameterized integrated modules and domain-specific synthesizers
- algorithms to compose and instantiate module classes as well as to efficiently manage large collections of modules
- utilities and builder applications so that casual users can assemble library components into higher-order electromechanical designs
- a library of instantiations of these classes, a few written by experts and many hierarchically composed by novices
- algorithms to autonomously verify and modify hierarchical designs as well as to generate complete robot designs from compositions

A few superclasses comprise the basis for the modular component library, and various algorithms act upon them to allow for their hierarchical composition and for design generation. This provides a general framework for modular design, and the current instantiations focus on robotic systems; the modules therefore encapsulate electrical, software, and mechanical information. Combining all subsystems into integrated components facilitates their simultaneous co-generation, allowing systems to interact and for robots to be holistically generated. Since the generated robots are rapidly prototyped and largely reusable, an iterative approach to personal robotics is encouraged.

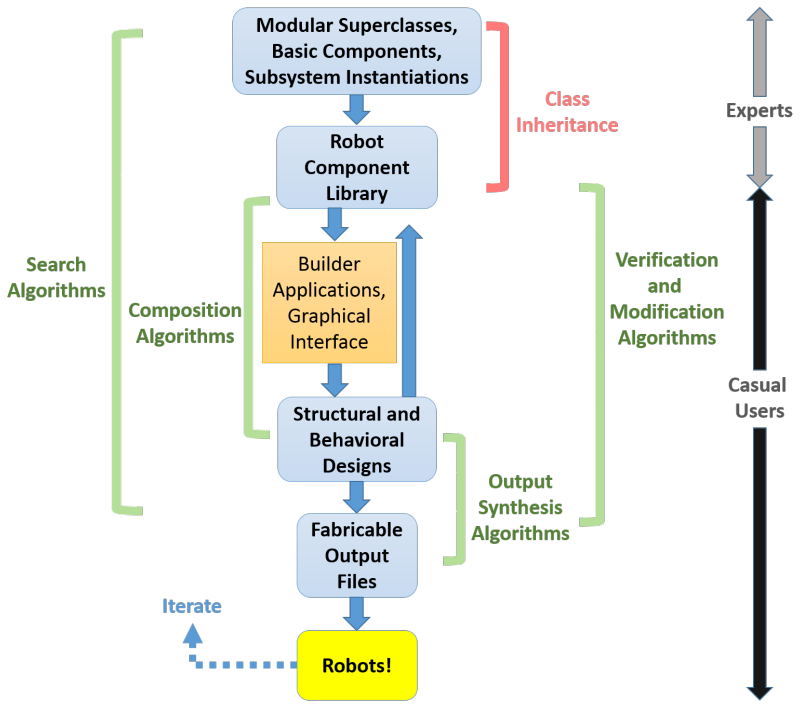


Figure 3.1: The robot compiler foundation consists of expert-defined superclasses that can be hierarchically composed into increasingly complex electromechanical designs by casual users. This creates a library of parts, each of which can be compiled into fabricable outputs. Suites of included algorithms operate in each phase of the process. The rapid and intuitive nature of this process encourages an iterative design approach.

3.1.1 Hierarchical Compositions of Parameterized Modules

The fundamental unit of abstraction in this design system is called a `component`. This represents an individual design element that can accomplish a self-contained set of functionality, and provides the required encapsulation to define and create that device. The simplest components are basic building blocks of electromechanical structures, such as a discrete LED, code method, UI toggle button, or mechanical beam. Expert users can directly generate these low-level building blocks, while both expert and casual users can hierarchically connect existing blocks as illustrated in Figure 3.2. These higher-order components can represent anything from mechanical assemblies and control systems to integrated electromechanical mechanisms and full robots. Each such hierarchical composition is itself a component, defined by its collection of sub-components, how their interfaces are connected, and which interfaces from its sub-components are exposed for future connections to the new

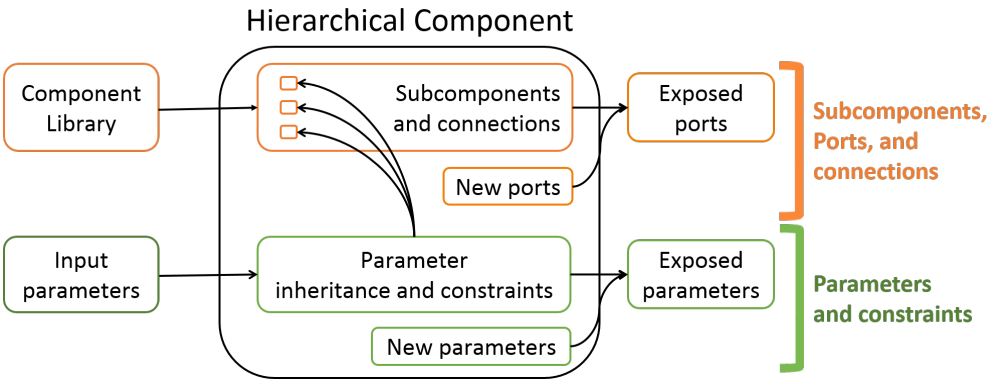


Figure 3.2: Components can be hierarchically composed to easily create more complex designs. Components contain subcomponents from the library, define how they interconnect, inherit or constrain their parameters to new higher-level parameters, and expose their ports for future connections. This new component can then be added back into the library.

super-component. A design library is initially populated with basic building blocks designed by experts to provide a core set of functionality, and then expands as users create new components by connecting existing components along exposed interfaces.

Components can be parameterized, allowing for fine-tuned design customization. These parameters define adjustable values that quantitatively but not qualitatively change the functionality of modules. Examples of parameters may include device models, voltages, feedback loop gains, or geometric dimensions. They allow casual users to customize a component's behavior without changing its fundamental function. Assigning values to all parameters of a component fully specifies that component, and is sufficient to generate fabricable design files for manufacturing. New components can inherit parameters from lower in the hierarchy, define new higher-level parameters, or create relationships and constraints between parameters. As users select components from the library to design their robot, they can set parameters of varying abstraction such as material thickness, finger dimensions, number of fingers on a gripper, number of arms on a body, or the payload capacity of a gripping robot.

This paradigm allows a high degree of component reuse, enabling incremental adaptation from earlier designs. Furthermore, the generated designs take the form of text-based code scripts, and are therefore easy to share, modify, adapt, and extend using open source tools.

3.1.2 Co-Design of Robot Subsystems from Modular Compositions

As a user defines the desired information flow by combining electromechanical modules, subsystem design information is collected behind the scenes to maintain an integrated design throughout the modular composition hierarchy. Complexity is managed by nesting hierarchical constructions in an intuitive design abstraction, allowing an inexperienced user to easily understand and utilize the design process. Ultimately, this high-level assembly of modules can be directly compiled to generate fabrication specifications to manufacture the robot.

Component hierarchies are compiled by the system into a collection of files necessary for a user to manufacture the specified design: the mechanical structure is made using 2D or 3D rapid fabrication processes from generated fabrication drawings, the user assembles the electrical subsystem onto that structure guided by a bill of materials and generated wiring instructions, and generated software gets loaded onto any microcontrollers. The resulting robot can be wirelessly controlled from a generated user interface, autonomously controlled from generated application software, or user-programmed with custom behaviors with help from a generated control library. A snapshot of this process is shown in Figure 3.3.

The current implementations of the modules focus on origami-style 2D manufacturing processes, which allow robots to be quickly cut and folded from paper or plastic. The electronics are assembled into the folded structure according to generated instructions, and the generated software is loaded directly onto the microcontrollers. This emphasis on rapid fabrication and prototyping also encourages an iterative design process that integrates well with the paradigm of designing based on information flow. A user can design a robot, fabricate it, test it, and then adjust the design accordingly to make a new robot. The electronics can be reused across robots, while the mechanical structure is inexpensive and rapid to produce. This enables the rapid iteration through designs or one-time use robots often demanded by custom personal robots as well as educational applications.

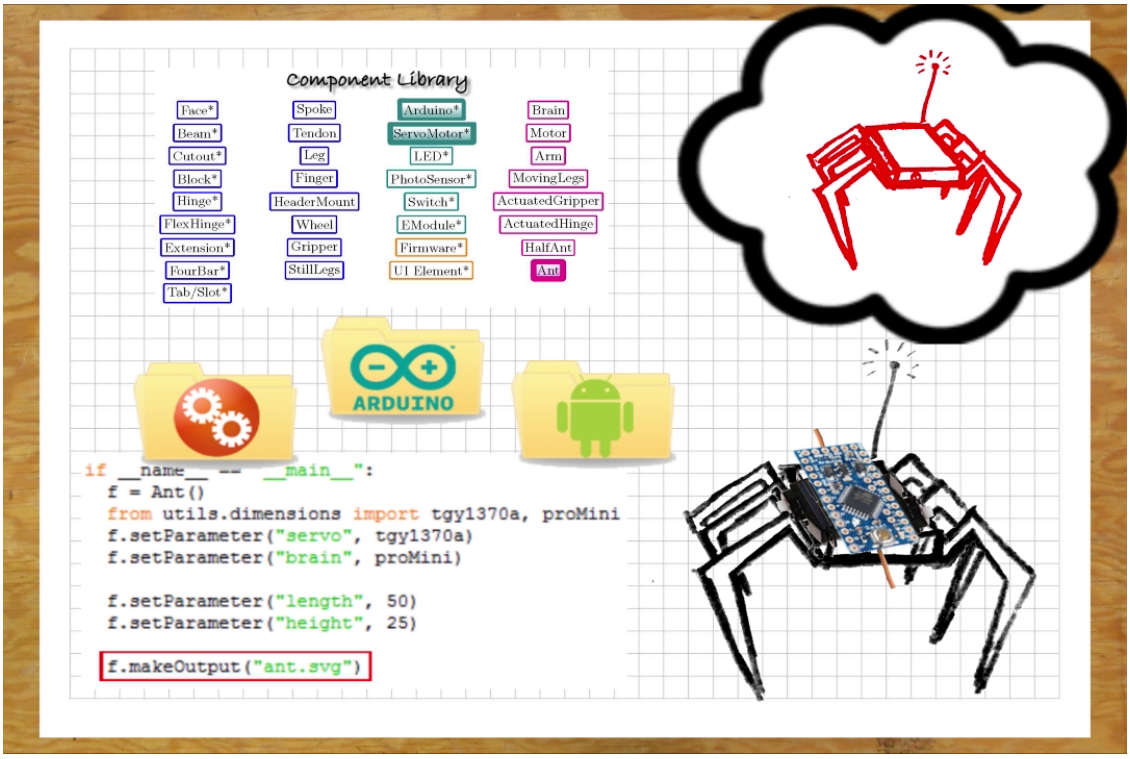


Figure 3.3: Once a design is finalized by choosing and connecting components from the library, final fabricable outputs are automatically generated from the collection of modules. Here, an envisioned ant robot is created using modules from the library, a subset of which is shown. Outputs then include mechanical files, such as 2D drawings for origami robots or CAD files for 3D-printable objects, electrical layouts and instructions, Arduino software to control the robot, and an Android user interface.[1]

[1]Figure 3.3 is a frame of a video that can be found at http://people.csail.mit.edu/delpreto

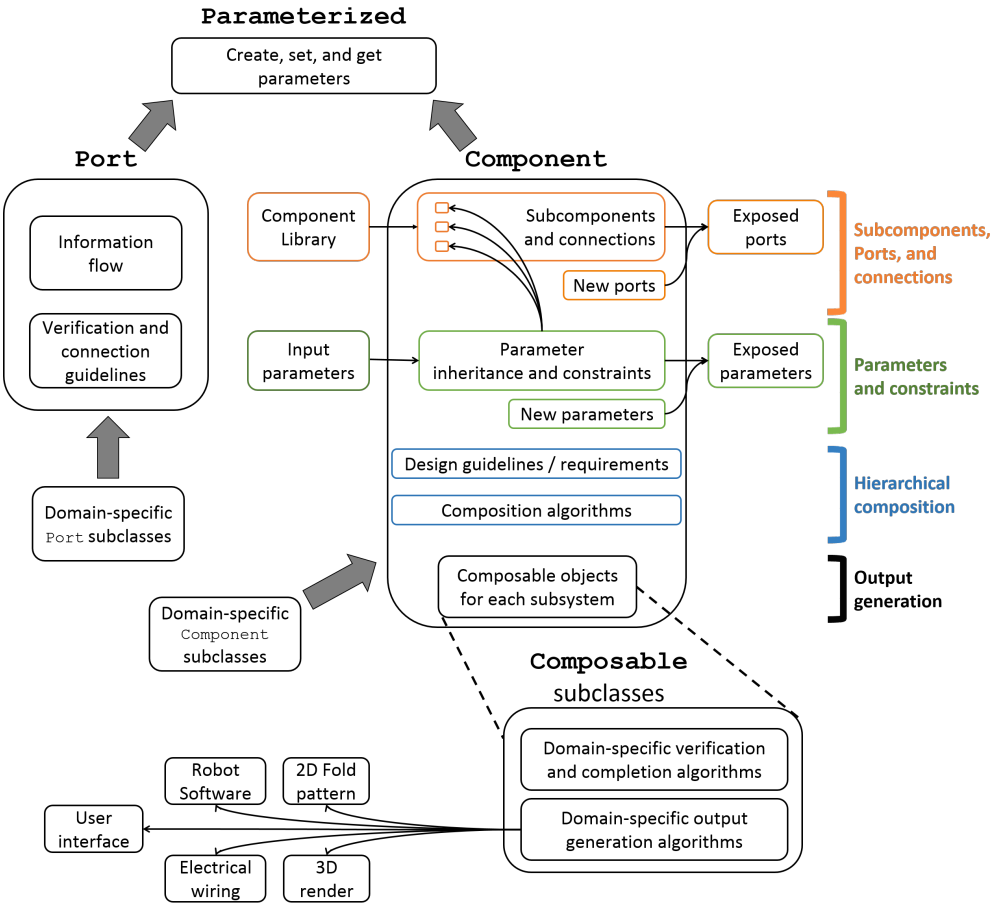


Figure 3.4: The core set of abstract classes include `Parameterized` for quantitatively customizing modules, `Port` for forging connections between modules, `Component` for representing a modular element, and `Composable` for running domain-specific verification and output algorithms. The class inheritance structure among components and ports facilitates creating new classes and reflects subsystem interaction.

3.2 Compiler Classes

The software infrastructure used to implement the modular paradigm is comprised of a few superclasses that allow for a wide range of instantiations depending on the application. They provide ways to represent the design topology and its constituent elements, and to convert that information into final electromechanical outputs. The relationships between the core superclasses are illustrated in Figure 3.4, and descriptions are provided below.

`Parameterized` is an abstract class that stores and sets parameters. It stores a dictionary of parameters and their default values, and handles getting and setting values as well as renaming or deleting parameters.

`Port` is an abstract subclass of `Parameterized` that represents an interface along which components can be connected. It also provides an infrastructure for verifying whether ports are allowed to connect to each other, facilitating automated design verification. Subclasses of `Port` can be created for each subsystem that extend the verification logic and form an informative class inheritance structure.

`Component` is an abstract subclass of `Parameterized` that represents a parameterized modular element and contains a set of hierarchically nested interconnected subcomponents. It stores lists of its subcomponents, interfaces, connections, parameters, which interfaces and parameters are inherited, constraints among parameters, and design requirements. Each component is itself a complete design, and creates a tree structure via the subcomponents and their connections. Similar to ports, the subclasses of `Component` can form an informative class inheritance structure that help to illustrate the various integrated subsystems and how they interact.

`Composable` is an abstract class that serves to process design information with domain-specific logic for verification and output generation. `Composable` classes form the software that defines the physical outputs specified by the components, combining the various mechanical, electrical, and software subsystems of the electromechanical device. Each subsystem can define a subclass of `Composable` that stores design information in a way that is meaningful for its specific domain and that facilitates the conversion of information into final outputs. `Composable` objects then interact with each other during the output phase, creating an intelligent co-design of integrated electrical, mechanical, and software designs.

Together, these classes and their subclasses form the foundation of an integrated robot library that can be used to intuitively design complex systems and generate complete outputs.

Experts can directly create new subclasses, facilitated by the class inheritance structure, and casual users can compose these blocks into designs of ever increasing complexity without worrying about the details. The following subsections discuss each of these classes as well as the library structure in more detail.

3.2.1 The `Port` Class

The `Port` class represents a pathway by which information passes to and from components in a hierarchical design. Like components themselves, ports inherit from the abstract `Parameterized` class and therefore have parameters that can be used to quantitatively customize ports during design and implementation. When instantiating a connection between interfaces of two components, ports can ensure that information logically flows from one component into the other.

Port requirements can be used to automatically determine appropriate interfaces to join. For example, when connecting an analog sensor to a microcontroller, the sensor's electrical output port must connect to an electrical input port that supports analog readings. Using the port requirements, the system can automatically select suitable interfaces for the connection. Similarly, the data, electrical, and mechanical subsystems will automatically be joined as appropriate. Additional rules for port matching can be programmed to provide more sophisticated design guidance and automation, and can be used to provide verification of user decisions.

To facilitate these rules for automatic connections and design principles, the `Port` subclasses defined by experts form a structure of class inheritance that illustrates how they relate to each other and to the underlying subsystems. At the most general level, the `Port` class allows for the specification of port types to which it should connect and port types to which it cannot connect, as well as basic rules for the verification and determination of connections. These principles are represented by a list of allowable mate types, a list of recommended mate types, and methods called `canMate` and `shouldMate`. The `Port` class' implementation

of `canMate` checks if any allowable types have been specified, and if so checks that the given port's type is a subclass of one of them. The `shouldMate` method is analogous but uses the recommended mate types. Subclass implementations of `Port` can then override these methods, calling the superclass methods to ensure basic compatibility while also providing more sophisticated matching criteria. Class inheritance is also encouraged by the superclass methods; if a subclass adds a recommended or allowable type that is a subtype of one already stored in the list, the higher-level type will be automatically removed and only the more specific one will be stored.

New types of ports defined by experts inherit from existing port types, and can extend their lists of recommended or forbidden types as well as their rules for verifying and forging connections. For example, the abstract subclass `OutputPort` specifies that it cannot connect to another `OutputPort` and recommends connecting to an `InputPort`, while the abstract subclass `DataPort` specifies that it can only connect to other `DataPort`s. A slightly more specialized port type such as `DataOutputPort` can inherit from both of these, automatically inheriting both of their connection information. Together, this means that a `DataOutputPort` will be mated with, for example, subclasses of `DataInputPort`. Class inheritance therefore achieves the desired behavior without requiring any additional effort from the designer. As each subclass can also add additional verification logic, port connection principles can quickly become quite sophisticated as the hierarchy grows. Examples of ports currently implementing such an inheritance hierarchy for each subsystem are described in Section 3.3.

3.2.2 The `Component` Class

`Component` is a class that embodies the modular paradigm and represents the core building block of the robot compiler library. It inherits from the abstract class `Parameterized`, which allows it to store a list of parameters that may define how the component is configured or how it behaves. The basic information that a component stores, as well as how it supports hierarchical composition and interacts with the other core classes, is illustrated in Figure 3.4.

Every component contains the following to define its state and its self-contained hierarchy:

- a list of its interfaces for connecting to other components
- references to its constituent subcomponents
- lists of how its subcomponents are interconnected
- information about how its own parameters and interfaces relate to those of its subcomponents
- a list of which types of `Composable` are relevant to its required subsystems
- design requirements, such as other components that must be included in the design to achieve desired functionality

This information represents the interconnected subcomponent hierarchy, and collections of methods are included that act upon this modular information to allow the `Composable` objects to produce final design outputs. Not all components will use all subsystems, but a derived component will use every subsystem used by the subcomponents comprising its hierarchy.

While a basic building block designed by an expert may not have any subcomponents, higher-level components are created by nesting and connecting subcomponents and defining relationships between them and the new component. A list of connected subcomponent ports is stored to represent connections between subcomponents. Certain subcomponent interfaces can also be “exposed," which means that the new component will inherit that interface and allow future components to connect to it. Similarly, new parameters can be added to a component, and subcomponent parameters can be inherited. Instead of directly inheriting parameters, however, components may also store constraints between parameters as equations in order to define how configurations or behaviors relate to each other. For example, dimensions of connected subcomponents can be constrained to reflect physical limitations, or voltage and current values can be equalized. By expressing the value of a subcomponent parameter in terms of the values of its parent's parameters, complex relationships can quickly evolve as the depth of the component hierarchy grows. This allows for a versatile framework of composing modules and facilitates the percolation of information throughout the design tree.

While nesting of subcomponents is a flexible method of hierarchically composing components into new designs for both casual users and experts, the `Component` subclasses written by experts also represent a hierarchy through a structure of class inheritance. Any component may serve as the superclass for a more specialized component, allowing new definitions to inherit rules, parameters, parameter constraints, ports, types, and design principles from previously designed components. This makes it easier for experts to design new components since most of the tedious details have already been managed by pre-existing, higher-level components. The current library therefore has expert-defined abstract subclasses of `Component` that represent components from various subsystems such as electrical, software, mechanical, and combinations thereof. Concrete subclasses of these then represent the basic building blocks for hierarchical composition. Examples of classes that currently implement such an inheritance structure are presented in Section 3.3.

The definition of a `Component` comprises lists of interfaces, subcomponents, parameters with constraints, and connections. Each of these can be converted to textual representations by using names of subcomponents and their class types, parameter names, string representations of equations and parameter values, and port names. Thus, a component can be conveniently stored as a human-readable text file. The current implementation uses a `YAML` file format [70], which is a human-readable data serialization standard. The `Component` class contains methods for saving itself as a yaml file, and for instantiating itself based on an existing yaml file. This provides a convenient way to both define and store derived components. Typically, experts will define a few abstract and concrete Python subclasses of `Component`, and hierarchically derived components that are created as collections of these classes are simply stored as yaml files that reference the basic building blocks.

3.2.3 The `Composable` Class

Composables are created for each specific subsystem to convert the design information stored by components into final output files. Separating the `Composable` class from the `Component` class allows for distinct design and output phases. This makes computation more efficient, since composables need not be instantiated during the design phase and outputs are only computed once the design is finalized. The encapsulation of output algorithms in a separate class inheritance structure also increases the adaptability and generalizability of the system. Subclasses of the `Composable` class, inheriting from ones already defined within the appropriate subsystem, enable experts to quickly provide support for new output processes and formats. For example, one can easily switch between outputting a robot as an origami-style fold pattern and as a 3D printed solid body, or even fabricate different parts of the robot using different techniques.

In addition to generating final outputs, composables contain methods for analyzing and validating their respective subsystems. For example, they can check port connections and component requirements, and add new components or automatically forge new connections. For an electrical `Composable` subclass, this may include choosing microcontrollers, assigning controller pins to devices, and inserting voltage or current converters. This therefore provides a means to encapsulate subsystem information and allow the system to perform meaningful automated design feedback and modification. The class inheritance structure also reduces the effort needed by experts to create new classes, since much of the logic and intelligence can be inherited from existing class definitions.

When a component is ready to be processed, it instantiates the `Composable` classes that each subcomponent requires and provides it with known design information such as subcomponents, ports, and connections. The composable converts this information into a form that is useful for its subsystem domain; for example, an electrical composable may only pay attention to electrical ports and may store connections based on which microcontroller they eventually affect. Composables from various subcomponents are then appended together, using provided methods for incorporating information from another composable.

Executing the output method of a composable generates output files that specify the creation of the respective subsystem. Electrical output files include a bill of materials, circuit diagrams, and wiring instructions to assemble the desired circuit. Software subsystems include a set of program files to be loaded onto the central microcontroller for both low-level control and high-level behaviors, off-board user interface apps for human control of the robot, and code libraries that simplify the creation of custom control programs. Mechanical output files include a 2D vector drawing that can be directly sent to a laser or vinyl cutter to create a cut-and-fold structure, and a solid object file that can be built using a 3D printer.

Although separate composables are created for each subsystem, they do not operate in isolation. When a design is verified and modified or compiled into final outputs, an iterative process consults each `Composable` object in turn to produce new information and possibly edit the design. In this fashion, each composable can impact decisions of each other composable. By exchanging information among subsystems, an integrated co-design process is implemented where the electrical, mechanical, and software systems interact with each other to produce a coherent integrated design. More details about the algorithms embedded within each `Composable` subclass will be discussed in Section 6.2 and Section 6.4.

3.3 Implemented Subsystems and Component Library

The provided class infrastructure can be used for many different modular systems, but four subsystems for robotic applications upon which the current system focuses are the electrical, software, data, and mechanical domains. The electrical, software, and mechanical systems have direct realizations in the final robot, while the data subsystem focuses on passing abstract data between modules to realize desired behavior.

All components and ports created for these subsystems are stored in an integrated library. An expert-designed basic component may take the form of a Python class inheriting from `Component`, or a Python script that instantiates a `Component` class and then manually specifies or augments its defining elements. Derived components, however, are specified only in terms of their subcomponent breakdown, and so can be stored in plain text using the `YAML` markup language. Derived components can alternatively be created with a Python script, giving greater configurability to an expert user. Currently, both a text-based console interface and a simple graphical interface exist to allow non-expert users to build robotic designs by intuitively assembling building blocks. A web interface is currently under development.

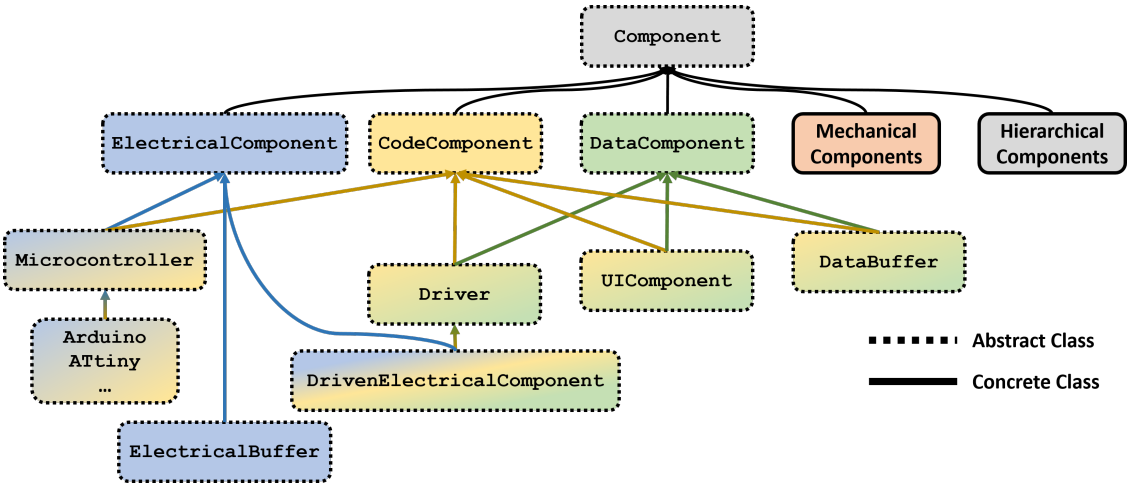


Figure 3.5: A core set of superclasses has been created for the electrical, data, software, and mechanical subsystems that forms the basis for concrete subclasses and hierarchical compositions. Class inheritance, indicated by arrows and color-coding, represents the interaction between subsystems.

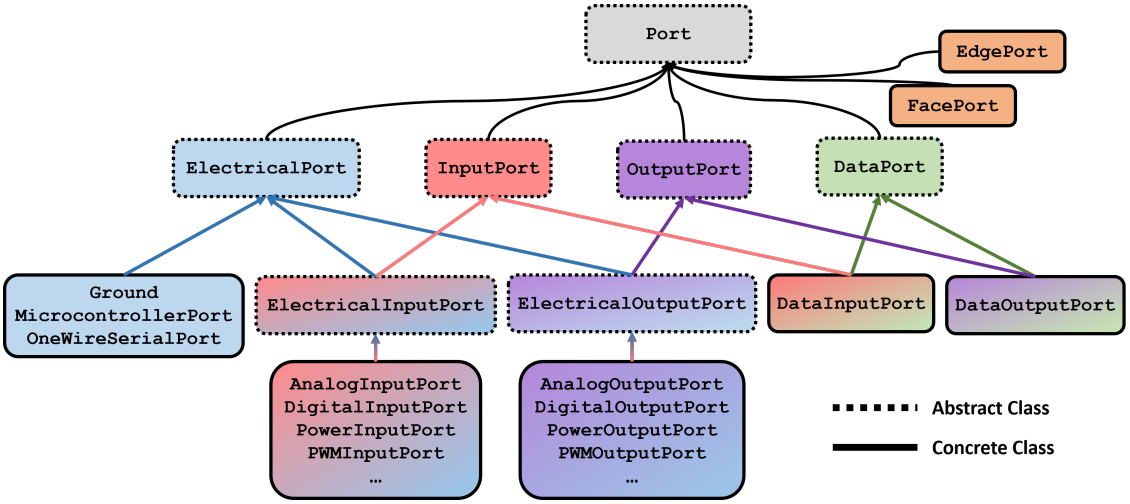


Figure 3.6: Various abstract `Port` subclasses are provided for creating connections between subsystem components, as manifested by their inheritance structure. Instantiations of the concrete classes can then be used to create information channels between components.

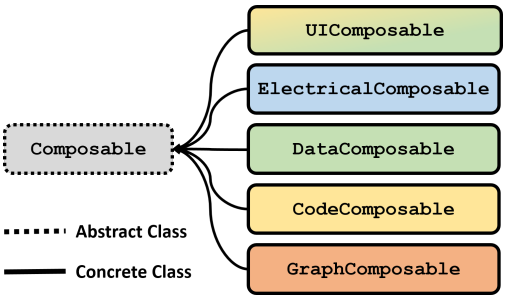


Figure 3.7: Subclasses of `Composable` allow each subsystem to assert domain-specific expertise during verification and output phases of the robot creation process.

The core set of `Component` superclasses defined for the robotic subsystems is illustrated in Figure 3.5. Similarly, Figure 3.6 illustrates the prominent `Port` superclasses used by these components. These expert-defined classes provide the building blocks for hierarchical compositions, and their class inheritance tree illustrates the interaction between subsystems.

For each subsystem, a `Composable` subclass is also created to process the design information with domain-specific algorithms. Shown in Figure 3.7, these allow for the interactive verification and output synthesis of user designs across the integrated domains.

3.3.1 Electrical Subsystem

Within a hierarchical composition of elements, the electrical subsystem is determined by the topology of the electrical devices and connections. Each device added to the design may contain electrical ports, and connections between them represent physical connections that describe how electrical information flows throughout the design. Components additionally containing other types of ports define how the electrical information interacts with other subsystems on the robot. A library of basic components and methods has been developed to address the typical electrical needs of a robotic system, namely various forms of sensing, actuation, processing, communication, and user interfacing.

Information Flow

The sources and sinks of electrical information can reveal the underlying structure of the design. Devices such as sensors or communication modules can source electrical information, and devices such as servos or LEDs can sink electrical information. Note that the overall information flow does not necessarily start or stop at these devices, but the electrical information does; for example, a communication module may take in a conceptual value and convert it to electrical information, and the servo takes in electrical information and converts it to mechanical information. These devices may therefore serve as electrical sources or sinks while being sinks or sources for other types of information, highlighting the interplay between subsystems.

Less informative ports such as power connections, and details such as particular microcontroller pins used, are abstracted away from the user during the design process. At fabrication time, the system automatically creates power connections, chooses particular pins and pin types, and inserts devices such as microcontrollers or power converters if necessary so that only the informational flow needs to be considered during design.

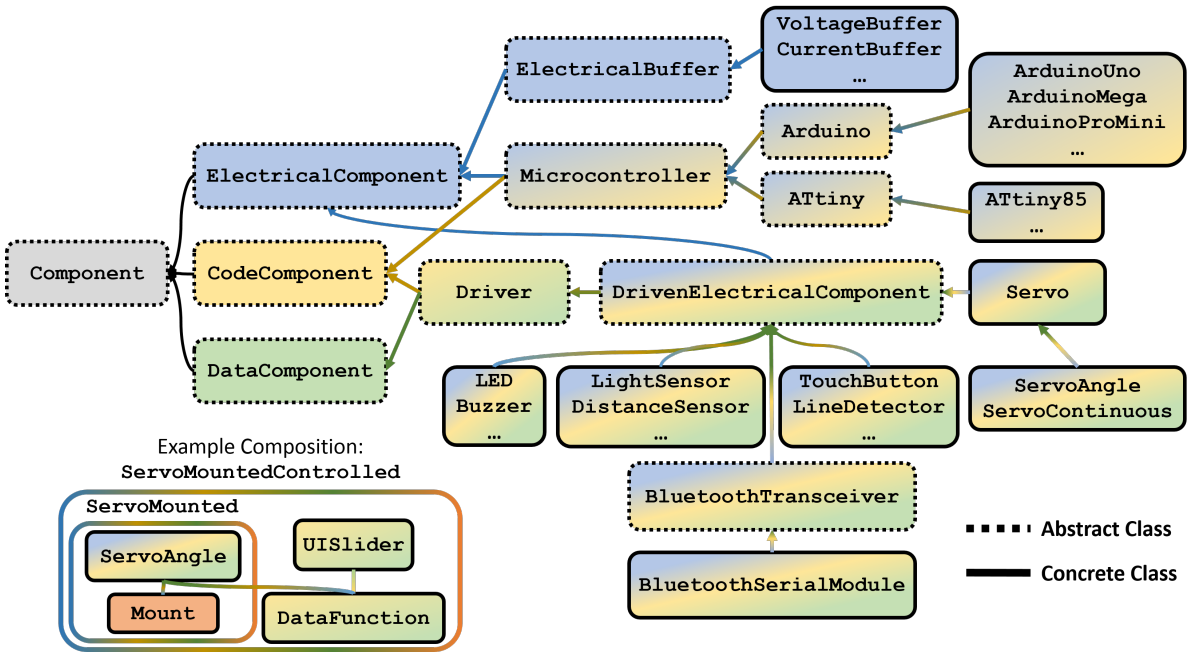


Figure 3.8: A combination of abstract and concrete classes is provided for creating electrical components. These provide the basis for more sophisticated hierarchical compositions or more specialized subclasses. The class inheritance, indicated by arrows and color-coding, illustrates that the electrical system closely interacts with the data and software systems.

Library Components and Ports

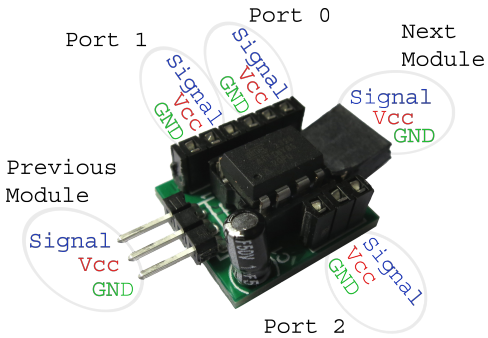
The expert-defined abstract subclass `ElectricalComponent` inherits directly from `Component`. It adds parameters such as voltage and current, and verification logic related to these parameters, to describe the electrical characteristics. Other subclasses then inherit from this component, and derived components are created by hierarchically composing such objects. Both cases allow for the new parameters such as voltage to be inherited or constrained, ensuring proper electrical functionality. In this way, a library of basic components has been developed that addresses the typical electrical needs of a robotic system.

The class hierarchy for the electrical subsystem as well as a sampling of some derived components are shown in Figure 3.8. Some components such as voltage buffers may be solely electrical devices, representing devices such as operational amplifiers or discrete LEDs. Most components in the library, however, span other subsystems as well. For example, microcontrollers include software and driven components include software and data ports.

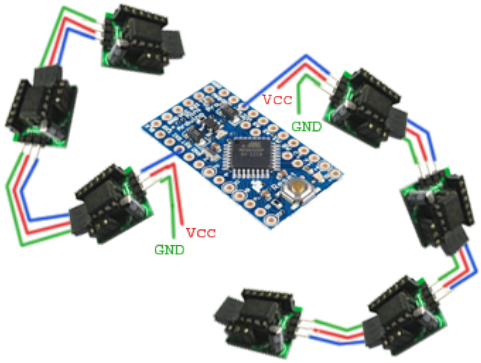
Some inherited classes such `ServoAngle` simply adjust parameters of their parent classes, while others such as `BluetoothTransceiver` offer new functionality. Each of the concrete classes can be further customized via their parameters, for example by choosing voltage levels or device models. An example of hierarchical composition is also illustrated by the mounted servo, which contains a mechanical mount and a driven servo, and by the controlled servo, which contains a mounted servo and a UI element.

The library has been populated with discrete electronic components that have standardized header connectors, allowing for simple plug-in connections between devices. This eliminates the requirement for custom printed circuit boards (PCBs) to handle electrical interconnects. However, the system does not preclude such design elements – the extensible nature of the component abstraction can allow an expert designer to implement a PCB `Composable` to enable more complex electrical devices and circuits, at the expense of in-home fabricability for a casual user.

A number of electrical ports, shown in Figure 3.6, have been defined that reflect typical connections seen in a robotic control system. These inherit from an abstract superclass `ElectricalPort`, which can be dynamically configured as various types via parameters. In particular, there are parameters for whether the port is analog or digital, and what type of protocol it expects (e.g. serial communication, I^2C, direct voltages, etc.). The rules for recommended or forbidden connections are then extended to check these parameters in addition to the usual checks based on class type. This allows for greater configuration flexibility without the need to create a new class for every possible port type. Furthermore, it allows electrical ports to dynamically change their configuration, which is particularly useful for microcontroller ports. The concrete classes shown such as `AnalogInputPort` can simply set the protocol and type parameters accordingly, simplifying the creation of these classes by experts. In addition, the `MicrocontrollerPort` can store an electrical port object for each pin along with a list of the possible protocols and types, allowing it to quickly check if connections to other devices are possible and to change configurations as ports are connected.



(a) Each module features connections for upstream and downstream modules as well as three ports for connecting devices such as servos, LEDs, or digital and analog sensors. These modules are designed to be plug-and-play and do not require reprogramming based on location or connected devices.



(b) Modules can be connected together to form distributed chains, allowing arbitrary numbers of devices and diverse physical layouts. In this case, two chains have been added to the brain and devices such as motors, LEDs, or sensors can then be plugged into any of the modules' ports.

Figure 3.9: Plug-and-play hardware modules serve as interfaces between devices and the main controller, expanding the capabilities and allowing the electrical layout to mirror the mechanical layout.

Hardware Modules

The electrical subsystem is not designed in isolation, and components may directly serve as interfaces between subsystems by containing ports of different types. Moreover, many electromechanical devices required to accomplish physical tasks are often distributed throughout the robot structure. Hardware modules have been developed to facilitate this co-design and allow the electrical layout to mirror the mechanical structure.

The modularity and scalability of the electrical system is enhanced by plug-and-play hardware modules that serve as interfaces between electrical devices and the main controller. Each module uses an `ATtiny85` microcontroller to drive three general ports, as shown in Figure 3.9a. These can be independently configured as digital outputs, PWM outputs, digital inputs, or analog inputs. Since these modules are designed to be plug-and-play, the code loaded on the modules does not change according to the robot design; on startup, the main controller sends the modules any necessary design-specific data such as what pin types to use. Communication is established between a module and the main controller via a one-

wire serial protocol, and messages are then exchanged such that users can attach devices to the modules as if they were being attached to the main controller. Modules can also be chained together, as shown in Figure 3.9b, in which case messages are passed along the chain until they reach the desired module.

In this way, the number of possible devices is no longer limited by the number of pins on the main controller. This approach also facilitates the physical distribution of devices across the robot while reducing the wiring complexity, thus allowing the electrical layout to naturally mirror the mechanical layout and reducing user effort during assembly. The flexible nature of the hardware modules can also be leveraged during automatic design, since the system can insert them where needed in order to join devices together.

`Composable` Subclass

The `ElectricalComposable` class is a subclass of `Composable` that processes the electrical subsystem. Its verification methods can check voltage and current requirements as well as microcontroller requirements. It can choose microcontrollers to use based on the collection of devices in the design, choose appropriate controller pins for devices, assign controller pin types, and address power requirements. It can also exchange information with the mechanical composable to determine if hardware modules would be desirable. Similarly, it interacts with the data and software composables since the wiring choices will affect how data is passed to and from microcontrollers and what software will need to be generated. At output time, it produces a bill of materials, circuit diagrams, and wiring instructions for the user.

3.3.2 Software and Data Subsystems

In general, electrical systems on a robot are controlled by processors such as microcontrollers, and thus the design of an electrical subsystem must directly interact with the design of a software subsystem. This subsystem includes driver firmware for controlling devices, higher-level microcontroller code, UI generation, and the ability to automatically generate code for robot behaviors. With this subsystem, components may pass information such as desired servo angles or UI slider positions as conceptual data values.

Information Flow

The software and data subsystems maintain an information flow among themselves and also create the infrastructure for information flow between other subsystems. Among themselves, subsystem topologies describe how conceptual data values move throughout the design. For example, UI elements such as toggle buttons or sliders can be information sources, and variables or UI displays can be information sinks. When coupled with other subsystems, data and software modules help to convert between information types and facilitate the codesign process. For example, the value from a UI slider may be converted to a mechanical output by a servo driver, and microcontrollers can generate code that controls robot devices or provides higher-level behavior. Data modules are frequently embedded as subcomponents within higher-order derived components, and are frequently automatically inserted by automated design verification and modification procedures.

An important function of this information flow is the graphical definition of robot behaviors. By connecting a servo to a light sensor via a scaling block, for example, a user can specify that a servo turn on in response to light; the system will automatically insert any additional blocks and software necessary to implement the desired functionality. User interfaces can be similarly defined by connecting UI elements directly to physical components, such as connecting a UI toggle switch to an LED. In this way, robot behaviors can be easily defined in an intuitive manner. For more arbitrary relationships and increased flexibility,

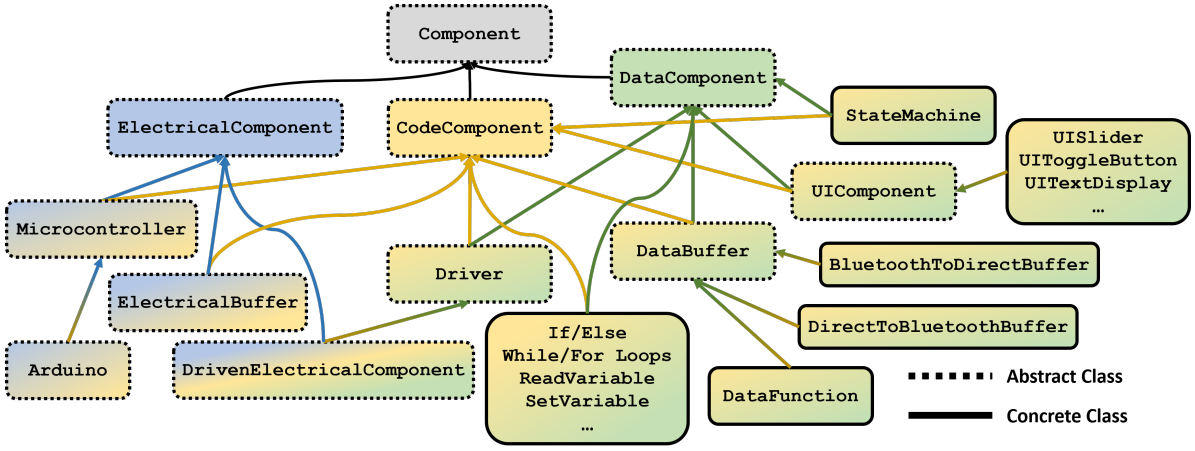


Figure 3.10: The basic building blocks created for the software and data subsystems allow information to flow between components and to generate control software. They enable data transformations and conversions, robot behavior via the direct linking of physical and conceptual modules, and graphical programming capabilities.

graphical programming blocks are also provided to allow intermediate users to graphically program the robot with arbitrary code that directly connects to the physical components. For example, an `if/else` block can be directly connected to a light sensor output for its conditional criteria. This therefore provides significant versatility and a way for novice users to generate sophisticated robots.

Library Components and Ports

A collection of abstract `Component` subclasses are provided in the library as the basis for software, data, and UI subsystems. Some of these are illustrated in Figure 3.10, and are described below.

- `CodeComponent` is an abstract subclass of `Component` that represents a component with associated software snippets. It can accept pointers to code files or folders, as well as direct string literals of code. This code can contain special tags that will be resolved to design-specific information when the robot designs are finalized.
- `DataComponent` is an abstract subclass of `Component` that represents a component accepting or outputting conceptual data. It contains data ports, which can be configured to support various protocols such as Bluetooth and various data types such as strings or integers.

- `Driver` is an abstract subclass of both `CodeComponent` and `DataComponent`, meant to represent a component that processes conceptual data and generates appropriate control software. As such, it has data ports to accept or output data, may perform data transformations, and stores and generates software snippets. These modules are frequently bundled with electrical components to represent controllable electrical devices.
- `DataBuffer` is an abstract subclass of both `CodeComponent` and `DataComponent` that can be used to convert between various types of data transfer. For example, it may convert a variable from a string to an integer, or interface between Bluetooth data and serial data. These often serve as interface blocks between subsystems by converting between types and protocols, performing data manipulations, or evaluating equations.
- `UIComponent` is an abstract subclass of both `CodeComponent` and `DataComponent` that represents a User Interface element. It generates and accepts conceptual data, such as for a text display or a control slider, and stores software snippets for generating the UI element. It is also parameterized to define, for example, what device is used for the interface and element-specific display configurations.

At fabrication time, the entire data network can be analyzed, the code tags resolved based on the component topology, and the software snippets pooled together; the result is auto-generated software that reflects the designed data flow. The collection of provided components allows users to obtain robot control software by designing at an abstraction level with which they are comfortable: expert users can write low-level code directly, intermediate users can use automatically generated code libraries to aid the writing of custom code, and novice users can intuitively link ports to specify behaviors and generate a graphical UI.

Controlling Physical Devices At the lowest level, code must be generated that allows the main controllers to directly interact with physical devices. Towards this end, `Driver` classes perform conversions between software, abstract data, and electrical signals; for example, a servo driver accepts as input a conceptual data value such as an angle, and contains a software snippet representing the knowledge of how to realize that value as an electrical signal. This software can also adapt to the design topology through the use of code tags and parameters. Drivers are therefore sources for the software subsystem and sinks for abstract data – they serve as indirect interfaces between the conceptual software realm and the physical electrical realm. Such examples illustrate that the designed sub-systems are not isolated

from each other, but rather interact both through the types of information they process and design parameters that affect how the information processing takes place.

Graphically Defining User Interfaces While hardware drivers are necessary abstraction barriers between the software and electrical realms, they are often included at a low level of the design hierarchy and not made transparent to the novice user. Other data sources such as UI elements, however, can be intuitively included in higher-level designs and allow humans to become information sources. Towards this end, `UIComponent` subclasses represent UI elements such as joysticks, buttons, switches, or sliders. These can accept conceptual data values that update the user interface display, or generate values that can be processed by other data blocks and ultimately control actuators or otherwise affect the robot's behavior. In this way, the user interface can be designed in parallel with the robot itself, such that the design process for the robot subsystems can interact with the design process for its human interface.

Robot Behavior via Data Manipulation Although drivers and UI elements serve as conceptual sources by translating data or human interaction into software and thereby enable direct control of devices, a robot should also be able to operate autonomously. An intuitive way to design such behavior is to link data sources and sinks together – for example, linking a distance sensor output to a servo angle input through some simple function can create a wall-following robot. To facilitate such information flow, various `DataFunction` blocks can manipulate data within the conceptual realm. For example, a simple transformation block data may accept data from a sensor and scale it to a value that is meaningful to a servo driver. By serving as an interface between sensors and actuators, this conversion enables autonomous behavior that is easily described in the design environment. Similarly, data may be converted from a human-readable version to a machine-readable version, facilitating human interaction with the final design. Thus, the flow of conceptual data within the design largely describes the resulting behavior of the robot.

Robot Behavior via Graphical Programming Data buffers and functions allow for the direct linking of devices throughout the design and the seamless integration of the conceptual and physical realms, but more advanced users may want to specify robot behaviors in a more arbitrary manner. Library components are therefore provided which allow for graphically writing arbitrary code. These blocks include `if/else` statements, loops, and the declaration and definition of variables or methods. Using these blocks, arbitrary code can be created to specify robot behavior. Such blocks also include data ports that allow the software to directly utilize the design's information flow; for example, a block creating a `while` loop may be directly connected to the output of a sensor for use in its conditional expression, or a `readVariable` block's output can be connected to an `LED` input. Details of how the data signals are converted into software, such as how the sensor value is obtained, can be encapsulated lower in the hierarchy and thus abstracted away from the user.

Code Generation and Software Sinks The various elements described above translate conceptual data into software snippets to realize the abstract flow of data defined by the design. Ultimately, these must be processed and pooled together into coherent packages. Towards this end, a microcontroller such as an `Arduino` may be a sink for the drivers' software, or an Android device may be a sink for the user interface software.

The software snippets written by experts and included in the `CodeComponent` objects can use various code tags that are processed once the design is complete. These may include pin numbers, device indices, counts of other devices in the design, device types, or other design parameters. These allow experts to write code snippets that are flexible and dependent upon the final design topology. In addition, they may write multiple code snippets and provide rules for choosing between them based upon design parameters – this allows the software sources to adjust their generated software according to the type of sink to which they are ultimately connected.

`Composable` Subclass

Once the flow of software and data is well defined, the software sinks can be pooled together into usable code. The `CodeComposable` subclass processes the aforementioned code tags to reflect the final design, generates complete software libraries that can be directly loaded onto the robot for both human control and autonomous behavior, and generates user interfaces for devices such as Android smartphones. Meanwhile, the `UIComposable` performs checks on the UI elements and synthesizes a final interface. The `DataComposable` similarly performs verification on the underlying data network to ensure that types and protocols are appropriate, automatically inserting converters and buffers as necessary.

3.3.3 Mechanical Subsystem

Mechanical building blocks define the physical structure and degrees of freedom of the robot body. They present input/output ports that specify physical positions and orientations for subsets of the mechanical design that interface with other components or with the environment. To maintain universality, designs are generated and stored in a process-independent data structure; process-specific plugins can then generate fabrication-ready outputs.

Information Flow

The ports of a mechanical structure describe locations along which additional mechanical elements can be physically attached; the information that flows through them is the spatial configuration of that patch. For rigid elements, the information that is assigned to the output port is an affine transform applied to the location of the input. For example, a beam can have one input and one output port, defined as the two ends of the beam: the value of the output port is the location of the input port offset by the length of the beam.

Mechanical components can be connected along mechanical ports to generate more complex geometries. Depending on the nature of these connections, additional mechanical ports

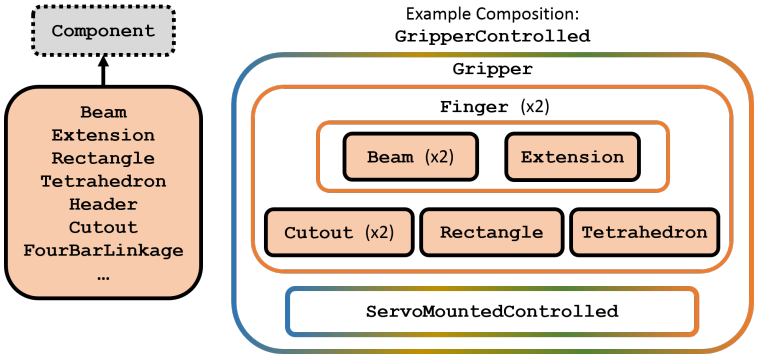


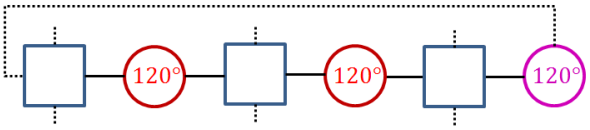
Figure 3.11: Structual primitives such as beams and rectangles inherit directly from `Component`, while more complex geometries can be created via composition. They can also be integrated with components from other subsystems to represent integrated electromechanical devices; here, a controlled gripper is made by composing a collection of fingers with the mounted controlled servo illustrated in Figure 3.8.

may be created in a composition if the resulting geometry has additional unconstrained degrees of freedom. A simple composite structure may include multiple beams connected together. In contrast to a primitive beam, this new design is entirely defined by its subcomponent structure and therefore can be defined graphically instead of using direct Python. The subcomponent ports are edges, connected by a flexible joint for compliant motion; this hinge creates an additional mechanical port for the angle of the flexure.

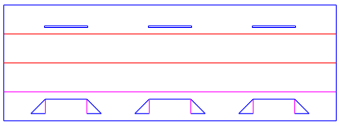
Mechanical components can also include degrees of freedom. In this case, setting an input value can result in a non-rigid deformation of the mechanical device. This is useful for generating motions of robotic mechanisms.

Library Components and Ports

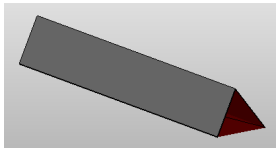
A number of components and ports are provided in the library as the foundation for describing mechanical assemblies. A portion of these is shown in Figure 3.11. These building blocks can then be hierarchically composed into more complex geometries and structures that implement desired kinematic and dynamic properties. Furthermore, they are often included in hierarchical compositions with components from other subsystems, for example to create a mounted servo or a gripper with associated electronics and actuation.



(a) A beam geometry can be represented by a face-edge graph that indicates connections and fold angles.



(b) This generated drawing of a beam can be sent to a 2D cutter such as a laser cutter or desktop vinyl cutter.



(c) A 3D solid model can also be generated, to visualize a folded pattern or to send to a 3D printer.

Figure 3.12: Mechanical geometries are stored using a face-edge graph that can be resolved to 2D or 3D fabrication outputs.

`Composable` Subclass

The mechanical information can be processed at output time to generate fabricable output files, and different `Composable` extensions can be created to provide support for multiple manufacturing techniques. The current library contains methods for generating 2D vector drawings for laser cutting, vinyl cutting, or hand cutting that create origami-style foldable robots. It also contains methods for generating solid-body objects to directly 3D print the robot parts.

Mechanical geometries are stored using a face-edge graph that can be resolved to both 2D and 3D shapes as required by specific fabrication processes. A basic example of this is the beam in Figure 3.12. The blue squares in the graph of Figure 3.12a represent the rectangular faces of the beam, connected to each other along folded edges represented by red circles. The unconnected dashed lines represent connections along which future components can be attached. A cut-and-fold pattern can be generated from the face graph, requiring the dotted edge to be replaced by a tab-and-slot connector. A 3D solid model can also be generated to display the structure resulting from folding the 2D pattern, or to directly generate a 3D object via 3D printing.

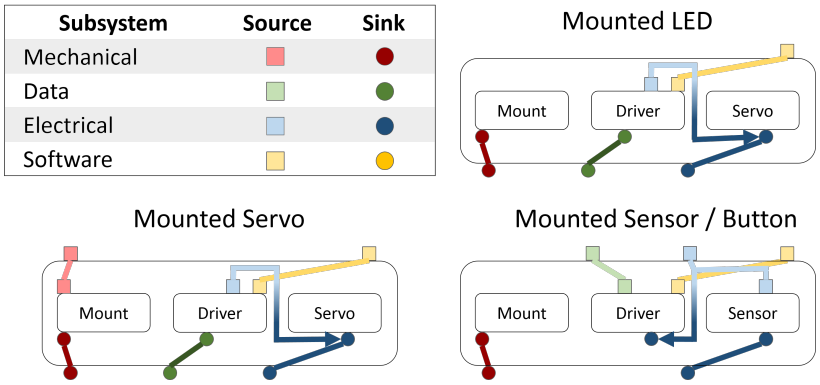


Figure 3.13: Integrated components can be defined by creating connections among encapsulated subcomponents and then choosing ports and parameters to expose for higher-level use.

3.3.4 Integrated Components

Because of the common API used by each component, design elements can be integrated across subsystems. A typical combination connects the electrical output of a driver to the input of an electromechanical transducer to give a logical actuator element driven by data signals. Connections such as these that encapsulate various subcomponents and then expose interfaces for future connections are shown in Figure 3.13. The abstract subclass `DrivenElectricalComponent`, which inherits from both `Driver` and `ElectricalComponent`, is also provided to represent this particular case. Higher levels of integration can further connect the block's data input to a UI data source and the block's mechanical output to a structural degree of freedom to yield a component representing a self-contained robotic mechanism.

Integrated designs can be achieved by inheriting from multiple subclasses spanning distinct subsystems, or by deriving hierarchical compositions. This facilitates the intuitive representation of complex devices that require electrical, mechanical, data, and software infrastructure. The `Composable` classes from each constituent subsystem then interact to enable an integrated co-design output generation process. This creates an augmented library of electromechanical modules that also contain associated code and UI elements, providing the foundation from which users can design a complete robotic system. The library currently

contains integrated components ranging from actuated grippers and wheels to mounted sensors and complete mobile platforms.

An example of using integrated components to design a two-segment robotic arm with attached gripper can be seen in Figure 3.14. A novice user only needs to connect the high-level integrated library components that represent the gripper, an arm segment, and a base. Behind the scenes, each of these components contains subcomponents and connections that implement the mechanical, electrical, data, and software subsystems required. Intermediate or expert users can choose the level of exposure with which they are comfortable, opening up levels of the hierarchy to view more details. Figure 3.14 chooses a view for the intermediate user that expands some components while still maintaining a relatively high level.

While Figure 3.14 illustrates the flow of the mechanical and data systems, similar connections exist for the electrical and software systems. The mechanical connections define the structure of the robot, while the UI elements directly connected to physical devices allow for user interface control. Furthermore, data connections among physical devices allow for autonomous behavior. For example, a user could add a light sensor connected to the joint servo and a touch button connected to the gripper servo, and customize the `DataFunction` blocks encapsulated within each of these to define scalings or other transformations. The realized robot could then move between two positions in response to light shining on the light sensor, and open or close its gripper in response to an object activating the touch sensor. In this way, using integrated components allows casual users to cogenerate mechanical structures, electrical wiring diagrams, behavioral software, and user interfaces.

3.4 Summary

Through a flexible collection of superclasses and expert-defined base classes, the provided framework supports the creation of complex hierarchical designs in an intuitive manner. This can be used for many different applications, and subsystems have been implemented that focus on robotic systems. These offer functionality such as sensing, processing, communication, data manipulation, mechanical structures, user interface creation, behavioral definition, and arbitrary graphical programming. Implementation details are managed behind the scenes by algorithms that will be discussed further in Chapter 6.

Using modules that integrate components from these subsystems, electrical, mechanical, data, and software systems can be defined in parallel by simply connecting ports to graphically represent the desired interactions. Ports from different subsystems can be directly connected, allowing users to easily express relationships among disparate domains. An included GUI enables user interaction with the library in a familiar way, and a more advanced web-based GUI is under development. By using the modular composition approach, users can choose an abstraction level with which they are comfortable. They can then quickly design custom personal robots by focusing on how information flows throughout the robot structure and software rather than on implementation details.

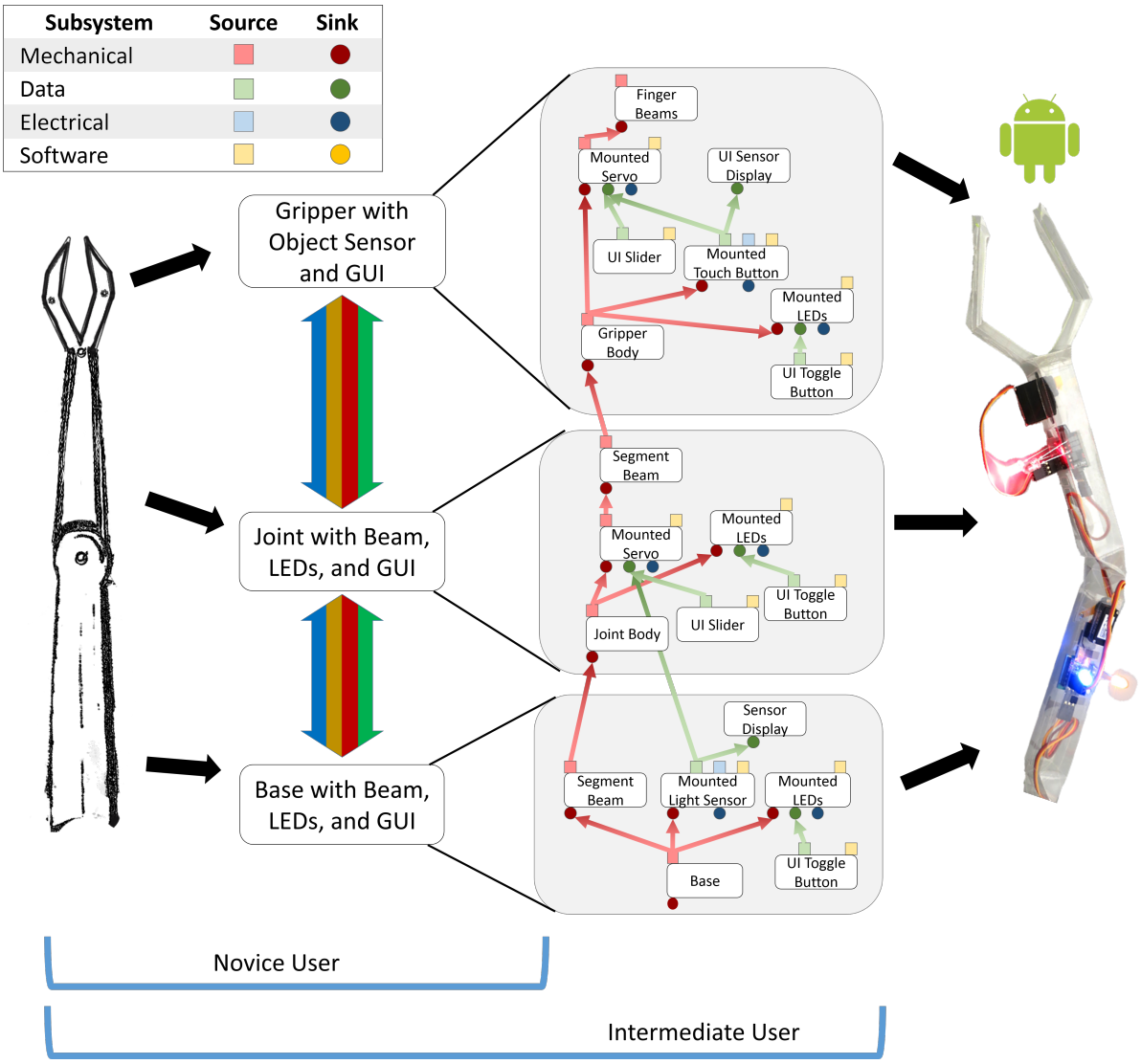


Figure 3.14: Integrated components that contain subcomponents or inheritance structures spanning multiple subsystems enable intuitive co-generation of complex electromechanical devices. Here, an arm with attached gripper is designed from derived components. The mechanical and data flows are chosen for illustration, representing the physical structure as well as the robot's behavior. Other subsystem flows, such as the electrical signals, are not pictured here.

Chapter 4

Software Generation: The Snippet Model

An idea that is developed and put into action is more important than an idea that exists only as an idea.

– Edward de Bono

Contents

A key capability of the robot compiler is automatically generating low-level drivers, high-level autonomous behaviors, and user interface functionality. Accomplishing this task for arbitrary designs requires a flexible framework for generating controller code that adapts to chosen topologies as well as a reliable method of exchanging data between controllers. This chapter addresses these challenges with a software template model where expert-defined snippets embedded in components are automatically pooled together and modified to reflect user-driven designs. This approach facilitates rapid creation of on-demand robotic systems by enabling automatic synthesis of software for both low-level control and high-level behaviors.

In order to allow complex customized software to be automatically generated for each new robot, an infrastructure has been created in which a software template is filled in and adjusted according to the user-driven design. Experts can define general code templates for using devices such as microcontrollers, and then write controller-independent code snippets for controlled devices such as servos. Each new device added to the library that requires code can add a simple snippet outlining code relevant to itself, and optionally use predefined code tags that enable dynamic adaptation to the design topology such as pin assignments and the presence of related devices. Algorithms included with the robot compiler can then pool together code from all components, process them according to the current design, and create coherent software packages. This flexible framework for processing software yields complex yet human-readable code to grant robots autonomous behavior and allow end users to easily write additional code.

Within this template, a collection of methods define how the data subsystem's ports and connections translate from the user-defined information flow to variables and functions that can be run by the final controller. Towards this end, a message format has been developed to standardize how ports are addressed and how information flows through the program – both from controller to controller on a robot, and between controllers and external devices such as wireless user interfaces.

4.1 Software Snippets

The software system is based on a collection of expert-defined snippets of code. As components are inserted and connected, their associated code snippets are filled in or modified to reflect the specific design, and then pooled together to create separate software packages for each controlling device. This approach allows the system to be very flexible and easily expandable, facilitates automated code processing and generation, and results in files that are accessible to human users that may want to expand upon the code.

For each device that will be executing code in the final implemented design, an expert can include a simple template for the main file when adding that device to the component library. As an example, there is currently an abstract subclass of `Microcontroller` called `Arduino` with a code template that simply has empty `loop` and `setup` methods. Each concrete subclass of `Arduino`, such as `ArduinoUno` or `ArduinoMega`, can inherit this main template while specifying the appropriate numbers of pins and types.

Similarly, each `CodeComponent` subclass requires code when connected to a controlling device, and includes code snippets that contain necessary methods or variables. For example, `Servo` inherits code from `DrivenElectricalComponent` then includes snippets for setting an angle or speed and, if needed, for startup calibration routines. Similarly, the abstract class `Driver` inherits from both `DataComponent` and `CodeComponent` then adds a code snippet that defines how the conceptual data network is realized in microcontroller code.

While the files specifying the main loop are controller specific, the system encourages every other code snippet to be independent of the controller with which it will be used. This allows the same snippets to be pulled from the component regardless of what microcontroller is chosen by the system or user, greatly increasing adaptability. This is enabled by files included in the library called *robot_ code.cpp* and *robot_ code.h* that declare methods and macros for the main initialization and loop, for reading and setting digital or analog pins, for dealing with PWM values, for setting pin types, and for other basic functions that are typically used by programmers when writing code for a microcontroller. Experts then define these methods in code snippets when adding each new microcontroller to the library. For example, the snippet of the `Arduino` class inserts calls to the *robot_ code*'s `robotLoop` and `robotSetup` methods into the Arduino `loop` and `setup` functions, respectively, and defines the `setDO` macro to call Arduino's default `digitalWrite` method. The code snippets of individual components can then use the microcontroller-independent methods from *robot_ code*, allowing them to run on any controller that includes those files. Note that the current system focuses on generating code for devices that run C++, but it is straightforward to extend this to other languages in the future.

```
@@declare
// Angle Servos
#define numAngleServos @deviceTypeCount
int angleServoPins[numAngleServos];

@@insert<void robotSetup()>
// Set up servo @deviceTypeIndex (angle control)
angleServoPins[@deviceTypeIndex] = @portID<signal>; // Connected to controller pin @pinNum<signal>
setServoAngle(@deviceTypeIndex, 0);

@@method<void setServoAngle(int servoNum, int angle)>
// Sets the angle of a servo
void setServoAngle(int servoNum, int angle)
{
  setPWM(angleServoPins[servoNum], servoAngleToDuty(angle, angleServoPins[servoNum]));
}

@@insert<void processData(const char* data, int sourceID, int destID)><@prepend>
// Servo @deviceTypeIndex (angle control)
if(destID == @dataInputID)
{
  int angle = (int) atof(data);
  setServoAngle(@deviceTypeIndex, angle);
}
```

Figure 4.1: This is most of a software snippet for the `ServoAngle` component. It supports multiple servos connected to the same controller, adds a new method for setting the angle of a servo, and inserts code for processing data sent to servo data input ports. The code tags, preceded by `@`, indicate robot compiler processing directives or placeholders for design-specific information.

Part of a sample snippet can be seen in Figure 4.1 for the `ServoAngle` component driver, which controls an angle servo and has a single data input port for the desired angle. Its associated code snippet declares variables for dealing with multiple servos connected to the controller, setting the length of the array to a design-specific value that corresponds to the number of servos. It also adds code to initialize these variables when the robot starts and to set the servos to a home position. A new method is created to set the angle of a servo, which uses microcontroller-independent functions defined in the generalized robot code for setting a PWM value and converting an angle to a duty cycle. Finally, it adds code to the `processData` method that checks if data is being sent to the servo's data input port and if so sets the servo angle accordingly. The design-specific information used in the declarations and in the new method are independent of the particular `ServoAngle` instance, and therefore the `CodeComposable` will detect the duplication and only include this code once. The initialization and data processing code, however, will be different for each `ServoAngle` instance once the tags are resolved – for example, each instance will be connected to a different microcontroller pin and have different data port IDs. The `CodeComposable` will insert

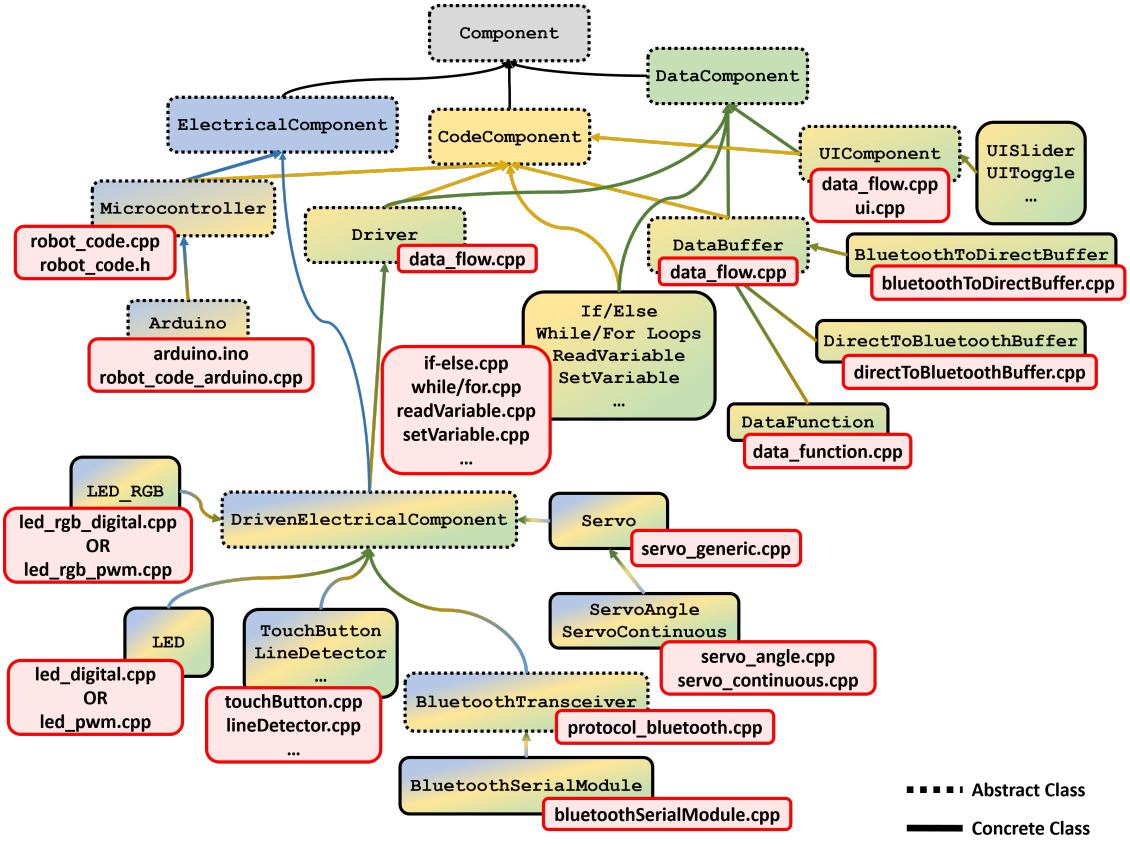


Figure 4.2: Software snippets can be included by any subclass of `CodeComponent`, and snippets will be inherited from superclasses. Shown here is the inheritance hierarchy among some prominent `CodeComponent` classes, where the overlapping red boxes indicate the snippets that each one adds.

separate code for each of these once the tags are resolved; for example, the `processData` method will contain a branch for each servo. Note that the included comments will make the final software package easier to understand, and can also include device-specific identifiers and information to better guide intermediate users.

Leveraging the inheritance structure among components described in Chapter 3 allows new components to inherit a substantial amount of code that will allow for proper functionality, greatly reducing the effort needed by experts when defining new capabilities. In addition, new derived components defined hierarchically by novice users will automatically contain all necessary code to function properly since the snippets will be gathered from all of the constituent subcomponents. Figure 4.2 illustrates some of the main `CodeComponent` sub-

classes and the code files that each one adds to designs. The included code then percolates throughout the hierarchy; for example, a `BluetoothSerialModule` will gain the software functionality of *bluetoothSerialModule.cpp*, *protocol_bluetooth.cpp*, and *data_flow.cpp* even though the first file is the only one it directly needs to add – the one that defines methods specifically for the serial module. Furthermore, components such as the `LED` can choose between code files to include based upon design-specific information, such as whether the user desires digital or analog control of the LED.

Within all of these files, code tags can be included that serve as processing directives for the robot compiler. These allow code snippets to create new methods and declarations or to insert code into methods that have been created by other components. In addition, code tags indicate how the robot compiler should adjust the code to make the generated software specific to the new robot. They therefore allow code from different components to affect each other and for the software package to grow organically as code is gathered from across the design and then holistically processed.

Some of the main template and snippet files that form the backbone of the generated software packages are described briefly below:

Generalized Robot Code (*robot_code.cpp*, *robot_code.h*) These files declare generalized functions for common microcontroller tasks, allowing future code snippets to be controller-independent. For example, they include an empty `robotLoop` function that other snippets will eventually populate with main control logic and a `robotSetup` function that contains code for configuring pin types by calling the generalized `setPinMode` function. These methods can all be extended by future code snippets as needed. They also declare arrays and variables that are used by the controller-independent framework, as well as methods and macros that will be defined by controller-specific files such as setting and reading pin values.

Microcontroller-Specific Main File (ex. *Arduino.ino*) This declares methods for the main setup and loop functions, which are typically empty at this point, in the form that will be expected by the microcontroller programmer such as the Arduino IDE. It also includes a code tag that tells the robot compiler that it is the main file, so appropriate include statements can be inserted here.

Microcontroller-Specific Robot Code File (ex. *robot_code_arduino.cpp*) These files define the generalized robot code functions for a specific controller, and insert calls to the `robotLoop` and `robotSetup` methods into the appropriate methods declared by the microcontroller-specific main file. For example, *robot_code_arduino* inserts a call to `robotLoop` in the `loop` method. It also defines the general robot methods such as those for configuring, reading, and setting pins or dealing with PWM values.

Data Flow (*data_flow.cpp*) The microcontroller-independent snippet defines how the conceptual data subsystem is realized as code, allowing data to flow between various ports to implement user-specified connections and behaviors. This will be described in more detail below.

Component-Specific Code Snippets These are included with each `CodeComponent` to define methods and variables used for that particular component. For example, the `Servo` class can define methods for setting angles or speeds, the `TemperatureSensor` can define methods for communicating with an I^2C temperature sensor, and the `LineDetector` can include startup calibration routines. These are written using the framework created by the higher-level microcontroller-independent functions. They can also extend methods created by other snippets and use code tags that resolve to design-specific information such as the chosen microcontroller pins or data port IDs.

4.2 Data Network Realization

One of the most important functions of the software template infrastructure is to define how the conceptual data subsystem is translated into concrete code. This allows ports to pass information between each other, both internally on a single controller and between controllers whether wired or wireless. This exchange of information allows the robot to be controlled from user interfaces, allows intermediate users to write their own behavioral code, and allows the robot to implement autonomous behavior described by the user-specified information flow.

This network is translated into code by the *data_flow.cpp* file included by top-level components that inherit from both `DataComponent` and `CodeComponent`. The main features of this file are described below.

- Declares variables and arrays that are used to define the data network's topology and connectivity. These include a 2D array describing the data mapping (i.e. port connections), an array of unique data output port IDs that correspond to those used in the mapping, and additional preprocessor definitions for creating and accessing these arrays. It also includes an array that specifies whether each port connection is set to "auto-poll"; this controls whether data should be retrieved from the output and sent to the connected inputs on every controller execution loop, or whether data only flows in response to some asynchronous trigger such as a UI button press.
- Declares and defines the `processData` method, which loops through every data connection that is set to auto-poll. It gets the most recent data from the output and gives it to the attached input(s) for processing.
- Inserts a call to the `processData` method in the main `robotLoop` function, thus ensuring that the data connections will be processed once per controller execution loop.
- Declares a `getData` method meant for data outputs that takes as arguments the output port ID from which data is being requested and, optionally, the input port ID of the requester. Component code snippets will add code to this method that checks if their port is being requested and, if so, performs any necessary operations to return data.
- Declares a second `processData` method meant for data inputs (distinguished from the main `processData` loop by its arguments) that takes as arguments the data to process, the input port ID that is receiving the data, and, optionally, the output port ID that is sending the data. Component code snippets will add code to this method that checks if their port is being targeted and, if so, performs any necessary operations based on the new data.

```
@@insert<char* getData(int sourceID, int destID)><@prepend>
// Data function @deviceTypeIndex
if(sourceID == @dataOutputID)
{
  int input = (int) atof(getData(@dataInputSourceID)); // Get input data
  if(!validGetData) // If there was an error, return empty string
  {
    outputData[0] = '\0';
    return outputData;
  }
  validGetData = true; // Indicate that the returned data is valid
  itoa((int) (@param<function>), outputData, 10); // Evaluate the user-specified function
  return outputData;
}

@@insert<void processData(const char* data, int sourceID, int destID)><@prepend>
// Data function @deviceTypeIndex: process data and forward the result to connected components
if(destID == @dataInputID)
{
  int input = (int) atof(data); // Convert the input to a number
  char outputData[10];
  itoa((int) (@param<function>), outputData, 10); // Evaluate the user-specified function
  // Forward the result to components connected to the data output
  for(int dataOutput = 0; dataOutput < NUM_DATA_OUTPUTS; dataOutput++)
  {
    if(dataOutputIDs[dataOutput] == @dataOutputID)
      processData(outputData, dataOutput, dataMapping[dataOutput], DATA_OUTDEGREE);
  }
}
```

Figure 4.3: The `DataFunction` block has a code snippet that performs a user-specified operation on any data sent to its input port, and forwards the result to ports connected to its output port.

Together, these variables and methods provide a coherent framework for realizing the data subsystem on the robot. They enable the requesting and receiving of data via output and input ports, and for automatically processing connected ports once per controller loop. Component code snippets then extend these methods to provide device-specific implementations of how data is acquired and processed.

The snippets shown in Figure 4.1 and Figure 4.3 illustrate some sample usages of this framework. The `DataFunction` has a single input and a single output, and performs a user-defined data transformation. It implements this behavior via the code snippet shown in Figure 4.3. It inserts code into the `processData` method that processes the data according to a dynamic user-specified transformation and then forwards the result to any ports connected to its output port. Code is also inserted into the `getData` method, which in turn calls `getData` on the port connected to its input before processing it and returning the result.

Both of these demonstrate how a component can add code to the general data flow methods that check whether their particular component is being addressed, and if so perform appropriate operations. In addition, they demonstrate how the data flow methods can be chained together to create complex behaviors.

This framework for realizing the conceptual data network is controller independent and very flexible, accommodating arbitrarily complex data topologies subject to the restrictions of the controller's performance capabilities. A combination of preprocessor directives and optimizations have been used in the template files to help make the processing efficient, allowing real-time control code to be defined by the high-level user specifications. Much of the memory-intensive arrays can also be stored in program memory, freeing the controller's RAM – this is important for small low-power controllers attempting to implement large data networks. The controller-specific code snippets can include definitions for how to store arrays in program memory, so the *data_flow* file can still be controller independent.

Using this infrastructure, component snippets can include their own logic for getting and processing data. This may include modifying pin states, adjusting system state, setting variables, or performing arbitrarily complex logic. It can also include using the `getData` and `processData` methods on other ports, allowing for the chaining of data components; for example, the main loop can call `getData` on the output of a data manipulation function block, which may in turn call `getData` on the ports connected to each of its inputs. Furthermore, these connections can be evaluated in a synchronous manner, once per controller execution loop, or in response to asynchronous triggers such as UI events or wireless communications. In this way, the method stack can become quite complex and sophisticated logic can be implemented with these relatively simple methods.

4.2.1 Data Message Formats

Components' method definitions may include logic for dealing with varied communication protocols such as wireless user interfaces. A message format has been developed that can be standardized across communication modes on the robot, such as wired serial or Bluetooth. This allows for sending data between ports on different controllers in a human-readable manner, and can be extended in the future for increased speed. Superclasses can provide code snippets that apply the protocol to their particular communication method, and concrete subclasses can then inherit the general framework and only define device-specific methods.

The message format specifies whether data is being sent or requested, and includes the port IDs involved in the transfer.

- For sending data, a message will be sent of the form `DATA$data$OutputID$InputID$` where the data `data` is being sent from port `OutputID` to port `InputID`. The `OutputID` is optional, and the `InputID` may be a list of hyphen-separated IDs. Additionally, `InputID` may be omitted if `OutputID` is included, in which case it is understood that all inputs connected to that output should receive the data.
- For requesting data, a message will be sent of the form `GET$OutputID$InputID$` where `OutputID` is the port from which data is requested and `InputID` is the port requesting the data. The `InputID` is optional. In response to this message, a controller sends a `DATA` message in the above format that includes the relevant port IDs; the sent and received messages are not directly linked, eliminating additional protocol overhead.

It is further asserted that each message will end with a null-termination character, namely `\0`. This messaging format can be used by many different communication methods, such as wired serial, I^2C, SPI, or Bluetooth. It can also be easily leveraged by intermediate or expert users who want to develop applications to directly interface with their robot; a list of the generated port IDs and their mapping is presented as one of the `DataComposable` outputs.

An example implementation of this protocol is found in the subset of devices that communicate via Bluetooth. The most prominent of these components are those that use user interface elements on devices such as smartphones. To define the messaging system for Bluetooth, the abstract superclass `BluetoothTransceiver` provides a file called *protocol_ bluetooth.cpp*

to declare methods for dealing with Bluetooth communication. It stores state such as whether or not Bluetooth is connected, which can be useful for switching between autonomous or controlled modes of operations, and for getting and sending data from or to ports via Bluetooth. Similar to the *data_flow* file, the *protocol_Bluetooth* file defines a framework that is then extended by concrete subclasses. The following are the main features of this framework:

- Declares data buffers for received messages and state variables for whether Bluetooth is connected. Inserts code into the `robotSetup` method to initialize this state.
- Declares and defines a `getBluetoothData` method, which reads data from a Bluetooth connection into a data buffer, and returns whether or not a complete message is ready to be processed. This method is non-blocking, meaning that it will not wait for messages to start or finish. It will simply add any available data to a running buffer and return whether or not a null-termination character was received. It leverages `getBluetoothChar` and `bluetoothAvailable` methods, which are implemented by concrete subclasses to deal with particular Bluetooth devices.
- Declares and defines a `sendBluetoothMessage` method, which sends a method over Bluetooth. It leverages a `sendBluetoothChar` method that is implemented by concrete subclasses to deal with particular Bluetooth devices, and ensures that the proper message format is used.
- Declares and defines a `processBluetoothData` method, which uses `getBluetoothData` to check if a complete message is ready and if so parses the message according to the standard format. If data is being received, it will call `processData` on the appropriate ports. If data is being requested, it will call `getData` on the desired output port and send a new message with the returned data using the `sendBluetoothMessage` method.
- Adds some global state to manage the frequency with which data messages are sent to particular Bluetooth devices. This is useful for ensuring that message rates are reasonable, especially when connections involving Bluetooth ports are set to auto-poll and would therefore try to update on every controller execution loop.
- Inserts a call to `processBluetoothData` in the main `processData` method, ensuring that Bluetooth messages will be processed once per controller execution loop.

This extends the data flow template to address concerns particular to the wireless application, such as asynchronous message arrivals and potential data corruption. Concrete subclasses of `BluetoothTransceiver`, including `BluetoothSerialModule`, only need to define device-specific methods such as how to send and receive single characters. This allows the library to be easily enhanced with new devices and protocols with minimal extra effort.

The above description outlines how a controller would deal with incoming messages from Bluetooth devices that are requesting or sending data. To complete the possible use cases, components representing the external elements such UI elements also typically allow the robot to query their output data ports. Their code snippets simply insert code into the main `getData` method that checks if their port ID is being requested, and if so sends a data request message over Bluetooth for the relevant data. It does not, however, wait for a response since this would incur large delays in the controller execution loop. Instead, it takes advantage of the fact that the request and data messages do not need to be coupled – it sends a request and then assumes that the external device will send the data to the desired ports soon. As described above, the general Bluetooth template includes code for managing the rate with which these messages are sent, ensuring that the external devices and the microcontroller are not inundated with messages. When a response data message arrives, it will be processed as usual and the input ports will receive the necessary data. This means that Bluetooth data may not be synchronized with data gathered from local ports, but this can be easily dealt with by snippets that require both external and local data.

Together with the data flow code, this allows components on the robot to communicate with off-board components via Bluetooth. The novice user using the design GUI, or the intermediate end user writing additional code, need not worry about how the communication is implemented behind the scenes or even whether the devices are physically connected. They need only focus on the data flow throughout the system via various ports.

4.3 Code Tags: Design-Driven Flexibility

While the above template system creates an extensible method of auto-generating complex code according to the design's components, this code needs to be adjusted to suit each unique component topology. To achieve this, a system has been created that allows experts to include code tags and processing directives in their snippets that will be interpreted and processed by the robot compiler when generating complete software libraries.

There are two main types of code tags: those that serve as processing directives, and those that serve as placeholders for design-specific information.

Processing Directives start with `@@` and include commands such as creating new methods, inserting code into existing methods, or declaring variables. When the `CodeComposable` processes the code files, it will pool together all declarations into header files, determine what methods are created by all components to split their declarations into header files and their definitions into code files, and will make desired insertions into existing methods. Processing directives can also be more complex, for example including `if/else` statements that condition upon design-specific information, or including loops whose conditionals and bodies are dependent upon the design. Some processing directives supported by the current system and their functions are shown in Table 4.1, where italicized text would be replaced by the snippet author.

Figure 4.4 shows an example snippet for the `UIComponent` class that uses some of these directives. Since it consists of declarations and initializations, the `declare` directive is used; this code will therefore be split between header files and code files as appropriate. The `uiDataMap` tag will resolve to an array initialization that creates a 2D array of port IDs indicating the data port mapping among ports related to the user interface. The `uiDescriptions` tag will resolve to a list of strings, each of which describes a UI element in a format understandable by the user interface device. To create an array of these strings, the `iterate` directive is used to loop through the resulting strings. When processed by the `CodeComposable`,

Table 4.1: Sample Processing Directives

Directive	Description
`@@iterate<`*tag*`>` `<@` *code* `@>`	Evaluates the provided tag, such as the data port mapping, and then creates a copy of the provided code for each item in the resulting list. In each of these copies, `@iterItem` will be replaced by the current item and `@iterIndex` will be replaced by the current item index.

`@@if<@`*condition*`@>`
`<@then` *code*`@>`
`<@else` *code*`@>`
	Evaluates the provided condition, which is given as Python code, and then yields either the code after `then` or the code after `else`. Each of these can themselves contain more tags and directives.

`@@file`	This file will be included verbatim, without additional processing.
`@@fileMain`	This is the main file for a microcontroller, so include statements will be inserted here.
`@@declare`	Declare and initialize methods or variables. When appropriate, will be split up such that declarations will be added to header files and initializations/assignments will be added to code files.

`@@insert`
`<`*methodDeclaration*`>`
`<`*code*`>`
	Insert the provided code into a method that matches the provided declaration. Optionally, can specify `@prepend` or `@append` to ensure that the code is inserted at the beginning or end of the existing method.

`@@method`
`<`*methodDeclaration*`>`
`<`*code*`>`
	Create a new method. A method declaration will be added an appropriate header file, and the definition will be added to the corresponding code file.

```
@@declare
// UI Descriptions
#define numUIDescriptions @numUIDescriptions // number of UI elements
#define uiRequest "UI_DESCRIPTION"           // sent by requesting UI devices
const char* const uiDataMap[] = @uiDataMap;  // data mapping for UI ports
const char* const uiDescriptions[] =         // UI elements, names, types, etc.
{
  @@iterate<@uiDescriptions><@
  uiDescription_@iterIndex,   // UI element number @iterIndex
  @>
};
```

Figure 4.4: The `UIComponent` class includes a snippet that, among other operations, declares an array of strings that describe the desired interface. It leverages the iteration-processing directive to loop through all added UI elements and add a string to the array for each one; these strings will then be sent to the UI device such as a smartphone.

this will resolve to an array initialization with a string literal for each UI element; a string descriptor will be obtained for each item in the list and inserted into a copy of the iteration code with the `iterItem` replaced by the string. These strings will be sent to the user interface device, such as an Android smartphone, to dynamically create the desired interface. This represents a relatively simple yet powerful example of using the processing directives, and more sophisticated design-dependent logic can easily be implemented as well.

Placeholders for Design-Specific Information start with `@` and may reference information such as port IDs, chosen microcontroller pins, or device counts. They can also take input arguments, allowing them to become more sophisticated. These will be processed by the `CodeComposable` and resolved to final values when software packages are generated. Some tags supported by the current system are shown in Table 4.2. Tags resolving to lists can be used with the `iterate` processing directive, but if they are used by themselves then they will be resolved to C++ compatible array declarations.

Example code snippets that leverage these tags have been shown throughout this section in Figure 4.1, Figure 4.3, and Figure 4.4.

Table 4.2: Sample Code Tags

Directive	Description
`@param<`*`paramName`*`>`	Evaluates to the value of the `Component` parameter with the given name. This enables direct access to user-specified configuration parameters.
`@portID<`*`portName`*`>`, `@dataInputPortID <`*`portName`*`>`, `@dataOutputPortID <`*`portName`*`>`	The port ID assigned to the component port with the given name. For the latter two, will only look for data input or output ports. If the port name is omitted, will use the first port found.
`@dataInputSourceID <`*`inputPortName`*`>`	The port ID assigned to the data output that is connected to the data input with the provided name.
`@pinNum<`*`portName`*`>`	The microcontroller pin assigned to the component port with the given name.
`@deviceName`	The name of this component.
`@deviceTypeCount<`*`type`*`>`	The number of devices of the given `Component` type connected to this microcontroller.
`@pinToDeviceTypeIndex <`*`pin`*`><`*`type`*`>`	The assigned index of the device that connects to the specified microcontroller pin number, among the connected devices of the specified `Component` type. This index may correspond to the index into a declared array of `Servo`s, for example.
`@dataMappings`	A 2D list of data port ID mappings, indicating the connected input and output ports. Will be used by the data flow code snippet.
`@controllerPins`	A list that indicates the microcontroller pins associated with each connected `Port`.
`@controllerPinTypes`	A list indicating the pin type for each microcontroller pin, such as digital inputs or PWM outputs. This will be used by the general robot code that configures pin types at startup.
`@uiDescriptions`	A list of strings, each of which describes a UI element in the design, which can be sent to a UI device such as a smartphone.

The commands allow snippets to be very general, supporting multiple components of the same type on a microcontroller and adapting to different configurations. The developed syntax creates a powerful framework in which experts can write their snippets in a generic yet straightforward way, allowing a sophisticated software library to be autonomously generated from basic building blocks to control complex robotic systems.

4.4 Summary

The software and data subsystems provide a foundation for interaction and communication. They allow information to flow among disparate subsystems, furthering the co-design paradigm and allowing users to specify complex behaviors by simply expressing how different parts of the robot should respond to each other and to the environment. To realize these relationships, a flexible software template model pools code snippets together from across the design and modifies them to reflect the unique topology. The class inheritance structure allows subclasses to inherit snippets from their ancestors, and hierarchically composed components contain all necessary snippets for proper functionality via their subcomponents.

This code can reflect unique topologies by leveraging code tags and processing directives. The robot compiler resolves these tags to design-specific information, increasing the applicability and flexibility of each snippet. Furthermore, a standardized message format is implemented that allows data to be passed between ports regardless of the underlying communication channel such as wired serial or Bluetooth.

Together, these additions allow the robot compiler to generate low-level control code, high-level behavioral code, and user interface code for each unique robot. The code and message format are both human readable, granting intermediate users the options of adding their own code or communicating with the robot from external devices once the robot is fabricated. The robot compiler can now not only generate the structure and electrical layout for the robot, but also provide all necessary control logic and drivers so that the user can immediately use their robot for the desired task.

Chapter 5

Custom Serial Communication Protocol

The single biggest problem with communication
is the illusion that it has taken place.

– George Bernard Shaw

Contents

While a framework has been discussed for relaying data between controllers to implement the desired data network, an effective means of transmitting that data needs to be employed. To address this issue, a custom two-wire clock-driven serial protocol has been designed and implemented to realize the communication channels created among connected components.

The new protocol allows microcontrollers connected in an arbitrary topology to communicate via software serial ports while eliminating message collisions, avoiding buffer overflow, reducing data corruption, allowing transmitters to know when transfers fail, and reducing the need to rely on sensitive timing functions. This protocol requires minimal additional overhead, and even increases communication speed for longer messages.

In particular, the designed protocol and associated software library achieve the following without requiring dedicated hardware or more than two wires:

- allow transmissions to be asynchronously initiated by any controller, but maintain synchronization using a clock signal within each transmission
- ensure that transmissions will only commence if the receiver is ready to receive the message and has buffer space to store the entire message
- mitigate simultaneous message collisions by ensuring that a receiver will only ever agree to receive messages on one serial port at any given time
- reduce the reliance on precise timing functions by using an interrupt-driven protocol based on a synchronizing clock signal generated by the receiver
- speed up the transmission of longer messages as compared to standard serial UART
- provide support for atomic transmissions of entire messages
- provide methods for transmitting and receiving that are drop-in replacements for existing software serial libraries, in addition to methods for reliably transmitting and receiving complete messages even in a complex mesh network
- add the ability to switch a port between the new protocol and the standard protocol, for compatibility with traditional serial devices

This new library can be used by the robot compiler to allow different kinds of microcontrollers to reliably communicate in complex networks via a standardized protocol without requiring experts to provide such functionality for each new microcontroller added to the library. Experts should only need to provide helper methods for managing interrupts on various pins of the controller, and timing functions for small delays that do not need to be as precise as for standard software serial protocols.

5.1 Existing Protocols and Chosen Approach

Most microcontrollers that will be used on compact, inexpensive, rapidly fabricated robot systems will contain a restrictive number of pins and an even more restrictive amount of hardware dedicated to implementing communication protocols. Since the goal is to enable multiple simultaneous communication channels, virtual ports will typically have to be used for communication. These mimic the functionality of hardware ports via code that uses the main processor rather than dedicated hardware to implement the chosen protocol. This provides additional flexibility at the cost of consuming processor resources. The chosen protocol must therefore be efficient, including only performing computation when transmissions actually occur, while still minimizing wiring complexity to both reduce pin usage and facilitate fabrication by novice users.

There are various communication protocols to choose from when transferring data between controllers. At a high level, data can be transmitted using parallel ports or can be serialized via time division multiplexing. Approaches leveraging parallel ports significantly increase the number of wires and thus the physical complexity. Additionally, there are associated timing difficulties, especially at higher speeds, since all pin values must be read simultaneously. Issues such as these make serial approaches more attractive for the task at hand.

Serial protocols can either be asynchronous or synchronous, depending on whether a clock signal is used to maintain synchronization between the transmitter and receiver. A very popular asynchronous protocol is the Universal Asynchronous Receiver Transmitter (UART), descriptions of which can be found in [71, 72]. This allows transmissions to be initiated asynchronously, using start and stop bits to alert the receiver to incoming transfers. Once a transfer begins, precisely tuned timing is employed to maintain synchronization and ensure that the receiver reads data during the correct intervals. The protocol is relatively simple and has minimal setup, making it an attractive option for controller networks with intermittent messages. However, there are significant challenges when implementing this as a virtual port rather than using dedicated hardware, and these issues are exacerbated when creating an auto-generated mesh network of controllers. For example, if multiple software serial ports

are in operation on a single-processor microcontroller and data is received simultaneously on more than one port, all involved messages will likely be irrecoverably corrupted, since the ports will interfere with each other's timing, or one message will be ignored depending on the implementation. Furthermore, there is no way of knowing whether a receiver is ready to receive a message, if their buffer will overflow at some point during the transmission, or if they are currently receiving a message from a different transmitter. Finally, the typical software serial implementation may be cumbersome to implement in a system that focuses on auto-generating code in a microcontroller-independent manner, since sensitive timing must be implemented differently for each particular microcontroller; this would lead to extra effort on the part of the experts to provide a serial library for every new controller that they wish to add to the library.

The timing issues associated with asynchronous protocols can be addressed by including a synchronizing clock signal, typically on a dedicated wire. For synchronous communications, data is sent as a continuous stream at a constant rate [73], thus often eliminating the need for start and stop bits between every byte of data. Popular synchronous protocols such as SPI or I^2C, which are described in [71, 72], can connect many devices together while using a small number of microcontroller pins and achieving high data rates. The intended recipients of messages can be indicated by dedicated lines or by assigning addresses to each device. However, these are master/slave protocols that rely upon a single master initiating transmissions; a slave device can only send to the master if the master polls the slave for data. Since the target application discussed here desires that any controller may send data asynchronously to any other controller, and choosing a master to frequently poll devices would be computationally expensive and inefficient, these protocols are not well suited.

Other commonly used synchronous protocols include High-Level Data Link Control (HDLC) and Manchester encoding. HDLC is often used in Wide Area Networks (WANs), and transferred frames include fields for a unique flag as well as an address and checksums – for a description of the protocol, see [73]. The flag is a series of bits that indicate the start and stop of a transmission, and bit stuffing is used to ensure that this code does not appear anywhere else in the data stream. This therefore achieves synchronization, but at the cost

of significantly increased complexity and overhead that would place heavy demands on the processor and be inefficient for sending relatively short messages. Manchester coding embeds the clock signal within the data stream by encoding bits as level transitions and thus ensures frequent rising and falling edges. This eliminates the need for a dedicated clock wire, but at the expense of doubling the required bandwidth and requiring complex decoding circuitry that implements a digital phase-locked loop [74]. These drawbacks make this an undesirable option for an auto-generated network of controllers in which computational cost and assembly complexity should both be minimized.

Taking the above considerations into account, a protocol is developed here that merges the simplicity of UART with the synchronization benefits of clock-driven protocols. It is loosely based on the serialization techniques of the standard UART, but adds a clock signal generated by the receiver in response to a transmitter's start bit. This structure allows for interrupt-based synchronization throughout data transfers, and also provides a built-in mechanism by which the receiver constantly acknowledges that it is ready to receive data and confirms that bits were successfully gathered. As is typical of synchronous methods, start and stop bits are unnecessary once a transfer begins, so longer messages can be sent faster than if standard UART was used. The protocol still only requires two wires, keeping fabrication complexity low, and is well suited to software implementations since it eliminates message collisions and has low computational requirements.

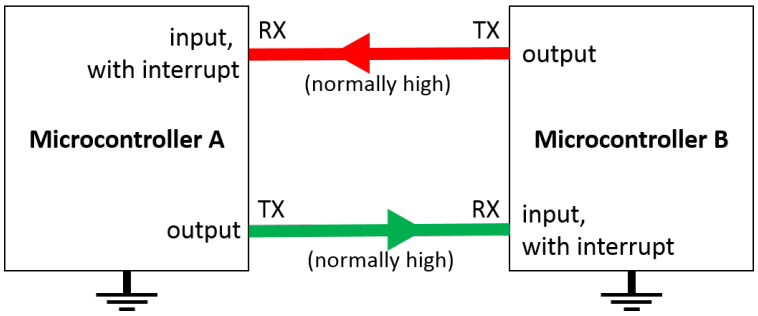


Figure 5.1: Serial communication requires two wires per communication channel, and data flows in a fixed direction on each one. The receive pin (RX) of one controller is connected to the transmit pin (TX) of the other controller. For normal logic, the voltages are high when at rest.

5.2 Standard UART Serial Protocol

The standard serial protocol traditionally implemented by serial devices is described here, which will be used as a baseline for evaluating the new protocol and whose serialization techniques will be leveraged. The communication channel is implemented by two wires that connect receive (RX) pins and transmit (TX) pins on each device as shown in Figure 5.1. An interrupt is typically attached to the RX pin on each microcontroller to reduce processing overhead and ensure timely responses to transmissions. The atomic unit of data transfer is a byte, so that messages are transmitted by individually transmitting each byte as a new message; detecting message terminations and verifying integrity is handled by user-defined higher-level logic.

The procedure for transmitting a single byte using the standard asynchronous serial protocol is illustrated in Figure 5.2. Either device can initiate a transfer, in which bytes are sent as serialized bits and the receiver and transmitter are not continuously synchronized. Start and stop bits are required, and a parity bit can be optionally included to provide some basic error detection. Before the program initiates, both controllers agree on a baud rate, start and stop bits, parity bits, and whether the transmission will be ordered with the least significant or most significant bit first. For example, a common configuration is to use 9600 bits per second (bps or baud), start and stop bits each one bit-width long, and no parity bit. In this case, a single 8-bit data byte requires $(8+2)\,bits \times \frac{1\,second}{9600\,bits} \approx 1.04\,ms$ to transmit.

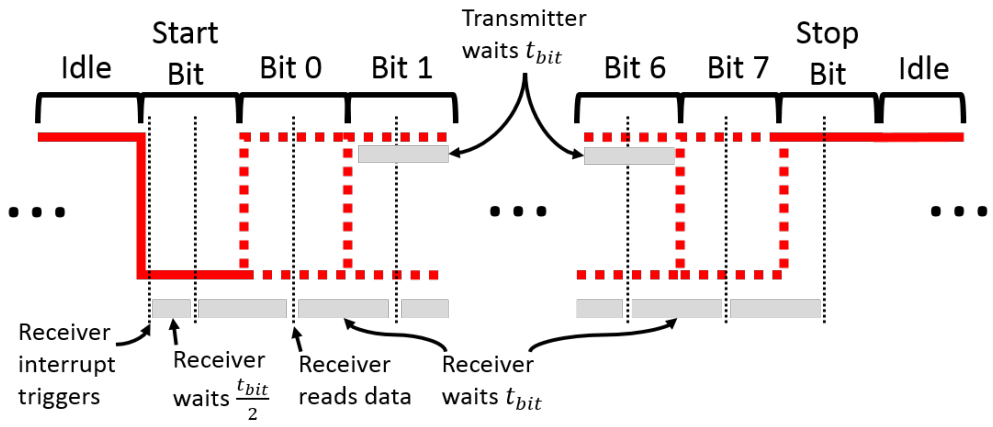


Figure 5.2: The standard UART serial protocol uses a start bit to establish synchronization, then tuned delay functions during transmission. An optional parity bit may also be transmitted between bit 7 and the stop bit. The indicated sequence happens on the transmitter's TX pin; the receiver's TX pin remains idle throughout the transfer. The baud rate defines the duration of each bit, represented here by t_{bit}.

When implemented as a virtual serial port, with software mimicking the functionality of a hardware serial port, interrupts are attached to the RX pins and then sensitive timing functions are used to maintain synchronization throughout a byte. A typical software implementation, included with the Arduino environment, can be found in [75]. Once the receiver detects a start signal using an interrupt, it will use a carefully tuned delay function to delay half of one bit-width. It will then similarly delay a full bit-width between reading each bit as it assembles the incoming byte. The transmitter will also use a tuned delay function to determine when to change its TX output value. This means that the receiver will read its RX pin at the center of each transmitted bit, but only if the interrupt triggers immediately and if the delay functions on both the transmitter and receiver are precisely tuned to within about 5% of the ideal delays. This can lead to a variety of complications for virtual serial ports, some of which are mentioned below:

- If the timing on either controller is unacceptably erroneous, if significant clock drift occurs, or if something else occupies the processor's attention during the transfer, then the synchronization will be lost and the data will be corrupted. This is particularly troublesome if the microcontroller is using interrupts to monitor other events such as sensors or timers in addition to the serial communication, since these will interfere with communication and cause data loss.
- If both devices attempt to send data at the same time, it is likely that the data will be corrupted or that both messages will be ignored. Depending on the implementation and assuming a single-processor microcontroller, receiving an interrupt will cause the transmission process to be paused or interleaved with the receive process. Either option which will adversely affect the timing.
- If multiple serial ports are operating on a controller, then their interrupts will interfere with each other if messages are received on more than one simultaneously. Depending on whether interrupts are disabled during interrupt routines, messages will be ignored or all messages will likely be corrupted due to timing issues.

In all failure modes, the senders will be unaware that their message was not successfully received and the receiver may not know that corrupted data is invalid. There are no checks for such issues integrated within the protocol, so higher-level logic is required to mitigate collisions, ensure that no other interrupts will interfere with communications, and check whether data is valid. These algorithms may need to be rewritten for each unique application that uses serial communication. While this protocol is often sufficient for two devices communicating intermittently, it becomes troublesome when many microcontrollers are frequently communicating in a mesh network.

5.3 New Clock-Driven Two-Wire Serial Protocol

The main innovation of the new protocol is to address reliability issues by synchronizing the transmitter and receiver using a clock signal generated by the receiver in response to an asynchronously initiated data transfer. Since the transfer is still asynchronous, the processor is not used unless a transmission is in progress. The clock then synchronizes the two controllers, reducing the reliance on precisely tuned delay functions. Furthermore, having the receiver generate the clock provides a built-mechanism by which the receiver acknowledges that it is ready to receive the message.

While the standard protocol technically allows for full duplex operation, meaning that messages can be sent and received simultaneously, this is not typically true of its software implementations. Since most microcontrollers used in the target applications have a single processor, only the receive method or the transmit method can be running at any given time; both cannot run concurrently, and interleaving the two via threads would corrupt data by interfering with the sensitive timing. This means that whenever a controller is receiving a message, its TX pin is unused. The new protocol therefore temporarily re-purposes this pin to transmit a clock signal for synchronization. This is facilitated by the fact that the receiver already has this wire configured as an output, and the transmitter already has an interrupt listening on this wire. Using this insight, only two wires are needed between the devices – the wiring for the new protocol is identical to that depicted in Figure 5.1.

5.3.1 Overview and Properties

The new protocol is illustrated in Figure 5.3. The atomic unit of data transfer can be either a single byte or a sequence of bytes. If sending a single byte, the data is immediately sent after an initial handshake. If sending a sequence of bytes, the first transferred byte indicates the length of the incoming sequence to allow the receiver to check that it has sufficient buffer space.

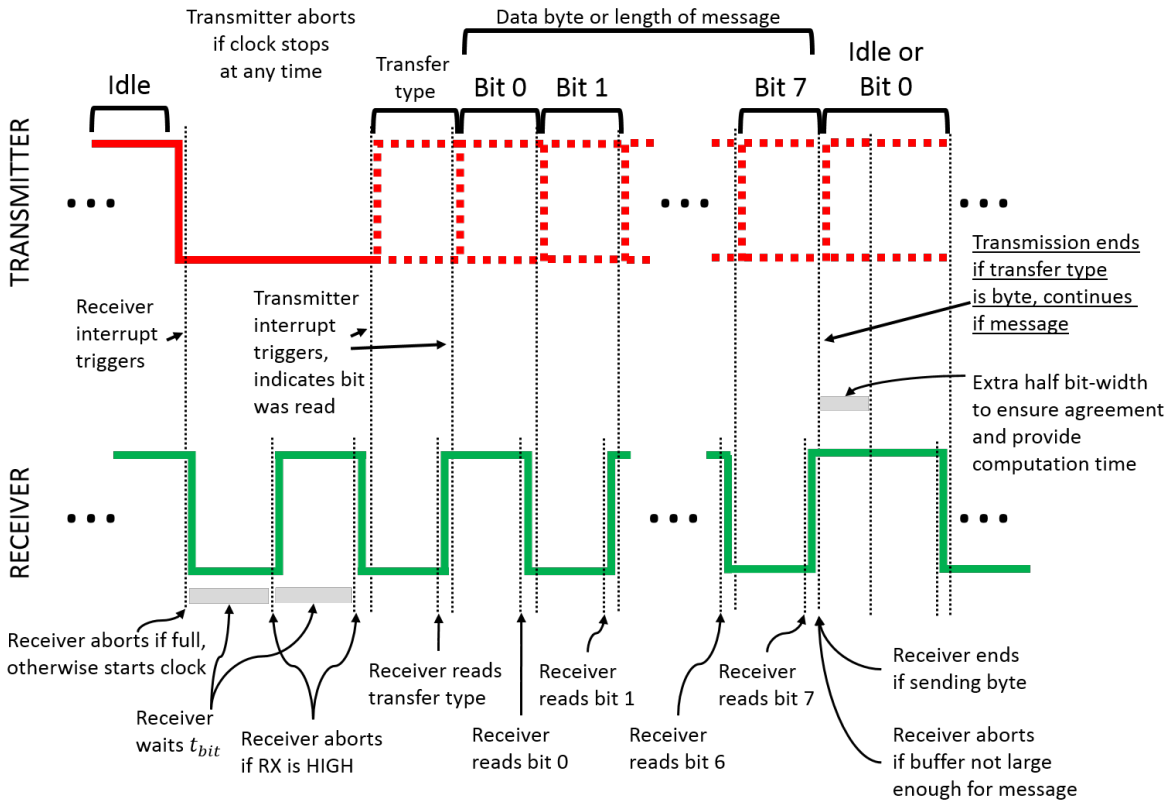


Figure 5.3: The new protocol synchronizes controllers during transmissions with a clock signal generated by the receiver; the top red signal is generated by the transmitter, and the bottom green signal is generated by the receiver. The initial three bit-widths establish synchronization, ensure that both controllers are active and in agreement, and indicates whether a byte or a message will be transferred.

When the transmitter asynchronously initiates a transfer, the receiver starts generating a clock pulse on its TX line if it has at least one byte free in its buffer. Two bit-widths then elapse as the receiver ensures that the transmitter is still waiting to transmit, and then the transmitter indicates whether a single byte or a message will be sent. During these three bit-widths, the transmitter also ensures that the receiver is sending a clock signal. This initial exchange addresses the possibility of both controllers attempting to initiate a transfer at the same time; they would each determine that the other is not sending a clock, and accordingly abort the transfer as failed.

By generating a clock, the receiver acknowledges that it is ready to receive and effectively issues a promise to not respond to any other messages that may arrive on other active

serial ports during the subsequent communication. If at any point the transmitter does not receive a clock edge in more than 1.5 bit-widths, it assumes that the receiver has rejected its request or crashed, and therefore knows that the transfer has failed. The provided higher-level methods that deal with sending messages can then decide to retry or to abort the attempt. As a result, complete message transfers can be viewed as atomic in the absence of microcontroller crashes or similarly anomalous events: either the receiver successfully receives the entire message, or the message is not transmitted.

A transfer only commences if the receiver is not currently receiving any messages on other ports, agrees to ignore future requests during the transfer, and has sufficient buffer space to receive the byte or message. During the subsequent transmission, the clock signal synchronizes the controllers such that bits are ready to be read when the receiver reads its pin. The receiver reads its RX pin just before creating a clock transition, and the transmitter sets its TX pin to the next level as soon as it detects a transition. Thus, each clock edge serves as a confirmation to the transmitter that the receiver received its latest bit. This also means that the transmitter has nearly an entire bit-width to ensure that its TX pin has reached steady-state at the appropriate level before the receiver reads it.

If a single byte is being sent, the transmission ends as soon as the last bit is read. If a message is being sent, the receiver checks that it has enough room in its buffer for the entire message and aborts if it does not. Otherwise the transmission continues, with the data bytes being sent in direct sequence; no start or stop bits are required between message bytes since the controllers have already been synchronized and the clock maintains that synchronization. An additional half bit-width is added at the end of each byte, however, to provide a little extra time for computation at the transmitter and to ensure that both controllers are still in agreement about their current state.

Since the initial synchronization exchange is 3 bit-widths, and the transferred byte has 8 bits, there is an odd number of bit-widths in the complete transfer and the final clock transition that confirms the last bit was read will be from low to high using normal logic. The transfer thus ends in the idle state, and no additional transition is needed to restore

normal resting levels. Note that this is true for messages transfers as well as single-byte transfers – since each data byte requires an even 8 bit-widths and no start or stop bits are needed between message bytes, the entire transfer will contain an even number of clock transitions regardless of the message length.

In this protocol, the receiver defines the timing of the transfer instead of the transmitter. This is somewhat more intuitive, since the transmitter wants to know that its sent bits are received and the receiver will be performing computation to store the incoming bits and check that its buffer is sufficiently empty. Since clock transitions are important to the protocol rather than the specific direction of the transition, the receiver can generate a clock signal that has a transition once every bit-width as defined by the baud rate; the clock period is thus twice the length of one bit-width. This means that the protocol does not increase the bandwidth by including a clock, as is necessary for an approach such as Manchester encoding. If it had been enforced that each transfer occurs on a rising edge, for example, a half of a bit-width would need to elapse between transitions and the clock period would be one bit-width, which greatly increases the demands on the microcontroller's timing resolution. Note that the receiver uses a tuned delay function to generate the clock signal and the transmitter uses a tuned delay function to check for timeouts, but the accuracy of this timing is no longer critical to successful data transmission since both controllers are not simultaneously trying to accurately time one bit-width. If the clock frequency drifts or is slightly off from ideal, the achieved baud rate will be affected but the data will still be reliably received since the transmitter uses interrupts to remain synchronized.

5.3.2 Implementation

A number of particularly interesting implementation issues can be considered when realizing this protocol as microcontroller code. One such issue is the efficient use of processor power between bit transitions, to ensure that data is ready to be read by the next clock transition and to facilitate the highest baud rates possible. Towards this end, the transmitter stores a variable that indicates the bit to be written at the next transition. When a transition

is detected by the interrupt attached to the RX pin, this variable is immediately written to the TX pin before any computation is performed. After this is complete, a variable that masks the byte being sent is shifted and the next bit to transmit is stored. This therefore moves the computation of the next bit from before the TX pin state is changed to after the TX pin state is changed – this gives the TX pin a full bit-width to reach steady state and minimizes the amount of processing that needs to be done between clock transitions. The more expensive task of storing the next byte to be sent and initializing the mask that selects the next bit can be performed before the transmitter initiates the communication, or during the extra half bit-width of time between bytes that the protocol provides.

The transmitter must also be able to detect if at any point the clock signal ceases to be transmitted, since this indicates that the receiver rejected the request or crashed. Once a transmission is started, the main transmit method waits in a loop while the interrupt routine performs the tasks of actually writing the data at the appropriate times and computing the next bit to be written. This loop can therefore be used to check that the expected amounts of time are passing between transitions; delay functions that wait approximately half of a bit-width are used in the loop to check if more than 1.5 bit-widths has elapsed since the last bit was written by the interrupt routine. If so, it can assume that the clock has stopped and that the transfer has failed. Note that interrupts can be triggered during the delay functions, so if the timing is not precise the transfer will still continue as expected.

Regarding the receiver, the protocol indicates that it must essentially promise to not respond to other data requests for the entire duration of a transmission. This is currently accomplished by disabling RX interrupts on all other virtual serial ports on the receiver just before the clock signal starts to be generated. If other controllers try to send data to the receiver while its interrupts are off, they will time out as described above since their write interrupts would not be triggered within 1.5 bit-widths; they would thereby determine that the receiver rejected their request. At the end of a transmission, either because of successful transfer or because insufficient buffer space is available for a message, the other RX interrupts are re-enabled.

If a receiver detects a request when its buffer is completely full, it will reject it by not generating a clock signal since it cannot accommodate the data regardless of whether a single byte or a message is sent. In this case, it also briefly delays before returning to the user. This ensures that if the user tries to transmit data from the receiver immediately after a rejected request, the rejected requester will not interpret the new request as the beginning of a clock signal. The protocol would still work without this delay, since the initial handshake would reveal that they are not in agreement, but the delay speeds up rejection since only the 1.5 bit-width timeout is needed as opposed to three bit-widths.

A single byte is currently used to indicate the length of a message. Interpreted as an unsigned integer, this implies an inherent message length limit of 255 bytes. Higher-level methods can be used to split longer messages into smaller chunks if necessary.

5.3.3 Transfer Speed

The increase in reliability comes with a slight speed reduction if sending individual bytes, since some overhead is added to ensure that the receiver is ready and to convey whether a single byte or a message is being transferred. Specifically, a single-byte transmission requires 11 bit-widths as opposed to the 10 bit-widths required by standard UART with no parity. However, the new protocol offers a speed increase for byte sequences. Start and stop bits are unnecessary after the initial handshake, reducing the number of bits per message byte.

As a result of these overhead reductions, the time per message byte decreases with increasing message length. More specifically, Equation 5.1 gives the total time to send a message and Equation 5.2 gives the time per message byte. The duration of a bit as defined by the baud rate is represented by t_bit, and N is the number of bytes in the message.

$$\text{time to send a message} = t_{bit} \times (3 + 8 + 8.5 \times N) \tag{5.1}$$

$$\text{transmission time per message byte} = (8.5 \times t_{bit}) + \left(\frac{t_{bit} \times (3 + 8)}{N}\right) \tag{5.2}$$

The additional 3 bit-widths in Equation 5.1 represent the overhead discussed earlier for initializing a transfer, and the additional 8 bit-widths represent the byte that indicates the number of bytes in the transmitted sequence. Each message byte requires 8.5 bit-widths since the receiver waits an additional 0.5 bit-widths before starting the clock pulse for a new word to ensure that the transmitter saw that the receiver acknowledged the last bit. This also provides a little extra computation time for the transmitter to prepare the next word, and ensures that the transmitter always has a full clock period to transition its TX pin to the next bit level – this can be vital when dealing with high baud rates and relatively slow microcontroller clocks such as the 8 MHz offered by an Arduino Pro Mini.

Since the standard protocol must include start and stop bits with every byte, the time per message byte is a constant 10 bit-widths without parity. Equation 5.3 compares this to the transfer time for the new protocol. For low values of N, this fraction will be larger than 1 and the transmission time for the new protocol will be longer than for the standard protocol. However, this fraction will reduce as N increases. If the message has more than 7 bytes, the transmission time for the new protocol will be shorter than for the standard protocol as indicated by Equation 5.4. For example, a message of 20 characters will be transmitted in approximately 90.5% of the time it would have taken with the standard protocol, saving about 2.0 ms at 9600 baud. As the message length continues to increase, Equation 5.3 will approach a constant 0.85 since the initial synchronization time becomes less significant and the number of bit-widths per message byte, defined to be 8.5, dominates. If 255 bytes are sent, currently the effective maximum message length, it will take approximately 85.4% of the time taken by standard UART and save about 39.8 ms at 9600 baud.

$$\text{fraction of standard UART time} = \frac{t_{bit} \times (3 + 8 + 8.5 \times N)}{t_{bit} \times (10 \times N)} = 0.85 + \frac{1.1}{N} \tag{5.3}$$

$$\text{speed increase if } t_{bit} \times (3 + 8 + 8.5 \times N) < t_{bit} \times (10 \times N) \Rightarrow N > 7.\overline{3} \tag{5.4}$$

5.3.4 Provided Library

The clock-driven serial protocol is particularly suited to transmitting frequent messages in a mesh network where many virtual serial ports will be used. Transmissions are initiated asynchronously and the lines can be left idle when no transmission is ongoing, so the processor is only used during communication. Synchronization is established by a brief bit-level handshake that ensures the receiver and transmitter are in agreement, and a clock signal maintains synchronization throughout the transmission. The protocol maintains synchronization between transmitter and receiver despite potential interruptions or timing drift, ensures that receivers are capable of receiving messages prior to transmission, and ensures that multiple transmissions will not interfere with each other on any given controller.

A library is provided for using this new protocol as a drop-in replacement for existing software serial libraries. It includes methods for sending and receiving bytes, and also has new methods related to atomic message transfers. For example, methods are provided that automatically retry transmissions after fixed or random delays until a certain number of tries or a certain timeout is reached. In addition, methods are provided for switching between the new clock-driven protocol and the standard timing-based protocol to support traditional serial devices such as sensors or Bluetooth serial modules.

The provided library implementing the new serial protocol can be accessed at [76]

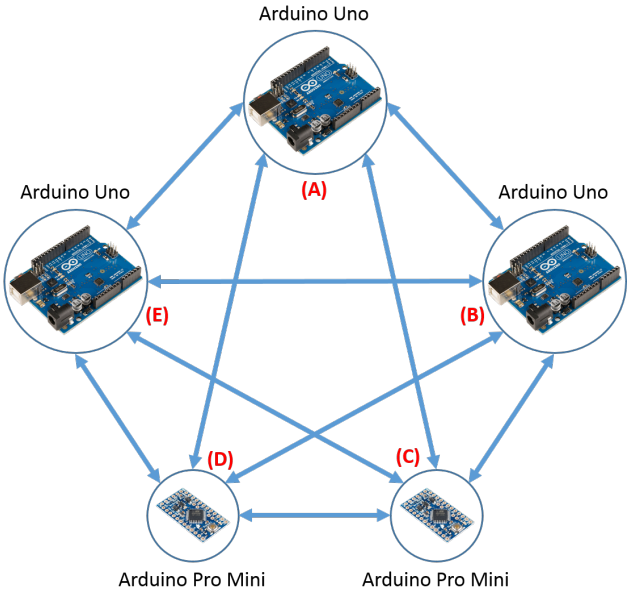


Figure 5.4: Five Arduino microcontrollers were wired together in a fully connected configuration to test the new serial protocol. Each one sent bytes or messages to each other controller at high rates, retrying when messages were rejected.

5.4 Experiments

Tests were performed to demonstrate the increased reliability of the clock-driven serial communication. Beginning with two microcontrollers and a single serial port, the numbers of ports and controllers were gradually increased to investigate scalability and the application to interconnected networks with frequent message transfers. Transmissions of single bytes and of complete messages were both tested. In addition, the baud rate was adjusted to determine whether the clock synchronization allows for higher communication rates by reducing the probability of timing errors.

The experimental setup on which the following tests were conducted featured five microcontrollers – three 16 MHz Arduino Uno's and two 8 MHz Arduino Pro Mini's – connected in a fully connected network configuration as shown in Figure 5.4. These controllers will be referred to below as **A** through **E**. Each controller was connected via a two-wire serial channel to each other controller; each controller therefore had four concurrent software serial ports running the new protocol. Note that the chosen Arduino boards only provide one

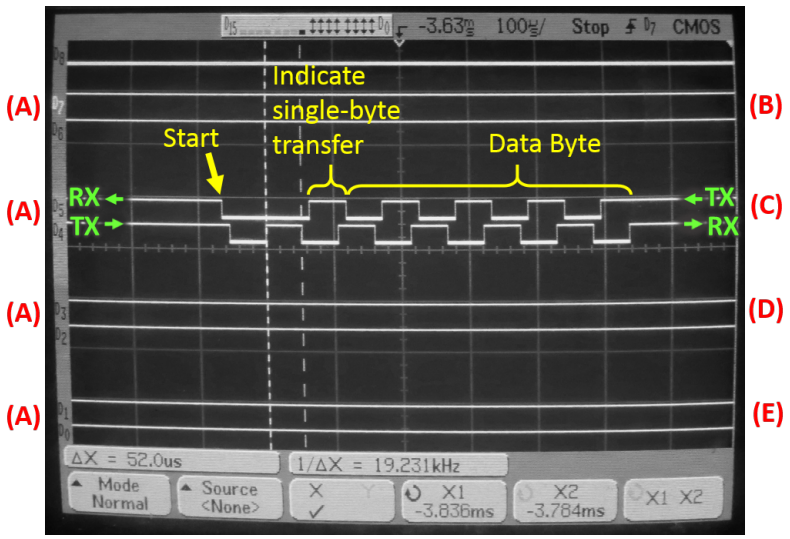


Figure 5.5: A successful single-byte transmission is recorded between microcontrollers **A** and **C**. The protocol operates as expected, and the period between clock transitions is 52.0 μs as desired for 19200 baud.

hardware serial port, which is used for USB communication, so the illustrated configuration requires the use of virtual serial ports. Each controller was programmed to continuously send bytes or messages to every other controller, using small random delays if their requests were denied. Each controller also maintained counts of messages successfully transmitted to each connected receiver. Every message then contained the value of this counter along with a sender ID to ensure that no messages were missed at the receivers.

5.4.1 Single-Byte Transmissions

The first experiment investigated the sending of individual bytes using the new protocol. Each microcontroller was programmed to send single-byte transmissions to each other microcontroller, with small random delays ranging from 0 to a few milliseconds. Within this scenario, the microcontrollers were able to successfully communicate without any corrupted data; transmitters were informed if a receiver was not ready to receive, and the clock signal successfully synchronized data transfers. The baud rate was typically held constant at 19200 bps but data was also reliably received at 38400 bps.

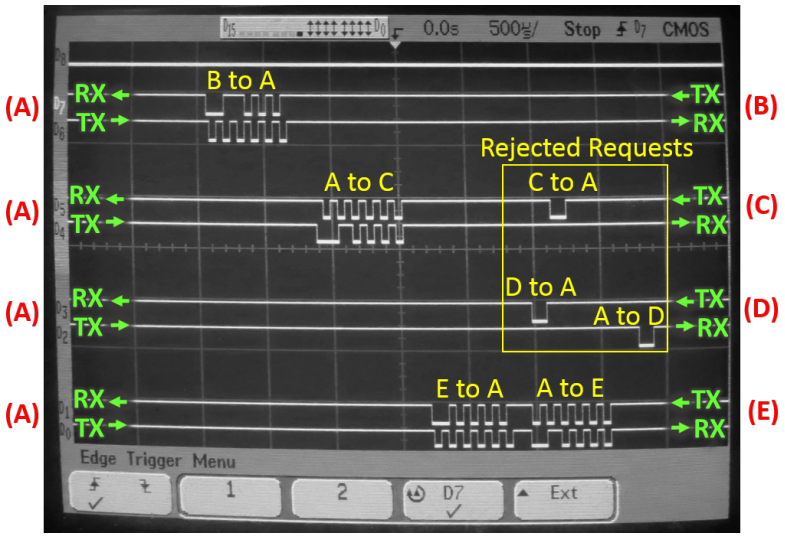


Figure 5.6: Bytes are sent back-to-back between microcontrollers **A**, **B**, and **E** in various directions. Attempts to send data to **A** while **A** is sending a byte to **E** successfully result in aborted transfers.

Figure 5.5 presents an oscilloscope trace of a single-byte transmission with the new protocol. Microcontroller **C** initiates the transfer, at which point microcontroller **A** begins generating a clock signal. The transmitter keeps its TX line low until the third bit-width, at which point it sends a high value to indicate that a single byte is being sent. The data is then sent as eight sequential bits: **A** reads each bit just before creating a transition, and **C** writes the next bit as soon as a transition is detected. The baud rate was set to 19200 bps, and the cursors indicate that the time between clock transitions is 52.0 μs as expected.

A more complex scenario is shown in Figure 5.6, where multiple single-byte transmissions are recorded. Four successful transfers can be seen between microcontrollers **A**, **B**, **C**, and **E**, many of which happen in quick succession. This indicates that directionality can be quickly reversed in accordance with which controller initiates the transfer. Within each exchange, the transmitter sends a high value on the third bit-width to indicate that a single byte is being transferred. Moreover, Figure 5.6 captures some requests that are rejected by the receiver. Microcontrollers **C** and **D** each attempt to send data to **A** while **A** is busy sending a byte to **E**. Since **A** is busy, it does not generate a clock signal for **C** and **D**, and they thus know that their requests have been denied and abort the transfer. There is also a rejected request from **A** to **D**, indicating that **D** is busy communicating over a channel not pictured.

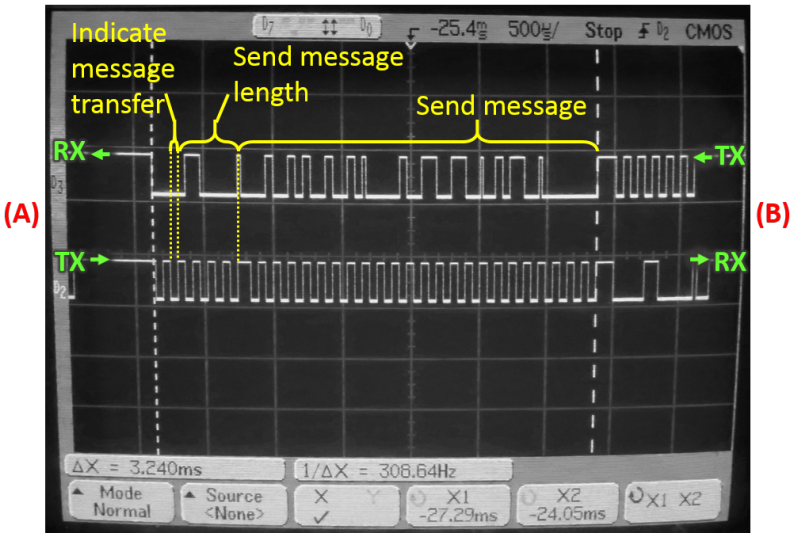


Figure 5.7: A message is successfully sent as an atomic transfer from microcontroller **B** to **A**. **B** indicates that a message is being sent, then transmits the size of the message, in this case 6 bytes. Since **A** continues to send a clock signal, **B** sends the entire message, in this case "HI 34\0".

5.4.2 Message Transmissions

In addition to sending individual bytes, experiments were performed where each controller sent atomic messages to each other controller. Similar to the previous case, messages were sent with brief random delays on the order of 0 to a few milliseconds.

Figure 5.7 presents an oscilloscope trace of an entire message being successfully transmitted from **B** to **A**. **B** keeps transmitting a low value on the third bit-width to indicate that a message is being sent, and then the subsequent byte informs **A** that 6 bytes will be sent in the message. **A** then confirms that its buffer is sufficiently empty for this message, and continues generating a clock signal. The message transfer then commences, with bits being sent on clock transitions and without start or stop bits between bytes. In this case, the null-terminated message "HI 34\0" is sent as a series of ASCII characters, where the 34 is an incrementing counter to ensure that no messages are missed at the receiver. Note that since this experiment was only between two controllers, the sender ID was omitted for this case. As indicated by the oscilloscope cursors, the entire transfer requires approximately 3.240 ms, which is within expected error of the 3.229 ms that Equation 5.1 predicts given

that the microcontrollers were configured for 19200 baud. The additional 11 μs is likely a combination of the time it takes **A**'s initial interrupt to trigger, slight imperfections of the timing functions, and measurement error regarding the oscilloscope cursor placement.

In the configuration with five interconnected microcontrollers, a nominal baud rate of 19200 bps was typically used but the maximum rate observed without any errors was 38400 bps. Higher rates started to reveal a limited amount of intermittent data corruption, likely due to the slower clock speed of the Arduino Pro Mini, which is 8 MHz as opposed to the 16 MHz of the Arduino Uno. A similar test was therefore performed with only the Arduino Uno's. In a fully connected configuration, all messages were successfully transmitted between these three controllers with baud rates up to about 57600 bps. The observed baud rates were typically very close to the nominal baud rates, as indicated by Figure 5.5 and Figure 5.7. As previously discussed, however, data will still be reliably transferred even if the timing functions are not as precisely tuned. Cases were observed in which the processors were temporarily consumed by other tasks and the clock signals were correspondingly distorted, but the data was still successfully transmitted.

Message Collision and Buffer Overflow Avoidance

With all five controllers interconnected and sending messages, the protocol successfully managed collisions and buffers, such that all messages were eventually received from all controllers. The numbers of messages received from each other controller was also relatively evenly distributed on each device, indicating that certain devices were not dominating the communication channels. The sender IDs and counters included in the messages also indicated that no messages were missed or corrupted at the receivers; there were no instances in which the protocol assumed a message was delivered successfully when it actually was not.

Figure 5.8 demonstrates the successful mitigation of message collisions that would cause data corruption on typical software serial port implementations of standard UART. Microcontroller **C** attempts to send data to **A** while **A** is busy receiving messages from **B**. Since **A** has effectively promised **B** that it would not respond to any other incoming requests during

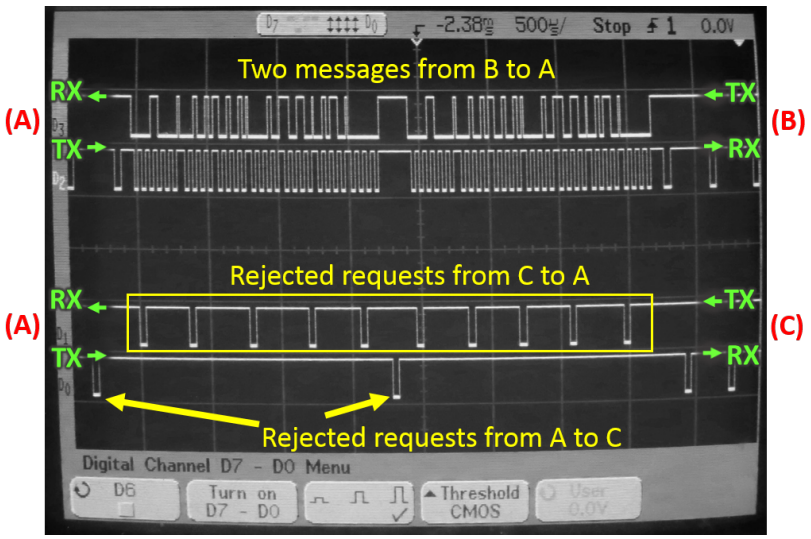
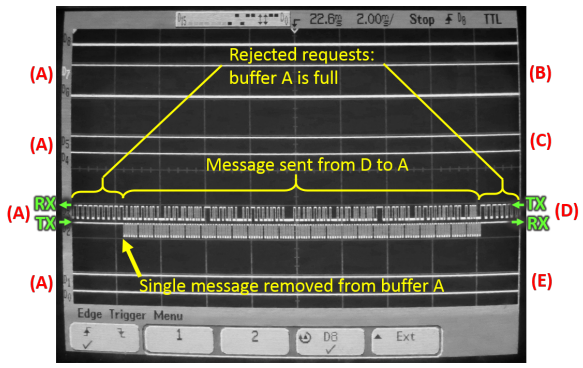


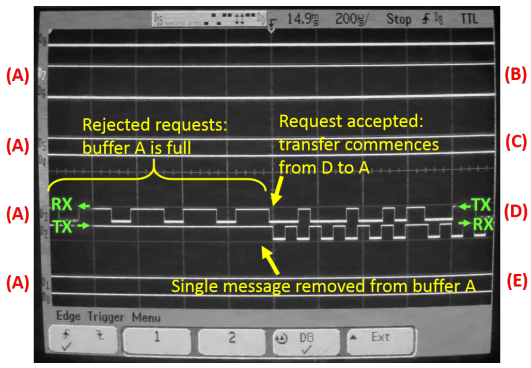
Figure 5.8: Microcontroller **C** attempts to send a message to **A** while **A** is busy receiving a message from **B**. No clock signal is generated since **A** has promised to listen exclusively to **B**, so **C** correctly aborts the transfers.

the transfer, it does not generate a clock signal for **C** so the attempts are aborted. **C** then knows that the attempt failed, does not increment its message counter, and tries again after a short delay. Similarly rejected attempts are seen from **A** to **C**, which are likely due to **C** being busy attempting to communicate with another microcontroller, performing other interrupt-driven tasks, or having a full buffer.

In addition to avoiding corruption due to simultaneous message transmissions on separate virtual ports, the protocol also successfully avoided buffer overflow. To test this case explicitly, the code on one microcontroller was altered such that it would only sporadically read from its buffer, thus allowing it to fill up in between flushes. The buffer size was also chosen to be an integer multiple of the message length, ensuring that the buffer would become completely full. Figure 5.9 presents a segment of such a trial. At the beginning of the recorded trace, **D**'s attempts to send data to **A** are all rejected by **A**. Note that no clock signal is generated at all, indicating that **A**'s buffer is completely full – it does not even wait to learn how many bytes the message contains. At one point, however, **A**'s code reads a message from the buffer and thus **A** can now accept **D**'s transfer request. The message is then successfully sent as an atomic unit. After the transfer, **A**'s buffer is once again full and subsequent requests are again denied.



(a) Requests to send data from **D** to **A** are immediately denied when **A**'s buffer is completely full. This is both before and after the message transfer, since a single message was flushed from **A**'s buffer and the buffer size is chosen to equal the message size.



(b) Since there is no free space in its buffer, **A** does not generate a clock signal at all, i.e. it does not wait to learn how long **D**'s message would be. As soon as there is free space, however, it can respond to incoming requests.

Figure 5.9: The protocol successfully avoids buffer overflow by allowing receivers to reject requests when their buffers are full. In this case, **A** denies requests from **D** until a message is flushed from **A**'s buffer, at which point the transfer can proceed.

Another interesting scenario that demonstrates the elimination of corruption due to buffer overflow is captured by Figure 5.10. The middle section of this trace records attempts to send messages from **A** to **B** and from **A** to **C**. The receivers' buffers are not completely full, so they generate clock signals and receive the byte that indicates the message length. They then determine that their buffers are not empty enough to hold the entire message, and cease generating a clock signal. **A** then detects that a transition was not received within 1.5 bit-widths, and correctly infers that the transfer was denied. The figure also includes some rejections of the type previously seen, where the receivers are busy and therefore do not respond at all.

Using the standard Arduino software serial library in these same configurations and with the same control logic, almost every message experienced some degree of data corruption. The new protocol therefore successfully mitigates both message collisions and buffer overflows, providing a reliable method of atomically transferring messages between controllers without hardware serial ports. In addition, the baud rates achieved are generally higher than the baud rates that were achieved in tests with the standard library where the control

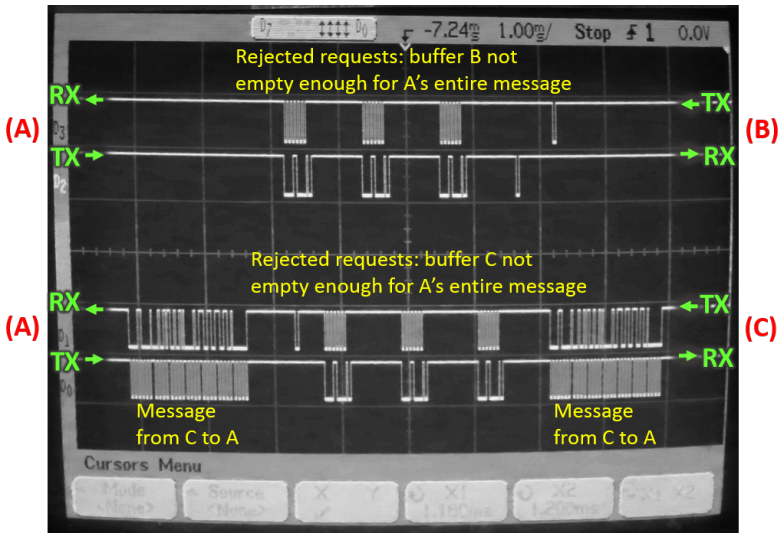


Figure 5.10: If a receiver's buffer has some free space but is not sufficiently empty to hold the entire message of a prospective transmitter, the attempt will be aborted after the message length is transmitted.

code was altered such that only one message is transmitted at a time. This is a result of the decreased reliance on precise software-defined timing functions and increased reliance on interrupts for synchronization.

5.5 Summary

The presented serial protocol allows microcontrollers to communicate over a two-wire interface using software-defined virtual serial ports even in the presence of multiple communication channels per controller and frequent message collisions. It combines the simplicity of standard UART with the timing advantages of synchronous clock-driven protocols to reliably transmit data without heavy reliance on the precision of tuned timing functions. Moreover, having the receiver generate the clock signal provides a built-in mechanism by which the transmitter knows that the receiver is ready to accept data and when each bit has been successfully read. Since the protocol re-purposes the receiver's TX pin for this clock signal, the communication still only requires two physical wires.

This approach manages data flow through complex mesh networks, allowing controllers to use software serial ports while eliminating data corruption due to message collisions. The additional overhead incurred for single-byte transmissions is negligible, and transmitting messages is actually faster as compared to standard UART at the same baud rate if the message is longer than 7 bytes. Another consequence of this protocol's message transfer is that byte sequences can be treated as atomic units. This can be very useful for high-level users, and the library includes methods for sending and receiving messages that automatically retry upon failure and that automatically incorporate null termination to demarcate messages within a receiver's buffer.

Experiments demonstrate that the current implementation operates as expected and successfully enables reliable communication in adversarial configurations. Message collisions and buffer overflows were avoided even with five microcontrollers rapidly sending bytes or messages in a fully connected configuration. Achievable baud rates were also generally higher with the new protocol than with standard software UART, due to the clock signal's synchronization.

This approach therefore facilitates communication among interconnected microcontrollers, a capability that is vital to the integrated systems designed by the robot compiler. Microcontrollers can now be inserted throughout a design and connected by serial communication channels without worrying about managing potential complications; the necessary guarantees are embedded within the protocol itself at the bit level. This reduces the effort required by experts when adding new microcontrollers to the library, and also grants the robot compiler additional autonomy and versatility. Together with the software template model, the robot compiler can now generate complete designs that include not only electrical and mechanical layouts but also control and communication logic.

Chapter 6

Design Algorithms: Generating Robots

"Data! Data! Data!" he cried impatiently. "I can't make bricks without clay."

– Sherlock Holmes

Contents

The provided library of electromechanical modules allows users to intuitively design robotic systems. Components can be chosen and connected without worrying about implementation details – the modular infrastructure handles integrated subsystem designs behind the scenes. As the library expands and more components are chosen, it is important to have efficient algorithms to aid the user and generate final fabricable outputs.

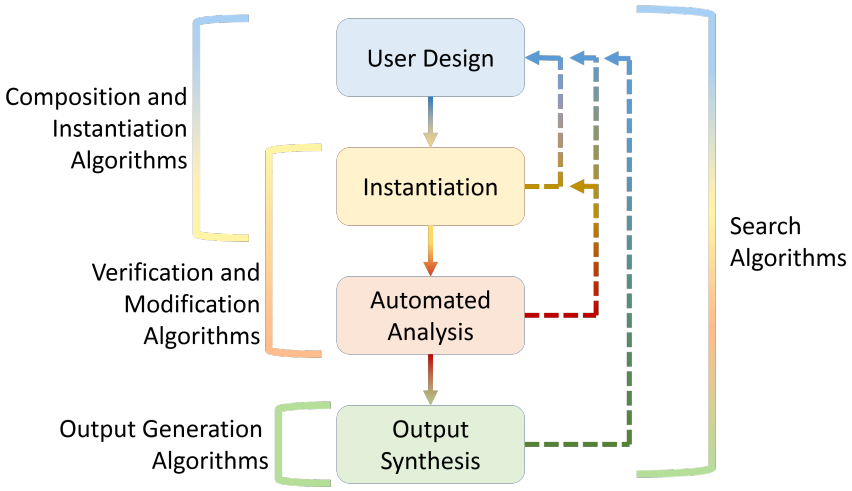


Figure 6.1: Suites of algorithms operate on collections of components throughout various stages of the design process. The dashed lines indicate iteration loops that may be followed by the user based upon feedback from integrated tools as the design proceeds. Each successive design phase uses more computational power, so this flow also represents a division among algorithms to make design more efficient and responsive.

Towards this end, algorithms have been developed to operate on collections of components. As illustrated in Figure 3.1 and Figure 6.1, they operate in various stages of the design process and can provide varying levels of feedback to the user. In addition, each successive design phase provides more sophisticated feedback at the expense of greater computational cost; the computation is thus compartmentalized and divided into different categories of algorithms so that they can run at strategic intervals and maintain a responsive design environment. There are generally four main categories of algorithms included in the robot compiler for these purposes:

Search Algorithms facilitate quickly selecting a subset of components from a larger collection. These allow users to query the library for components that meet certain criteria, making the design process faster and smoother. Search algorithms also allow the system to find components that it determines might work well in the design, allowing it to autonomously make design decisions or suggestions.

Composition and Instantiation Algorithms operate on a design hierarchy to instantiate the chosen components and compose their respective elements. These convert textual descriptions of components, parameters, constraints, and connections to Python object instances. The result resembles a linked list tree structure that represents the design topology. This is a middle ground between textual descriptions and the production of final output files, allowing for additional design verification and completion without the need to perform computationally intensive output operations.

Design Verification Algorithms analyze a collection of instantiated objects to verify and possibly edit the design. These can check lower-level requirements such as connected port compatibilities, or higher-level requirements such as components that depend upon each other for functionality. When possible, these algorithms autonomously adjust the design to address issues or otherwise interact with the user. Each subsystem may have verification processes that affect other subsystems, and thus these algorithms are often iterative to facilitate the co-design structure.

Output Generation Algorithms process the final collection of instantiated objects to produce integrated design outputs. These may include electrical diagrams and instructions, bills of materials, microcontroller software, UI software, and mechanical fabrication files.

There are thus a few levels of computation performed as components are composed. Search algorithms run on textual descriptions alone, or on a limited number of instantiated port objects. These computations can be run on demand, each time the user makes an action, or autonomously throughout the process to provide some user verification. At a lower frequency, the system can instantiate components to provide more feedback – verify parameter constraints, check component requirements and interdependencies, and proccess the hierarchy as a whole. This allows the system to make more refined suggestions, detect more errors, and even modify or auto-complete portions of the design. Only at the end of the design, once everything is finalized, does the system need to perform the more computationally intensive task of converting the stored subsystem information into complete output files.

6.1 Search Algorithms

As the user creates their design, it is helpful to be able to search the library for appropriate components. In addition, the system itself should be able to search the library to autonomously make suggestions and verify or adjust the design.

Search for components or ports based on class type

One available method for searching the library is to filter based on component or port type. For example, a list of all electrical components can be found by retrieving a list of components that inherit from `ElectricalComponent`. More sophisticated filtering can also be achieved by searching for components with multiple parents, or specifying a desired level in the ancestry tree. For example, one could request only components whose direct superclass is `ElectricalComponent`, or components that inherit from both `ElectricalComponent` and `CodeComponent` anywhere in their ancestry. Similarly for ports, one could request only `DataPort`s, or only ports that are `Output`s. All of these searches can be refined to only return concrete classes rather than abstract classes, thereby only showing results that can be directly used by casual users. In this way, a relatively refined subset of the original library can be obtained for presentation to the user or for autonomous analysis.

An advantage of this type of search is that it can be done without instantiating any of the component classes. The inheritance topology can be determined from the Python meta-information about the class definition without creating an object of the class. This makes the search much more efficient, allowing it to more readily deal with a rapidly expanding library. In the future, a tree representing the class inheritance structure can be stored as part of the library to facilitate this searching; as new components are created or modified, the tree is simply adjusted to reflect the change as necessary. This would amortize the cost of inheritance analysis and essentially pre-compute the class relationships, allowing desired class types to be retrieved from the library even faster.

Search for components based on port types

More refined searching can also be achieved by filtering components based on what ports they contain. For example, a user or algorithm may want to know all available components that support a PWM output port, or that can be connected to a selected component. As opposed to searching for components on an individual basis, this provides a higher-level design search that involves multiple components and their connectivity information. It facilitates organically growing an existing design, makes it easier to find suitable devices, and facilitates both user experience and autonomous design assistance.

The implementation of these searches can be made relatively efficient. If a desired port type is provided, and a list of components containing such ports is requested, then the search can usually be completed without instantiating component classes. This is because most of the components in the library are stored as textual yaml files, which include a list of ports and their class types. Thus, these textual files can be searched for the desired port type to find a list of suitable components. Components defined only as scripts or subclasses would need to be instantiated, but the `Component` superclass provides methods for saving itself as a yaml file; so components defined in this way could be saved as text files when they are created, allowing the search to only concern itself with textual representations.

If a desired port is selected and a list of components that can connect to that port is requested, then the search will need to perform additional work. It will need to instantiate the selected port, and execute its `canMate` and `shouldMate` methods with an instantiated object of each other known port type. Once a list of compatible ports is found, the previously described search can be conducted to find components with at least one matching port without instantiating the components themselves. Since in general there are far fewer port types than component types in the library, this search is still relatively efficient.

In the future, much of the information required by these searches can be precomputed and stored in a tree-like data structure. This would enhance the efficiency of the algorithms and potentially avoid the need for any object instantiation.

Search for components based on functionality

Although not yet implemented in the current system, the library could also be searched based on component function. Some information about the function can be automatically extracted simply from the component definition, such as what types of ports, subcomponents, and parameters it contains. Furthermore, the components could contain descriptions and keywords that describe the intended use and function. This meta-information could be partly user-defined, but also partly auto-generated. To some degree, functional descriptions for new components could be inferred from the meta-information of parent classes as well as embedded subcomponents.

Search for components based on previous designs

A higher-level search may be able to retrieve components from the library based on designs that have been created in the past. By analyzing a current design and its similarities to other designs (i.e. other components in the library), likely additions and topologies can be suggested to the user or automatically chosen to address errors. By allowing the system to learn from experience, it could provide more sophisticated aid to the user and even complete portions of the design on its own, allowing the user specifications to remain at an even more abstract level. This would also allow the library to expand more quickly and dynamically since the computer could make its own additions to the library with minimal user involvement.

6.2 Composition and Instantiation Algorithms

Once a user finds components in the library or creates derived components, they can connect them according to the desired information flow and compose components hierarchically to create more complex structures. As components are composed in this way, the system manages their encapsulated information for each subsystem and ultimately generates the final robot design files. By allowing each subsystem to be modularly encapsulated in the components, their designs can be created in parallel and interact with each other.

As the design is created, it is populated with information such as definitions of subcomponents, interfaces, interconnections, parameters, and parameter constraints or functions. The depth and breadth of the hierarchy can expand very quickly, greatly increasing the underlying complexity. Storing and modifying instantiated objects throughout this process could become unwieldy and inefficient.

To address this issue, the definition of a design by connecting and composing components is done at an abstract level without requiring the instantiation of class objects. The system can efficiently manage large designs of increasing hierarchical complexity by only storing the names of chosen classes, names of ports that the user chooses to connect, selected parameter values, parameter constraints and expressions, names of inherited interfaces, and so on. This results in a textual description of the design that can be saved in the `YAML` file format. In addition, it simplifies the processing required during the design phase and facilitates a responsive design environment.

To actively make suggestions and participate in the design process, certain information about objects can be precomputed or a limited number of objects can be temporarily instantiated. This allows the search algorithms to run, and for basic verification to be completed such as port type compatibility when connections are made.

6.2.1 Design Instantiation

Once a design is specified, it must be instantiated to begin the process of validation or output generation. Towards this end, the `Component` class has a method called `make` that processes the design hierarchy and creates fully defined Python objects for each component. It also instantiates the required `Composable` classes and informs them of the design topology so that they can store this information in a subsystem-specific manner. This method has a few consecutive phases, and is outlined in Algorithm 1.

Through these phases, the textual description of the component hierarchy is processed and converted into a collection of instantiated Python objects. The connections and subcom-

ponents are evaluated, the parameter relationships are resolved, and the subsystem design information is incrementally updated to prepare for outputting final design files. Note that `make` is an iterative method, traveling throughout the design tree by calling itself on every subcomponent. The nested subcomponents and the connections between subcomponent ports create a virtual linked list structure that represents the overall design. Once the call to `make` terminates, the component has a `Composable` instance for each subsystem represented in the design hierarchy, and each such instance has been informed of the design topology and appended with the `Composable` instances of parent components as the recursion percolates. The result is a single `Composable` instance for each subsystem, and the system is now ready to begin the verification or output generation processes.

The structure of this method is such that it can process each subcomponent quickly, simply instantiating the class and its various ports. The information about connectivity, parameters, and constraints is also evaluated by the `Composable` objects, being stored in a manner that will facilitate generating outputs, but a final design output is not yet processed. This means that `make` can be run periodically as the user is designing to allow for more in-depth analysis and design feedback while not requiring excessive computation.

6.3 Design Verification Algorithms

Once the design has been instantiated, automated verification and modification can take place. Towards this end, the `makeDesign` method of the `Component` class iteratively allows each `Composable` instance an opportunity to make design edits until all agree that no new changes need to take place. This iteration allows each composable to affect the other composables' decisions, facilitating an integrated co-design.

General requirements specified by each component can first be evaluated, and the design can be modified to satisfy them. Algorithm 2 describes how the `checkRequirements` method achieves this through an iterative process in which modifications by one composable can spur further changes by other composables through the co-design verification process. In the

current implementation, the main supported requirement type is that another component of a specified type be included in the design. For example, a certain user interface may require that a Bluetooth transceiver be included on the robot; in this case, the system will search for a concrete class inheriting from the abstract `BluetoothTransceiver` class and add it to the design. Connections will be determined later in the process as described below if not specified by the component requirement.

In addition to evaluating general requirements, each composable can also perform domain-specific verification and modification. The `makeDesign` method, outlined in Algorithm 3, will iteratively give each composable object an opportunity to do so. If modifications are made, general component requirements can be rechecked as described above. This sequence is repeated until no composable makes any changes and all general requirements are satisfied.

This `makeDesign` method is therefore a recursive, iterative algorithm that acts upon a user-specified design to make it ready for final compilation. It allows the `Composable`s a chance to interact with each other while making domain-specific decisions. This creates a co-design environment in which the electrical, mechanical, and software systems can be designed simultaneously.

6.3.1 Topological Sorting of `Composable` Objects

Some composables depend upon the work of other composables to behave appropriately and to make the most intelligent decisions possible. For example, the software composable should know about all microcontrollers that the electrical composable inserts, and the data composable should know about all UI devices that will be generating or processing data. Each `Composable` subclass can therefore store a list of dependencies, and these dependencies can be used to create a topological ordering. This topological sorting defines the order in which the composables should be processed when making design choices. For example, the `makeDesign` method asks composables for design verification or modification in this topological order to ensure best results and minimize the number of needed iterations.

6.3.2 Electrical Subsystem

The `ElectricalComposable` class will check the design using electrical design guidelines. For example, it will ensure that connected ports have compatible voltage and current requirements, and that microcontroller pins are configured correctly. If an electrical port in the design is unconnected, it will find an available pin on a microcontroller that supports the desired type and add the new connection. If power supplies are missing, it will choose an appropriate battery or other source. If a new microcontroller is needed, it will select one from the library based on the devices that need to be connected. It can either choose to add independent microcontrollers, or have a central controller command slave controllers such as the hardware modules described in Section 3.3.1 that are distributed across the robot.

In order to allow inserted microcontrollers to communicate with each other, the electrical composable can create a Minimum Spanning Tree (MST) of the controllers, and weight the edges according to the geometrical layout obtained from the mechanical composable. Serial communication links can then be added along the edges of the tree, creating a connected grid of controllers while attempting to minimize the amount of wiring. The software serial library described in Section 5.3 can be used to facilitate robust communications.

To choose microcontroller pins automatically, the ports that need to be connected can be assembled and then the controller pins can be checked to see if they support being configured as a compatible port according to the `canMate` and `shouldMate` methods. Configurations can then be chosen for the connected controller pins, defining their state and affecting future pin connections. While the current implementation uses greedy algorithms to find a matching between unconnected device ports and controller pins, in the future a network flow algorithm can be used to find all connections at once. A graph can be constructed, using the unconnected ports and the microcontroller pins, that respects the port compatibility requirements. Computing a minimum cost maximum flow on this graph would then calculate all needed connections, or determine that no possible connection topology exists. If the connections are not feasible, a new microcontroller can be chosen or slave controllers can be added for increased capacity.

If two incompatible ports are connected in the design, or no suitable connection between ports can be found, the `ElectricalComposable` will automatically search the library for components that can act as buffers by converting the incongruous parameters of the ports and allowing for a transitive connection. For example, a motor may be connected directly to a microcontroller output, but the current requirements would probably be mismatched. In this case, the composable would search the library for a subclass of `ElectricalBuffer` with inputs and outputs that can connect to the microcontroller and motor, respectively. In this case it would find a voltage buffer or motor controller that can drive the motor, depending on the necessary communication protocol. It would then break the connection between the motor and microcontroller, and add the new buffer in between. Similar scenarios can occur for voltages, in which case step-up or step-down converters would be located, or even for protocols, such as converting from PWM to analog values or from Bluetooth to wired serial.

If the composable cannot find a buffer that satisfies both sides of the connection, but does find some that satisfy one side, the algorithm will start a tree of possible buffers and iteratively branch the tree outward; the root will always connect to one side of the connection, and each leaf will represent a different port type that can be reached by some combination of buffers. The tree will then branch out as the library is iteratively searched for new buffers to add to the leaves, until the end port type is reached or no new buffer can be added. In this way, arbitrarily long chains of buffers may be found to convert between two ports that were originally incompatible. As a simple example, this may result in a current converter and a voltage converter in series. This allows the system to be very flexible, and to find complex solutions to design issues.

6.3.3 Data Subsystem

The `DataComposable` class will perform checks on the design that are similar to the electrical composable, but that relate specifically to the data network. It will check that all data ports have unique IDs, which will be used by the software templates and snippets for passing data between ports. It can also check the overall connection topology, and resolve errors or data conflicts. For example, two output ports may be connected to the same input port, which could result in a conflict at that input port. The composable may choose to break those connections and insert `Multiplexer`s, allowing for one of the outputs to be passed to the input according to a third input. The composable can choose an appropriate control input if known, but can also consult the user for advice or for a design change.

In addition, the composable will check that types and protocols have been fully specified, and can choose default values if they seem appropriate. It will then check that all connected ports have compatible data types and protocols. If incompatible ports are connected, or if two ports need to be connected, it may search the library for `DataBuffer` subclasses that can convert between data types or protocols and insert them as needed. The algorithmic structure to achieve this is the same as described above for the `ElectricalComposable`, and may result in a chain of buffers to transitively make the desired conversion. A user design that simply connects a toggle switch on a computer UI to a toggle switch on a smartphone UI may result in an XBee interfacing between the computer and a microcontroller and a Bluetooth module interfacing between the microcontroller and the smartphone – all needed implementation details and devices have been automatically chosen by the system to realize the abstract information flow desired by the user. This example also highlights an interaction between the `DataComposable` and the `ElectricalComposable`, since they each need to insert new devices in response to new modifications.

This allows intuitive user specifications that span multiple subsystems to be translated into fabricable designs. For example, a user's entire design may consist of a servo directly connected to a UI slider in order indicate that the servo should be controlled from a smartphone. When the design is verified, the system will notice that the slider's data output port has a

Bluetooth protocol specified, but the servo data input has a direct protocol specified; the system will therefore search for a suitable `Buffer`, find that the `BluetoothToDirectBuffer` is appropriate, and add this element as an intermediate in the servo-slider connection. This new component, in turn, has a requirement that a `BluetoothTransceiver` be included in the design. When the main component that started the design verification process re-checks the design requirements, it will see this as unsatisfied and insert a new component that inherits from `BluetoothTransceiver` – perhaps an instance of the `BluetoothSerialModule` subclass, which represents a Bluetooth chip that connects to a microcontroller via a standard two-wire serial channel. The `ElectricalComposable`, meanwhile, will respond to this by choosing an appropriate microcontroller for the unconnected electrical pins and wiring them appropriately. Thus the simple user-specified design now has everything needed to control a servo over Bluetooth from a smartphone UI. The `CodeComposable` will then generate all necessary software by pooling together and processing the various code snippets.

6.3.4 Code Subsystem

The `CodeComposable` has some basic verification to perform on the included software snippets. For example, it will check that each `CodeComponent` object in the design has an associated sink for its software. Typically, this means being connected to a UI device or a microcontroller. If one is not found, it will see if it is transitively connected to one (i.e. connected to a component that is connected to a software sink), attempt to find a sink elsewhere in the design that seems appropriate, or add a new one if needed and possible.

During the `make` phase, the `CodeComposable` class also begins to pool the software snippets described in Section 4.1 from various components into a coherent library, but not yet create the final output. This follows the paradigm of dividing labor between the instantiation and the output generation phases, allowing for preliminary analysis as the user creates the design.

This first stage consists of parsing each software snippet into declarations, methods, and insertions according to the processing directives described in Section 4.3. These are sorted into files and folders according to their associated controller. The result is a folder for each controller that contains a file of declarations, a folder of methods, and a folder of insertions. Each of the two subfolders contains a file for each relevant method. During this process, duplicates are eliminated by first replacing device-specific tags with tags that uniquely identify the source device, and then checking for code overlap. The replacements are necessary since two files with the exact same raw code may ultimately become different code once the code tags are resolved; for example, the port IDs of two LEDs will be different but referenced by the same `@portID` tag.

Once this processing is complete, a skeleton of the code library is obtained and can be analyzed. This can be particularly useful for users writing code using the graphical programming blocks described in Section 3.3.2, since those modules can now present lists of available methods and variables. This information is also now stored in a way that facilitates parsing by the output generation algorithm, which will process the code tags and create the final software packages.

6.4 Output Generation Algorithms

Once all of the general requirements are satisfied and each composable can find no more necessary changes, the design is completely specified and the output generation process can commence. This structure decouples the computation required for output generation from that required for design instantiation and verification, allowing for a more responsive and efficient design environment. It also facilitates the expert creation of new extensions and plugins to support new processes or outputs.

The output phase is encapsulated in the `makeOutput` method of the `Component` class. This essentially calls the `makeOutput` method of each composable object, in topological order. This ordering allows the outputs of certain subsystems to depend upon outputs from other

subsystems, facilitating a co-design process similar to that seen above for the `makeDesign` verification phase. This also allows each subsystem to define its own algorithms for creating final outputs, and for multiple possible output formats to be supported. The main outputs currently implemented within the system are described briefly below.

6.4.1 Electrical Subsystem

The `ElectricalComposable` processes the underlying electrical network. Since the autonomous decisions were made during the `makeDesign` phase, the output phase can simply focus on generating the desired outputs. Its output files include a bill of materials detailing what components will be needed, along with links and prices if known. In addition, it provides circuit diagrams and wiring instructions that the user can use to assemble the final design. Since the modules currently included in the library are plug-in devices, these instructions are usable by novice users. In the future, the electrical outputs could also include simulation-ready circuit diagrams that can be run, for example, in `LTSpice` [77]. Furthermore, while the current system focuses on through-hole components and plug-and-play devices, future extensions can easily be added to the composable to support generating PCB layouts and diagrams.

6.4.2 Code Subsystem

The `CodeComposable` processes the data topology and code components to create all software necessary to use the robot. It pools together all software snippets stored in the various components, and analyzes any code tags that reference design-specific information. It gathers this information from the topology acquired during the `make` and `makeDesign` phases, and correspondingly edits the code files to create coherent packages.

In particular, it will use the folders and files outputted during the `makeDesign` phase to create final software packages. A folder was previously created for each controller that will require code, and the declarations, methods, and insertions were sorted appropriately. In

addition, device-specific code tags were replaced with tags that included unique references to the desired devices. The `makeOutput` algorithm now processes these tags, resolving them to final values. It can utilize the search algorithms discussed in Section 6.1 to locate referenced devices if necessary, and can now calculate information that pertains to the whole design such as connected ports and numbers of devices of certain types. Note that this information could not have been known before the `makeDesign` method was run, since the design was not yet finalized. Method insertions can then be written into the files that store the desired methods, and the methods and declarations can be written into a final software package. This includes splitting declarations and definitions into header ($.h$) files and code ($.cpp$) files.

The result is a software package that can be compiled and programmed onto each microcontroller in the design. Separate folders will be made for each controller with all necessary files; the user therefore gets complete code packages that can be immediately loaded onto the robot. The current implementation produces C++ code files that can be used with the Arduino software environment for controllers such as Arduino's or ATtiny's, but extensions can be added in the future to support other languages. The generated code will contain code for any autonomous behavior specified by the linking of data ports, as well as useful libraries of functions that make it straightforward for intermediate users to write additional code for the robot. This process may also include generating code for interfacing with the hardware modules described in Section 3.3.1, if any are present, by abstracting away the implementation details from users of the final code library.

When the system analyzed the overall topology in the `makeDesign` phase, it assigned each port a unique ID. These can then be used as "virtual pin numbers," as shown in Figure 6.2, and this list of virtual pins is presented to the user along with the building instructions. When editing a generated Arduino file, for example, users can interface with the attached devices by simply using virtual pin numbers – if a sensor was assigned a virtual pin number of 3, a user can simply call `robot.analogRead(3)` whether or not the device is physically connected to the main controller. The software template and snippets included for the main controller and for the hardware modules will determine where the device is actually located. If on the main controller, it will convert the virtual ID to an actual pin number. If on

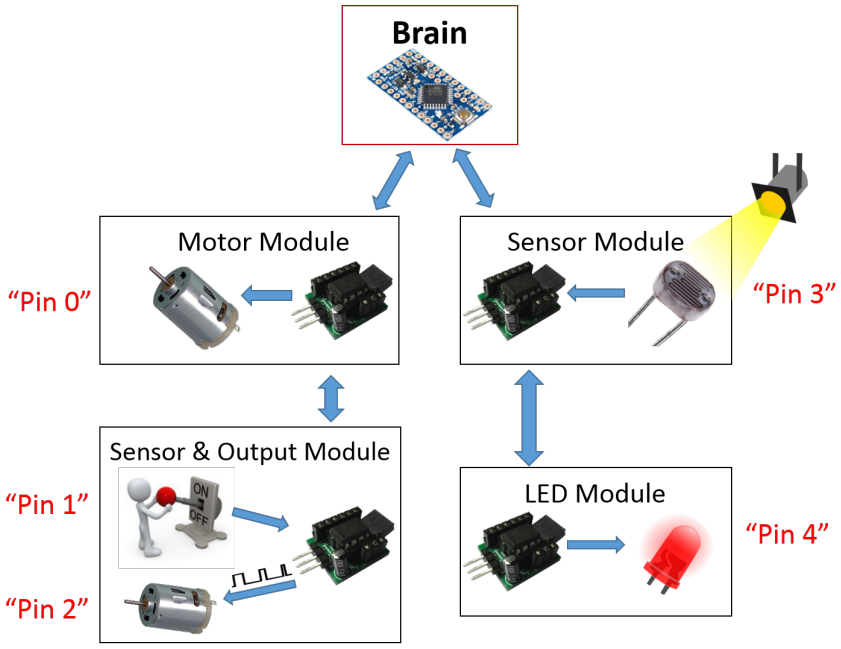


Figure 6.2: When the design is verified by the data and code subsystems, each port is automatically assigned a virtual pin number. The generated software library allows intermediate users to control the robot using these IDs, so that knowledge of the implemented configuration and communication topology is not required.

an auxiliary module, it will find the corresponding module and physical pin, then send the command along the appropriate module chain. The user can therefore program as they normally would program an Arduino, and all of the work for interfacing with the actual electrical layout is done behind the scenes.

In addition, this microcontroller code will be pre-configured to work with an Android app if any such UI elements were included, allowing the smartphone to generate a customized user interface for the new robot. The system currently uses a pre-written Android app to implement this capability. The microcontroller configures its Bluetooth transceiver, which would have been automatically added and connected by the composables during the `makeDesign` phase, to have an appropriate name such that the app can automatically find the device and establish a pairing. The app then communicates with the controller to determine what user interface controls should be presented, and the software template and snippets accompanying the UI elements allow the robot to respond appropriately. Some of the snippet code for

generating the transmitted descriptions was shown in Figure 4.4. The user interface layout is typically defined by user-added elements such as sliders or buttons, but may also be automatically added to the design by the composables. The user interface is then dynamically created by the Android app, and the user can intuitively control the robot. Thus, the user only needs to install a single app on their phone to have a unique interface for every robot they create.

6.4.3 Mechanical Subsystem

The Mechanical `Composable` processes the physical structures of the design to create fabricable output files. These currently include a 2D vector drawing that can be directly sent to a laser or vinyl cutter to create a cut-and-fold origami-style structure, as well as a solid object file that can be built using a 3D printer. In the future, these processes could also be chosen on a per-component basis, allowing part of the robot to be folded and part of the robot to be 3D printed. This may be especially useful, for example, if part of the robot should be flexible but other parts should be rigid. More extensions can also be added to support more diverse manufacturing methods. In particular, methods can be employed that integrate the electrical circuitry with the mechanical structure. This would allow an entire functional robot to be manufactured at once. It would also take advantage of the co-design framework established by the composables, leveraging the interaction between the mechanical and electrical subsystem algorithms.

6.5 Summary

To deal with a growing collection of hierarchically composed modular elements, a suite of algorithms has been developed to efficiently manage and process robot designs. These include search algorithms for locating desired components and instantiation algorithms for converting textual representations to executable objects. Verification algorithms also autonomously modify designs and validate functionality, providing the user with feedback and making independent design choices to reduce user input complexity. Finally, output generation algorithms create complete robot designs that can immediate create functional robots for the user's task. These algorithms separate computational requirements in order to reduce processing overhead, and allow subsystems to interact with each during verification and synthesis phases. Together with the component library, these algorithms create a responsive and intuitive design environment that allows users to quickly create fully customizable personal robots without extensive prior experience.

Algorithm 1 Composition Instantiation

```
1: procedure MAKE

2:     MODIFY PARAMETERS ▷ experts can override this abstract method to modify
           parameters according to application-specific rules
3:     RESOLVE SUBCOMPONENTS ▷ create a dictionary of instantiated subcomponents
           from the list of subcomponent names and their class types
4:     EVALUATE CONSTRAINTS ▷ evaluate the expressions and constraints defined
           among parameters, setting the subcomponents' parameters according to
           the user-defined relationships
5:
6:     ▷ At this point, the current component is instantiated and the various constraints
           have been evaluated. The following phases recursively make the component
           tree, and instantiate and populate the Composable elements.
7:
8:     ▷ Evaluate Components:
9:     for all subcomponents in the hierarchy do
10:        MAKE the subcomponent
11:        for all composables required by the subcomponent do
12:            if this component already has a composable of that type then
13:                Append the subcomponent's composable to this component's composable
14:            else
15:                Create a new composable object of that type
16:            end if
17:        end for
18:    end for
19:
20:    ▷ At this point, the main component has a single instance for each type of Composable
           required by at least one subcomponent in its hierarchy. The following
           informs them of the design topology for subsystem-specific processing.
21:
22:    ▷ Process the subcomponents according to domain-specific rules
23:    for all stored composable objects do
24:        Inform the instance of all known subcomponents
25:    end for
26:
27:    EVALUATE CONNECTIONS ▷ inform each stored Composable instance of the known
           connections between subcomponents
28:    ASSEMBLE ▷ experts can override this abstract method to perform computations
           before the final output is generated
29:    EVALUATE INTERFACES ▷ inform each Composable instance of the known
           interfaces, in particular ones that are unconnected and therefore were not
           seen previously when evaluating connections
30:
31: end procedure
```

Algorithm 2 Verification and Modification: Check and Satisfy Component Requirements

```
1:  procedure checkRequirements

2:      ▷ Check and try to satisfy all general subcomponent requirements
3:      for all subcomponents in component hierarchy do
4:          for all requirements in this subcomponent do
5:              if requirement is not satisfied by current design then
6:                  Try to automatically fix the design
7:                  ▷ For example, search library for and insert a missing component type
8:                  Alert user if no solution is found
9:              end if
10:         end for
11:     end for
12:     ▷ Recursively check requirements throughout the hierarchy
13:     for all subcomponents in component hierarchy do
14:         Call checkRequirements on this subcomponent
15:     end for
16:     if all requirements are satisfied then return True
17:     else return False
18:     end if
19:
20: end procedure
```

Algorithm 3 Verification and Modification: Iterative Domain-Specific Analysis

```
1:  procedure MAKEDESIGN
2:      CHECKREQUIREMENTS          ▷ Check and try to satisfy all general requirements
3:      MAKE                       ▷ Instantiate Components and Composables
4:      designComplete ← False
5:      Topologically sort Composable objects
6:      while designComplete is False do
7:          designComplete ← True
8:          ▷ Allow each Composable to perform domain-specific verification and modification
9:          for all composables in this component, in topological order do
10:             if COMPOSABLE.MAKEDESIGN is False then
11:                 ▷ The Composable modified the design, so another iteration is needed
12:                 designComplete ← False
13:                 Break the for loop
14:             end if
15:         end for
16:         ▷ Check general requirements, including newly added components
17:         designComplete ← designComplete and CHECKREQUIREMENTS
18:         if designComplete is False then
19:             ▷ Modifications were made, so prepare for another iteration
20:             Clear Composable states
21:             MAKE                   ▷ Instantiate any newly added components
22:         end if
23:     end while
24:
25: end procedure
```

Chapter 7

Case Studies: Making Robots!

It's kind of fun to do the impossible.

– Walt Disney

Contents

The library of integrated electromechanical modules and associated algorithms for hierarchical composition, verification, design augmentation, communication, and output generation create an environment in which users can quickly create functional prototypes from abstract specifications. Many different robots and electromechanical systems have been created with this system to demonstrate its capabilities and dynamically expand the library.

Videos, design files, details, and more case studies can be found at http://people.csail.mit.edu/delpreto

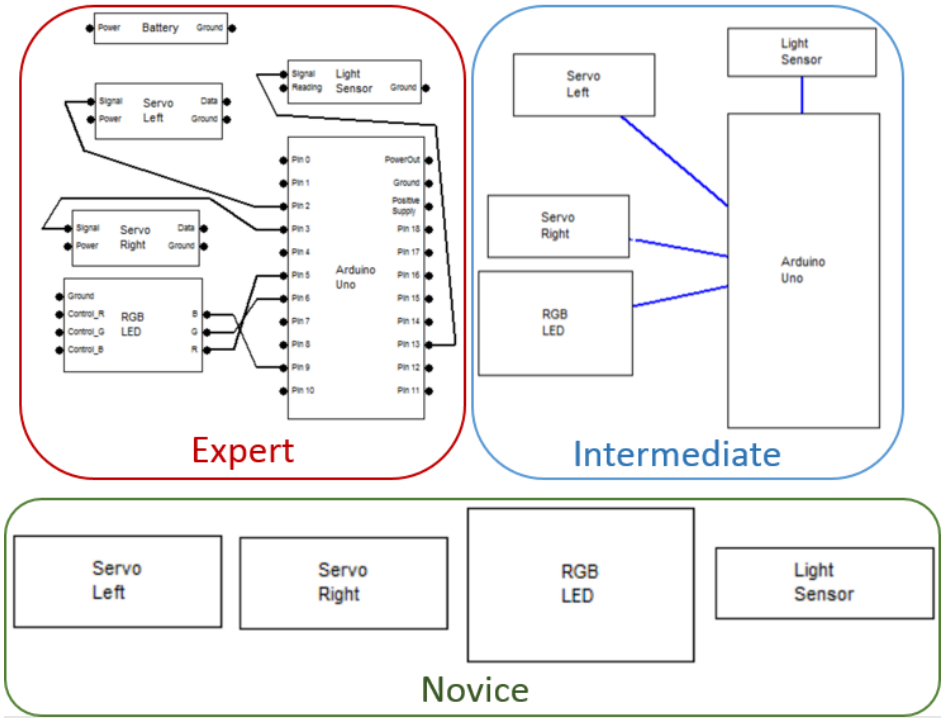


Figure 7.1: A graphical interface is provided for the compiler, which allows components to be selected from the library and dragged around the design space. Various levels of detail can be chosen to grant experts flexibility while still remaining comfortable for novices. All three views shown here will result in the same design once processed, but they each reveal different levels of complexity.

Users can currently interface with the system via a basic graphical interface that provides library filtering and dragging-and-dropping of components. In addition, all exposed parameters of a component can be adjusted to enable customization. As shown in Figure 7.1, it also allows users to choose their desired level of detail. Experts can view individual port connections and set all details, intermediate users can connect components and let the system choose the ports, and novice users can simply view the chosen components and let the system choose the connections. This graphical interface focuses mostly on the electrical and software subsystems, although a separate interface focusing on the mechanical aspects is currently under development [78] and can be integrated in the future. More control and flexibility can be achieved by using a separate textual interface to the system, and intermediate users can also write Python scripts to define a design component. An online graphical interface is planned for the future as well.

In order to realize a custom product to solve a particular task, a user breaks the task into simple subtasks and identifies modules from the library to accomplish each one. If a specific item does not exist, a similar one may be adapted or a new module may be composed from lower-level components. In the end, the design is compiled to generate the required fabrication files. Drawings can be sent to a cutter to fabricate the cut-and-fold mechanical structures, instructions on how to connect and mount the electrical and electromechanical components are displayed, and the generated software is programmed directly onto the brain modules. The user then assembles the physical elements to create the final desired system. The result can be controlled by the auto-generated UI or by auto-generated behavioral software if a desired behavior was specified.

This system is very versatile and capable of producing diverse functional designs. Various robots are presented here that highlight the co-design of electrical, mechanical, and software systems. Additional flexibility is also demonstrated by miscellaneous applications that leverage the electrical and software subsystem outputs to make product prototypes that automate tasks or provide user assistance.

7.1 Centralized Robots

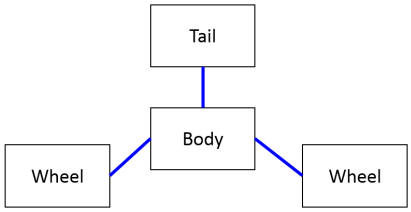
The design environment was used to create a variety of different robots with a centralized control structure. Because the system is process agnostic, any of a number of rapid prototyping manufacturing techniques can be used to realize the generated designs. The robots in this section were all cut from a 0.010 inch (0.25 mm) thick polyester sheet using a laser cutter, vinyl cutter, or scissors, then folded into their final 3D geometries. The electronic components were incorporated into the structure during the folding process according to generated layouts and instructions. Generated drawings guide the user along the steps in the folding process, allowing novice users to fabricate these robots. Generated microcontroller software was then programmed onto the robots, and the provided Android app could immediately display a unique control interface. More hands-off fabrication processes can be used to reduce the skill requirements on the user; for example, some similar designs were made using 3D printed structures and origami self-folding laminates in [79]. These techniques could be incorporated into the robot compiler system in the future.

7.1.1 Two-Wheeled Robot

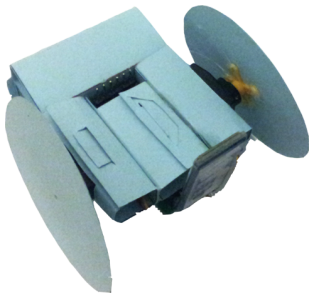
A two-wheeled mobile robot is shown in Figure 7.2. The robot, nicknamed the Seg, is specified by three parameters:

- the specific microcontroller used and its dimensions (in this case the Arduino Pro Mini)
- the specific continuous rotation servos used as drive motors and their dimensions (in this case Turnigy TGY-1370)
- the desired ground clearance (in this case 25 mm)

The user can design a basic Seg from the electromechanical component library by attaching two motors with mounts to a central body and a tail for stability as shown in Figure 7.2a. Each of these modular blocks has nested subcomponents that encapsulate the necessary information and hide the underlying complexity. For example, the `Wheel` contains an electrical servo, a mechanical mount, and the wheel structure as well as appropriate



(a) A novice user can define a two-wheeled robot by connecting two wheel modules to a central body. Each of these blocks contain nested components that define the structure and electronics, and these could be revealed in the expert mode of the GUI.



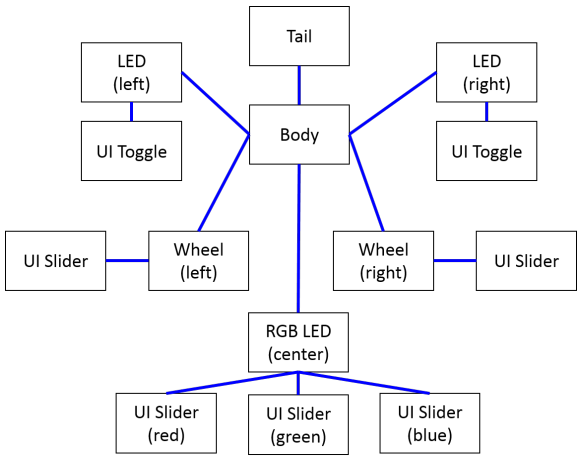
(b) The user design generates an electrical layout with instructions, software, and a 2D fold pattern. Here, the robot has been cut and folded from cardstock paper.

Figure 7.2: A functional two-wheeled robot can be created from a simple graphical design.

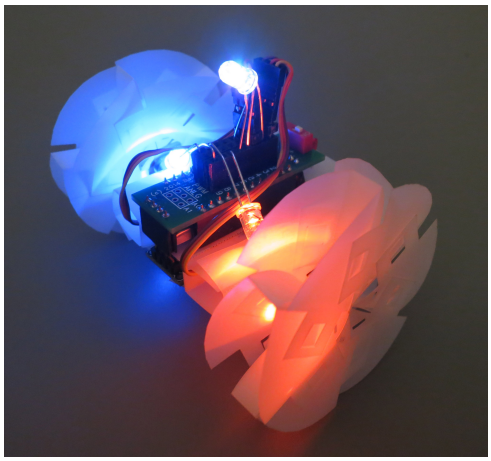
connections. Novice users can connect the high-level components and allow the system to choose ports and otherwise complete the design, while expert users can select specific ports and adjust all parameters.

Rapid Prototyping Iteration

Following the iterative rapid prototyping paradigm, this wheeled design can be quickly expanded to address additional user requirements and a new robot can be fabricated. Figure 7.3 illustrates one such expansion. In addition to the base Seg, the robot now has two digital LEDs and an RGB LED within which each channel can be set to an analog value. Furthermore, user interface elements have been added to control the robot from a smartphone. The functionality of the robot is defined by the flow of information, so the user simply connects the UI elements as information sources directly to the drive wheels as mechanical sinks or to the LEDs as electrical sinks. This will simultaneously define the robot structure, the robot functionality, and an Android interface. Parameters such as scaling factors, minimum and maximum speeds, component names, and UI labels can be set by selecting the modules and viewing available parameters.



(a) The Seg design is augmented with two digital LEDs, an RGB LED, and associated User Interface elements for controlling the robot via a smartphone.

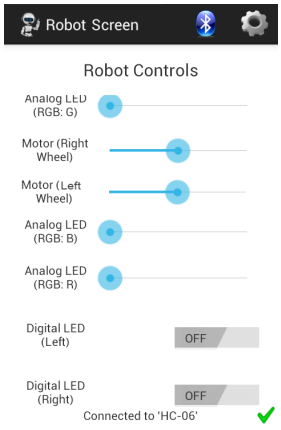


(b) The new robot is folded from plastic instead of paper, and has controllable digital and analog LEDs mounted on its main body.

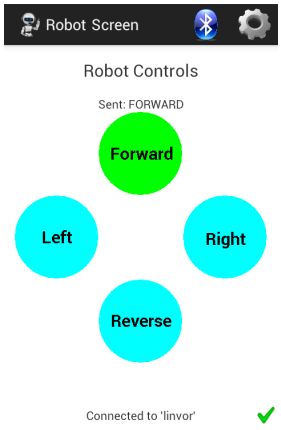
Figure 7.3: Following the iterative prototyping paradigm, a second wheeled robot is made that reuses the components of the first robot but adds additional LEDs and uses new materials.

As the design is verified, adjusted, and compiled, this flow will be broken down and expanded to make the necessary data conversions and ensure proper functionality. For the input shown in Figure 7.3a, this includes adding data buffers to manage the Bluetooth protocol, adding a Bluetooth transceiver serial module, choosing a microcontroller, and determining appropriate microcontroller pins for each connection. This process is facilitated by the multilayer component hierarchy; for instance, the servo device, motor mount, and driver components are all wrapped into a single `Wheel` component that encapsulates the needed subsystems and exposes relevant ports from its subcomponents.

Since the necessary mechanical, electrical, and software designs are encapsulated within the components, compilation of the complete design creates mechanical drawings for the body and wheels as well as code for the central microcontroller. The electrical subsystem gets resolved into a wiring diagram, software and firmware snippets from each component get pooled together and modified to create a software package, and mechanical mounts get physically linked. Assembly instructions are then displayed to the user, and a smartphone app can immediately generate the desired custom UI and drive the robot via Bluetooth.



(a) A user interface is automatically generated by the Android app that reflects the user-specified UI components.



(b) A custom interface was also written that leverages the generated port IDs and communication protocol to create a more interactive experience.

Figure 7.4: The second wheeled robot can be immediately controlled via an auto-generated Android interface, or a custom interface can be quickly created using the generated library.

Table 7.1: Design and Performance Metrics for the Two-Wheeled Robot

Metric	Result	
Approximate design time	1	hr
Approximate fabrication time	20	min
Approximate Cost	20.00	USD
Approximate Weight	42	g
Maximum speed	23	cm/s
Turning radius (both wheels activated)	0	cm
Turning radius (one wheel activated)	4	cm

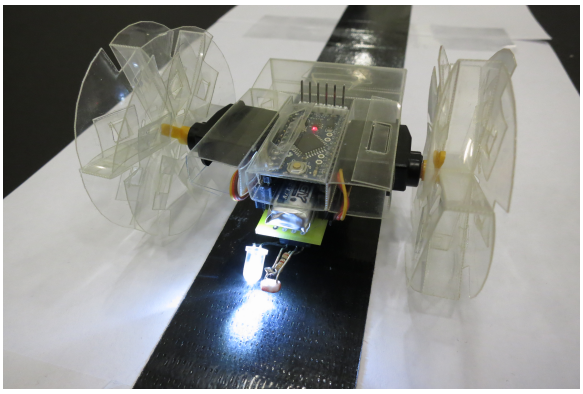
The new robot, which is now folded from plastic instead of paper, is shown in Figure 7.3b and the associated user interface is shown in Figure 7.4a. A custom Android interface is also shown in Figure 7.4b that was written with the aid of the generated library, in particular leveraging the communication protocol described in Section 4.2.1. A summary of the robot's characteristics are provided in Table 7.1. All of the electronics from the paper version in Figure 7.2b can be reused in the new plastic version, and a few LEDs are simply added. Since plug-in components are used, the assembly is straightforward for a novice user. Additional functionality including both hardware and software have now been added to a second version of the robot in minimal time, illustrating the benefits of rapid prototyping iteration.

Graphically Defined Autonomous Behavior

The design environment is also able to generate autonomous behavioral code for this robot. Instead of including visual LEDs as described above, a `LineDetector` can be added to the basic Seg of Figure 7.2. This sensor comprises an LED and a photoresistor, each of which is an integrated derived component containing a pure electrical subcomponent, a software driver, and a mechanical mount. The detector's data output can be connected to the data input of the integrated `Wheel`, potentially replacing the previous UI elements as information sources. Data manipulation blocks can also be used in this connection to define a mapping from sensor output to motor speed; in this case, one wheel should turn on when black is detected and the other wheel should turn on when white is detected. The generated code will now autonomously drive the robot in an edge-following pattern.

Furthermore, UI modules and graphical programming blocks can be inserted to allow the Android app to control the speed of the edge-following or temporarily pause the robot. The data outputs from the UI elements can be used to set speed and control variables, and these variables can be used as sources that control the wheel speeds. Thus, the graphical programming blocks interface directly with both UI blocks and physical device blocks.

The resulting robot is shown in Figure 7.5a. This iteration has been folded from yet another substrate, and the line detector is mounted on the bottom of the robot. When programmed with the auto-generated code, the robot blinks its LED to prompt the user to place it over black and then over white to calibrate the sensor, and then immediately begins following the line. This calibration behavior is included by the `LineDetector`'s software snippet, and the user instructions include details about this process. When the Android app is opened, it will connect to the robot and, upon communication with the robot, generate the interface shown in Figure 7.5b. Changing the slider or toggle switch will send a Bluetooth message to the robot, specifying the appropriate data port, and the local variables created by the graphical programming blocks will be set; the software snippets from these blocks inserted the necessary logic to accomplish this into the `processData` method. The user can therefore immediately control the speed of the robot or pause its behavior as desired.



(a) The line-following Seg is folded from clear plastic and has a light sensor mounted on the bottom next to a white LED.

(b) User interface elements to control the robot's speed as well as to pause its behavior are presented in the Android app.

Figure 7.5: Another iteration of the Seg robot features a line detector. Programming blocks graphically defined its behavior, and the fabricated robot successfully follows the edge of a line.

Educational Applications

This robot can also be used as a platform to explore custom software development and teach students about programming. Due to the inexpensive structural materials used, the reusability of electronics, and the rapid fabrication time, these robots can be easily manufactured in classrooms for many students. Depending on the students' experience levels, different modes of the graphical interface can be used to expose an appropriate amount of detail. Due to their ability to directly interact with physical components, the graphical programming blocks provide an intuitive way to generate arbitrary behavior. Students can therefore learn to think about logical flows of programs and immediately translate their ideas into physical instantiations.

Intermediate users can also leverage the auto-generated code library's methods to write custom C++ code in the Arduino environment. For example, they could use included methods to determine if the Android app is currently connected and switch between Android control and autonomous line-following control. Many of the implementation details are hidden by the generated helper functions; for example, support is included for virtual pin numbers as described in Section 6.4. This facilitates the exploration of sophisticated programs by

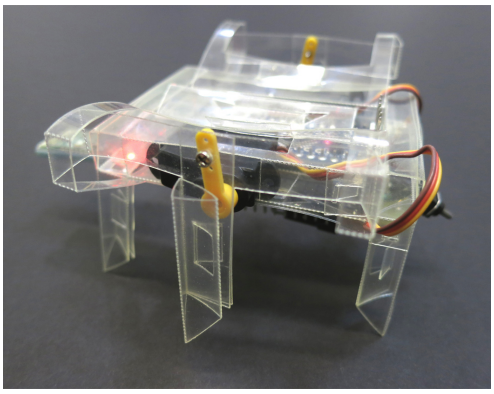
casual users. UI elements can also be added to the graphical design that display current sensor values in real time to allow more interactive results. Furthermore, an advanced user can use the defined communication protocol to write a more aesthetic Android interface for the robot, such as the virtual joystick shown in Figure 7.4b.

A sample curriculum was developed with this robot, incorporating the design and fabrication processes as well as the programming and behavior. It introduces finite state machines as a way to approach robot programming, and a graphical programming environment is used to implement these controllers. The curriculum and its associated materials, which were entered in the AFRON Ultra Affordable Educational Robot Challenge [80], can be found in [76, 81]. The Seg won first place in the hardware and curriculum categories, and second place in the software category. During demonstrations, elementary school and high school students have been very receptive to these ideas and enjoy watching robots implement their desired behaviors.

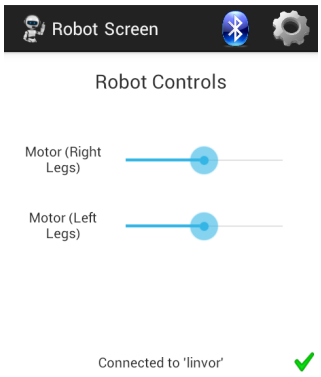
7.1.2 Hexapod Walker

An insect-like legged robot can be created using compliant joints to add kinematic degrees of freedom for a more complex design. Four non-moving legs form a stationary base, while two other legs are circularly actuated by drive motors to produce a walking gait. The moving legs remain parallel, and are constrained to move in a plane by flexural four-bar linkages. The design of this robot was adapted from the earlier Seg design, with many components directly reused since the electrical and software subsystems are largely the same. This was enabled by the modular design paradigm, which greatly simplified and sped up the creation of the hexapod.

An information flow similar to that of the wheeled robot defines the design, with an additional mechanical component defining a four bar linkage translating the circular mechanical output of the motor shaft into the walking gait of the moving legs. `Leg` blocks then replace the earlier `Wheel` blocks, and UI sliders can be attached to the legs to control their



(a) The ant was folded from clear plastic, and features two central moving legs in between four stationary support legs.



(b) The Android app generates an interface that includes the UI blocks for setting the speed of each leg.

Figure 7.6: A hexapod walker was designed using the system. The graphical design is very similar to the wheeled robot, with legs replacing the wheels, and the electronic components can be reused.

speed. The resulting structure and smartphone interface can be seen in Figure 7.6. A custom user interface similar to the one in Figure 7.4b can also be used for Bluetooth control, and autonomous behavior can be specified if desired.

7.1.3 Manipulator Arm

A markedly different configuration is created for the multi-segment manipulator arm shown in Figure 7.8. On this robot, an actuated gripper is positioned by a sequence of actuated hinge joints. As before, a novice user only needs to concern themselves with the top-level components for this design; a graphical design yielding the arm is shown in Figure 7.7, where a number of arm segments and a gripper are connected in series. Parameters such as names and dimensions can be set by selecting the various components. Figure 7.7 also illustrates the hierarchical composition; for example, each arm segment is an integrated electromechanical mechanism that contains structural elements such as hinges and beams, and electrical elements such as LEDs and servos. In the presented design, these even encapsulate UI elements such as a single toggle switch to control two LEDs and a slider to control the joint angle. These modules can be further broken down into their constituent

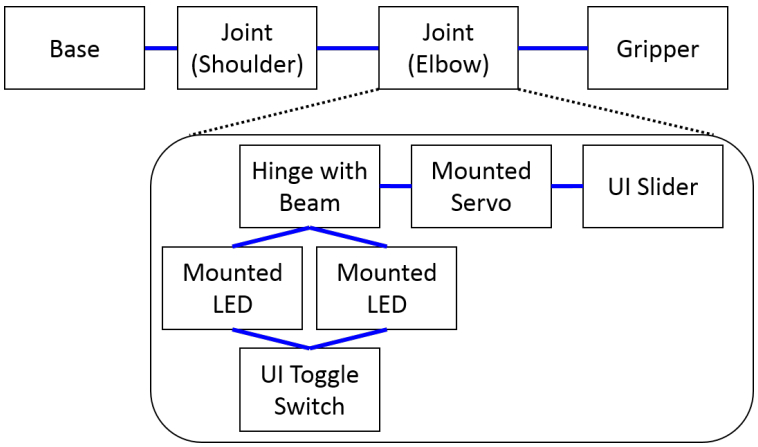


Figure 7.7: A robotic arm is defined by connecting two actuated hinges with associated arm segments in series with a gripper and a base. Each of these high-level blocks hierarchically contain lower-level building blocks including electrical devices, structural elements, and UI elements.

subsystem blocks, such as flexures and software drivers. A novice user can choose to use the highest-level view, while intermediate and expert users can choose to see successively more detail and specific port connections. A more exploded view of the information flow throughout the arm was previously presented and discussed in Figure 3.14.

When the design is verified, components such as microcontrollers, Bluetooth transceivers, and power supplies will be automatically added and electrical connections will be automatically chosen. This robot employs the electrical hardware modules described in Section 3.3.1, such that each actuated hinge and gripper module contains an independent integrated mechanical structure, actuator, drive circuit, and control logic. The plug-and-play electrical modules enable a distributed electrical system along the arm, allowing the electrical system to mirror the mechanical layout while simplifying user assembly and wiring. Elements such as LEDs and servos are connected to nearby modules, and modules are connected to each other with simple 3-wire servo cables. Since the code on these modules does not need to change, they can be readily reused across prototypes. Some performance metrics of the fabricated robot are presented in Table 7.2.

Since the high-level blocks in this design contain UI elements, the arm automatically generates a smartphone UI as shown in Figure 7.9a to allow immediate human control. The

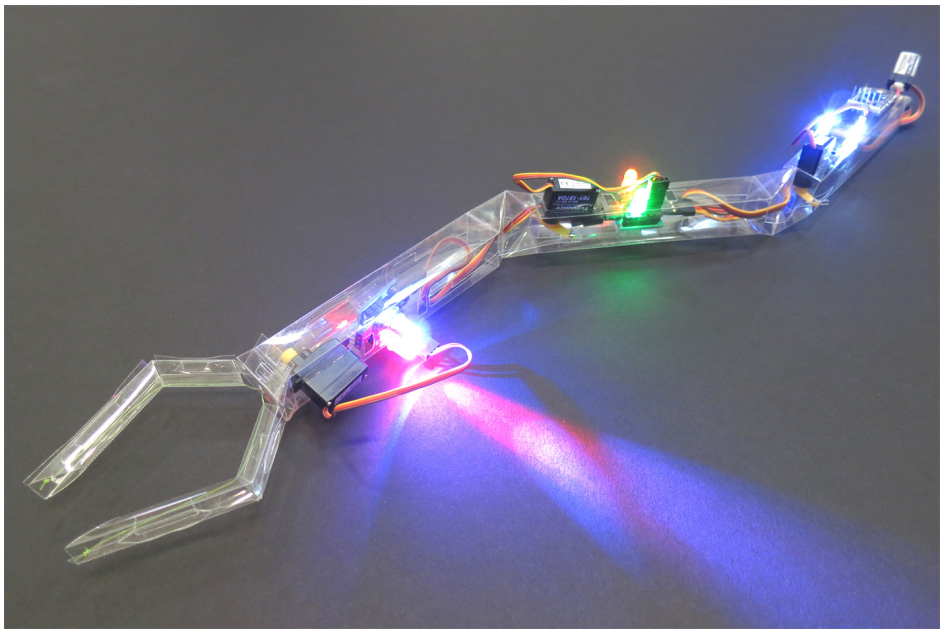
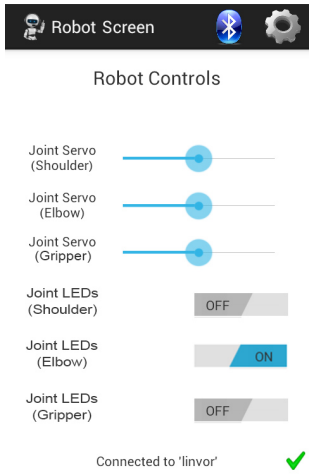
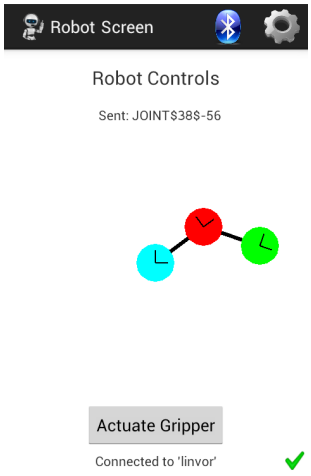


Figure 7.8: A manipulator arm with two actuated hinges and an actuated gripper is rapidly prototyped using the integrated design environment.

provided communication protocol, message format, and port ID mapping can also be used to easily implement a custom interface as shown in Figure 7.9b. In addition, a code library for the microcontroller was generated that includes a mapping of devices to virtual pins. In this case, the methods for accessing the virtual pins also transparently send commands to the various hardware modules, automatically determining the proper module chain and device index. In this way, intermediate users can write custom control logic for the arm and easily interface with the devices as if they were connected directly to the main microcontroller.



(a) A user interface, generated by the UI modules chosen in the design, allow each device to be controlled wirelessly.



(b) A custom user interface can be written using the communication protocol and port IDs defined in the generated microcontroller software. The red segment is currently being dragged by the user.

Figure 7.9: The manipulator arm can be immediately controlled using an auto-generated Android user interface, or a custom interactive interface can be quickly written.

Table 7.2: Design and Performance Metrics for the Robotic Arm

Metric	Result	
Approximate design time	1	hr
Approximate fabrication time	30	min
Approximate cost	27.00	USD
Weight	60	g
Maximum joint angle (actuation)	±35	deg
Maximum joint angle (mechanism)	±110	deg
Gripper Strength (on 1.5 cm object)	100	mN

Figure 7.10: The distributed robot garden provides an aesthetic platform for teaching robotics and engineering or demonstrating computer science algorithms. It features nearly 100 origami flowers that can open and close, many of which can also light up and change color.

7.2 Distributed Robot Garden

In addition to performing strictly functional tasks, electromechanical systems can capture the attention and imagination of all audiences by employing motion, light, and other sensory feedback. Towards this end, the robot compiler system has aided the rapid prototyping design, fabrication, and operation of a robot garden system. This system comprises a heterogeneous, distributed, multi-robot swarm, and is a scalable platform on which to demonstrate and evangelize robotics and computation. The creation of the system employed a number of rapid design and fabrication tools and techniques to create several distinct robots, which operate together in a unified aesthetic display. This installation can operate autonomously or be controlled from an integrated user interface. It can be used as a showcase for robot design and fabrication processes, a testbed for distributed algorithms, or a launchpad for technical education.

Some of the concepts presented in Section 7.2 are also discussed in [4]

Figure 7.11: There are multiple types of flowers in the garden, which are pneumatically actuated to open close. There are also hexapod walkers (lower left), and a swimming crane (lower right).

7.2.1 Garden Overview and Infrastructure

The garden, shown in Figure 7.10, consists of nearly 100 printable robotic flowers of eight different types that can move and change color as well as insect-like robots. The movement of each flower is controllable using a printable pneumatic actuator called a pouch motor [82] that operates by inflating and deflating a polyethylene pouch. Some flowers also include LED lights that allow the flowers to light up and change color. The flowers are organized into 16 separate garden tiles, and robot behavior can be controlled for individual flowers, for individual tiles, or for distributed subsets across the entire garden.

In addition to flowers, the garden incorporates other types of printable robots as seen in Figure 7.11. The hexapod insect discussed in Section 7.1.2 can wander throughout the garden, and a remotely controlled crane robot swims on a pond in the center of the garden. The swimming robot is an origami crane robot fabricated using a print-and-self-fold technique [83] and equipped with a permanent magnet for actuation. The design utilizes a crease pattern proposed in [84], and the self-folding process was carried out in an oven. The remote magnetic control is achieved by four electromagnetic coils situated under the pond;

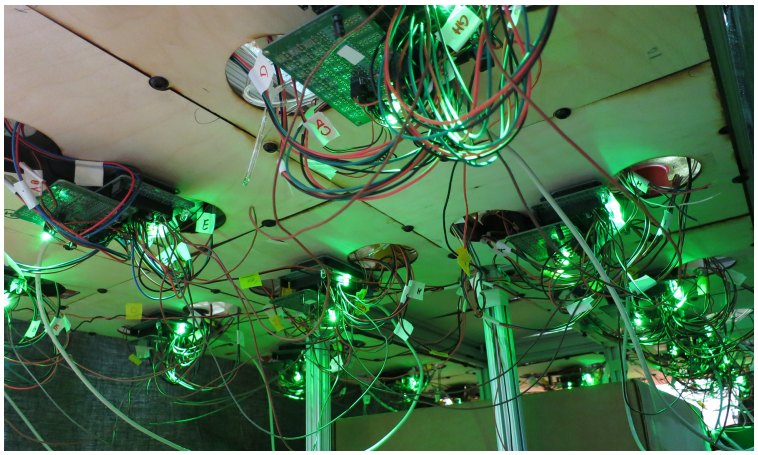


Figure 7.12: The modular electronics mirror the mechanical structure; each tile is controlled by an Arduino Mega 2560, which interfaces with a pneumatic pump and up to eight flowers via custom PCBs. Neighboring Arduinos can communicate via wires serial channels.

each coil is tilted 45 degrees relative to a central symmetric axis [85], enabling the crane robot to move in an arbitrary direction while floating on water.

Each of the 16 tiles in the garden can support control of up to eight flowers with LEDs and pouch motors, and uses an Arduino Mega 2560 microcontroller equipped with additional custom PCBs designed to service all pump and LED connections. As shown in Figure 7.12, the electronics are mounted beneath the garden in a modular way that mirrors the physical distribution of the flowers and tiles.

Each Arduino board can be connected to a Bluetooth chip to allow Bluetooth communication between the tile and a computer running a Graphical User Interface developed in Python. In practice, only one tile has this capability activated, and it serves as a connection point to the wireless controller that locally distributes commands to other tiles.

The Arduino board on each tile is connected via wired serial ports to its orthogonal neighbors as shown in Figure 7.13, creating a wired mesh network in which each tile is a node. This communication uses the custom serial protocol described in Section 5.3 in order to achieve reliable communication without requiring hardware serial ports, which are limited on the chosen microcontroller. Given the nature of the distributed algorithms, there

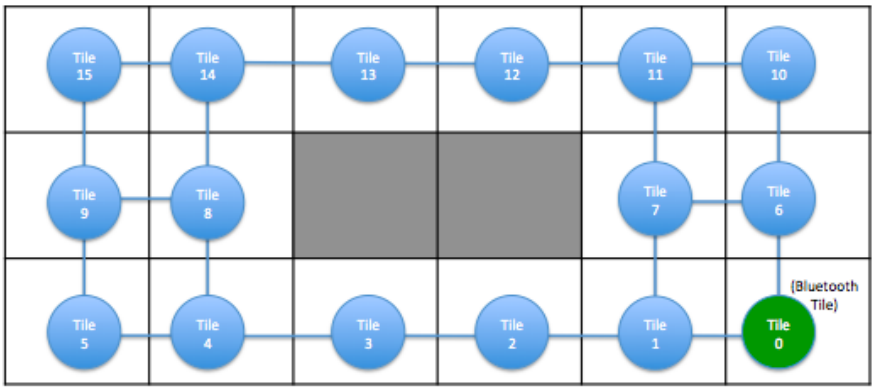


Figure 7.13: Each Arduino can communicate with its orthogonal neighbors via two-wire serial channels. One tile is also equipped with a Bluetooth module to communicate with a remote controller such as the provided GUI. This mesh network enables messages to flow throughout the garden and facilitates distributed algorithms.

may be many messages passing throughout the garden and a tile can frequently receive simultaneous messages from different neighbors. Using the standard software serial protocol, these scenarios would quickly result in data corruption and loss. The enhanced clock-driven protocol, however, provides reliable communication over software serial ports and manages the potential for collisions and simultaneous arrivals on multiple ports.

7.2.2 Visualizing Distributed Algorithms

When the garden is turned on, each node determines which local neighbors are present. They then collectively decide upon a unique address for each tile. These addresses are used in future algorithms to route messages through the garden, avoiding obstacles such as the center tiles that are removed to accommodate the pond. This enables the computer GUI to send messages to individual tiles via the single Bluetooth-connected tile, and allows complex distributed algorithms implemented on the garden to create emergent behavior.

The garden can be used to demonstrate distributed behavior, or to depict graph traversal algorithms and classic computer science concepts in a visually pleasing way by using the inflation and coloring of flowers. Currently, the system supports algorithms such as flood-

Figure 7.14: The garden can effectively demonstrate graph coloring algorithms. The tiles communicate with each other to determine a coloring such that no two adjacent nodes, including diagonals, have the same color.

Figure 7.15: The garden can also accomplish distance coloring, in which tiles communicate and choose a color for their flowers that indicates how many "hops" they are from the origin tile.

ing, graph coloring, breadth-first search, depth-first search, wave propagation, and distance coloring. Graph coloring and distance coloring are shown in Figure 7.14 and Figure 7.15, while breadth-first search and depth-first search are shown in Figure 7.16 and Figure 7.17.

A music mode is also offered in which the user can supply an arbitrary music source from a headphone jack and the garden acts as a real-time volume meter. Each tile determines a unique volume threshold based on its location in the garden, and one tile samples the music volume and shares it with the rest of the tiles. Each Arduino then controls its flower LEDs in response to the music volume and its unique threshold. The result is an exciting display that dynamically responds to any musical input by utilizing a distributed algorithm, where more of the flowers light up in response to louder music.

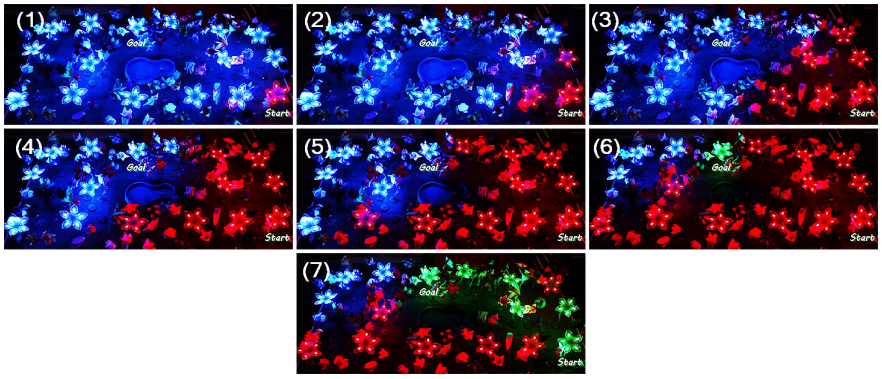


Figure 7.16: A distributed version of breadth-first search allows the garden to illustrate searching for a goal tile from a source tile. It gradually expands its search radius from the source until the goal is found, at which point the shortest path is also determined and highlighted in green.

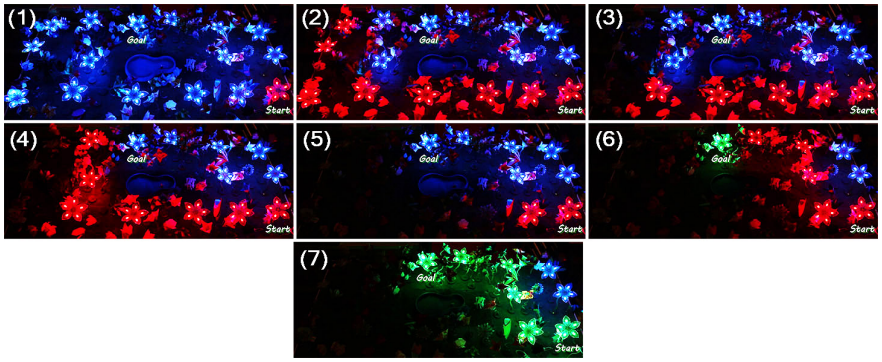


Figure 7.17: A distributed version of depth-first search illustrates searching for a goal tile from a source tile by traveling as far as possible in one direction and then backtracking and choosing different directions as necessary. The located path is then highlighted in green.

7.2.3 Educational Applications

The garden provides compelling aesthetic visualizations of important computer science concepts, and provides a powerful educational platform. The robot compiler can be used to generate the basic software library needed for the lower-level functions, as well as provide assistance with the electrical layout and wiring. In the future, the pouch motor fabrication process can also be added as a mechanical `Composable` extension, allowing for complete integrated designs.

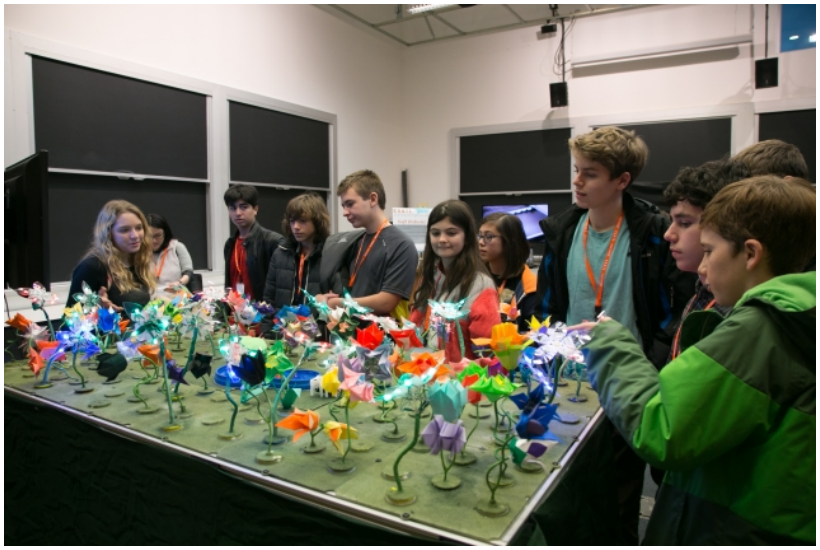


Figure 7.18: The garden provides an exciting platform that engages young students and stimulates interest in science and engineering.

This system has been demonstrated to young students, such as the session shown in Figure 7.18. At events including the Hour of Code [86] and Hubweek [87], as well as smaller-scale events, interactive experiences with the garden have successfully engaged students' attention and captured their imaginations. Through such outreach sessions, the garden demonstrates its potential to visualize important concepts and increase interest in pursuing scientific disciplines.

7.3 General Electromechanical Applications

In addition to generating fully functional robots, the presented system can be used for other prototyping tasks that require integrated electrical and software subsystems. A few applications are presented here that demonstrate the system's versatility and ability to enable the rapid fabrication of novel designs that help automate everyday tasks.

7.3.1 Wireless Camera Controller

Being able to remotely control cameras can be an indispensable tool when setting up creative photoshoots, performing time-lapse photography, or taking pictures that include the photographer. While many Digital Single-Lens Reflex (DSLR) cameras can be controlled via an infrared remote, this limits the controllable range and requires that the remote be pointed directly at the receiver. In order to develop a more useful remote controller, the robot compiler was used to generate a product that controls the camera via Bluetooth using an Android app.

A basic graphical layout for a first iteration of the design is shown in Figure 7.19a, where an infrared LED is simply connected to a UI button. This was the only input required by the system; the Bluetooth transceiver, a microcontroller, and power source were all added to the design behind the scenes to achieve functionality, and the electrical wiring was automatically determined as shown in Figure 7.19b. In this case, an ATtiny85 microcontroller was used, and was connected to both a Bluetooth serial module and an infrared LED. When opening the Android app, the controller sends the desired UI description and the smartphone screen is populated accordingly.

A small amount of custom code was then added to the auto-generated microcontroller code. Instead of simply turning on the infrared LED when a data message is sent to the LED's input data port, a method was added for pulsing the LED such that it mimics the pulse sequence performed by the commercial infrared remote.

Videos, documentation, and additional applications can be found at http://people.csail.mit.edu/delpreto

(a) The information flow of the camera controller is simply a UI button connected to an infrared LED.

(b) The electrical layout includes an ATtiny85 microcontroller, the LED, a Bluetooth serial module, and a battery (not shown).

Figure 7.19: A design for a wireless camera controller can be specified as just an LED and a UI button, and the implementation details such as wiring are automatically handled.

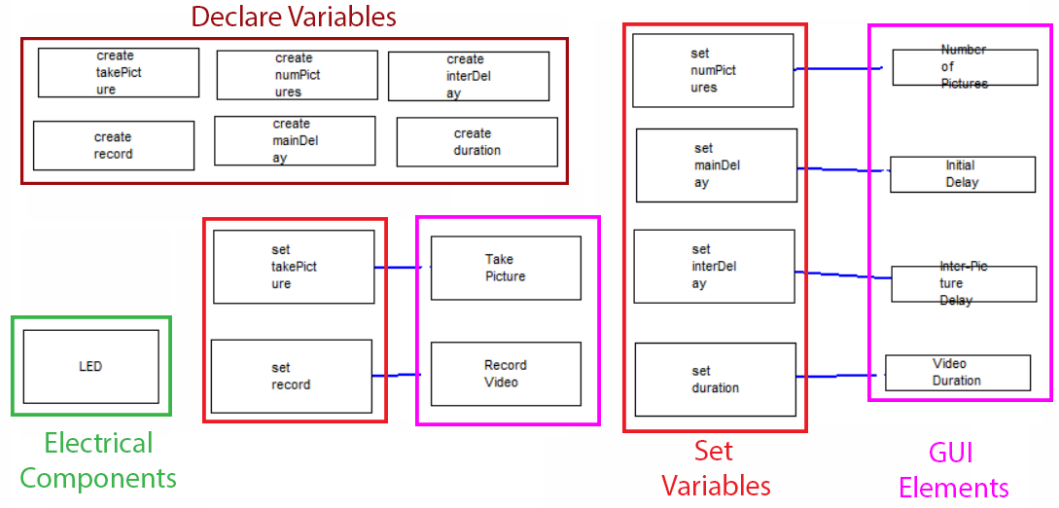
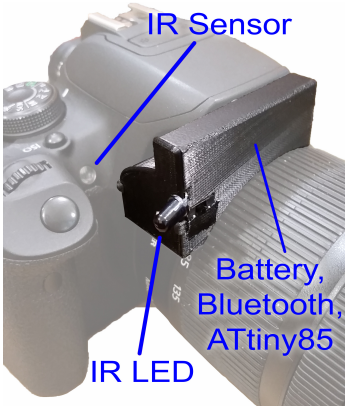


Figure 7.20: A more sophisticated Android interface can include elements for taking multiple pictures and setting delays, while the electrical layout remains the same. Software blocks are used to set variables based on these elements, and then control logic is manually inserted.

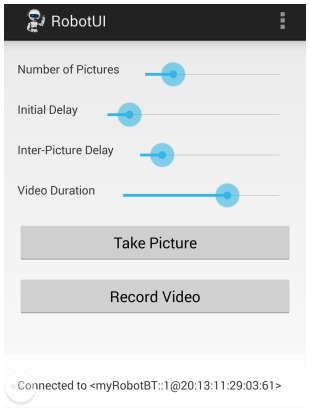
The result is shown in Figure 7.21a, where the electronics have been packaged into a custom 3D printed mechanical housing that fits snugly on top of the camera's lens. Pressing the auto-generated button on the Android app causes the camera to take a picture by pulsing the infrared LED.

A second iteration was then implemented that uses the same electronics but enhances the software. Instead of a simple UI button to take a picture, controls are added for setting an initial delay, setting the number of pictures to take, and setting the delay between pictures. A button is also added for video control in addition to a button for picture control; video control is supported by the commercial remote, and a slight modification was made to the custom

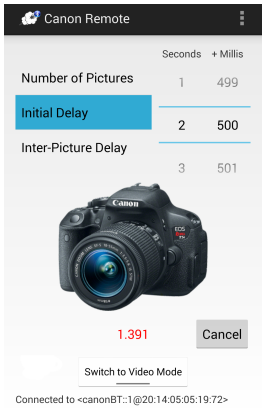
microcontroller method to implement the altered LED pulse sequence. The graphical design for this new system is shown in Figure 7.20, and the resulting Android interface is shown in Figure 7.21b. In this case, the robot compiler was used to create the Android layout and the communication and port ID infrastructure, as well as to set microcontroller variables based on the Android interface – the user then leveraged provided library methods for accessing data ports and the generated variables to write the simple control logic. Another case study that uses graphical programming blocks to simultaneously generate the interface and the control code will be discussed below in Section 7.3.3. A custom Android app was also written for the camera controller that implements the same functionality but in a more visually pleasing interface, as shown in Figure 7.21c. Since it uses the same message protocol and data ports, the microcontroller code can be the same when using either app. The electrical layout is the same as in the first iteration, as indicated by the single LED in the graphical design.



(a) The completed prototype of the Bluetooth camera controller fits snugly on top of the lens. The 3D-printed housing contains the electrical components, which were wired according to the generated instructions.



(b) An auto-generated Android app allows for taking multiple pictures, setting delays, and controlling video.



(c) A custom app was also written, using the generated port IDs, communication protocol, and code variables.

Figure 7.21: The final product allows the user to trigger pictures or video via Bluetooth using an Android app.

This application used the robot compiler to generate coupled electrical and software outputs, while the mechanical structure was built separately. It allowed for the rapid production of a prototype that addresses a specific user challenge, demonstrating some of the benefits of enabling custom on-demand devices for personal applications.

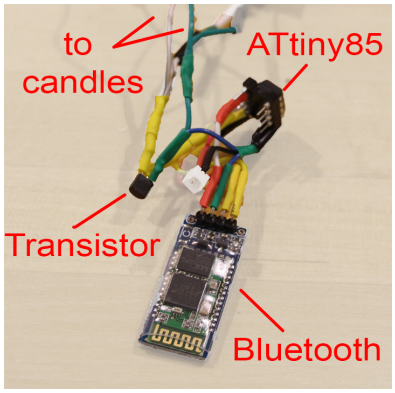
7.3.2 Wireless Relay Control

The application of the remote camera controller was also extended to a more general system that remotely turns an arbitrary device on or off. The graphical design is largely the same as shown above in Figure 7.19a, but now a UI toggle switch is connected to a relay or transistor. The Bluetooth devices and microcontrollers are added automatically, and wiring connections are automatically determined. Following the generated wiring instructions and using the generated microcontroller code results in a small product that allows for the wireless control of any electronic device; toggling the UI switch on an Android smartphone toggles whether the relay or transistor is activated, thus turning on or off any device that is connected to the switch. If desired, the robot compiler could also choose appropriate relays based on user-specified current or voltage requirements.

One potential application of this is shown in Figure 7.22, where electric birthday candles are now remotely controlled from the Android app. A custom graphical interface was also written for this project by leveraging the communication protocol implemented by the generated controller code. This project has been particularly entertaining when the app is used discretely, making it appear as if the unsuspecting birthday person has “blown out" the electric candles. In the future, sensors could conceivably be added to the graphical design that detect a gust of wind, thus automatically turning off the relay at the opportune moment.

This application is another demonstration of the ability for software-defined hardware to address physical challenges. Just as an app can address a computational challenge, the robot compiler can help create rapid prototypes that address tasks in the real world.

(a) Electric candles are connected to batteries through the controlled transistor. The electronics are hidden beneath a pile of brownies.



(b) Electronics are wired according to the generated instructions.

(c) A custom Android app is written that leverages the robot compiler's message protocol to communicate with the generated microcontroller code.

Figure 7.22: The wireless relay control prototype is used to remotely control electric birthday candles, so that an unsuspecting guest can be convinced to "blow out" the candles and be pleasantly surprised.

7.3.3 Automated Vacuum Cooker

Using the programming blocks of the graphical GUI, more sophisticated control logic can be implemented. One application in particular that demonstrates this capability is generating the electrical layout and control software for an automated sous-vide vacuum cooker. The goal of this device is to cook food by sealing it in an airtight bag and placing it in a temperature-controlled water bath. In order to achieve consistent cooking throughout the food while retaining moisture, the bath temperature is kept lower than for normal cooking and the food is left in the cooker longer than usual. For this project, the implemented design consisted of a water bath, a temperature sensor, a heater, and a pump. The heater should turn on if the water temperature is below a settable threshold, and the pump can be turned on to ensure an even temperature distribution throughout the bath.

The desired user interaction with this device can take place via two separate interfaces: a series of physical buttons and an LCD display mounted on the device itself, and an Android app that mimics these elements to allow for wireless control. This was implemented in the graphical compiler as shown in Figure 7.23, which uses blocks that represent physical elements as well as blocks that represent conceptual software elements and programming blocks. It creates variables for the current and desired temperatures, as well as whether the heater and pump should be activated. These are then modified or read by the various buttons, UI elements, and displays, and their values are used as data inputs to the electromechanical components. An outline of how the desired behavior is achieved with the graphical layout is described below:

- Global variables are declared for the current and desired temperatures as well as the current pump state.
- `If/Else` statements are inserted that condition upon whether the up/down physical buttons are pressed, and adjust the desired temperature variable accordingly.
- A UI slider is inserted for the desired temperature set-point, connected to a block that sets the desired temperature variable. The current position data input of the slider is also connected to a block that reads the desired temperature variable; this feedback loop allows the app to reflect any adjustments made by using the device's physical buttons.

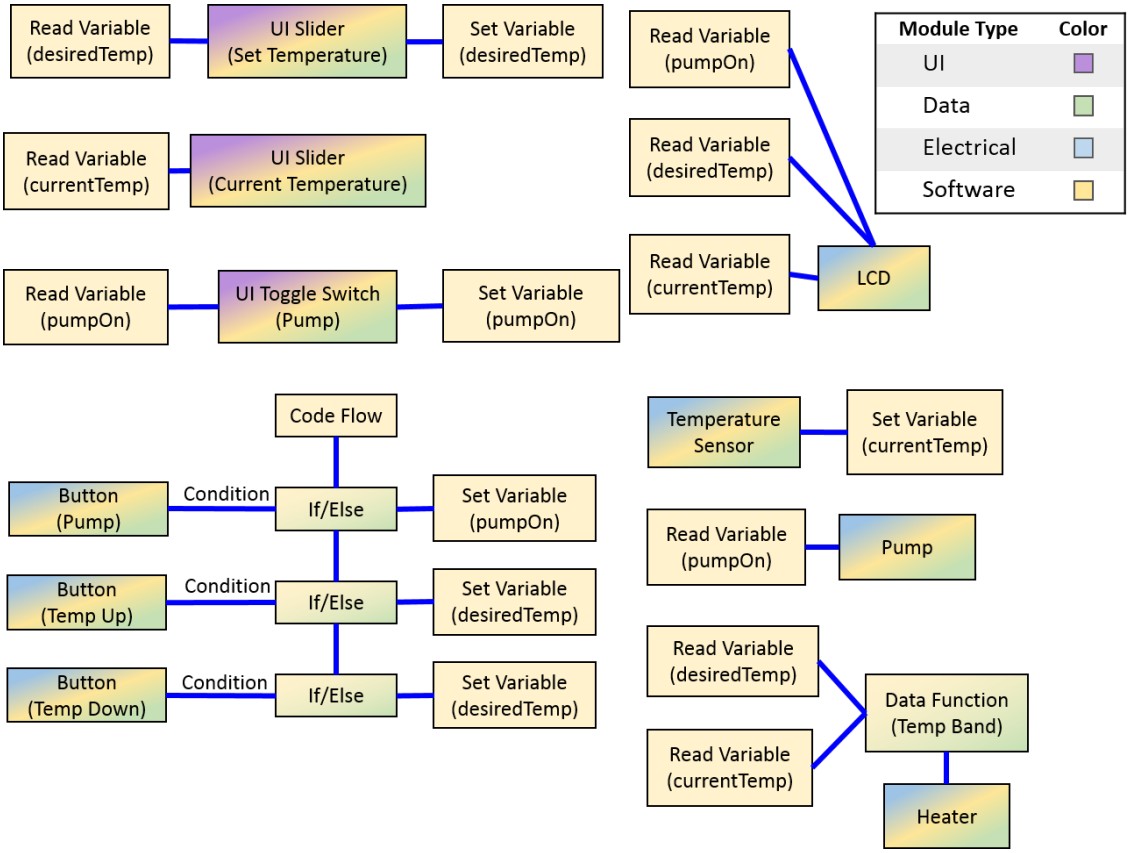


Figure 7.23: Electrical devices, UI elements, and programming blocks can be directly connected to concurrently implement both the electrical system and the behavior of an automated cooker. The design includes a heater, a pump, a temperature sensor, physical buttons, an Android interface, and control logic to maintain cooking temperature.

- A UI toggle switch is added to control the pump, and both the toggle switch and a physical button are set to edit the global pump state variable. The input data port of the UI switch is also connected to the pump state variable, allowing the switch to reflect physical button presses.
- The block representing the pump has its data input wired to the output of a block that reads the pump state variable.
- The data output of the temperature sensor is connected to a block that sets the current temperature variable.
- A UI slider is connected to a block that reads the current temperature variable to indicate the temperature in real time.
- A data manipulation block checks if the current temperature is below the desired temperature by a certain amount. The output of this data manipulation block is then connected to the input of the heater.

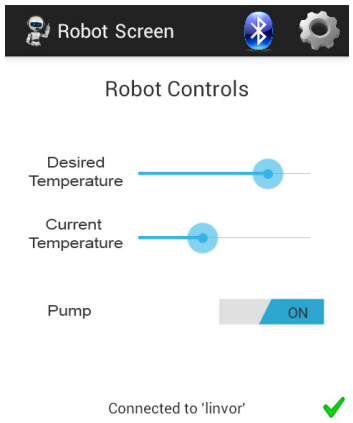


Figure 7.24: The cooker contains a water bath, control electronics, actuation, sensing, physical buttons for setting the temperature, and an LCD. A generated Android app shown here can set the desired temperature, control the pump, and display the device status in real-time.

This design illustrates the versatility achieved by allowing modules from different subsystems to be directly connected to each other. Programming blocks and data manipulation blocks can be directly connected to each other and to blocks that represent physical devices. This allows the user to intuitively translate their conceptual model into the robot compiler design space; for example, the temperature sensor and buttons are wired to variables, and variables are wired to heaters and pumps through appropriate functions. UI elements, physical buttons, textual outputs, graphical outputs, and physical actuators can all be treated interchangeably to implement desired behavior.

From this design based on information flow, an electrical layout is generated that includes an Arduino, a Bluetooth serial module, relays for the heater and pump, the physical buttons, and the LCD. Software is generated that contains low-level driver code such as interfacing with the heater and the I^2C temperature sensor. The software also contains high-level logic that will allow the user to set the desired temperature by using the physical buttons or the Android app, and that will turn on the heater whenever the water gets too cold. The app will automatically populate itself with a toggle switch for the pump, a slider that sets the desired temperature and that updates when the physical buttons are pressed on the cooker, and a slider that indicates the current temperature in real-time. The generated Android interface is shown in Figure 7.24, which was successfully used to control the fabricated device.

7.4 Summary

The robot compiler is a flexible system capable of generating complete robot designs from high-level user specifications for a diverse range of applications. Sample robots have been fabricated by composing modular blocks from the library in ways that represent desired information flows, and then following the instructions of the subsequent outputs. Each one is a customized personal robot that can be immediately used for specific tasks. Their mechanical structures are quick to produce and inexpensive, and their electronics can be readily reused in new robots. As such, the system greatly facilitates rapid prototyping and the iterative improvement of new ideas and concepts that can be used for wide ranges of personal tasks as well as education.

A distributed system has also been presented that aims to create a visually pleasing platform for the demonstration of computer science concepts. The robot compiler can aid in the design of such a system, and the result is an engaging educational platform that can help teach young students about engineering and programming.

Examples of prototypes are also presented for electromechanical products that leverage the coupled electrical and software outputs, demonstrating the usefulness of the robot compiler in everyday life. By providing an intuitive interface for complex systems that encapsulates implementation details, physical devices can be rapidly created to address physical challenges.

The presented case studies demonstrate the paradigm of software-defined hardware. Its associated flexibility and customizability allows real-world physical problems to be treated the same way as users currently treat computational problems. A library of basic blocks and methods for creating new ones are provided, but the outputs are only limited by imagination.

Chapter 8

Higher-Level Algorithms: Design from Functional Specifications

Logic will get you from A to B. Imagination will take you everywhere.

– Albert Einstein

Contents

The robot compiler described so far accepts structural descriptions and behavioral relationships, then produces integrated electromechanical designs including software. This can now be used as a foundation on which to develop higher-level systems. One such extension abstracts the user input even further, generating robots from functional task descriptions.

Some of the concepts presented in this chapter are also discussed in [5]

The robot compiler has been extended to accept more intuitive initial input, namely functional specifications in the form of relationships between atomic robot action primitives. Linear Temporal Logic (LTL) is used to formally represent a structured task description, and the modular robot component library is used to ground its actuation and sensing propositions to obtain structural specifications. Connectivity and parameter relationships are automatically derived from the generated finite state machine controller and from the geometric layout. Complete mechanical, electrical, and software designs are then automatically synthesized that implement the hardware as well as the autonomous behavior.

Some of the main enhancements added on top of the robot compiler platform to make this possible are outlined below:

- an adaptation of linear temporal logic for robot creation, converting a Structured English task specification into a control automaton that implements desired behavior
- a process for grounding the propositions of an LTL specification to components from the robot compiler library to generate a user-guided robot configuration
- a process for detecting potential behavioral conflicts during the grounding process, and automatically correcting the LTL specification when possible
- generation of a complete structural specification, including user-guided physical layout and automated controller synthesis, for the compilation of integrated robot designs
- an implementation of all of the above as an integrated end-to-end system generating robots from Structured English task descriptions

These enhancements allow users to design from a vision of behavior rather than structure. They abstract the required input to a more intuitive level for novice users, augmenting the iterative prototyping loop and enabling more complex robot behaviors.

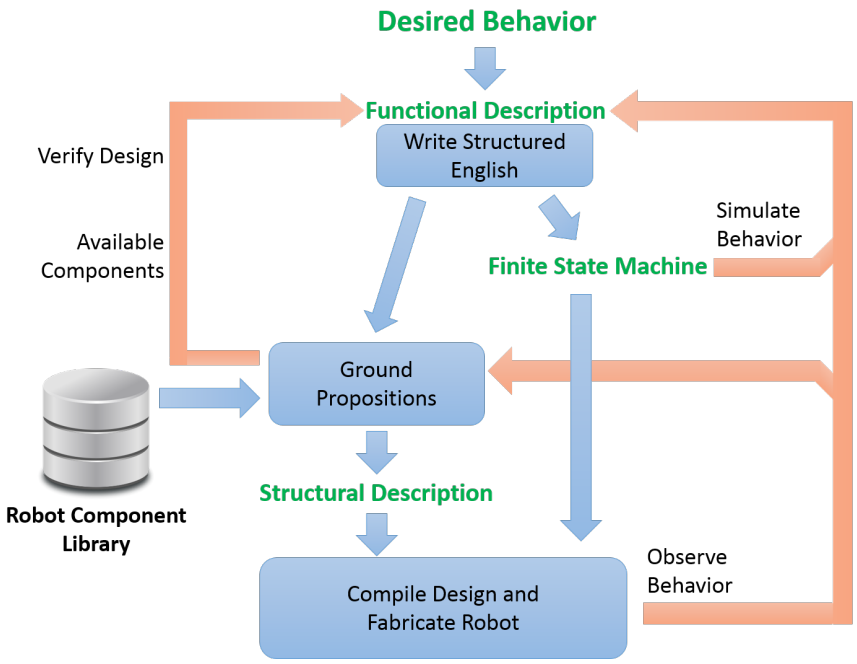


Figure 8.1: The system decomposes the design flow into a series of manageable stages. Blue boxes indicate user tasks. The process can also become iterative at each stage, using feedback from simulation or fabricated devices to encourage rapid prototyping.

8.1 Design Experience

To facilitate the rapid prototyping of custom robots from a description of desired behavior, the design flow has been implemented in a user-friendly environment comprising a suite of integrated tools that break the process into a series of well-defined, computer-aided stages. An outline of this flow is illustrated in Figure 8.1. It encourages an iterative process enabled by rapid prototyping that incorporates feedback from both virtual simulations and fabricated robots.

To define desired behavior, the user first writes a **task description** in Structured English [88] that captures the requirements and goals of the robot. Though not to the level of natural language programming, this allows a casual user to describe rather than command how the robot should operate through the use of primitive elements called propositions. These propositions may be, for example, a robot action primitive such as closing a gripper or a sensing capability such as detecting an object. The collection of necessary proposi-

tions therefore enumerates the necessary capabilities of the robot, and creates a **functional specification** of the robot.

The task description is then a series of logical linkings between these propositions that essentially describes a state machine in an intuitive manner. This input specification maps directly into linear temporal logic formulas, which are the inputs to a controller synthesis algorithm [89]. If there exists a finite state machine capable of achieving the goals given an adversarial environment, a controller will be automatically generated. This controller can be simulated by the user to ensure proper functionality.

The propositions are then used to create a **structural specification**, which defines an information flow among components as was previously described in Section 1.4 and Section 3.3 as the user input to the robot compiler. This is constructed by mapping the functional actuator and sensor propositions to parameterized robotic building blocks drawn from the robot component library. The system filters the library to recommend components appropriate to each proposition, aiding the user in grounding the specification. In addition, the system assesses the mapped propositions for possible behavioral conflicts, correcting the task description when possible. Depending on the components chosen by the user, a single functional specification may generate different physical robot configurations that accomplish the same goal. The selected components are then automatically configured and connected, though advanced users can edit and create custom configurations. The result is a parameterized robot design capable of accomplishing the task.

Upon setting desired parameters, the structural specification can be compiled to synthesize mechanical fabrication files, electrical wiring instructions, and software for the custom robot. The robot controller generated from the functional specification can be analyzed in simulation, and then autonomously converted to microcontroller code and directly programmed onto the robot. Once the user has built the robot using the given instructions, the desired behavior will be exhibited by the created robot.

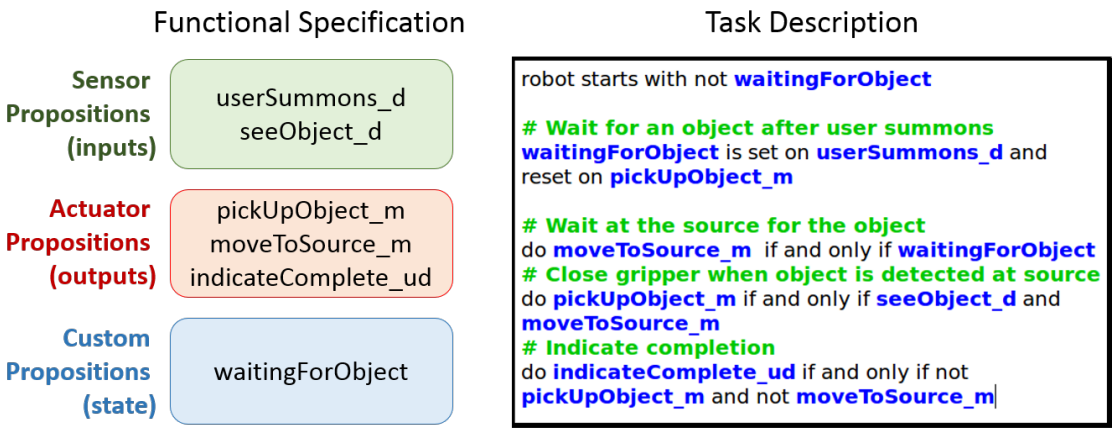


Figure 8.2: The desired behavior of a robotic grasper can be cast into Structured English, from which a finite state machine is automatically generated. A functional specification is also naturally extracted as a collection of propositions. These are shown in bold blue in the task description and are organized according to type on the left.

The below example will be used throughout the following sections to illustrate various stages of the design process:

> **Example Use Case:** The user wants to build a robot that can conduct a pick-and-place grasper task. When the robot receives a request from the user, it will move to a pick-up location and wait for an object to be presented. The robot will then pick up the object, return to the original position, release the object, and inform the user that it has completed the task.

8.2 Desired Behavior to Functional Specification

To create a custom robot from a description of desired behavior, the user starts by writing a mission specification for the target task. Using the Linear Temporal Logic MissiOn Planning toolkit (`LTLMoP`) [90], the user can write a specification in Structured English by first defining binary propositions. The translation of Example 8.1 into propositions and Structured English can be seen in Figure 8.2.

Propositions, which are abstracted from the robot's location, actions, and environment, can be divided into four different types:

- **Region Propositions**: If a map is provided, it can be decomposed into exclusive regions where each region is one proposition. During controller execution, only one of the region propositions can be true at any given time, representing the current location of the robot. In Example 8.1, there are no region propositions for simplicity.
- **Sensor Propositions**: These propositions relate to the robot's surrounding environment, and can be thought of as controller inputs. In Example 8.1, the presence of an object is abstracted into a sensor proposition called `seeObject_d` that is true when an object is observed and false otherwise. Note that even though these propositions describe different sensing capabilities of the robot, they are independent of how this capability is implemented; the propositions specify the functionality of the component rather than the actual structural component chosen during the grounding phase. For example, the detection of an object may be grounded to such components as a push button, a light sensor, or a vision input. In Example 8.1, `userSummons_d` is also a sensor proposition.
- **Actuator Propositions**: These propositions correspond to possible robot actions, and can be thought of as controller outputs. In Example 8.1, activating the actuator proposition `moveToSource_m` specifies that the robot should move to the pick-up location, and deactivating the proposition implies that the robot should be at the drop-off location. As with sensor propositions, these actions are functional rather than structural and therefore independent of implementation; for example, a robot with legs will move differently than a robot with wheels but both implementations could be chosen for a locomotive proposition. In Example 8.1, `pickUpObject_m` and `indicateComplete_ud` are also actuator propositions.
- **Custom Propositions**: These propositions, directly linked to neither robot sensors nor actuators, are necessary for specifying more complex behaviors and can be thought of as controller state. In Example 8.1, once `userSummons_d` becomes true, the custom proposition `waitingForObject` becomes and remains true until `pickUpObject_m` is true.

Using these propositions, the user can follow the grammar outlined in [88] to write a task description in Structured English. The propositions used, along with the task description, yield a functional specification of the robot that enumerates its required capabilities. A robot controller can then be generated with the synthesis algorithm in [89]. This correct-by-construction robot controller, in the form of a finite state machine, will be automatically generated using the toolkit if the mission statement from the user is feasible regardless of how the environment behaves.

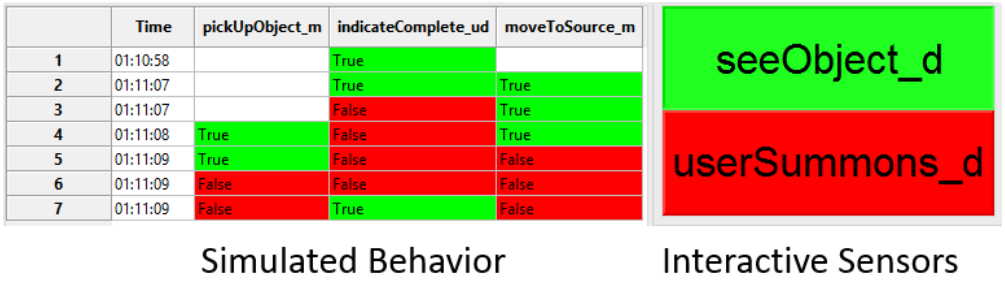


Figure 8.3: The generated finite state machine can be simulated to ensure desired behavior and encourage iterative design. Here, the behavior of a pick-and-place grasper is being simulated.

Once a controller is created, the finite state machine can be evaluated with an integrated simulation engine in which sensors can be interactively triggered and the proposition states can be visualized. A sample visualization of the engine for Example 8.1 is shown in Figure Figure 8.3. This facilitates an iterative process in which the user can immediately see how the robot would behave and accordingly adjust the specification or functional requirements.

8.3 Functional Description to Structural Specification

Once a functional description of the robot has been obtained, a physical instantiation of the robot that achieves the target task must be determined. In particular, the action and sensing tasks to be performed by the robot can now be grounded to available robot components to generate a structural specification.

8.3.1 Grounding

To ground the functional propositions to structural components, the grounding editor in the toolkit was modified from [90] to accommodate robot creation. Shown in Figure 8.4, it first retrieves the library of possible modular robotic components from the robot compiler. As described in Chapter 3, each component in the library is parameterized for customizability and encapsulates the relevant design and fabrication information such as the mechanical

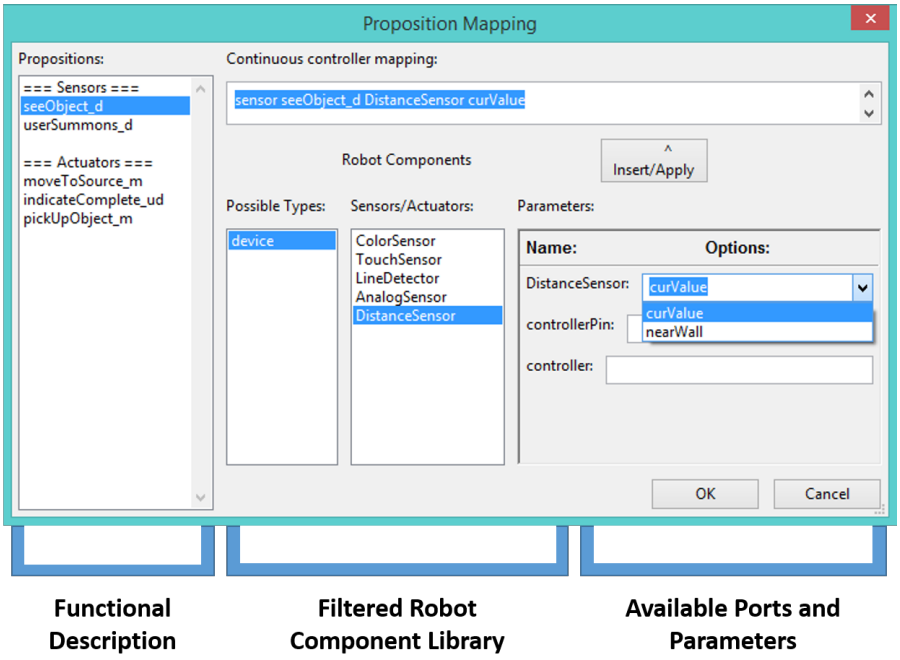


Figure 8.4: A functional description can be converted to a structural description by selecting modular components from the robot library for each action and sensor proposition. Filtered lists of possibilities are automatically provided, and the user can choose to customize them by adjusting parameters.

structure, electrical properties, and software snippets. The grounder currently divides components into three types:

- mechanical components, which require constructed structural elements to interface an electromechanical transducer with the environment
- device components, which are discrete elements with completely self-contained actions
- UI components, which are purely virtual components that include smartphone interface elements such as sliders or toggle switches

A user can specify desired possible component types for a proposition by suffixing its name; based on that suffix, the grounding editor leverages the search algorithms described in Section 6.1 to display a filtered list of allowable components. In Example 8.1, a list of mechanical actuator components is shown for the proposition `pickUpObject_m`, while a list of device and UI actuator components is shown for the actuator proposition `indicateComplete_ud`.

The user can then ground each proposition to a port on one of the available components to realize the desired actions and sensing capabilities. When a component is chosen from the library, a list of the component parameters is presented so the user can customize the component if needed, and a list of its ports is presented so that one can be chosen for the grounding. Through this process, each actuator and sensor proposition is mapped to a port on a component.

Some possible groundings of the propositions in Example 8.1 are shown in Figure 8.5. The conversion from functional description to structural specification is aided by the toolkit but ultimately chosen by the user; the user asserts control over the design according to personal preference and task-specific requirements, such as environmental considerations and component availability. Since there are often many components that can be grounded to the same proposition, many different robots can satisfy from the same functional description. For example, a human-triggered sensor input may be mapped to a button, a microphone, or a UI element, while an indicator action may be mapped to a light, a buzzer, or a flag waver. This approach simplifies and guides the robot design process for novice users without restricting expert users.

8.3.2 The Finite State Machine Component

To facilitate the grounding process, a custom component called `FiniteStateMachine` has been created that inherits from `CodeComponent` and `DataComponent`. This block is automatically inserted by the grounding editor into the information flow to represent the finite state machine controller that was generated from the Structured English description. The robot compiler autonomously modifies this component's parameters as appropriate, and creates `DataInputPort`s for each sensor proposition and `DataOutputPort`s for each actuator proposition. As groundings are chosen by the user, connections are forged between these ports and the chosen ports of the appropriate components.

This block can be viewed as an elaborate data manipulation block; it takes as input the current state via the sensor proposition ports, and generates an output for each actuator proposition port. The logic for switching between states based on the current sensor readings is encapsulated within the finite state machine via code snippets. The module contains pointers to templated microcontroller code snippets that provide the framework for running a finite state machine on a microcontroller. In particular, it uses the code tags described in Section 4.3 to inject the following into the auto-generated microcontroller software packages:

- arrays that store the defining sensor readings and actuator outputs of each controller state
- a 2D array that stores the possible successor states for each given current state
- code added to the `robotSetup` method that sets the initial robot state
- code added to the main `processData` method that checks if data is being sent to a sensor proposition input port and, if so, records the new state input
- code added to the `robotLoop` method that computes the next state based on the current state and the current recorded sensor inputs once per microcontroller execution loop, and updates the actuator commands by calling `processData` on the data input ports connected to each grounded actuator proposition output port

This therefore uses code snippets to extend the software template model described in Section 4.1 and convert the generated finite state machine into microcontroller code. Note that since this is simply another `CodeComponent` with another code snippet to include, it works seamlessly with existing data and code components; this allows it to be easily integrated within complex designs that may involve asynchronous wireless data transfers, user interfaces, data manipulation blocks, or any other component from the robot library.

Such an enhancement demonstrates the flexibility and expandability of the robot compiler. By creating a single additional subclass, an expert is able to easily add support for an entirely new integration that greatly expands the compiler's capabilities and further abstracts the necessary user input.

8.3.3 Design Feedback and Assumptions

The current process allows a user to map more than one proposition to a single component. This may be desirable if multiple propositions are intended to dictate different behaviors for the same component. As an example, the user may ground propositions `secureObject` and `releaseObject` to the same physical gripper, with the intent of having the gripper open when `releaseObject` is true and having the gripper close when `secureObject` is true. However, a problem may arise if the Structured English specification allows both `secureObject` and `releaseObject` to be true simultaneously, resulting in contradictory commands to the same physical component. This would prevent the robot from achieving the desired behavior, and may damage the robot. Note that this especially applies to actuator propositions, since this corresponds to connecting two output ports to the same input port and thereby shorts the output ports together. Connecting two sensor propositions to the same component port is not necessarily problematic, although it might not be an informative design choice since both propositions will always have identical state.

The grounding interface addresses this issue by evaluating the grounded propositions and assessing any propositions grounded to the same component. The system then appends mutual exclusion clauses to the Structured English specification so that the propositions can never be true simultaneously, and alerts the user to the change so they can ensure that desired behavior is preserved. This process is demonstrated by the Fetch Robot case study described in Section 8.5.2. When parsing the specification into complete designs, the robot compiler will automatically insert `Multiplexer` components into the data flow around the `FiniteStateMachine` in order to ensure that commands are properly routed.

Propositions of the LTL specification currently assume boolean signals, but some robotic components employ analog signals. Analog sensors therefore get thresholded before becoming inputs to the finite state machine, and binary actuator commands from the controller get scaled to user-specified analog values. This thresholding and scaling can be accomplished with data manipulation blocks, which will be automatically inserted into the design by the robot compiler during the design verification and modification phase.

Through such interaction, the grounding editor creates a closed-loop design process as amendments are made to the functional specification based on the structural specification. Not only does the functional specification affect the structural specification, but the structural specification set by the user affects the functional specification.

8.3.4 Mechanical Connections

A structural specification also requires a geometric layout of the physical components to create a single integrated electromechanical device. Though the design space of geometric configurations can get intractably large, the system once again aids a novice user by presenting a reduced set of options to handle general cases. An expert user can bypass the filter and create arbitrary mechanical connections constrained only by available interface points designed to limit component collision.

The mechanical components preferentially presented for grounding are designed to mostly fit into a rectangular prism bounding box. This allows for physical composition by tiling the selected components into orthogonal regions. The user can select whether a particular component belongs in the front, back, left, right, or center of the robot; the system then iterates through the full list of mechanical components and appends them onto the core controller module, growing the robot as it goes. Components with parameterized dimensions get scaled to fit the entire collection.

In a similar manner, the remaining non-mechanical device components such as electrical sensors get mounted on any exposed face of the robot. The user can specify whether the device should be facing forwards, backwards, left, right, up, or down, and the system will mount the device onto the respective structures assembled in the previous step.

8.4 Structural Specification to Integrated Robot Designs

Once the complete structural specifications have been obtained, the robot compiler processes the modular design into design files for the complete robot. These algorithms, including composition and instantiation, design verification and modification, and output generation, have been described in Chapter 6. The result is a set of mechanical fabrication files, electrical wiring instructions, microcontroller code, and a user interface.

Since the finite state machine was implemented as a new `Component` subclass, no modifications need to be made to the robot compiler's verification or output algorithms. The `FiniteStateMachine` component behaves as any other component with data ports and code snippets, allowing the compiler to adjust the design and provide feedback as usual. The insertions of data manipulation blocks for converting between analog and digital values will happen automatically if the `DataComposable` detects a mismatch between connected data types and searches the library for an appropriate `Buffer`. The `ElectricalComposable` will deal with choosing a microcontroller and determining appropriate pin connections and pin types. In addition, the conversion of the finite state machine into microcontroller code was implemented as a straightforward code snippet, so the `CodeComposable` will process this code as usual and the final software will implement the desired robot behavior.

As before, the mechanical structure is fabricated using an origami-inspired cut-and-fold process. The generated fabrication file gets sent to a desktop vinyl cutter or laser cutter to be cut from a 2D sheet of plastic, and the user follows the folding instructions and the generated wiring instructions to fabricate the robot.

Finally, the automatically generated software can be loaded onto the main controller, ranging from low-level drivers to the implementation of the finite state machine created from the LTL specification. The robot can then be simply powered on to achieve the task goals initially provided by the user, assuming an appropriate grounding scheme was chosen.

8.5 Case Studies

Two example tasks were explored to demonstrate the capabilities of the enhanced system. For each case, the robot compiler generated mechanical fabrication files in the form of 2D drawings that could be cut and folded, electrical wiring instructions, microcontroller code implementing the generated finite state machine, and an Android interface if appropriate. The structures were then folded, the electronics added, and the code programmed onto the microcontrollers. Once complete, the robots immediately performed the desired tasks.

8.5.1 Pick-and-Place Grasper

A sample robot made using this system is a robotic grasper for the task described in Example 8.1. A robot is desired which, when prompted by the user, moves to a starting location and waits for an object; when an object is detected, the robot grasps it, moves to a target location, and notifies the user. This behavior can be written in a Structured English description as shown in Figure 8.2, and the generated finite state machine can be simulated as shown in Figure 8.3. There are a variety of ways in which this can be grounded to generate a structural description, and a few such possibilities along with the one chosen here are depicted in Figure 8.5. Custom propositions represent internal state that do not become grounded in robot components. During the process of choosing robot components from the library, various parameters such as arm length or gripper size can be set by the user according to the task's environment and restrictions.

Once the grounding is complete, the robot compiler generates a fold pattern along with electrical instructions and Arduino code. The resulting robot is shown in Figure 8.6. After uploading the generated code, the arm demonstrates the desired behavior. When the user claps, the robot moves to the source location and waits for an object. It grasps the object upon detection, moves to the target location, releases the object, and indicates completion using a buzzer. Table 8.1 presents metrics regarding the robot's design and performance.

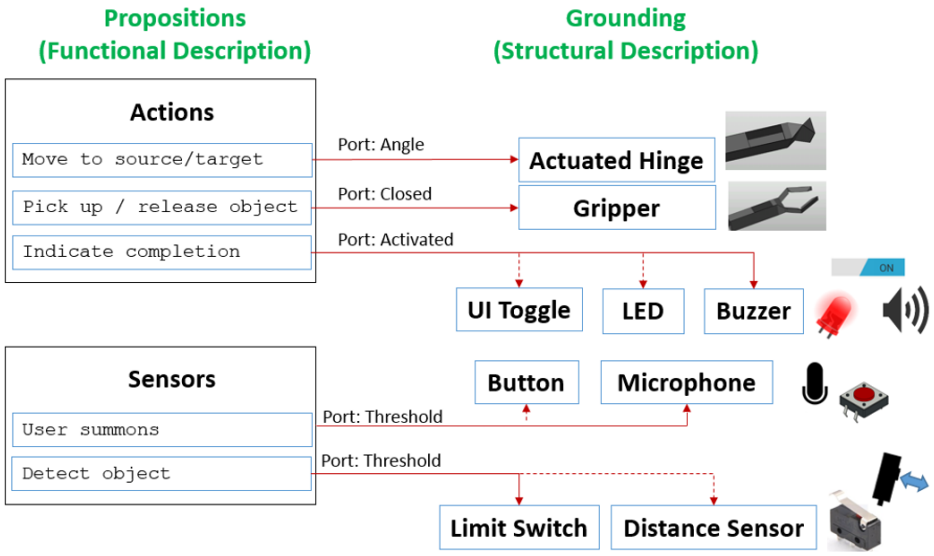


Figure 8.5: There are numerous components in the library that can implement each functional proposition of the robotic grasper. A few such mappings are shown here, where solid lines indicate those chosen.

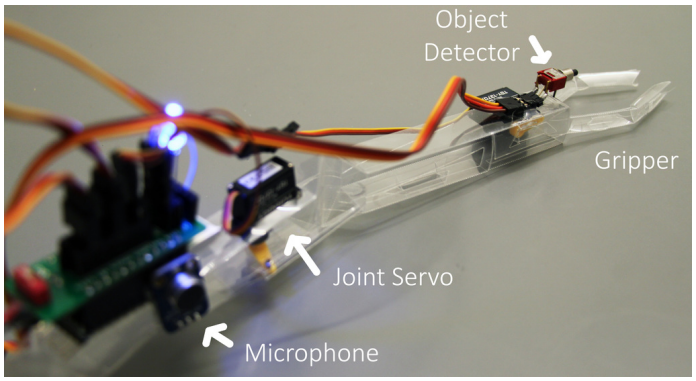


Figure 8.6: A pick-and-place robotic grasper was designed using the presented system by starting with a desired behavior.

Table 8.1: Design and Performance Metrics for the Robotic Grasper

Metric	Result	
Approximate design time	30	min
Approximate fabrication time	30	min
Approximate cost	25	USD
Mass	49.4	g
Maximum actuated joint angle	±35	deg
Gripper strength (on 1.5 cm object)	100	mN
Maximum gripper opening	110	mm

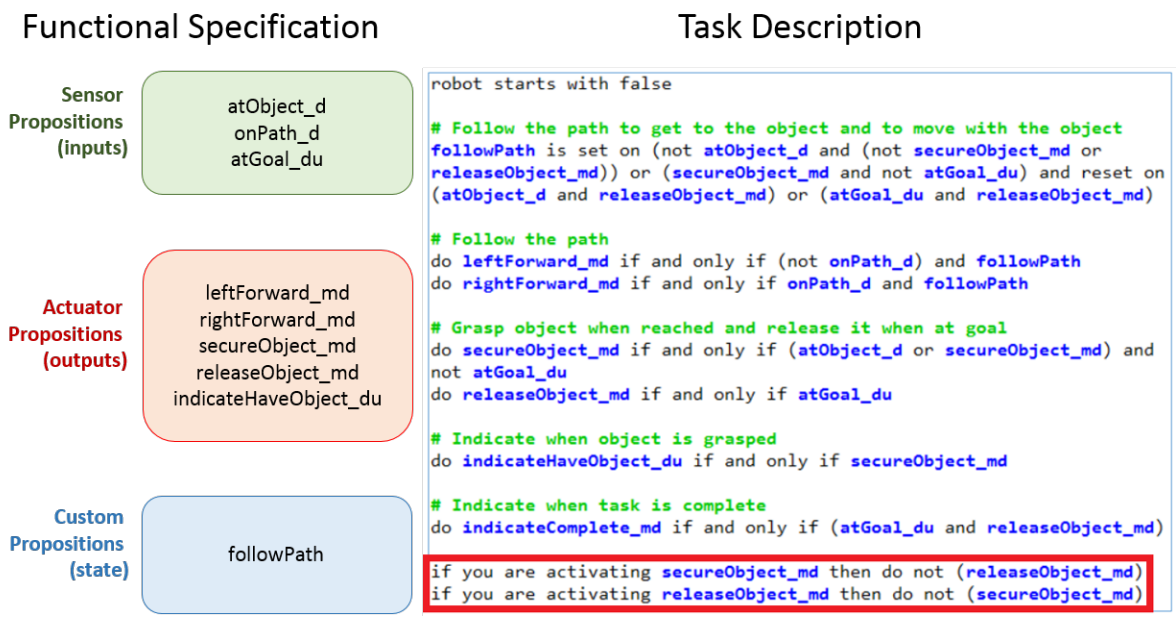


Figure 8.7: The desired behavior of a path-following object fetcher can be defined using Structured English with various propositions. The highlighted statements are necessary to enforce a mutual exclusion condition on propositions grounded to the same physical component, and are automatically generated and added to the functional specification.

8.5.2 Fetch Robot

A second example is a mobile robot with an attached manipulator for retrieving an object placed along a path. The desired behavior is to follow a path until the object is reached, secure the object, continue following the path until the goal is reached, release the object, and indicate completion. This behavior can be written using the Structured English of `LTLMoP` as shown in Figure 8.7.

To demonstrate versatility and the potential for rapid prototyping, two different sets of groundings were implemented for the same functional description. The chosen components are enumerated in Table 8.2, and the completed robots can be seen in Figure 8.8. Although they use different approaches and devices, such as line-following versus wall-following, flags versus buzzers, touch sensors versus light sensors, or Android UI switches versus microphones, they both achieve the same abstract path-following and object retrieval behavior. Some metrics regarding the design and performance of these robots are summarized in Table 8.3.

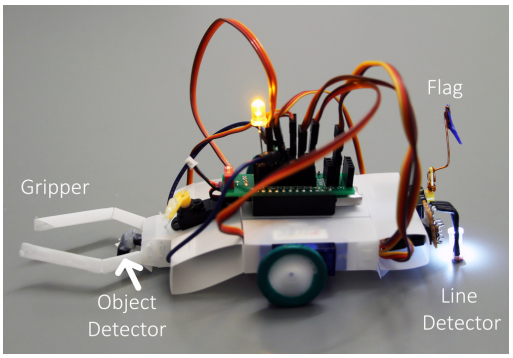
Table 8.2: Groundings for the Path-Following Fetch Robots

Functional Proposition	Line Follower	Wall Follower
Move forward and left	Wheel 1	Wheel 1
Move forward and right	Wheel 2	Wheel 2
Detect path	Line detector	Distance sensor
Detect object	Touch sensor	Light sensor
Detect goal	UI toggle switch 1	Microphone
Secure object, release object	Gripper	Forklift
Indicate object secured	UI toggle switch 2	LED
Indicate complete	Wheel 1 & Flag	Buzzer

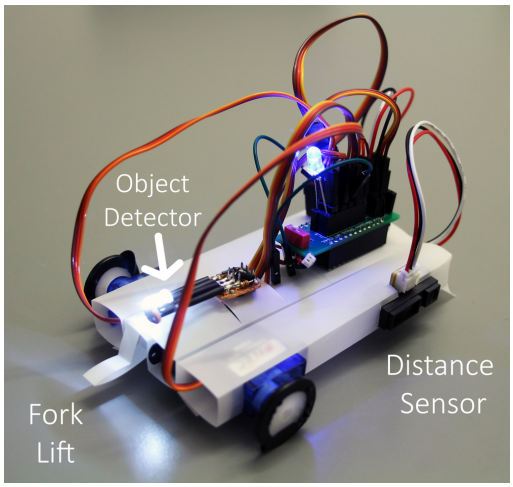
Both cases involve two propositions, `releaseObject_md` and `secureObject_md`, that are mapped to the same port of the same component – either `Gripper` or `Forklift` depending on the robot. In addition, the line follower instantiation maps both `indicateComplete_md` and `leftForward_md` to the same wheel servo in order to indicate completion with a "victory dance" behavior by spinning in a circle or to drive forward as appropriate. `LTLMoP` detects these potential conflicts and automatically generates additional statements necessary to enforce a mutual exclusion condition on the relevant propositions as shown in Figure 8.7. The user is also notified of this addition, so they can ensure that desired behavior is preserved.

Once programmed with the generated code, both robots performed the desired task. The code snippets for the sensors also include calibration routines for the sensors. The robot prompts the user to provide the minimum and maximum values for each sensor, then uses this information to determine suitable threshold values for converting analog readings to boolean variables for the state machine inputs. For example, the line follower will be placed over white and then over black, and the wall follower will be placed near a wall and then far from a wall. This also grants the user some runtime control over the behavior; for example, they can adjust how close the robot stays to the wall by adjusting the calibration positions.

The line-following robot design also includes UI elements; the provided Android app will automatically connect to and communicate with the generated robot via Bluetooth and display the appropriate user interface. In this case it will contain two toggle switches: one can be pressed to indicate the goal state and thereby stop the robot, and the other automatically updates in real-time to reflect whether the robot has grasped the object.



(a) This robot follows the path using a line follower, detects objects using a touch sensor, grasps objects using a claw, detects the goal location via user input with an Android UI switch, and indicates completion via a raised flag as well as a victory dance.



(b) This robot follows the path using a wall sensor, detects objects using a light sensor, grasps objects using a forklift, detects the goal location via a user input with a microphone, and indicates completion via a buzzer.

Figure 8.8: Two very different physical instantiations of a fetch robot are generated for the same task description by choosing different components to ground the same functional propositions. Both successfully exhibit the desired abstract behavior.

Table 8.3: Performance of the Path-Following Fetch Robots

Metric	**Result**		
	Line Follower	**Wall Follower**	
Approximate design time	30	30	min
Approximate fabrication time	60	45	min
Approximate cost	30	45	USD
Mass	64.1	72.1	g
Speed	11.1	11.0	cm/sec
Maximum gripper opening	45	N/A	mm

8.6 Summary

The integration of `LTLMoP` with the robot compiler allows the user to interact with the system at a more abstract level and design based on desired behavior. Starting with an idea of how the robot should act, and afterwards deciding what it should look like to enable those actions, is often a more intuitive approach for novice users than starting with a vision of what components should comprise the robot.

By providing the user with suggestions and breaking the process into a series of well-defined computer-assisted tasks, casual users are able to generate robots for their tasks. In addition, the design feedback and the rapid prototyping methods allow this to be an iterative process; the task descriptions as well as the components and parameters can be adjusted in response to both simulation and fabricated robots.

With this extension, the robot compiler can address a wider range of users and tasks. The barrier to entry for robotics is lowered even further, and the generated robots are even more sophisticated. This therefore marks a significant advancement towards the vision of customized on-demand personal robots.

Chapter 9

Higher-Level Algorithms: Behavioral Verification and Simulation

Life is trying things to see if they work.

– Ray Bradbury

Contents

A second extension to the robot compiler features `React`, a new programming language designed for robots. It provides an alternative framework for specifying behaviors, and adds tools for verification, simulation, and parameter determination that can handle systems of robots. It thereby furthers the paradigm of treating customized robots as software systems.

The concepts presented in this chapter are also discussed in [6]

To integrate with the robot compiler, `React` specifies the actions that the desired robot may take as well as sequencing constraints to create a controller program. Each action is then realized in the final assembled robot as an integrated module from the component library, yielding the required electronic devices, structural elements, and software. This may be done for multiple robots simultaneously, and overall system-wide behavior can be verified such as collision avoidance. Parameterized models can be obtained from the robot compiler, and then both physical parameters and controller parameters can be explored concurrently to determine a robot structure and a controller that guarantee desirable behavioral properties. A controller running code generated by `React` can directly interact with the generated robots by using the robot compiler's protocol for interacting with data ports.

In this way, `React` represents a powerful extension for intermediate users. By writing code in this language, robot structures and controllers can be closely coupled, concurrently simulated, and adjusted to achieve high-level behavioral goals.

9.1 Overview of the `React` Language

`React` is a partly event-driven, partly timed language that focuses on supporting the development of robot controllers. The basic building blocks of `React` include finite state machines for the control structure, periodic loops, and event handlers. It simplifies controller programming with simple, clearly defined semantics and domain-specific constructs that eliminate much of the boiler-plate code characterizing other approaches. It is designed as a domain-specific language embedded in a general purpose programming language.

As a domain-specific language for programming robots, `React` simplifies controller programming and closely integrates with design and manufacturing tools to verify the robot and its controller. It incorporates several expressive and powerful programming constructs such as periodic control loops [50, 58], event-driven programming [91, 92], and finite state machines to modularize computation and analysis. These constructs allow developers to write concise control logic with clear semantics. `React` is implemented as a domain-specific

```
controller ::= Robot ident
                 variable declaration*
                 [ initialState ident]
                 ( handler | task | state )*
     state ::= state ident ( handler | task )*
   handler ::= sensor type ident stmnt
             | on stmnt
      task ::= every integer stmnt
     stmnt ::= nextState ident
             | publish ident expr
             | statement in the host language
```

Figure 9.1: The React syntax includes the base Robot object as well as constructs for handling states, events, and periodic or event-driven tasks.

language in Scala to facilitate interoperability with existing robotic ecosystems; the runtime can interface with the Robot Operating System (ROS) [67] through RosJava, or directly with a microcontroller running code generated by the robot compiler by using the message protocol described in Section 4.2.1.

React has formally defined semantics that enable the use of automated verification tools, including a model checker and a satisfiability modulo theory (SMT) solver. Instead of exploring only specific execution traces as typically done via simulation, these tools allow an exhaustive search of the state space, providing better coverage of the possible behaviors at the expense of a higher computational cost. The model checker returns a complete state trace in the case of an error, simplifying debugging of the concurrent system since errors that depend on a specific interleaving of events are otherwise hard to reproduce.

9.1.1 Syntax and Semantics

Figure 9.1 presents the abstract syntax of React. As a domain-specific language, it only introduces the domain-specific constructs; the host language, Scala, provides the expressions, statements, and types. Some core features introduced in React are presented below.

Robots are the base objects in `React`. Like actors in the actor model [91], robots execute concurrently and exchange messages as events. A robot connects to and manages a variety of hardware elements, and is represented by a set of five variables:

- a unique identifier
- the current control state
- a map from identifiers to values, representing the state of the controller's local variables
- a list of pending events to send
- a set of periodic tasks associating operations with an amount of time remaining until those operations should be executed

Events and Labels realize the communication between robots, sensors, actuators, and external inputs. There is a specific sensor handler as well as a generic event handler. Both allow the programmer to provide a partial function that executes each time a received event matches one of several provided patterns. Events are then associated with topics; with the ROS runtime, there is a topic for each robot identifier that carries associated events. Sensor event handlers, depending on the runtime, require connections to specific hardware resources.

A transition between controller states is labeled by an event or a time, and events store both an identifier and the information to exchange during the communication.

Periodic Loops are usually used for main control logic that consists of one or more periodically executed functions. Because `React` follows the sample-hold controller paradigm, it does not provide guarded transitions as in some other languages. This approach avoids anomalies associated with guarded transitions and increases the analyzability of programs.

Finite State Machines offer modularity within robots by allowing the user to encapsulate event handlers and controllers within control states. These states are similar to the control modes of hybrid automata [54], and to those generated by the integration with `LTLMoP` discussed in Chapter 8. In `React`, each robot identifies an initial state and then the body of the controller uses a `nextState` method to transition between control states.

9.1.2 Evolution of a Single Robot Controller

The model of computation used in React, as is standard in the field of reactive systems, assumes computation is instantaneous so that time elapses only between events. This model accurately captures the behavior of the system as long as the computations execute quickly enough to enable the system to meet its periodic loop deadlines. A worst-case execution time analysis [52] can therefore be used to check the accuracy of the model. Tasks are executed periodically; a task is rescheduled if the controller stays in the same state upon completion, but if the controller moves to a new state then that state's tasks are scheduled.

9.1.3 Multiple Robot Controllers

React assumes that time is global and elapses at the same speed for every robot. Even though real systems are subject to clock drift, global time semantics are required to successfully add multiple physical robots to the system. It is also assumed that the number robots is constant during a single state transition, but that the number of robots may change between transitions if robots are turned on or off.

For communication, React uses publish/subscribe semantics similar to broadcast calculi [92]. Every process that can handle a message at the time it is produced will receive it. When interacting with robots running code auto-generated from the robot compiler, the message format protocol developed in Section 4.2.1 is used to easily interface with the virtual data port network.

9.2 Integration with the Robot Compiler

While `React` provides a framework for programming existing robots, it can also help drive the design of custom robots. It can be used to simulate single robots or systems of multiple robots to validate desired behavior, and can also be used to determine a suitable set of parameters when defining the robot design.

To do so, the robot compiler has been augmented to output a system of equations that describe the dynamics and kinematics of the user-specified robot. This model is correspondingly parameterized, and details the evolution of the robot state as a function of its control inputs. In the extracted model, each structural element is represented by a 3D vector for position and a unit quaternion for orientation. The dimensions of the elements and joints correspond to equations that appropriately relate the position and orientation variables. Active elements, such as motors, also have specific associated equations.

In `React`, these models of robots are represented as robots operating in parallel with the controllers. Compared to a robot controller, a model has only one control state, but has some additional special variables that represent the position of the robot in space and other parameters related to physical objects such as speeds and angles of joints. These variable are updated during the passage of time according to the robot dynamics and kinematics.

`React` can then be used to create a controller for the robot or system of robots that defines the behavior and the interactions with the environment. This control code is executed by a central processor overseeing all robots in the system, each of which is running code automatically generated by the robot compiler as discussed in Chapter 6. Depending on the components that are selected, each physical robot will receive messages related to its actuators and periodically send messages related to its sensors by using the message protocol within the software template model of Section 4.1. Then the model is subject to the same semantics as a typical robot controller in `React`, except for the inclusion of the kinematic and dynamic evolution of the system.

9.2.1 Systematic Testing and Simulation

The goal of the `React` verification is to perform an exhaustive, or bounded, exploration of the system's behaviors to check important properties of interest such as line-following for a single robot or collision avoidance in a multi-robot system. A model checker is implemented to check such safety properties, represented as explicit state for the `React` program. It uses the dReal SMT solver [93] to reason about the equations generated by the robot compiler, alternating the discrete transitions of the controller with the continuous evolution of the robot.

This verification checks two kinds of properties: runtime exceptions such as uncaught exceptions, and user-defined properties such as two robots not colliding. The user must provide a verification scenario that specifies the world, including the relevant robots and objects. To close the system under test, the user must also provide ghost agents that simulate user input or any other active part of the environment.

Verification is known to be computational hard in general. `React` is therefore designed such that it can be efficiently mapped to hybrid automata, the theoretical foundation of the verification method. Moreover, it implements the following optimizations:

- Discretization of the world, and use of interval domains for continuous variables [94]. Such an abstraction can deliver an over-approximation or an under-approximation of the system's behaviors. In `React` case studies, it has been observed that the over-approximation leads to a large number of spurious counterexamples; the under-approximation may miss some errors, but the counter-examples it finds are more interesting for the development process.

- Commutativity analysis for the events. When multiple events or periodic tasks occur at the same time, all permutations of those events may need to be explored. To reduce the number of interleavings that need to addressed, a commutativity analysis between events is implemented [95].

- Quotient time [96], since the controllers have periodic behaviors. The least common multiple of the periods of all tasks can be computed to obtain a global period p, and continuously increasing time t can then be replaced by $tmodp$.

9.2.2 Design Synthesis and Parameter Exploration

Beyond verification, the system can also work with parametric models; in this setting, verification becomes synthesis. Instead of verifying that a finalized design exhibits a desired property, parameter values are determined that enable an incomplete model to satisfy the property. The synthesis algorithm solves both the discrete controller steps and the continuous part of the model simultaneously using a bounded synthesis. The controller executions are unfolded into a tree that encodes the possible executions as a logical formula where different branches of the tree correspond to disjunctions. This controller formula is then conjoined with the formulas describing the robot's kinematics and with the property of interest, and the final formula is sent as a single query to the solver. If the query is satisfiable, then the solver delivers parameter values that generate a model satisfying the property.

Challenges with this approach include the depth of unrolling, which creates large formulas, and the quantifier alternation. A limited unrolling depth means that this method is applicable when short runs are representative of the robot's behaviors. The quantifier alternation is in the best case supported by the solver, but otherwise it is possible to use methods such as the CEGIS algorithm [97].

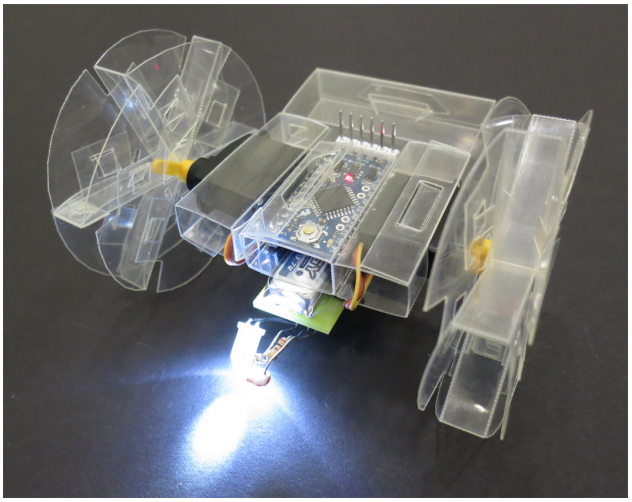


Figure 9.2: A two-wheeled robot is equipped with a line detector, consisting of a photoresistor and an LED, and a Bluetooth module. This robot is wirelessly controlled by a `React` program to follow the edge of a line.

9.3 Case Studies

This section presents examples of using `React` and the robot compiler to program, test, and build robots. Robot control, simulation, verification, and synthesis are demonstrated through a series of sample applications.

9.3.1 Single-Robot Behavior: Line-Following Robot

The process flow of using the robot compiler with `React` is demonstrated by creating a program that controls the robot in Figure 9.2 and causes it to follow the edge of a line. The software running on the robot is auto-generated by the robot compiler, and an external computer runs `React` control code. The computer controls the robot via Bluetooth by sending messages to data ports as specified by the software template model's message format.

Figure 9.3 shows the `React` code that produces the desired line-following behavior. The Seg turns to the right when it is over a black surface, and turns to the left when it is over a white surface. Since turning is accomplished by only activating one wheel, the robot pivots around the opposite wheel and progresses along the line. This is the same behavior that

```
class FollowTheLine(port: String) extends Robot(port) {
  var onTarget = false
  sensor[Bool](lightSensor){case Bool(b) => onTarget = b}
  every(100) {
    if (onTarget) {
      publish(servoLeft, Int16(15))
      publish(servoRight, Int16(0))
    } else {
      publish(servoLeft, Int16(0))
      publish(servoRight, Int16(15))
    }
  }
}
```

Figure 9.3: This `React` code causes the Seg robot to follow the edge of a line.

was previously implemented by graphically connecting data ports in Section 7.1.1, but it is now demonstrated in the context of the new programming language that will be able to perform additional simulation and verification upon the program. The graphical design used to generate the physical robot presented here is the same as that previously discussed in Section 7.1.1, and the generated microcontroller code allows the robot to be controlled via Bluetooth from the off-board controller.

The controller in Figure 9.3 extends the class `Robot`, which takes an argument that defines a namespace when using ROS or a data port ID when controlling an Arduino via the robot compiler. Inside the controller are variables such as `onTarget`, an event handler, and periodic loops. The first element is the event handler that receives the readings from the light sensor, and the identifier inside parenthesis specifies how to connect to the sensor. This would be a ROS topic if using ROS, but in this example is a constant since it is the ID of the sensor's data output port. These IDs are presented to the user as one of the `DataComposable` outputs when the robot compiler processes the design. The next element in the code is a periodic loop, which in this case sets the motor speeds once every 100 ms. As with the sensor, the values of `servoLeft` and `servoRight` are generated by the robot compiler and are IDs of the actuators' data input ports. Since the chosen Arduino works with 16-bit integers, the values are wrapped into `Int16` messages.

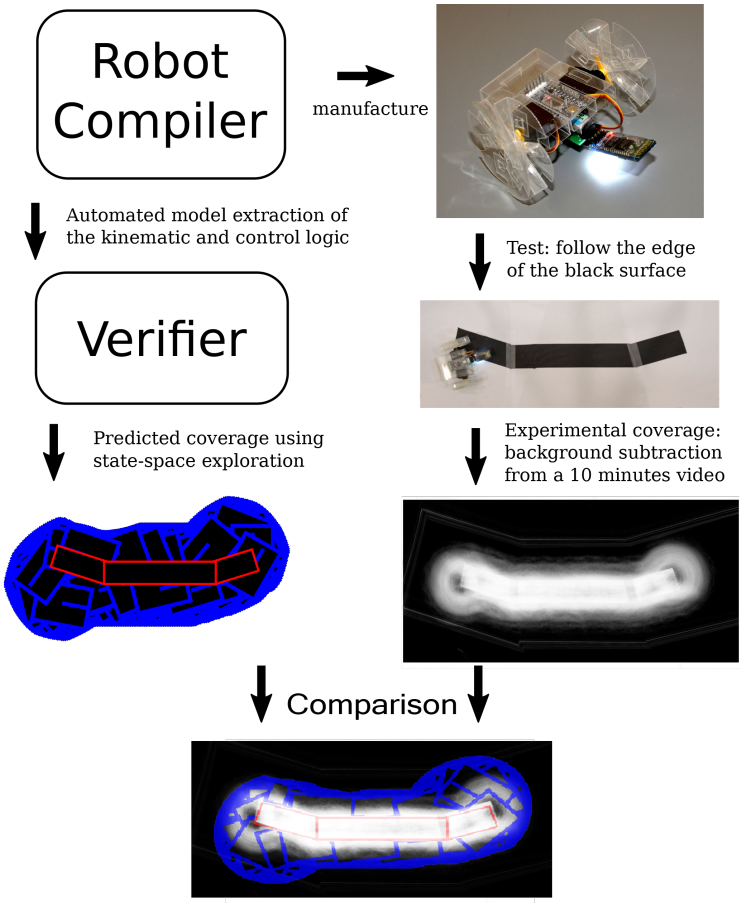


Figure 9.4: The line-following robot generated by the robot compiler can be controlled by a central computer running `React` code, and the simulated behavior can be compared to the observed behavior of the fabricated robot. Here, bounding boxes indicate simulation results and video background subtraction yields experimental results.

By automatically extracting the kinematics and control logic from the model of the robot, the model checker can analyze and visualize the resultant behavior. Figure 9.4 presents the analysis results, which took 12 seconds to compute using a discretization of 1 mm and which contains 133,487 states. On the left, the figure shows the simulated coverage according to the model checker; a bounding box of the robot is shown for every reachable state. On the right, the figure shows the actual coverage of the fabricated robot over a period of ten minutes; background subtraction is applied to footage recorded by an overhead video camera to obtain robot locations, and the brightness of the final image indicates how often the robot visited a particular location. By comparing these images, it is evident that the results of the simulation accurately reflect the experimental results of the fabricated robot.

9.3.2 Multi-Robot Behavioral Verification: Avoiding Collisions

`React` can also be used to verify properties about the behavior of systems that may contain more than one robot. In addition, these simulations can be used to determine appropriate parameters for the robot controllers such that the desired property is manifested. To demonstrate this, a system involving multiple robots is analyzed and verified for a complex safety property, namely the absence of collisions.

In this experiment, two copies of a wheeled robot with an infrared distance sensor are generated from the robot compiler as shown in Figure 9.5. The distance sensor is mounted on a servo and can thus be turned from side to side. As with the previous example, each robot is fabricated by following designs and instructions generated by the robot compiler, and the microcontrollers are programmed with the auto-generated software. They are then controlled via Bluetooth messages by a computer running `React`.

The controller causes each robot to alternate between two control modes. First, they scan the area in front and on the sides to determine the closest object by rotating the distance sensor. Then, depending on the distance of the closest object, they either move forward for a fixed duration or turn in place for a fixed angle.

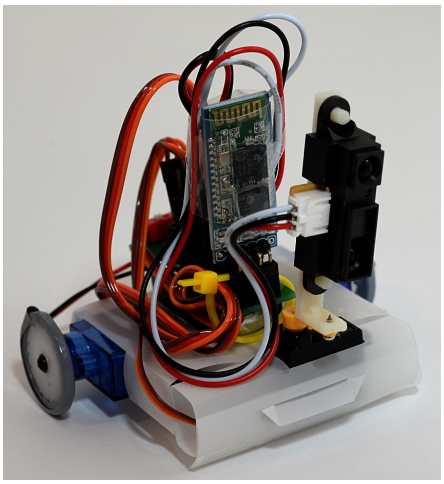


Figure 9.5: A two-wheeled robot is equipped with a distance sensor mounted on a servo. The sensor can be rotated back and forth to sweep a wide area. `React` code can be used to control multiple copies of this robot such that they do not collide with each other or with obstacles.

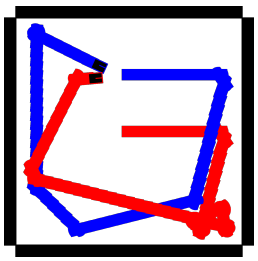


Figure 9.6: A counter-example is found by the verification system that indicates the current parameters may result in the robots colliding.

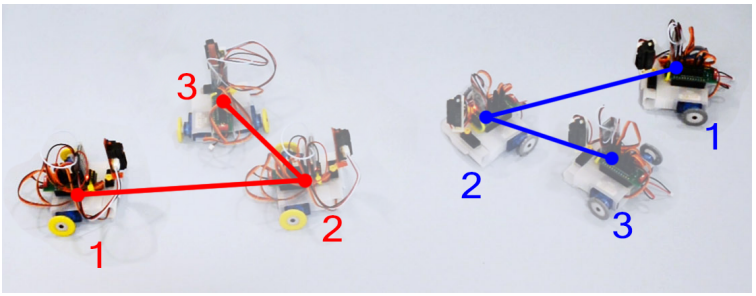


Figure 9.7: The fabricated robots successfully drove around the arena without colliding with each or with the walls. Here, the robot images indicate places where the robots were stopped and using the distance sensor to scan for obstacles. At point 2, they avoid each other by turning.

In the model checker, a scenario was created with two robots and used to test different configurations of the controller. The scanning breadth of the sensor sweep when the servo rotates, the speed of locomotion, the distance threshold for moving forward, and the turning angle are all adjustable in order to determine suitable parameters that ensure collision avoidance. As a counter-example, Figure 9.6 shows that a collision can occur if the robot moves too fast for the current safe distance.

Once a satisfactory set of parameters is obtained, the robots are built and used to experimentally confirm the result. Figure 9.7 shows the two robots in a pen being controlled by a central controller via Bluetooth. The robots scan the surrounding area for obstacles while stopped, and then drive forward or turn depending on whether a nearby obstacle was detected. In the compiled figure, the robot images represent points at which they were stopped. At point 2, the robots avoid a collision by detecting each other and deciding to turn rather than drive forward. As desired, the robots successfully explored the area without colliding with each other or with the walls.

For this experiment, the model checker used a 2 m x 2 m test environment with a 1 cm discretization. Each robot had a bounding box of 9 cm x 12 cm. For the collision detected in Figure 9.6, 24,642 states were explored in 1.7 minutes. For the safe case with suitable parameters, 1,510,525 states were explored in 11.57 minutes.

9.3.3 Design Synthesis and Parameter Determination: Manipulator Arm

The `React` environment can also be used to determine suitable parameters for the components chosen in the robot compiler. In this case, parameters are synthesized for an incomplete robot design and for its controller simultaneously. To demonstrate this, a printable arm composed of three segments was used as shown in Figure 9.8. The goal for the arm is to repeatedly visit three different target zones; to illustrate the performance, the arm holds a pencil and traces its path upon a sheet of paper that has the targets marked. The robot design has the length of each segment as a parameter, and the controller has the commanded joint angles as parameters.

The system automatically unfolds the controller tree to determine the commands sent to the motors, unfolds the equations describing the arm's kinematic model over these outputs, and adds additional constraints that encode the targets to reach.

The experiment was performed with two different models: an arm with two segments (one revolving joint), and an arm with three segments (two revolving joints). It is not possible to meet the objective with the first configuration because one joint is not enough to reach the three targets, and the system quickly verifies that this is the case. The second configuration actually has many solutions, and the system successfully returns one of them. In these cases, the model has an accuracy of approximately 1 mm, and the model generated formulas with 107 variables and 434 constraints. The solver determined that there was no solution for the two-segment configuration in 12 seconds, and found a solution for the three-segment configuration in 2.9 minutes.

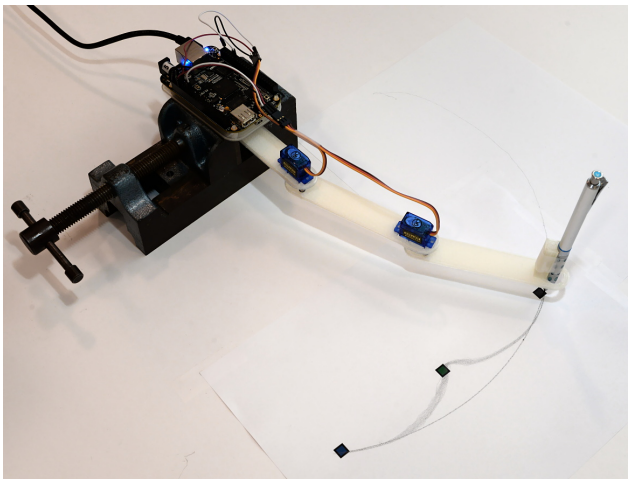


Figure 9.8: A three-segment plotter arm is used to demonstrate design synthesis; the lengths of the segments and commanded joint angles are all parameters that are explored by the system to determine a configuration that can reach the indicated three target destinations.

An arm was then fabricated with the parameter values derived by the solver, and used to experimentally confirmed that the objectives are satisfied. The arm shown in Figure 9.8 was fabricated using a 3D printer, and uses a Beaglebone Black as the controller.

9.4 Summary

By integrating the robot compiler with `React`, its capabilities are extended to include more sophisticated simulation, verification, and design synthesis. Complex controllers can be written for single robots or for systems of robots, and the resulting behavior can be simulated and analyzed. Controller parameters and `Component` parameters can be explored simultaneously to generate robots that satisfy desired behavioral guarantees.

In this way, customized personal robots can be rapidly tested and iterated in software and with physical instantiations. This augments the idea of software-defined hardware to include behavior, making it easier to address physical situations with custom hardware. This can allow users to generate swarms of rapidly fabricated robots that collectively address a target task, bringing the goal of on-demand personal robotics closer to reality.

Chapter 10

Conclusion

The journey of a thousand miles begins with a single step.

– Lao Tzu

Contents

10.1 Future Work

The current system embodies many important advances towards the goal of pervasive personal robots, but numerous steps remain to be taken. The most basic next step is to expand the component library by having more users interact with the system. Through the creation of new robots using the inherently iterative design process, many new hierarchical and base components will be derived that address new challenges and increase the applicability and usefulness of the library. Additionally, a larger set of functioning designs provides a stronger basis upon which the system can make autonomous decisions. Algorithms can be written that allow the system to learn from user designs, enabling more sophisticated autonomy and recommendations while making the system's interactions with the user more intuitive.

As more robots are created with the system, the iterative paradigm will be applied not only to each individual robot but also to the system as a whole. More extensive testing will undoubtedly lead to a more versatile system that addresses currently unforeseen challenges. This testing will be important for the robot compiler as a whole, and also for developing the serial communication protocol. The current implementation of the protocol has been tested on Arduino microcontrollers, and extending this to other platforms will provide an instructive means of demonstrating its adaptability. More specific timing tolerances can also be determined, and the serial library can be executed in parallel with other interrupt-based libraries to test how it interacts with other commonly used time-sensitive operations.

One significant application of the robot compiler system that requires additional user testing is education. The preliminary curriculums developed around some of the printable robots and the robot garden can be expanded and developed further, covering more topics and applying to a wider range of experience levels. Customizable on-demand robots have the potential to significantly impact the engagement of young students with computer science and engineering, and this avenue should certainly be explored in greater depth and actively pursued in real classroom settings.

An important direction in which to scale the system as new scenarios are explored will be adding more output formats and fabrication processes. This may include, for example, making robots from metal instead of plastic in order to generate stronger robots on larger scales. This would simply require a few additional plugins to the mechanical composable, and a few new library components added to the electrical and software systems. In addition, the electrical platform can be expanded by allowing for more customizable fabrication techniques such as PCBs, and further exploring the realm of analog circuitry by integrating simulation tools such as LTSpice. The software generation infrastructure can also be extended by providing more snippet directives and providing more integrated support for multiple programming languages.

Finally, the user interaction with the system can be abstracted to an even higher level. Tools have been discussed that enable the description of tasks and the analysis of behavior,

but they are not yet at the point of natural language interactions. Developing ways for the system to reason about behavior and make more intelligent design choices would reduce the burden on users and allow them to focus on the desired goal to a greater extent. Allowing people to interact with the robot compiler in the same way they would interact with a human designer would mark an important turning point in the development of custom robotics and change the way society interacts with technology on a daily basis.

10.2 Conclusion

The presented system achieves many milestones along the route to pervasive personal robots. Integrated robot designs can be synthesized from high-level user specifications, and the rapidly fabricated robots can immediately perform target tasks.

The key principle of the system is modularizing robotic elements in a manner that allows for the encapsulation of all necessary subsystem design information. Using a library of these modularized robotic components, novice users can compose arbitrarily complex electromechanical devices in an intuitive interface using a paradigm that already exists in other common customization industries. The implementation details are managed behind the scenes, with the system automatically maintaining integrated co-design information throughout the hierarchical component tree. Using domain-specific guidelines embedded in the modules, the system can perform design verification and modification and then generate complete fabricable outputs. These include electrical layouts and wiring instructions, mechanical drawings, control software, and user interface software.

General frameworks for software generation and inter-controller communication have also been developed that enable the rapid generation of complex networks of microcontrollers. The software template model and associated code snippets allow experts to easily add new components to the library in a microcontroller-independent manner with minimal amounts of new code. The robot compiler pools together and modifies code snippets from throughout the design to create coherent software packages for the final robot.

A custom serial communication protocol enables mesh networks of microcontrollers by providing a robust way to exchange data via two-wire software serial ports. This novel protocol synchronizes transmitters and receivers using a receiver-generated clock signal, eliminates message collisions and buffer overflows, and speeds up transmissions for longer messages. Incorporating this protocol into the robot compiler allows the system to automatically insert controllers throughout the robot design in accordance with the mechanical layout, and to achieve transparent information flow throughout the network in a reliable manner.

Numerous case studies illustrate the potential for the robot compiler to produce varied electromechanical devices for many different tasks, exploring several potential applications of this system and demonstrating the advantages of co-designing subsystem outputs. Origami-inspired print-and-fold robots enable inexpensive fabrication and facilitate rapid prototyping. They can then be controlled by automatic user interfaces or user-added control code. Furthermore, they can be programmed with automatic control code that implements behavior specified graphically by the user via cross-discipline component connections. These robots have been used for a variety of applications, including education, and a sample curriculum has been designed around one of the auto-generated robots for use in classrooms where each student could have access to a customized robot. The system has also aided the creation of a large distributed robot garden for the visualization of computer science concepts and the evangelization of technical education. Finally, the electrical and software subsystems have been leveraged to produce products that address daily tasks and custom projects that would otherwise require long design processes repeated for each specific application.

Extensions to the robot compiler provide further input abstraction, allowing for an even more intuitive experience for novice users. Behavioral task descriptions can be provided as initial input, and the system will guide the user through the process of choosing appropriate components from the library for instantiation. The `LTLMoP` program converts this behavior to a finite state machine that the robot compiler can implement as microcontroller code, allowing the generated robot to immediately perform the specified function. Example robots designed in this way demonstrate the ability of the system to address user-specified tasks, and for multiple robot configurations to accomplish the same abstract behavior.

Additionally, tools are incorporated for simulation and parameter exploration that can handle single robots or systems of robots. The `React` programming language facilitates behavioral programming and enables simulations that can demonstrate system-level behavioral guarantees. Integration with the robot compiler allows for parameter exploration, determining suitable robot configurations and controllers that satisfy desired behavioral guarantees. Example case studies demonstrate the ability for simulations to match the observed behavior of generated robots, and for structures and controllers to be modified for a particular task.

Overall, the robot compiler enables rapid generation of customized robots by novice users. It works towards the goal of on-demand robots in everyday life, where people can think of robots for physical tasks as they currently think of software for computational tasks. Robots such as these have captured imaginations for generations, and with current technology we are on the verge of bringing those fantasies into reality.

Figure 10.1: The robot compiler has successfully produced a wide range of robots and electromechanical devices. With continued improvement, the vision of a robot fabrication kiosk may soon be within reach.

Bibliography

[1] Ankur Mehta, Joseph DelPreto, and Daniela Rus. Integrated codesign of printable robots. *Journal of Mechanisms and Robotics*, 7(2):021015, 2015.

[2] Ankur Mehta, Joseph DelPreto, Benjamin Shaya, and Daniela Rus. 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*, pages 2892–2897. IEEE, 2014.

[3] Joseph Delpreto, Ankur Mehta, and Daniela Rus. Cogeneration of electrical and software designs from structural specifications (extended abstract). In *Robotics Science and Systems (RSS), Robot Makers Workshop*, 2014.

[4] L. Sanneman, D. Ajilo, J. DelPreto, A. Mehta, S. Miyashita, N.A. Poorheravi, C. Ramirez, Sehyuk Yim, Sangbae Kim, and D. Rus. A distributed robot garden system. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 6120–6127, May 2015.

[5] Ankur Mehta, Joseph DelPreto, Kai Weng Wong, Scott Hamill, Hadas Kress-Gazit, and Daniela Rus. Robot creation from functional specifications. In *The International Symposium on Robotics Research (ISRR)*, Sestri Levante, Italy, September 2015.

[6] Damien Zufferey, Ankur Mehta, Joseph DelPreto, Stelios Sidiroglou-Douskos, Martin Rinard, and Daniela Rus. Talos: Full stack robot compilation, simulation, and synthesis. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, 2015 (submitted).

[7] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[8] Shane Farritor and Steven Dubowsky. On modular design of field robotic systems. *Autonomous Robots*, 10(1):57–65, 2001.

[9] Jay Davey, Ngai Kwok, and Mark Yim. Emulating self-reconfigurable robots-design of the smores system. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4464–4469. IEEE, 2012.

[10] G.S. Hornby, H. Lipson, and J.B. Pollack. Generative representations for the automated design of modular physical robots. *Robotics and Automation, IEEE Transactions on*, 19(4):703–719, Aug 2003.

[11] littleBits. `http://www.littlebits.cc`.

[12] MOSS Modular Robotics. `http://www.modrobotics.com/moss`.

[13] VEX Robotics. http://www.vexrobotics.com.

[14] LEGO Mindstorms. http://mindstorms.lego.com.

[15] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[16] OpenSCAD.

[17] Marc Freese, Surya Singh, Fumio Ozaki, and Nobuto Matsuhira. Virtual robot experimentation platform v-rep: A versatile 3d robot simulator. In Noriaki Ando, Stephen Balakirsky, Thomas Hemker, Monica Reggiani, and Oskar von Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 6472 of *Lecture Notes in Computer Science*, pages 51–62. Springer Berlin Heidelberg, 2010.

[18] N. Bezzo, Junkil Park, A. King, P. Gebhard, R. Ivanov, and Insup Lee. Demo abstract: Roslab – a modular programming environment for robotic applications. In *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pages 214–214, April 2014.

[19] MakerBot 3D Printer. http://www.makerbot.com/.

[20] RepRap 3D Printer. http://www.reprap.org/.

[21] Constantinos Mavroidis, Kathryn J DeLaurentis, Jey Won, and Munshi Alam. Fabrication of non-assembly mechanisms and robotic systems using rapid prototyping. *Journal of Mechanical Design*, 123(4):516–524, 2001.

[22] Charles Richter and Hod Lipson. Untethered hovering flapping flight of a 3d-printed mechanical insect. *Artificial life*, 17(2):73–86, 2011.

[23] Jonathan Rossiter, Peter Walters, and Boyko Stoimenov. Printing 3d dielectric elastomer actuators for soft robotics. In *SPIE Smart Structures and Materials+ Nondestructive Evaluation and Health Monitoring*, pages 72870H–72870H. International Society for Optics and Photonics, 2009.

[24] Aaron M Hoover and Ronald S Fearing. Fast scale prototyping for folded millirobots. In *Robotics and Automation (ICRA), 2008.*, pages 886–892. IEEE, 2008.

[25] Y. Liu, J. Boyles, J. Genzer, and M. Dickey. Self-folding of polymer sheets using local light absorption. *Soft Matter*, 8:1764–1769, 2012.

[26] Isao Shimoyama, Hirofumi Miura, Kenji Suzuki, and Yuichi Ezura. Insect-like microrobots with external skeletons. *Control Systems, IEEE*, 13(1):37–41, 1993.

[27] S. Brittain et al. Microorigami: Fabrication of small, three-dimensional, metallic structures. *Journal of Physical Chemistry B*, 105(2):347–350, 2001.

[28] Elliot Hawkes et al. Programmable matter by folding. *Proceedings of the National Academy of Sciences*, 107(28):12441–12445, 2010.

[29] M. Tolley, S. Felton, S. Miyashita, L. Xu, B. Shin, M. Zhou, D. Rus, and R. Wood. Self-folding shape memory laminates for automated fabrication. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013.

[30] Cagdas Denizel Onal, Robert J Wood, and Daniela Rus. Towards printable robotics: Origami-inspired planar fabrication of three-dimensional mechanisms. In *Robotics and Automation (ICRA)*, pages 4608–4613. IEEE, 2011.

[31] Paul Birkmeyer, Kevin Peterson, and Ronald S Fearing. Dash: A dynamic 16g hexapedal robot. In *Intelligent Robots and Systems (IROS)*, pages 2683–2689. IEEE, 2009.

[32] C.D. Onal, R.J. Wood, and D. Rus. An origami-inspired approach to worm robots. *Mechatronics, IEEE/ASME Transactions on*, 18(2):430–438, April 2013.

[33] Ankur Mehta, Daniela Rus, Kartik Mohta, Yash Mulgaonkar, Matthew Piccoli, and Vijay Kumar. A scripted printable quadrotor: Rapid design and fabrication of a folded MAV. In *16th International Symposium on Robotics Research*, 2013 (to appear).

[34] Ankur Mehta and Daniela Rus. An end-to-end system for designing mechanical structures for print-and-fold robots. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, 2014 (to appear).

[35] Erik D Demaine and Tomohiro Tachi. Origamizer: A practical algorithm for folding any polyhedron, 2009.

[36] Robert Lang. *Origami design secrets : mathematical methods for an ancient art*. A K Peters/CRC Press, 2012.

[37] Pepakura designer. `http://www.tamasoft.co.jp/pepakura-en/`.

[38] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from temporal logic specifications. *IEEE Trans. Automat. Contr.*, 53(1):287–297, 2008.

[39] Georgios E. Fainekos, Hadas Kress-Gazit, and George J. Pappas. Temporal logic motion planning for mobile robots. In *Robotics and Automation (ICRA)*, pages 2020–2025, 2005.

[40] Scott C. Livingston, Pavithra Prabhakar, Alex B. Jose, and Richard M. Murray. Patching task-level robot controllers based on a local mu-calculus formula. In *Robotics and Automation (ICRA)*, pages 4588–4595, 2013.

[41] A. I. Medina Ayala, Sean B. Andersson, and Calin Belta. Probabilistic control from time-bounded temporal logic specifications in dynamic environments. In *Robotics and Automation (ICRA)*, pages 4705–4710, 2012.

[42] Amit Bhatia, Lydia E. Kavraki, and Moshe Y. Vardi. Sampling-based motion planning with temporal goals. In *Robotics and Automation (ICRA)*, pages 2689–2696, 2010.

[43] Eric M. Wolff, Ufuk Topcu, and Richard M. Murray. Optimization-based trajectory generation with linear temporal logic specifications. In *Robotics and Automation (ICRA)*, pages 5319–5325, 2014.

[44] S. Karaman and E. Frazzoli. Complex mission optimization for multiple-UAVs using linear temporal logic. In *American Control Conference*, pages 2003 – 2009, Seattle, WA, June 2008.

[45] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proc. of the 2nd IJCAI*, 1971.

[46] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – the planning domain definition language – version 1.2. Technical report, Yale Center for Computational Vision and Control, October 1998.

[47] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Where's waldo? sensor-based temporal logic motion planning. In *Robotics and Automation (ICRA)*, pages 3116–3121, 2007.

[48] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[49] Rajeev Alur and Parthasarathy Madhusudan. Decision problems for timed automata: A survey. In *Formal Methods for the Design of Real-Time Systems*, pages 1–24. Springer, 2004.

[50] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, Jan 2003.

[51] Labview system design software. `http://www.ni.com/labview/`.

[52] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[53] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 138(1):3–34, 1995.

[54] Thomas A Henzinger. *The theory of hybrid automata*. Springer, 2000.

[55] André Platzer et al. Analog and hybrid computation: Dynamical systems and programming languages. *Bulletin of EATCS*, 3(114), 2014.

[56] Paulo Tabuada. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009.

[57] Rajeev Alur. Formal verification of hybrid systems. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 273–278. IEEE, 2011.

[58] Thomas A Henzinger and Peter W Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221(1):369–392, 1999.

[59] Oded Maler. Algorithmic verification of continuous and hybrid systems. *arXiv preprint arXiv:1403.0952*, 2014.

[60] Jérôme Feret. Static analysis of digital filters. In *Programming Languages and Systems*, pages 33–48. Springer, 2004.

[61] Ankur Taly and Ashish Tiwari. Deductive verification of continuous dynamical systems. In *FSTTCS*, volume 4, pages 383–394, 2009.

[62] Khalil Ghorbal, Andrew Sogokon, and André Platzer. Invariance of conjunctions of polynomial equalities for algebraic differential equations. In *Static Analysis*, pages 151–167. Springer, 2014.

[63] Khalil Ghorbal, Andrew Sogokon, and André Platzer. A hierarchy of proof rules for checking differential invariance of algebraic sets. In *Verification, Model Checking, and Abstract Interpretation*, pages 431–448. Springer, 2014.

[64] Simulink - simulation and model-based design. `http://www.mathworks.com/products/simulink/`.

[65] Sven Erik Mattson, Hilding Elmqvist, and Jan F Broenink. Modelica: An international effort to design the next generation modelling language. *Journal A*, 38(3):16–19, 1997.

[66] Henry M Paynter. *Analysis and design of engineering systems*. MIT press, 1961.

[67] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[68] Hernando Barragán. Wiring: Prototyping physical interaction design. *Interaction Design Institute, Ivrea, Italy*, 2004.

[69] Casey Reas and Ben Fry. *Processing: a programming handbook for visual designers and artists*, volume 6812. Mit Press, 2007.

[70] YAML file format. `http://www.yaml.org/`.

[71] B.A. Jones, R. Reese, and J.W. Bruce. *Microcontrollers, Second Edition*. Course Technology, 2014.

[72] R. Kamal. *Embedded Systems 2E*. McGraw-Hill Education (India) Pvt Limited, 2008.

[73] J. Cowley. *Communications and Networking: An Introduction*. The computer communications and networks series. Springer London, 2006.

[74] R. Forster. Manchester encoding: opposing definitions resolved. *Engineering Science and Education Journal*, 9(6):278–280, Dec 2000.

[75] Arduino softwareserial library. `http://www.arduino.cc/en/Reference/SoftwareSerial`.

[76] Joseph delpreto's personal website. `http://people.csail.mit.edu/delpreto`.

[77] Ltspice, design simulation and device models. `http://www.linear.com/designtools/software/`.

[78] Adriana Schulz, Cynthia Sung, Andrew Spielberg, Wei Zhao, Yu Cheng, Ankur Mehta, Eitan Grinspun, Daniela Rus, and Wojciech Matusik. Interactive robogami: data-driven design for 3d print and fold robots with ground locomotion. In *SIGGRAPH 2015: Studio*, page 1. ACM, 2015.

[79] Ankur Mehta, Nicola Bezzo, Byoungkwon An, Peter Gebhard, Vijay Kumar, Insup Lee, and Daniela Rus. A design environment for the rapid specification and fabrication of printable robots. In *International Symposium on Experimental Robotics (ISER)*, 2014 (to appear).

[80] The ultra affordable educational robot project. `http://robotics-africa.org/2014-design-challenge`.

[81] The mit seg: An origami-inspired segway robot. `http://sites.google.com/site/mitprintablerobots/`.

[82] Ryuma Niiyama, Daniela Rus, and Sangbae Kim. Pouch motors: Printable/inflatable soft actuators for robotics. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 6332–6337. IEEE, 2014.

[83] Shuhei Miyashita, Cagdas D Onal, and Daniela Rus. Self-pop-up cylindrical structure by global heating. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 4065–4071. IEEE, 2013.

[84] Samuel M Felton, Michael T Tolley, ByungHyun Shin, Cagdas D Onal, Erik D Demaine, Daniela Rus, and Robert J Wood. Self-folding with shape memory composites. *Soft Matter*, 9(32):7688–7694, 2013.

[85] Shuhei Miyashita, Steven Guitron, Marvin Ludersdorfer, C Sung, and Daniela Rus. An untethered miniature origami robot that self-folds, walks, swims, and degrades. In *IEEE International Conference on Robotics and Automation (ICRA), submitted*.

[86] Can an led-filled "robot garden" make coding more accessible? `http://news.mit.edu/2015/can-led-robot-garden-make-coding-more-accessible-0218`.

[87] Neighborhood of innovation.
`http://news.mit.edu/2015/neighborhood-innovation`.

[88] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Translating structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[89] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv SaÊijar. Synthesis of reactive (1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[90] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1988–1993. IEEE, 2010.

[91] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[92] Kuchi VS Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25(2):285–327, 1995.

[93] Sicun Gao, Jeremy Avigad, and Edmund M Clarke. Delta-decidability over the reals. In *Logic in Computer Science (LICS), 2012 27th Annual IEEE Symposium on*, pages 305–314. IEEE, 2012.

[94] Venkatesh Mysore and Bud Mishra. Algorithmic algebraic model checking iii: approximate methods. *Electronic Notes in Theoretical Computer Science*, 149(1):61–77, 2006.

[95] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, volume 40, pages 110–121. ACM, 2005.

[96] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE, 1990.

[97] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.