Distributed Dataflow Machine Controllers

by

Jake Robert Read

B.Arch., University of Waterloo (2016)

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

Author . . . .
Program in Media Arts and Sciences,
School of Architecture and Planning,
January 17, 2020

Certified by . . . .
Prof. Neil Gershenfeld
Director, MIT Center for Bits and Atoms
Thesis Supervisor

Accepted by . . . .
Prof. Tod Machover
Academic Head, Program in Media Arts and Sciences

Distributed Dataflow Machine Controllers

by

Jake Robert Read

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on January 17, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Workflows in Digital Fabrication require coordination across heterogenous computing systems, from the design tools used to describe component geometries to the embedded control systems used to interact with the physical world in order to produce those components. In the state of the art, workflows are typically static and opaque, especially within embedded controllers themselves. This makes them difficult to modify or develop, and places barriers between high level computing and low level control. An opportunity exists to develop an open platform for interoperability and reconfigurability that spans low- and high-level workflow components, that could collapse much of the heterogeneity found in these systems into cohesive representations. To do so, this thesis develops a systems architecture based on reconfigurable graphs of dataflow objects. It embeds virtual dataflow graphs of modular software elements within physical dataflow graphs of modular hardware elements, recasting heterogenous systems as cohesive graphs all the way down. The architecture is reduced to practice across high-level browser computing and low-level embedded control, through mixed networking links. It is deployed on two machine systems: one that collapses path planning and path execution for a small milling machine, marking a departure from the historic use of G Codes, and another that aligns computer vision based measurement with low-level motor control and sensor acquisition, to open access to materials measurement.

Thesis Supervisor: Prof. Neil Gershenfeld
Title: Director, MIT Center for Bits and Atoms

Distributed Dataflow Machine Controllers

by

Jake Robert Read

This thesis has been reviewed and approved by the following committee members:

Acknowledgments

inside the small dome
a candle, lit in the wind
hot fire. hot fire now

About six years ago now I properly entered the world of digital fabrication as a novice. I should first express my thanks to Saul Griffith and Otherlab for hiring me on as an intern, and for agreeing to bring me back sequentially to work in domains I had no formal qualification for. The permission to tinker in Otherlab's basement on nights and weekends brought me to my first futile attempts at solving the problems that this thesis (in part) tackles, and set me down a path that landed me at the Center for Bits and Atoms, where Dr. Gershenfeld has assembled the tiny heaven of equipment and ideas that no mere mortal should expect to exist in one location. For the tools, guidance, and continued internal-state perturbances, and also for the periods of unperturbed time to focus, I have Neil to thank. The privilege of studying and working at MIT is great, and I endeavour every day to use this time in the production of tools that might leave the lab and allow others to build similar environments for themselves and their communities.

To my mother, I send my sincere thanks for the kind of unconditional love that every son should hope for, and that granted me some solace during years spent as a simply 'weird' child, not yet an interesting one. To my father, for the same, and for never failing to answer my questions about internal combustion engines, suspension geometries, hydrodynamics, potential aircraft configurations, manufacturing strategies, drivetrain ratios, hovercraft designs, etc. Also, for teaching me how to drive fast, and sometimes sideways, and to always enjoy the ride. In my brother I find a friend for life, a co-conspirator, and a bouncing board for programming questions who has un-stuck me from not a small handful of problems. To my entire extended family: uncles, aunts, cousins, grandmothers: thank you for the unconditional support.

To my friends in Canada: Elizabeth, Clara, Pat, Thom (and Thom), Fysal, Taylor, Alex, Morgan, Matt: thank you for allowing a misfit of misfits amongst yourselves.

To Connor especially, for believing in the possibility of a triple crown.

To Dr. Forrest Meggers, Dr. Glogowski, and Matthew Bye, I am lucky to count you amongst my mentors.

Thanks of course to my thesis readers, for taking the time out of their busy schedules to engage with the contents of this document, and offer their diverse insights.

My local environment at MIT is so full of brilliance that I feel as though six years - a previously unimaginably long time at school - may not be long enough. For the appearance of friends who inspire - Adam, Devo, - and sages like Sam (the Gandalf to my Frodo) - I am full of thanks. To Sean, for always listening and often helping me out of the DOM, and Erik for recounting tales of C++ compilers past. To Agnes and Gary for some quiet time soaking in the dialectical conversations at the Egg House, the smoothest of spaces. To the Kleins, for always reminding me to DTR, just in time.

To my lab mates at the CBA - past, present and future - thank you for helping to make the space and for filling it with inspiring work. I owe a real debt to Nadya, Ilan, Jonathan and all of the CBA machine builders in the lineage for carving such a strange niche into this heterogeneous space. To collaborators in this work: Ruben, Leo, Jens, Frikk, Jakob, and the extended (and growing) machine building family, to Sherry and Luciano for fostering the global Fab Lab community, and to the extended network of open-source hardware developers, I hope that the labour contained in this thesis will benefit each of us as we endeavour to reclaim the means. Let us meet the challenge together.

Contents

Chapter 1

Introduction

That's the great thing about making things: you get to make 'em how you like 'em.
- Brad Leone, 2019

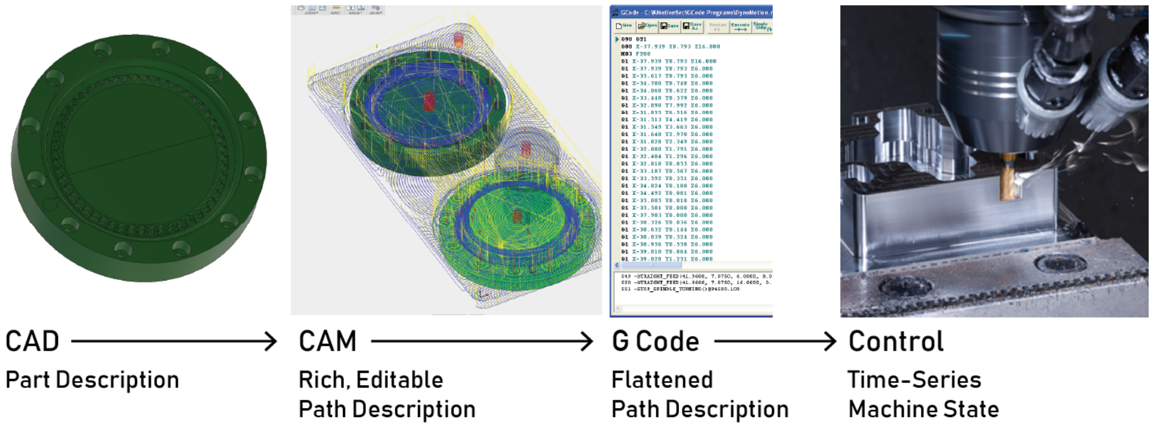
1.1 End-to-end Workflows

In the state of the art, workflows in digital fabrication operate on four or more representations between part description in Computer Aided Design (CAD) softwares, down to the machine control (CNC) outputs that render these descriptions into physical artefact. Figure 1-1 diagrams one of the most common workflows, for CNC Milling: a part is designed in CAD software, where it is then saved as a solid model and loaded into a Computer Aided Machining (CAM) software, where machining plans (paths) for the part are made. CAM softwares output machine-specific static path representations known as G Codes, which are then saved and loaded into machine controllers, where they are finally interpreted by low level controllers that operate motors and process equipment in order to finally manufacture the part.

CNC Milling[1] and FDM 3D Printing[2] produce varied outputs but employ fundamentally similar workflows: each traverses a solid model, a toolpath, and a machine controller. Many of the algorithms, and the desired outcomes, are similar, as are

[1]CNC Mills use cutting tools to subtractively produce artefacts.

[2]Fusion Deposition Modelling 3D Printers use thermoplastic feed stock to additively build artefacts.



```
G90 G21
G00 X-37.939 Y8.793 Z16.000
M03 F200
G1 X-37.939 Y8.793 Z16.000
G1 X-37.939 Y8.793 Z6.000
G1 X-35.617 Y8.793 Z6.000
G1 X-34.780 Y8.748 Z6.000
G1 X-34.060 Y8.622 Z6.000
G1 X-33.448 Y8.379 Z6.000
G1 X-32.890 Y7.992 Z6.000
G1 X-31.855 Y6.516 Z6.000
G1 X-31.513 Y4.419 Z6.000
G1 X-31.549 Y3.663 Z6.000
G1 X-31.648 Y2.970 Z6.000
G1 X-31.828 Y2.349 Z6.000
G1 X-32.080 Y1.791 Z6.000
G1 X-32.404 Y1.296 Z6.000
G1 X-32.818 Y0.855 Z6.000
G1 X-33.187 Y0.567 Z6.000
G1 X-33.592 Y0.351 Z6.000
G1 X-34.024 Y0.180 Z6.000
G1 X-34.492 Y0.081 Z6.000
G1 X-35.005 Y0.018 Z6.000
G1 X-35.581 Y0.000 Z6.000
G1 X-37.903 Y0.000 Z6.000
G1 X-38.326 Y0.036 Z6.000
G1 X-38.632 Y0.144 Z6.000
G1 X-38.839 Y0.324 Z6.000
G1 X-38.956 Y0.558 Z6.000
G1 X-39.010 Y0.864 Z6.000
G1 X-39.028 Y1.251 Z6.000
```

```
549 -STRAIGHT_FEED(41.9600, 7.8750, 6.0000, 0.0
550 -STRAIGHT_FEED(41.9600, 7.8750, 16.0000, 0.
551 -STOP_SPINDLE_TURNING()@94580.109
```

Figure 1-1: Even extremely common digital fabrication workflows, like using a CNC mill to manufacture a part, traverse through a minimum of four different tools and data representations.

the machines themselves. In practice, machines cannot be rapidly modified to accomodate for even these subtle changes in workflows. Instead, custom softwares and bespoke machines are used in both of these cases.

Digital fabrication systems do require this application specificity: each process is unique, and involves new mixtures of algorithm and hardware. However, many core competencies could be shared. Almost all include some aspect of acceleration and motor control, and almost all (hence the name) interface with digital design tools, as well as the humans who operate them. Many benefit from integrated measurement (or probing) systems, and more would benefit from integrated computer vision systems. However, state of the art system architectures make it difficult to re-use core components across varying processes: controllers are typically static and opaque, and rely on outdated representations and hidden internal state in order to operate. Where new applications are developed, much effort is spent re-developing these core competencies into new, standalone devices.

Digital fabrication systems are also largely feed forward, and especially so over workflows. By this I mean that when a machining plan arrives at the machine controller to be interpreted, the outcome is already baked in. When workflows require the use of lengthy and cumbersome representations like G Code, it becomes very difficult to add intelligence to processes. A human machinist can, for example, adjust feed rate and cut depth to adapt to the chatter or resistance they feel during the process of machining: our state of the art controllers can only do their best to faithfully follow paths that were written for them offline. There is not a route with which low level data from online controllers (like motor torques, and vibration measurements), can be sent back 'up stream' to path planning softwares, such that plans might more intelligently be generated.

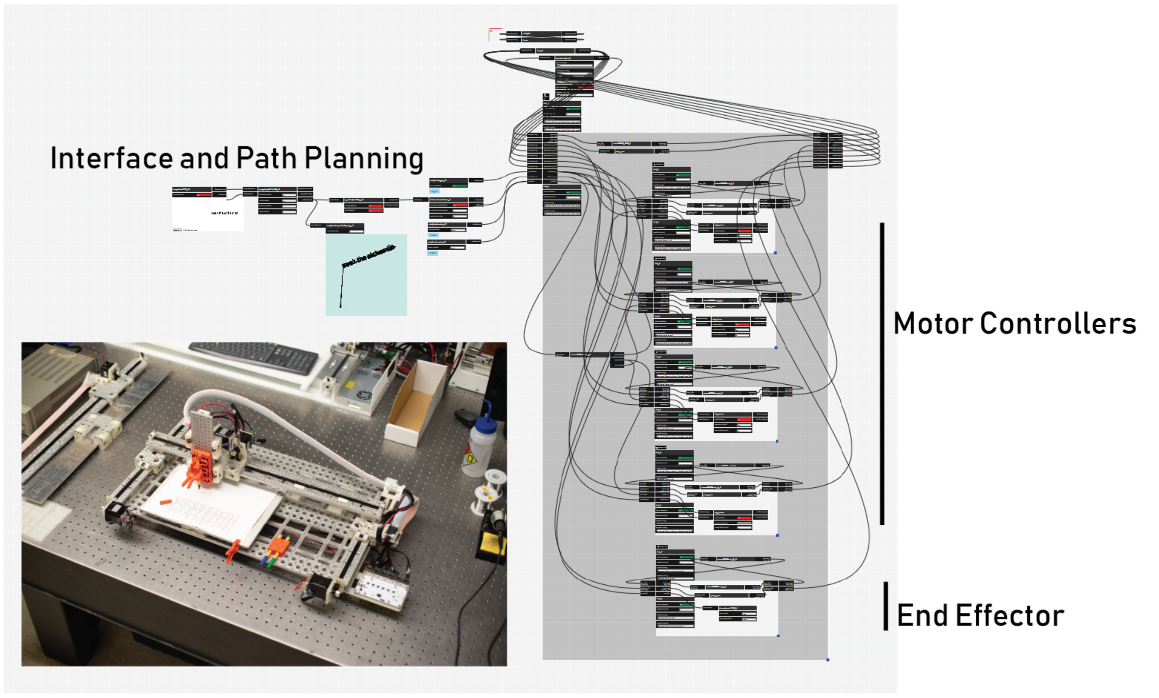


Figure 1-2: A milling machine developed for this thesis that integrates path planning and path execution, under the distributed dataflow platform.

The system architecture presented in this thesis, figured in 1-2, redevelops workflows within a platform where constituent components are reconfigurable across all levels of operation: it delivers modularity in hardware and in software. By doing so it allows core competencies to be re-used across specialized systems, reducing the cost-to-build for new machines, and allowing for rapid reconfiguration of existing machines and workflows.

Rather than attempting to standardize a centralized, monolithic device to accomplish a wide variety of tasks, workflows here are networks of heterogenous computing elements, each of which can be specialized for its particular function. Using a dataflow representation, almost any software or hardware object can be successfully represented, allowing the architecture to capture highly heterogenous codes and hardware devices. Any element in these graphs can receive or transmit data, meaning that low level control elements can feed back data in real time to high level algorithms, opening the possibility of developing more intelligent controllers than the static path interperters that are commonly found in the state of the art. In this architecture, data never traverse static or opaque representations like G Code: instead, workflows are exactly that - flows of data that can be re-routed and modified on the fly. Because machine controllers are made up of collections of networked devices, adding and removing components to modify the physical configuration of a machine can be done rapidly and composing new machines from a stock of existing hardwares is possible. The thesis also develos a tool that allows these graphs of networked controllers to be visualized, edited, and recomposed during runtime, meaning that novices are able to assemble and modify controllers with little or no programming experience.

To accomodate a wide variety of current and future processes, the architecture takes the end-to-end principle as a guiding insight. This idea, championed by early internet architects, simply states that complexity should reside at the edges, not in the centers, of a system [36]. That is: the low levels of system operation should not be burdened with all of the functionality which may eventually be present in the system. Instead, 'systems' should encompass only very small sets of functionality, and advanced system functions should be moved "upward in a layered system closer to the application that uses that function".

1.2 System Heterogeneity

Reconfigurability and re-use of existing solutions is routine practice for software engineers, where standardized runtime environments, compilers, and high orders of abstractions can be used to quickly integrate existing and openly available codes with specialized application specific code. State of the art machine learning and computer vision algorithms are readily available to system developers with libraries like tensor flow [1] and opencv [5]. Indeed, it is even possible for software engineers to rapidly and flexibly expand their access to computing *hardware*, with *Platform as a Service* cloud computing like AWS rising to prominence in the past decade [24]. The benefits of the rapid assembly of software product is evident to anyone who has observed the immense and monstrous growth of the software and internet giants of the last three decades.

Indeed, the rapid assembly of the information economy that has dominated global transformation over the past three decades is built on a foundational platform of inter-networked computing devices. The internet allows software engineers to share code, strategies, and direct solutions with very minimal overhead.

Alas, hardware remains difficult[3] [19]. Why is it that we are so far unable to engineer a control platform that could do for advanced manufacturing what the internet did for computing? Real computing and control challenges must be surmounted in order to develop any physical control system, and doing so within a representation that is easily modifed presents the technical challenge at the heart of the thesis.

The answer lies in the physical heterogeneity of the computing resources used in machine controllers, itself driven by intensive timing constraints that arise from physical process control. For example, a motor controller is a unique piece of hardware that, at its core, switches high voltage and high current devices using computer logic. These power stages switch on the order of 10 to 100kHz, or between one hundred and ten microsecond periods ($10^{-4}s$ and $10^{-5}s$). This is referred to as PWM[4] switching, and while it is possible to achieve in some cases with generalizeable software, it is typically carried out with the aid of specialized circuits [20]. Beyond PWM, controllers must perform many other low-level and high-speed tasks, all requiring tight computational coupling into peripheral hardware functions. These include the manipulation of physical interfaces to data transferring hardware like SPI[5], UART[6], I2C[7], Ethernet PHYs and the like) that bus data between sensors and other computing resources,

[3]A common adage amongst startup developers: 'Hardware is Hard'.
[4]Pulse Width Modulation
[5]Serial Peripheral Interface.
[6]Universal Asynchronous Receiver/Transmitter(s).
[7]A two-wire interface common for reading small digital sensors.

and also in the reading and setting of analog voltages; devices known as ADCs[8] and their output counterparts, DACs[9]. We should not forget Timers themselves, highly accurate clocks that are used to coordinate precise sequences of these tasks.

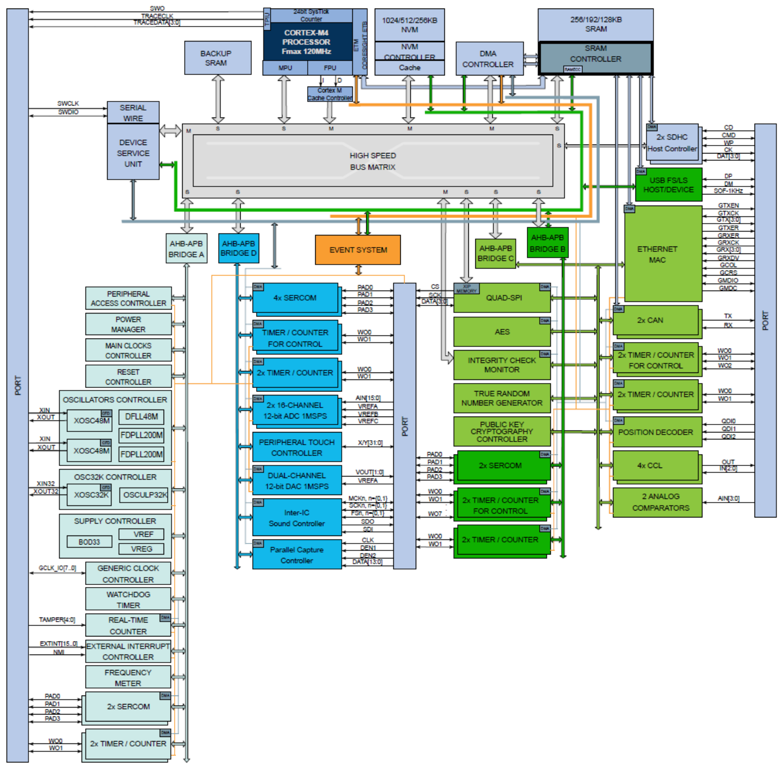


Figure 1-3: An example of a microcontroller, the ATSAMD51, featuring the core CPU, but also featuring a rich set of peripherals: specialized circuits that carry out low-level control tasks, often involving micro- or nano-second timing. The small deep blue rectangle in the top-center is the CPU, all other blocks in the diagram are peripherals that it are manages using specialized codes and registers.

To develop computational interfaces to these circuits, specialized processors known as microcontrollers have been developed in industry. Microcontrollers (also known also as MCUs) include some computing core operating under a familiar Von Neumann Architecture, as well as a collection of these specialized devices known as *peripherals*. Programs written to run on MCUs, known as *firmwares*[10], access peripherals via Special Function Registers (abbreviated to SFR), which are memory addresses whose

[8]Analog to Digital Converters
[9]Digital to Analog Converters
[10]Somewhere between hardware and software!

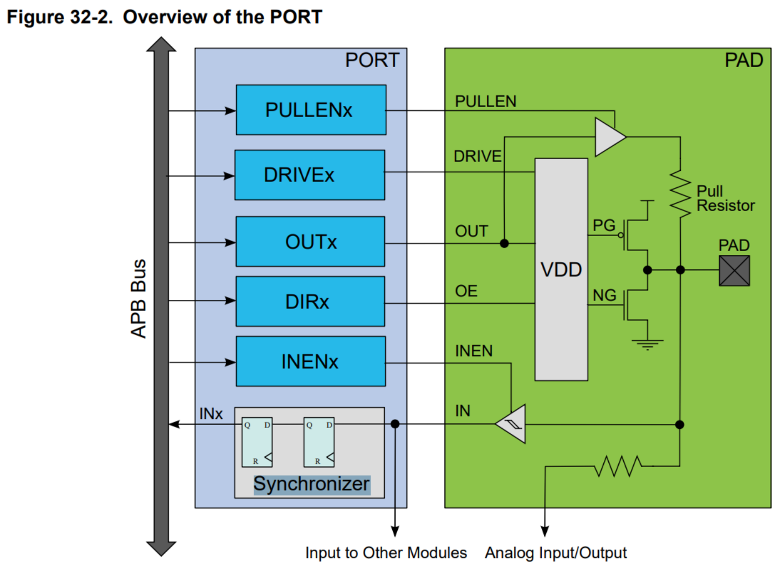


Figure 1-4: An expansion of one peripheral, the Port, from a microcontroller's block diagram.

individual bits are mapped to logic level switches that are connected to the specialized circuitry inside of a peripheral. For example, most microcontrollers feature output 'ports' (a peripheral), whose output states are written by writing a 'one' to a reciprocal memory address; to set the 6th pin on port A, for instance, a software routine would write the byte $0b0100000$ onto one of that ports' Special Function Registers. We might also think of a microprocessor as a small computer surrounded by ASICs[11], interfacing with those ASICs using a memory structure. In turn, those ASICs interface with the physical world by switching (or reading) voltage states.

No two microcontrollers are the same. Each includes a unique set of peripherals addressed at varying SFR addresses. Given the available mixture of microcontrollers, special peripherals, and their connections to hard physical resources like power switches and sensors, broad computational abstractions for their firmwares are not possible to develop. Indeed, particular microcontrollers are chosen for particular applications due to their uniqueness in suitability for that task. Microcontrollers are physically integrated into their application - they are soldered on to a circuit board - and specialized firmwares are written for each new device. This is counter point

[11]Application Specific Integrated Circuits

to the development of modern operating systems that perform exactly the task of abstracting hardwares from the softwares that run 'on' that operating system. Indeed, wherever physical computing or control is necessary, operating systems become largely moot: it is exactly the tight coupling between physical resource and software operation that operating systems eliminate, that is necessary in these applications.

However, microcontrollers contain only small amounts of computing power, on the order of MHz of clock speed and MBs of memory. This is often enough to perform the computing necessary for control of a small system: an exemplary motor controller developed recently at MIT closes two current control loops at $4.5kHz$, sampling currents at $40kHz$ (using an ADC peripheral), as well as a position encoder at $4.5kHz$ (using an SPI peripheral). Loops are kept deterministic using a timing periperal and microprocessor interrupt routines [23]. Firmwares like this, that receive bounded sets of inputs and outputs, are static after compile time, and only serve limited interfaces (Katz lists 5 interface values). They are almost exclusively written in low level languages like C and C++ (often including snippets of assembly), that hardly serve novice-friendly programming environments. Knowledge of their APIs must be known by system intergrators prior to their arrival during system integration. Firmware reconfiguration often involves rebuilding and reloading firmwares: because each micorcontroller is different, this also means that systems integrators must deploy a new build environment for each device in their system.

Microcontrollers constitute the low-level computing elements in digital fabrication workflows. On the other end, we find CAD and CAM softwares that are some of the most demanding for desktop computing, requiring multiple GHz of clock and tens of GBs of memory. These vastly different computing regimes are all present in the heterogenous collections of computing elements that comprise digital fabrication workflows, and it is in developing a systems representation that spans multiple instances of each of these components where this thesis takes shape.

1.3 State of the Art and Related Work

Before addressing the contribution that this thesis makes towards the development of a computing architecture that spans this divide, I should first survey the state of the art in machine control, discuss efforts that have been made to improve it, as well as survey background for the method that this thesis implements.

1.3.1 Monolithic Control

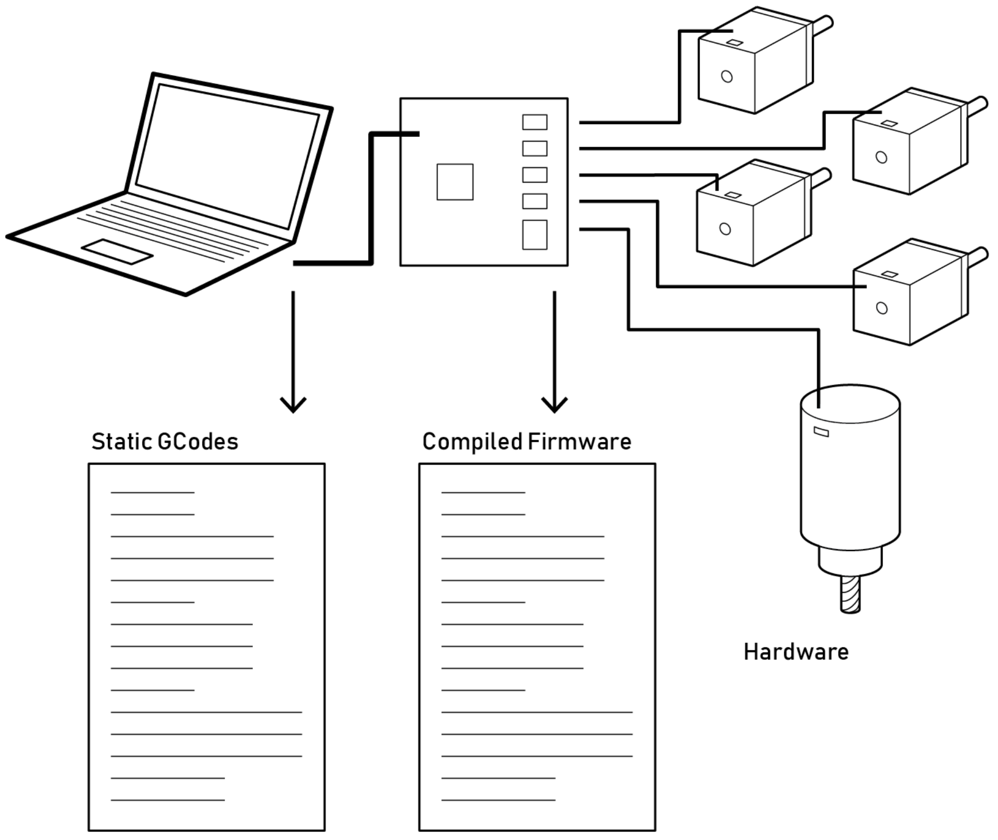


Figure 1-5: Monolithic, GCode interpreting controllers are deployed with static connections between output devices (motors, sensors) and control electronics. Connections between components are not typically data links, but are hard-wired electrical systems: monolithic controllers embed physical device drivers, and motor currents are directly connected to them. This means that adding new or different motors, or new sensors and actuators, involves re-designing the controller's single circuit board.

By monolithic, I refer to control systems where all of the hardware outputs required to operate some system are included with a singular computing device (probably one microcontroller) that is used to operate that hardware. The vast majority of commercially accessible digital fabrication tools are controlled under the system architecture described in Figure 1-5. Under this paradigm, a singular MCU is responsible for executing control of a machine. For example, I include the *smoothieboard* in figure 1-6 [41], an openly available controller that can be used for traditional cartesian machines. To get a sense for the family of these controllers, I also include the *RAMBo* controller in Figure 1-7a, and the *Duet* in Figure 1-7b.

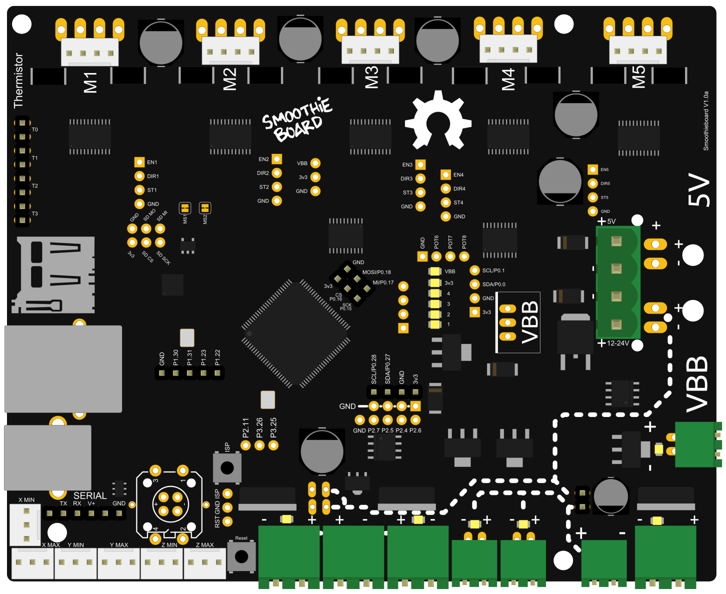
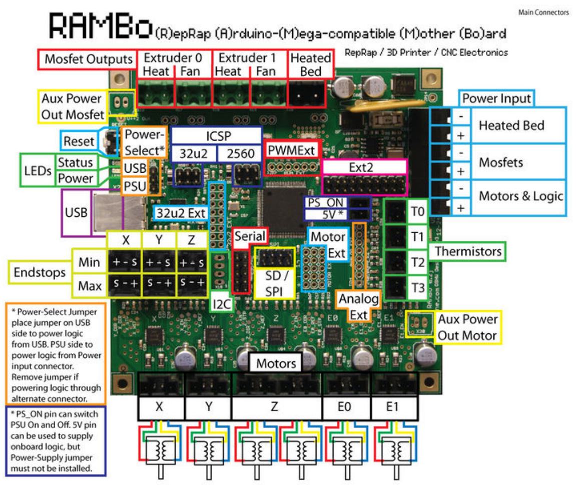
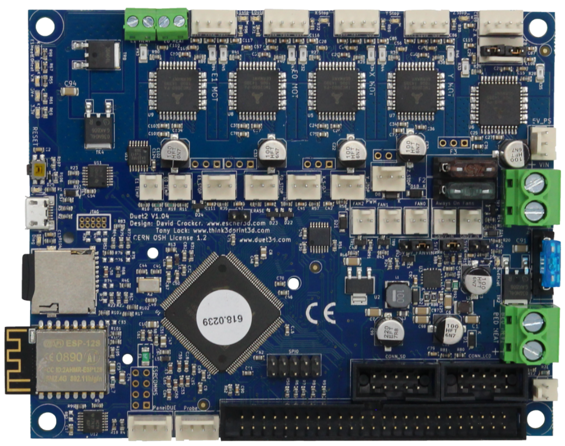


Figure 1-6: The smoothieboard controller, from [41].

As I discussed, microcontrollers have a limited set of physical inputs and outputs, or pins, as well as a finite set of peripherals with which to manipulate them. This means that any microcontroller has an absolute limit not only to the amount of computing it can perform, but to the number of motors, sensors and other devices that it can operate. Many existing controllers, including these listed, cannot operate more than five motors. In addition, the type of motors, sensors and other actuators cannot be modified without modifying and re-designing the circuit boards which host MCUs and other power electronic equipment. For example, replacing a stepper motor in any of these systems with brushless servo motors - conceptually a simple task -


RAMBo (R)epRap (A)rduino-(M)ega-compatible (M)other (Bo)ard
Main Connectors
RepRap / 3D Printer / CNC Electronics
Mosfet Outputs
Extruder 0 Heat Fan
Extruder 1 Heat Fan
Heated Bed
Aux Power Out Mosfet
Power Input
Heated Bed
Mosfets
Motors & Logic
Reset
Power-Select*
ICSP 32u2 2560
PWMExt
Ext2
LEDs
Status
Power
USB
PSU
32u2 Ext
PS_ON 5V*
T0
T1
T2
T3
Thermistors
X Y Z
Serial
Motor Ext
Endstops
Min
Max
SD / SPI
Analog Ext
I2C
Aux Power Out Motor
Motors
X Y Z E0 E1
* Power-Select Jumper place jumper on USB side to power logic from USB. PSU side to power logic from Power input connector. Remove jumper if powering logic through alternate connector.
* PS_ON pin can switch PSU On and Off. 5V pin can be used to supply onboard logic, but Power-Supply jumper must not be installed.

(a) The RAMBo control board, from [35].



(b) The duet control board, from [39].

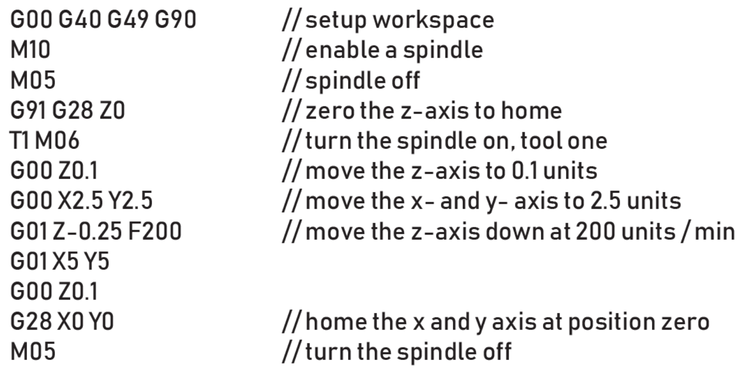
Figure 1-7: A survey of three openly available machine controllers. Each are monolithic, meaning that each has a finite limit to the number and type of motors, sensors, or other actuators that they can control.

is impossible without re-developing not only the firmware for the controller, but also the circuit itself.

1.3.2 GCode, KRL and RAPID

The most common interface between low-level machine controllers and upstream workflows is G Code. G Codes are ordered, sequential lists of steps that are meant to be interpreted faithfully by low level controllers. The specification for G Code was originally drawn up in the 1950s, and used to process instructions for the world's first computer numerically controlled machine. A snippet of gcode is included in figure 1-8.

While the language was developed as a standard for use across any manufacturing machine (formally, G Code is RS-274-D or ISO 6983), many conflicting versions and specifications exist: real success in standardizing G Code would imply standardization of machines themselves, but manufacturing is naturally filled with heterogeneity: machines are each unique in the number of axis they have, types of feed (rotary, linear), and processes they use (turning, milling, or 3D printing machines each use G Codes). To cope with this heterogeneity, equipment vendors extend G Code to their



```
G00 G40 G49 G90        //setup workspace
M10                    //enable a spindle
M05                    //spindle off
G91 G28 Z0             //zero the z-axis to home
T1 M06                 //turn the spindle on, tool one
G00 Z0.1               //move the z-axis to 0.1 units
G00 X2.5 Y2.5          //move the x- and y- axis to 2.5 units
G01 Z-0.25 F200        //move the z-axis down at 200 units / min
G01 X5 Y5
G00 Z0.1
G28 X0 Y0              //home the x and y axis at position zero
M05                    //turn the spindle off
```

Figure 1-8: A snippet of gcode.

own liking, and attempts to standardize between vendors are rare. Since G Code fails in universality, CAM softwares must 'post-process' the plans they generate into vendor-specific 'flavors' of the gcode language: these flavors must be known ahead of time by particular CAM softwares. Additionally, G Code interpreters must be configured per-machine, such that, for example, the 'G0 X' commands are properly routed to operate the motor connected to a particular machines' actual X-axis. This configuration is typically baked-in to machine controllers, and is not possible to change without recompiling control software and reloading it onto physical hardware. This is also true of machines' unique functions: in order to interface to non-standard hardware, new codes must be implemented on the machine-side as well as the CAM-side, and the function of those codes must be communicated to the machine user. While this is not exceptionally troublesome for industrial machine developers, it does make the development of ad-hoc machinery or non standard equipments more difficult than is necessary.

G Codes are static: the plans they encode cannot be modified after they arrive at the machine. This means that real trouble arises when users look to integrate any kind of computing in the processes they are describing: even evaluating conditional statements is not possible. Incorporating more advanced computing into the G Code reprsentation, like Computer Vision or data-based learning is not possible. Finally, G Codes cannot be used to describe processes that span more than one machine: developing workflows - not singular process tasks - is not possible with this representation.

Kuka's Kuka Robot Language (KRL) and ABB's RAPID are both proprietary programming languages developed as an interface to robotic arms used in industrial settings. KRL and RAPID look and feel similar to G Codes, in that they contain sequential movements that robots execute, though they are also turing complete: both can evaluate if/else statements and branch operation. However, neither KRL or RAPID allow access to low level motor control parameters or configurations. Rather, they are better understood as APIs for those controllers' interfaces, that ultimately serve joint angles and position-targeting moves much like G Code.

I consider G Code, KRL and RAPID to be leaky abstractions in two senses. First, they assume correct configuration of the machines they are sent to: that the correct motor output is identified as the 'x' axis (etc), that the machine is in an appropriate coordinate system when it begins executing the code, and that actuator-space control (of rotary positions) are correctly mapped to program-space control (of linear positions, typically). This becomes especially cumbersome when nonlinear mappings between these spaces exist, as is the case with any rotary axis, or novel motion systems like corexy [29]. In these cases, actual machine operation is highly obscured. In another sense, G Codes can rarely be faithfully interpreted: they specify linear process speeds but ignore the fact that machines must accelerate into and out of corners: changes in speed or direction are not instantaneous, as a faithful reading of G Code would imply. This means that even though some processes may be 'specified' to occur at a particular feedrate, they are rarely actually executed at these rates [27]. Indeed, between G Code and actual control outcomes, rather long lists of control routines are followed, but are rarely made visible to machine operators [25].

That G Codes define control outcomes in terms of position and time mean that they prevent other representations from being used in process development. I.E. rather than specifying a weld take place at a certain speed, applications may want to specify that a specific surface temperature be reached along each point in the path, and then close a loop around speed in order to control for this. For processes like cutting and folding, we may not want to specify tooling cut depths and feed rates, but instead specify particular pressures be applied during a portion of motion - this would require active control over the torque one axis is delivering, while the others are set to faithfully move along a path. In the case of milling, we may seek only to specify the cutting force a tool should not exceed, and have the controller optimize against its own physical limits to try to meet but not exceed this bound during machining.

We can consider the difference between a human machinist manufacturing a part on a hand-operated mill, to a computer numerically controlled mill. The human is able to actively adapt their machining strategy during the cutting process: they can listen to motor loads and vibrations, observe surface finish, and accordingly adjust their cutting depth, forward velocity, and spindle velocity as they machine the part. A computer numerically controlled machine that is interpreting a G Code file only knows where it is pre-programmed to move and at what velocity: this is static control *sequence.* A control language does not exist that allows an operator of a digital manufacturing tool to describe an intelligent control *policy* that the machine should operate within.

1.3.3 Object Oriented Hardware

One strategy for machine tool reconfigurability is presented in Nadya Peek's PhD thesis [33]. Peek develops a machine building paradigm where machines are collections of instantiations of modular, coupled hardware/software objects. These are controlled over a networking bus with a centralized coordination tool, a *virtual* controller, written in a high level language. Ilan Moyer's thesis [30] outlines the systems' implementation: a library written in Python allowed virtual hardware objects to be created and coordinated as software objects. Software objects were paired to hardware twins via an RS-485 Serial Bus, themselves running firmware written in C. A schematic of the controller's physical architecture is shown in figure 1-9 from [30, 33].

Peek and Moyer's work represents a step away from the monolithic controllers still found in state-of-the-art equipment, demonstrating the viability of a distributed

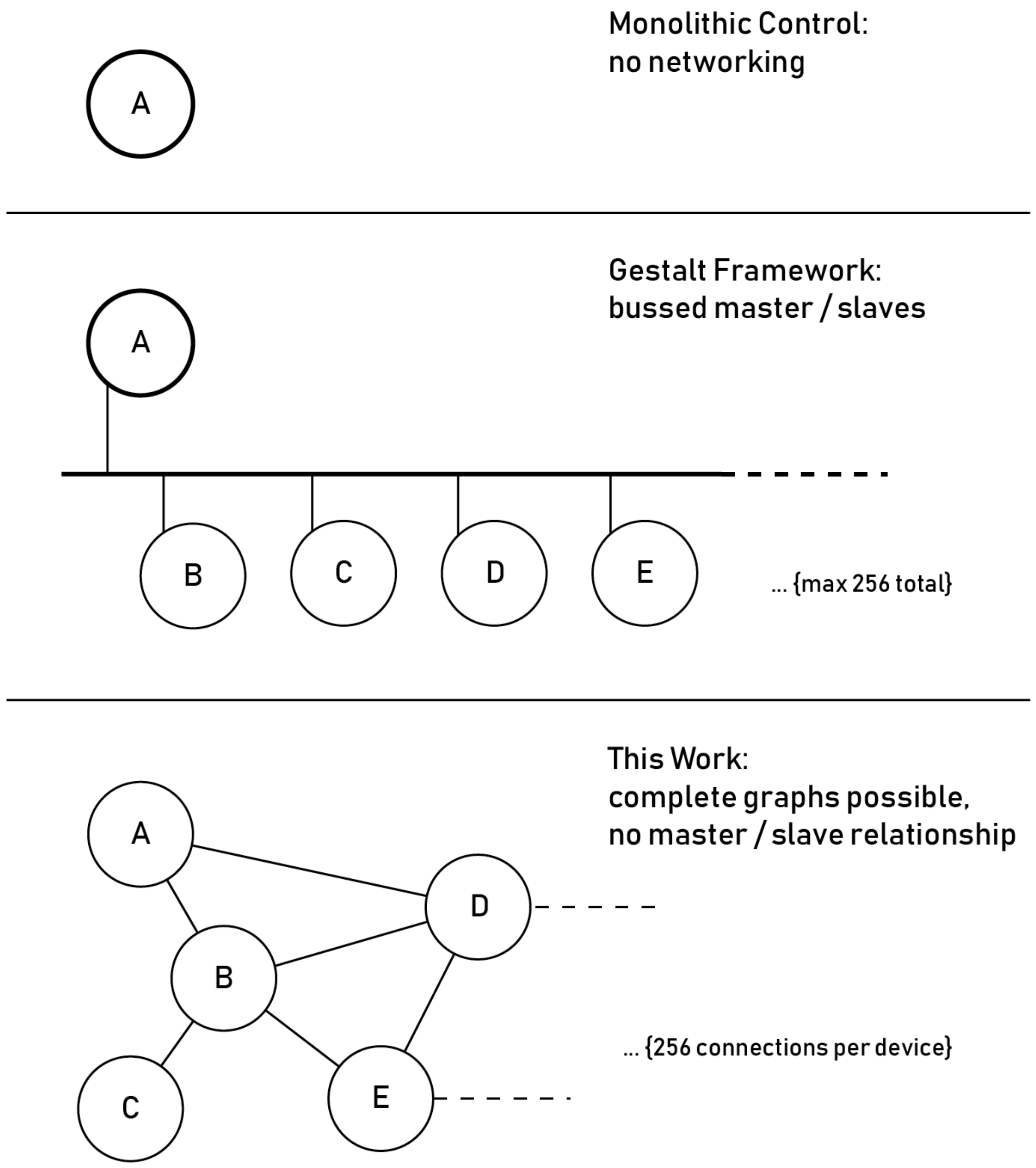
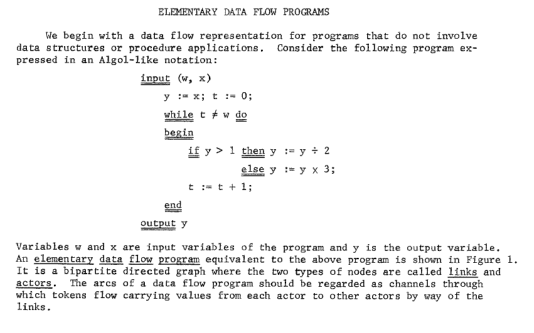


Figure 1-9: Comparing topologies of static controllers, the gestalt framework, and this thesis.

machine control architecture and the use of 'virtual' controllers. However, it retained a static language between high- and low-level systems. That is, function calls available on the remote hardware objects were opaque to system assemblers as they built virtual controllers. Actual operation of low-level devices could not be readily edited or modified without recompiling firmware. Loops could be closed at a high level, but not within the context of low-level computing, nor between two low-level controllers. In section 1.3.8, I explain similar limits found in commercial modular hardware reconfiguration tools like LabView and Simulink, and begin to show how the work in this thesis is set apart from these approaches.

1.3.4 Dataflow

Understanding the Dataflow computing paradigm is essential for understanding the work in this thesis. Pioneered by Jack Dennis at MIT in the 1960's, dataflow stands in contrast to programming languages developed for Von Neumann or Harvard architectures that suppose all data is immediately accessible; dataflow computing is conceived in a complete spatial reality. A dataflow computer does not read and write to random access memory, rather, data flows through a graph and is operated on along the way. Nodes in the graph are compute units, performing operations on data that arrive at the node via graph edges. Programs are assembled by describing the topologies of these graphs. An examplary dataflow program, from [12], is rendered in figure 1-10.



ELEMENTARY DATA FLOW PROGRAMS

We begin with a data flow representation for programs that do not involve data structures or procedure applications. Consider the following program expressed in an Algol-like notation:

```
input (w, x)
    y := x; t := 0;
    while t ≠ w do
    begin
        if y > 1 then y := y ÷ 2
                 else y := y x 3;
        t := t + 1;
    end
output y
```

Variables w and x are input variables of the program and y is the output variable. An elementary data flow program equivalent to the above program is shown in Figure 1. It is a bipartite directed graph where the two types of nodes are called links and actors. The arcs of a data flow program should be regarded as channels through which tokens flow carrying values from each actor to other actors by way of the links.

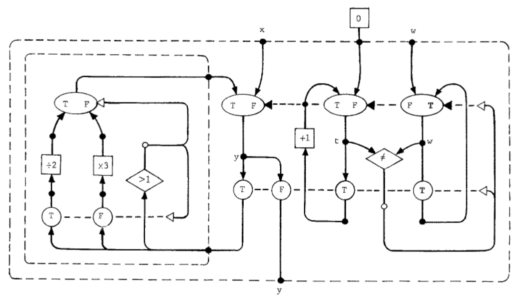


Figure 1. An elementary data flow program.

Figure 1-10: Elementary Data Flow Program, from [12]

This thesis proposes that dataflow is a valuable tool for the organization of machine controllers for a number of reasons. First, it can successfully represent the heterogeneous, parallel computing resources that make up a modular machine controller like the one presented in Peek and Moyer's work - including a representation of the co-ordinating network itself. Of course, dataflow was imagined with this kind of parallelism in mind [13]. Second, the dataflow conception of program execution holds a strong parallel to typical representations of control systems, a likeness that we hope will make the implementation and understanding of feedback controllers within Dataflow Machine Controllers straightfoward and elegant for novices and experts alike. It might also be worth noticing a close parallel between diagrams presented in figures 1-3 and 1-4, that describe a microcontroller's internal operation, and the figure 1-10 here.

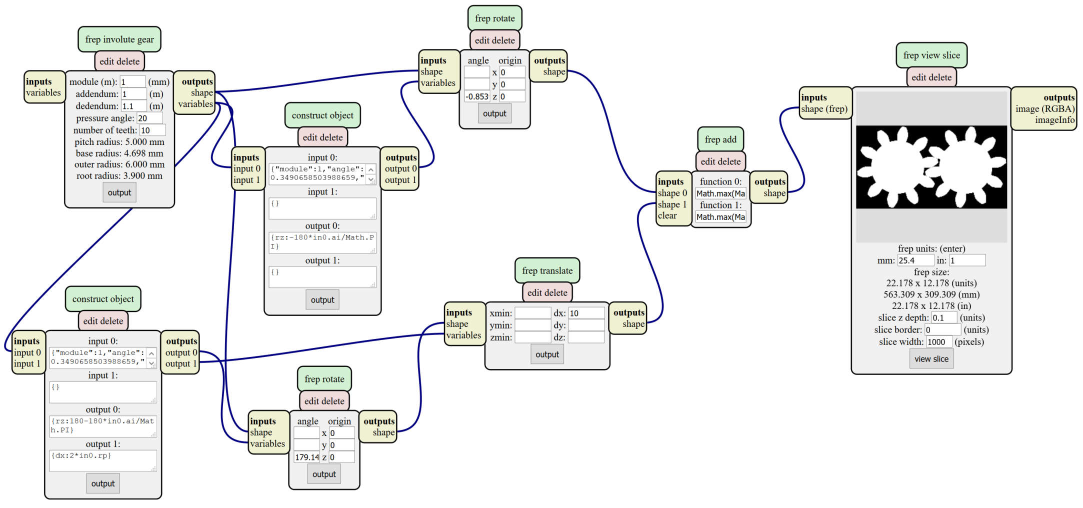
Finally, because dataflow is itself a spatial conception of computing, it lends itself wonderfully to graphical representations and manipulations - developing dataflow structures and programs is a task that can be undertaken by individuals with little or no programming experience. Many examples exist of dataflow softwares that succesfully expose computational tools to novices such as PureData [34], LabView [21] and Rhino Grasshopper [9].

1.3.5 Mods: Dataflow for CAD / CAM Workflows

Gershenfeld [18] has developed a dataflow programming framework for assembly of digital fabrication's software workflows. The *mods* project combines user interface with computing in the browser. A screenshot from the program is rendered in figure 1-11. Mods allows users to program new workflows, and use existing workflows, by arranging and interacting with modular blocks of code. Mods shows that many high level computation tasks for the control of digital fabrication equipment, namely interface, CAM, and even some CAD, can be developed in a dataflow language.

1.3.6 APA: Scalable Networking

It is also important to mention Dr. Gershenfeld's related work on a source-routed port-forwarding network protocol with bit-level flow control via a token-passing arrangement, published [17] and [15]. The APA project represents an early attempt to develop an 'internet-0', a networking scheme suitable for implementation on sys-


frep involute gear
module (m): 1 (mm)
addendum: 1 (m)
dedendum: 1.1 (m)
pressure angle: 20
number of teeth: 10
pitch radius: 5.000 mm
base radius: 4.698 mm
outer radius: 6.000 mm
root radius: 3.900 mm

construct object
input 0: {"module":1,"angle":0.3490658503988659,"
input 1: {}
output 0: {rz:-180*in0.ai/Math.PI}
output 1: {}

frep rotate
angle: -0.853
origin: x 0, y 0, z 0

construct object
input 0: {"module":1,"angle":0.3490658503988659,"
input 1: {}
output 0: {rz:180-180*in0.ai/Math.PI}
output 1: {dx:2*in0.rp}

frep rotate
angle: 179.14
origin: x 0, y 0, z 0

frep translate
xmin:
ymin:
zmin:
dx: 10
dy:
dz:

frep add
function 0: Math.max(Ma
function 1: Math.max(Ma

frep view slice
frep units: (enter)
mm: 25.4 in: 1
frep size:
22.178 x 12.178 (units)
563.309 x 309.309 (mm)
22.178 x 12.178 (in)
slice z depth: 0.1 (units)
slice border: 0 (units)
slice width: 1000 (pixels)

Figure 1-11: A mods workflow, from [18]

tems with typically small computing resources, i.e. microcontrollers. The networking scheme developed in this thesis takes inspiration from APA's straightforward and lightweight port-forwarding, source-routed structure.

1.3.7 Industrial Networks

The field of Industrial Networking would be well characterized as an unresolved competition for standardization across vendors of automation equipment [31]. Competing technologies are also mixed across the OSI Networking Model [10], which defines layers from the Physical, Data Link, Network, Transport, through to Session, Presentation, and Application of networked systems.

All of these technologies define some component of the OSI model's lower levels. For example, CAN Bus (developed in the Automotive industry) defines standards for the Physical through Transport layers of the model. Many application layer approaches have been constructed on top of the CAN lower level protocols: up to *twenty* different approaches exist, with variants seemingly emerging within industries: *CANopen* and *DeviceNET* in industrial automation, *ISOBUS* in agriculture, *MilCAN* in military vehicles, even *CSP*, or CubeSatProtocol exists, and General Motors has forked their protocol into the *GMLAN* branch. CAN is favoured in some contexts for its simple implementation and bussed architecture, and is commonly used for larger networks of smaller devices, like peripheral heaters, coolers, windows, lights etc in an automobile.

Determinism is an important metric for success in industrial networking, as control loops across individual nodes need to be closed under particular time limits. For this reason, many early industrial network technologies relied on new *fieldbusses*, custom hardwares (defining physical through transport layers) often built on top of RS-485 signaling. These were often paired with proprietary application layer systems operating on PLCs. More recently, these technologies are being replaced with networking systems built on top of switched ethernet. Using switched ethernet can bring determinism, low-cost, and interoperability to industrial networks [28]. EtherCAT is an exception worth mentioning, that also adds componentry to the Data Link layer of otherwise standard Ethernet hardwares.

Each of the modern Ethernet-based industrial automation standards: EtherCAT, ProfiNET, EtherNet/IP, and Sercos III (to name a fiew) are still proprietary systems, and define varying levels of the remainder of the OSI model [14]. Because so many of these protocols exist, industrial control networks themselves are often heterogenous, as is diagrammed in figure 1-12 from [31]. While lower levels of these networks - motor and robot controllers - require strong determinism, and so employ technologies like fieldbusses and Ether-CAT, upper levels of these networks - human machine interfaces, enterprise resourse management systems, etc - are simply built with standard internetworking devices on TCP/IP links.

While the work in this thesis proposes a strategy for industrial networking, it resides entirely in the Application Layer of the OSI model, meaning that it does not define any particulars of the physical data links, transport, or routing layers of networking. Instead, networking in this thesis is a kind of virtual networking, meaning that the dataflow controllers I develop here could be applied to work across any of the networking technologies mentioned in this section, or across any ad-hoc or custom networking devices, or any other standard technology like WiFi, Bluetooth, etc. In

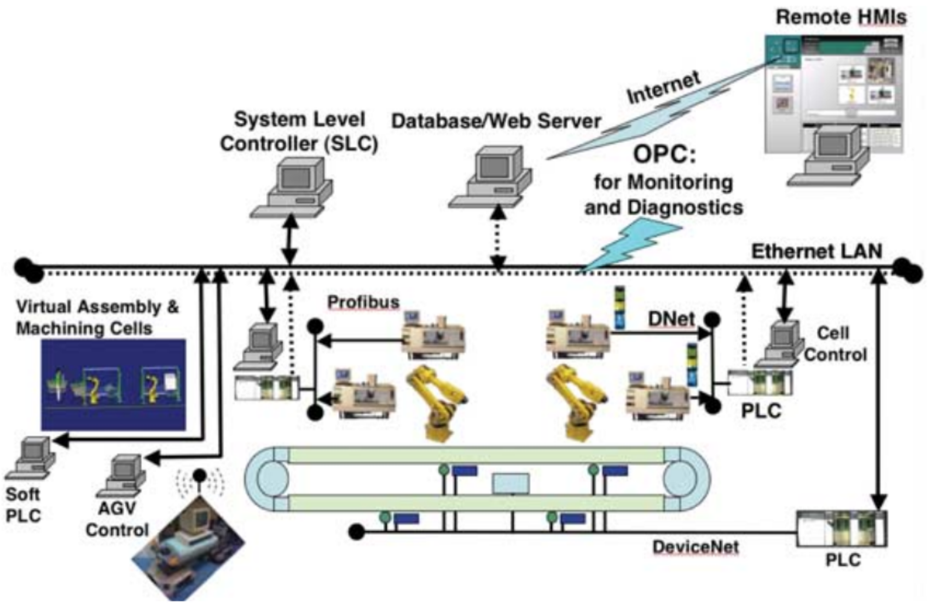


Figure 1-12: Diagram of a highly heterogenous network control system, from [31].

section 3.1 I will explain how this is accomplished with an approach to interoperability that is designed to separate the architecture from a particular relience on physically heterogeneous connections, connectors, and data links. I will also explain in section 4.1 how this approach to system configuration differs from many industrial standards in its configuration-sourcing (as opposed to configuration-imposing) strategy.

1.3.8 LabView and Simulink

LabView [4], a dataflow programming tool for instrumentation and process equipment, and Simulink [8], a dataflow modelling software that integrates with the MATLAB computing language, both integrate dataflow with hardware elements. The work presented in this thesis differs from both systems in a similar fashion, which I will discuss here.

The major departure that this thesis makes from these systems is in decentralized control. In order to operate remote LabView or Simulink hardware, connections between the devices in use and the computer running these softwares must be made directly. That is, data flows only between each remote device and the LabView / Simulink controller - flows cannot be routed between remote devices themselves without first passing through the centralized software. Additionally, no dataflow computation happens on these remote hardwares, meaning that any user-configured computation is centralized as well. This is important for high-speed controllers, or large systems, where routing all data and performing all computation at some central point bottlenecks speed.

In chapter 2, I will explain how the architecture outlined in this thesis actually embeds a dataflow model within each remote environment, meaning that modular software elements can be instantiated on remote devices themselves, and in chapter 3 I will explain how these remote dataflow elements can pass messages to one another without any data routing taking place between some central location and other remote devices. For now, figure 1-13 is here to diagram the structural difference.

LabView does not allow any remote reconfiguration of hardware devices: all calls made to these objects are precompiled or predefined by device developers, and codes are rarely open source. In order to add new hardwares to the system, device drivers must first be downloaded and installed on either LabView or Simulink centers. The architecture presented here allows new devices to be added to a network without any prior knowledge of their internal configurations or capaibilities: instead, devices themselves carry with them an embedded dataflow environment that is capable of reporting its current configuration, and provides handles for further development of its internal graph from a graphical tool. I will explain this in more detail in section 4.1.

Simulink does better than MatLab to integrate open source electronics, including Arduino [3] and Raspberry Pi [26]. Indeed, the software even allows native embedded code to be developed within the Simulink environment, compiled, and loaded onto these remote resources. However, additional libraries to do so must first be downloaded and configured in the Simulink environment. These codes are static at runtime, and their internal operation's native representation is pure C or C++ code, not dataflow like the rest of Simulink.

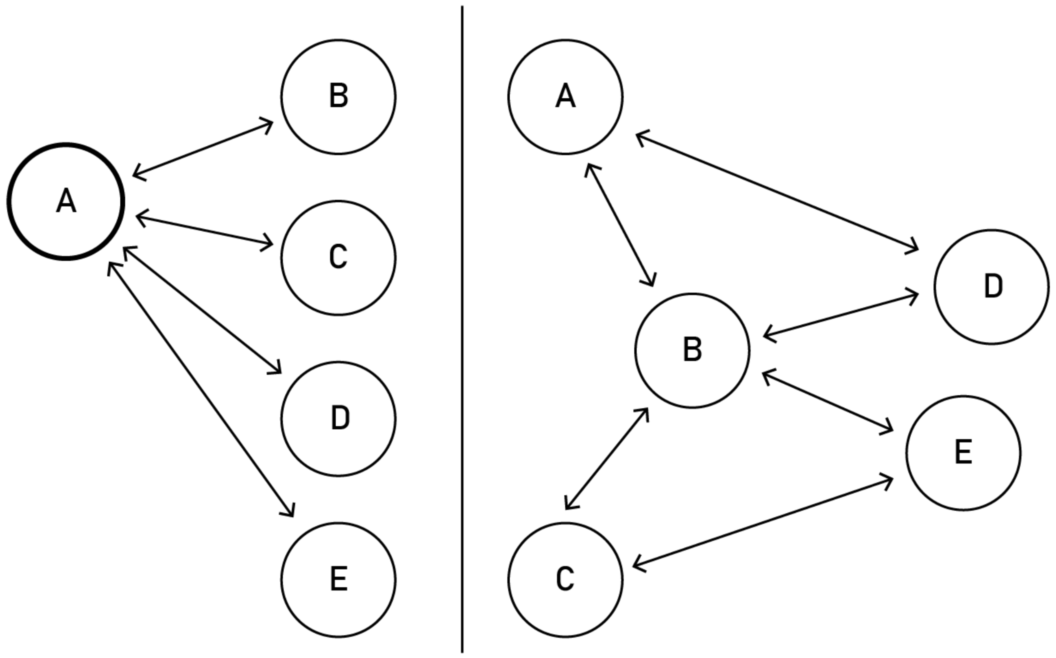


Figure 1-13: LabView or Simulink both run centralized virtual controllers. Connections made to these softwares with remote hardware are always made in the star topology on the left, and all coordination between remote elements is done from within the central node. The work presented in this thesis allows control flows to exist between any node in the network, meaning that systems developed can be genuinely decentralized.

Indeed, any Simulink dataflow program is compiled and then executed: online editing of controllers is not possible. The architecture presented here merges dataflow execution with dataflow editing, to allow for online reconfiguration.

Most importantly, network operation in Labview and Simulink is obscured to the user of these softwares, rather than integral to their operation. Neither softwares overlay network representation with dataflow representation, which is a core tenet of the work contained in this thesis.

A non-technical point, but of some interest, is that these softwares are not open source, and so the addition of new devices into their hardware libraries requires arbitration with the organizations that develop the softwares. The approach taken in this thesis is open, and given that all configuration of networked devices is done by interfacing with the devices themselves and not any central daemon, any new devices that adhere to the protocols involved can ostensibly be added to the edges of a network. This is the route to growth that early internet architects took.

1.3.9 Embedded Computing

Early in this chapter, I discussed the difficulties faced in bringing abstract computing models to use with microcontrollers: their particularities, namely, their special function registers and peripherals each being unique to one another, makes code standardization across microcontroller platforms difficult. Since an attempt to render these complex, specialized codes into a high-level description is at the heart of this thesis, I should survey other efforts to develop programming models or languages for these devices, that are meant to serve a similar purpose. That is, to render their complexities in a manner that can be approached by novices, or a manner that can be rapidly reconfigured by experts.

The world of microcontrollers, embedded computing and circuit commissioning is commonly known as a dark, gruesome abyss relegated to 'the best of us' - iron clad engineers decked with 2000 page datasheets, an apparently infinite supply of patience, kits of debugging equipment, good luck, steady hands, and capital resources. To express the difficulties, we resort to a passage from Neal Stephenson's *Diamond Age* [38].

> Therapies adminstered included but were not limited to: turning things off, then on again; picking them up a couple of inches and then dropping them; turning off nonessential appliances in this and other rooms; removing lids and wiggling circuit boards; extracting small contaminants, such as insects and their egg cases, with nonconducting chopsticks; cable-wiggling; incense-burning; putting folded up pieces of paper beneath table legs; drinking tea and sulking; invoking unseen powers; sending runners to ther rooms, buildings, or precincts with exquisitely calligraphed notes and waiting for them to come back carrying spare parts in dusty, yellowed cardboard boxes; and a similarly diverse suite of troubleshooting techniques in the realm of software.

This was especially true prior to the advent of the Arduino project [2], an 'electronics prototyping platform'. To provide easily understood software abstractions for novices, Arduino wraps common microcontroller peripheral functions in well named and globally accessible libraries (function calls). It includes a friendly development environment and common debugging tools like console logging via a serial port. However, being an abstraction, it obscures some internal function, and imposes significant performance overhead. For example, the *ring test*, a full description of which can be fonud in section 5.3.1 of this thesis[12], has a period of $0.687us$ undertaken with hand-written (specialized) code on a Cortex-M0 processor. With Arduino's libraries, the same test has a period of $6.14us$, one order of magnitude slower. This is the cost of code generalization over a swath of supported microcontrollers.

Arduino does a great service to novices: it encapsulates the complexity of accessing their microcontroller's Special Function Registers in well named and cross-platform software libraries. However, it does so at the expense of the performance that can be achieved precisely by that specialization. Here, the challenge would be to address the specialization necessary for high performance applications (a-la Katz's motor controller from 1.2), but still enable systems to be redeveloped without the encumberances of compiling for various targets, specifiying software libraries for Special Function Registers, etc. In section 4.1, I will discuss how the architecture in this thesis allows me to write specialized codes with native performance on microcontrollers but still assemble modular, well-named codes into application specific programs by manipulating a dataflow representation rather than a software API representation.

Besides abstracting embedded software into a generalized API, Arduino does not easily extend across computing domains: it does not include a model for which the microcontroller's resources can be easily described or interfaced with when it is included as a component in a larger program, i.e. with an external user interface operating on desktop-scale computing. The Firmata firmware and messaging protocol is an attempt to do so [37]. Firmata wraps the Arduino API in a variant of Remote Procedure Calls [32] that can be made within higher-level programming languages, like JavaScript or Processing. Indeed, a wrapper for the dataflow environment PureData

[12]Briefly: the ring test is a measure of encumberances between the processing unit in an embedded device, and its hardware output. Smaller periods, faster oscillations, are tighter couplings, and preferrable results.

is provided. However, the list of calls is limited to only a core set of functions available on the Arduino - setting and reading from physical pins on the microcontroller. Firmata provides no programming model that operates across platforms, rather, it runs blocking code that makes requests to remote resources, waiting for returns before computation proceeds in the higher level language.

1.4 The Contribution of this Thesis

Working across multiple levels of computing and inter-networking, this thesis develops an execution model and descriptive abstraction for dataflow programs that captures the operation of low- and high-level computing resources, as well as the networks that connect them. It does so by first developing a model for virtual dataflow environments: these are effectively lightweight operating systems that can instantiate modular codes, and that connect modular codes with dataflow interfaces to form dataflow programs, discussed in chapter 2. These virtual dataflow environments, or VDEs, are then embedded into inter-networked physical dataflow networks of modular hardwares, and a scheme is developed to extend dataflow operation across multiple VDEs operating together, discussed in chapter 3. Finally, a tool and messaging system is developed that allows the internal graph of each VDE to be made visible and reconfigurable to system configurators, in chapter 4. In chapter 5, I discuss how the tool and architecture was reduced to practice, and address some performance evaluations.

Chapter 6 shows that the architecture can be used to comprise systems whose execution is cohesive and coeherent across heterogeneous computing resources, and with heterogenous function. Only a small set of standardized data type serializations as well as standardized system assembly messages must be developed in order to do so. Rather than the architecture imposing constraints on what resources should be available at any given level, individual VDEs are granted runtime authority over what codes are run within their bounds, and self-describe their internal configurations. This minimizes architectural complexity in favour of endpoint complexity, in an approach that is aligned with the end-to-end principle adopted by early internet architects.

To clarify, the thesis re-casts monolithic controllers that use internally developed task representations (like G Codes) as distributed networks of controllers that use a unified algorithmic representation: dataflow. Software and hardware are both modular, and both represented under the same system representation: dataflow networks. I call these systems Distributed Dataflow Machine Controllers.

Chapter 2

Virtual Dataflow

The next two chapters constitute a description of distributed dataflow controllers that is best taken from the inside out. In this section, I will describe virtual dataflow code *elements*, and the virtual dataflow *environments* they operate within.

2.1 Code Hunks

The smallest unit of assembly in this system is something that I call a *Hunk*. These are modular codes - small programs - that each have a *Dataflow Interface* to the rest of the system. To those familiar with dataflow, these are nodes - operating units - in the graph.

A dataflow interface is a series of virtual, directed ports. Each hunk has a set of *inputs*, from which it can receive data, and a set of *outputs*, to which it can put data. Basic operation occurs when hunks retrieve data from their inputs, operate on those datas, and (depending on flow control conditions diagrammed in section 2.2.1) release new data to their outputs.

Hunks can be written in nearly any computing language; in this implementation of distributed dataflow I have written them in both JavaScript and C++. The system architecture is not interested in their composition as code, it is merely interested in their operation as a dataflow object.

At the center of this approach is the notion that we can reduce most computing

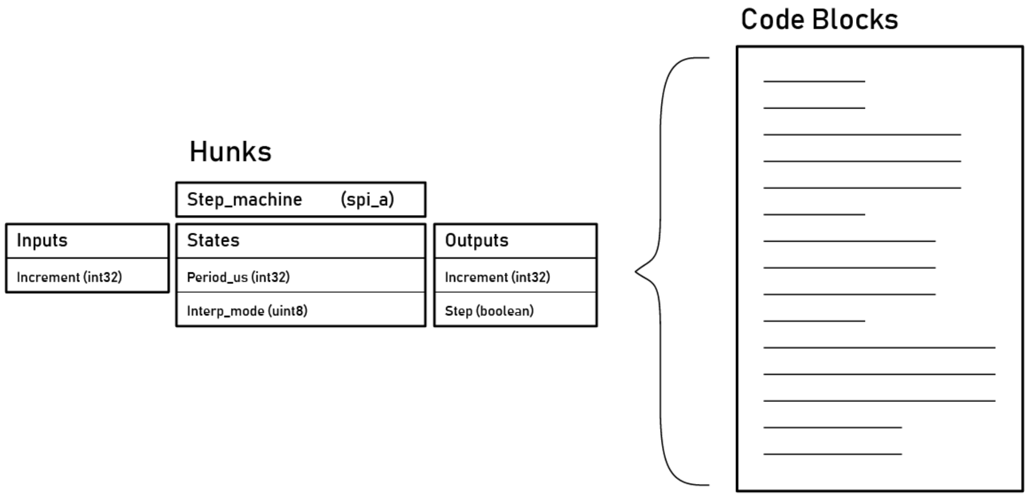


Figure 2-1: Hunks are virtual dataflow wrappers for native code blocks. They have Dataflow Interfaces: Inputs, Outputs, and optional State Variables.

participants to some description of inputs, outputs, and states. This is a natural representation for networked computing devices, as data literally flows through a network during system operation. If this is taken one step inside of the system, and we intentionally represent both our modular codes and modular devices in this manner, we can avoid breaking or changing abstractions as we traverse some heterogenous system's entire configuration.

2.1.1 Example Hunk Codes

To demonstrate how hunks are structured internally, I have included two examples here: one for a hunk written in JavaScript (listing 2.1.1) for the browser, and another written in cpp for an embedded context (listing 2.1.1). Both perform the same function of adding two number together.

Listing 2.1: A hunk composed to add numbers A and B together, under dataflow constraints, written in C++

```
include "hunk_adder.h"
Adder::Adder(){
  // we have some setup to do,
  type_ = "math/adder";
  inputs[0] = a;
  inputs[0] = b;
  numInputs = 2;
  outputs[0] = c;
  numOutputs = 1;
}

void Adder::init(void){
  // init codes can run when the hunk is instantiated
}

void Adder::loop(void){
  // we check if our downstream output is clear,
  // and if our inputs are both occupied
  if(a->io && b->io && !(c->io)){
    c->put(a->get() + b->get());
    } else {
      // otherwise, we exit
    }
  }
```

Listing 2.2: A hunk composed to add numbers A and B together, under dataflow constraints, written in JavaScript

```
import { Hunkify, Input, Output, State } from '../hunks.js'

function Adder(){
Hunkify(this)
let a = new Input('number', 'a', this)
let b = new Input('number', 'b', this)
this.inputs.push(a, b)
let c = new Output('number', 'c', this)

this.init = () => {
  //
}
```

```
  this.loop = () => {
    if(a.io && b.io && !(c.io)){
      c.put(a.get() + b.get())
    }
  }
  }

  export default Adder
```

Casting Von Neumann processes as dataflow objects is on-trend with recent trends towards coarser-grained dataflow approaches[22], where the concept is used largely as an assembly tool for codes that might otherwise be used as software libraries.

2.2 System Execution

To develop programs, graphs of hunks are connected by wiring outputs to inputs. The wires, in this case, are edges in the graph of nodes, and constitute directed flow. It is important to distinguish that these graphs are not 'interpreted' as is sometimes the case in graph programming models. Rather, it is more accurate to say that they *execute* themselves. What I mean is that these are simply network diagrams of modular codes that operate by ferrying data along graph edges: this is a message-passing dataflow model. The codes themselves run internal *loop* functions, akin to a physical systems' runtime loop, where the state of inputs and outputs are checked, and data can be retrieved from inputs, computed on, and output.

2.2.1 Flow Control

To coordinate data flows, each connection between an output and an input represents a stateful wire - or graph edge - it either has data on it, or it does not. This is akin to other token-passing dataflow methods. Hunks have access to the state of these wires via the state of their inputs or outputs. Flow control is performed when hunks use this information to orchestrate their internal operation. For example, recall the

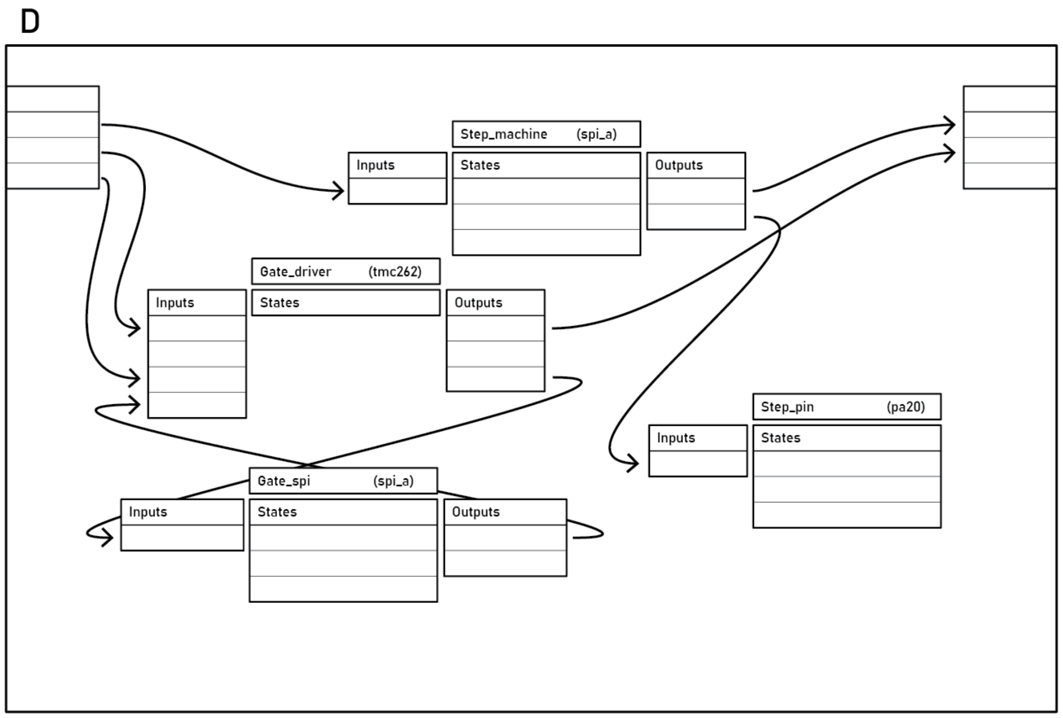


Figure 2-2: A virtual dataflow environment, containing a small collection of Hunks. These are connected to one another with directed links between Outputs and Inputs, and optionally each have State variables. Some collection of Hunks, and their connections, makes up a graph.

example code blocks in listing 2.1.1: during a loop function, the hunk checks to ensure that inputs for A and B are occupied before it attempts to pull data from these wires, operate on them, and output a result to the *clear* output, C.

I picture one such flow controlled sequence in 2-3, where a series of virtual system cycles is explicated to perform the addition of two numbers. Flow control is an excellent tool for our system because it can be used to coordinate execution of the graph subject to logical constraints, and to normalize computation of otherwise asynchronous events. The simple case of addition shows how flow control can be used to ensure that certain computations are performed only when new data is properly available for computation. I will later explain how flow control also becomes an important tool as operation is extended across network links, as it ensures that datas generated in high speed devices does not overrun lower speed devices' abilities to

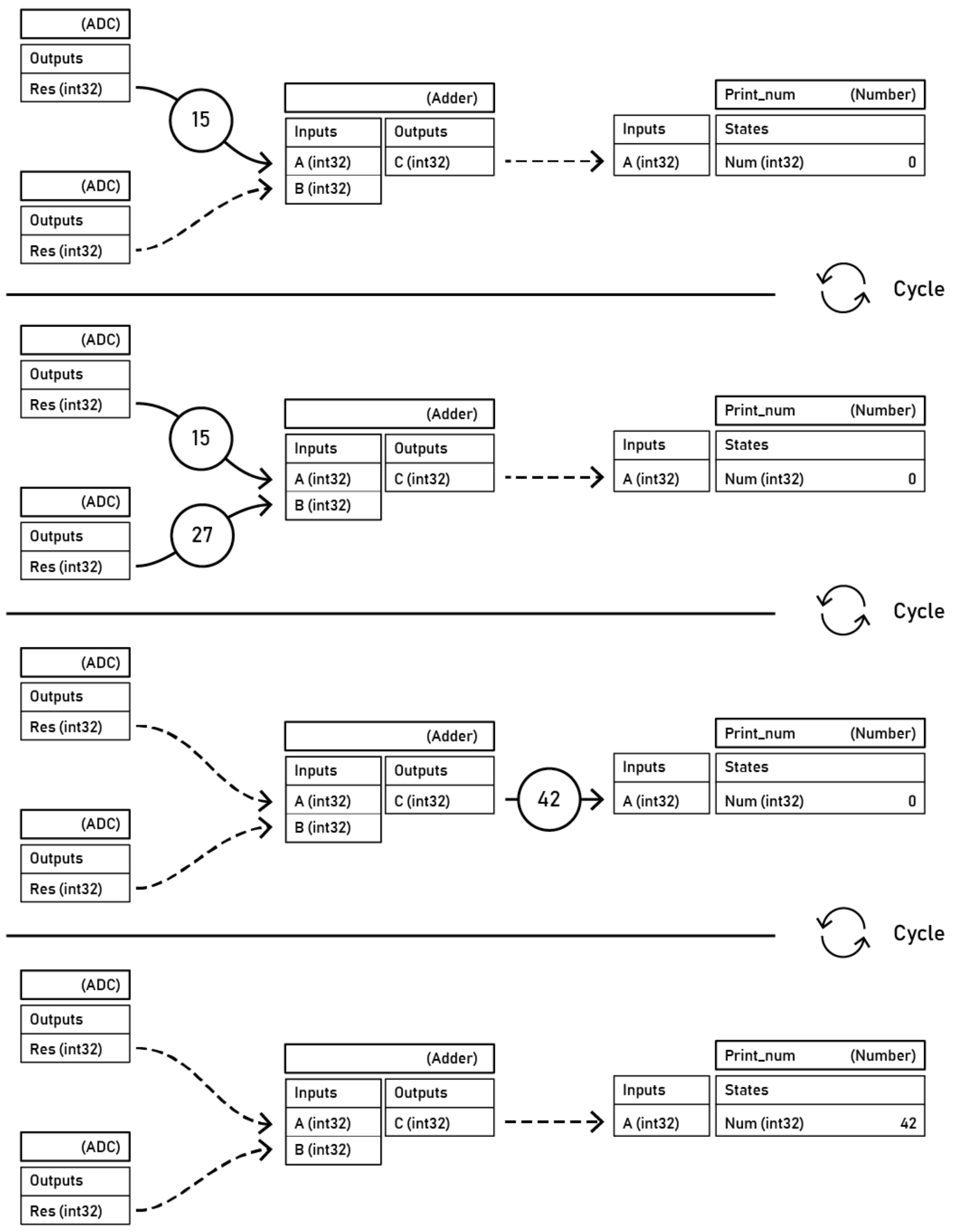


Figure 2-3: The dataflow implementation uses token-passing based flowcontrol. In the code snippet 2.1.1, we see that hunks check whether their inputs and outputs are occupied ('.io' = true meaning that they are-occupied), before operating. In this case, the adder waits for both inputs A and B to be occupied with data, as well as for its output C to be clear, before pulling data from those inputs, thus clearing them, and proceeding to put new data on its output, C.

process those datas.

2.2.2 Virtual Dataflow Environments

Virtual Dataflow Environments, or VDEs, are where Hunks take up residence. VDEs are akin to operating systems; they are programs that apportion a physical computing device's time and memory to multiple smaller programs - in this case, the hunks, and facilitates data transfer and connections between them.

For a VDE to accurately represent the parallel processing we should expect from a dataflow environment, they are operated in system cycles. Within each cycle, VDEs simply traverse each Hunk in their domain, and call their *loop* functions, examples of which are above. It is during these loop functions that data is retrieved from inputs, or delievered to outputs.

VDEs are also responsible for oversight of their local graphs of dataflow elements. VDEs can instantiate new objects, remove them, change state variables, and add and remove wires between inputs and outputs.

2.2.3 Typing

While many modern languages allow automatic and ad-hoc typing to occur, this model for dataflow uses typed inputs, outputs, and state variables. This is critical when we arrive at operation over network links, because data types must be serialized by one physical graph participant, and interpreted by another. Indeed, a common approach to typing and serialization into packet structures is one of the only core tenants across virtual dataflow environments written in different languages used in this thesis, as it is at the network interface where these environments must share common ground, or protocol, in order to interoperate. I diagram the core types used in this implementation in figure 3-2.

While typing can feel cumbersome in langauges like JavaScript, it is only natural in C++ or any computing environment where memory is made spatial, and especially in embedded environents where memory is also limited. Type information is also widely useful for system assembly, as it naturally informs which outputs and inputs can be connected to one another. Besides naming dataflow objects, type data is in fact often more important for system configurators to accurately ascertain how hunks can be connected to one another to form graphs. As a goal of the work here is to smoothly merge high- and low-level computing with network operation, I should aspire to include strategies that allow complex, compound data types to flourish within system architecture. At the moment, this work is allocated for the future, and types used to date are typically one dimensional in nature.

2.2.4 Hunk Polymorphism

To cover wide use without writing excess amounts of code, Hunks can be developed as polymorphic objects; they are given control over their form: their collections of state, input, and output objects, and those objects' types are all mutable, during runtime. This is a useful tool in particular for the development of blocks of code that might produce new numbers of outputs or inputs given the constituents of their inputs, or given new state. In example, the *Link* hunk that will be described in 3.1 can instantiate new inputs and outputs on the fly to develop deeper interfaces into the virtual dataflow environment to which they are connected. In addition, the thesis implements its motion planner's central state machine with polymorphic outputs, modifying motor outputs when its state variable for the number of axis that are to be controlled synchronously in a machine is set. An example of a polymorphic hunk is rendered in figure 2-4.

2.3 The Size of a Hunk

Hunks are the smallest configurable unit in our system, and their size (relative complexity) directly affects the level of modularity and reconfigurability present in the architecture. As we have seen they are simply dataflow interfaces for compiled (or interpreted) computer code. This means that when users assemble collections of hunks into graphs, and graphs into systems, they rely on descriptive identifiers (names:

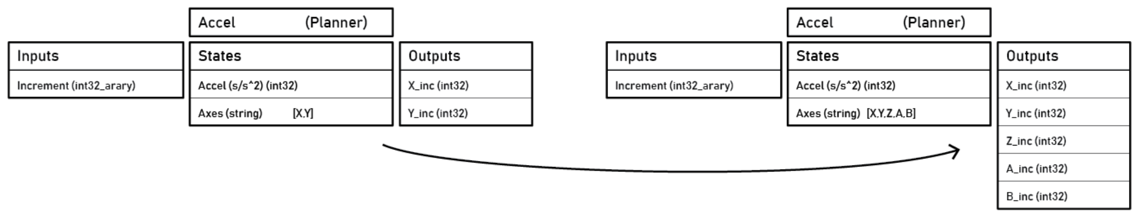


Figure 2-4: A polymorphic hunk: the planner can spawn outputs for new motors as they are added to the system, when its state variable for the number of axis to be controlled synchronously increases or decreases.

strings) in the hunk to determine what their functions will be. The actual compiled code is, in most cases, not possible to edit without further engagement with the source. Instead, hunk function must be inferred by users based on their names, the form of their inputs and outputs, and those objects' types.

That hunks are just code wrappers is a both 'a feature and a bug'. On one hand, it means that developers can easily wrap existing softwares in dataflow structures: I can include existing code libraries with little re-development. I can also easily build VDEs for existing computing environments, with off-the-shelf toolchains. However, because hunks are only descriptive identifiers, and not formal definitions of function, it is not possible to claim that this thesis presents a computing model: the graph itself does not, on its own, determine the action of the graph during operation. Rather, it is a tool for assembly and description of systems.

This means that the design and development of the hunks in any given context is a task that should be undertaken carefully. It also means that our architecture, like the GCodes we are interested in surpassing, is susceptible to making assumptions about what system assemblers might do with the system.

The development of hunks also takes time. At the time of writing, only a limited set are available in the system, as the task to carefully engineer sets of hunks that could be claimed to provide universal computation has not been undertaken.

However, the system still provides immense value during the assembly of heterogeneous systems. Since the execution model is standardized, data still follows predictable paths. Because development tools, like simple console loggers, as well as graphic data outputs, are available in the system, users can easily interrogate the internal function of hunks by instantiating them in highly instrumented graphs. Since codes are developed in the open domain, source can always be referenced when it needs to be. It is my hope my initial work on this computing model will be well structure enough that many participants will be able to collaboratively develop a library of Hunks and VDEs for a plethora of computing environments and devices, consituting a wider platform than I alone could develop.

Chapter 3

Physical Dataflow

No, but even *That* only flickers now briefly across a bit of Slothropian lobe-terrain, and melts into its surface, vanishing.
from *Gravity's Rainbow*, Pynchon

3.1 Execution Across Network Links

I have so far discussed many of the details in this computing model; we have seen how graphs are made up of hunks of code, and how datas are ferried between hunks to make up programs, executed within virtual dataflow environments. I can now describe how these singular computing environments are made to operate across network links, to compose programs that span multiple VDEs embedded in physical dataflow graphs.

3.1.1 Virtual Networking with Links and Data Ports

To remain inclusive of as many computing environments and networking technologies as possible, distributed dataflow in this thesis develops a model of virtual networking that can be implemented across almost any physical networking layer. To abstract, I assume that inter-networking layers are simply devices to which bytes can be written to transmit, and from which bytes that have been received can be read. I also take for granted that these bytes arrive in packets, and depart in packets. The particulars of packet delineation, addressing, and manipulation of the physical medium is outside of the domain of distributed dataflow per se, rather, I include in my VDEs some hunks that are drivers for whichever network technology we would like to use to carry information between the physical nodes in our distributed graph. In this way, we can assemble systems across WiFi, USB, SPI busses, EtherCAT, ProfiNET, or even simpler links like microcontrollers' UART ports.

This is an important distinction to make between the work of this thesis and existing industrial networking technologies mentioned in section 1.3.7. These technologies are largely network technologies that operate between layers one to four of the OSI networking model[10]. The architecture presented here more accurately resides in the Application Layer of the OSI model: it is virtual networking that can be overlaid on any physical model.

I developed a system-core hunk that I call a *Link*. Links form the bridge between VDEs, and perform the task of packaging dataflow messages for transmission, and receiving them from other VDEs. In order to actually transmit these packages, links serialize messages and output them on one of their dataflow ports that is in turn connected to a hunk that operates the physical network driver. This relationship is diagrammed in figure 3-1. We can think of our Links as virtual network ports. They are, to one VDE, the dataflow representation (an object having inputs, outputs, and state), of the interface to some remote VDE. They additionally track flow control conditions across the bridge, only retrieving data from their inputs when those inputs' reciprocal outputs are known to be clear.

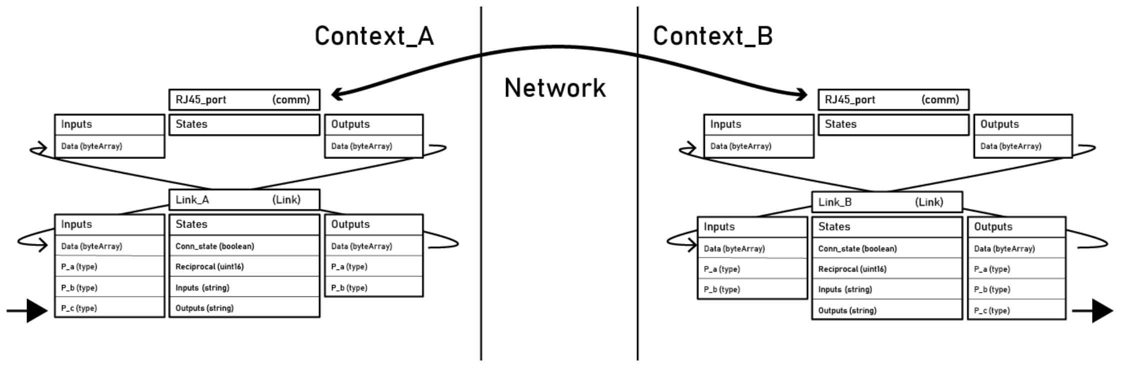


Figure 3-1: The link: these are system-core hunks that package dataflows from one context for their delivery into another context. Inputs in one context comprise the outputs of the link's reciprocal partner in a separate context.

Computationally, to any given VDE, links simply appears as another type of hunk. However, the inputs it presents are indeed reciprocal outputs on the other side of some data port, and its outputs represent some remote link's reciprocal inputs. Order is maintained between links such that input and reciprocal outputs are matched in arrangement and type. To perform the flow control mentioned, they use a packet-level per-port acknowledgement routine.

The link-to-link topology described here additionally imposes no master-slave relationship as is common in many of the industrial networking strategies mentioned in section 1.3.7.

3.1.2 Serialization and Packet Structures

When a link sends a message across its data port, it performs a serialization of the data-type for that given port. It is only here, at serialization, that system wide standard for data interpretation must be set. The thesis implements a custom serialization protocol that is meant to be simple, performative, and expandable into compound data types. Figure 3-2 outlines the core type set, which is strongly aligned against standard data types from C.

In overview, each basic type is given a key-code as a single byte, and this code is appended to the first byte of any data word for that type. While this limits uniquely serializable data types to 255 in length, minus two for link-specific communication, we consider the 64 types implemented a rather complete set of core datas. Types that have variable word lengths (i.e. arrays, strings) have two additional bytes appended to their word, following the key code, to describe the succeeding data's length in terms of byte size. To complete serialization, the link also appends the messages' port number to the head of the packet, i.e an input pulled from one link's 3rd port has a '3' appended to the packet before it is transmitted, such that it will be output on the reciprocal links' 3rd output. This is akin to source-routing in network operation. Some example packets are presented here in table 3-3.

Data Types		Byte-Keys	Length Bytes?
Boolean	True/False Values	32	No
Boolean_Array		33	Yes
Byte	‘Raw’ Datas, Messages	35	No
Byte_Array		36	Yes
Uint[8, 16, 32, 64]	Unsigned Integers	38, 40, 42, 44	No
Uint[8, 16, 32, 64]_Array		39, 41, 43, 45	Yes
Int[8, 16, 32, 64]	Signed Integers	50, 52, 54, 56	No
Int[8, 16, 32, 64]_Array		51, 53, 55, 57	Yes
Float[32, 64]	Floats	60, 62	No
Float[32, 64]_Array		61, 63	Yes
String	Strings	70	Yes
String_Array		71	Yes

Figure 3-2: A list of the core data types for which links can serialize output data, and deserialize input data. These types, their serialization standards, and byte codes (identifiers) are on of the only components of the system that each VDE must match against one another, all other internal representations of data are valid. These closely align with standard C types.

Byte Function	Link Port Number	Type Key	Length Bytes ?	Data Bytes
Byte Count	<1>	<1>	<2>	<n>
Packet Meaning	for output 2	‘uint32’ key	-	65537
Packet Bytes	2	42	-	0, 0, 1, 0
Packet Meaning	for output 4	‘string’ key	5	‘hello’
Packet Bytes	4	70	0, 5	104, 101, 108, 108, 111

Figure 3-3: Packet serialization structures.

3.1.3 Network Routing with Links

I have described how one context can execute alongside another context, by ferring data across any byte-carrying network device via our links. But what about routing messages through multiple layers?

Under this architecture, no explicit routing scheme at a network scale exists. Instead, virtual dataflow programs are made into direct routers by passing messages from one link to another as shown in 3-4, where context D is configured to route duplex messages from context A to contexts B and C, through $Link\_A$, $Link\_B$ and $Link\_C$'s input and output ports.

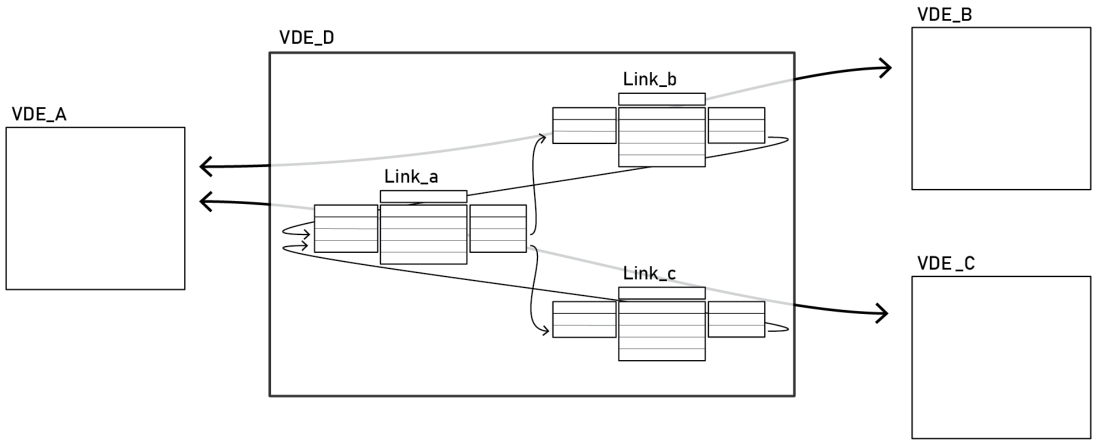


Figure 3-4: Here we see one context's virtual dataflow environment configured to blindly forward messages along from one context, A, onto contexts B and C. Direct message routing is not an explicit system task, rather, routing is developed as a subset of possible graphs.

In this case, a 'routing' layer is not explicit, it is simply a configuration of the globally executed dataflow program. In distributed dataflow, the layers of program configuration and routing configuration are collapsed.

In a more likely case, physical devices that might typically be conceived of as purpose-built 'routers' also serve some computation function. For clarity, an image of one such device that is included nearby in figure 3-5. In example, many tasks that require close coordination between nodes should find the computing resources that undertake that coordination located as close as possible in the graph to the endpoints which they coordinate. For example, when I developed an acceleration planning scheme, I included the acceleration controller closest to the degrees of freedom for which it controls acceleration. In 3-6, I show a layout where context D, previously acting solely to pass messages, is used also to perform some computation on higher level messages originating in context A. Indeed, being able to locate computing on the computing resources where it is most useful is exactly the point. I do not wish to

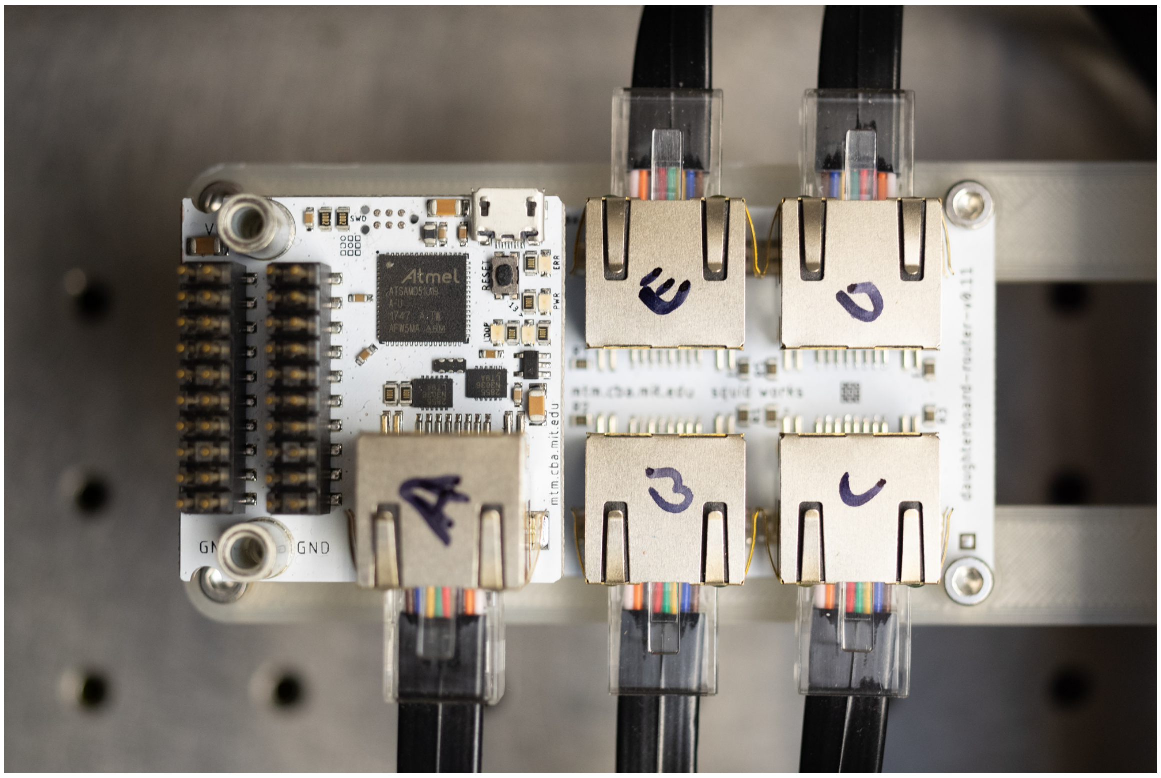


Figure 3-5: An image, here, of the device-that-looks-like-a-router, but is also a tiny configurable-compute-center.

abstract away the network, I want to be able to more easily coordinate systems that span all of its resources, message passing hardware included.

3.2 Networked, Specialized Controllers

Finally, figure 3-7, diagrams one example of a physical system architecture. This represents a typical set of machine control devices connected to one another across point-to-point physical links. Each resource in the system, comprising an interface at B, a message passing device at A, motors $C - D$ and the milling spindle at G, is a specialized dataflow device, made to operate the particular hardware to which it is attached. In this way, we can develop systems whose physical sub-components bring with them the computing and control needed to operate those components. For an example of the particular devices developed for this thesis, see section 5.1.

The physical network in figure 3-7 can be diagrammed with the graph drawn in

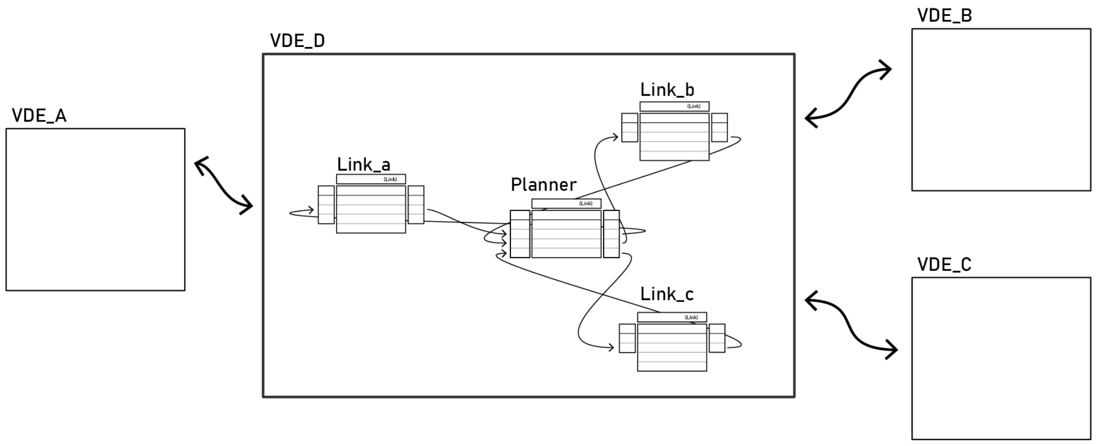


Figure 3-6: Here I show a more likely port forwarding case, as opposed to figure 3-4. Very often, we would like to perform the computation associated with a branch of our graph at the node that splits that branch from the rest of the graph. In example, here I show how some code can be executed in a context that may otherwise simply route messages, offloading some control from the upstream context A, and more tightly coordinating between downstream contexts B and C, due to shorter overall network paths between associated VDEs.

figure **??**, which shows how virtual dataflow graphs are nested into *physical* dataflow nodes. Figure 3-8 depicts how this interlay between nested contexts is handled graphically in a development tool, discussed in the next chapter.

In figure 3-9, I diagram how additional message passing devices and motors can be added to the system to extend a three-axis milling machine controller into a five-axis controller. As the graph can include any tree structure, there is no limit to the number of physical or computing resources that can be added to one cohesive system.

This also means that adding more complex, or different local controllers is possible. For example, figure 3-9 also shows one stepper motor being replaced with a closed-loop subsystem comprising of one servo motor and a linear encoders. While this might normally involve redesigning control boards, and rewriting firmware, we can easily add the resource to the system without modifying other parts of the controller.

While graphs extend into more and more complex nets of control, the architecture never falls short of having enough computing resource to coordinate that control globally. It becomes possible to develop systems that harbor intelligence at low levels:

for example, the control needed to coordinate between the linear encoder and motor on the associated axis in figure 3-9 can be deployed between those two resources alone, and network traffic between the two need not extend beyond their local domain. This is an important distinction to make between the architecture presented in this thesis and earlier work reviewed in section 1.3.3. Here, control is not centralized into any one virtual controller. Rather, networks of virtual controllers path datas to one another, potentially forming deep nets of distributed control.

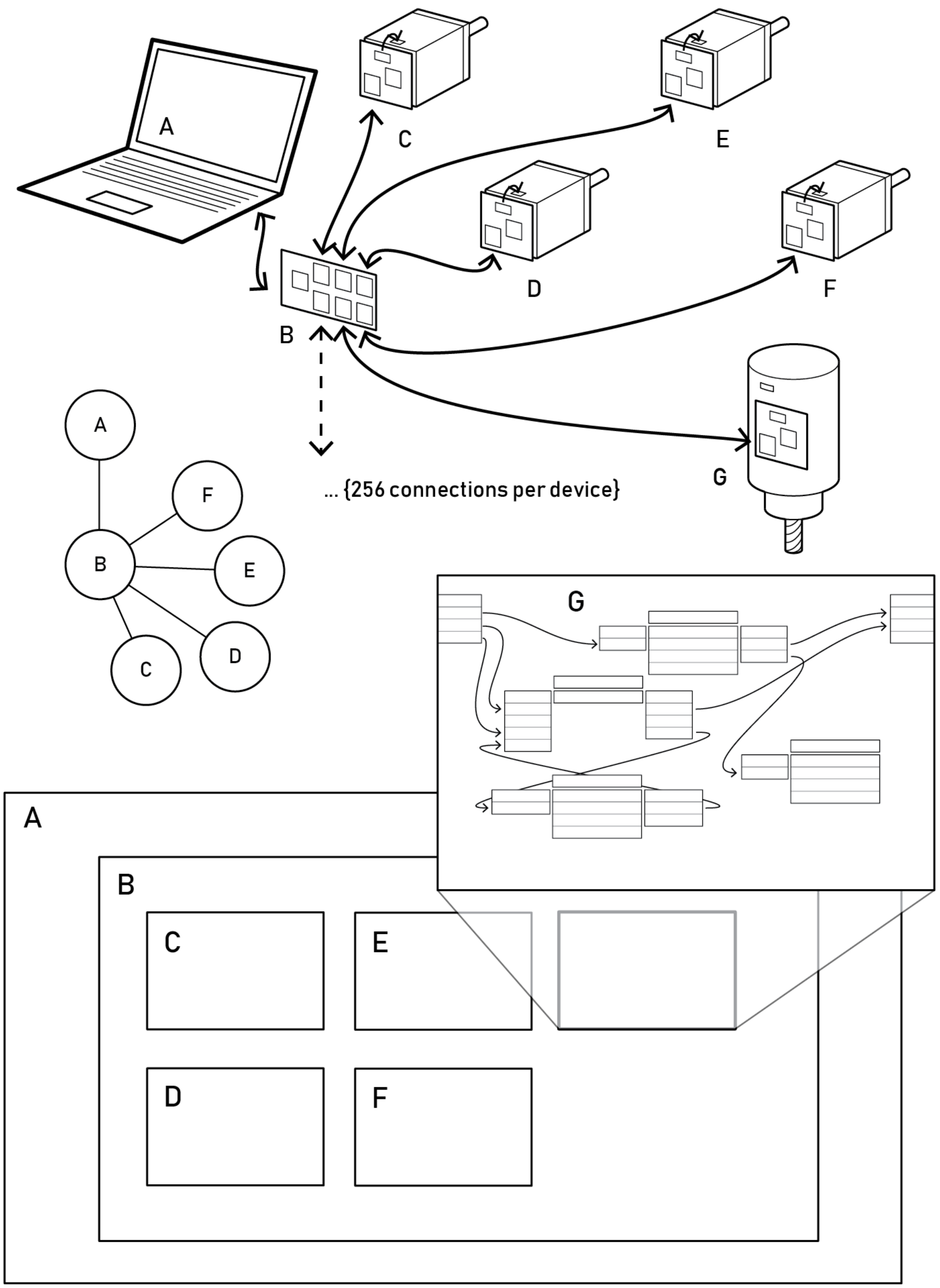


Figure 3-7: This figure outlines three views of one system: in physical diagram, abstract network topology, and a reciprocal nested-block view, where each block is one of the connected virtual dataflow environments.

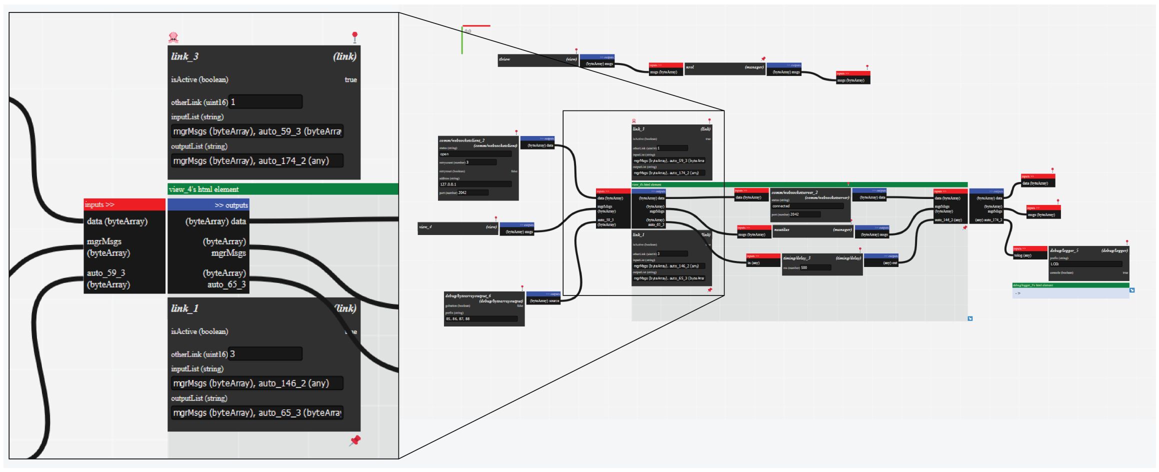


Figure 3-8: Two views from the development tool: the top level renders hunks operating within the tool itself, and a lower level, where hunks are nested within a link to represent network traversal.

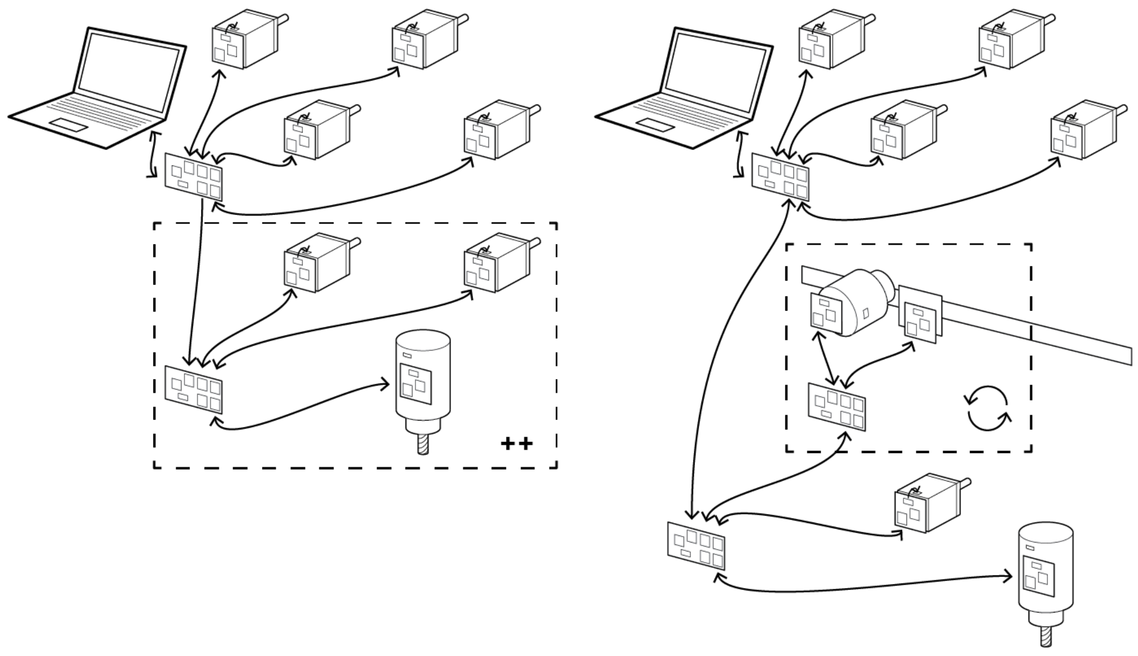


Figure 3-9: By growing the network, and extending computing resources as hardware resources are added, numerous components can be added to a single global controller. Control loops can be closed locally, on branches of the network where they are relevant. Reciprocal graphs and views of these systems are diagrammed in figure 3-10.

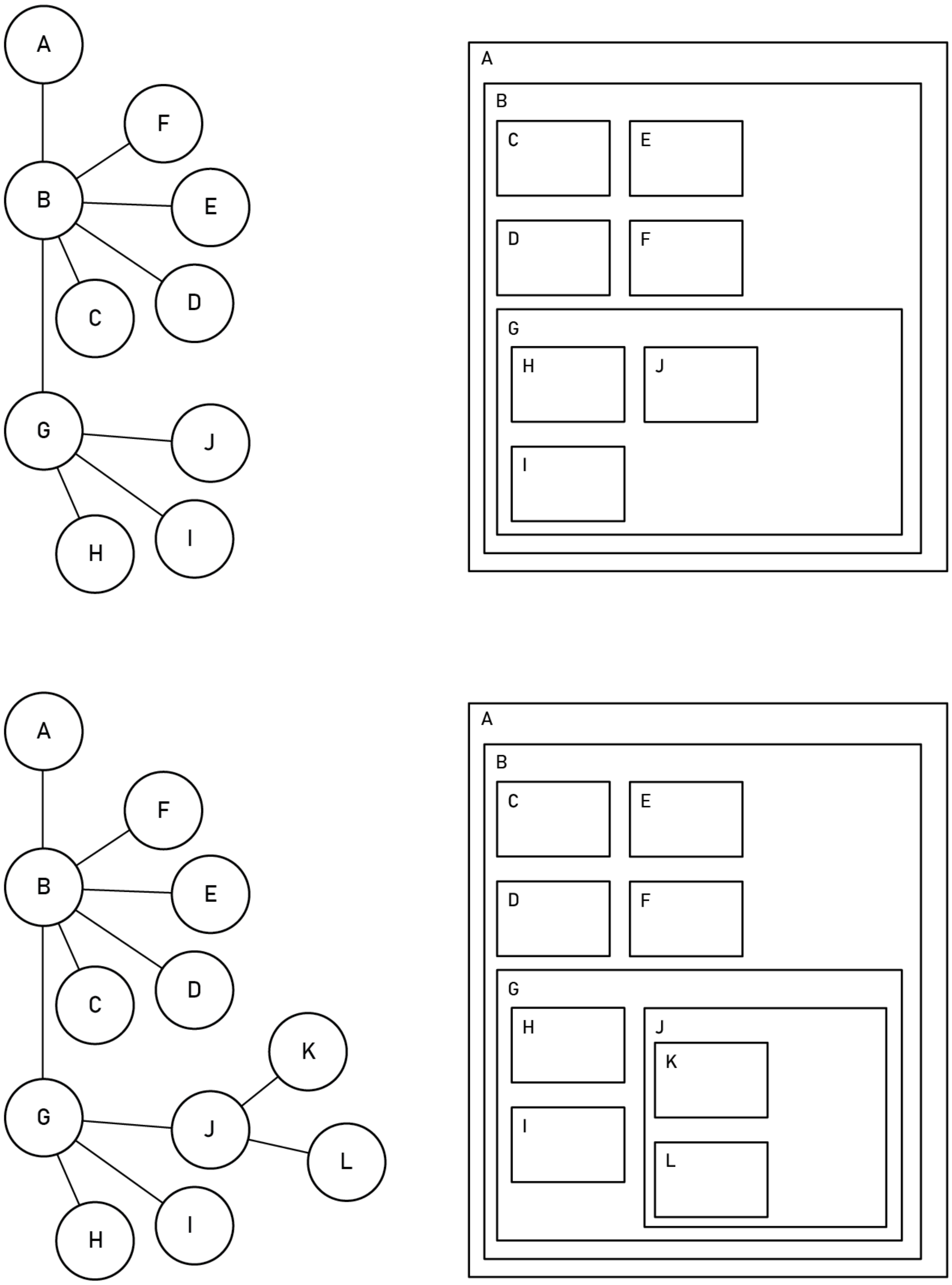


Figure 3-10: These are reciprocal representations for the network extensions shown in figure 3-9. This shows abstract node diagrams, as well as nested-view representations, as are rendered in the assembly tool from chapter 4, and shown here in figure 3-8.

Chapter 4

A Development Environment

I have so far discussed how distributed dataflow graphs execute within Virtual Dataflow Environments (VDEs) and how those graphs are embedded and extended into physical dataflow networks, to build programs that span collections of networked computing devices.

The last example from chapter 3, diagrammed in figure 3-9, shows a system composed of twelve different devices on a dataflow network, each of which runs a virtual dataflow environment that contains some collection of modular codes instantiated as hunks. If I suggest to build systems in such a way, where each component of the system is modular across these levels, I should also develop a method with which system configurators can ascertain the shape of these graphs - phsical and virtual - in order that they become useful, reconfigurable sets of building blocks. One such tool was built during the course of this thesis, and we can cover its operation briefly in this chapter.

4.1 A Model-View-Controller

The model-view-controller is a software design pattern commonly used in the web, to develop user interfaces that manipulate some remote system state. In an MVC, the *model* refers to some objects whose state we are interested in. In many cases, this is a static database on a web server. The *view* is the interface served to an individual that would like to edit that state, and the *controller*[1] is a program that arbitrates between the system's actual state and the user's view[11], using a messaging model between the two programs to do so. A diagram of this relationship is included in figure 4-1.

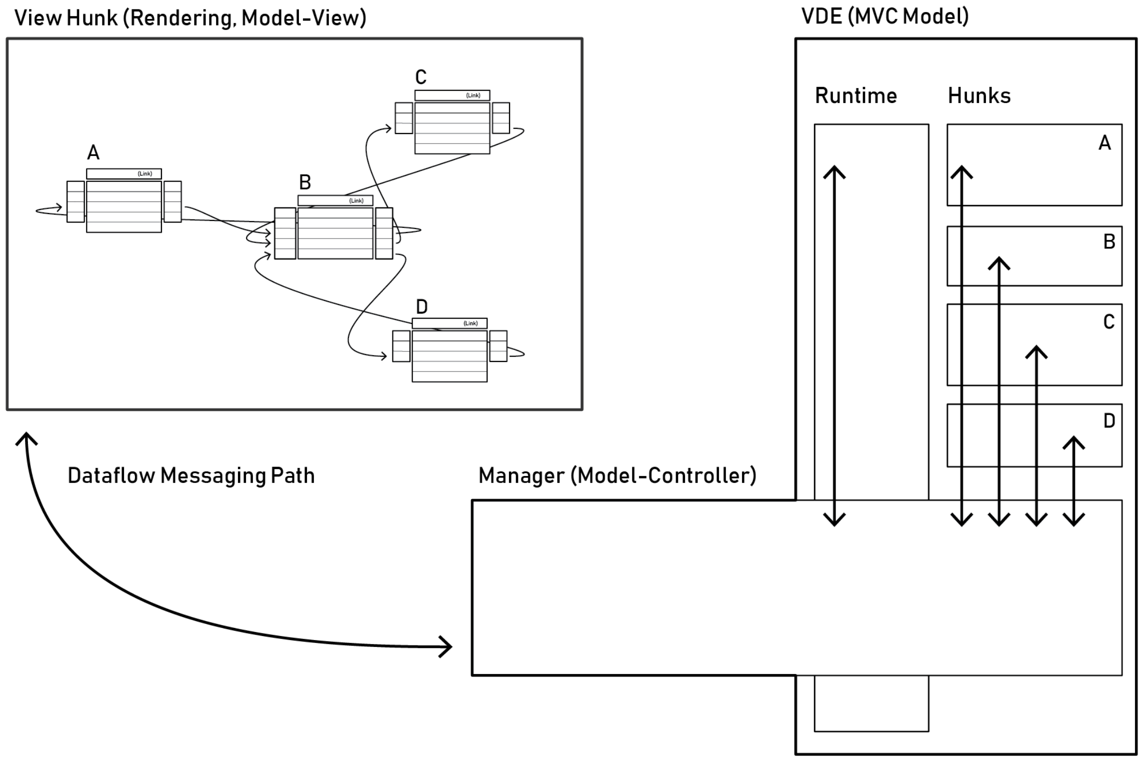
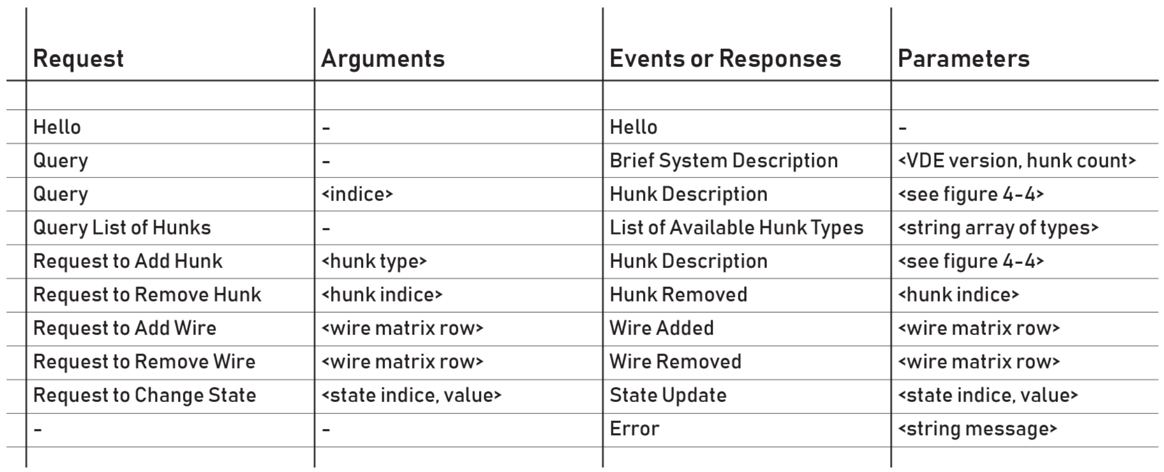


Figure 4-1: An abstract diagram of the relationship between our model-view (in the editing tool) and model-controller (at the VDE).

It is sufficient to explain that this graph development tool implements an hierarchical set of model-view-controllers, one for each VDE in a graph. To get an overview of how these systems work, I diagram messages that make up interactions between Views and our VDEs in figure 4-2.

Using these messages, the view captures remote graph state and renders it locally, so that remote graph configurations can be made into editable objects for system configurators. The core set of messages required to exchange requests and updates between views and VDEs is small. The tool can use chains of these simple messages

[1]The word *controller* here takes on a meaning different than what many readers may be accustomed to.



Request	Arguments	Events or Responses	Parameters
Hello	-	Hello	-
Query	-	Brief System Description	<VDE version, hunk count>
Query	<indice>	Hunk Description	<see figure 4-4>
Query List of Hunks	-	List of Available Hunk Types	<string array of types>
Request to Add Hunk	<hunk type>	Hunk Description	<see figure 4-4>
Request to Remove Hunk	<hunk indice>	Hunk Removed	<hunk indice>
Request to Add Wire	<wire matrix row>	Wire Added	<wire matrix row>
Request to Remove Wire	<wire matrix row>	Wire Removed	<wire matrix row>
Request to Change State	<state indice, value>	State Update	<state indice, value>
-	-	Error	<string message>

Figure 4-2: A table of the messages that furnish the relationship between the model-view and model-controller.

as *recepies* that execute complex operations like graph re-instantiation and merging. In this way, the complexity required to develop, save, and load graphs within the development tool is simplified, and lightweight, robust remote computing environments can be developed, as complex operations are relegated to the tool itself. This is particularely helpful when interfacing with embedded VDEs, where memory and computing resources are limited.

In this architecture, I do not take a compile-time or software approach to system abstraction like the one discussed in section 1.3.9. Instead, I can write native codes (hunks) for specialized microcontrollers, that each accomplish some core task that might be required at the application level. I wrap these codes in meta-softwares (VDEs) that are dataflow assemblers, and construct a tool that interfaces with the VDE and can assemble hunks into graphs at runtime, via communication over the network with an MVC.

Because VDEs, which are embedded in devices, can communicate with the tool during graph assembly, specialized devices can be delivered to application developers already containing all of the modular codes that suit the particular device. Libraries for these remote devices do not need to be added to the tool prior to the device's arrival on the network, as is the case with LabView or Simulink from section 1.3.8. Instead, application developers can simply connect the device to an existing network of control, and begin querying its MVC for the available modular components, and then instantiate and integrate them by connecting those modular components' inputs and outputs to their system. This method follows the end-to-end principle to minimize complexity: new capability can be added at the edges of a system without modifying any of the protocol required to integrate that capability.

4.1.1 Saving Graphs

After graph state has been ascertained, edited and configured to our liking, the development tool can be used to save a snapshot of the graph's current state - its hunks, their connections to one another, and their state variables - to memory, so that it can be re-instantiated at a later date for use or editing. I save graph state in the JSON object format, a JavaScript-native but widely used serialization for generic, non-circular data objects in the language. Figure 4-3 shows one such list. When graphs are distributed across multiple physical devices, representations are simply nested within one another. This nesting in memory mirrors the network's hierarchy: lower level systems are 'contained' within the links from higher levels which connect to them.

4.1.2 Reinstating Graphs

In order to reload these systems, model-view-controllers sequentially and hierarchichally reconnect remote state based on saved program representations. This operation is one such *recipe* mentioned earlier, and it is described below:

1. Send the hello message, establishing an MVC connection.
2. Query the manager's top level.
3. Query hunks from the 0th to the length of the manager's list.

The view now has an up-to-date model of the remote environment, and can perform a merge.

1. For each hunk in the saved graph,

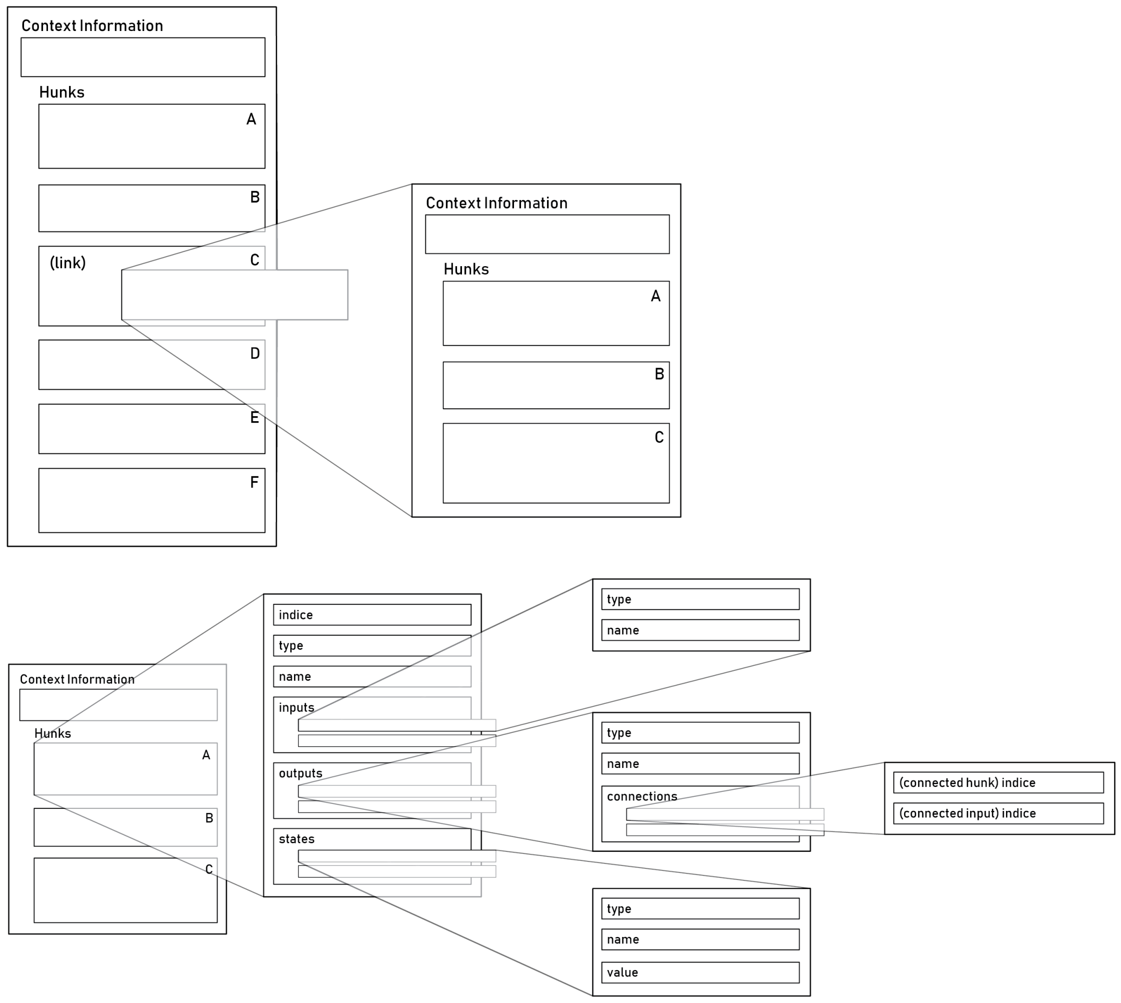


Figure 4-3: To save the contents of one virtual dataflow environment, the development tool writes a data object containing first a description of the VDE - its name and version - and a list of the hunks it has currently instantiated and described. Hunks are stored as data objects containing descriptions of their inputs, outputs, and state variables, including types and, at the output, a list of the inputs to which that output is currently connected. This is enough information to form the messages that will re-instantiate the graph, and check the re-instantiated version against the stored version. Systems that contain multiple VDEs are saved together in a nested, hierarchichal fashion that matches network topology.

   (a) If this hunk does not already exist in the graph, request its instantiation.

   (b) If it does, check that states are identical, if not, sequentially request state updates.

   (c) Complete.

2. For each wire in the saved graph, request the addition of such a wire.

This way, we can forward-declare ideal system graphs, but final outcomes are negotiated between the tool and the VDEs that actually operate the graph.

4.1.3 Piecewise Assembly, not Forward Declaration

Loading the graphs that we have been occasionally calling 'programs' is more like re-establishing state in a remote system, than it is to 'loading' a program into some computer's memory, and is done in negotiation with an VDE through an exchange of the messages figured in 4-2.

This marks a key distinction from compiled programs, where some structured language is written using a set of human-readable symbols (i.e. variable and function identifiers as string literals), and then these symbols and structures are interpreted by a compiler into hardware operable codes that can be executed in a computing substrate. For example, C code compiles into assembly language which is in turn written into a binary file. This binary file is then written into some cpu's memory, and on startup the cpu enters the program at a known memory address, to begin following instructions. Because the actual physical structures that perform computation are well understood, it can be known a-priori what a certain binary structures will 'do' when a cpu is pointed towards it. It is precisely the problem of forming an accurate understanding of what the device will do when given certain instructions that is particularely difficult with computing for microcontrollers, as they are a highly specific, difficult to abstract form of computing, where particular memory addresses are not memory but indeed are switches that operate perhipheral machinery. This is exactly what must be ascertained when embedded programmers reckon their codes against microcontrollers' typically-2000-page datasheets. Only a close reading of the datasheet can explain to program developers what will happen when they assign certain bits to certain memory addresses. Additionally, microcontrollers are embedded in a particular physical application - a collection of physical hardware: sensors, switches, etc - whose operation beyond the computing medium itself cannot either be intelligently abstracted in software.

The distributed dataflow approach is to write custom VDEs for custom devices and, rather than attempting to normalize a translation between a code-level representation and device-level physical output, as is done in the Arduino project mentioned in section 1.3.9, to normalize across a descriptive level, and to normalize an interface - our model view controller - that gives system assemblers access to edit that level. One insight is that descriptions based on dataflow - where core objects have inputs, outputs, and states, can be used to successfully describe almost all and any computing device that is connected to a network. To add, this description can be successfully employed to describe the network itself, meaning that we don't have to carry two different representations around to successfully describe our entire system state.

I should stress here again, as I did in 2.3 that the model is only descriptive - while graphs themselves behave in a standardized manner, the internal operation of each hunk is obscured to the user.

In a certain sense, where G Code can be read as an attempt to develop a subset of messages that can be reliably interpreted in time - to specify at a high level the ideal system output of a lower level system, this tool can be understood as a collection of messaging standards, alongside an execution model, that does not describe ideal control outcomes, but instead describes control algorithms themselves. Rather than delivering a layer of abstraction for runtime outcomes, it provides a layer of abstraction for runtime editing.

Chapter 5

Implementation

I have completed a description of the system architecture, describing how virtual dataflow environments are embedded within physical dataflow nodes, to collapse configurations of both modular codes and modular hardwares into cohesive program representations. I have also shown how to use a hierarchichal model-view-controller and graphic interface to view, edit, save, and restore these systems.

This architecture was implemented during the course of this thesis on custom hardwares, network interfaces and codes. In this chapter I will briefly examine these systems as implemented, and discuss their performance.

5.1 Supporting Hardware

5.1.1 Circuits

Modular circuits were developed to operate modular hardwares. Pictured in figure 5-1 are the core module, as well as its implementation as a message passing device, and a stepper motor driver. Figure 5-2 shows it's implementation as a brushless servo driver, and a DC motor driver.

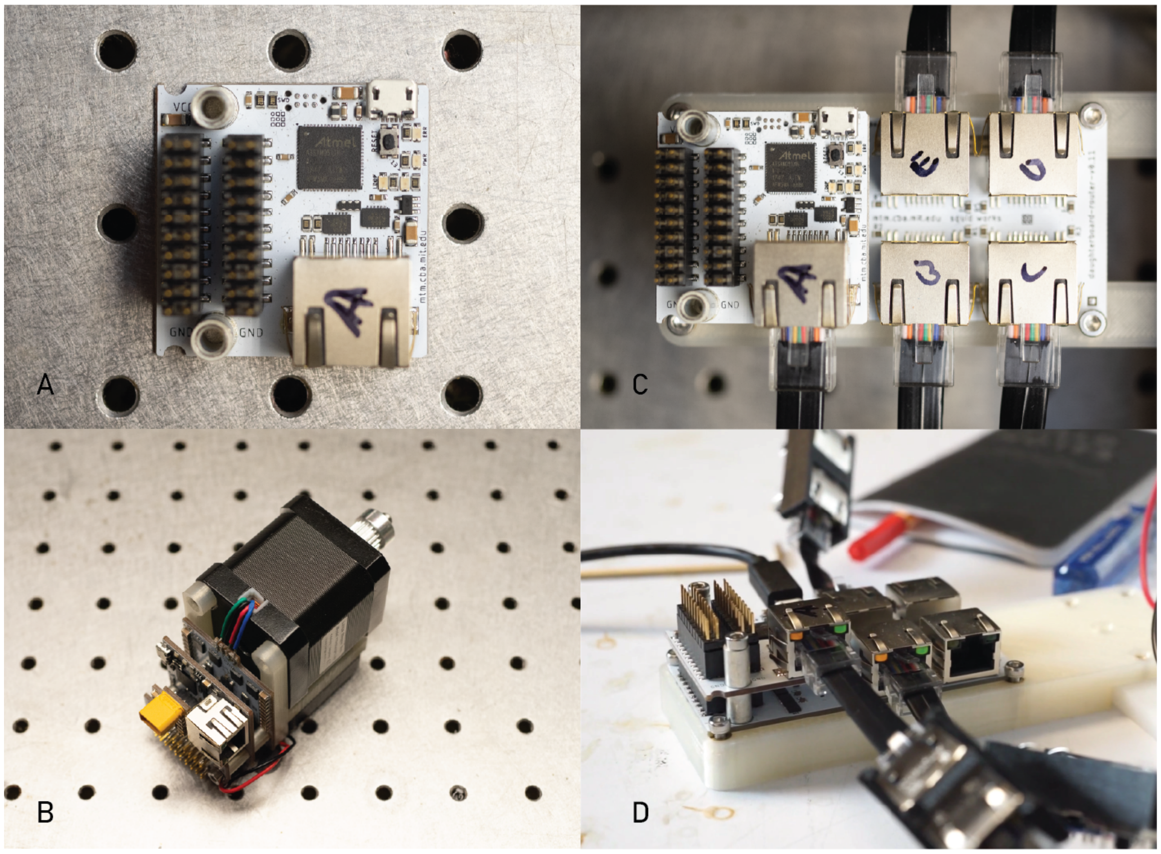


Figure 5-1: Circuits were implemented using a core module (A), that plays host to a number of varying sub-modules, a stepper driver (B), and a message passing device (C) and (D).

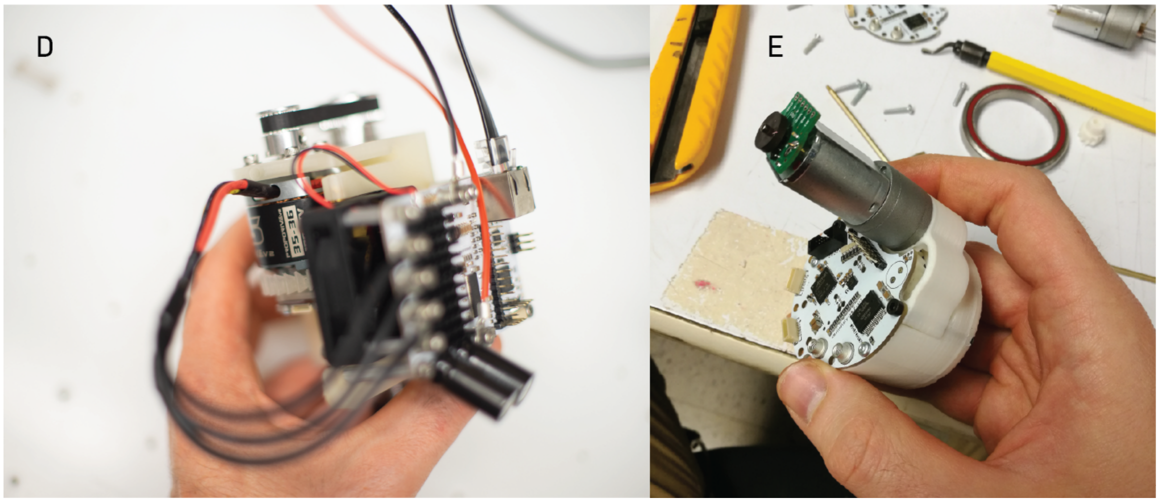


Figure 5-2: A Networked Brushless Motor Driver (D) and DC Motor Driver (E).

5.1.2 Embedded COBSS Physical Layer

To network between low-level contexts, a small networking physical layer was developed. Electrically, the link is driven by RS-485 differential drivers across modular

RJ45 connectors. The host microcontroller signals the RS-485 drivers with a UART peripheral, and operates link-status lights that aid in debugging. To optimize performance, bytes are received on an interrupt, and transmitting is similarely triggered, preventing the host microcontroller from locking out of other operations during the messages' relatively long transmit and receive times. The implemented baud-rate is 3Mbit, and packets are delineated using COBS[1][7], a simple zero-terminating delineation scheme that incurs low computational overhead and has deterministic packet length overhead. I call this a *COBSS Physical Layer*. This networking scheme was developed for low-level contexts over technologies mentioned in section 1.3.7 for its simplicity and ease of integration. Particularely, this scheme treats physical ports as network addresses, and links are always assumed to be point-to-point, rather than bussed.

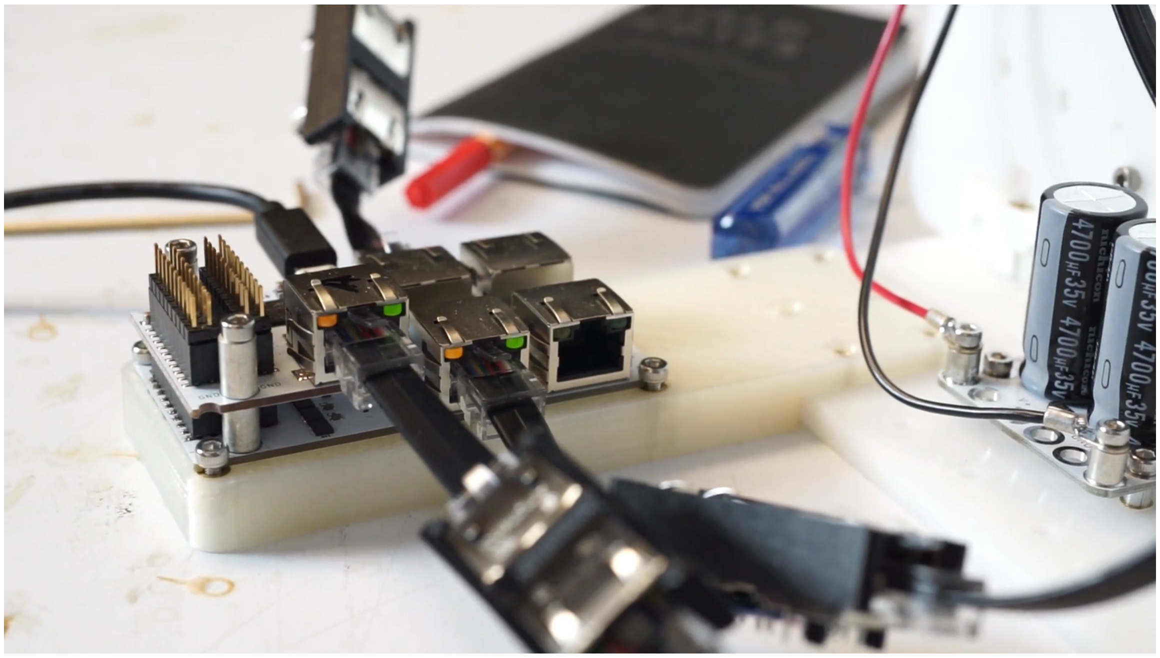


Figure 5-3: A custom PHY was developed for message passing between embedded contexts. While not as performant as Ethernet, Profinet, CAN or other existing Physical Layers, it is incredibly simple and requires no configuration for source routing.

[1]Consistent Overhead Byte Stuffing

5.2 Implementing Virtual Dataflow Environments

To run modular dataflow codes on hardware, the thesis implements a Virtual Dataflow Environment in C++ compiled against the ARM-Cortex-M4F ATSAMD51 microcontroller. This implements hunks as C++ classes, allowing them to be instantiated or destroyed during runtime. Links are stored in memory and data transport between hunks uses simple memory copying routines. The system operates in cycles, to simulate parallel execution of embedded hunks.

A virtual dataflow environment was also developed for the browser, using ES6 JavaScript modules to instantiate hunks on the fly. Moden JavaScript interpreters allow code to be dynamically added, and so the browser environment can also reload modular source codes during development at runtime. Datas are transported between hunks through virtual copying functions, allowing also for type conversions to happen between types where a conversion function has been established. The system's cycles take place between each turn of the JavaScript engine's own event loop. The browser VDE doubly serves as the tool discussed in chapter 4, rendering remote graph states as well as operating browser-native computing.

To connect to local hardware, the browser's dataflow environment is served from a small server application. This also functions as a bridge between the browser and server-side hardware, like USB Serial ports.

5.3 Timing

To understand the relative performance of the architecture, I performed a simple timing test within and across its layers. This helps to answer the question posed in section 2.3, about the appropriate sizes of our modular code blocks. Since native code can be implemented within a hunk, it is possible to re-create most of state-of-the art machine control simply via effort in programming known controllers into abstract 'hunk' wrappers. However, the timing restrictions that arise via the architecture's operation between these modular codes is what determines the level of modularity that can be made available in the system. I.E. if an encoder-driver hunk and a motor-controller hunk can reliably transmit datas to one another at frequency greater than $1kHz$, the system will be useful to develop closed-loop controllers between these two modules at bandwidths of $1kHz$, otherwise, the loop must be closed within one hunk that operates both hardwares, meaning we have failed in modularity in this 'level' of control.

The analysis here includes operation within a singular dataflow context as well as operation across a collection of network links, through varying contexts. This informs where modularity is available at the per-hunk level, as well as between unique hardware elements. That is: it demonstrates the performances that could be expected for control loops closed between modular code blocks, and then between modular pieces of hardware.

5.3.1 The Ring Test

The Center for Bits and Atoms has an interest in the couplings between computing systems and the physical resources they manipulate. To explore the relationship, we have compiled a library of *ring tests* performed on various computing environments[6]; the ring test sets a baseline for how quickly information can traverse from a computing environment, into the physical world, and back.

In the ring test, a processor is used to invert some physical state. An output pin is connected to another (an input), whose state (high voltage, or low voltage) is read into the CPU, inverted, and written out again on the first pin. This forms a ring oscillator; the speed of its oscillation is a measure of the system's bandwidth between its cpu and physical outputs. We include table 5.1 of selected results as well as the chart in figure 5-4.

In particular, we should note the ATSAMD51, the microcontroller used in this thesis to develop a low-level computing context. With optimized code, it runs a loop time of $270ns$, roughly $3.7MHz$. It should be noted that in this example, the microcontroller is wholly occupied with this test, running no other routines but checking the state of its input, and inverting the output.

Table 5.1: Selected Results from the Processor Ring Test.

Processor	Langauge (Framework)	Period (us)	Frequency (MHz)
PRU on PocketBeagle	C	0.06	16.7
ATSAMD51	C	0.27	3.70
ATSAMD21 (M0)	C	0.687	1.45
ATSAMD21 (M0)	C++ (Arduino)	6.14	0.163
Raspberry Pi	Python	10.0	0.10
Raspberry Pi	C	0.4	2.50

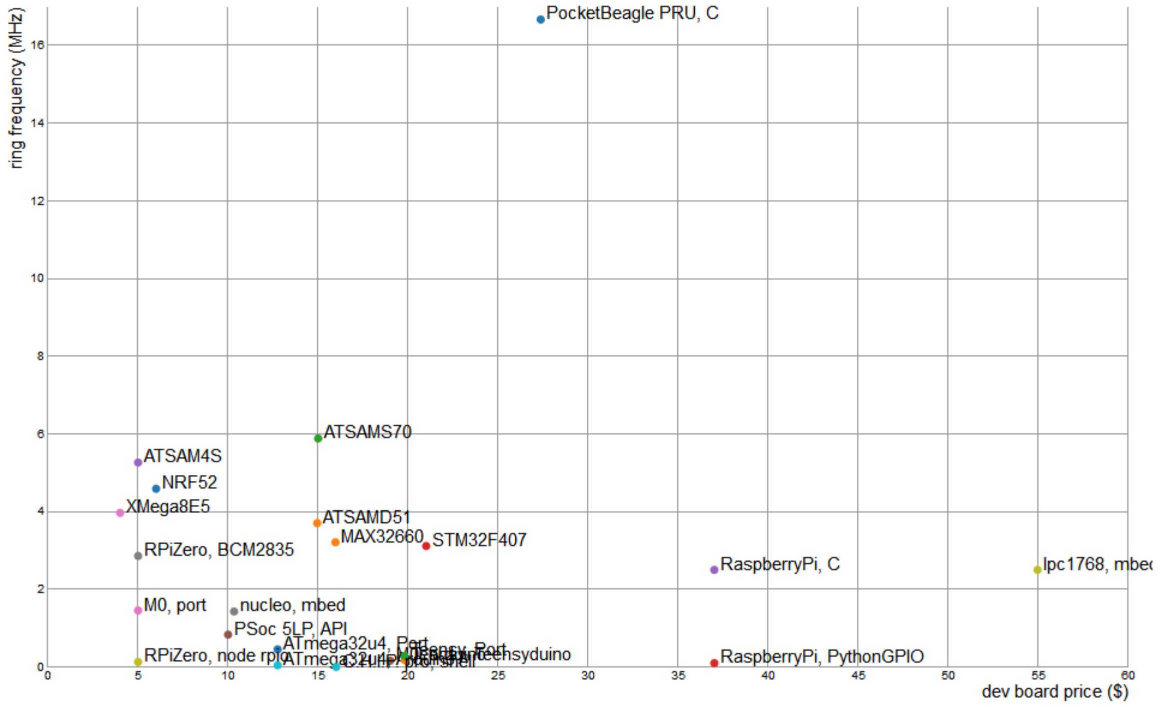


Figure 5-4: The ring-test establishes a baseline bandwidth between a computing device's CPU and its hardware outputs, or pins.

To compare native performance to our dataflow environment, I implemented a similar test across the series of computing hierarchies available within the system as implemented. For brevity, I include the results of this test in table 5.2 below. Figure 5-5 displays the ring test operating across increasingly deep graphs, as rendered in the assembly tool from chapter three.

5.3.2 Within One Hunk

First, I implement the ring test within one code block. This hunk uses its loop function to check the pin's state, and then invert it. This oscillates at $245kHz$, a period of $4.08us$. Reduced bandwidth here is an artefact of the system cycles implemented in order to create a virtually parallel environment. Over each system cycle, each hunk runs some loop code. In this case, the VDE is also running loop codes for its networking hardware.

5.3.3 Within the Embedded VDE

I then abstract the test across the virtual dataflow environment itself, reading the input pin's state with one modular code block, and transporting that state through the virtual dataflow environment, where it is read-in by another code block and inverted before being written to the output pin. This runs at $54.9kHz$, an $18.21us$ period. Extra cycles are consumed in abstracting the pin's state to a generic boolean value, and copying and transferring that value along data paths within the virtual environment.

5.3.4 Across one COBSS Physical Link

The test was then performed while taking sequentially larger steps through a physical network. Operation across the first link - the custom PHY mentioned in section 5.1.2 - drops the bandwidth to $3.21KHz$, or $311.5us$. The bit-time[2], is only $16.8us$ for boolean messages, but recall that each message sent across a link is also acknowledged, and that the $18.21us$ data transferring time on either side of the link is included, within each virtual environment. Additional performance is lost as datas in this test must be transferred from pin-manipulating hunks, to a link (networking) hunk, which then frames the data in a packet, and finally to the COBSS Physical Layer, itself represented in-system with another hunk. All said: to exit one VDE and arrive

[2]The time that it takes to transfer messages across the physical link, a simple function of the bitrate and number of bits.

within another, the data must traverse all layers of the architecture presented.

5.3.5 Via USB, into JavaScript

Finally I added a third layer, bringing the logical operation into the browser runtime, connected to the first two layers via a serial usb connection. This connection is then ferried through a websocket into the browser, where the logic state is inverted, and returned.

Here I find a major slowdown, as the transport layers are now encumbered in operation within what are effectively two other operating systems: first, the server must retrieve data from the USB Serial port, before passing it along to the WebSocket, where the browser is connected. The browser computing environment provides more indeterminism, as it runs an internal event loop that asynchronously catches this websocket event and finally serves it into the virtual dataflow environment.

This ring test has an average frequency of $14.6Hz$, an average period of $68.5ms$ and importantly, it shows a large distribution - the minimum period is only $35.4ms$, but the largest is a complete $120.2ms$.

This is an important point: as graphs move into larger computing resources, virtual dataflow contexts as implemented are themselves running within a virtual environment - the operating system. While the flowcontrol model holds up, and action is still synchronized to global execution, that execution is asynchronous, and in cases like these, it is gated by the highest latency device in the loop.

It is my strong suspicion that much of this delay arises from latency in the serialization codes used in JavaScript, a language where retrieving low-level byte representations of data is non-straightfoward. The language is making progress in this regards, and we anticipate improving the performance of these routines.

However, as is often the case, I do not need high bandwidth controllers in the top layers of these controllers. The browser is much more useful for path planning than for path execution, for example - an event that normally occurs only once per job, or in the eventual case of realtime path re-planning, may only happen once per second. It is also useful when running computer vision operations that still benefit from the

GHz and GBs of computing available in modern personal computing, as opposed to the microcontrollers we use in lower levels of control. Critically, long latencies at the upper levels of a system do not affect latencies between lower levels, as discussed in section 3.2, where small control loops between, for example, a motor and encoder, can be closed exclusively between those two devices, and need not traverse through all layers of the graph.

Overall, I anticipate that future work will use reconfigurable embedded computing as a tool to move more system operation away from cumbersome environments like the browser and other operating-system bound programs. Ideally, a core set of compute hunks will be developed for headless embedded systems and, given the architecture's affinity for parallelization, high speed, high-level routines will be developed with massively parallel computing across collections of low cost microcontrollers.

5.3.6 Ring Test Conclusions

While it is a small exercise, the ring test gives a sense of where in the system hierarchy various types of controllers can be located, and of which types must be written in native code blocks, to operate under appropriate bandwidths.

Table 5.2: Results from the Distributed Dataflow Ring Test.

Data Path	Frequency
Native Hunk	$245kHz$
Internal to one Virtual Dataflow Context	$54.9kHz$
Embedded VDE / COBSS / Embedded VDE	$3.21kHz$
Embedded VDE / COBSS / Embedded VDE / USB / WebSocket / Browser VDE	$14.6Hz$

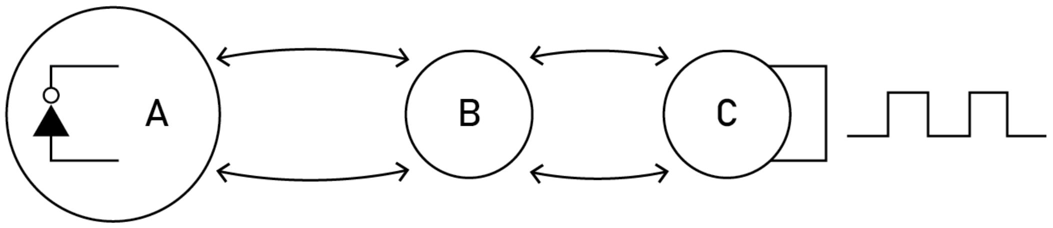


Figure 5-5: A diagram of data flow in one ring test, where physical state is inverted logically within the VDE at A, transported through physical dataflow between AB, BC, and inverted physically using the VDE at C.

Chapter 6

Application

machines in a week
it's easy, so to speak
in minutia is mayhem

6.1 Parts, Paths, Planning and Execution

In the introduction, I discussed the many software and hardware layers that lay between the representations we use to describe the components we would like to manufacture, and the eventual process control events that occur in order to render them into physical artefact. A key tenet of this work is that, by describing codes as well as hardwares with one dataflow representation, many of these layers could be collapsed into cohesive systems: represenations of programs that span between these layers, without asking us to juggle multiple abstractions. In this section, I show that the architecture can span the layers between path planning and path execution.

6.1.1 Generalist CNC

A general purpose Linear Gantry Machine was constructed. This is outfitted with four stepper motor modules to operate its linear axis, one module to operate a brushless router spindle, and one message passing device. These are connected to the browser, where the tool from chapter 4 was deployed to build the low level path execution system required to accurately position the machine during operation (subect to acceleration constraints), and where interface and computing elements were deployed to ingest part descriptions as well as develop path-plans to render those descriptions into manufactured parts.

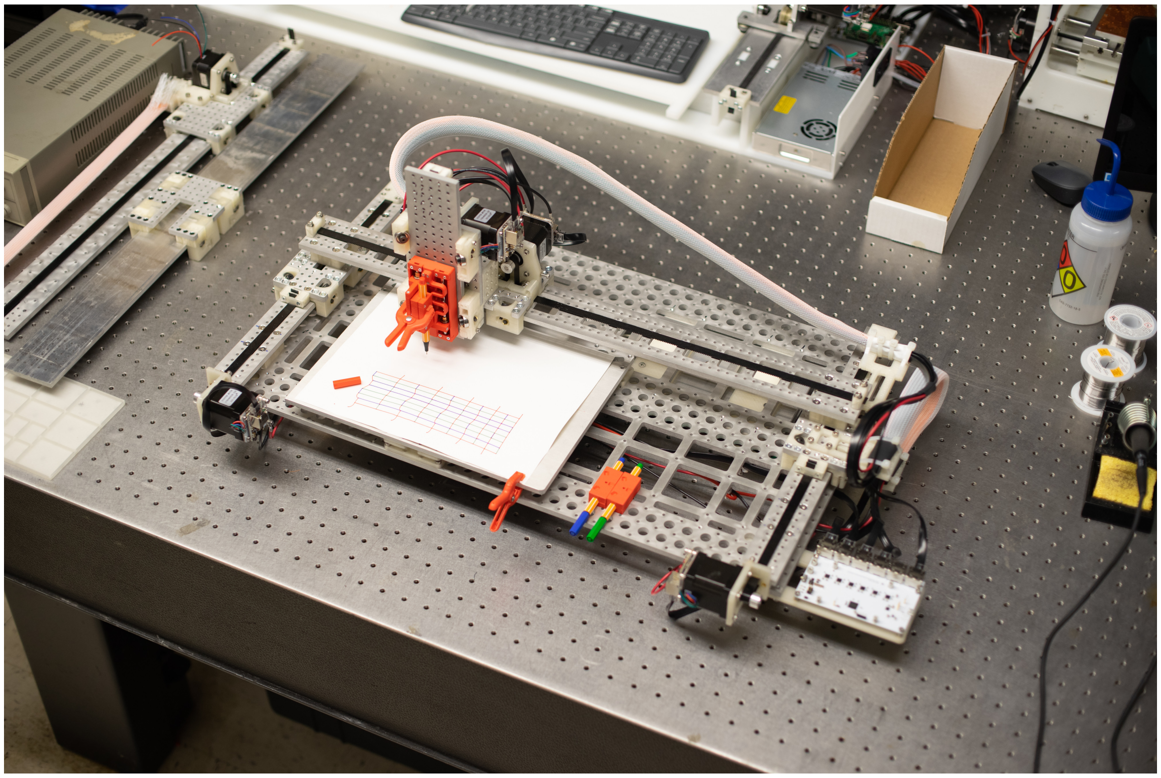


Figure 6-1: A cartesian machine generalist, the 'Little Rascal' is designed to be outfitted with a range of end effectors, from spindles, passive drag-knives, active rotary tools, pick and place equipment, etc.

All of the codes operating every level of the machine are hunks, and configuration of the machine's axis-to-motor mappings are simply routing structures in the graph that spans between those codes and the networked devices that house them. Some of this is explained in the caption for figure 6-2. The illegibility of controllers in global view like this is noted, and represent and ongoing challenge to succinctly render relatively complex systems in a coherent manner. This also represents the visual result of the challenge I discussed in section 2.3: each of these blocks represent some abstract set of code, whose internal function is not apparent in a rendering of the

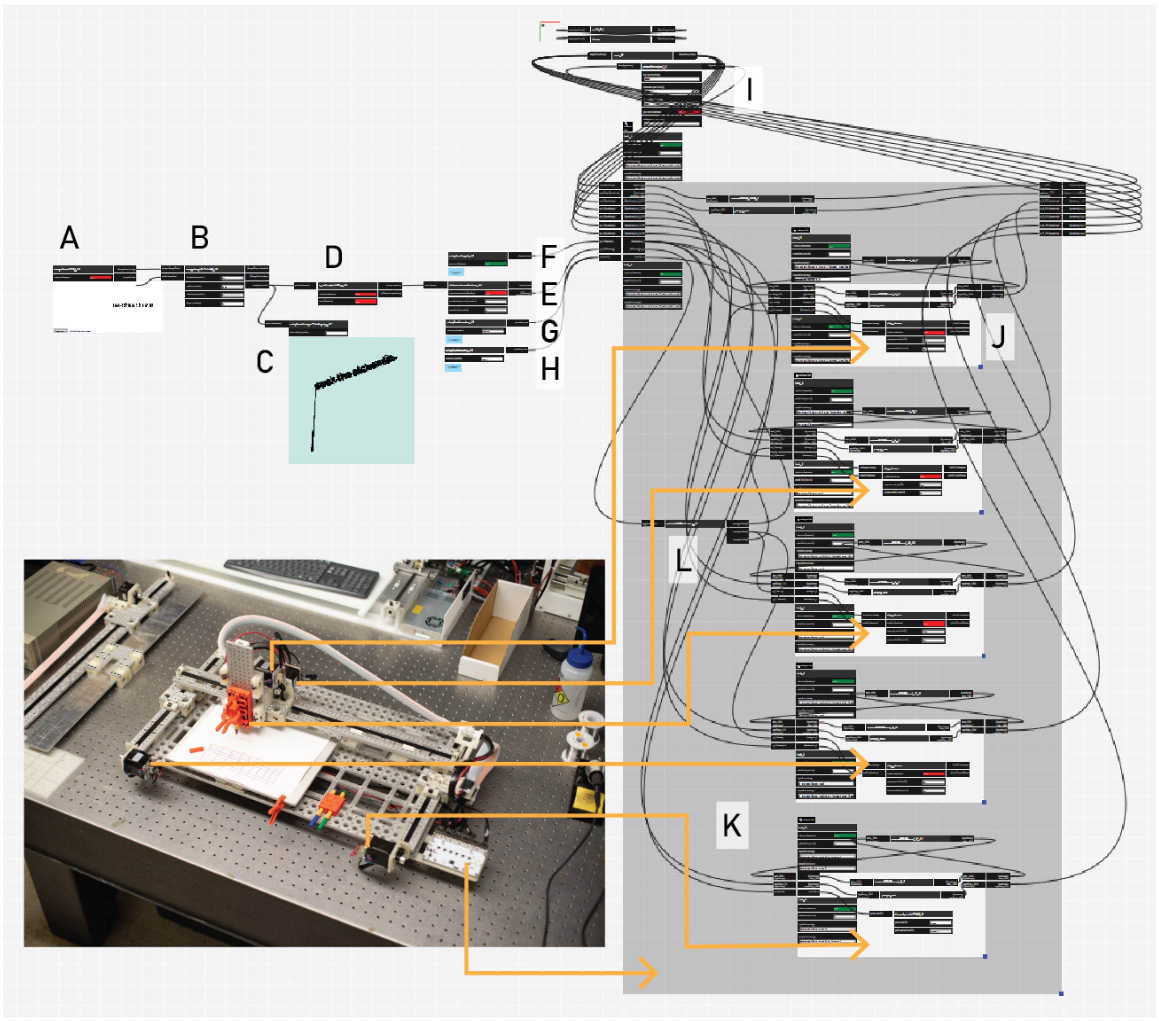


Figure 6-2: The distributed dataflow controller that operates the Little Rascal. Here, I've attempted to indicate which dataflow window represents which physical resource, and labelled some consituent hunks in the graph. *A* is an image input, this loads pixel data from the user's computer. *B* transforms that pixel data into a machine path: an array of 3 dimensional points. *C* displays that data in 3D. *D* releases this array point by point into *E*, a feed-forward acceleration control code, that ferries the moves on to motor-control VDEs like the one at *J*, which operates a motor. *L* transforms these multi-dimensional motion segments into single-dimensional segments, for individual motors. *F* enables or disables motors by sending either 'true' or 'false' to the same motor drivers. *G* and *H* send RPM commands to the spindle. The window beneath *K* indicates the VDE context for the message passing device (lower-right on the machine). Many of the hunks included, like *I*, represent drivers for networking hardware: *I* itself represents the serial port that ferries data between the browser tool and the message passing device.

graph. It is worth re-iterating here that the system developed is more of an assembly tool than a programming model: it manipulates modules, but does not itself define

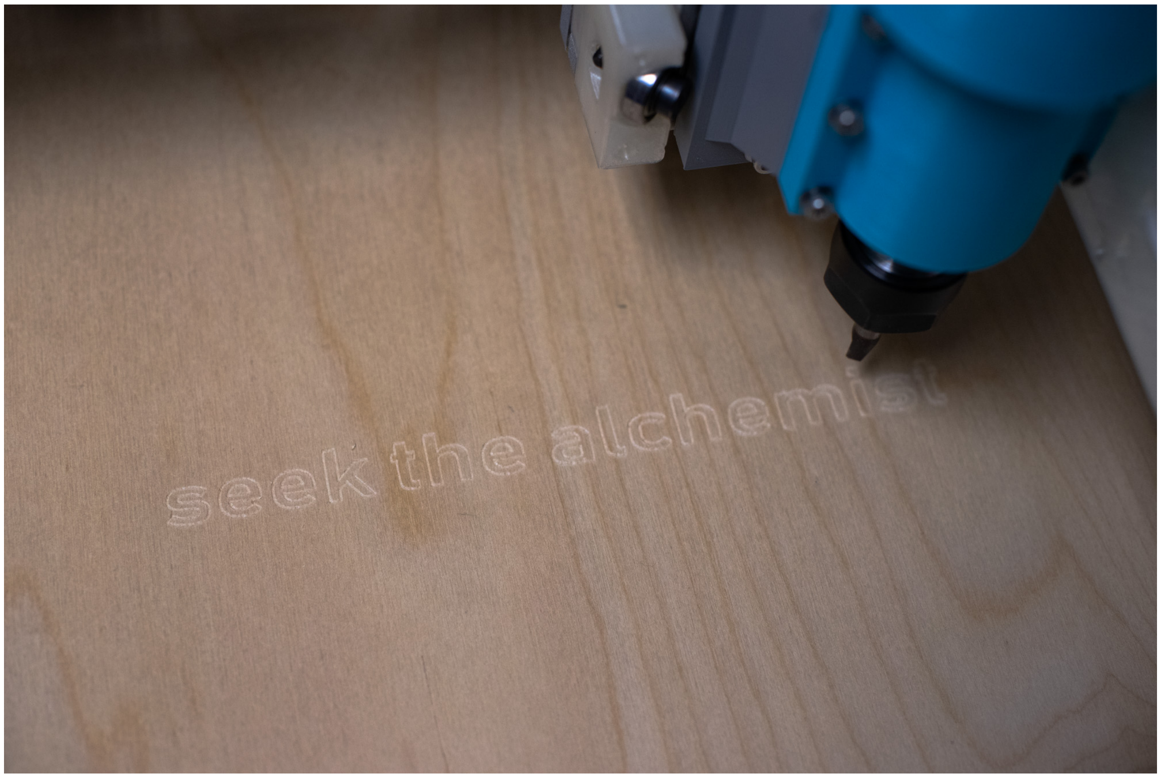
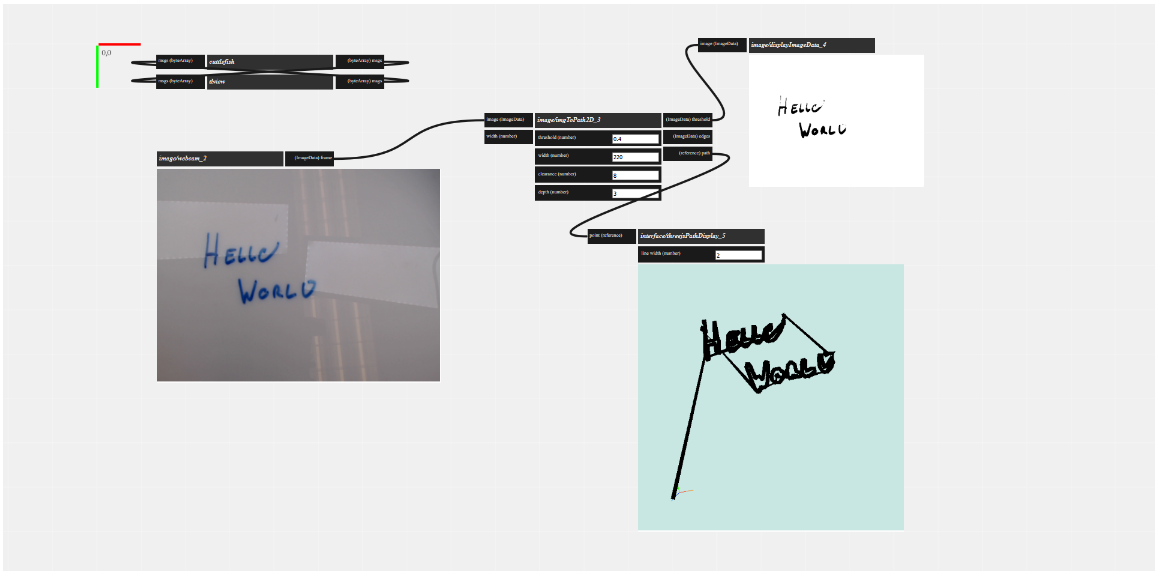


Figure 6-3: Output from the controller and machine.

their function. In future work, I hope to develop a core set of well documented code modules for basic data manipulation and mathematics, such that a wider range of applications could be developed without implementing or writing new code modules.

Building this machine and application also involved writing an acceleration-limiting feed-forward controller that is cognizant of the *times* that individual motion segments will take to execute, as well as their speed and acceleration constraints. Doing so greatly improved the scheme's ability to injest complex motion paths and distribute those moves across a network that imposes non-zero latenc. Path plans are described not with gcode but with straightfoward arrays of positions, marking a departure from the antiquated language. At each layer in the graph, data flows are made of decomposable types that are easily inspected.

To extend this work, I also demonstrate in figure 6-4 the possibility of connecting realtime part description (a drawing, captured using a webcam) into the path-planning program. I hope that the ability to quickly mix non-traditional workflows with one another will allow us to collapse some of the barriers between the experts who currently have access to advanced manufacturing and the novices or craftspeople who would like to gain access to the space.



threshold (number) 0.4
width (number) 220
clearance (number) 8
depth (number) 3
line width (number) 2

Figure 6-4: With modular codes, workflows can be quickly recombined, here using a webcam (rather than a saved image) as an input for machine path generation.

Finally, I show in figure 6-5 that realtime data from networked machines can be quickly retrieved from low levels - here I am simply visualizing torque feedback from one of the motors. Future work will incorporate these types of feedback into higher levels of controller - for example, rapidly adapting machine controllers' parameters to new processes, or even discovering optimal controllers from scratch.

6.2 Instrumentation: The Displacement Exercise

A stress and strain machine, commonly labelled as an *UTM*[1] or known by the name of one company that manufactures them[2], is a relatively simple device. Most UTMs feature one degree of freedom, to exert a strain on a material sample, and one load cell, to measure stress.

The UTM developed here, pictured in figure 6-6 was engineered to test bioplastic samples to contribute to an effort to develop sustainable, locally sourced materials.

[1]Universal Testing Machine
[2]Instron

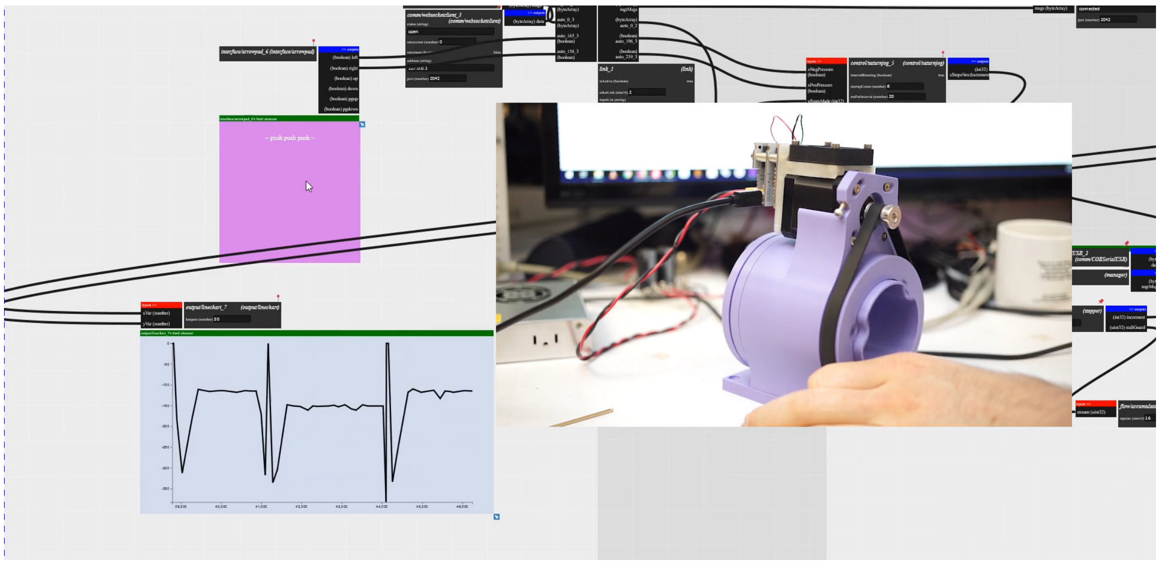


Figure 6-5: Here I visualize torque feedback from this rotary device in real-time, by routing an heuristic torque measurement (approximated with back-EMF sensing) output from the stepper driver into the browser, where they are plotted. The spikes in this chart were generated as the rotary device accelerated through direction changes.

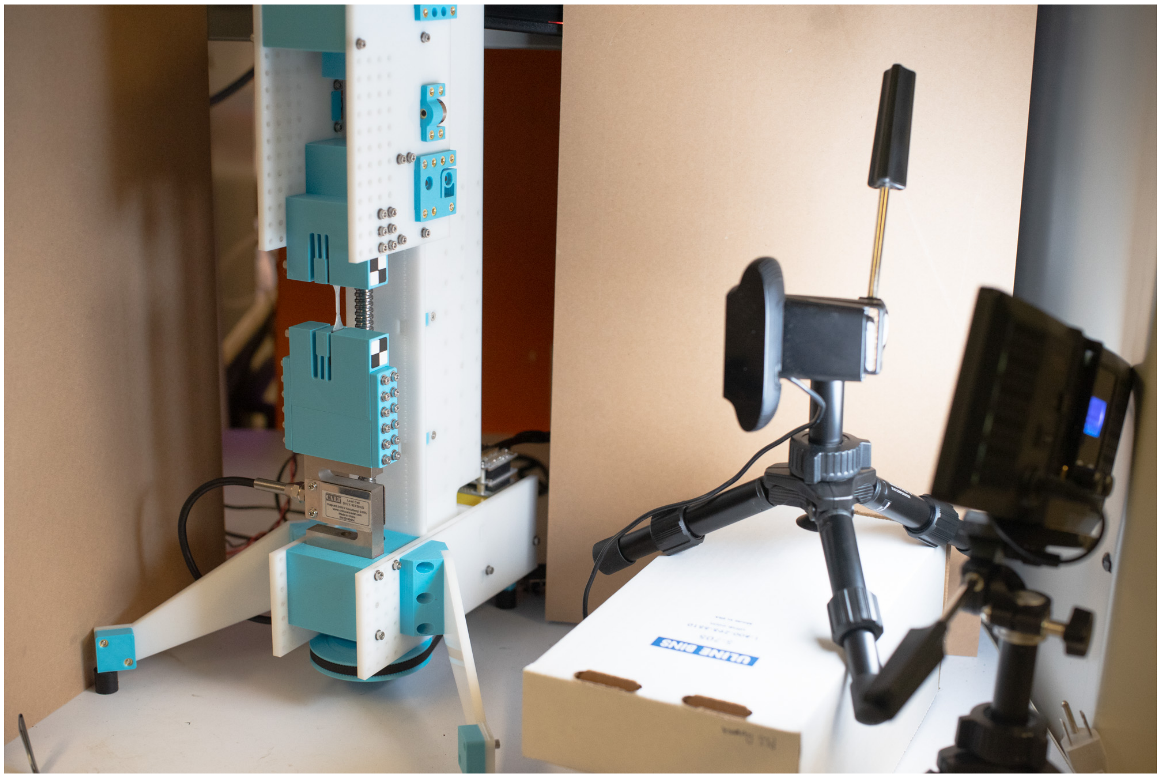


Figure 6-6: A small Stress and Strain measuring machine, using computer vision displacement measurement.

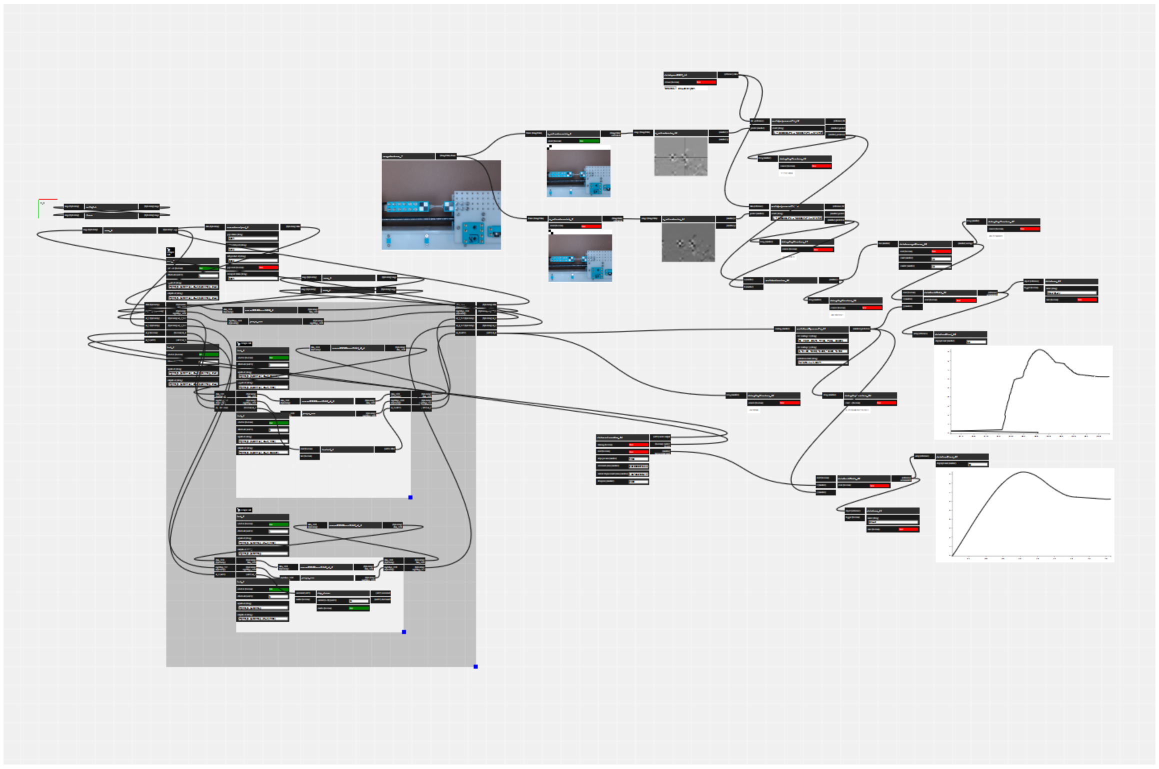


Figure 6-7: The distributed dataflow controller for a small stress and strain machine. This contains low-level motor controllers and sensor acquisition (left), computer-vision algorithms (top) and data linearization, transformation, display, and output systems (right).

Its controller consists of one stepper motor, one message passing device, and a load cell amplifier.

The controller performs tests using a cyclical routine. In each cycle, motors are incremented at a fine resolution, and a load cell reading is performed. Linear increments execute quickly, but load cell measurements take about ten times as long to resolve. Here, I was able to use flow control princples to block execution of motion increments while the loadcell is converting results.

However, the machine is itself somewhat deformable, the chassis was manufactured in a Fab Lab with acrylic, where commercial UTMs are manufactured with high strength steels, such that the machine's own stiffness can be essentially removed from the measurements of a given sample's stiffness. Given the flexibility of this instrument, I cannot simply measure strain with an open-loop calculation of the machine's kinematics, rather, its deflection must be taken into account.

To measure around this deflection, I was able to integrate realtime compter vision processing in the stress and strain machine's controller. Here, I composed hunks to track markers within a webcam frame, and measure their position using a subpixel method[40]. In order to calibrate between pixel-space locations and real-world measurements, I were able to use the distributed controller to capture open-loop measurements of displacement in real-world coordinates against computer-vision pixel measurements in an unloaded condition where little to no deflection of the machine was occurring. The calibration routine is pictured in figure 6-8. With this, I built a calibrated model of the relationship between pixels and real-world units. I then integrated this calibration during operation of the machine during loaded conditions, as pictured in figure 6-7.

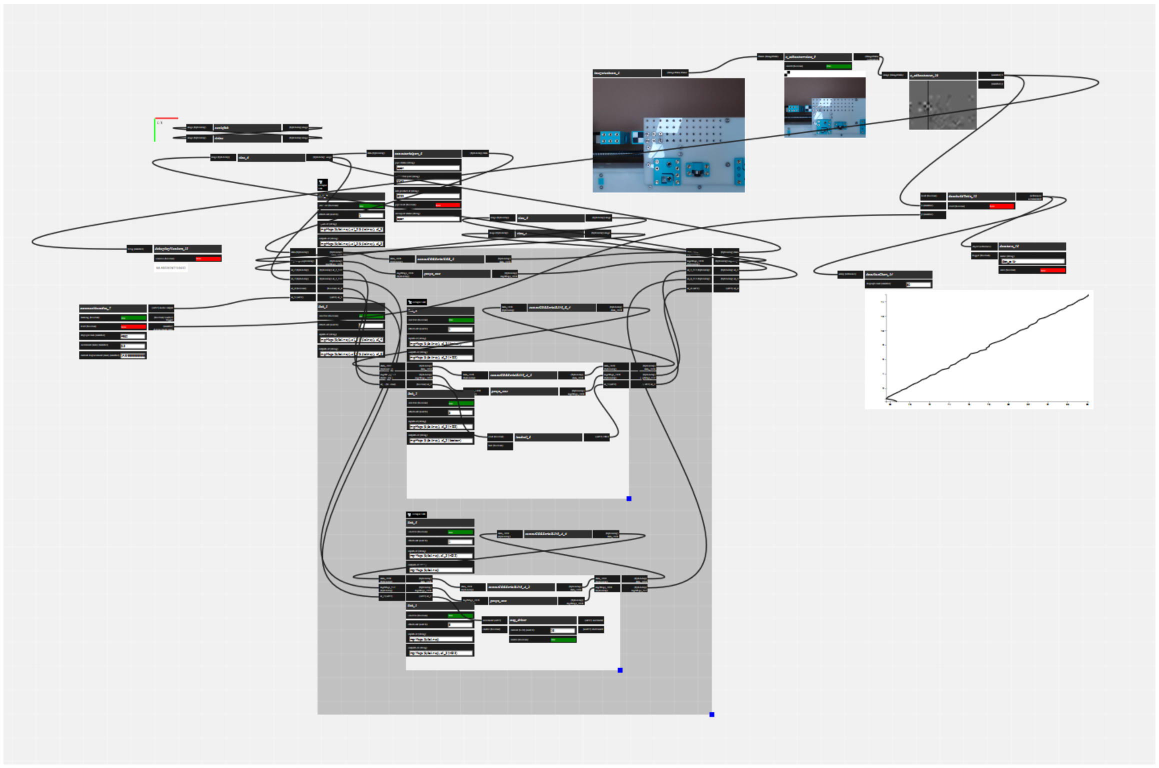


Figure 6-8: The controller, overlaid separately on the same hardware, that was used to build a calibration table for the relationship between pixel-space coordinates and real-world units.

With the stress and strain machine, I show that high level algorithms - the vision system - can be coupled with low level controllers, and that I can use their well coupled relationship to intelligently integrate them into processes that are greater than the sum of their parts. Additionally, the stress and strain machine's physical controller - simply a motor and a sensor interface - was developed within one afternoon using motor controllers previously engineered for use in our linear gantry machine, and the modular circuit I discuss in section 5.1.1. This shows that the architecture allows re-use of specialized hardwares for some variety of system-specific tasks.

6.3 Approachability

The system's approachability was tested during the 2019 cycle of Dr. Gershenfeld's class *How to Make Almost Anything* at MIT. During one week of the course, groups of students were challenged to design, build, and control a CNC Machine from scratch. To support their efforts, I assembled kits of the circuits listed in the previous chapter, and ran a tutorial explaining the systems architecture implemented in this thesis. One group of fifteen students had success with this system, implementing a distributed dataflow controller to operate a small lathe (for turning apples).

Machine operation was configured using existing hunks and circuits, showing that the architecture allows core components to be re-used across varied applications, and that new machine systems can be commissioned by novices without prior computer programming experience and without engaging in circuit design. However, system operation was not entirely clear to many students. Here I would refer again to section 2.3, where I discuss the limits of a systems assembly architecture such as the one presented to elucidate system operation. Because the systems' core componets - hunks - are descriptive wrappers on some block of code, it is impossible to ascertain graph operation purely from a reading of the graph: one needs also to understand the codes running within graph components. It became evident during this machine building exercise that naming conventions and structures familiar to myself, that allowed me to rapidly reconfigure or re-use existing codes, were foreign to novices. The machine controllers they assembled were not singular black boxes, but collections of black boxes wired together in a dataflow abstraction.

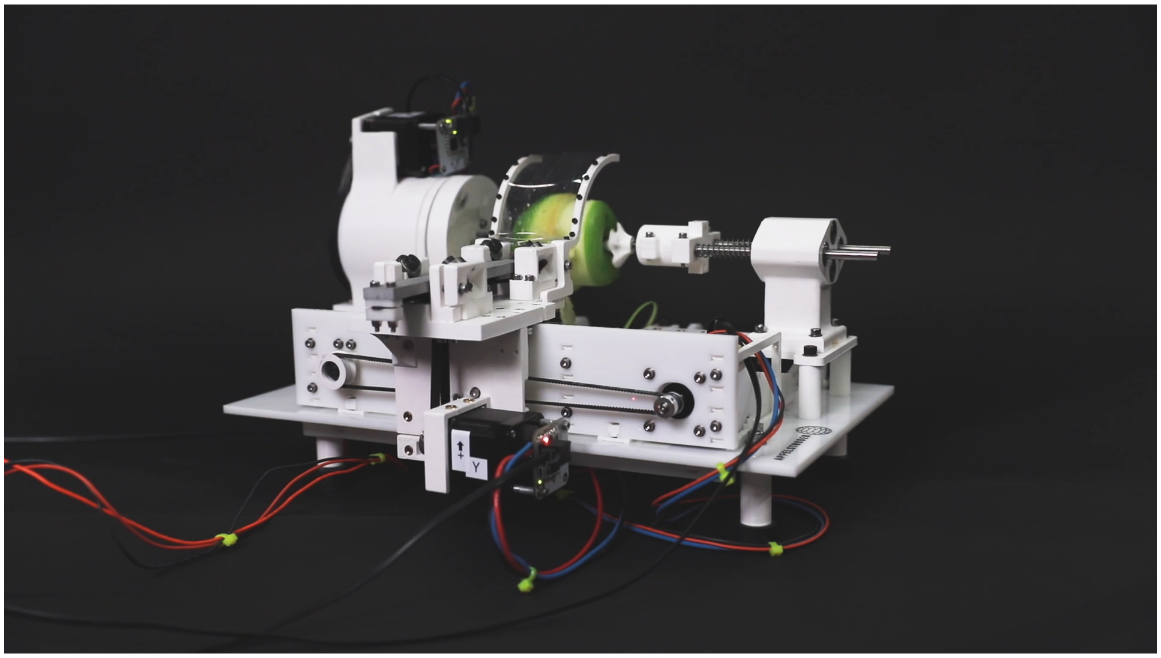


Figure 6-9: A toy lathe, developed over the course of one week by novices using the system architecture described in this thesis.

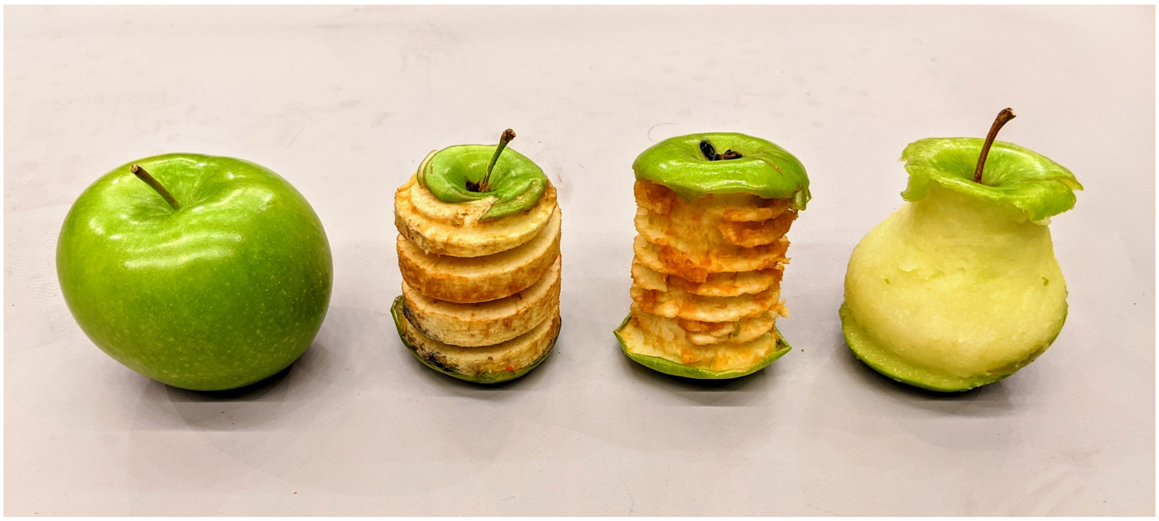


Figure 6-10: Apples 'machined' with a Distributed Dataflow Controller.

6.4 Evaluation

Given these outputs, I can discuss the successes and shortcomings of this system architecture. Earlier, I outlined a timing-based evaluation of system performance, but it is worth taking a moment now to discuss some of the purely architectural aspects of the scheme, in relation to frustrations I described in the introduction.

First, I took up issue with the static hardware configuration of many existing controllers (section 1.3.1). This prevents most controllers from being readily extended, modified, and often prevents their core competencies from being re-used in new systems. To this point, I can claim some success. In this chapter, I demonstrate two machines of very different nature, each having been developed with a common set of components that are simply configured on different networks, with different virtual graphs between them.

The architecture also succesfully extends execution of tasks that are often handled within singular computing environments across multiple computing environments, and shows that the use of virtual dataflow contexts within networks of physical dataflow allows a cohesion between the two models that eliminates the necessity of switching representations at the boundaries between networked elements, and between the high- and low-level computing regimes of interface, planning, and the microcontroller respectively. This capability - to develop controllers that are collections of modular devices - is not particular to this thesis; as [33] and [30] worked to develop object-oriented networked control architectures, and many networked industrial control systems exist. The particular contribution of this thesis is to bring modular hardware together with modular software under one system representation, meaning that these object-oriented systems can be entirely configured without re-writing any codes: rather, we reconfigure graphs of code and and graphs of hardware under one scheme.

I also expressed concern about the black-box nature of many controllers: that their softwares are statically configured and opaque in operation. This architecture can be explored and edited by end users, using the tool from chapter 4 to do so. This tool is 'alive' - meaning that it does not use forward-declaration of systems that we then impose on particular hardwares: instead, graphs are developed in dialogue with the computing environments - the VDEs - where they are embedded, using an MVC. Indeed, an unintentional result of the architecture is that systems development is also collapsed with system use: the tool from chapter 4 is, in many cases, a suitable operational interface as well. This is a pleasing result: the internal operation of machines developed under this architecture are readily discoverable by curious operators. However, the basic unit - the hunk - is only a descriptive wrapper on a block of code, the block itself is often obscured from the user. As a system designer, I can make every attempt to name and type these hunks such that users might properly infer their operation. I can work to develop only small, fundamental types of them, but their exact function (without a careful reading of the source code) can only be properly ascertained via discovery. In essence, rather than monolithic black boxes, I have constructed intricate networks of smaller black boxes. I hope the reader can appreciate that this is still a step forward, however, it is perhaps not the leap we had hoped for.

The last aspect had to do with the representations currently used to describe machine operation, in particular the GCodes discussed in section 1.3.2. Here, too, I can claim some succcess. In the machines developed here, control flows never pass through cumbersome representations like G Code. Rather, every component in the stack between our high-level representations of part geometry, down to the very low-level represenations of machine motion, is visible, and data routes between them are clearly delineated types that flow according to a unified flow control scheme.

Chapter 7

Conclusions

7.1 Libraries of Physical Resources

In the first chapter, I discussed a difference between the development of hardware and the development of software, the latter of which is greatly eased by the fact that existing codes can be included in application specific projects via openly available software libraries. This is because the underlying systems used to develop both are the same: processors with identical instruction sets, and hardware abstracted by an operating system. I discussed how the same kind of unifying abstraction is not available for physical systems, and found the root of this problem in the heterogeneity of the computing resources used for hardware control: microcontrollers, and their unique and varied peripherals. It is precisely for their specialized functions that we use microcontrollers in practice: an attempt to standardize their outputs with a unifying software abstraction would defeat the point. Besides, microcontrollers are integrated directly with physical hardware: even instances of the same make and model of microcontroller are integrated across a wide set of physical hardwares: motor drivers, sensors, power controllers, etc.

Rather than developing abstract software libraries that can cross-compile onto inherently specialized systems, as is done in the Arduino project, this thesis re-casts the devices themselves as participants in application assembly, by wrapping specialized VDEs in an MVC interface. It does not standardize code, but standardizes code assembly in a dataflow model that mimics the very networks which connect various hardware resources. By allowing any given VDE to self-report its internal reconfigurability, capability, and interface, I consider the device itself as a library of physical function packaged with a matched library of software and computing resource to manipulate that function. This means that systems integrators do not need to download or compile any new software during integration: rather, they can simply connect that physical resource to an existing network of control, and query its MVC interface. Here, adding hardware to a project is akin to adding a software library to a program.

7.2 Future Work

7.2.1 End to End Interoperability

What this thesis doesn't mention overtly, but is a core assumption, is that somewhere in the loop is an intelligent creature devising and assembling controllers. It seeks to deliver an architecture that can expose the right resources and descriptions to these creatures, granting tools that allow low level abstractions to be coherently developed into globally expressive devices. The work seeks to do so without making assumptions about the task that assemblers are trying to achieve, following the end-to-end principle to impose constraints only in core tasks that any computing system must achieve: it organizes flow-controlled execution and data typing, alongside a small set of MVC messages. Beyond this, complexity is left at the edges of the architecture. The internet flourished on the same principle: that inter-networking should be accomplished with only a small set of requirements, and all other system function be relegated to the highest levels of application. In this way, the internet could grow in complexity at the edges: new resources could be added without necessitating the removal of older components.

In a sense, the architecture presented in this thesis represents an attempt to add to end-to-end inter-networking the caveat of inter-operability: an assumption that the system globally executes some program. It does so while maintaining an adherence to the end-to-end principle, and proposes that the natural representation for networked programs is dataflow: an execution model that is already based on data transport. The applications and examples explored in this thesis are focused on equipment used for digital fabrication, but this is by no means inherent to the architecture. Discovering the usefulness of distributed dataflow control in a wider set of applications is an exciting future direction, and might include the operation of building control systems, avionics, automobile sensor and actuator networks, environmental sensing, energy generation or even entire grids, etc.

7.2.2 Compound Data Types

Successful data typing is a core requirement of system interoperability: computed outcomes from some control participants must be readily interpreted by others. If systems are to include many different types of computing, it is only where their data inputs and outputs meet the network - at data serialization - that strictly defined data representations must be used. The distributed dataflow controllers in this thesis basically propose that if typing and execution (with flow control) are standardized, almost everything else can be left to the particulars of each node, under the end-to-end principle. However, in order to handle more complex tasks I often want to manipulate compound data types. The architecture does not currently include methods to define these compound types or to serialize them between devices: developing a more succinct data typing, serialization and naming scheme is also first order work, moving forward.

7.2.3 Closing Loops

G Code, mentioned in the opening chapter, describes process outcomes rather than process control. This limits the scope of machine control to predetermined outcomes, and assumes that machines are unintelligent feed-forward devices. This thesis shows how we might use a representation of algorithm, not outcome, as a universal tool across manufacturing processes. It demonstrates that this representation can capture systems which encorporate low-level control (motors and sensors) with high-level algorithm (like computer vision), but does not go very far to develop other forms of algorithmic control. Future work should focus on using this algorithmic representation of machine control to learn how manufacturing controllers might use low level data to intelligently plan global operation.

One intuition is that this can be done hierarchichally, by first closing loops around motor controllers, and then around machine-space accelerations, and then coupling motion with process controllers for milling, 3D printing and the like. Constraints discovered in lower layers (motor torques, power limits, etc) can constrain parameter spaces in higher levels, and distributed computing at each level could be used to learn the particular complexities of each layer in computing that is local to that layer, as well as on appropriate time scales: control loops operating hardware might be learned within seconds, while machine-scale control parameters and configurations discovered over minutes, process-scale parameters learned over hours of practice. At the highest level, process constraints discovered in machine control might constrain or inform design choices made in CAD softwares - learning a kind of design-for-manufacturing model specific to each machine; this would be tantamount to developing a machine intuition for manufacturing. To close this loop, we might then use these intuitions as we design and develop new machines and new processes.

7.3 The Protocol is the Platform

One of the early promises of Digital Fabrication was the supposition that it would enable us to build a distributed manufacturing economy: individuals or small communities with '3D Printers' or similar, small manufactories[1] would form a vast network of physical output devices that could alter the current landscape of globally specialized top-down manufacturing. A hope emerged that we might be able to use locally sourced materials to regionally fabricate designs developed and refined globally in the

[1] '3D Printing' quickly became the media's favorite noun to encompass the wider set of direct-write Digital Fabrication equipment: Milling Machines, Laser Cutters, etc...

open source. With localized manufacturing, we could close the material loop, reducing waste, and close the gap between have- and have-not, by releasing the agency of manufacturing to the masses [16]. This dream is predicated on the notion that digital fabrication should be easily translated into digital manufacturing: that prototyping could scale into production, meaning that small designers and manufacturers should be able to bring their products to market without engaging in major capital investment.

The reality has been more complex: global supply chains can employ specialized processes that produce more advanced artefacts for lower costs than standalone, generalized machines like the desktop 3D Printer or CNC Mill. Almost none of the objects we find in everyday use is manufactured using only one process; even the humble disposable coffee cup is manufactured in about five steps that span a paper mill, specialized coating and forming machinery, and is capped with a mass-produced thermoformed lid. A ballpoint pen might integrate three injection molded components of varying plastics, with an extruded tube to carry ink, a finely machined tip and precisely ground ball-point. An increasingly efficient global distribution system has lowered, not raised, the cost of integrating long, fragile supply chains across borders and oceans.

If our dream of an open sourced, localized manufacturing economy is to thrive, system developers need to look beyond standalone 'desktop' devices towards integrated manufacturing chains that span multiple processes, and to open platforms for machine development. We need to radically change the way we develop equipment in the open source, to include a broader and more specialized set of processes, and to vastly reduce the time-to-build for digital fabrication equipment itself.

Fourtunately, almost all of the unique processes we hope to develop involve similar sets of core competencies. Each and every one involves some span between motor control, sensor acquisition, and a slightly broader set of process equipment: spindles for milling, heating and cooling devices for thermoplastic manufacturing, cutting tools for flat stocks, etc. Across the board these physical devices need to be integrated into controllers that receive input from some planning algorithms that orchestrate machine movement, and those planning algorithms need to receive input from some design softwares, or otherwise developed part descriptions. Each process requires some interface with human operators, system configurators, and the like.

By developing specialized process modules in the open source, and standardizing an interface layer between them, we could develop a wide variety of unique equipment that are each composed of varied collections of these modules. The overarching goal for this thesis was to develop a systems assembly platform that could serve as the assembly layer between these modules. It carried on work begun by Nadya Peek[33] and Ilan Moyer[30] to build controllers that are virtual in nature, and Dr. Gershenfeld's work to develop a dataflow software tool for digital fabrication workflows [18]. By embedding virtual dataflow computing into hardware dataflow networks, the thesis collapses heterogenous systems into cohesive graph processing networks. By aligning this goal with the end-to-end principle, the architecture distils into a set of protocols that allow new modular elements to be added to the system without any complexity being added to the system itself, promoting open development of new resources at the edges. By developing a graphic tool and technique for remote editing of virtual graphs, it allows novices to approach the development or extension of machine controllers with limited or no prior knowledge of machine systems or programming.

Many individuals are hard at work in the open source developing incredible tools for motor control, sensor acquisition, building automation, and scientific apparatus. However, we lack a framework that allows rapid assembly of these components into application specific systems. Industrial processes are massively complex, and recapturing years of past practice here in a democratized fashion will take a distributed effort. By collecting the work contained in this thesis into a coherent and readily implemented protocol, it is my hope that we might be able to collectively begin integrating this broad set of unique, heterogenous modules, reclaiming a distributed means of production on an open platform.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Massimo Banzi and Michael Shiloh. *Getting started with Arduino: the open source electronics prototyping platform.* Maker Media, Inc., 2014.

[3] Ramón Barber, M Horra, and Jonathan Crespo. Control practices using simulink with arduino as low cost hardware. *IFAC Proceedings Volumes*, 46(17):250–255, 2013.

[4] Rick Bitter, Taqi Mohiuddin, and Matt Nawrocki. *LabVIEW: Advanced programming techniques.* Crc Press, 2017.

[5] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[6] Sam Calisch, Neil Gershenfeld, and Jake Read. Ring oscillators. https://pub.pages.cba.mit.edu/ring/, 2019. [Online; accessed 30-July-2019].

[7] Stuart Cheshire and Mary Baker. Consistent overhead byte stuffing. *IEEE/ACM Transactions on networking*, 7(2):159–172, 1999.

[8] James B Dabney and Thomas L Harman. *Mastering simulink.* Pearson, 2004.

[9] McNeel David Rutten and Associates. Grasshopper 3d. https://www.grasshopper3d.com/, 2019. [Online; accessed 30-July-2019].

[10] John D Day and Hubert Zimmermann. The osi reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[11] John Deacon. Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.

[12] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[13] Jack B Dennis. Data flow supercomputers. *Computer*, (11):48–56, 1980.

[14] Brendan Galloway and Gerhard P Hancke. Introduction to industrial control networks. *IEEE Communications surveys & tutorials*, 15(2):860–880, 2012.

[15] Neil Gershenfeld and Danny Cohen. Internet 0: Interdevice internetworking-end-to-end modulation for embedded networks. *IEEE Circuits and Devices Magazine*, 22(5):48–55, 2006.

[16] Neil Gershenfeld, Alan Gershenfeld, and Joel Cutcher-Gershenfeld. *Designing reality: How to survive and thrive in the third digital revolution*. Basic Books, 2017.

[17] Neil Gershenfeld, Raffi Krikorian, and Danny Cohen. The internet of things. *Scientific American*, 291(4):76–81, 2004.

[18] Neil Gershenfeld and Nadya Meile Peek. Mods: Browser-based rapid prototyping workflow composition.

[19] Erin Griffith. Why do startups fail? because hardware is hard. *Wired*, 2017.

[20] Paul Horowitz and Winfield Hill. The art of electronics 3rd edition. 2015.

[21] Gary W Johnson. *LabVIEW graphical programming*. Tata McGraw-Hill Education, 1997.

[22] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

[23] Benjamin G. Katz. A low cost modular actuator for dynamic robots. Master's thesis, Massachusetts Institute of Technology, 2018.

[24] Eric Keller and Jennifer Rexford. The" platform as a service" model for networking. *INM/WREN*, 10:95–108, 2010.

[25] Dong-Il Kim, Jin-Il Song, and Sungkwun Kim. Dependence of machining accuracy on acceleration/deceleration and interpolation methods in cnc machine tools. In *Proceedings of 1994 IEEE Industry Applications Society Annual Meeting*, volume 2, pages 1898–1905. IEEE, 1994.

[26] Agus Kurniawan. *Getting Started with Matlab Simulink and Raspberry PI*. PE Press, 2013.

[27] Sylvain Lavernhe, Christophe Tournier, and Claire Lartigue. Optimization of 5-axis high-speed machining using a surface based approach. *Computer-Aided Design*, 40(10-11):1015–1023, 2008.

[28] Kyung Chang Lee and Suk Lee. Performance evaluation of switched ethernet for real-time industrial communications. *Computer standards & interfaces*, 24(5):411–423, 2002.

[29] Ilan Ellison Moyer. Core xy, 2012.

[30] Ilan Ellison Moyer. A gestalt framework for virtual machine control of automated tools. Master's thesis, Massachusetts Institute of Technology, 2013.

[31] James R Moyne and Dawn M Tilbury. The emergence of industrial control networks for manufacturing control, diagnostics, and safety data. *Proceedings of the IEEE*, 95(1):29–47, 2007.

[32] Bruce Jay Nelson. Remote procedure call. *Palo Alto Research Center*, 1981.

[33] Nadya Meile Peek. *Making machines that make: object-oriented hardware meets object-oriented software.* PhD thesis, Massachusetts Institute of Technology, 2016.

[34] Miller S Puckette et al. Pure data. In *ICMC*, 1997.

[35] Johnny Russell. Rambo. https://reprap.org/wiki/Rambo, 2019. [Online; accessed 30-July-2019].

[36] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *Technology*, 100:0661, 1984.

[37] Hans-Christoph Steiner. Firmata: Towards making microcontrollers act like extensions of the computer. In *NIME*, pages 125–130, 2009.

[38] Neal Stephenson. *The Diamond Age: Or, A Young Lady's Illustrated Primer*. Bantam Spectra, 1995.

[39] Think3dPrint3d. Duet. https://reprap.org/wiki/Duet, 2019. [Online; accessed 30-July-2019].

[40] Qi Tian and Michael N Huhns. Algorithms for subpixel registration. *Computer Vision, Graphics, and Image Processing*, 35(2):220–233, 1986.

[41] Arthur Wolf. Smoothieboard. https://reprap.org/wiki/Smoothieboard, 2019. [Online; accessed 30-July-2019].