Physical-Digital Programming

Jasper Tran O'Leary

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2024

Reading Commitee:

Nadya Peek, Chair

Jeffrey Heer

Jennifer Mankoff

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Physical-Digital Programming

Jasper Tran O'Leary

Chair of the Supervisory Committee:
Nadya Peek
Human Centered Design and Engineering
Paul G. Allen School of Computer Science & Engineering

While digital fabrication technology—using computer-controlled machines to create physical artifacts—has existed for over half a century in factories and machine shops, desktop-class tools have made it accessible to new practitioners across diverse domains. Pioneering practitioners in science, art, and engineering are developing experimental, domain-specific manufacturing workflows that differ from conventional fabrication practice. However, today's software tools prioritize conventional workflows and delay the development of new ones. Despite the "digital" in digital fabrication, experimental fabricators do not benefit from the affordances of writing code, namely, the ability to reason about, experiment with, and reproduce the work of others.

In this dissertation, I argue for elevating the subordinate status of code in digital fabrication by framing novel workflow development as full-fledged programming, not just as program execution. In this physical-digital programming paradigm, the physical contingencies of machines and materials are represented in a programming language, not left unwritten. To prototype physical-digital reasoning, I invented a *machine grammar* that lets fabricators denote domain-specific rules governing machine choice and usage. For visual reasoning, I developed *a method for generating*

application-specific visualizations of a given machine toolpath. To encourage experimentation and testing, I built *a programming environment* where fabricators develop entire manufacturing workflows as computational notebook programs. Finally, to aid in the reproducibility of workflows-as-programs, I added *a library* that lets fabricators interface with existing computer-aided design and manufacturing software, denote manual steps via an augmented reality machine interface, and write assertions in code for quality control.

Though physical-digital programming primarily aims to help a nascent base of experimental fabricators, it also carries implications for what programming itself entails and for society's relationship with machines. By placing direct engagement of physical automation tasks in the same medium as programmatic control, I take a first step beyond today's dominant computational paradigm, where data come from elsewhere, to a more sustainable computing future, where care of machines, materials, and fabricators is integral to programming itself.

Contents

Acknowledgments

Finishing this PhD has been a long and difficult ordeal, and I would not have even come close to getting where I am today without the tireless mentorship, camaraderie, and care that others have shown me.

First, thank you to my incredible doctoral advisor, Nadya Peek. Thank you for taking me as your first student back in 2018, when I was a first-year PhD student stricken with an intellectual identity crisis, and for sticking with me non-stop until now. You have given me the resolve to stand at the edge of human knowledge for year after year, failure after failure, until I found answers. Your relentless commitment to technical excellence—in building and maintaining machines, in understanding physical-digital systems underneath the hood, and engaging other fabricators in and beyond the academy—have given me an understanding of technology and of society that I would never have gained anywhere else. Thank you also for giving me the courage to speak the truth and to draw attention to perspectives that are overlooked. I am inspired to bring the same drive for technical expertise and justice you have shown me to my future career.

Thank you to the rest of my doctoral committee, including Rastislav Bodík, Jeffrey Heer, Eric Klavins, and Jennifer Mankoff. Jen, thank you so much for mentoring me early on when I was in graduate school and helping me grow into a researcher who could meaningfully engage with digital fabrication among a larger community of researchers. Jeff, thank you for showing me the value of data visualization in technical communication. Your work on the future of scholarly communication has greatly shaped my own work with Tandem, and has made committed to reproducible sharing of digital fabrication knowledge. Eric, thank you for all of your insight on CNC practice and on lab automation languages and systems. Finally, a great deal of thanks goes to Ras, as this entire dissertation really began in his class on domain-specific languages in Winter 2019. Thank you, Ras, for showing me that programming languages are not only about correctness, but also about elegant abstractions that empower programmers to think in fresh ways about the problems that matter most to them—even if, back in 2019, it was only about stacking cardboard machine parts together.

Thank you to all the other professors at the University of Washington who have helped me succeed in graduate school. Thank you to Xu Chen, Jon Froehlich, Zachary Tatlock for serving on my qualifying and general exam committees, and for your mentorship and insight. Thank you to Adriana Schulz for teaching me the bread and butter of computational techniques for approaching fabrication problems. Thank you to Daniela Rosner for being the first person who advocated for me to be here at the University of Washington and for instilling in me concern for care, maintenance, and repair of technical objects. Thank you to Amy Ko for helping me dive deep into the transformative power of code on people's lives. Thank you to Jennifer Turns and David Ribes for giving me the skills to reflect and think critically about epistemologies and lineages of research; it is your teaching that has transformed this dissertation from a collection of papers into an argument for one part of a just future of computing. Thank you also to all the professors outside the University of Washington who have uplifted and supported me, especially Sarah Chasins, Elena Glassman, Andrew Head, Sean Follmer, Stefanie Mueller, Jennifer Jacobs, Pedro Lopes, Cyrus Omar, Thijs Roumen, Valkyrie Savage, Oliver Schneider, Cynthia Sung, and Emily Whiting.

This dissertation was inspired by all of my labmates in Machine Agency who have provided enduring support over the years and have pushed my thinking outside disciplinary boundaries.

Thank you to Hannah Twigg-Smith for being there with me from nearly the start and for teaching me so much about relentlessly advancing state of the art in web engineering. I am always inspired by your work to teach your students the joy of building their own tools, rather than accepting a reality where somebody else will build them. Thank you to Sonya Vasquez for teaching me so much about practical machine-building; while I do not think I will ever come near your level of hardware hacking skills, just watching you build has taught me so much. Thank you to Gina Clepper, Kellie Dunn, Cindy Feng, Sam Ferguson, Frikk Fossdal, Orlando de Lange, Danny Leen, Danli Luo, Marita Sauerwein, Blair Subbaraman for the late-night camaraderie in the lab, the hikes to glacial lakes (and sometimes into them), and arduous journeys to and from certain academic conferences. Thank you to Matthew Sorensen for teaching me about machines and sharing your velocity planner implementation which underlies the algorithm detailed in Figure 4.5. Thank you to all my brave and hardworking mentees, including Khang Lee, Maja Ling Han, Octi Zhang, and Thrisha Ramesh; you have taught me as much about research as I have tried to teach you. Finally, thank you to Gabrielle Benabdallah, whose cogent, profound thinking has opened me up to entire schools of thought, most notably Gilbert Simondon's technical mentality. Your understanding of philosophical works and debates has helped me connect my own work to centuries-long conversations about the relationship between humans and machines.

I would like to thank the Alfred P. Sloan foundation for providing financial support for my research; your commitment to a society that engages directly with science and engineering has given me freedom to think deeply and has motivated many of the technical decisions in my work. My gratitude extends as well to the anonymous participants of my user studies and to the anonymous peer reviews of my papers.

While I have been fortunate to have excellent peers during the PhD, I must specifically name three women who have provided me with crucial mentorship. Thank you to Chandrakana Nandi for working with me as we explored the intersection of digital fabrication and programming languages. Your thoughtful explanations transformed initially intimidating concepts in programming languages research (like those “giant fractions”) into ordinary, accessible, and useful ones. Eunice Jun, it was an absolute joy to think through and write a paper with you. Your drive to empower statistical non-experts through the affordances of programming languages continues to inspire my own work. Finally, thank you to Ștefania Drugă, an incredible friend who has helped me navigate the ever-shifting landscapes of: trends in artificial intelligence research, the job market, and life lived to its fullest.

Thank you to many other colleagues in the PhD who have inspired me over the years, especially Liang He, Chris Geeng, Miranda Wei, Yasaman Sefidgar, Naveena Karusala, Esther Jang, Galen Weld, Joe Breda, William Agnew, Anna Spiro, Madeleine Grunde-McLaughlin, Kentrell Owens, Josh Horowitz, Matthew Conlen, Alex Kale, Dominik Moritz, Cynthia Bennett, Kristin Dew, Kylie Sobel, Samantha Kolovson, Talia Ringer, Gus Smith, Ben Kushigian, Remy Wang, Amy Zhu, and Ruotong Wang. Thank you especially to Tracy Tran, a fellow curious mind from the very beginning. Special thanks goes to Max Willsey for sharing his dissertation’s LaTeX template which forms the foundation of the one used for this document.

Thank you also to my academic peers and friends in the greater academic community, including Dave Preiss, Jake Read, Leo McElroy, Mustafa Doğa Doğan, Martin Nisser, Ticha Sethapakdi, Harrison Goldstein, Lea Albaugh, Parker Ziegler, Justin Lubin, Jeremy Warner, Eldon Schoop, Eric Rawn, Katherine Song, Jane E, Parastoo Abtahi, Eric Gonzalez, Alexa Siu, Evan Strasnick, Michelle Lee, Mert Toka, Samuelle Bourgault, and Josh Urban Davis. You all have paid attention to my own work and pushed it further—it was a pleasure to “grow up” with you all. Special thanks to Mackenzie Leake, who kindly and candidly helped me navigate my post-doctoral options.

A huge thank-you goes to the academic staff of the Department of Computer Science & Engineering and the Department of Human-Centered Design and Engineering. In particular, thank you to Elise de Goede Dorough; without you, I would not have had the opportunity to continue study in computer science. Thank you for helping create one of the best environments possible in which I could have completed a PhD. Thank you also to Allen Lee, who went above and beyond in helping me navigate many financial constraints and bureaucracy so that I could focus on my studies.

Thank you to United Auto Workers Local 4121 for helping survive as a graduate student when, through the lens of social power, the odds were heavily stacked against me. This union's sacrifices have ensured that I could have adequate wages, healthcare, and protection from discrimination while completing my PhD. I am also grateful to have been supported by the Research University Alliance program in connecting with scholars at other institutions.

In-line with my calling for open-source fabrication, thank you to all the software developers and maintainers of open-source software that have been foundational to all the software that I have written in graduate school. Nearly every single software tool I used has been built and maintained, almost always for free, by and incredible community whose really should be treated as a crucial infrastructure by those in positions of power.

I cannot extend enough gratitude to Sandy Kaplan, who provided editing and feedback over the years on nearly all the text that made it into this dissertation. Writing is a technology that restructures thought, according to Walter Ong. Through writing, Sandy has helped me fundamentally restructure my thinking for the better. Her care in how to approach arguments, cut unnecessary wording, and phrase my points concisely has fundamentally improved not only my research, but has also made writing a tool to think through and solve problems more generally.

Right before my time at the University of Washington, I was fortunate to intern at Adobe Research, where my mentors gave me foundational skills that have served me for the whole of my research career. Thank you to Morgan Dixon, Holger Winnemöller, Wilmot Li, and Mira Dontcheva for your mentorship. Special thanks to Wil for teaching me how to write research papers, and to Mira for teaching me how to speak about research to an audience. Thank you to my co-interns Yea-Seul Kim, Ailie Fraser, Daniel Epstein, and Amanda Swearngin for teaching me the ins and outs of research during my formative years, and for hiking to the top of a mountain with me when I had not yet experienced the beauty of the Pacific Northwest. Mira-cles for life!

Before that, I was nurtured by incredible mentors and colleagues at the University of California, Berkeley, without whom I would never have tried my hand at research at all. Most of all, thank you to César Torres and Eric Paulos for taking a chance on me as an undergraduate who absolutely nothing about computer science research. César, your mentorship and guidance has been the foundation of my own research career to this day. I have always strived to maintain the same care of explanation, diversity of approaches, and fluidity between science and art that you have taught me when I do research or mentor other researchers. Thank you to the rest of the Hybrid Ecologies Lab who all supported me, including Tim Campbell, Rundong Kevin Tian, Christine Dierk, Molly Nicholas, Sarah Sterman, and Chris Myers. Thank you to Christopher Hunn for suggesting I try undergraduate research, which got me started on my path. Thank you to Ann DiFruscia, Lars Rohrbach, and the rest of the EECS Helpdesk, and thank you to Andy Carle, Björn Hartmann, and Abigail de Kosnik. Thank you to everybody at the Berkeley Student Cooperative for helping me afford my undergraduate studies, and for showing me see what it really means to make democracy work up close.

A huge round of thanks goes to my friends who have supported me in both wonderful and difficult times in my PhD, especially Jingyi Li, Pema Maretich, Akhila Raju, Husna Hadi, Jeremy Warner, Jane E, Ștefania Drugă, Cathy Yuan, and Orlando de Lange. Thank you to all of my friends at fellow housemates at 1625U for the camaraderie and cooperativism post-undergrad, particularly Ethan Weinberger, Aishwarya Mandyam, Philip Garrison, Jiaqi Li, Aidan Mellon-Reyes, Sophie Haeuber, and Michael Tappel—but especially Summer Wu and Lionel Jin on December 25th, 2020.

When I started my PhD, I took a "learn to row" class, not yet knowing that rowing would become one of the most important parts of my life. Rowing was instrumental for me in finishing grad school as a way to recollect myself mentally and physically—and to get myself in the best shape of my life. Thank you to Teddi McGuire for being the first to help me into a rowing shell, and thank you to all my other coaches and teachers, including Andrew Rees, Theresa Batty, Hugh Lade, Joel Osborn, Amy Hildebrandt, Alex Mazick, Rachel Dooley, and Rachel Le Mieux. Thank you to the rest of my rowing family, including Alex Lund, Bennett Shultz, Rachel Wong, Allison Thomas, Kelly Johnson, Megan Northey, Annie Grieff, Kate Case, Meghan Ricci, Samantha Petrelli, Matt Curran, Eleanna Bez, Maggie Coit, Sheamin Kim, Sam Nutt, Amke Nimmrich, Katia Meuleman, Mychal Richardson, Matt Marshall, Tariq Omar, Darren Montgomery, Ted Leslie, Kyle Foote, and Nathan Haase. Special thanks to Alex Weatbrook for all the pitch-black hours spent in the pair, Cecilia Krause for your leadership, Pati Casebolt for showing me how the spiritual side of rowing could help me through the dark pandemic times, and KC Dietz for encouraging me to step into the single. Finally, I will never forget my good friends and dream team in the Head of the Lake mixed eight: Killian Ballard, Magdalene Vidal, Maya Hartleben, Sophie Blackburn, Daniel Knorr, Charlie Inman, Nick Sweet, and Julia White. You all have given me the power push my body and mind to their limits before the sun even rises, a profound understanding of how many bodies can move gracefully as one, and an *ever so slight* taste for pain.

Among all of my friends—rowing and otherwise—thank you to Stella Willoughby and Jeffrey Miles for being the ones I could always turn to. Stella, I would not have made it through the pandemic years without you. Here's to many more years together in the pair!

Last and certainly not least, I would like to thank my family who have given me privilege of being able to grow and study in the world. Thank you to my aunts—Lily Jolly, Theresa Tran, Audrey Le—for being my advocates and my examplars. Thank you to Millie and Brian Tran-Rosenthal for your encouragement and your examples of excellence in writing. Thank you to Gabriela and Adolfo Capestany, Rob Hallock Ava Jolly. Thank you to my grandmother, Truc Mai Nguyen, for all of your sacrifices and for always affirming the value of education—all of this work is for you.

Thank you to my brother, Julian O'Leary, for always having my back through all the years; you inspire me every day with your deep knowledge and mastery of everything you set your mind to. Thank you to my parents, Veronica Tran-O'Leary and Stephen O'Leary, for all the love and for all the sacrifices you made to raise me and give me the opportunity to study. You both have instilled in me both a deep respect for others and an innate curiosity of how the world works, which are the foundation for all of this work. Finally, thank you to my partner, Brett Sagan, for supporting me with endless patience during some of the most challenging times of the PhD. I am so grateful to finish this work with you by my side.

Dedication

For Bà Ngoại.

Chapter 1

Introduction

"By distancing thought, alienating it from its original habitat in sounded words, writing raises consciousness. Alienation from a natural milieu can be good for us and indeed is in many ways essential for fuller human life. To live and to understand fully, we need not only proximity but also distance. This writing provides for, thereby accelerating the evolution of consciousness as nothing else before it does."

— Walter Ong, *Writing Is a Technology that Restructures Thought* [Ong92]

Watching carefully as an open-source machine for programmable motion slowly extrudes thin traces of electrode, a materials scientist prototypes the design of a microscale 3D printable battery. While conventionally produced, flat-electrode batteries trade off power and energy lifetime, new volumetric battery architectures could overcome this fundamental limitation. However, the performance of 3D printed batteries depends on tightly coupled materials and fabrication processes. Along with theoretical knowledge, the materials scientist must experiment with multiple designs and fabrication parameters before arriving at a feasible process.

Elsewhere, in a pottery studio, artists interactively 3D print a clay jar with audio data embedded as small "blips" on the surface. The clay printer steadily encodes the resulting "listening cup" to serve as a memento of a conversation, song, or other snippet of sound. One of the artists, a lifelong ceramicist attuned to the feel and behavior of clay, repeatedly adjusts the clay extrusion rate from its auger, switches back to machine instructions stored in Excel spreadsheets to edit the machine's pause timing, and places her hand under overhangs that look like they might collapse.

Finally, researchers in a mechanical engineering laboratory write code to generate impact-protection lattice designs for 3D printing. With so many potential designs, the researchers set up a small factory of 3D printers to parallelize the printing process, complete with robotic arms removing finished samples from printer beds. Yet, testing the dynamic impact resistances of the lattice samples requires an expensive and time-consuming process. To overcome this constraint, the researchers devise a novel process: test most of the lattices on cheaper static testing machines, then infer dynamic responses of most designs using a machine learning model with many static and few dynamic tests.

* * *

In all three scenarios, practitioners from diverse disciplines and with diverse uses cases are engaging in *digital fabrication*, i.e., using a computer to control a 3D printer, a computer numerical control (CNC) mill, or pipetting machine[1] to create a physical artifact. Yet, how these practitioners work bears little resemblance to conventional digital fabrication practices I describe below.

For the last half-century, digital fabrication has been hardly digital, with fabricators themselves performing the automation that one would expect a computer to do. In part due to inaccessible methods of controlling machines, CNC tools have been mostly limited to industrial contexts, like factories and large-scale manufacturing sites. Here, fabricators interact with machines,

[1]Lab automation machines would typically fall outside the scope of digital fabrication because they do not create a new artifact, but I consider it within scope because they manipulate matter through programmed actions. Moreover, the term "digital fabrication" itself is an umbrella term, which practitioners do not often use by default.

typically lathes, mills, and, more recently, additive machines like 3D printers, through a fairly rigid, unidirectional pipeline that echoes the batch programming paradigm of the 1950s in which CNC was first developed.

- First, using computer-aided design (CAD) software, a fabricator develops a 3D representation of the object they want to produce.
- Next, the fabricator plans the *toolpaths*, or motions of the machine's tool to produce the artifact, using computer-aided manufacturing (CAM) software.[2]
- Finally, the user transfers output of CAM software, i.e., low-level *machine instructions*, also called "codes," to CNC control software.

In this *canonical workflow* [TSTOP21], each step of the process typically constitutes its own discrete software application. Here, I define a *workflow* to be a series of steps in a fabrication process leading to a manufactured object; these steps span both digital code and physical processes.

Since the 2010s, CNC capability has expanded beyond industrial and machine shop settings into scientific laboratories, artist studios, academic makerspaces, and even home settings [BBPJ21]. Fueled by efforts such as the Maker Movement, Fab Academy, and academic interest, desktop-class fabrication machines have become more available and accessible to potential fabricators from many walks of life. In fact, Fab Academy founder Neil Gershenfeld likened this increased ubiquity of fabrication machines—*personal fabrication* [Ger07]—to Mark Weiser's vision of personal computing via increasingly ubiquitous and personal computers [Wei99]. Though fabrication machines are becoming more personal, the tools available to control them have not kept pace. They remain rigid, focused on helping fabricators create faithful replications of 3D models, not on enabling the full range of expressibility needed for innovative domain-specific use.

[2] In the case of 3D printing, fabricators typically encounter CAM software in the form of slicers, which plan the printer's toolpath along horizontal slices of an object. However, there is nothing inherent about slice-based approaches to 3D printing, and several projects from research and practice eschew conventional slicing techniques.

How are the aforementioned scientists, artists, and engineers actually fabricating volumetric battery electrodes, clay cups encoded with audio data, or robotically tested impact-protection lattices? In each instance, the fabricator must forgo the predictability of the canonical CNC workflow in favor of *application-specific control and experimentation.* Previously, fabricators could imagine an idealized result and faithfully "compile"[3] the object to physical form down a step-by-step pipeline.[4] Today's fabricators, however, are materials-driven, at the mercy of the physical response of machines and materials as they pioneer manufacturing processes that have never before been implemented. In essence, *what these fabricators are doing looks much less like moving down a traditional factory assembly line and much more like concocting a strange brew of programming, crafting, and experimenting.*

This dissertation addresses the *digital-physical divide* that people who build novel workflows must navigate.[5] The world around us continues to heavily favor either the physical or the digital, the former through experience gained via crafts, machine shop practices, or machine maintenance and the latter through writing code, optimizing and simulating geometries, and automating precise machine tasks. However, experimental digital fabricators must reason about *both* code and physical concerns as they iterate again and again to shape materials and designs that meet their vision while adhering to material and machine constraints. How can we transcend this physical-digital divide and instead build tools that are *physical-digital*, respecting contingencies and ways of working learned from both physical realities and computational thinking?

[3]Viewing fabrication as compilation is a relatively new lens, adopted implicitly [BM17] or coined explicitly [Nan21], though some researchers claim that this lens can unintentionally exclude how fabricators write their own software as part of a process [HBJP23]. Still, the latter paper does not define the meaning of "compilation" (e.g., preserving input language semantics versus a unidirectional compiler pipeline) in the first place.

[4]Philosopher Gilbert Simondon attributes this matter-follows-form style of thinking to the "hylomorphic schema," which traces its roots to ancient Athens [BP24]. Anthropologist Tim Ingold contrasts hylomorphism with *morphogenesis,* where a preconceived idealized form is modified throughout a design process by material constraints that the designer encounters [Ing13].

[5]A similar concept of "bits and atoms" was coined by Gershenfeld [Ger07] and was adopted by Ishii in his concept of tangible bits [IU97]. I contend that Ishii's line of thought views atoms as programmable matter—something to take form *according to* bits—ultimately privileging bits rather than granting equal footing to both.

1.1 Novel Workflow Development in Digital Fabrication

Given these goals, I turn to an established tool, programming languages, which let people write down exactly what they would like a computing machine[6] to do. When writing code, a programmer represents their thought process in a standardized computer language that provides three crucial affordances, among many others. First, programmers can read the code and *reason about* what the computing machine will do. Second, programmers can run their code to *test its performance*, making adjustments to the code as gleaned from testing. Third, code that is written down means that others—even those who are not in the programmers' immediate circle—can read, understand, and *replicate the performance* of the computing machine for themselves.

Programmers rely heavily on these affordances to write code that powers the contemporary world. Our very society hinges precisely on the act of writing code, and on writing itself: externalizing, reflecting on, accumulating and sharing our thought processes as we collaborate with the computing machine. I posit that we should extend this same written externalization of human-machine collaboration to the broader space of fabrication machines.

Unfortunately, current programming languages do not offer the same resources to physical-digital tasks that they offer to purely digital ones. There is no cohesive way to document, reason about, test, and reproduce physical-digital workflows in the same way that digital-only programmers do. Fabricators must focus so intently on getting machines to run at all that they lose sight of the overall *process* that programming makes explicit in code. *They are forced into a paradigm of digital fabrication as program execution, rather than one of fabrication as full-fledged programming.*

For example, fabricators might string together many existing GUIs and scripts to generate

[6]While we normally refer to computers simply as "computers," the reality is that, in their contemporary form, they are also machines. Anybody who writes programs in a programming language is necessarily encoding a series of steps that a central processing unit (CPU) will perform. I refer to these devices as *computing machines* in this context to emphasize that successful development of software is already predicated on a healthy human-machine relationship, albeit without the physical challenges inherent to novel workflow development.

low-level instructions, like G-code, for machine control. However, the inner workings of each GUI and script are often opaque; fabricators are only tweaking parameters—fiddling with knobs—of what others have built. Further, they must painstakingly pass changes through a series of disparate applications by repeatedly exporting and importing data down an ad hoc pipeline. Because experimental fabricators repeatedly experiment to understand machine behavior, they must laboriously repeat this process upon every iteration.

And worse still, the discoveries and expertise that fabricators develop while iterating go mostly unwritten. The crucial combinations of machine and material choices, the novel algorithmic strategies, the "warning signs" and "green lights" that a fabricator learns through experimentation are elided from the final workflow; these experimental results might be documented as a prose "how-to" guide for juggling several software tools, if written at all. The steps an experimental fabricator took to get a workflow to actually work cannot be shared with other fabricators, who are now doomed to repeat much of the unnecessary trial and error themselves. As a result, fabricators cannot build and accumulate knowledge as can digital-only programmers who read and write language.

While not all expertise gathered while developing a novel workflow can be written down, much of it can be *if it is written in an appropriate (programming) language.* Such a language should support representing *physical contingencies* that constrain and guide the work of experimental fabricators. These contingencies might include the choice of machine and material, the hardware settings on a machine, the physical performance of a machine as measured by empirical tests, different algorithms for generating geometries and toolpaths, the choice and condition of a machine's tool or end effector, ways of visualizing machine behavior, and so on. Though existing GUIs and scripts might handle some of these contingencies on their own, *they do not let a fabricator holistically lay out all of their concerns in a single environment.* In addition, when contingencies are implicit, it is difficult for fabricators to deal with interactions between several of them.

Through this lens, *I argue in this dissertation that fabrication workflow development should be framed as a programming task*, not just as program execution. In her treatise on domain-specific programming done by non-expert programmers[7], Nardi argued that programming, as opposed to purely graphical user interfaces, empowers domain-specific programmers to create code formalisms that appropriately match visual aspects of their given problem [Nar93]. Nardi pointed out that"the compactness of a textual language permits users to write complex formulas in a very small space" [Nar93, p. 86], whereas flow-based programming tools suffer from visual clutter as the complexity of the program grows. Further, as Chasins et al. write, "while both GUIs and languages are often designed around making it easy for users to say common things, a language empowers users to say uncommon things too" [CGS21].

Given the barriers, how can we aid practitioners in programming to address the myriad uncommon physical challenges that arise in experimental digital fabrication? To do so, I propose *physical-digital programming*, which I define to mean *programming that includes real-world, physical contingencies in source code.* Using physical-digital programming, physical concerns can be programmatically inspected, visualized, debugged, and/or verified. As much as possible, programming languages, environments, and techniques should capture physical concerns in-code without requiring separate documentation.

1.2 Weaving the Physical World into Language

Currently, considerable parts of workflow development are tacit,[8] either confined to personal experience or specialized to one use case only. Unlike digital-only open-source software, ex-

[7]Nardi uses the term *end-user programming* which maintains a distinction between end-users and programmers, where the former is more interested in a specific application where programming could useful and the latter is more interested in programming as a medium. I do not believe this distinction applies in the case of experimental fabrication, where required areas expertise can vary between workflows, and will hence not use the term.

[8]In the sense used by Michael Polanyi [Pol67].

perimental fabrication *lacks its own language*; as a consequence, it is difficult for fabricators to represent their workflows in a way that could be easily developed or understood by others. I argue that *programming languages in particular are uniquely suited as infrastructure for experimental fabrication, more so than simply patching on additional functionality to conventional CAD and CAM tools, building better GUIs, or writing more tutorials.*

To understand the challenges of representing the physical, I turn to programming languages research because it has spent years examining how programmers enact and reflect on computation through the medium of text. Prior research has studied how the design of programming languages can shape the thinking of programmers when they use code that causes *effects*, i.e., does things "outside the program." Because they break referential transparency [Qui60, ASS02], effects inherently threaten notions of abstraction and encapsulation in computer science as a means of controlling the complexity of programs [Lam73]. Prior research has addressed this problem through the design of languages themselves, most notably through functional programming approaches, like modeling effects as monads [PJW93], language-level effect systems [Lei14], or even coeffect systems [POM14].

However, in digital fabrication, programming languages *literally effect change in the physical world.* There are currently few to no attempts—in programming languages themselves—to conceptualize and handle the reality that executing experimental digital fabrication code will cause change in the physical world. Existing CAM software typically provides non-extendable visualizations that represent parameters typed into fields, which suffices for conventional workflows but provides little support for experimental ones. As experimental fabricators turn to code to handle even less predictable processes, they will need better ways of reasoning about, visualizing, and iterating on their workflows.

Along with work in programming languages, *computer graphics and human-computer interaction research* has more recently investigated challenges in digital fabrication [BM17]. However, it has tended to approach fabrication workflows as geometry modeling and optimization problems; as such, it has paid less attention to physical contingencies and interoperation between machines, materials, and code that fabricators encounter in their daily work. The majority of work in computational fabrication research tends to end as soon as the bulk of the digital work does, i.e., at the end of the CAD stage. Without reasoning about how fabricators make workflows work practically—both in the common cases and in the atypical microfluidic, high-throughput, and craft-based ones—computational techniques like simulation, optimization, and inverse design conceptualize digital fabrication as an idealized pipeline only; this overlooks the central challenge of fabrication: the invention of novel processes. As a result, it can be difficult for experimental fabricators to adopt software tools built as part of research projects when researchers assume that workflows are a fixed process; such tools typically do not provide "horizontal movement," or the freedom to mix and match software tools and representations, that is required for imagining and building new workflows [LRR+23].

Overall, a physical-digital programming language must define this process orientation to include machine and material considerations as well as manual manipulation. I posit three key facets of programming languages that could make fabrication workflow development easier and more expressive.

1. **Reasoning.** Having a shared vocabulary and shared ability to reason about approaches and constraints for programming or for making, modifying, and maintaining a physical object. Being able to share information about what will happen by executing code and/or running a machine under a given set of conditions, many of them unseen.

2. **Experimentation.** Receiving quick feedback, either through trying out code (e.g., a minimum working example, such as sharing a CodePen on Stack Overflow) or testing fabrication jobs.

3. **Reproducibility.** Running other people's code or fabrication workflows and modifying it, as needed. Building communities where people see what others are doing, share infrastructure, and support one another.

While the bulk of this dissertation empirically addresses these three facets, in Chapter 7 I discuss a fourth, **quality control**. Quality control refers to the ability to manage code quality and readability via test suites and development tools; in the case of fabrication, quality control concerns machine maintenance and adjustment to match requirements expressed in code, and vice versa.

1.3 Thesis Statement and Approach

Bringing together the aforementioned discussion points, this dissertation investigates the following statement.

> ***Thesis:*** *Physical-digital programming, which encapsulates physical contingencies as programming language constructs, promotes reasoning, experimentation, and reproducibility in digital fabrication workflow development.*

To support this claim, I present four system prototypes that demonstrate the potential of programming languages to aid with reasoning, experimentation, and reproducibility—even in highly experimental physical-digital problems. I briefly describe each facet of fabricator-centered programming languages in a subsection below, the system or systems built to address that facet, and resulting evaluation that supports the thesis claim.

1.3.1 Reasoning

To begin, I consider both textual and visual reasoning which each offer different types of insight.

Textual Reasoning

A fundamental requirement of programming is understanding what a piece of code *means*. "What will happen when I run this code?" is a ubiquitous and recurring question a programmer must ask themselves as they write code. While the *syntax* of a programming language dictates how programs must be composed out of symbols, the *semantics* of a language governs what those symbols actually "mean" or "do" when compiled or executed by an interpreter. Programmers must develop an intuitive understanding of a language's semantics to successfully apply it to a given problem. Moreover, programmers often write programs for domains that are highly specific and differ substantially from the basic functionality of a general-purpose programming language.

To provide the foundation for programming language semantics in digital fabrication, my collaborators and I built **Taxon**: a domain-specific language (DSL) for formally representing machine capabilities and high-level semantics of workflows (or steps a user would want to take with the machine). Taxon features a way to author *rules of thumb,* which are heuristics for selecting machines for a given task or using machines properly in a workflow, all drawing from a formal representation of machines. We evaluated Taxon by writing demonstration programs of 5 existing workflows in the literature.

Visual Reasoning

Still, 3D printers, CNC mills, and pipetting machines differ fundamentally from purely digital applications because they deal with the physical world. While in most cases programmers can assume that running a line of code will do the same thing regardless of platform, what will happen on a *machine* can range from ho-hum to entirely non-deterministic. This is because machines' behavior differs based on a variety of factors, which range from manageable to entirely unexpected, e.g., the choice of materials (including manufacturing defects, its exposure to moisture during storage, and so on), the condition of the machine (loose belts, non-level bed, worn cutting tools, clogged nozzles), physical effects of tools (overheating, chatter), hardware configurations (kinematics limits, positional offsets), and more.

With an approach like Taxon, which hinges on formalizing certain properties into a programming language and solving logical satisfaction problems, we can cover only the "basics" and not the "nitty-gritty" before our sets become intractably large. Instead, I shift this approach: we can help fabricators build models of hidden semantics by *visualizing machine behavior* in the non-common cases. For example, could we visualize where a 3D printer or CNC mill might produce unwanted artifacts within its toolpath due to vibration? Could we visualize parts of a stock surface where a laser cutter might generate too much heat due to detailing? Or, could we visualize how a viscous fluid's deposition might deviate from a designated trajectory?

To understand how to enable more experimental programming for fabrication, we turn to *exploratory data visualization,* which focuses on making sense of data *before* formulating and testing hypotheses [Tuk77]. To this end, we built a means of generating task-specific visualizations of machine behavior using machine instruction interpreters with user-defined graphical semantics, called **Toolpath Stylesheets** (TSS). TSS provide lightweight, flexible visualizations of multiple facets of a machine's behavior that capture a subset of hidden semantics between code and physical output, thus affording fabricators suites of visual checks. An individual TSS takes as input the instructions that constitute a toolpath, e.g., G-Code, along with material and machine parameters and output a case-specific visualization for vibration artifacts, heat, or viscous fluid flow.

We evaluated this approach by creating several examples of TSS using two compilation approaches: a one-to-one mapping of instructions to visual elements, and a method leveraging insights from modern data visualization techniques—in particular, the grammar of graphics. In addition, we created a proof-of-concept dashboard for visualizing toolpaths using multiple TSS, interactively exploring subsets of the toolpath, and visualizing the effect of changing kinematic limits such as maximum acceleration and velocity around corners.

1.3.2 Experimentation

Beyond code-based reasoning and visualization, experimental fabricators require a programming environment that enables quick iterations and exploratory deployments of code to the machine. Traditional CAD and CAM tools necessitate stringing together discrete programs into ad hoc pipelines. To change part of a workflow, a fabricator must make a change in one program—for example, tweaking the geometry of a model in CAD—before manually exporting and importing files down the informal chain of programs. These historical divisions between parts of a conventional fabrication process, while useful in the mid-20th century, now constrain experimentation and quick process testing [Pee16].

Following other examples of exploratory data tools, we first built **Verso**, a programming environment where fabricators can express their workflow as a single program in a specialized editor. The Verso editor provides within-code GUI widgets based on prior work in programming environments [OMB+21] to initialize machines, preview toolpaths via TSS, and perform other tasks amenable to graphical control.

To further extend this concept of workflow-as-program, we built **Imprimer**, a front-end library for building experimental fabrication workflows as computational notebooks. Imprimer allows an experimental fabricator to quickly write code to generate toolpaths, visualize toolpaths via TSS, and deploy toolpaths directly from a single computational notebook. We evaluated Imprimer by writing three example experimental workflows as notebooks: sketch-based milling, surface milling by sampling mathematical functions of two variables, and a parametric shelf generator. Further, we conducted a user study with six participants who had backgrounds in computational notebooks and/or CNC milling.

1.3.3 Reproducibility

How do experimental fabricators move from experimental code to a complete workflow that can be shared with others? In comparison, open source software relies on sharing, using, and building upon the code of others [Egh20]. It is only through being able to actually *use* other programmers' code that communities of practice can form around code in the first place. To make experimental fabrication less of an ad hoc fix and more of an infrastructure for a growing community of experimental fabricators across domains, workflows-as-programs must *work* for fabricators other than the one who built the workflow in the first place.

Getting an experimental fabrication workflow to work at all is much harder than getting a digital-only program to run, much less on somebody else's computing environment and with the machines and materials they have on hand. I defined three specific issues that make it exceptionally challenging to reproduce another fabricator's workflow—challenges that do not occur in sharing open source digital-only software. First, most workflows beyond simple experiments rely heavily on CAD and CAM applications to create and process 3D geometries and toolpaths that fabricators associate with those geometries; it is infeasible to reimplement in a text-based programming environment all the functionalities of a geometry kernel that are typical in CAD and CAM tools. Second, working with machines extends beyond the mere digital; at a minimum, fabricators must physically prepare, test, and operate the machine, e.g., CNC mill fabricators must frequently position the workpiece to be milled in proper locations on the machine's bed and localize the machine's digital state according to the physical setup. Finally, errors occur frequently in the physical world that can be difficult to trace back to code; recovering from collapsed prints, overly deep milling cuts, or improperly pipetted material can be a daunting task that involves reasoning about both physical and digital state.

To enable reproducibility, we improved the Imprimer programming environment, yielding

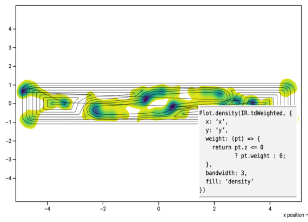
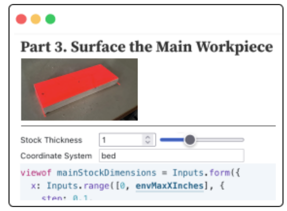
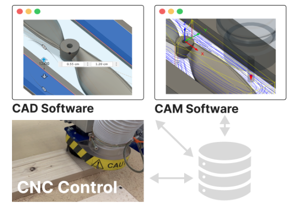
Reasoning (Textual)	*Reasoning (Visual)*	*Experimentation*	*Reproducibility*
			
Taxon	**Toolpath Stylesheets**	**Imprimer**	**Tandem**

Table 1.1: The Mapping between Facets of Physical-Digital Programming to Projects.

computational notebooks that could be more than experiments. Our result was **Tandem**, a library and associated backend for reproducing workflows that may rely on conventional CAD/CAM as well as manual interventions, the latter facilitated with a grammar for augmented-reality based guidance and verification of physical state. Tandem also lets fabricators denote potential mismatches between the physical and the digital as explicit assertions in code. Using two-sided CNC milling as an example, we demonstrate how to implement a complex workflow as a single program that can be re-run by others while supporting quality control and improving reproducibility.

Though Tandem is still a proof of concept defined for a single experimental workflow, it represents a first step in moving from open source software to open source fabrication.

1.4 Summary of Contributions and Thesis Outline

This dissertation presents the four projects mentioned above—Taxon, Toolpath Stylesheets, Imprimer, and Tandem (Table 1.1). Each project maps to one of the facets of fabricator-centered programming—reasoning, experimentation, and reproducibility—as described in the chapters detailed below.

- Chapter 2 reviews prior literature related to fabricator-centered programming languages. In particular, though many fields have addressed particular issues that arise during experimental fabrication, I observe that almost no prior work has both: (1) cast both experimental fabrication itself as a *process* to be iterated upon, and (2) explicitly represented and reasoned about that process using programming tools beyond manual trial and error.

- Chapter 3 shows a proof-of-concept language and graphical environment I envisioned, Taxon, for programming languages that represent physical-world constraints with machines and workflows. *Facet covered: reasoning (textual).*

- Chapter 4 identifies how we implemented my notion of Toolpath Stylesheets, task-specific visualizations of machine behavior compiled from machine instructions. It also discusses an early implementation of an experimental programming environment for workflows, Verso, that led to the design of Imprimer. *Facets covered: reasoning (visual) and experimentation.*

- Chapter 5, Imprimer, describes the first iteration of my concept of representing workflows as computational notebook programs. *Facet covered: experimentation.*

- Chapter 6, Tandem, encapsulates my idea of moving from computational notebooks as experiments to notebooks as reproducible processes that help fabricators share and re-implement workflows across time and space. *Facet covered: reproducibility.*

- Chapter 7 moves from the foundation of workflow-as-program discussed heretofore towards a future I envision of maintenance, repair, and deeper knowledge of manufactured and manufacturing objects, as embodied in a debugging tool for physical understanding and repair of experimental workflows, called Coxswain.

- Chapter 8 concludes the dissertation.

1.4.1 Inclusion of Previously Published Work

This dissertation contains text from four papers that have been published at scholarly venues.

- *Taxon: a Language for Formal Reasoning with Digital Fabrication Machines* [TONLP21], published at the ACM Symposium on User Interface and Software Technology (UIST) in 2021 and coauthored with Chandrakana Nandi, Khang Lee, and Nadya Peek.
- *Improving Programming for Exploratory Digital Fabrication with Inline Machine Control and Styled Toolpath Visualizations* [TOJP22], published at the ACM Symposium on Computational Fabrication (SCF) in 2022 and coauthored with Eunice Jun and Nadya Peek.
- *Imprimer: Computational Notebooks for CNC Milling* [TOBP23], published at the ACM Conference on Human Factors in Computing (CHI) in 2023 and coauthored with Gabrielle Benabdallah and Nadya Peek.
- *Tandem: Reproducible Digital Fabrication Workflows as Multimodal Programs* [TORZP24], published at the ACM Conference on Human Factors in Computing (CHI) in 2024 and coauthored with Thrisha Ramesh, Octi Zhang, and Nadya Peek.

In each of these prior publications, I served as the first author and contributed the foundational idea for the given paper, developed the majority of each system's implementation, and wrote the majority of each manuscript. As such, I use the contributions in each paper to argue my own thesis statement, which I defined in Section 1.3. Nonetheless, the work described in each paper is joint work between me and all coauthors; credit for all work detailed in the above papers is shared with all coauthors.

As a general note on pronoun usage, I use the pronoun "I" to discuss my own arguments and views in this chapter (Chapter 1), the discussion chapter (Chapter 7), and the concluding chapter

(Chapter 8). In other chapters, I use the pronoun "we" to represent a statement from all coauthors of the corresponding paper.

In terms of the textual content of this dissertation, Chapter 3 through Chapter 6 contains the text of each paper as previously published modulo the following changes.

First, Chapter 4 has been heavily edited to account for post-publication improvements to visualization techniques, namely, using data visualization techniques at all as opposed to only compiler-based approaches. At the time of publication, the paper presented the experimental programming interface along with an initial version of Toolpath Stylesheets that proposed a one-to-one mapping between a machine instruction and a visual element. In this dissertation, I instead present the initial compiler-based concept for visualization, an updated grammar of a graphics-based approach using modern data visualization libraries, and an abridged description of the experimental programming interface that leads into the description of Imprimer's programmer interface in Chapter 5. Thrisha Ramesh collaborated with me in implementing the grammar of graphics-based approach for TSS, which is still under development at the time of writing.

Second, the introductions of all papers have been lightly edited to create both a cohesive flow between chapters and a more direct mapping of contributions back to the thesis statement.

Finally, the majority of the Related Work sections of all papers have been consolidated into Chapter 2.

Chapter 2

Background and Related Work

2.1 Applicability to Digital Fabrication Practices

This dissertation focuses on a type of digital fabrication practitioners that I have termed *experimental* digital fabricators. However, many people use digital fabrication tools across many sites of practice. In designing physical-digital programming, I mostly pay attention to experimental fabricators, even though they currently constitute a minority of digital fabrication users.

To get a sense of the landscape of digital fabrication usage, I identify four clusters of fabrication practice that I have formed by grouping fabricators' sites of production, their priorities, and the tools they use. I arrange these clusters of fabrication practice on a spectrum described in Table 2.1, beginning with the cluster for which physical-digital programming is least applicable and ending with the cluster for which it is most applicable.

Of course, not every fabricator falls neatly into one of these clusters, which are coarse-grain rather than highly descriptive. Still, it is useful to at least conceptualize these clusters to get a sense of different themes of practice. Moreover, it is useful to have some concept of "conventional versus experimental" when trying to conceptualize various forms of digital fabrication practice.

	Conventional Fabrication	*Experimental Fabrication*
Applicability of ΦDP	Low	High
Workflows	Learning and practicing established workflows well	Developing new workflows
Machines	Factory-scale and desktop-class machines	Desktop-class machines, experimental and/or open-source hardware
Design Spaces	Known set of parameters, practices, and instrumentation	Less constrained possibilities of machine usage, unforeseen design spaces
Communities of Practice	Established, often represented by labor unions	Dispersed across heterogeneous applications, not tightly united between domains
Manufacturing Constraints	Well-defined and standardized	Requirements only become apparent with experimentation and iteration
Communication	Shared, often unspoken understanding of specifications across fabrication processes	No unifying language, piecemeal use of GUIs and programming languages
Scale	Larger-scale manufacturing	One or few copies of an artifact
Product	An artifact manufactured to a tolerance of a specification	A new process, alongside the artifact it produces

Table 2.1: Coarse-Grain Spectrum of Digital Fabrication Practice. "Conventional fabrication" refers to practices closer to professional CNC machining. "Experimental fabrication" refers to the development of novel fabrication workflows for niche applications. Physical-digital programming (ΦDP) is targeted towards practices that fall on the experimental end of the spectrum.

2.1.1 Least Applicable: Occupational Machinists

CNC machinists are professionals who are typically trained in working with digital fabrication tools as a trade. Their work involves fabricating machine parts or other artifacts, typically out of metal, using CNC mills, lathes, and other machines. They follow established practices for controlling levels of precision as required by clients, with the accompanying visual language of engineering drawings. In contrast to experimental fabricators, occupational machinists follow a narrow and set of fabrication practices that are well-documented and bounded.

Many of the challenges addressed by physical-digital computing do not apply to machine shop practice. For example, an inevitable challenge of CNC milling is making sure a workpiece—the stock material from which the mill will subtract excess—is localized to the physical machine. For unconventional tasks, such as two-sided milling with a 3-axis mill, Tandem (Chapter 6) helps fabricators match the location of the workpiece with uncertain machine dimensions using constraints expressed in code. However, in machining practice, there is a more established way of localizing a workpiece on the machine using datums measured with specified instruments.

Knowing how to localize is a skill passed down, is typically not explicit, and is only written in handbooks or reference materials—if at all. In contrast, the goal of physical-digital programming is to make such steps explicit in code for unconventional fabrication processes that do not have an established way of doing things.

2.1.2 Maybe Applicable: Computer Science Researchers

Research on CNC machines has been a topic of research in mechanical engineering since the widespread adoption of CNC tools. More recently, increased affordability of desktop-class machines has brought access to digital fabrication tools outside of engineering shops and into labs and educational environments. In particular, publications in computer science have increasingly been studying digital fabrication from a computational perspective.

Subdisciplines of computer science that have taken an interest in fabrication include: human-computer interaction, computer graphics, programming languages, and robotics. The applications of digital fabrication in these works are respectively myriad, focusing on topics such as: the adoption of DIY 3D printed prosthetics [HHHM16], decomposition of complex 3D models into millable components [MAYZ+20], simplification of CAD models that were inferred from mesh representations of 3D models [NWA+20], and synthesis of robotic arms for specific tasks [HCA+18]. Later sections of this chapter review relevant research from each of these areas with respect to the design of physical-digital computing.

Despite the diversity of perspectives, the vast majority of computational perspectives share one commonality: they presume a fixed fabrication step that is separate from the computational or algorithmic contribution. Put another way, with this assumption, the physical fabrication process is not "in the loop." In many cases, the authors may never actually fabricate any artifacts, opting instead for simulations that more readily confirm the goals of their algorithmic contribution, rather than dealing with the messiness of physical hardware. I would argue that the vast majority of publications in computer science on the topic of digital fabrication implicitly adopt this digital-first, physical-second viewpoint.

While there is nothing inherently wrong with keeping focus on algorithmic and computational techniques, especially in computer science research venues, this approach can cause researchers to focus too much on computational approaches that are purportedly related to fabrication, yet never connect back to the tangible manufacturing of objects. In contrast, physical-digital computing attempts to bridge this divide by representing physical concerns—that constrain and guide the design of computational techniques—in code itself. I believe that physical-digital programming, if adopted more widely by computer science researchers, could make it easier for research to consider the fabrication constraints, material sensitivity, rather than adopting an idealized view of fabrication. This is one of my goals in proposing physical-digital computing: to convince more computer scientists that physical contingencies are important and not to be brushed aside. Being able to access such contingencies through code is a key part of getting this message across.

Finally, in contrast to the dominant viewpoint, a smaller share of research in computer science actually does focus on the physical behavior of a machine first and foremost, working backwards from empirical behavior to computational abstractions. Some examples are: developing grammars of material deformations [TPBM21], using augmented reality and human input for material-aware robotic plastering [MEJV⁺22] fabrication-in-the-loop reinforcement learning for viscous material deposition [PFX⁺22], and composing unconventional 3D printing workflows with input from interactive data sources [KZT⁺18]. The projects presented in this dissertation draw from these existing computational techniques and present the ideas in a programming medium that can be more easily reused by others.

2.1.3 More Applicable: Students in Academic Makerspaces

One of the fastest-growing sites of digital fabrication tool use are makerspaces. Makerspaces are community-oriented workspaces with digital fabrication machines, manual tools, electronics equipment and more. Makerspaces can be described with several classifiers, for example: hackerspaces, hobbyist makerspaces, bio-makerspaces, and academic makerspaces. For this cluster, I focus specifically on those who are learning to use digital fabrication tools under the auspices of an academic program.

A common theme that unites makerspaces is sharing of knowledge, either through peer education, teaching workshops, formal classes, or other techniques. Academic research has studied the transfer of knowledge in makerspaces, either through individual publications or through entire academic conferences such as the International Symposium on Academic Makerspaces

(ISAM). Programs such as Fab Academy, a 5-month course on digital fabrication skills taught in makerspaces worldwide, bolster the notion of a single curriculum that teaches students a broad range of hardware and software techniques. This contrasts with, for example, a CNC machinist's curriculum which is more specialized.

While physical-digital computing is not primarily designed for use in makerspaces, its adoption could be impactful. Makerspaces prioritize knowledge-sharing alongside empowering students to build projects that apply to their own tastes and use-cases. As a result, makerspace users could benefit from software tools that make more of content they need to learn explicit in code, rather than existing only "off the page." With stronger programming foundations, students could potentially have an easier time building fabrication workflows that deviate from what off-the-shelf software tools do.

One critique of this argument is that not all students want to program, for example, makerspace users who are primarily fine artists. My rebuttal to this is that students in makerspaces will already need to program if they use digital fabrication machines at all. However, they accomplish the act of "telling the machine what to do" by stringing together multiple GUI tools. This method of programming often suffices to complete the assignment, but does not give artists and other makerspace users *vertical movement* [LRR+23]: the ability to work at different levels of abstraction while making art. In fact, I argue that the current ecosystem of software tools used in makerspaces and taught in digital fabrication curricula enforce a *normative ground* [LRR+23] of what fabricators can and cannot do. In contrast, physical-digital programming could broaden the space of what is even considered a feasible project in a makerspace.

2.1.4 Most Applicable: Experimental Fabricators

Revisiting experimental fabricators in science, art, and engineering whom I discussed in Chapter 1, we can now get a better sense of why they differ from other clusters of practice. First, as opposed to other types of practice, experimental fabricators work on nice, one-off problems that are highly specific given application. This differs from professional machinists who benefit from shared language and understanding around a conventional set of fabrication workflows. Even academic researchers of fabrication and academic makerspace users typically approach problems preconceived notion of a "right way" to do digital fabrication. In contrast, with experimental fabrication, the demands of a particular application eclipse established practices.

Second, as mentioned earlier, an experimental fabricator's product, more than any other cluster of practice, as the novel fabrication process itself, rather than just the artifact. As the name suggests, experimental fabrications rely on repeated experimentation to understand their problem space and design requirements. They stand to benefit the most from physical-digital programming languages that let them borrow from previous experimental workflows and share their own work.

Overall, understanding the conventional-experimental spectrum of fabrication practices helps us bound the scope of physical-digital computing. While many fabricators would not benefit much from the tools proposed in this dissertation, other, unlikely makers could.

Finally, I note that this spectrum is not necessarily correlated with expertise. Many academic works on fabrication posit that computational tools are valuable because they lower the barrier for those who are unfamiliar with digital fabrication, at least for predetermined tasks. However, I am not designing for "just novices"—physical-digital programming might be useful for both experts and notices. Further, the notion of experimental workflow development complicates the dichotomy of novice and expert in the first place: somebody who is an expert at once process will not necessarily be an expert at a completely novel workflow.

2.2 Experimental Digital Fabrication

2.2.1 Applications in Art, Science, and Engineering

The ability to program machine movement to create forms that are too difficult to craft by hand has become more accessible in recent years. As a longstanding patent on FDM 3D printing technology expired in 2009 [Cru92], non-industrial-scale fabrication machines have become affordable and feasible to use in a variety of settings. Maker communities, researchers, and companies have developed machines [VTSTOP20, IPPA15], as well as an ecosystem of software tools.

Artists have used code to explore computationally driven forms in clay [MJS+20, Tih18, DT19, Hus06], wood [Csu68, CPG+15, WVMR18], jewelry [TFRSB17, TLP16, JZ15], and bio-materials [SBS+20]. In each of these cases, artists leverage unique properties of the material that are not supported "out of the box" with conventional software. Instead, artists often navigated the tension between physical and digital by "hacking" their process. For example, in the case of Coral Cup [RLR18], the artists programmed a reaction-diffusion system to derive a coral pattern [KM10] as well designed a mold-making pipeline to slip cast [MTP08] the final forms. The trend of building one's own tools follows with weaving and knitting machines, where several artists and researchers have explored unconventional ways of "programming" textiles, such as through image and crowdsourced data [AHYD20] or techniques derived from mathematics [HMH20, NAH+18]. In all of these cases, as artists generate new forms, they also generate new fabrication workflows ad-hoc.

In materials science, the addition of programmable machine precision for depositing materials has unlocked a vast design space of new material forms. For example, materials scientists have exploited nano-, micro-, and meso-scale structures to generate metamaterials [IWKB17, IFW+16, IKS+18, SBR+15], sensor materials [WGD+19], laser-driven nanoparticle printing [ULSF10], and soft robot actuators with auxetic materials [NN20, FTD+14, LMM+18]. Materials scientists have also used additive manufacturing to work precisely with biological materials, such as spirulina [FIM+21], bio-inks, [GBT19] shell chitin [MSDRO14, LWK+20], and living yeast as a catalyst [JFP+20, SJS+18]. In all of these cases, materials scientists must engineer machines to perform highly specific, high-precision tasks. In fact, most research papers report the scientific principles for their manufacturing process, but the technical implementation and the means of exploration are rarely included.

In contrast to creating a physical product, biologists have sought to automate parts of their experiments through *high-throughput experimentation* techniques [IA08]. While most of this automation technology has historically only been feasible for well-defined, large-scale tasks, increasing accessibility of robotics and fabrication machines has made low-cost, DIY automation possible [DSTM17, CGDE20]. Though several past works have contributed ways of standardizing biology protocols in programming languages [AT10, Car20] as well as assemblages of hardware and software for tasks like imaging [BSH+19, BAH+20, THL+18] and maintaining plant samples [XNB+21], such solutions are typically one-off, and if a biologist needs to adjust functionality, they must “hack” the existing implementations.

Summary. *Practitioners in emerging domains seek fine-grain control and visibility of machines through programming, but lack tools for these programming tools for workflows outside of common-case tasks.*

2.2.2 Programming Interfaces for Digital Fabrication

Widely used existing tools for digital fabrication include CAD software [Rob22], CAM software like slicers for 3D printers [Ult21] or toolpathing software for CNC mills [Vec], and machine-specific control suites [Gin22]. A large body of work in HCI has focused on the notion of *personal* *fabrication* [BM17] drawing from Weiser's notion of *personal computing* [Wei99]. In this vision, digital fabrication machines seamlessly For researchers, this means that building novel workflows

Besides contributing novel workflows themselves, research in HCI and graphics has proposed new paradigms for making with digital fabrication tools [KZT+18, WXW+11, MLB12], while a smaller subset of work has critically investigated challenges that arise during the process. For example, Yildirim et al. interviewed professionals about their needs and challenges using these tools, noting that "negotiation of [fabrication] trade-offs" was a crucial part of a maker's skill set, yet was not facilitated by most tools [YMZ20]. Hudson et al. studied challenges to 3D printing for newcomers, noting that one way to reduce barriers would be "to 'weave in' expert tips, advice, and explanations throughout the printing workflow" [HAC16]. Torres et al. examined strategies or rituals that makers adopted to account for the possibilities of failure with fabrication machines [TSN+18]. Design researchers such as Andersen et al. [AWDM19], Albaugh et al. [AHYD20], and Devendorf et al. [DDKMR16] have foregrounded the experience of blending human craft with machine precision, along with developing mindsets that account for tensions between the two. From a high level, these studies highlight that the capabilities and limitations of a given machine must be accounted for at all parts of the making process. Schoop et al. [SNL+16] and Knibbe et al. [KGF15] built prototypes of makerspace equipment augmented with sensors and projectors to monitor equipment use and provide alerts and recommendations to the maker. Another example is Hofmann et al.'s PARTS plug-in for the Fusion 360 CAD program, which lets designers of 3D models embed additional information on design intent in the 3D model [HHHM18].

Despite these innovative framings and improvements, these prior works still tend to assume a linear, one-directional workflow, rather than exploratory fabrication. Such exploration during digital fabrication can foster important discoveries. By employing a bottom-up approach—where makers first specify a single hair without a CAD model for their final outcome—makers using Cilllia can create hair arrays on flat and curved surfaces [ODC+16]. Defextiles leverage under-extrusion of plastic filament ("defects") on unmodified machines to create textile patterns useful for rapidly prototyping fashion designs, for example [FDFI20]. To develop Cilllia, Defextiles, and similar approaches, makers must explore the trade-offs between print speed and the extrusion multiplier, which is similar to the materials scientist's exploration of gel deposition. Verso aims to help makers, including researchers and hobbyists alike, more easily assess machine capabilities and materials and, ultimately, develop novel fabrication techniques and applications.

Exploration is also common among hobbyists, such as makers on Twitter's #PlotterTwitter community who experiment with plotting techniques and materials [TSTOP21]. Additionally, Li et al. [LBM+20] find that artists using fabrication machines (1) write custom software to circumvent forms of automation that does not let them intervene, (2) seek multiple levels of abstraction in their software tools to have more fine-grained control over aesthetics, and (3) experience workflow breakdowns when moving between multiple tools non-linearly. Based on these insights, Li et al. advocate for software that facilitates "rapid digital-physical transitions." Verso addresses this need. It lets makers preview and manipulate aspects of a fabrication workflow in one unified environment, write code at different levels of abstraction, and make non-linear revisions, and iterate with minimal cost in time and materials.

In contrast to the aforementioned work—which does not support workflow-scale experimentation—some researchers have proposed flow-based programming environments specialized digital fabrication. Flow-based programming environments comprise *nodes* or *modules* which take input data visualized as wires and output data which can be connected to subsequent nodes. For example, Mods provides a web-based environment for generating toolpaths by chaining together routines like thresholding and vectorizing an image [PG18]. Dynamic Toolchains [TSP24] extends this concept with increased extensibility in the browser while Vespidae [FNH+23] provides similar functionality within the Rhinoceros [Rob22] and Grasshopper [McN23] CAD and scripting environment. However, this approach tends to focus on movement from the digital to the physical only; it is conceptually difficult to apply input from the physical world back into a flow-based programming environment beyond the scale of an individual node. The visual-first environment can become messy as workflows grow increasingly complex, and adding new functionality can require writing an entirely new module. Finally, flow-based programs can be difficult for fabricators other than the author to reproduce because the programs do not make assumptions about machine and material setups explicit.

Physical-digital computing attempts to solve some of these shortcomings by leveraging established techniques from programming languages. Namely: movement from physical back to digital is handled through programming language constructs, workflow steps are linearized and organized in text rather than being dispersed in 2D visual space, and machine and material requirements are explicitly represented.

Summary. *Most prior research in software for digital fabrication optimizes a part of a presumed workflow, or assumes a black box workflow. Visual tools for developing workflows suffer from limitations clutter and implicit hardware requirements. Physical-digital computing enables whole-workflow development that is relatively easier for others to reproduce.*

2.2.3 Human-Robot Interaction

A major goal of physical-digital programming is to help fabricators adjust physical circumstances. For example, a physical-digital program should help the fabricator adjust the physical setup on a CNC milling machine or change the position of the machine's tool, reflecting any changes made in the physical world back in code. In the case where the fabricator needs to manually adjust their machine or material, our system, Tandem, provides a grammar of AR-based interactions that guides the fabricator to carry out these physical steps (see Section 6.3). This functionality complements an existing body of work in fabrication and human-robot interaction (HRI) that leverages AR, for example: machine maintenance with AR [TSBS+23, FMS93], CNC milling with a camera feed that supports direct manipulation [MBF+18], AR-supported collaborative making with a robot arm [TP21], and projecting CNC machine parameters onto machine safety glass [OGL08].

Outside of CNC milling, many HRI papers leverage AR to enhance interactions between humans and robots; to organize AR-based HRI, Suzuki et al. created a taxonomy of several design dimensions for this emerging space [SKX+22]. One focus area is helping people work with understanding [KLP+20] and debugging [CPT+22] printed circuit boards by projecting data from schematics onto the physical boards. Other works ([FCLI10, WHAG15]) use AR to prototype interactions for returning physical 3D scan data back to a programming interface. Researchers have further studied pain points that fabricators face when working with data from existing physical constraints ([MJMA19]). AR methods have also been leveraged for mixed-reality 3D modeling [LZB+17], 3D printing [PBW+18, YK16, EPF+17], carving [HT19a, HT19b, ZP13], laser cutting [MLB12, RM18], building-scale construction [YIO+15], and for providing tutorials for manual tools [SNL+16] fabrication machines [HQW+21, YCC+20, Rya22].

Summary. *Prior work on human-machine AR prototypes fixed interactions between a machine and an AR interface. In contrast, Tandem uses programmable AR interfaces within a coding environment that fabricators use to specify what other fabricators must do to accurately replicate a workflow.*

2.3 Programming Languages and Environments

2.3.1 Exploratory Programming for Digital Data

To motivate the need for programming language innovations for digital fabrication, I draw inspiration from a burgeoning body of work in data science and visualization which leverages novel programming environments and language features. The very notion of using code to process, explore, and act on data stems largely from Tukey's notion of *exploratory data analysis* [Tuk77]. Tukey held that too much emphasis was placed on the dominant approach of hypothesis testing, which he could called *confirmatory data analysis*. In contrast, the aim of *exploratory data analysis* is to explore the data through both numerical and graphical means, which Tukey calls "detective" in character rather than the "judicial or quasi-judicial" aims or confirmatory data analysis. Anscombe's quartet [Ans73] (and later, the *Datasaurus Dozen* [MF17]) showed the importance of visualizing data to develop understanding, rather than relying on numerical descriptions alone; the same intuition motivates Toolpath Stylesheets (Chapter 4) for understanding machine behavior.

Over forty years later, languages [R P93, WW05, Had07, Kib22, PK20, JDR+19], environments [KRKP+16, CH18, CVTH21], and tools [KHM17, SBP21, WFB+21, HHB+19] for exploratory data analysis proliferate. To design Taxon, I adopted a strategy used by the Vega and Vega-Lite grammars, where all information required to generate lower-level code for a visualization (in D3.js) is represented declaratively in JSON format [SWH14, SMWH17, BOH11]. Formalizing visualization properties in this way enables automated recommendation of visualization chart types [WMA+16] and formalizing design best practices as constraints [MWN+19].

In addition, physical-digital programming tools like Tandem (Chapter 6) apply programming language techniques common in data science tools to digital fabrication. Such techniques for digital-only applications include: live programming, multimodal input, and frequent views of data.

In particular, live programming envisions a programming environment that provides coders with continuous feedback as they author a program [Tan90]. Programming environments that support liveness (e.g. [SRHH16]) can tighten the feedback loop between code and its effects, enabling faster iteration and debugging. Omar et al. introduced liveness in programs through user-defined GUIs called livelits [OMB+21]. End-users invoke a livelit they previously defined by calling the function in a program, which introduces a GUI in-line.

Moreover, program visualizations that describe a program's state increase code understanding and debugging [HSH18]. Prior work has investigated application debugging through targeted inspection [BBKE13] and always-on visualizations [LBM14, KG17]; many of these contributions influenced the design of state-of-the-art web developer tools. Physical-digital computing extends such visualization and interactive debugging techniques to digital-physical workflows.

Summary. *Innovations in programming languages and environments have facilitated exploration and understanding of digital data, but do not yet extend to digital-physical workflows.*

2.3.2 Computational Notebooks

I argue that one exploratory programming environment in particular provides a fruitful foundation for physical-digital programming is *computational notebooks.* Compared to the myriad of existing programming techniques for digital fabrication that we have reviewed above, why might computational notebooks be uniquely suited for fabrication workflow development? My key insight is that makers can best understand machine behavior by writing code, prose, and visualizations in tandem.

Computational notebooks find their origins in literate programming gas proposed by Donald Knuth [Knu92], which envisions source code that can be flexibly rearranged to complement documentation in natural language, and, more recently, with graphics, visualizations, and other multimedia. We argue that computational notebooks offer a medium to quickly explore not only datasets, but also physical machine behaviors and interaction techniques.

Currently, computational notebooks are popular tools for data science and machine learning; they have not yet been widely explored for machine control tasks. Yet, even within these digital-only contexts, computational notebooks are still evolving programming environments. Researchers have discussed the limitations of computational notebooks by identifying nine pain points data scientists face [CPH+20] when using popular computational notebooks like Jupyter Notebook [KRKP+16]. Among these pain points is "Reproduce and Reuse," where the authors discuss the challenges data scientists run into when trying to reproduce or adapt existing notebooks. An analysis of 1 million notebooks on Github and found that a quarter contained no explanation, revealing a tension between exploration and explanation [RTH18]. In response to this Head and colleagues proposed systems to organize existing notebooks into "cleaned up" slices of analysis [HHB+19] or flexibly organized tutorials [HJS+20]. The Observable notebook in particular addresses some of these problems by providing a live, topological runtime environment that permits notebook cells to be arranged in any order [Obs23b]. We argue that these techniques for organizing notebook code could also help makers experiment and iterate quickly on novel milling workflows.

Other research on computational notebooks in HCI has examined more powerful interaction techniques. For example, direct manipulation techniques [KRH+20, WHS20] generate code by manipulating visualizations within the notebook, forking and backtracking let notebook users explore alternative approaches [WDBD21], and programming-by-example techniques allow data scientists to generate general code from example data wrangling tasks [DBG+20]. One domain-specific language for data science workflow supports live programming with interactive results [DeL21], and another featured within-code graphical components that can be transformed across exploratory coding and presentation-focused dashboard interfaces [BCH+22].

Summary. *Prior research has explored and addressed issues with notebooks for data science and machine learning. In contrast, physical-digital computing extends notebook programming to the development of workflows for digital fabrication.*

2.3.3 Recognizing and Navigating the Physical-Digital Divide

While computer systems that interface with the physical world date back to the development of computers themselves, experimental programming of physical effects became more popular with *physical computing*: the practice of using microcontrollers, sensors, and actuators to prototype interactive systems that respond to physical input [OI04]. Facilitated by the development of the microcontroller boards and their associated languages, physical computing retains a DIY ethos also enjoyed by researchers [KP10]. As in other areas of embedded computing, these languages are typically low-level assembly or C-like languages [Her03, ard05] which have enabled programmers to handle sensor and actuator data across applications, but also introduce significant overhead for the programmer [BEGK11, SG08]. However, MicroPython [Geo14] and CircuitPython [Cir24] have reduced some of this overhead by bringing physical computing to Python's ecosystem.

While physical computing provides a theoretical foundation for physical-digital computing, the latter's goal is different. Whereas physical computing lets programmers build interactions that incorporate sensors and actuators, physical-digital programming is more specialized to multi-stage workflows involving fabrication machines and materials. Physical computing is broad in scope. For example, Arduino programs could interface with any device and allow the programmer to first build simple logic like blinking an LED when an object is close to a distance sensor/ More complicated interactions require the programmer to write correspondingly more complex logic in code. In contrast, physical-digital programming features programming language constructs that represent physical contingencies when working specifically with fabrication machines. For example, toolpath stylesheets (Chapter 4) quickly visualize changes from editing code which generates toolpaths and physical-digital assertions (Section 6.4) declare prerequisites for manufacturing steps of a workflow.

Within digital fabrication in HCI, researchers have explored techniques for bridging physical concerns with code. Sensicut detects and alerts makers to material properties that impact laser cutting and enables them to apply their knowledge of material cutting via a GUI [DACS+21]. As an example of this seamless integration of software controls and physical feedback, Tian et al. created a GUI to associate maker software interactions with a physical lathe; it provides controls for specifying physical constraints and enables the looping of repetitive actions. In addition, haptic feedback helps makers become aware of the machine's state, also bridging the gulf of evaluation. Gulay and Lucero [GL19] introduce "integrated workflows," which are similar to Tian et al.'s vision of "lucid" fabrication workflows that blend the digital and physical [TSK+19, TP21]. Fossdal et al. extend CAD environments to encourage toolpath and material exploration [FHP21].

Again, the drawback of these systems is that they are one-off. It is difficult to apply these techniques—especially AR systems—to new workflows without rebuilding from the ground up. One solution to making these solutions more reproducible is to incorporate them into programming languages that practitioners use. Willsey et al. contribute an example of this in the domain of digital microfluidics by creating a full-stack domain-specific language (DSL) compled with microfluidics and imaging software [WST+19]. Another exception is p5.fab by Subbaraman and Peek [SP22] which incorporated low level machine instructions as Javascript library alongside a programming environment that provided simple toolpath visualization. In a workshop with artists, this library helped participants create a variety of novel forms that required exploration of parameters such as extrusion rates, pausing, and velocities. This approach embodies the goals of Taxon and Verso, which is to represent expressive control over machine behaviors in a programming language. Verso seeks to build further on the imperative paradigm [Ion20] used by p5.fab by decoupling the flow of data from physical concerns via functional programming, as well as handle issues like machine synchronization and toolpath visualizations concisely, without requiring large amounts of boilerplate code.

Finally, on the more theoretical side, Petricek et al. propose *coeffect systems* which formalize program semantics that depend on factors beyond a variable context; for example, specifying how functions behave when a computing resource like a clock is or is not available [POM14]. Such a framework could be useful for thinking about the design of physical-digital programming.

Summary. *Prior work has sketched ways that programming tools can synchronize with empirical physical data, but have not yet considered how such synchronization can be first-class programming constructs.*

Chapter 3

Formal Reasoning about Digital Fabrication Machines

While most industrial and academic work focuses on new possibilities for making through digital fabrication, we still lack of common infrastructure for formally representing a machine and its capabilities. Knowing which machine is best-suited to a manufacturing task is often learned by trial and error. Many novice and would-be users of digital fabrication tools encounter a considerable learning curve when faced with fundamental questions that many expert users take for granted, namely: How does the machine work? Which machine is right for the job? How do we integrate a machine into a manufacturing process with digital and physical materials? The open challenge is how to empower a diverse set of fabrication machine users to achieve their manufacturing goals amidst a piecemeal ecosystem of fabrication software, hardware, and domain expertise.

In recent years, desktop-class digital fabrication machines are gaining in popularity, with feature-heavy variants and sophisticated manufacturing workflows frequently introduced. Beyond increasing educational resources, we argue that digital fabrication needs a unifying formal representation of what a machine is.

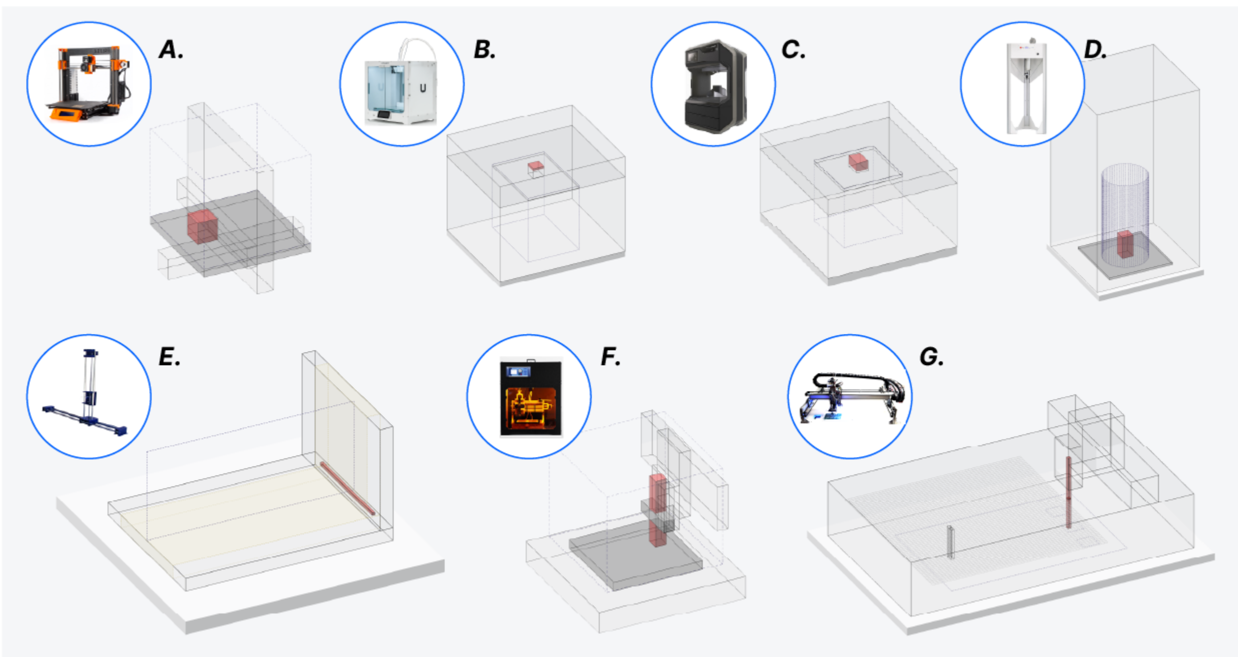


Figure 3.1: Taxon Concept. The Taxon language can represent a breadth of digital fabrication machines as programs that compile to abstracted machine simulations. Shown are Taxon implementations of A. Prusa i3-mk3, B. Ultimaker S5, C. Makerbot Method, D. Delta WASP 2040 Clay printer, E. hot wire cutter, F. xPrint modular liquid printer, and G. LitePlacer pick and place machine.

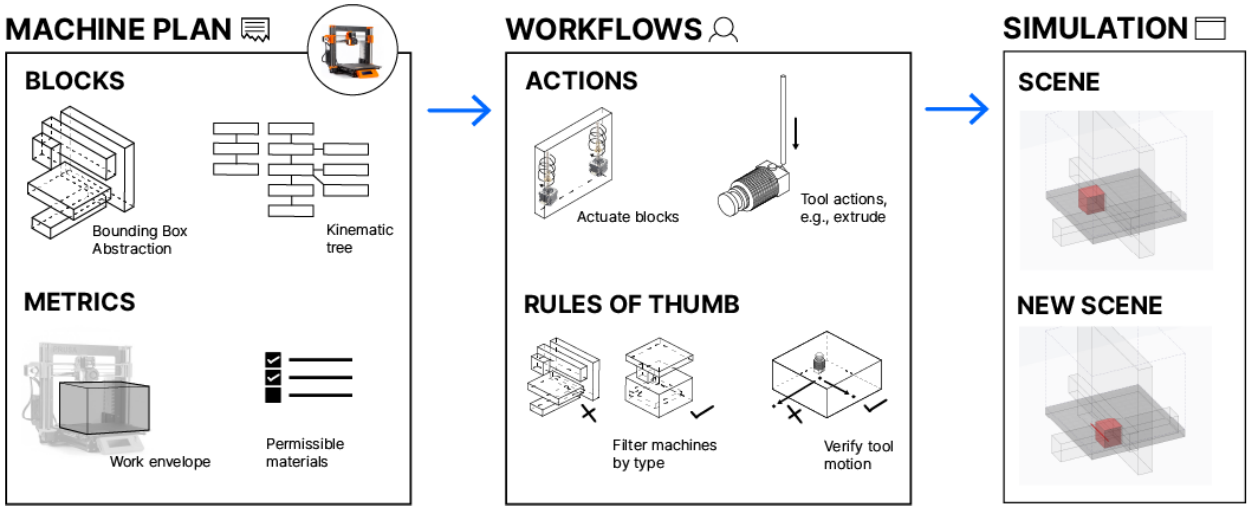


Figure 3.2: System Architecture. Makers represent a physical machine as a Taxon Machine Plan program. The program contains: *blocks*, which provide an abstracted composition of the machine; and *metrics*, which describe high-level machine characteristics, such as the volume that its tool can access (work envelope) and material compatibility. Taxon Machine Plans can be used in workflows containing *actions*, which simulate motion and fabrication. *Rules of thumb* are a database of community-contributed rules that can be used when analyzing machine plans and actions for appropriate machine choice for a manufacturing task or safety of actions. Executing an action updates the machine in the simulation, here, actuating the blocks such that the print head (in red) moves to the directed position.

We argue that a programming language for machines would most effectively help makers adapt to an increasingly complex space of manufacturing workflows. Compared to one-off software tools, programming languages lend themselves to being extended as programmers develop libraries and share repositories of code. A common syntax and semantics let makers more readily understand features and trade-offs between different machines, as well as use program analysis techniques to enforce best practices. While it might seem tempting to create a simple how-to guide to selecting and using machines, one can never fully anticipate how a maker might want to chose machines based on their own. Instead, we aim to let makers author their own rules of thumb for selection and usage that can selectively be applied to machines-as-programs.

As a first step towards a common enabling infrastructure, we present Taxon, a language for specifying a digital fabrication machine's composition, characteristics, and simulated use cases as programs. Taxon helps users to gather a large repository of machines in a common format, query and compare different machines, and script simulated interactions with machines, digital models, and physical materials. Taxon programs contain three parts: *blocks* and *metrics*, which together form the machine plan, and the *workflow* composed of a sequence of *actions*. Blocks are abstracted black boxes of machine parts that provide enough information to reason about the machine's kinematic and mechanical properties without being prohibitively low-level or verbose. Metrics describe innate characteristics of a machine that determine when it should and should not be used. Workflows comprise actions that simulate a machine's movements and interactions with material. Makers can also author and enforce *rules of thumb*, which are user-defined checks that Taxon enforces about machine selection and use.

We integrate the Taxon language into a web-based user interface that lets users search for machines based on their needs, learn the machine's composition and kinematics, and experiment with using the machine in simulation before moving on to using a physical machine. Overall, Taxon is *descriptive*, i.e., characterizing existing machines and workflows, rather than prescriptive of a single fabrication pipeline. Through the language and web interface, we allow development of an interactive and extensible taxonomy of digital fabrication machines along with a foundation upon which future applications can reason precisely about a machine.

This paper's contributions include:

- **A domain-specific language** for formally representing digital fabrication machine plans and workflows
- **A user interface** that lets users contribute to and browse a machine database and experiment with workflows
- **Several examples of novel fabrication workflows** from research and practice expressed

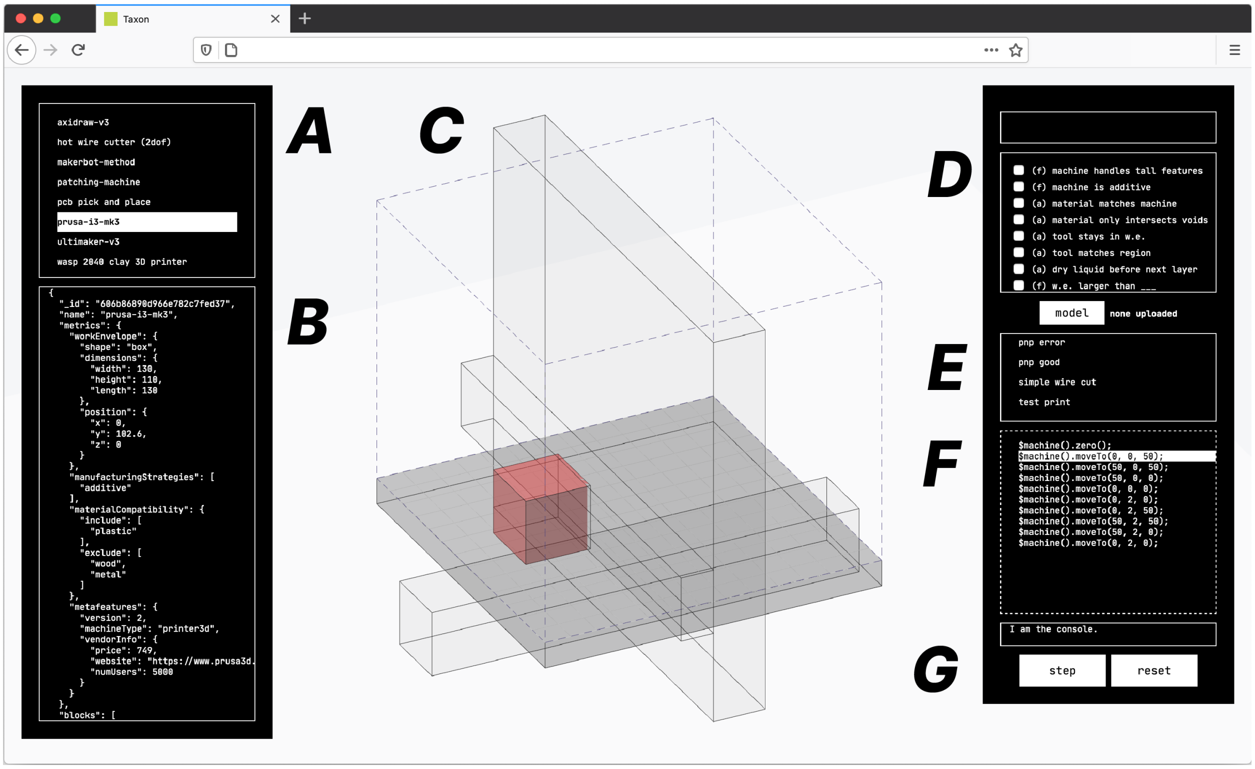


Figure 3.3: Taxon User Interface. The interface runs in a browser-based environment and contains the following parts: A) a database of machine plans, B) the selected machine plan containing metrics and blocks, C) the scene rendered from the current plan, D) rules of thumb—each can be double-clicked to show its implementation, E) a database of workflows, F) the current workflow containing actions that operate on the scene, model and material, executed one line at a time via G) the execution controls.

in Taxon

While material choices and digital models are important parts of workflows, in this paper we focus primarily on representing machines and rules for their use, which in turn provides a foundation for future extensions for materials and models.

3.1 System Architecture

The system we propose has the following design goals:

- To represent a breadth of digital fabrication machines in one standardized format
- To formalize a machine's high-level characteristics and trade-offs and when/when not to use it
- To formalize how a machine moves, how it works with models and materials, and how to incorporate this information into a desired workflow

To address these goals, we implement[1] the Taxon language to (1) represent high-level features, constraints, and composition of machines, and (2) help users compare and simulate basic machine tasks. Figure 3.6 shows the core components of Taxon's grammar in Backus-Naur form — a Taxon program consists of a machine-plan and a workflow:

- **Machine Plan:** describes the composition of the machine from a functional level and contains two parts:
  - **Metrics:** a collection of various high-level properties (*metrics*) about a machine, such as which materials the machine can use and structural considerations of a machine for fabricating certain types of models
  - **Blocks:** a collection of abstracting bounding boxes for functional parts of a machine—each known as a *block*—where each block represents volumetric and kinematic properties the given region of the machine
- **Machine Workflow:** composed of *actions*—statements that are executed by the Taxon interpreter which is specialized [Fut83] for the current machine plan. Actions are valid Javascript statements and are a strict subset of Javascript. Actions simulate various tasks with part or all of the machine, with materials, and with uploaded 2D and 3D models to

[1]Our source code is available online at `https://github.com/machineagency/taxon`.

both visualize the machine and to check for potential errors using codified *rules of thumb*. Rules of thumb are user-selected static and run-time checks that filter machines (*filtering* rules of thumb) and raise warnings for actions that are risky or incompatible with a given machine (*action* rules of thumb).

Taxon also consists of a browser-based user interface that supports browsing and filtering *machine plans* (Figure 3.3, left) and simulating and checking *machine actions* (Figure 3.2 and Figure 3.3, right). To frame the design of Taxon, we envision and refer to two users throughout the paper:

- **The Programmer/Contributor:** a machine manufacturer or community expert who writes machine plan programs to describe real-world machines. We envision machine enthusiasts, e.g., from online hobbyist communities, contributing (1) machine plans for machines on which they have expertise, (2) Rules of Thumb that govern proper machine use as checked by the Taxon interpreter, and (3) extensions to the Taxon language.

- **The Maker:** a fabricator with a rough idea of what they would like to manufacture. These users have some background in digital fabrication but would like to explore a breadth of machines and fabrication tasks beyond their current familiarity. They are not necessarily a programmer and are not interested in modifying machine plans. They want to learn how possible machines and workflows can fabricate their ideas, compare machines based on metrics, and visualize machine performance and limitations.

These user types are not mutually exclusive; people who contribute to Taxon's infrastructure may also use it for their own manufacturing processes, and vice versa. Real-life makers exist on a spectrum between these archetypes each with varying levels of expertise and domain knowledge. We use rules of thumb avoid prescribing universal ways of using machines and also to give more flexibility to expert users. For now, we adopt the makers' perspective as we step through the user interface (shown in Figure 3.3) to explain how the each of the languages work to support this user.

To begin, makers browses a database containing a list of machine plans (Figure 3.3a). They can filter the list of machines shown by querying the database using filtering rules of thumb (Figure 3.3d). For example, they can check the "machine is an FDM 3d printer" rule of thumb to view all FDM 3D printers (printers using plastic filament) or "machine is rigid enough for milling" to show machines whose movements on the workplane are driven by mechanisms rigid enough to withstand high-force applications (such as milling).

Once makers select a machine, the machine plan is loaded into the plan viewer Figure 3.3b. The scene compiler then compiles and renders the plan into an interactive *scene* in 3D simulation Figure 3.3c. The scene is implemented in Javascript so the plan compiler's output is a Javascript object containing a complete THREE.js scene and animation system [Cab14]. Makers can then double-click machine blocks in text in the plan viewer to highlight the corresponding block in the simulation, or double-click blocks in the simulation to jump to the definition in the plan viewer.

Given a selected machine plan and corresponding simulated scene, makers can now program the machine using actions Figure 3.3f. Pre-made action workflows from a community database are also provided Figure 3.3e. Actions animate individual blocks from the selected machine as well as the entire selected machine. They then select action rules of thumb from the database Figure 3.3d, where each rule enforces a constraint on actions in the workflow. For example, the rule *"material must match machine"* requires that any material used in the program be contained in the machine's `acceptableMaterials` metric. Once the machine, model, material, and rules of thumb are selected, the maker writes or modifies actions in the editor and steps through the program using execution controls Figure 3.3g. If any actions violate rules of thumb, the action pane explains the error and proposes alternative machine, model, or material choices.

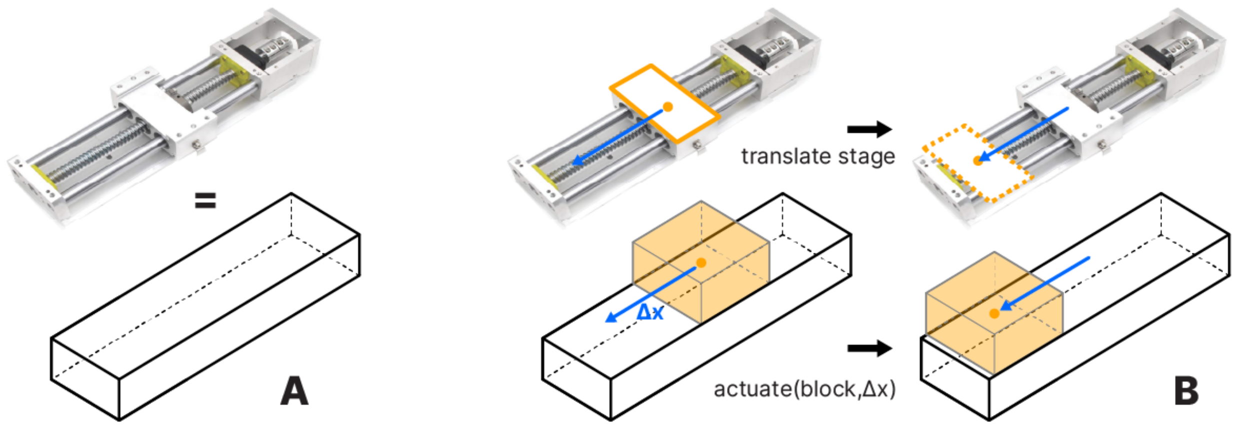


Figure 3.4: Principle of Actuation. In this example, a block (A, bottom) represents a linear actuator (A, top) made up of a driving motor, a lead screw, and a translating stage. When the driving motor steps, the lead screw turns, moving the stage along its actuation axis (B, top). When our block abstraction is actuated, it does not itself move, rather it moves all of its descendants (B, bottom, shown in orange) in the kinematic tree. Each block connection makes up an edge in the kinematic tree. Details such as the step-displacement ratio of a block are listed in the `attributes` property.

3.2 Machine Plans: Blocks

The core of Taxon is its representation of machines as a group of bounding boxes called *blocks*. (syntax shown in Figure 3.6). A block is an abstraction of subassemblies of physical machines. Each block encodes information about the machine's physical size, kinematics, and tool functionality. The goal of the blocks part of a machine plan is to provide one standard representation for many different types of digital fabrication machines; this allows us to access a diverse range of machines in one programmatic representation.

Blocks are JSON objects that are rendered as block-shaped bounding boxes in the scene. Conceptually, a block represents one functional moving unit of a machine. For example, the Prusa i3 3D printer example shown in Figure 3.3 is divided into three blocks each representing one axis of movement, and additional blocks for its build plate and extruder. The organization of a machine is hierarchical in that blocks have *connections* to their children, and connections are one-directional

```
{
  "name": "prusa-i3-mk3",
  "metrics": { ... },
  "blocks:[ ... ]
}
```

```
{
  "machineClass": "printer3d",
  "workEnvelope": {
    "shape": "box",
    "dimensions" : {
      "width": 130,
      "height": 110,
      "length": 130
    },
    "position": {
      "x": 0,
      "y": 102.6,
      "z": 0
    }
  },
  "manufacturingStrategies": [
    "additive"
  ],
  "materialCompatibility": {
    "include": [
      "plastic"
    ],
    "exclude": [
      "wood", "metal"
    ]
  },
  "resolution": 0.001,
  "maxTravelSpeed": 200,
  "metafeatures": {
    "version": 2.0,
    "vendorInfo": {
      "priceUSD": 749,
      "website": "www.prusa3d.com",
      "numUsers" : 5000
    }
  }
}
```

```
{
  "name": "verticalLeadScrewFrame",
  "blockType": "redundantLinear",
  "actuationAxes": [ "y" ],
  "dimensions": {
    "width": 20,
    "height": 150,
    "length": 170
  },
  "position": {
    "x": 0,
    "y": 87.5,
    "z": 0
  },
  "attributes": {
    "driveMechanism": "leadScrew"
  },
  "connections": [
    {
      "child": "crossbarAssembly",
      "offset": {
        "x": 16.25,
        "y": 27.5,
        "z": 0
      }
    }
  ]
}
```

```
{
    "name": "extruder",
    "blockType": "nonActuating",
    "isTool": true,
    "dimensions": {
        "width": 25,
        "height": 25,
        "length": 25
    },
    "attributes": {
        "toolType": "print3dFDM",
        "nozzleCount": 1
    }
}
```

Figure 3.5: (Top Left) Top level properties of a machine plan: name, metrics, and blocks. (Top Right) Example of an actuating block. (Bottom Left) Example metric. (Bottom Right) Example of a (non-actuating) tool block. Not all of this machine's blocks are shown.

⟨*name*⟩ ::= string width $\in \mathbb{R}$ height $\in \mathbb{R}$ length $\in \mathbb{R}$

⟨*position*⟩ ::= $(\mathbb{R}, \mathbb{R}, \mathbb{R})$

⟨*dimensions*⟩ ::= width height length

⟨*connection*⟩ ::= block offset offset $\in (\mathbb{R}, \mathbb{R}, \mathbb{R})$

⟨*actuation-axis*⟩ ::= X | Y | Z

⟨*attribute*⟩ ::= is-tool
| is-platform
| drive-type
| step-displacement-count | …

⟨*block-type*⟩ ::= non-actuating
| linear
| redundant-linear
| cross
| delta-bot

⟨*block*⟩ ::= name block-type dimensions [position] actuation-axis* attribute* connection*

⟨*work-envelope*⟩ ::= shape dimensions position

⟨*metric-value*⟩ ::= work-envelope | $\mathbb{R}$ | string | object | …

⟨*metric*⟩ ::= name metric-value

⟨*shape*⟩ ::= box | cylinder | rectangle | …

⟨*machine-plan*⟩ ::= name metric* block*

⟨*method*⟩ ::= MoveTo (machine-plan, position, $\mathbb{R}$)
| Actuate (block, $\mathbb{R}$) | …

⟨*constructor*⟩ ::= SelectBlock
| SelectMachine
| SelectTool
| SelectMaterial | …

⟨*selector*⟩ ::= constructor name

⟨*action*⟩ ::= selector method

⟨*workflow*⟩ ::= action*

⟨*program*⟩ ::= machine-plan workflow

Figure 3.6: Syntax of core components of the Taxon Language.

and parent-child. For example, in many 3D printers, the filament extruder is attached to a timing belt; in Taxon parlance, we say that the belt connects to the extruder, and the extruder is the belt's child.

3.2.1 Block Syntax

Figure 3.5 shows an example of using Taxon's block feature to specify parts of a Prusa-i3 3D printer [Pru12]. Blocks both present information (e.g. the volume of various parts of the machine) and model how the machine moves in space. We define *moving* to mean that the position of a block changes; we define *actuating* to mean that an actuating block does not move, but all of its descendent blocks move on the actuating block's `actuatingAxes` (see Figure 3.4). Note that the y-axis is the "up" direction in our convention. All blocks contain a `dimensions` field denoting the size of its bounding box, while a block's *position* is coordinates of the box's centroid. A block's position is defined *explicitly* through the `position` property or *implicitly*, where the block is a child to another block and its position snaps to its parent.

Each block has a `blockType` depending on the type of actuation it supports and the number of motors it contains. The bounding box of a block is assumed to contain the motors which e.g., *step* (turn a fixed angular amount) to drive its actuation. For example, the most common type of block is a *linear block* which is driven by one underlying stepper motor and has exactly one actuation axis; a linear block provides one degree of freedom of movement. Examples of linear blocks include each of the actuating blocks in the Prusa 3D printer machine plan. In contrast, a *cross block* represents a more complex assembly with parallel kinematics, where multiple underlying motors interdependently control multiple axes. Examples of cross blocks include the block representing the XZ axes in the Jubilee machine plan, which are driven by CoreXY kinematics [VTSTOP20]. *Redundant linear blocks* are driven by more than one motor but move in a single degree of freedom;

such blocks typically represent linear synchronous motion, for example, the two vertical lead screws on the Prusa 3D printer. Not all blocks actuate. An example of such a *non-actuating* block is the build plate of a 3D printer; while the build plate itself moves, it neither actuates nor provides additional movement to any other blocks. Finally, a tool is a special class of block whose movement can change the scene—for example, extruding 3D printer filament, carving material out of stock, or picking and placing an object. A block is designated as a tool by setting `''isTool'': true`.

Blocks may also have `attributes`, which contain block-specific information. Example attributes include `driveType`, which specifies the physical mechanism used to provide motion, e.g. rack and pinion or timing belt, `stepDisplacementRatio`, which states how much linear displacement along the actuation axes results from one step (the minimum amount of rotation) of the driving motor, and `isPlatform`, indicating that the block is a moving platform holds a model or workpiece. This field is useful particularly for tools because they may have different properties which may be difficult to generalize over all types of tools.

Declaring Motors Explicitly

In early iterations of Taxon, programmers also needed to manually declare motors and specify how motors and blocks paired to provide motion. This let us reason about inverse kinematics, that is, how motors would need to turn to implement given machine movements. Each block type had an underlying kinematics equation that mapped a block's actuation to one or more motors. However, in formative studies, programmers reported that reasoning about motors was confusing and distracted them from their overall goal of representing functional units of movement. As a result, we removed the concept of motors as separate abstractions from the blocks part of a machine plan. Thus, a block is understood to actuate on its actuation axes without the programmer needing to programmatically declare any motors to drive the block's actuation. In future work, we anticipate introducing a lower level of abstraction, where programmers can specify in more

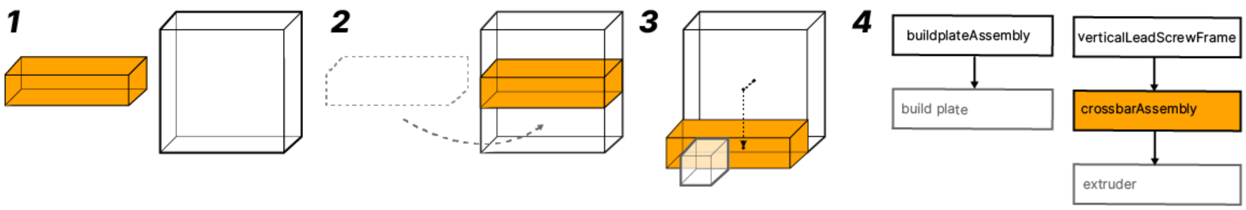


Figure 3.7: Building a Kinematic Tree with Connections. This example uses two blocks, the crossbar block (1, Left) and the lead screw frame block (1, Right) from the Prusa-i3 3D printer example. 1) A connection involves a child (left) and parent block; 2) the child block is translated such that its centroid is at the same point as the parent's centroid; 3) a user-specified offset translates the child relative to the parent's centroid, resulting in the child's final position. Any descendants of the child block (gray) receive the same total translation. 4) shows the kinematic tree for the entire 3D printer, with black arrows denoting connections.

detail how a given block actuates with respect to motors.

3.2.2 Connections

Once programmers have abstracted parts of the machine as blocks, they next define how blocks actuate and move other blocks to move the tool in a controlled manner. They order blocks into a `kinematic tree`, where blocks have parent-child *connections* that make the entire subtree of a parent block move in space whenever the parent block actuates. Intuitively, a connection means that the child block is "placed on" or "attached to" the parent, thus having its position defined in relationship to the parent and moving when the parent actuates.

For example, in Figure 3.7, `verticalLeadScrewFrame` shows a connection with the `crossbarAssembly.` In this case, the crossbar assembly is the child of the vertical lead screw frame, which does two things. First, the position of the child block "snaps" to the position of the parent block such that their centroids occupy the same point, plus the user-defined `offset`. In this example, the crossbar assembly is positioned in the lower front of the vertical leadscrew frame. Second, a connection produces a directed edge in the machine's *kinematic graph*, with the parent and child blocks as nodes. Actuating any node in a machine's kinematic graph actuates its sub-tree along the same axes. For example, if the vertical lead screw frame were actuated, the crossbar assembly and any of its own descendants would move up or down in the parent's sole actuation axis. A block may have more than one child.

3.3 Machine Plans: Metrics

In addition to containing a list of blocks, a machine plan includes *metrics*, i.e. high-level characteristics about the machine such as which materials can a machine use and its resolution. In contrast to blocks, which describe individual parts of the machine, metrics describe properties innate to the machine as a whole. In the Taxon language, metrics are a JSON object where each property is the name of a metric, and the corresponding value is the value of that metric, which can be a string, number, boolean, array, or object. Metrics let users meaningfully compare machines based on their high-level characteristics, for example, checking which machine out of several options has the highest rigidity for effectively milling dense materials. They also provide a way to search for machines based on their high-level characteristics using filtering rules of thumb. For example, if users need to 3D print or mill a model with very fine vertical features, they can search the machine database for all machines with movement resolution below a certain amount. In addition to presenting high-level machine characteristics up front, metrics let programmers specify constraints in action rules of thumb about advisable actions for a specific machine, given its metrics. This lets programmers check whether a given machine can accommodate steps in a manufacturing process as programmed in actions, and if not, find ways to fix them.

It is the programmers' responsibility to list the metrics for a given machine. Generally, machine manufacturers or hobbyist machine builders include information online about a machine's work envelope, resolution, and more specialized details. In this case, programmers need only copy this information into the machine plan. In other cases, such as a list of materials compatible with a machine, programmers might need to gather more information from other users. A list of possible metrics could be maintained and standardized by Taxon contributors, with new additions being vetted and added. Metrics are included on a best-effort basis; not every metric must be defined for every machine. If a rule of thumb needs to check a metric that is not listed, Taxon alerts users that it cannot verify the given rule.

As an example, we describe the metrics for the Prusa 3D printer as shown in the bottom right code listing of Figure 3.5.

- **Machine Class:** a broad designation of this machine's class, e.g., 3D printer, laser cutter, mill, etc.
- **Work Envelope:** the bounding box in which the tool can safely move. The size of the machine's work envelope limits the size of the models it can manufacture. Mobile machines such as the Piccolo plotter [SRPX13] could be represented as having work envelopes with infinite dimensions on certain axes.
- **Manufacturing Strategies:** can include additive, subtractive, drawing, pick and place, etc.
- **Material Compatibility**: a list of materials that the given machine is known to be able to work with (e.g., plastic for the 3D printer) and a list of materials that the machine should explicitly not work with (e.g., metal for any 3D printer). Materials not contained in these lists have unknown compatibility with the current machine, and users must exercise caution if using them.
- **Resolution:** the minimum movement in millimeters that the machine can support. Machines with lower resolutions cannot support very fine features in models.
- **Max Travel Speed:** the maximum speed in millimeters per second that the machine's tool can move while not extruding or cutting material

```
/* Actuate the crossbar block back
   and forth on its axis. */
$b('crossbarAssembly').wiggle();
/* Actuate the lead screw frame 20mm
   on its actuating axis. */
$b('verticalLeadScrewFrame')
  .actuate(20)
/* Extrude 10mm of filament. */
$t('extruder').extrude(10)
/* Establish a coordinate system
   with tool's current location
   as the origin. */
$machine().zero();
/* Move the tool to the given
   coordinates. */
$machine().moveTo({x:50, y:50, z:50});
```

```
$machine().zero();
/* Place the the material that the
   user has chosen from a fixed
   list of options. */
$material().placeAt(10, 0, 30);
/* From the user's uploaded model,
   generate toolpaths for the
   mill bit to follow. */
$model().placeAt(0, 0, 0);
$model().generateToolpath(options);
/* Activate the spindle to spin the
   mill bit at 3000 RPM. */
$t('spindle').setSpeed(3000);
/* Move the spindle along the
   toolpath to cut the material. */
$machine().runToolpath();
```

Figure 3.8: (Left) Workflow testing various movements on the Prusa 3D printer. (Right) Workflow loading sheet material and a 3D model, creating a toolpath from the model, positioning the material and model, and running the mill over the toolpath.

- **Metafeatures:** a collection of various features that do not affect the machine's operation, for example, its cost, website, and estimated size of its active user base.

3.4 Workflow and Actions

We already noted that a machine plan includes both functional blocks of a machine and the metrics that describe the machine's high-level characteristics. In addition to representing the machines, Taxon helps users make the simulated machine *do* things by composing *workflows*. In digital fabrication, a workflow consists of a series of steps that progress from models, materials, and machine to finished product. In Taxon, these workflows are implemented programatically as a sequence of statements called *actions*, which let users simulate motion in the scene and load materials and digital models into the scene. Intuitively, an action is one "step" of the workflow, which could include anything from testing small perturbations to identify how the machine moves to programming a broad-strokes plan of the entire physical workflow from raw material to finished product. By expressing machine movements and model and material concerns in code, we expose opportunities for users to learn and error-check in simulation before they work with dangerous or expensive physical machines and materials.

Actions are a strict subset of Javascript, where programs are composed of multiple statements (also called *actions*) that are separated by semicolons. The action interpreter executes one statement at a time as programmers step through the workflow, and each action modifies the scene's state, which consists of the machine's blocks, their current positions, models and toolpaths uploaded by users, and any material extruded, placed, or cut. Each statement contains exactly one selector that is followed by any number of methods. The selector *evaluates* to an object that then supports a set of *methods* depending on the object's type. Executing a method modifies the scene's state, for example, by turning motors and moving blocks around, by transforming the model's geometry, or by cutting or extruding material. For example, `$machine()` evaluates to a `Machine` object that supports methods like `.moveTo(coordinates, extrusion)`, which actuates all machines blocks so the tool arrives at the desired (x, y, z) coordinate, possibly extruding material while moving; `$t('penAssembly')` evaluates to a `Tool` object that supports methods like `.raisePen()` and `.lowerPen()`. Before and during runtime, the machine action interpreter checks statements against the rules of thumb and highlights any rule violations, for example, moving a block past an acceptable value.

In addition to controlling the machine, actions let makers load materials and digital models into the scene. Like blocks, materials are represented by a bounding box that is placed in the scene, shown in Figure 3.8 (right). Each material must have a *material class*, which can be *additive*, i.e., it is deposited in the scene by actions with additive machines, or *subtractive*, i.e., makers must place the material first and then process it using tool actions. Materials also have names, e.g., "wood" or "plastic," which are typically listed in a machine's `materialCompatibility` metric. This high-level description of materials suffices for high-level simulations. Actions also support uploading digital files (e.g., STL files) and placing them in the scene. Taxon's modular design affords easy extensions to the language for slicing STL files into tool paths and processing geometry; we leave these for future work.

3.4.1 Machine Action Language Example

To illustrate Taxon, we present the examples in Figure 3.8 where a novice user tests various actions using the Prusa 3D printer featured in previous examples. The selector evaluates to an object of type `Block`, `Tool`, `Machine`, `Model` or `Material`, where each type supports its own set of methods. For example, the first two lines of Figure 3.8 select the blocks from the running 3D printer example (type `Block`); then call `.wiggle()` on the crossbar assembly, causing it to actuate back and forth, and `actuate()` on the vertical lead screw frame, causing it to actuate upwards. The next statement selects the printer's extruder (type `Tool`) and extrudes 10 millimeters of filament. The following statement has `$machine()`, which selects the entire `Machine` object and calls `.zero()`, which establishes a Cartesian coordinate system with the machine's tool's current position as the origin. For any actions executing coordinate movement, a coordinate system must be instantiated first by calling `.zero()`. Finally, given the coordinate system, `.moveTo(coords)` moves the machine's tool to the (x, y, z) coordinates specified in `coords`. If a bounds-checking rule of thumb is enabled, the interpreter throws an error if the coordinates are outside the machine's work envelope.

3.4.2 Checking Machine Actions with Rules of Thumb

The motivation behind simulating machine actions is twofold: first, we want users to be able to visualize how a given machine works and to experiment with it, and second, we want users to be able to formally state what they want to do with a given machine in order to enforce best practices.

```
(action, store) => {
  try {
    if (action.constructor === '$model') {
      let we = $metrics.workEnvelope;
      let modelName = action.query;
      let modelDims = modelName.dimensions;
      if (we && modelDims.width > we.width
        || modelDims.height > we.height
        || modelDims.length > we.length) {
        console.error('The model is too large for this machine.');
        return false;
    }
    return true;
  }
  catch (e) {
    console.error('Could not verify "model fits in w.e."');
    return true;
  }
});

(machine) => {
  machine.blocks.forEach((block) => {
    if (block.attributes.isPlatform) {
      let pb = block.parentBlock;
      if (pb.actuationAxes.includes('x')
        || pb.actuationAxes.includes('z')) {
        return false;
      }
    }
  });
  return true;
});
```

Figure 3.9: Rules of Thumb (Action and Filtering) implemented in Javascript. The first fires when an action attempts to load a digital model into the scene; if an error is thrown during checking, e.g. if a metric is `undefined`, then the implementation reports that it could not complete its checking, but returns `true` as a default. The second checks that a machine does not use a platform moving in non-vertical directions; machines meeting this property are better suited to print tall features.

In particular, the act of formally describing one's steps and using error checking offers a useful way to introduce machine knowledge to users. Rather than merely presenting the user with a list of best practices, users can program various machine actions and learn about potential issues and errors in the context of what they are trying to do.

Our solution to these goals are *rules of thumb*, which analyze machine plans and actions to offer suggestions and enforce safe machine usage. Rules of thumb are modular and stored in a community-contributed database; users can select which rules they want to enforce for a given task. To enable these rules of thumb, the user selects which rules they wish to enforce in the Action pane (Figure 3.3g). There are two types of rules of thumb:

- ***Filtering* rules of thumb** examine only the machine plan, filtering a list of feasible machines. For example, users planning to use a 3D printer would likely want to require that any 3D model they will print to fit within the printer's work envelope. Or, someone who is 3D printing a model with tall and skinny features would want to select a printer design that is well equipped to print those features without excessive ringing or vibration.

- ***Action* rules of thumb** examine both the machine plan and actions in the workflow that the interpreter executes. For example, milling machines should have their end mills intersect with and carve away stock material only when the spindle is actively spinning the end mill; otherwise, the end mill will break. One rule of thumb might be to check that whenever a move command is executed with a milling machine, its spindle will be turned on if that move would intersect with any stock material.

Figure 3.9 shows the implementations of two rules of thumb in Javascript, where programmers can access the machine plan as Javascript objects with the same selector syntax used in action programs. Because the machine plan is a JSON object, it can be parsed and traversed in Javascript in the rule of thumb's implementation. The first rule, "model must fit in envelope," is an action rule of thumb that fires whenever the method `Model.placeAt()` is called; here, the rule checks that the model placed in a scene can fit in the current machine's work envelope. The second rule of thumb, "machine handles tall features," is a filtering rule which removes from the machine plan list any machines unable to manufacture tall and skinny features due to their composition.

One limitation with rules of thumb is that best practices can generally be codified only partially, and sometimes not at all. For example, users of 3D printer might want to print with ABS filament, which requires the printer's extruder to be set to a higher temperature. If there are no Taxon actions that simulate printer temperatures, a rule author's best bet is to detect the use of ABS filament and issue a warning or a list of written recommendations. Rules of thumb can also examine the bounding box or metadata of a digital model for filtering machines that can fabricate models of the given size, as well as suggest machines based on the model's file type—for example, suggesting a laser cutting approach for thin STL files. Eventually, as programmers extend the Taxon language by adding more metrics, blocks, and actions, increasing numbers of best practices can be codified as rules of thumb.

3.5 Evaluation: Adding to the Language though Demonstration Programs

Programmers can most readily contribute to Taxon's knowledge base by adding machine plans and rules of thumb. However, they can also extend the language itself: defining new metrics, new kinds of blocks, and new actions as new challenges arise.

Taxon Program				
Machine Plan	*Workflow*	*Rules of Thumb*	*Additions To Core Language*	*Lines of Code*
Three Off-the-Shelf 3D Printers	N/A	"work envelope larger than ___"	dependent rules of thumb	(132*, 0, 15)
Hot Wire Cutter	cuting an airfoil from styrofoam	"material intersects voids only"	work envelope orientation, collision voids	(99, 19, 20)
Wasp Clay 3D Printer [Was19]	basic clay print	N/A	delta bot block, cylindrical work envelope	(102, 16, 0)
Wang et al. xPrint [WYW+16]	controlled deposition of natto cell culture	"dry liquid before next layer"	active tools	(209, 13, 21)
Liteplacer Pick and Place Machine [Juh21]	placing SMD components in footprints on a PCB	"tool matches region"	envelope regions, tool changing	(259, 12, 24)

Table 3.1: Demonstration Machines and Workflows. Entries in the rightmost column denote the lines of code in the machine plan, the workflow, and associated rules of thumb, respectively. *Average lines of code per machine plan.

3.5.1 Evaluation Method

The goal of our evaluation is to understand where Taxon can—and in some cases, cannot yet—gracefully extend to different fabrication tasks. To evaluate Taxon's expressivity, we investigate six demonstration[2] machines that showcase ways of formalizing existing machines and workflows. We use Taxon to represent the breadth of machine types and workflows used in both common practice and digital fabrication research. Table 3.1 summarizes the different machines we use for evaluation, which are also shown in Figure 3.1. Each demonstration is more than just a proof of concept — for each machine and new workflow, we highlight challenges and discuss new rules of thumb and new constructs in the action part of the language. We then implement these new features and add them to Taxon's core. For example, in implementing pick and place machines, we designate parts of the work envelope as *regions*, namely, regions to store parts that the machine will later place, and a region dedicated as a parking location for tool changes. Further, we add rules of thumb based on the regions. For example, we add rules of thumb that enforce that the region where parts are stored cannot overlap with the PCB material, and regions that hold very small parts can be entered only when the appropriate tool for handling small SMD parts is currently in use.

Note that this paper does not assess usability: it focuses instead on flexible infrastructure and language expressivity that let users build new interactions on top of a formal foundation. Furthermore, because the concept of establishing formal checks on fabrication tasks is relatively novel, and therefore lacks precedent, it is premature to compare Taxon's usability to a non-existent baseline [GB08, Ols07]. Once the language has developed sufficient new settings, we can evaluate its ease of use. In the meantime, our primary concern is evaluating the expressivity of the language.

[2] We use Ledo et al.'s definition of a demonstration as an evaluation [LHV+18], which Zhang et al. exemplify in evaluating their programming language for online community governance [ZHB20].

3.5.2 Comparing Off-the-Shelf 3D Printers

As an initial task, we implemented the metrics and blocks for 3 different commercial 3D printers: the Prusa i3-mk3—described in the prior sections—the Makerbot Method, and the Ultimaker S5. In addition, we implemented the metrics only for 20 different 3D printers; all of these machine plans can be found online in our repository. These printers vary significantly in price, material compatibility, and construction, and comparing many printers can quickly become tedious. As metrics for each printer, we added the machines' work envelope dimensions, material compatibility, resolution, and other features listed on the printer's website. When implementing machine plans for these printers, our choice of blocks varied based on each printer's physical construction. For example, the Prusa's construction uses a build platform for the in-progress print that moves side-to-side, whereas the Ultimaker's platform only moves vertically. As a result, the in-progress print for the Ultimaker never moves side-to-side since all such motion is accomplished by the uppermost assembly which only moves the extruder head. This construction is useful when printing tall and skinny features (Figure 3.9); however, the Prusa is cheaper, and, depending on the user's model, such construction might not be necessary. In addition, as attribute properties for each actuated block, we identify its type of drive mechanism (e.g., lead screw, timing belt, or a rack and pinion). This information could be used to implement rules of thumb that optimize machine choices based on choice of drive train, e.g., filtering for machines with all lead screw actuators due to their ability to handle high-force applications.

Makers use filtering rules of thumb to compare different machines. Again, the choice of rules of thumb for filtering rather than built-in sorting features is because the requirements that makers would want to filter around change based on the task as well as over time as machines change. However, in this case, a rule of thumb may need additional information from users; for example, users might not have a particular model in mind, yet they still want to see only 3D printers with work envelopes larger than a certain value. To support this, we implemented *dependent rules of thumb*, which take into account a value from the user when filtering (see Figure 3.10). For example, the dependent rule of thumb named `''work envelope is larger than ___''` prompts the user to enter a bounding box with a desired width, height, and length. The rule then filters for machines that have a work envelope with a box shape whose dimensions equal or exceed the values listed. The notion of dependent rules of thumb comes from the concept of dependent types in programming languages, which are types (e.g. `String` or `Array`) that depend on a value—for example, a type representing all arrays of length 3 [BD09]. In this light, this dependent rule of thumb implicitly assigns a dependent type to all machines based on the dimension of their work envelope and filters them accordingly.

3.5.3 Cutting a Styrofoam Airfoil with a Hot Wire Cutter

Hot wire cutters work by heating a suspended wire with an applied voltage and then moving the wire through a work piece of styrofoam, which cuts the material by melting it. In this example we model a typical CNC (Computer Numerical Controlled) hot wire cutter topology with two degrees of freedom, namely, vertical and horizontal wire movement (see Figure 3.11). We observed that both redundant linear blocks consume significant space for their bounding boxes that, in reality, are empty space where the tool (the wire) and the workpiece would go. In the hot wire cutter's implementation, blocks in this machine readily intersect with one another. Assume we want to create rules of thumb that distinguish between an intersection that is physically harmless versus an intersection where machine parts are actually crashing into one another. A "harmless" intersection in this case is how the foam work piece's volume might intersect with the vertical machine block's empty space. An unacceptable intersection occurs when the work piece intersects with the vertical block's solid parts. To this end, we implemented the notion of a *void*, an optional

```
(machine, depValue) => {
  try {
    /* Assume the dependent value
       is a string in the form
       "[width, height, length]" . */
    let dimArray = JSON.parse(depValue);
    /* Javascript destructures dimArray
       into three variable assignments.
    */
    let [minWidth, minHeight, minLength]
        = dimArray;
    let we = $metrics.workEnvelope;
    return we.width >= minWidth
        && we.height >= minHeight
        && we.length >= minLength;
  }
  catch (e) {
    return false;
  }
};
```

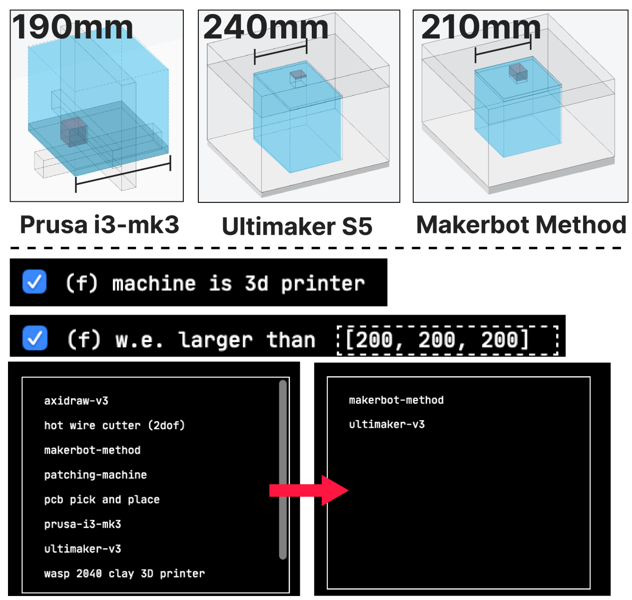


Figure 3.10: Dependent Filtering Rule of Thumb for Comparing 3D Printers. (Left) Implementation of the rule of thumb that takes an additional argument from the user interface, `depValue`, representing the minimum allowable size of the machine's work envelope. (Right, Top) The three commercial 3D printers in our evaluation rendered in the scene which the work envelope highlighted in blue. (Right, Bottom) The checked rules of thumb filter the machine plan list down to only two satisfactory machines.

property that can be added to a block's attributes property. A void is a bounding box that fits in the block's bounding box that represents empty space within the block. As a result, we can be confident that other blocks intersecting with a block in its void will not cause a collision. We implemented a rule of thumb to check that a machine does not collide with itself, referencing blocks and voids. Though this rule is not strictly necessary for the hot wire cutter, it would likely be useful for more complex machines and workflows where collisions are more likely to happen.

3.5.4 Printing with Clay

We implemented a Delta Wasp 2040 Clay 3D printer and an example workflow for it that 3D prints a vase out of clay [Was19]. This printer uses delta bot kinematics, which feature three levers connected with revolute joints to a rigid body. The end of each lever that is not fixed to the rigid body moves independently via a drive mechanism, and the position of the levers' ends determines the location of the rigid body. As a result, the rigid body has three Degrees of Freedom (DOF) and depends on three or more motors to drive the levers, which exceeds the linear (1 DOF) and cross (2 DOF) blocks that we have thus far implemented. Therefore, we implemented a 3DOF `DeltaBotBlock` that encompasses the majority of the printer; in fact, the printer's machine plan has only three blocks: the delta bot block, the tool, and an immobile platform. One feature of delta bot kinematics is a cylindrical work envelope as opposed to a box-shaped one, which we implemented as well.

In addition to the delta bot block and the cylindrical work envelope, we could not feasibly address other important features in the Wasp printer with the current level of abstraction in Taxon. In formative interviews with makers, we learned that a key challenge in 3D printing with clay comes from the material properties of clay itself. In all such 3D printers, the clay must be packed into a container tube and forced evenly through the nozzle. The Wasp printer uses an

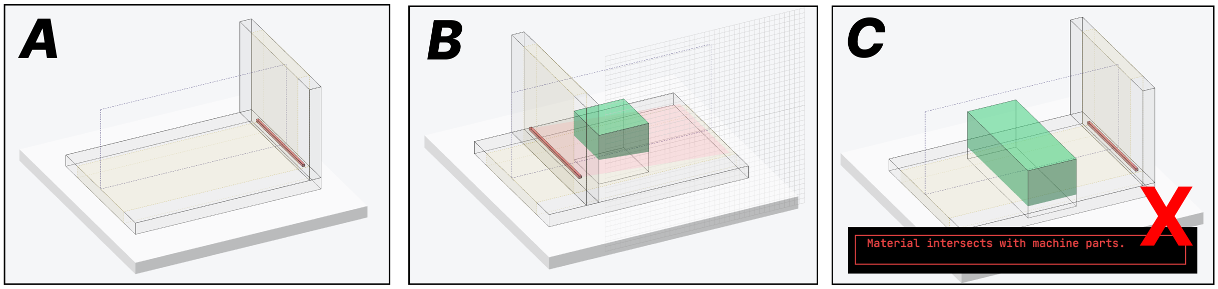


Figure 3.11: Hot Wire Cutter Example. A) The hot wire cutter rendered in the scene, where the red block is the tool (wire), the dashed line is the work envelope, and the goldenrod regions are user-defined voids within the non-tool blocks (gray). B) A simple workflow consisting of placing a block of foam material, zeroing the machine, and moving the wire to cut through the foam with the cut plane illustrated in pink. C) If the material is placed in such a way where it intersects the blocks outside the voids, i.e. overlapping with machine parts, it triggers an error from a rule of thumb that requires that materials may only intersect blocks in their voids.

external compressed air source along with a rotating auger for this. Unlike the plug-and-play nature of plastic 3D printer filament, here the maker must mix their own clay and find the right consistency: too watery and the extruded material may collapse, or too thick and the clay will extrude unevenly. One ceramicist noted that she knew from her own experience with clay what an ideal consistency felt like, how to vary it based on the type of print she was attempting, and when and when not to manually intervene during a print to reshape features by hand. To codify this sort of tacit knowledge, Taxon would need to support materials with enough detail to do basic physics simulation, and, even then, such rules of thumb based on the material properties of clay would be "ballpark" estimates at best. We decided that this level of physics-based material representation is beyond the current scope of Taxon but would be fruitful future work.

3.5.5 Bio-actuated Textiles

In this demonstration, we replicated a workflow that Wang et al. demonstrated with xPrint, a machine with modular tooling for printing with liquid-based smart materials [WYW+16]. They

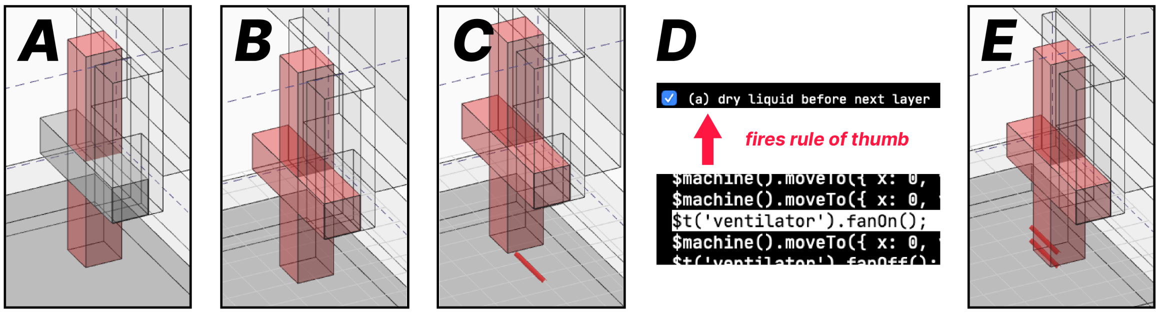


Figure 3.12: Depositing Natto Cell Culture with xPrint. A) This xPrint configuration contains a dispenser (red) and an inactive ventilator (dark gray). B) An action activates the ventilator (now red), allowing the user to turn its fan on and off. C) Extruding a thin layer of culture with a `moveTo` action. D) Calling `fanOn` dries the current layer, which triggers the relevant rule of thumb "dry liquid before next layer" to update its state. E) Depositing the second layer. If the first layer were not dried with the fan, the rule of thumb would have thrown an error.

created discrete components—such as a liquid dispenser, a mechanical stirrer, and a ventilator unit—that can be manually attached and removed as necessary for the desired workflow. In addition, they created a plug-in for the Grasshopper/Rhinoceros CAD program that lets makers create custom tool paths for the machine. We focused on one workflow they presented: depositing liquid natto cell culture onto textiles, which makes the textiles curl when exposed to moisture (see Figure 3.12). Depending on the pattern of culture deposited, makers can control the direction and degree of curl. In Wang et al.'s paper, this workflow involved attaching a solution container equipped with a mechanical stirrer for keeping the natto cells suspended, a dispenser for dispensing the culture, and a combined ventilation unit in addition to a heated bed to evaporate the current layer of culture before the next layer is deposited.

Despite the complex nature of tooling in this workflow, we found that implementing the machine plan and the workflow in Taxon was less challenging than expected. This is largely because much of the complexity comes from the choice of tooling to install, rather than from the process of choosing or controlling the machine. While multiple tool components can be attached to the machine at any given time, once attached, no further automated tool changes are involved.

In addition, the machine itself is a fairly straightforward 3-axis machine, which we implemented using three linear blocks: one non-actuating block for the build plate, one non-actuating block for the tool component substrate, and non-actuating tool blocks for the dispenser, solution container, and ventilator. We implemented rules of thumb that govern which tools must be equipped to work with a given material. In our case, if the xPrint machine were selected, and if natto cell culture were used as a material, then the machine would need a dispenser, solution container, stirrer, and ventilator. To implement the notion of "active," we added a new property to a tool block's attributes property, called `active`, which can be true or false. If false, while the tool is included in the machine's plan, it is not rendered to the scene and cannot be accessed by the action interpreter unless marked with the `.activate` method. We also implemented a rule of thumb (shown in Figure 3.13) that requires the maker to dry the current layer with the fan before moving on to the next layer.

3.5.6 Robotic Assembly of PCB Components

We implemented the Liteplacer pick and place machine, an open source, do-it-yourself machine kit created by Juha Kuusama [Juh21]. This machine picks up SMD components for printed circuit boards and places them precisely within the components' respective footprints on the PCB board itself. The machine receives as input locations of the footprints on the PCB and uses a camera-based localization routine to zero itself and calculate trajectories for moving components from the supply region to the footprints.

This is a relatively complex machine and workflow. For the machine plan, we implemented the machine to use two linear blocks for both the lower assembly, which moves along the x axis, and the crossbar assembly, which moves on the z axis. The tool assembly was more complicated to model, consisting of a linear block that moves on the y axis, a *rotary block* that rotates any

```
(action, store) => {
  try {
    /* If we are turning the fan on, then
       we can assume that the most
       recently deposited layer will have
       dried. */
    if (action.methodName === 'fanOn') {
      store['highestLayerWet'] = false;
    }
    if (action.methodName === 'moveTo'
      && action.args[1] !== undefined) {
      /* If we are depositing more culture
         on top of a layer that has not
         been dried, signal an error and
         return failure. */
      if (store['highestLayerWet']) {
        console.error('Must dry the previous layer first.');
        return false;
      }
      /* Otherwise, deposit the culture and mark the current layer
         as wet. */
      store['highestLayerWet'] = true;
    }
    return true;
  }
  catch (e) {
    console.log('Cannot enforce "dry liquid before next layer."');
    return true;
  }
}
```

Figure 3.13: Implementation of the "dry liquid before next layer" Rule of Thumb. This action rule of thumb accesses the `store` argument which references a state that the program keeps between different executions of the rule of thumb. This allows the implementation to keep state for whether the most recent layer was dried by the fan or not.

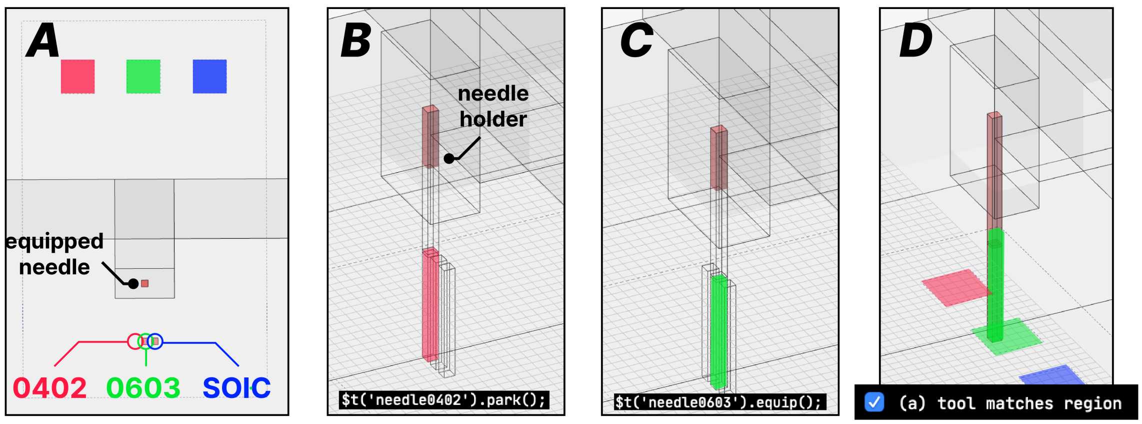


Figure 3.14: Verifying Correct Tool Configurations in the Pick and Place Machine. A) A bird's-eye view of the pick and place machine's setup with three differently sized needles for picking up SMD components of respective sizes (0402, 0603, and SOIC-8) and placing them on a PCB. At the bottom are three regions for parking each needle and at the top are three regions where each type of component is stored before being picked. B) The needle holder travels to the 0402 parking region and parks the needle there. C) The needle holder travels to the 0603 parking region and picks up that needle. D) The needle holder moves to the 0603 storage region successfully because the 0603 needle is currently equipped. If the needle had moved there before swapping needles, the "tool matches region" rule of thumb would have thrown an error because the needle would be incorrectly sized for picking up 0603 components.

```
(action, store) => {
  /* Try/catch logic is omitted here. */
  if (action.methodName === 'moveTo') {
    let moveArgs = action.args;
    let movePt = moveArgs[0];
    /* Translate the moveTo method's
       target coordinates into the world
       position the tool would occupy
       after the action executes. */
    let toolPos = $kinematics
      .coordsToWorldPosition(movePt);
    let equippedTool = $machine
      .getEquippedTool();
    /* Iterate through all envelope
       regions and check whether the
       currently equipped tool matches
       the region's required tool. */
    $metrics.envelopeRegions
      .forEach((er) => {
      if (er.containsPoint(toolPos)) {
        if (equippedTool.attributes
              .pcbSize !== er.name) {
          console.error(`The ${er.name} needle must be equipped to enter the region.`);
          return false;
        }
      }
    });
  }
  return true;
};
```

Figure 3.15: Implementation of the "tool matches region" Rule of Thumb.

PCBs that are currently picked up, and several suction-cup-tipped needles for actually picking up the SMD components. We chose not to implement rotary blocks for now because rotational motion complicates our model of machine motion; it would require us to solve general-case inverse kinematics rather than the simpler cases with only linear motion. Instead, we focused on the machine's tool changing feature and on correct tool usage, as shown in Figure 3.14.

Given our definition of a tool block as the block that interfaces with material, we designated only the needles as tools; this means that needle blocks are marked as `''isTool'': true` and support tool-based methods when selected in actions. There are several sizes of needle for picking up different types of SMD components and we named each needle after the size of component it can pick up, i.e., 0402, 0603, and SOIC-8. These needles are parked in a needle holder in the work envelope, and the machine must perform a tool changing operation to attach the needle of the correct size before picking up components of that size. Typically, programmers must manually program these tool changing routines either in raw G-Code or in whatever convention the program controlling the pick and place machine's control board uses, which is prone to errors.

In this light, a goal of Taxon is to codify best practices—here, ensuring that the correct needle is attached before attempting to pick up an SMD component. Enforcing this logic necessitated the creation of a new feature, *envelope regions*, which lets the programmer designate and label subspaces within the work envelope. Similar to a work envelope itself, an envelope region can be specified as a box that checks for inclusion within the region based on x, y, and z coordinates or as a rectangle that checks only two coordinates. We created a rectangle XZ region where each needle is parked, a region in which to place stock SMD components corresponding to each size of needle, and a region in which to place the PCB itself. An automated tool changing routine would involves traveling to the region for parking the current tool, having the machine deposit the current tool, traveling to the region to park the new tool, and then attaching it to the tool holding assembly. We then implemented an action rule of thumb (shown in Figure 3.15) stating that before a tool can enter a region for stock SMD components, it must have the correctly sized needle currently attached.

3.6 Limitations

While implementing the six demonstration programs for the evaluation, we encountered several limitations. First, simulating a workflow often requires us to reason about how a material will react when acted upon by the machine. For example, in the case of the hot wire cutter, having the wire pass through a styrofoam workpiece put foam in the path of the wire. Or, for the clay 3D printer, the consistency of the clay greatly affects its behavior when deposited by the extruder. However, modeling the physical behavior of material—even in a greatly simplified setting—remains challenging. Because material properties always affect the semantics of actions, they must be implemented in the core Taxon language, as opposed to being added in a modular fashion using rules of thumb. The need to incorporate more physics into Taxon than we initially anticipated created a bottleneck that we must address before more workflows can be fully represented in code.

In addition, nontrivial geometry processing is not yet implemented in Taxon. For the evaluation, we attempted to replicate the workflow proposed in a paper by Teibrich et al., which features a machine that can patch existing 3D printed objects by removing material with a mill and then reprinting new material with an 3D printer extruder [TMG+15]. These subroutines depend on access to the 3D model, the tool path, and robust software for detecting collisions. While it is possible to implement these subroutines as rules of thumb, such as "avoid collisions by milling material that blocks access" and "rotate the build plate to minimize milled material", we would need to add substantial geometry processing to implement such collision detection and optimization programatically. Currently in Taxon, we can reason only about collision with bounding boxes, not with the precise geometry of the work piece. We defer such features to an extension in future work.

Finally, the codifiable space of possible concerns in digital fabrication is vast; Taxon can address only a small subspace of these concerns in its current form. In this initial implementation, we intended to codify concerns that would present the highest barrier to novices, namely, the capabilities of each machine from a high level, a machine's composition, and its basic action, e.g., "extrude" versus "cut." Many lower level concerns, such as software-supported bed leveling or smoother motor movement through microstepping, are difficult to represent. However, a key benefit of designing a new programming language is that we can organize these concerns into various levels of abstraction. For example, higher levels of abstraction could abstract away details like extrusion rate, whereas lower level ones could let programmers add detailed information about machine parts in the blocks. We envision creating this stack of abstractions to be a crucial next step for expanding the language going forward.

3.7 Future Work: Affordances of Formal Representation

Although Taxon currently handles a limited space of concerns, it is a language that contributors can extend both by authoring rules of thumb, machine plans, and workflows and also by implementing features in the language itself. This is a massive step forward from not having expertise about machines codified at all. A growing formal representation of digital fabrication machines will enable an ecosystem of tools that can reason about what machines are, how they move, how they interact with materials, and what is considered to be appropriate actions. We aim to add more robust support for materials and digital models in future iterations. In addition, because Taxon helps programmers reason about digital fabrication tool use in software, we envision several newly possible lines of work.

- **Structured Querying for Machines**. Similar to how the Voyager tool enabled structured exploration of visualizations based on their grammars [WMA+16], we could allow for structured exploration of machine options beyond filtering rules of thumb given a user-specified workflow. Makers would then be able to explore trade-offs along speed, precision, cost, and other factors that are influenced by machine choices for the workflow.

- **Optimizing Instructions for Machine Kinematics**. In many machines, dedicated control software translates instructions, e.g., in G-Code, into motor pulses to optimize machine movement based on its physical characteristics. Given a Taxon program that lets users infer volumes, masses (in the future), and kinematics, it would be possible to optimize machine instructions for any machine rather than hard-coding these optimizations for one machine. One example optimization is *input shaping*, which involves generating signals that cancel vibrations associated with moving machine parts and result in higher quality surface finish [Rob02].

- **Online Infrastructure for Sharing Machines and Workflows**. Taxon makes possible a rich online ecosystem of machine and workflow descriptions. Members of online communities could share their custom machine builds and small production tasks as Taxon programs. Other members could run these custom setups in simulation and remix Taxon programs. Taxon could serve as a quasi API for creating Instructables-like tutorials with interactive previews built into each step. In addition, existing open-source specifications of machines such as Cura's 3D printer profiles [Ult21] could be ported to Taxon machine plans.

- **Scheduling High-Throughput Production in Existing Maker Spaces**. Maker space managers could create a list of machine plans that represent the machines they have available in their maker space. Given a large production task, such as producing personal protective equipment requiring multiple machines, programmers could write tools that solve for the optimal allocation of production tasks to machines in the maker space.

- **Program Synthesis for Fabrication.** We can view a digital fabrication pipeline as a compiler that compiles a digital model into a physical object. In computational fabrication, *inverse design problems* solve for digital models given some physical constraints, e.g. the design of nanophotonic devices given nanoscale fabrication limitations [PPSV17]. However, inverse problems to solve for machines themselves have not been explored. By providing a formal specification and programmatic description for machines, Taxon could let researchers use program synthesis techniques [SLTB+06, VG19] to infer a machine specification that best suits the fabrication of a digital model under physical constraints.

3.8 Conclusion

We contributed and described the design of Taxon, a grammar for formally specifying abstract machine properties that enables description, comparison, and simulation of physical machines. The promise of digital fabrication lies in how it might allow new practitioners to create objects that uniquely suit their own contexts. The capabilities and limitations of machines will always be important factors in both making and research on making, and they ought to be made explicit. We aim for Taxon to become a useful standard in fabrication research and practice that lets researchers integrate machine-level concerns into novel tools and provides makers with helpful infrastructure to guide their manufacturing processes.

Chapter 4

Visual Reasoning through Toolpath Stylesheets

When building and iterating on workflows, fabricators must understand what the machine will do when it executes a toolpath. Whereas software engineers can test their programs by repeatedly running unit tests, digital fabrication programs are time-consuming and wasteful to execute repeatedly. Instead, fabricators often rely on visualizations to identify potential issues, such as whether the toolpath is placed in the wrong location, before executing the toolpath on the physical machine. Further, visualizations are vital ways to understand how data and materials fit together in a workflow.

Existing GUI tools typically include visualizations of digital model geometries modified for the manufacturing technique, e.g., 3D printing or milling. However, visualizations provided by state-of-the-art tools are typically not customizable and visualize only geometry, which obscures potentially crucial low-level information available only at the instruction level.

As an example of when a custom visualization would be useful, consider the use case of a laser cutter that is cutting fine-grain patterns generated by a fabricator's script. In this case, an

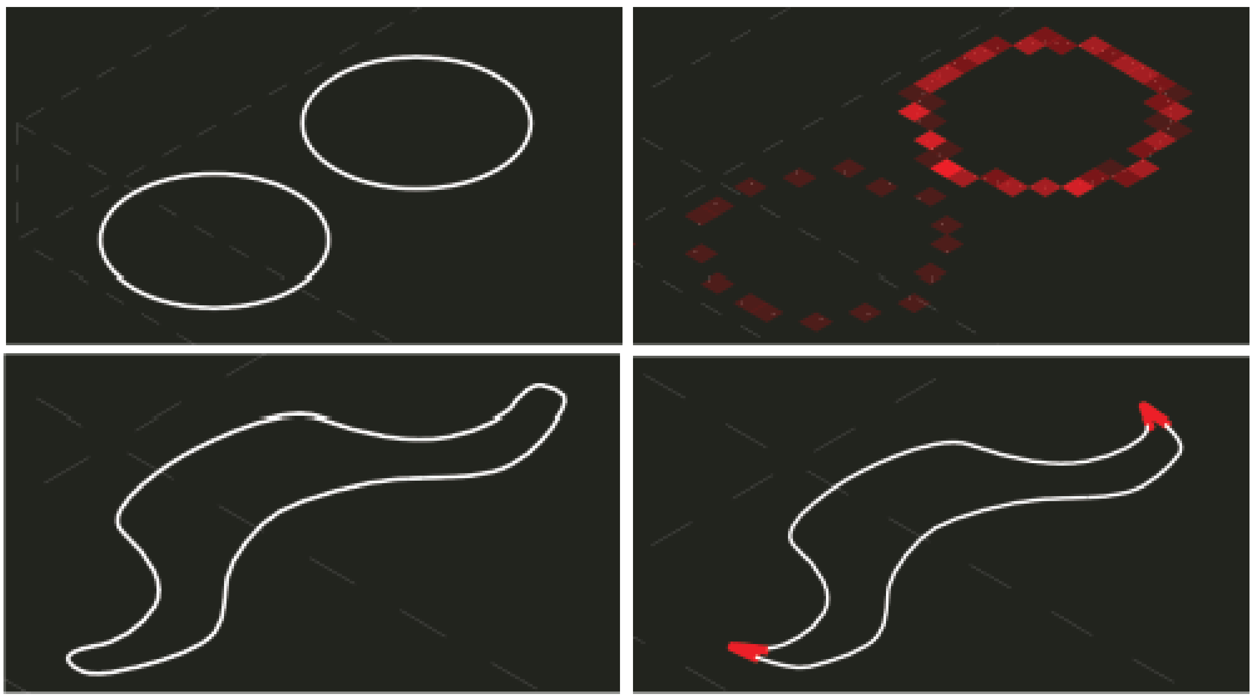


Figure 4.1: One-to-One Compiled TSS Highlighting Salient Details. Top left: the geometry of a toolpath of two circles is shown on the left. The expected energy spent from laser cutting is shown on the top right. This TSS highlights that the rightmost circle contains duplicate paths; excess heat applied could ignite the material. Bottom left: sharp corners in a toolpath could pose problems in gel extrusion. The TSS on the bottom right highlights any sharp corners below a threshold angle set by the fabricator.

inevitable concern with laser cutters is the heat the laser generates. If the laser cuts too much in a small area in a short time, the material being cut will become scorched or even combust. A fabricator must carefully adjust several parameters, including the size, position, and geometry of that cut, as well as the laser's power and speed.

Unfortunately, typical toolpath visualizations do not reveal any of this heat sensitivity information. The only way to reason about scorching or possible combustion when running the job on the machine is to run many cuts and repeatedly adjust parameters, even though running multiple cuts could result in dangerous conditions. Though experimenting with test cuts is a useful technique, a goal of physical-digital computing is to accumulate knowledge of physical behavior as gleaned by experience for others to benefit so they can potentially avoid unsafe consequences.

In this case, how could we do better than guess-and-check understanding of nuanced, specific machine-material interactions that play a major role in developing experimental workflows? Our insight is to provide fabricators with multiple "views" of a given toolpath, where each view calls visual attention to a specific aspect or concern that the fabricator might have about a toolpath, e.g., the distribution of thermal energy with respect to time when laser cutting. While such niche visualizations might be less necessary for more conventional laser cutting tasks, experimental ones are riskier since a fabricator might not be sure that the cuts are even feasible.

In this sense, we draw an analogy between concern-specific visualizations and *continuous integration* in digital-only software engineering practice. In the latter case, continuous integration runs a programmer-defined test suite on potentially risky code before code deployment and requires tests to pass. Similarly, concern-specific views would let a fabricator visually confirm that nothing is "wrong"—like a strong chance of scorching the material in a particular area—before deploying the toolpath physically to a machine. Note that visualization would be useful here as opposed to pass-or-fail tests because there is often no clear boundary for when to reject a toolpath. Moreover, just as a digital-only programmer would run many tests, a fabricator would have access to many views of a single toolpath to check for a broad range of concerns.

We then formalize this notion of a concern-specific view, which we call a *toolpath stylesheet (TSS)*. A TSS is a mapping from a toolpath, which we define to be an ordered collection of low-level machine instructions like G-Code, to a collection of machine parameters, such as maximum tool acceleration or laser power, and a visual output. That is, for toolpath T, parameters P, and visual output V, define a TSS to be

$$TSS : T \times P \rightarrow V.$$

Different TSS map the same set of instructions and parameters to different visualizations that can be visually dissimilar, the idea being that each TSS should expose different aspects of machine behavior. For example, given two TSS—TSS_{heat}, which produces the aforementioned heat map of a laser cutter job and $TSS_{sharp,}$ which highlights sharp angles in a cut—along with a fixed toolpath T_i and parameter collection P_j, we can consider these TSS as mapping the toolpath and parameter to visualizations conditioned to the combination of TSS, toolpath, and parameter collection, as follows:

$$TSS_{heat}(T_i, P_j) \rightarrow V_{heat_{ij}}$$
$$TSS_{sharp}(T_i, P_j) \rightarrow V_{sharp_{ij}}.$$

The visual output of both TSS is shown in Figure 4.1. Finally, note that the content of P could possibly depend on a TSS, e.g., a TSS that is not meant for a laser cutter would not require laser power to be passed as a parameter in the collection.

4.1 One-to-One Compilation

The first way we could conceptualize building a TSS is by mapping each machine instruction to a distinct visual element. With this viewpoint, a TSS is a compiler that takes a line of code in an input language—in our case, machine instructions like G-code—and outputs code as a graphical element in a visualization as represented by code in a visualization language. In our implementation, we used the Three.js library [Cab14] as a target language; Three.js generates meshes of geometric primitives such as cylinders that can be composed into more complex 3D visualizations.

Given a one-to-one approach, we can think of a TSS-as-compiler as implementing an *alternative semantics* for a given machine instruction. For example, a typical "semantics" of two tool movement instructions (e.g., the `G1` command in G-code) on a vinyl cutter with the cutting blade down is to produce two cuts in the physical world. Under TSS_{sharp}, which highlights sharp angles in a toolpath, the compiler instead generates two corresponding visual elements to represent the tool movements, checks if the lines form an angle below the minimum threshold in P, and generates a highlight if so.

One advantage of considering a TSS to be compilation is that we can think at the instruction-level of machine behavior: from the bottom up rather than the typical top-down viewpoint. If a fabricator wants to implement their own TSS, they need only think of how instructions (such as movement commands or blade pickups and set-downs) map to discrete visual elements in an output visualization.

However, this same one-to-one simplicity of the approach is also a major weakness: if a fabricator needs to write a TSS where machine instructions do not map cleanly to visual output, then designing the mapping from single instruction to visual elements becomes increasingly complicated. At the very least, the fabricator would need to take multiple pre-processing passes over the instructions before generating the visualization. For example, with the implementation

Require: *instructions*
  $points \leftarrow [\ \ ]$
  for *instruction* in *instructions* **do**
    $opcode \leftarrow parseOpcode(instruction)$
    if $opcode$ = "G0" **or** $opcode$ = "G1" **then**
      $X, Y, Z, F, E \leftarrow parseArgs(instruction)$
      $push(points, Vector3(X, Y, Z))$
    end if
  end for
  $curve \leftarrow interpolate(points)$

  $colors \leftarrow [\ \ ]$
  $f \leftarrow ...$ {Choose a constant frequency for cycling colors.}
  $\phi_r, \phi_g, \phi_b \leftarrow ...$ {Choose a constant phase offset per channel.}
  for $(\_, index)$ in *curve* **do**
    $red \leftarrow sin(f \times index + \phi_r)$
    $green \leftarrow sin(f \times index + \phi_g)$
    $blue \leftarrow sin(f \times index + \phi_b)$
    $push(colors, Color(red, blue, green))$
  end for

  $meshes \leftarrow [\ \ ]$
  for $(segment, index)$ in *curves* **do**
    $color \leftarrow colors[index]$
    $mesh \leftarrow Mesh(curve, color)$
    $push(meshes, mesh)$
  end for

  return *meshes*

Figure 4.2: One-to-One Approach for a TSS that Maps a Instructions in a Toolpath to a Color Spectrum. This implementation assumes a toolpath using G-code instructions.

of TSS_{heat} described in Figure 4.3, there is no immediate correspondence between an instruction and a shaded area on the heat map. A fabricator implementing this TSS must first bin write the logic to sample the toolpath, bin the samples across subspaces of the work envelope, and color the bins based on relative frequency of samples in each bin.

4.2 Grammar of Graphics-Based Compilation

To address the issues raised by one-to-one compilation, we naturally turn to techniques from data visualization. In particular, we identify the *grammar of graphics* as the approach from which we borrow [WW05]. The grammar of graphics is a theoretical framework for organizing visual elements in a visualization where the elements are categorized and formalized in a grammar. A visualization is decomposed into axes, marks, transforms, and more high-level concepts that are specified declaratively.

To illustrate why this approach gives us more expressive power for building a TSS, consider the plots shown in Figure 4.8. With a one-to-one mapping approach, we could straightforwardly generate the line plot (top-left) by just placing a line on the graph for the line segment corresponding to a move instruction. However, for the histogram (top-right) and density (bottom-left) plots, we encounter the same problem we observed with TSS_{heat}, i.e., we must compute many intermediate processing steps to generate different visual elements, even though the underlying data is the same. In contrast, because the grammar of graphics is declarative, we need only add a *transform* to concisely obtain a large visual change that provides a completely different view of the same data. It is exactly this declarative, expressive language that lets a TSS quickly visually ascertain highly distinct concerns given a fixed toolpath and parameter collection.

Require: *instructions*
  $curve \leftarrow$ ... {Build a curve as done in the previous TSS.}
  $rate \leftarrow 100$ {E.g., 100 to sample a point every 100ms.}
  $points \leftarrow sampleCurve(curve, rate)$

  $binHeight, binWidth \leftarrow$ ...{Choose bin sizes for the histogram.}
  $grid \leftarrow [\ ][\ ]$
  $setZeroAll(grid)$
  for *point* in *points* **do**
    $row \leftarrow \frac{pt.y}{binHeight}$
    $col \leftarrow \frac{pt.x}{binWidth}$
    $grid[row][col] \leftarrow grid[row][col] + 1$
  end for

  $maxCount \leftarrow max(grid)$
  $meshes \leftarrow [\ ]$
  for (row, col) in *grid* **do**
    $count \leftarrow grid[row][col]$
    $opacity \leftarrow (\frac{count}{maxCount})^3$ {Exponent darkens low counts.}
    $box \leftarrow boxAt(row \times binHeight, col \times binWidth)$
    $mesh \leftarrow Mesh(box, opacity)$
    $push(meshes, mesh)$
  end for

  return *meshes*

Figure 4.3: Naïve Heat Map TSS Using One-to-One Compilation. This approach makes it difficult to reason about the temporal closeness of points, namely, that we should bin sampled points only if they occur close to each other in time. Otherwise, laser passes over an area of space might get binned even if they occur far apart in time and thus have little risk of causing a fire.

4.2.1 Compilation Approach

To implement a TSS using a grammar of graphics approach, we take three passes over the data, again beginning with initial machine instructions and ending with visual *marks* on plots.

1. Lower the machine instructions into a simulated machine trajectory. A trajectory comprises a series of segments each with an initial and final velocity. These segments provide velocity, acceleration, and time data for downstream visualization.

2. Transform the trajectory into one of several possible intermediate representations (IRs). This is akin to parsing tokens into an abstract syntax tree in traditional compilation.

3. Visualize the intermediate representation by declaring marks in the visualization grammar. A mark is a mapping from data—here the IR—into visual elements. This step is similar to the code generation step of traditional compilation.

In our implementation, we use the Javascript-based Observable Plot grammar [Obs23a], which is similar to the Vega-Lite grammar [SMWH17], itself an implementation of the grammar of graphics with support for interactivity.

Using a declarative grammar-based approach rather than being forced to consider one-to-one mappings of instructions to visual elements frees us from two crucial limitations. First, it lets us more readily create visualizations where visual elements are not necessarily mapped to 3D space. With a compiler-based approach, we would tend to map instructions to where things "happen" in 3D space. In contrast, with an expressive grammar, a TSS implementor has the opportunity to explore diverse representations established in data visualization literature that do not map directly to 3D, but that more effectively call attention to important aspects of a toolpath.

Second, we step away from the imperfect analogy of a machine instruction as an assembly instruction for a CPU. While assembly code comprises primitives that must necessarily be sup-

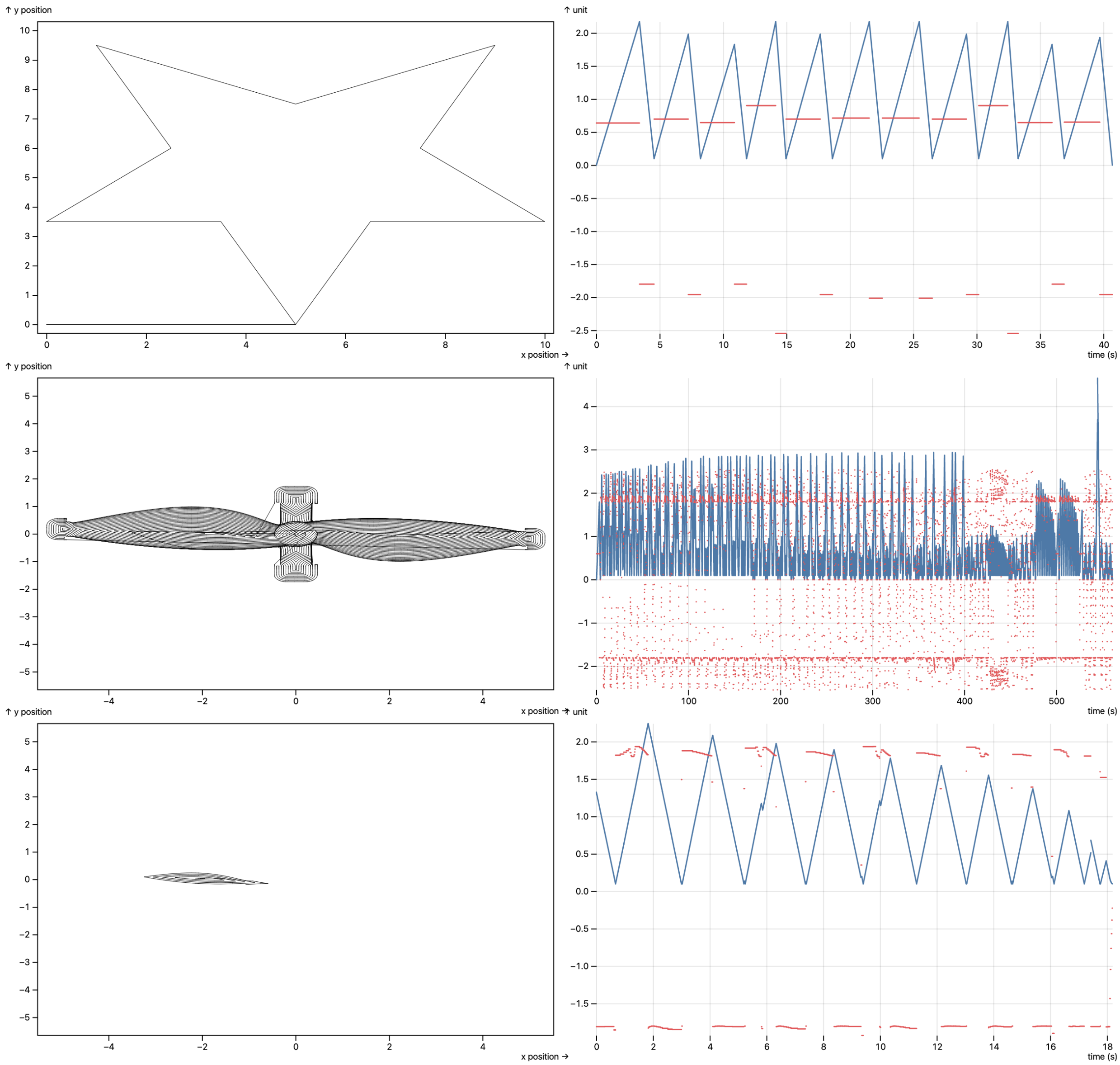


Figure 4.4: Toolpaths (Left) and Respective Trajectories (Right). In the first row, the trajectory for the star toolpath shows the velocity (blue) dropping to a pre-defined junction velocity close to zero as the tool turns the star's corners. The second row shows the trajectory for a finishing pass for milling a propeller, while the third shows just a subset of the same toolpath. All units are in inches (position), in / s (velocity, blue) and in / s^2 (acceleration, red).

ported by its CPU's architecture, the same is not true for instructions for physical machines, whose execution could have multiple possible outcomes in the physical world depending on the machine's kinematic limits (e.g., maximum acceleration and velocity ramping profile), among other physical realities of a machine. By processing instructions into velocity and acceleration data points, like machine controller firmware would, we unlock finer-grain visualization than even many off-the-shelf toolpath visualizers provide, such as the one found in Autodesk Fusion 360 [Aut22].

4.2.2 Lower Instructions to a Trajectory

The first step in the process is to generate a trajectory that the machine's tool will follow. The trajectory implements the positional requirements specified in machine instructions, but also adds velocity, acceleration, and time information that make the toolpath physically possible. These additional physical parameters give us far more data to transform and visualize than machine instructions alone.

Concretely, a trajectory is composed of segments, where each segment moves from a *start* to an end in $\mathbb{R}^3$, thus also representing motion along a unit vector $\hat{u}$. Each segment has a *profile* describing its kinematic information: its start and end velocity (v_0, v), its acceleration a, the time t it takes to complete the segment, and the magnitude of movement m. The main idea behind trajectory generation is set the velocities of the segments as high as possible while also respecting the need to slow around corners, the kinematic limits of the machine, and the need to accelerate before reaching a given velocity. Figure 4.4 displays examples of toolpaths and their respective trajectories.

In the implementation described in Figure 4.5, we assume that all segments are lines with a linear velocity profile; each segment can only accelerate a constant amount and can only move in

Require: $instructions, kl$ {kl stands for "kinematic limits" provided by the fabricator.}
$zeroPass \leftarrow sample(instructions)$ {An array of line segments that comprise the trajectory.}

$forwardPass \leftarrow [\ \ ]$
$v_{start} \leftarrow 0$
for ls in $zeroPass$ **do**
  $v_{scale} \leftarrow maxVelocityScaling(ls.\hat{u}, kl.\vec{v_{max}})$ {$ls.\hat{u}$ is segment's unit vector.}
  $v_j \leftarrow calculateJunctionVelocity(ls, ls.previous, kl)$ {Slow down as needed for corners.}
  $v_{start} \leftarrow min(v_{start}, v_j)$
  if $(ls.v_0 > v_{scale}) \lor (ls.v > v_{scale})$ **then**
    $ls \leftarrow clipVelocities(ls, v_{scale})$
  end if
  if $|ls.a| > ls.a_{max}$ **then**
    $ls \leftarrow clipAcceleration(\frac{ls.a_{max} \times ls.a}{|ls.a|})$
  end if
  $(lsA, lsB) \leftarrow planSegment(ls, v_{start})$ {Two half-length segments that may change slope.}
  $v_{start} \leftarrow lsB.v$
  $forwardPass.push(ls, lsB)$
end for

$backwardPass \leftarrow [\ \ ]$
$v_{start} \leftarrow forwardPass[length(forwardPass) - 1].v$
for ls in $reverse(forwardPass)$ **do**
  $(lsA, lsB) \leftarrow planSegment(ls, v_{start})$
  $v_{start} \leftarrow lsA.v_0$
  $backwardPass.push(lsA, lsB)$
end for

return $reverse(backwardPass)$

Figure 4.5: Trajectory Generation. At a high level, the algorithm divides a toolpath into discrete segments, takes a first pass over the segments' profiles containing initial and final velocities $v0$ and v, and takes a pass in the other direction to correct discontinuous velocities between segments.

Require: $lineSegments, size_w$
  $cost \leftarrow \lambda(d, \Delta t) \rightarrow max(0, \frac{\ln d^{-1}}{\Delta t})$ {Edge case: if $d = 0 \vee \Delta t = 0$, return 0.}
  $i_{center} \leftarrow \lfloor \frac{w_{size}}{2} \rfloor$
  $weightedPoints \leftarrow [\ ]$
  for (ls, i) in $lineSegments$ **do**
    $window \leftarrow lineSegments[i - i_{center} : i + i_{center}]$ {Skip the loop when i is out of bounds.}
    $center \leftarrow window[i_{center}]$
    $residuals \leftarrow [\ ]$
    for (ls, i) in $window$ **do**
      $d \leftarrow \|ls.start, center.start\|_2$
      $subwindow \leftarrow [\ ]$
      if $i < i_{center}$ **then**
        $subwindow \leftarrow lineSegments[i : i_{center}]$
      else
        $subwindow \leftarrow lineSegments[i_{center} : i]$
      end if
      $\Delta t \leftarrow reduce(subwindow, \lambda(t, ls) \rightarrow t + ls.t)$
      $residuals.push(cost(d, \Delta t))$
    end for
    $weightedPoints.push(\{x : ls.x, \ldots, weight : reduce(residuals, sum)\})$
  end for

  return $weightedPoints$

Figure 4.6: Parsing Line Segments to Time-Distance-Weighted Points. For a given segment, the algorithm considers a window of points of nearby segments in the toolpath down-weighted by distance and time away. Segments near many other points in time-distance window are weighted highly, and vice versa for those far from others.

Require: $lineSegments, \theta_{max}, \Delta z_{max}$
  $sharpPairs \leftarrow [\ \ ]$
  for (ls, i) in $lineSegments$ **do**
    $ls_{next} \leftarrow lineSegments[i+1]$ {Skip the case where $i + 1 = length(lineSegments)$}
    if $ls.start.z < 0 \wedge ls_{next}.start.z < 0$ **then**
      if $|ls.start.z - ls_{next}.start.z \leq \Delta z_{max}|$ **then**
        $\theta \leftarrow \arccos\left(-\frac{ls}{\|ls\|_2} \cdot \frac{ls_{next}}{\|ls_{next}\|_2}\right)$
        if $\theta < \theta_{max}$ **then**
          $sharpPairs.push((ls, ls_{next}))$
        end if
      end if
    end if
  end for

  return $sharpPairs$

Figure 4.7: Parsing Line Segments to Sharp Angle Pairs. The algorithm outputs an intermediate representation which comprises pairs of line segments where the interior angle of the segments is less than a fabricator-supplied θ_{max}. Segment pairs whose displacement on the z-axis is greater than Δz_{max} are not considered.

a straight line. Most desktop-class machine controllers discretize curved parts of a toolpath into non-smooth line segments which are short enough as to not compromise precision in most cases. However, more advanced controllers can also produce smooth curves at the hardware level. For now, we sample instructions that specify curves into small line segments, though functionality for curves and higher-order profiles could be added in the future.

4.2.3 Parse to Intermediate Representations using Transforms

After gaining kinematic data, we must transform the data to a representation better suited for task-specific visualizations. Transforms are an established component of the grammar of graphics; their role is to do the "heavy lifting" of getting the data into the right shape for a given visualization. Similar to other grammar of graphics-based libraries, Observable Plot comes with several built-in transforms such as binning, filtering, and grouping. However, in the case of task-specific visualizations for machine behavior, we will need to define our own transforms.

The result of the transforms are intermediate representations that are more amenable to translating into visual elements. For example, in Figure 4.6, for any given line segment, we consider a window of other line segments that are nearby in both space and time. The more other segments are clustered around a given segment, the "hotter" the segment's neighborhood will be. To accurately model this heat, we need time data gained by trajectory generation. The IR produced by this algorithm is a collection of points with additional weights calculated by weighting neighboring segments in the window according to a predefined cost function. As another example, Figure 4.7 outputs an IR of pairs of segments whose internal angle is less than a fabricator-specified maximum angle.

What is the benefit of generating intermediate representations? First, it solves the problem encountered with the one-to-one approach where we need to map low-level machine data to aggregated visual elements. Transforming data into IRs lets us filter, combine, and group kinematic data similar to pre-defined transforms. Second, IRs act as an interface between: the heavy computation of trajectories and post-processing described so far, and the computationally lighter front-end task of actually rendering the data using marks, as described below. These performance considerations become more important as toolpaths become larger and faster interaction response times are needed.

4.2.4 Generate Visual Marks

Now that we have intermediate representations that are well-suited to visualization, all we need to do is pass the IRs to marks in a plot. A mark is a visual primitive that generates visual output depending on the shape of the data passed to it. For example, Figure 4.8 show the same time-

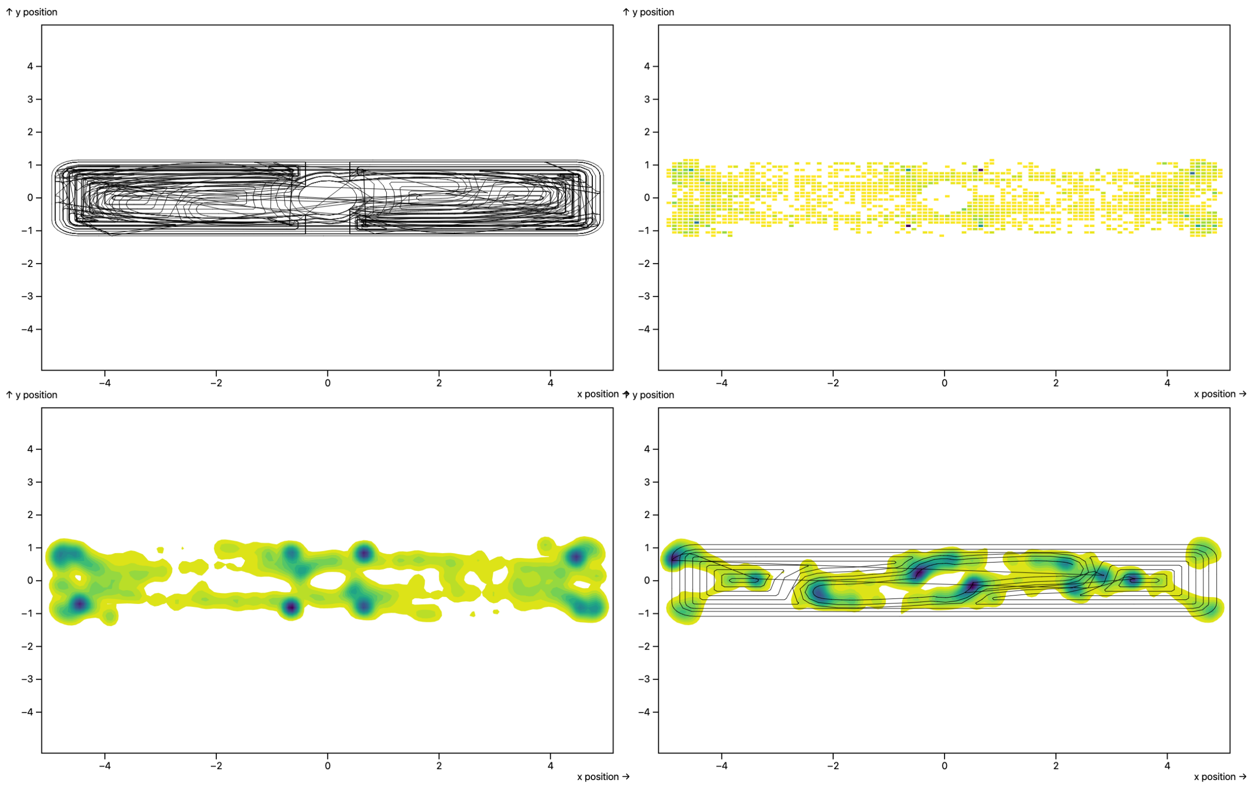


Figure 4.8: Heat Maps using Marks. Top-left) a roughing pass for a propeller with segments rendered with the `line` mark. Top-right) the same toolpath with the time-distance-weighted intermediate representation (IR) rendered using the `rect` mark. Bottom-left) the same IR rendered using the `density` mark. Bottom-right) Slicing only the second milling pass of toolpath with both `line` and `density` marks displayed. Units are inches.

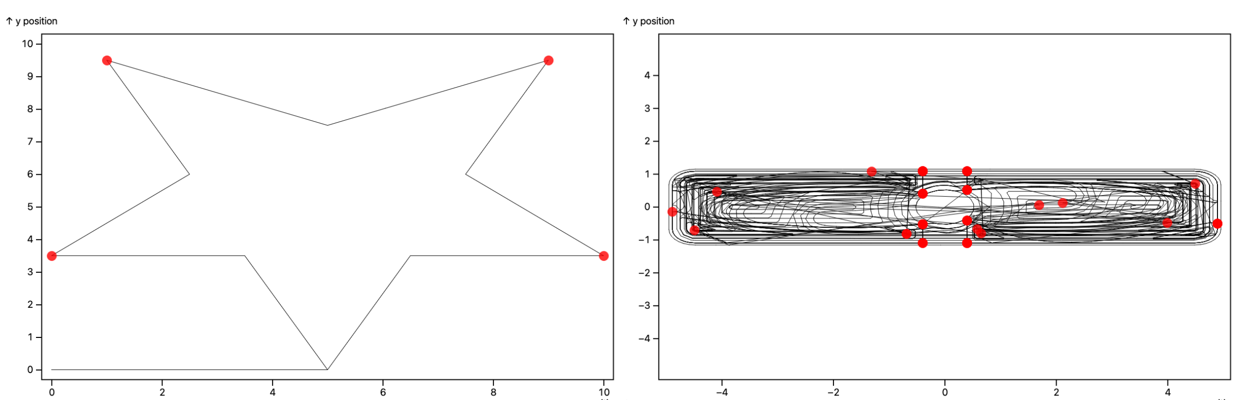


Figure 4.9: Sharp Angle Visualization using Marks. Left) Using the sharp angle detection IR, the plot marks all junctions in the IR with the `dot` mark. Right) the same visualization for the propeller roughing pass. Units in inches.

```
Plot.rect(IR.tdWeighted,
  Plot.bin(
    { fill: 'proportion' },
    {
      x: { value: 'x',
           interval: 0.1 },
      y: { value: 'y',
           interval: 0.1 }
    }
)
```

```
Plot.density(IR.tdWeighted, {
  x: 'x',
  y: 'y',
  weight: (pt) => {
    return pt.z <= 0
           ? pt.weight : 0;
  },
  bandwidth: 3,
  fill: 'density'
})
```

```
Plot.dot(IR.sharpAnglePairs, {
    r: 5,
    opacity: 0.8,
    strokWidth: 0,
    fill: 'red',
    x: (pair)
        => pair[0].end.x,
    y: (pair)
        => pair[0].end.y
})
```

Figure 4.10: Observable Plot Marks for the Heat Histogram, Heat Density, and Sharp Angle TSS. All marks take as data an intermediate representation (here `tdWeighted` and `sharpAnglePairs`) and specify only visual properties.

distance-weighted points being rendered with the `rect` mark (top-right) and with the `density` mark (bottom-left). The top-left and bottom-right plots also contain `line` marks with line segments passed in. Multiple marks can be included in a single plot.

As another example, Figure 4.9 displays the `dot` marks whose X and Y positions are the middle points of pairs in the sharp angle IR. The complete specification of all these marks are written in Figure 4.10.

The major advantage of marks is that, given appropriate IRs, fabricators can rapidly experiment with new visual forms of data by switching out different marks while keeping the rest of the parameters of the plot the same. However, Observable Plot, alongside other grammar of graphics libraries, includes other parameters with which we have not yet experimented, for example: faceting, scales, and fill and stroke channels. More exploration of these features is left as future work.

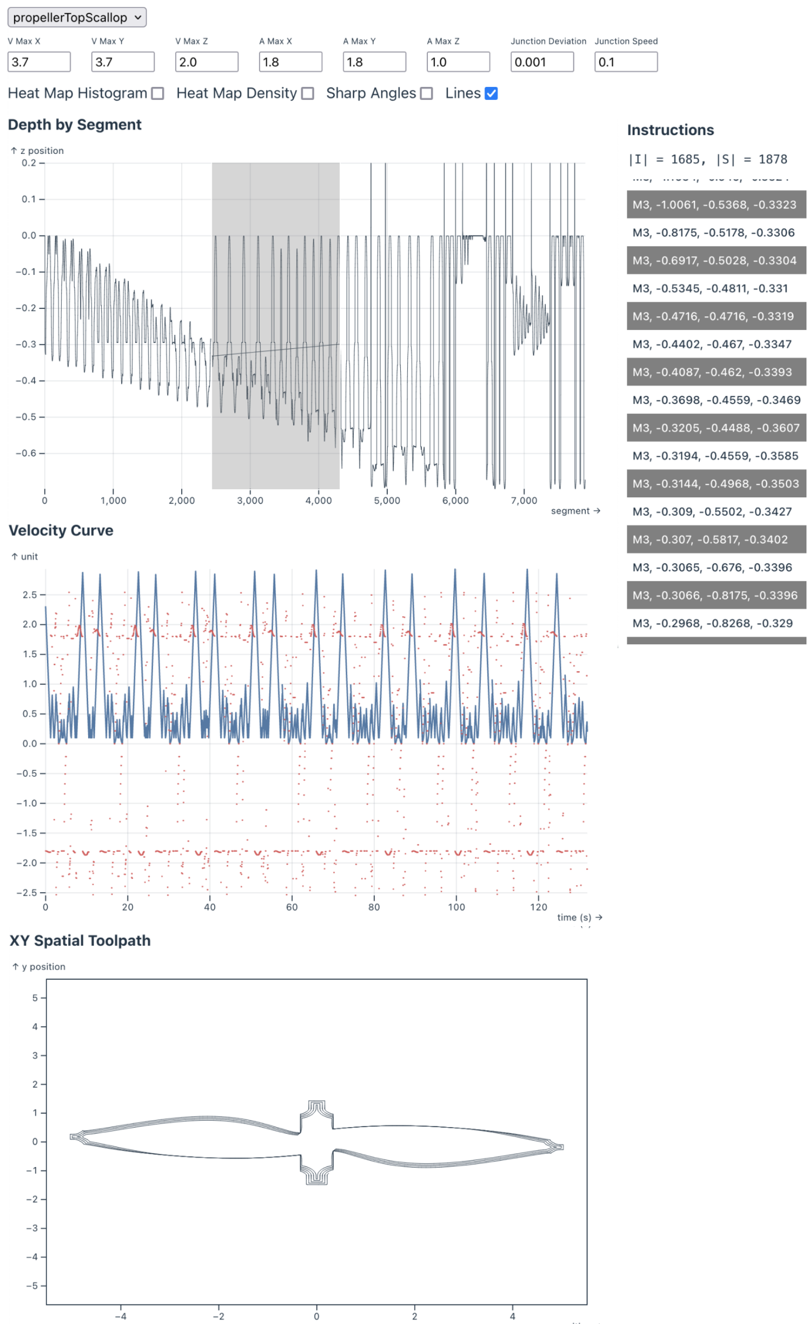


Figure 4.11: Dashboard for Interactively Exploring TSS. The fabricator chooses different toolpaths (drop-down menu), kinematic limits (number fields), and TSS (checkboxes). The top shows the z-position of the toolpath over time; fabricators can *brush* this profile to select only a subset of the toolpath to visualize. The subset's trajectory and XY plots are shown thereunder, and the constituent machine instructions for the subset are displayed on the right.

4.3 Dashboards and Interaction

So far, we have built a visual compiler pipeline using modern data visualization infrastructure. While we have likened a TSS to compilation, there is one advantage distinct from traditional compilation that the grammar of graphics-based approach provides: interaction. This is because, for experimental fabrication, static visualizations are not enough. Fabricators need to: drill down and explore fine-grain parts of a toolpath, zoom back out to ascertain high-level trends, and experiment with different visualizations.

To this end, we prototyped a dashboard with various interactions as shown in Figure 4.11. In this dashboard, a fabricator *brushes* the top plot, which displays the z-position of the tool (Y-axis) with respect to the segment in the sequence (X-axis). Brushing selects a subset of this toolpath, causing the trajectory plot and XY plot with TSS to re-compute and re-render to show data for only the subset. The dashboard also displays the corresponding machine instructions for the selection to the right, in case the fabricator needs to debug particular problematic instructions.

In addition, the fabricator can adjust the kinematic limits at the top of the dashboard; such fiddling of parameters is a common task for fabricators who currently must do so with only intuition as a guide. Changing kinematic limits also re-computes the trajectory and and re-renders associated visualizations, showing often-significant differences as demonstrated in Figure 4.12.

While we were building the dashboard from custom code, the developers of Observable Plot released Observable Framework [Obs24], which automatically generates dashboards from source code. In particular, fabricators can implement the first two parts of the approach (trajectory generation and IR generation) in any language while keeping the plotting functionality in Javascript or Typescript. We aim to migrate to Observable Framework moving forward.

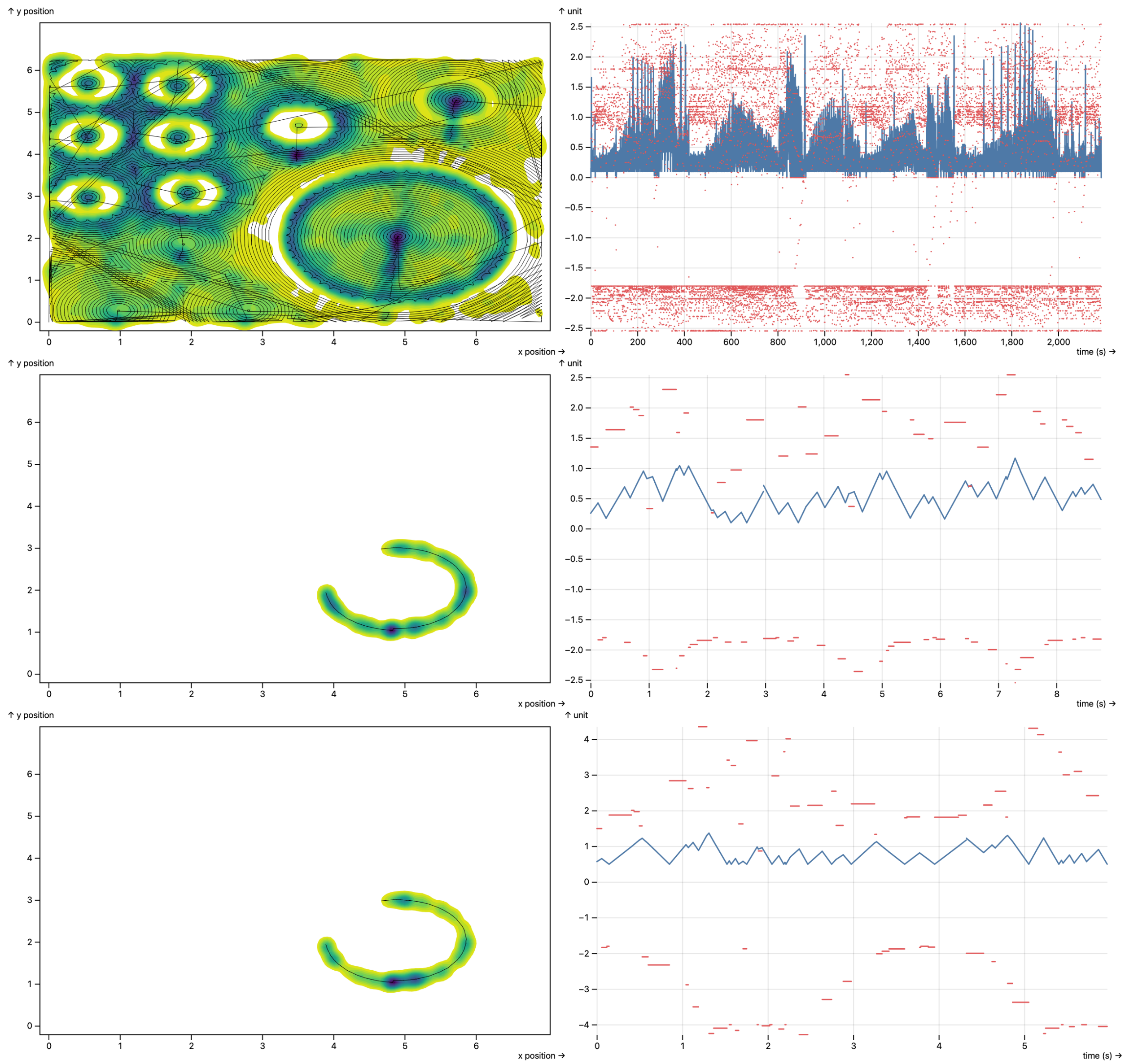


Figure 4.12: Experimenting with Kinematic Limits. Top-left) linear and density-based heatmap TSS for a CNC milling toolpath for multiple gears. The maximum XY velocities are set to 3.7 in / s and maximum accelerations set 1.4 in / s. Top-right) the respective trajectory. Middle) the same TSS of a slice of the toolpath. Bottom) the same with modified, anisotropic limits: a maximum X velocity of 10 in / s, a maximum Y velocity of 6 in / s, and accelerations half thereof on respective axes. The changed limits result in a lower milling time, a lower-frequency trajectory, and a slightly different heat distribution.

4.4 Verso: an Early Programming Environment for TSS

We wanted to place TSS in an interactive programming environment. Before we adapted computational notebooks as a foundation for this environment (described in Chapter 5 and Chapter 6), we created our own prototypical environment called Verso. We describe Verso's approach for historical reasons and to motivate some of the later functionality that we further explored with bona fide computational notebooks.

We implemented Verso as a prototype computational notebook environment using Typescript and React.js, with a backend written using Node.js. The application consists primarily of an in-browser editor that lets fabricators write workflow code in Javascript, a language accessible to many fabricators. As the fabricator types, Verso assesses input events and continuously re-evaluates the fabricator's code. It provides a library of classes for representing common concepts in digital fabrication, such as `Machine` and `Geometry`.

In addition to raw code, fabricators can use modules by writing a module function. A *module function* is a function that begins with the `$` symbol, such as `$geometryGallery`. As soon as the fabricator types in a module function, Verso generates the module's GUI immediately below the program line containing that function. The fabricator can then use the module's GUI to make graphical changes, which can affect the return value of the module function. As described further in Section 4.4.1, modules are *I/O monads* that handle any input-output in a separate evaluation context from the main workflow program. As a result, modules can communicate with the backend to perform tasks such as selecting a geometry file, sending data over a serial connection to a machine, or guiding the user through calibrating a projection. Deleting the module function causes the module's associated GUI to disappear.

Using raw code and modules, the fabricator eventually generates machine *instructions*, the low-level commands that control a machine. To develop an understanding of what the machine would do if it ran a set of instructions, including any potential issues that could occur, fabricators use TSS to translate a set of instructions into a visualization. Each TSS is designed for a specific fabrication scenario, and each provides a different "view" of a set of instructions. Fabricators use the TOOLPATHVISUALIZER module to select the desired TSS for the set of instructions currently passed into the module function. Together, changes to the workflow code and to modules result in near-immediate visual feedback via TSS.

For our current implementation, we chose to prototype Verso's features as a standalone computational notebook environment. This allowed us to more freely build and test ideas without concern for how well they would integrate with an existing notebook system. In Chapter 5, we will implement Verso's features in popular computational notebook systems such as Observable [Bos18].

4.4.1 Modules: Embedded GUIs that Encapsulate Physical Control

Though representing digital fabrication workflows as live programs offers advantages, it introduces additional issues. First, many tasks in digital fabrication are graphical by nature. For example, selecting and processing geometries, tuning machine parameters, and previewing toolpaths require graphical interfaces for smooth experimentation. Second, fabricators must constantly communicate with external digital and physical processes, e.g., writing instructions to the physical machine over a serial connection.

In many programming languages, to perform tasks like I/O, programmers write *impure functions*, which are functions that cause side effects. Such a side effect might include logging data to the console, or, in Verso's case, causing a machine to move. Verso's live environment means that functions might be called many times per second. In this case, impure functions would cause many unnecessary and potentially unsafe side effects like repeated machine movements. Furthermore, as noted previously, our design aims to maintain a clean boundary between the flow of data in the program and code dedicated to I/O.

To solve both issues, Verso instead represents an I/O step of the workflow as a *pure function* that defers the *effectful code*, which may cause side effects separate from the main workflow's evaluation context. Module functions are pure functions that generate an associated GUI. While module functions may be called repeatedly by the live editor, effectful code is called within the module's own evaluation context, which is separate from the main workflow.

Modules afford graphical control in a programmatic context while also providing a layer of encapsulation around effectful code. Their functions optionally accept arguments and return values. Each module's associated GUI is rendered inline with code. When fabricators interact with the module's GUI (e.g., scrubbing a slider), the output of the module function's output changes accordingly. For example, the dispatcher module lets fabricators stage toolpaths to send to the physical machine; these are only sent after the fabricator clicks the "dispatch" button in the module's GUI. To do this, the `$dispatcher` module function accepts an array of toolpaths and returns an empty value. At the same time, each toolpath passed to the module function as an argument appears in the module's GUI. Once the fabricator clicks the "dispatch" button, only then does effectful code execute, and it executes within the module's evaluation context and not within the live workflow program. Modules reflect *monadic I/O* patterns from functional programming [PJJ98, Wad92] and build off of *livelits* by Omar et al [OMB+21].

4.4.2 Example Modules

Verso currently supports several modules that bridge code and the physical world. We briefly elaborate on three modules for accomplishing tasks from the usage scenario's plotter example.

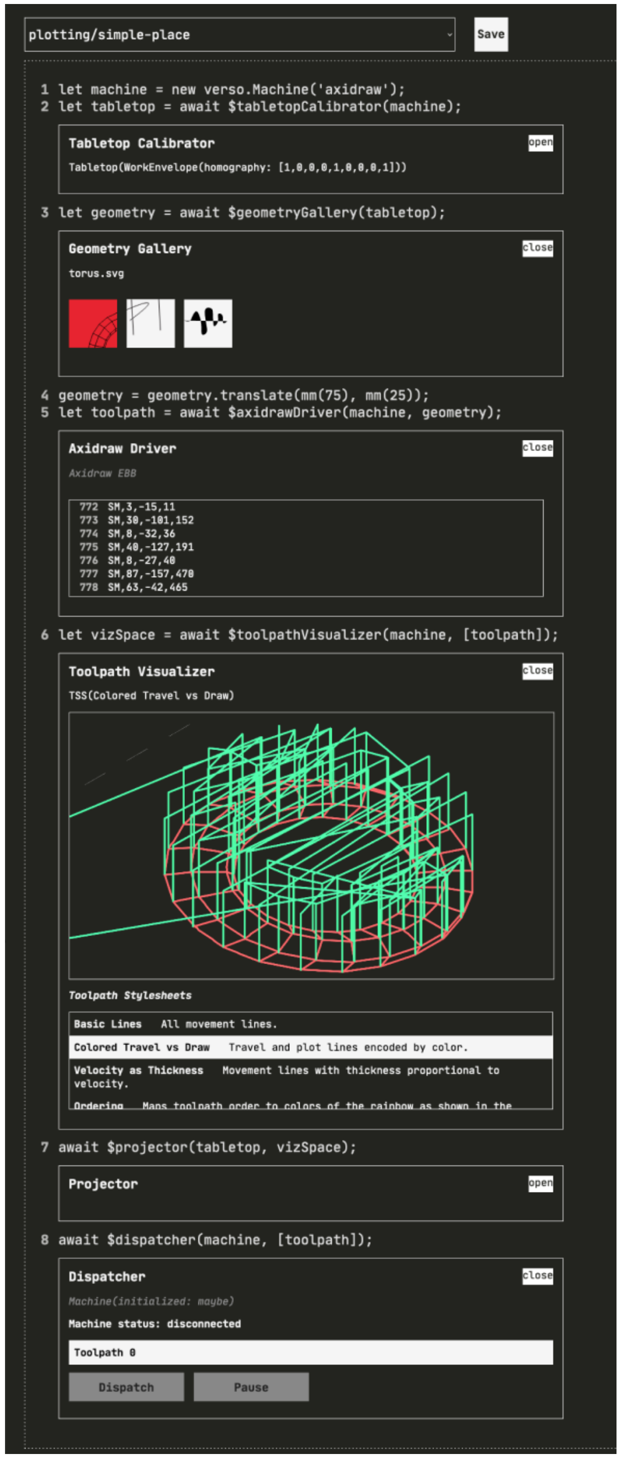


Figure 4.13: Plotting a Torus in Verso. When writing code in the editor, fabricators can add calls to module functions that generate associated embedded GUIs.

A module that calibrates a projected visualization for a machine's physical space. A Verso `Tabletop` object represents a virtual machine work space along with a *homography* that maps the virtual space to its true physical location. To calculate the homography, fabricators must map four virtual points to four physical points and then solve for a $3x3$ matrix representing the transformation. To do this, tabletopCalibrator projects a box representing a 2D projection of the machine's *work envelope*, that is, the total space in which the machine's tool can move. The module then prompts fabricators to use a mouse to drag the box's corners to match the physical ground truth work envelope, e.g., the boundaries of the machine's bed or a maximal box drawn by machines without a bed. Once this routine is complete, the module calculates the homography. The associated `$tabletopCalibrator` function in the workflow returns `Tabletop` objects with this homography such that any toolpaths and visualizations generated using the `Tabletop` will use correct physical coordinates.

A module that leverages existing machine toolpathing software. Verso can interface with current software tools because modules provide a thin abstraction around external processes. For example, the axidrawDriver module's associated function takes as input a vector geometry and outputs a toolpath of EBB commands that the Axidraw plotter understands. axidrawDriver communicates with the open-source Axidraw driver [Win22] to compute the toolpath. Our current implementation implements a backend function that forks a process that runs the Axidraw driver. When running this process on the the backend server, the module submits an HTTP request to pass the geometry data to the server, which in turn passes it to the process and returns the resulting toolpath as an HTTP response. In this case, we reverse engineered a placeholder to intercept instructions that the driver output. In general, the effort needed to implement a wrapper module depends on the availability of APIs from existing software tools.

A module that connects to and initializes a physical machine. Many machines require initialization procedures before each use, including *homing* the machine's axes to establish absolute bounds and *zeroing* the machine by having the end-user fabricator set a physical point to represent the origin. Verso's MACHINEINITIALIZER module handles these tasks; it is parameterized over the type of machine passed in as input to the `$machineInitializer` function. For example, for the Jubilee CNC machine [VTSTOP20], the module exposes subroutines for connecting to the machine over a serial port and sends G-Code to the machine to home its axes in the required order. Fabricators execute these subroutines using graphical input and can debug and edit module subroutines as needed. The `$machineInitializer` function returns a `Machine` object marked as {`initialized: true`} if and only if the module has received input from the physical machine that it was homed and zeroed correctly.

4.4.3 Implementing Custom Modules

Fabricators can add new modules or extend existing ones as additional applications arise. To create a module, they must provide a new class that extends the `Module` superclass in Verso, maintains its own `state`, and defines the following:

- **`expand()`**: a method that synthesizes the module function to be run in the workflow code. For example, the `expand` method for TABLETOPCALIBRATOR returns a string representation of a function named `$tabletopCalibrator` that will be inserted into the workflow immediately before runtime. The function takes a `Tabletop` object as a parameter and returns an adjusted `Tabletop` object as a result of the manual calibration, as shown in Figure 4.14.

- **`render()`**: a method that returns HTML for the module's GUI, including any text, buttons, and visualizations shown in the GUI. This method shadows `React.Component`'s render method and is called whenever the module's state is updated.

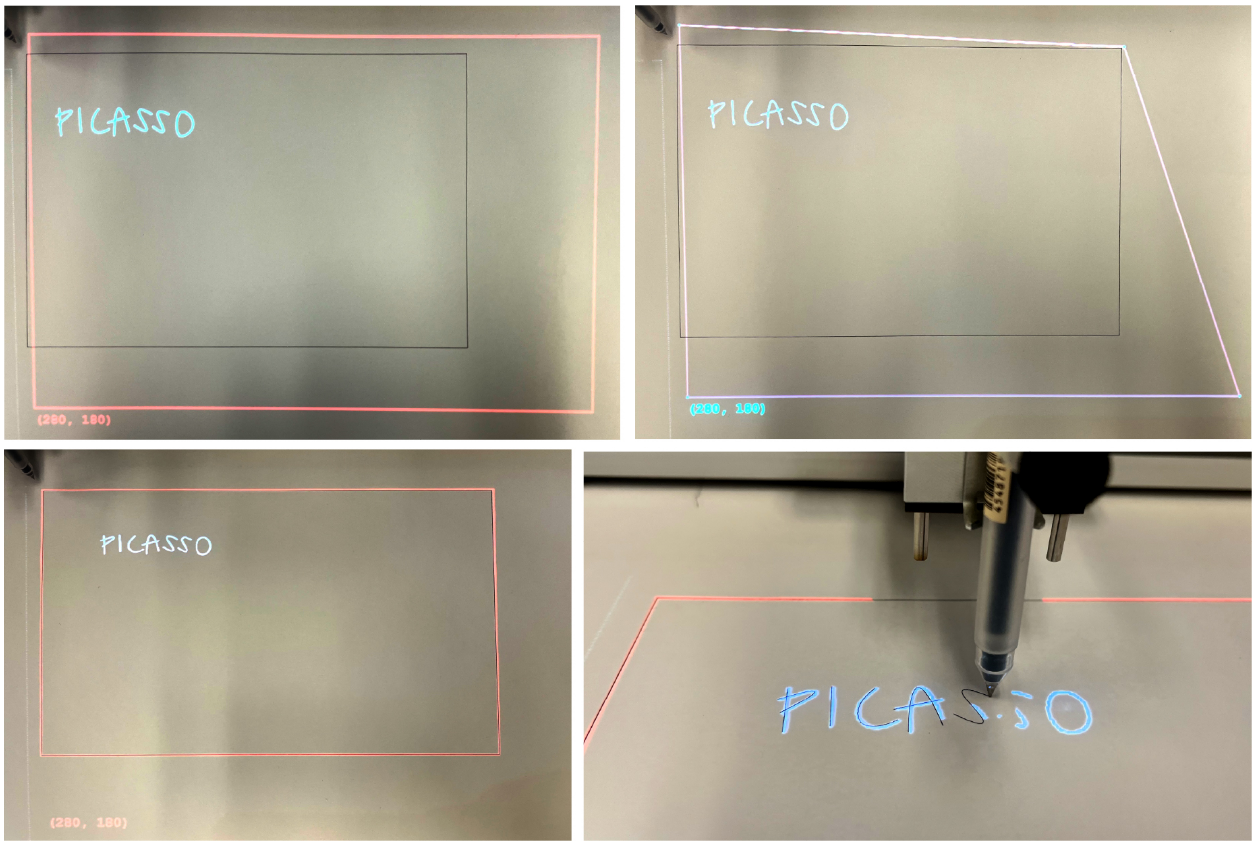


Figure 4.14: Calibrating Visualizations for a Physical Machine. 2D projections of a visualization do not initially match the machine's coordinate space. Using the TabletopCalibrator module (not pictured), the plotter draws a ground truth bounding box (upper left). The fabricator uses the mouse to interactively draw the projected bounding box to match the physical one (upper right). Once the bounding boxes match (lower left), TabletopCalibrator computes a homography to correctly map the projection to the machine coordinate space so visualizations precisely match where the plotter moves (lower right).

- **`handleInput()`** (multiple): methods that handle fabricator input (e.g., slider scrub, button press). These methods can modify the module's state, which causes the module to re-render. They can also call action methods to perform I/O.

- **`action()`** (multiple): methods for performing I/O. For example, DISPATCHER accesses the serial port associated with the machine and writes data over it. These methods can interface with a server over HTTP, as well. They also perform any needed calculations, such as computing the homography to localize visualizations to visual space.

Chapter 5

Experimental Workflows as Computational Notebook Programs

"[The] learner always gets the experience of interactively controlling the lower-level details, understanding them, developing trust in them, before handing off that control to an abstraction and moving to a higher level of control."

— Bret Victor, *Learnable Programming* [Vic12]

In the last few decades, digital fabrication tools that previously existed only in professional machine shops have become increasingly available to a wider and more varied group of makers [And14, Ger07]. As computer systems moved out of the work place to become increasingly prevalent in domestic, educational, ludic, and aesthetic contexts [Bø06], so did digital fabrication systems reach new sites and groups, creating new priorities and challenges for digital fabrication research. In particular, subtractive manufacturing machines such as CNC (computer numerically controlled) mills—as opposed to additive manufacturing machines like 3D printers—present additional barriers to makers without experience in traditional machine shops, such as safety and tool knowledge.

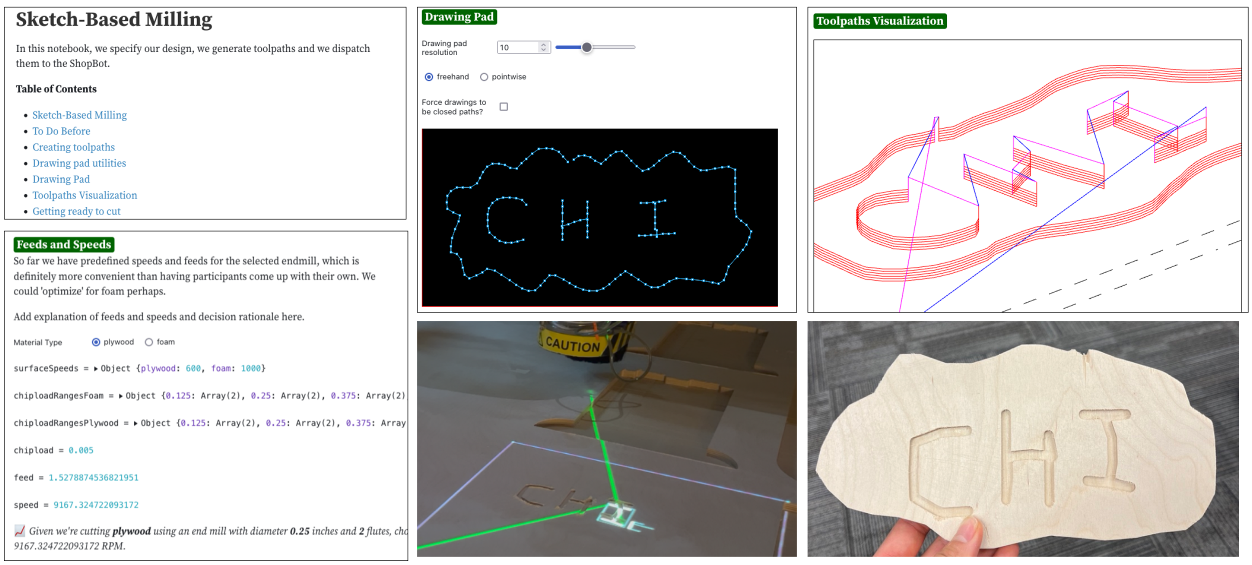


Figure 5.1: Snippets from the **D1: QuickDraw** Notebook. Left: the notebook is divided up into separate parts (top), for example, a step for calculating feed and speed values based on the user's tooling and material setup (bottom). Middle: the user can sketch a geometry freehand, by defining control points, or by importing geometry (top) and then visualize the toolpath in-situ using the AR overlay. Right: users can experiment with the cutting depth of the toolpath (top) and dispatch the job to obtain a quickly milled prototype (bottom).

Yet, the existence of digital fabrication systems outside of exclusively industrial contexts indicate a new opportunity: to rethink digital fabrication systems in the light of new priorities, namely expressivity and customization rather than replication and mass production. Artists, designers, entrepreneurs, hobbyists, and other makers prioritize the exploration of new physical forms, the development of original workflows, and the production of bespoke or customized artifacts, rather than the high volume reproduction of a single geometry [JZ15, TSTOP21, Zor16].

We consider the example of a fictional user, Talia. Talia is an artist and product designer who makes ceramic home decorations and objects: drinkware, candle holders, plant pots and accessories. The designs are unique and rely on custom molds that Talia makes herself with a CNC mill. She generates the mold designs in CAD software and then creates toolpaths using a CAM tool, which she then sends to her studio CNC mill. She uses the milled positives to create plaster molds which she uses to create the ceramic pieces.

While her small local business is thriving, Talia is often frustrated with the process of developing new products and designs because of the labor- and resource-intensive nature of experimenting with new designs. For example, one of her best-selling pieces is a mug with a gradient glaze and pronounced ridges generated algorithmically. Talia has spent much time sketching and creating other patterns in CAD and testing them on a mill, but every time she wants to adjust her toolpath or tweak an aspect of the design, she either has to regenerate an entire G-code file in CAM or go back to the model in CAD and go through the CAD-CAM-CNC pipeline again. It is also difficult for Talia to document and share her process with other fabricators at her business because they frequently invent new techniques or improvement to algorithms which require documentation to be constantly rewritten.

In this example, we highlight that the software tools used for CNC milling are locked into a rigid interaction paradigm that focuses on faithful replication, rather than exploration. Even though CNC mills have now become accessible in non-professional spaces (such as fab labs, makerspaces, professional studios and homes), the software tools to control them are still built after the model of this replication-based interaction.

We argue that conventional and hobbyist CAM software tends to discourage makers from "straying from the path" or exploring new designs, as in Talia's example, because they based in a paradigm of *fabrication as executing programs*. In these tools, machining operations are packaged into programs which are relatively easy for makers to execute, but sacrifice the low-level machine control needed to pioneer novel techniques with machines and materials. Low-level machine control allows makers to quickly dispatch commands to the machine, verify operations in real time, and adjust their designs accordingly. As such, it supports improvisational and exploratory fabrication processes. However, writing machine code for direct control of mills and lathes is largely restricted to specific scenarios in professional machine shops [DKM+21, Smi07]. It is difficult to abstract to more complex interactions; its low-level nature also obscures readability and portability for those besides the author. Moreover, programs written in machine code (e.g., G-code) afford very little documentation and explanation which becomes crucial for less experienced makers.

5.1 Enabling Experimentation Beyond the Digital

Altogether, it is difficult to interact with CNC milling machines in an exploratory way. We define exploratory fabrication as focusing on discovering new material behaviors and developing novel workflows rather than the faithful translation of digital models. To achieve exploratory fabrication with CNC mills, makers require control software that allows for low-level machine control that can be quickly iterated on, is richly documented, preserves safety checks, and discourages risky operations.

How can we better support exploratory fabrication with CNC milling machines? To address this question, we argue for a vision that treats *fabrication as writing programs*—not just executing them. The knowledge required for writing programs for fabrication needs to be properly scaffolded. To extend direct machine programming in a supported way, we turn towards *literate programming* where programs are human-readable documents that interleave prose, graphics, and source code [Knu92].

Computational notebooks, a common example of literate programming, aid with digital-only programming problems in data science, machine learning, and related domains [KRKP+16, Per21]. We argue that a literate programming paradigm, if modified properly, could catalyze exploratory fabrication for CNC milling by allowing makers to quickly send commands to the machine, adjust and fine tune their design iteratively, and acquire the necessary programming and manufacturing knowledge to develop their own workflows. Further, literate programming could democratize exploratory fabrication by letting users replicate existing workflows simply by reading and deploying code from a computational notebook.

To test this hypothesis, we present Imprimer, a machine infrastructure for structured, direct control of a CNC mill from a computational notebook—in our case, the Shopbot CNC router [Sho22] and the Observable computational notebook [Obs23b]. In Imprimer, makers can prototype new fabrication workflows by writing and modifying code. Our library provides custom visualizations both in the notebook and projected onto the machine in-situ. Makers document their making process by interleaving text and visual documentation. Imprimer blends traditionally separate parts of the fabrication process into a single, live environment where makers can make changes to code, visualize the results, and deploy cutting jobs immediately to the machine. Makers can also view or hide the underlying code for each cell; by hiding all code, makers use the notebook as they would with any other graphical user interface or tutorial document.

We stress that Imprimer is the beginning of a new paradigm: sharing ideas about ways to fabricate, not just running code to fabricate. Thus, it does not yet cover all the functionalities currently available in conventional CNC software tools. Rather, it is a step towards a different interaction paradigm which lets makers discover machine behavior and possibilities gradually, and over time implement their own functionalities by writing code.

To guide the development of this paradigm, we developed three demonstration notebooks that represent exploratory workflows: **D1: QuickDraw**, AR-assisted sketch-based milling (Section 5.4.1), **D2: FunctionTile**, surface milling by sampling mathematical functions of two variables (Section 5.4.2), and **D3: MiniShelf**, parametric generation of bookshelves (Section 5.4.3), alongside two tutorial notebooks that introduce Imprimer's connection and material setup functionalities. We conducted an in-shop user study with participants holding a range of backgrounds in CNC milling and computational notebook programming (Section 5.5). From these demonstrations and evaluations, we discuss the challenges, rewards, and future possibilities of literate programming as an interaction model for CNC milling.

5.2 End-to-End Example

To provide a concrete example of using Imprimer, we consider an improvisational milling workflow with Talia, who wants to explore different patterns for ceramic molds. She intends to mill shallow surfaces with various patterns and cast these test pieces in plaster to assess which would work best for cups. This process, which would traditionally take place over several CAD and CAM tools, all happens within a single notebook, enabling rich exploration and quick iteration.

Step 1: Connection. To begin, Talia connects all the different parts of the system—the Observable notebook, the augmented reality overlay and the ShopBot CNC milling machine—over the provided WebSocket. She sends a test command to the ShopBot, "`MX,5`", to make sure the notebook effectively communicates with the milling machine. The ShopBot moves accordingly and the connection is successful.

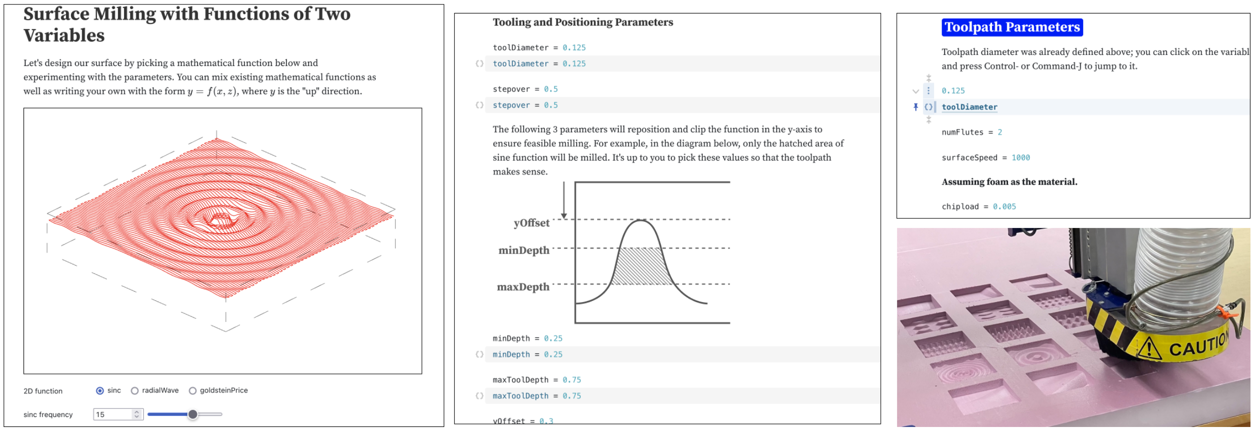


Figure 5.2: Snippets from the **D2: FunctionTile** Notebook. Left: users choose a mathematical function of two variables and use graphical elements to experiment with function parameters and see the toolpath visualization update in real time. Middle: to ensure feasible milling, users define variables to offset and clip the function within a desirable bounds as described in a hand-drawn diagram. Right: users specify tooling parameters to calculate feeds and speeds (top) before dispatching the job to the CNC mill (bottom).

Step 2: Material setup. Once Talia has established a connection, she goes through a series of steps to prepare the machine and the material for milling: she measures her material (in this case, insulating foam, a cheap and forgiving material for pattern exploration) and fixes it to the machine bed using screws and clamps. Given the thickness of the piece of foam she uses (2 inches), she chooses a long square 1/8th inch end mill and installs it. Finally, she zeroes her axes and moves back to the Observable notebook to start designing her toolpaths.

Step 3: Overlay calibration. Next, using the AR overlay notebook (Section 5.4.1), she runs through the overlay's calibration procedure to match the projected toolpath to where the machine will actually move. Her material already has some test cuts made in it, but using the AR overlay she is able to position the toolpaths into the spaces on the stock with material remaining.

Steps 4 and 5: Designing toolpaths and milling. After this, Talia uses code from the **D2: FunctionTile** notebook which lets her create 3D surface milling patterns using mathematical functions of two variables. She decides to explore three in particular: sinc, sine/cosine, and the

Goldstein-Price functions. With the notebook, she designs her patterns *by designing her toolpaths* and iteratively explores the effect of various machining parameters on her design, such as the maximum cutting depth, the stepover, the frequency of the patterns, the feeds and speeds, among others. With each new swatch, she tweaks a few parameters by making simple edits to the code in the respective cells that calculate these parameters. She documents each variation both in prose and in a dictionary of parameter values in a notebook code cell. Within two hours, she has created over 20 swatches with three different functions that she is ready to cast to create her test pieces. Once she is done, she adds notes and comments in the notebook for future reference, and to eventually share the notebook with her community.

5.2.1 Target Audience: Novices and Experts Alike

Generalizing from Talia's case, Imprimer is a paradigm shift towards using code, visualization, and documentation to pioneer and share knowledge about exploratory fabrication workflows. Users of varying skills in both programming and in CNC milling can engage with Imprimer. Because Observable notebooks can operate with the code completely hidden, those without programming experience can still use the notebooks with full functionality apart from code-level customization. In parallel, while there are increasingly sophisticated tools for learning programming, there are relatively few innovations in digital fabrication education. Programmers who lack CNC milling experience can follow Imprimer notebooks that explain the process where documentation is interleaved with the controls; this contrasts with conventional CAD/CAM tools where documentation is separate.

On the other hand, users who are experienced in both programming and CNC milling can remix notebook code and develop entirely new functionality. Unlike conventional CAD/CAM tools, the source code that drives an interaction can be readily tailored, from simple customization of toolpath algorithms, to custom forks of notebooks, to entirely new notebooks representing new workflows [Mø95]. Imprimer's library and direct machine communication helps experienced developers focus on and test their high level goals. Finally, users of all backgrounds can benefit from Imprimer's free and open source nature, whereas common CAM software can cost up to thousands of dollars per year.

5.3 Integrating Machines and Computational Notebooks

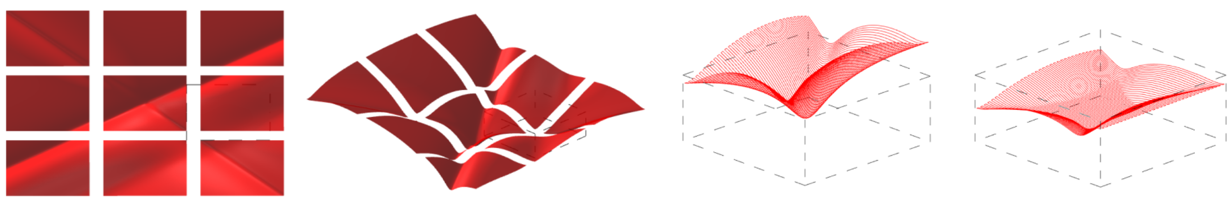
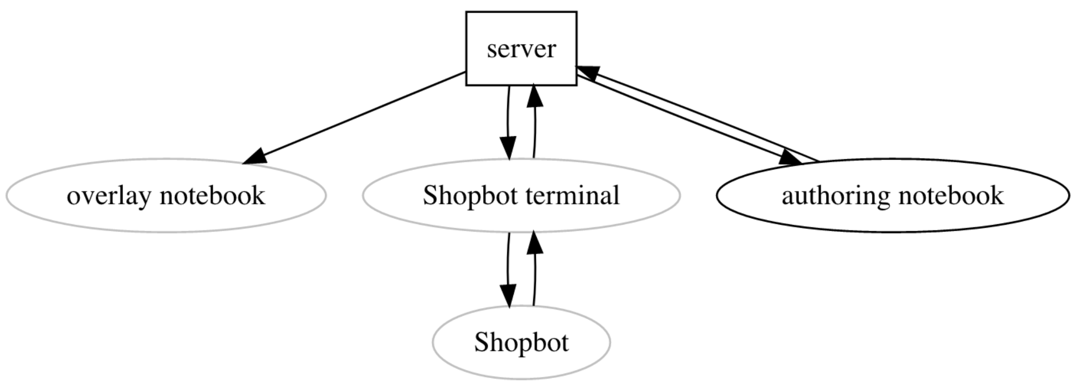


Figure 5.3: Milling Subsets of the Goldstein-Price Function in the **D2: FunctionTile** notebook. 1) Top-down view of a surface rendering of nine subsets surrounding the global minimum of the Goldstein-Price function. 2) An isomorphic view of the same rendering. 3) Toolpath rendering of the single subset containing the global minimum (center of the nine), currently infeasible to mill due to the toolpath lying almost entirely above the material. 4) After interactively processing the toolpath by dividing the function output, adding a vertical offset, and clamping the function to a desired bounds, the toolpath becomes feasible to mill into a mold.

To provide the foundation for Imprimer, we built a network infrastructure to afford direct control of a CNC mill from a computational notebook environment. To support the creation of workflows, we contribute a library of functions for a live computational notebook that facilitate: connecting to and controlling the mill, navigating tooling options, visualizing geometries and toolpaths, and more.

5.3.1 Machine Network Infrastructure

To support this workflow, Imprimer comprises a web server that coordinates communication between desktop application that relays information to a full size Shopbot PRSalpha with a 96x60" cutting bed[1] and a projector-based augmented reality interface called the *AR overlay*. The Shopbot accepts both its native SBP instruction set alongside the more commonplace G-code. Figure 5.4 illustrates the relationship between the user's current notebook ("authoring notebook"), the server, and a desktop computer that communicates over a wired connection to the Shopbot ("Shopbot terminal"). As is done in our tutorial notebook on connecting to the machine, we can generate Figure 5.4 representing the network architecture using the code that follows the figure.




```
dot`digraph "networkGraph" {
    server [color=${connectedNodes.server
        ? "black" : "gray"}, shape=box]
    // More styling info omitted

    "authoring notebook" -> server
    server -> "overlay notebook"
    "Shopbot terminal" -> server
    // ...
}
```

Figure 5.4: Visualizing the status of the machine-notebook network. `connectedNodes` is a mutable cell that is updated with packets from the server whenever a client connects or disconnects, recoloring the diagram appropriately

[1]The Shopbot's default unit of length is inches rather than meters.

To connect the notebooks, we implemented the network using a WebSocket interface building on the code used by Li et al. [LJCH17], which is publicly available. In particular, we built the Shopbot terminal and server to facilitate direct interaction between multiple notebooks and with the machine. We installed the terminal on the computer connected to the Shopbot and deployed the server code in the cloud. This direct notebook-to-machine job dispatch is not a typical functionality for the Shopbot, which requires exporting and importing files, even though similar functionality exists off-the-shelf for hobbyist 3D printers [Emi21].

5.3.2 Implementation in Observable Notebooks

To control Imprimer's machine infrastructure, we implemented a library for Observable, a browser-based computational notebook popular for data science applications. Observable runs a modified version of Javascript, letting it interoperate with many existing technologies. The notebook differs from other computational notebooks like Jupyter Notebook (Python) or RMarkdown (R) in that code cells run in *topological order*, meaning that cells are automatically recomputed whenever any cells they depend on (*parent cells*) are themselves recomputed. This produces a live programming [Tan13] environment where changes in one cell propagate throughout the notebook; graphical input, visualizations, and underlying data are always up-to-date, affording quick and interactive iteration. This contrasts with other notebooks whose code is run in a linear order or must be manually re-run; rather, live topological allows us to arrange code in the best order for fabrication workflows and teaching—a key tenet of literate programming [Knu92].

Live Cells and Cell Names

All parts of an Observable notebook are contained in cells which written Markdown, HTML, or Javascript. For example, we represent the following cell that computes a value and associates it

```
stepdownPercent = 0.5
```

```
stepdownPercent = 0.5
```

Figure 5.5: Declaring a cell variable.

with the cell name `stepdownPercent`, representing the proportion of the end mill's diameter that the mill advances downward per cutting pass.

Whenever the the cell `stepdownPercent` is recomputed, for example, if the maker enters a new percentage, the notebook recomputes following cell which calculates the number of cutting passes required to cut through through the material.

```
numPasses = 6
```

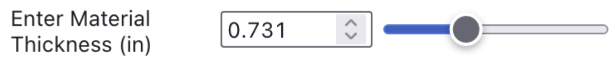
```
numPasses = {
    let stepdown = stepdownPercent * installedEndMill.diameter;
    return Math.ceil(thickness / stepdown);
}
```

Figure 5.6: A child cell reactively computes results when a parent is recomputed. `installedEndMill` and `thickness` are themselves cell values defined elsewhere in the notebook.

Cells can also be Markdown or HTML, so notebooks contain diagrams, videos, and prose that instruct the maker about various considerations while programming for CNC machines, as shown in Figure 5.1, Figure 5.2, and Figure 5.9.

HTML Templating and Views

Cells can also return HTML that is generated by Javascript, affording custom visualizations called *views*. A view has two parts: the view itself, which is typically an interactive DOM element; and the value, which is any JavaScript value. For example, consider the following view.




```
viewof thickness = Inputs.range([0.0625, 2], {
  label: "Enter Material Thickness (in)",
  step: 0.0001
})
```

Figure 5.7: A slider view for setting material thickness.

The result of the right-hand side of the cell evaluates to DOM element which is rendered above the cell. But, by using the `viewof` keyword, the value of `thickness` is instead the *current value* of the DOM element—the value currently selected in the slider: 0.731. Views can be arbitrarily complex because their appearance and interactivity can be programmed in HTML and/or Javascript. Imprimer builds extensively on this functionality to provide intuitive interfaces for different stages of CNC control.

Bringing Machine State into the Notebooks

Imprimer's library provides a streamlined interface to synchronize notebook state with physical machine state. We configure the Shopbot terminal to periodically emit packets containing the Shopbot's current state to the authoring notebook through the server. For example, whenever the end mill changes position, the Shopbot terminal detects this change and sends the new position in a packet to the server. On the notebook end, we keep track of the machine state for library functions to query.

5.4 Demonstrations

To explore the versatility of Imprimer, we implemented three demonstration notebooks: sketch-based milling with augmented reality (**D1: QuickDraw**), surface milling molds by sampling functions of two variables (**D2: FunctionTile**), and a parametric shelf generator with associated debugging tools (**D3: MiniShelf**).

5.4.1 QuickDraw: Sketch-Based Milling with Augmented Reality

Following the thread of work on direct drawing with CNC machines introduced by Li et al. [LJCH17], we built QuickDraw, a notebook that lets users quickly sketch geometries to engrave or cut through, visualize the resulting toolpaths in-situ on the physical material using an augmented reality overlay, make adjustments, and directly mill the job from the notebook. The goal of QuickDraw is for users to quickly prototype low-fidelity versions of more complicated form they might want to test later, or test out the effects of different end mills on material finish. Assuming the machine's tooling and material are set up correctly, and that the AR overlay has been calibrated, users can walk up to the notebook with no prior design file, sketch a quick concept as in Figure 5.1, and mill it out completely in around ten minutes. Once satisfied, users can also import and process existing 2D geometries, create 3D toolpaths while experimenting with scale and cutting depth, and check the toolpaths using the AR overlay. They can then move into other notebooks to work at a higher fidelity. We defer importing 3D geometries for future work.

Augmented Reality Previews

We sought to extend frequent visualization of data beyond the notebook environment into physical space. To this end, we installed a projector above the bed of the Shopbot that projects toolpath visualizations directly onto the machine's bed. This provides an AR overlay which can project 2D views onto the material in-situ; users must flatten 3D views into 2D before rendering the with the overlay by calculating a 2D projection or "slice" onto a desired cutting plane. The visualizations are rendered by a separate notebook which is connected to the network and listens for updates from the authoring notebook, for example, any changes in the toolpath's instructions or in machine parameters that affect the toolpath. We include a calibration routine that the maker performs when setting up their stock material to ensure that the visualized path matches the end mill's physical location as proposed by Tran O'Leary et al. [TOJP22]. This routine involves the maker dragging four projected points to match the ground truth corners of their stock material; the notebook then computes a homography from the mapping of points and uses the homography to transform all elements of the visualization to match the toolpath's true physical location.

5.4.2 FunctionTile: Surface Milling Tile Molds by Sampling Functions of Two Variables

One of Imprimer's strengths is that notebook authors can combine in a single, documented programming environment parts of a fabrication pipeline that would otherwise be split across applications. In this demonstration, we created a means of surface milling molds for plaster tiles based on forms derived from mathematical functions. **D2: FunctionTile** includes a visualization for both the surface of a function of two variables alongside a derived toolpath that can be dispatched to a CNC mill (Figure 5.2). For example, we implemented the two-variable unnormalized sinc function

$$z = \frac{\sin(f \cdot \sqrt{x^2 + y^2})}{\sqrt{x^2 + y^2}}$$

where f is the frequency or "waviness" of the function, shown in the cells in Figure 5.8, which results in the plot in Figure 5.2 and a slider which recomputes the plot in real time.

sinc = ƒ(x, y)

```
function sinc(x, y) {
  let r = Math.sqrt(x ** 2 + y ** 2);
  let z = Math.sin(sincFreq * r) / r;
  return THREE.Vector3(x, y, z);
}
```

sinc frequency 12

```
viewof sincFreq = Inputs.range([0, 24])
```

Figure 5.8: Defining the sinc function with a slider for experimenting with frequency.

We provided three example functions of two variables: the sinc function, a sine/cosine function, and the Goldstein-Price function [MS05]. Users can modify these functions or write their own functions in code as needed.

We then sample points across the surface of the plot and connect them to generate a toolpath. Several decisions greatly affect how this toolpath is generated, for example: whether the path passes row-by-row or column-by-column (e.g. for anisotropic materials like wood with a grain), the stepover which controls how much of the end mill's diameter the mill will remove per pass, and the z-offset at which the surface is milled relative to the top of the material. These toolpathing decisions, alongside the mathematical parameters of the chosen function, are all interconnected and affect the form and quality of the milled mold. For example, very thin parts of the surface could break depending on the material, stepdown, and chosen feed and speed.

With conventional CAD and CAM, it is possible to create similar molds, but a user would have to work across several programs, changing parameters in one program before exporting and importing to another. Even programmatic tools such as Grasshopper plus RhinoCAM [Har22] which support generating 2D surfaces still require back-and-forth between the Grasshopper window ("programming side"), the Rhino interface ("CAM side"), and the RhinoCAM options interface ("CAM side") where the user has to know in advance how changing one parameter affects the design. In contrast, Imprimer lets the notebook author quickly visualize the effect of each parameter change on the toolpath and document the flow of data. The notebook exposes both input elements for experimentation and the underlying source code, making it easier for users to follow along without invested expertise in experimental CAD/CAM or programming tools.

Translating Abstract Concepts into Millable Forms

Among the three example functions, the Goldstein-Price function proved especially challenging to translate into something that could be physically milled. This is largely because the function was designed to test optimization algorithms and has volatile behavior spanning several orders of magnitude around its global minimum. We were particularly inspired to explore the function as a creative inspiration when one participant in our user study (Section 5.5) wanted to have additional views that showed how the function could be translated into a millable mold. If we naively map part of the function to the space occupied by the tile mold, we risk "squashing" much of its the sloping topology. To address this, as Figure 5.3 shows, we sampled several subsets of the function around its global minimum. We then defined a graphical input so that a notebook user can pick one subset at a time and experiment with a division term and vertical offset to fit just a subset into a mold space while preserving as much detail as possible for the given subset. They then repeat the process with other subsets to produce a set of tiles that together show the topology of the function.

5.4.3 MiniShelf: a Parametric Shelf Generator with Associated Debugging Tools

MiniShelf is a lightweight parametric toolpath generator for fabricating shelves that includes tools for troubleshooting the fabrication process. MiniShelf accepts as parameters: the height,

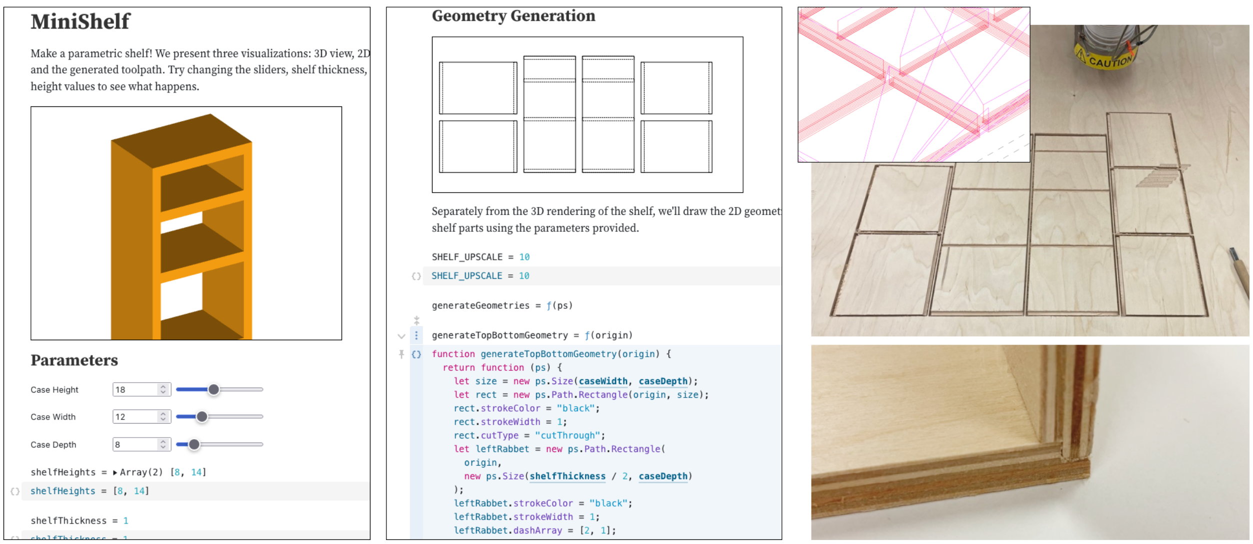


Figure 5.9: Snippets from the **D3: MiniShelf** notebook. Left: notebook users can experiment with different shelf dimensions and see a 3D rendering of the assembled shelf update immediately. Middle: when any shelf parameters are changed, the notebook generates new planar geometries of the shelf parts as defined in notebook cells which users can edit if desired. Right: A generated 3D toolpath (top) which can then be dispatched to a CNC mill (middle); rabbet joint parameters can be adjusted to achieve a tight fit (bottom).

width, and depth of the unit, the number and spacing of shelf spans, and the thickness of the stock material. For now, the selection of joints are fixed, though this could be easily extended in the future; we use assembly-friendly rabbet and groove joints to respectively connect the sides of the shelf case and to connect the shelf spans to the case. In the MiniShelf notebook, we implemented from scratch each part of the fabrication process—from machine setup, to material selection, to parameter selection, to toolpath generation, to visualization, to job dispatch.

Upon first implementing this workflow, fabricated shelves did not yet fit correctly, and we relied heavily on Imprimer's features to debug our own process. While multiple things can go wrong while exploring a new fabrication workflow, by expressing the workflow in code using Imprimer, we could debug our process by debugging code. As a result, we not only contributed a parametric pipeline, but also an associated set of features that we used to debug the process which we have incorporated into Imprimer's library which we describe below.

Debugging Incorrect Logic in Toolpath Algorithms.

To create rabbets and grooves in the shelf pieces, and to cut out all the pieces to size, we wrote all the necessary toolpath generation algorithms by hand. Writing bespoke toolpath algorithms allowed us and future users of the notebook to readily generate just the right toolpaths for fabricating the shelves, rather than having to adapt output from a conventional CAM interface. Naturally, while writing the algorithms, we implemented buggy functionality, for example, generating the rabbets in the wrong location. We were able to catch many of these issues using the top-down view (Figure 5.9, right). Though, some issues could not reasonably be identified until runtime; for example, by not "overshooting" the rabbet toolpath over the designated region, we were left with artifacts where the mill could not reach (Figure 5.10, left). We could have addressed these issues by implementing dog bone fillets [Nic17]; in our case, we chose to overshoot the rabbet toolpaths by editing the algorithm and left a note about this design decision in an adjacent cell.

Tolerancing with Joints

An inevitable issue with getting joints to fit together is fabricating both parts within an acceptable tolerance. While conventional machining practice uses rigorous techniques like geometric dimensioning and tolerancing to systematically address these issues [Ame19], these techniques are generally too heavyweight for most hobbyist applications. In our case, to debug issues of getting joints to fit together, we wrote a function that takes in machine and maker-set parameters in the notebook and generates small testing pieces (Figure 5.10, middle). Once we found the optimal milling depth for the joints, we set it as a cell variable in the MiniShelf notebook.

5.5 User Study

To better understand how users conceptualize literate programming as a technique for CNC milling, we conducted a user study with 6 participants with varying levels of experience in CNC milling and in programming. We recruited participants from our professional connections as well as from a makerspace email list. Participants' CNC backgrounds ranged from no prior experience to those experienced with advanced CAD, CAM, and manual woodworking practice. Similarly, participants' programming skills ranged from passing knowledge to high expertise in scientific computing and computational notebooks.

Specifically, P1 is a researcher who studies computational notebooks, P2 is a materials scientist who has used a Shopbot for a digital fabrication course, P3 is a professor with background in CAD and computational notebooks, P4 is a user experience designer with background in CAD and CAM for mechanical engineering, P5 is a teaching professor who uses computational notebooks for teaching control theory, and P6 is a student who uses CAD and 3D printing to design custom camera components. P3 and P5 also each have over five years of experience with manual woodworking.

Rather than focus on pure usability, our goal for this user study was to better understand how users experienced Imprimer as a novel paradigm. During a 2-hour in-shop session, we asked each participant to walk through two tutorial notebooks to connect to the Shopbot and set up the machine and the material. Next, participants used either the **D1: QuickDraw** (Section 5.4.1) or **D2: FunctionTile** (Section 5.4.2) notebooks to mill a sketch or tile[2]. We paid particular attention to moments of learning, pain points expressed, and code reading and tailoring. After completing the study, we conducted brief semi-structured interview where we asked participants on their experience using the notebooks, how they learned to use the system, and improvements they would like to see implemented

5.5.1 Participants Brought Diverse Fabrication Goals to the Study

Participants came from varied backgrounds and brought their own desires for CNC milling to the study. These included creating custom furniture from precisely specified 2D geometries, rastering and engraving images, and teaching others. Because Imprimer is a prototype for a new paradigm for controlling CNC mills, participants understood that it did not yet have the all of the functionality of more established tools, but were willing to work with the limitations. For example, one participant enjoyed using the FunctionTile notebook's sliders to experiment with tile forms, but wanted additional functionality beyond defining mathematical functions for modifying the surface.

> **P6** I think for many applications, you wouldn't want to use math at all; there are only so many 2D functions that look interesting. I would imagine having [direct manipulation] like with Rhino where the user could have control points and edit the surface that way.

[2]We did not ask any participants to use the MiniShelf notebook (Section 5.4.3) due to the relatively long milling time required.

We plan on adding control points to 3D surface views as future work. Another participant wanted to be able to specify geometric dimensions in a programmatic way, which we had only implemented as low level functionality and not yet as a full-fledged API in **D1: QuickDraw**. Nonetheless, P3 praised the fact that notebooks could be extended later on and that functionality could be imported or swapped out, remarking:

> **P3** Overall, besides [not having the API], the concept is neat and I see a lot of applications. It seems if you can swap out the toolpath generation part to match an application—as long as you could write [code for] toolpath generation—then there's a lot of real world use. Using a notebook feels like a more natural way to interact with a machine.

5.5.2 Code Became Crucial in Understanding Exploratory Milling Processes

All participants, even with those with less experience programming, navigated code cells in the notebooks, often exploring, reading, and experimenting with code from different parts of a given notebook. In some instances, participants stated that they wanted to do things differently than we had already written them in the notebook and proposed their changes by speaking about them in terms of code.

For example, as a matter of preference, P2 wanted to cut a little deeper than the thickness of the stock material into the machine bed to make sure that there were not any artifacts left over. To do this, they created a new cell, `epsilon = 0.01`, to define an additional value to add to the last cut-through pass. P2 then used calipers to visualize one hundredth of an inch physically and then realized that it was too small to be perceptible practically speaking. They then manually edited the `epsilon` cell to be one sixteenth of an inch, i.e., `epsilon = 0.0625`. When asked why they picked this value, P2 reported that they had regularly worked with such increments in the past.

Introducing code to CNC milling also prompted to participants to think about how they might learn either basic CNC functionality through the notebooks, or how they might navigate more experimental workflows. Some participants preferred *writing code* in a scaffolded manner to understand what they were implementing.

> **P4** [For learning] if I started from scratch, I'd think about the different things I could make, the different materials. I would have liked to have written the notebook I'm using myself.

Other participants preferred *reading code*, particularly when exploring novel forms in the **D2: FunctionTile** notebook.

> **P5** I like having text in front of me. I like to start with having many examples, I like to remix what's already there ... Seeing what's already written helps me understand what's possible at all, versus starting with a blank slate.

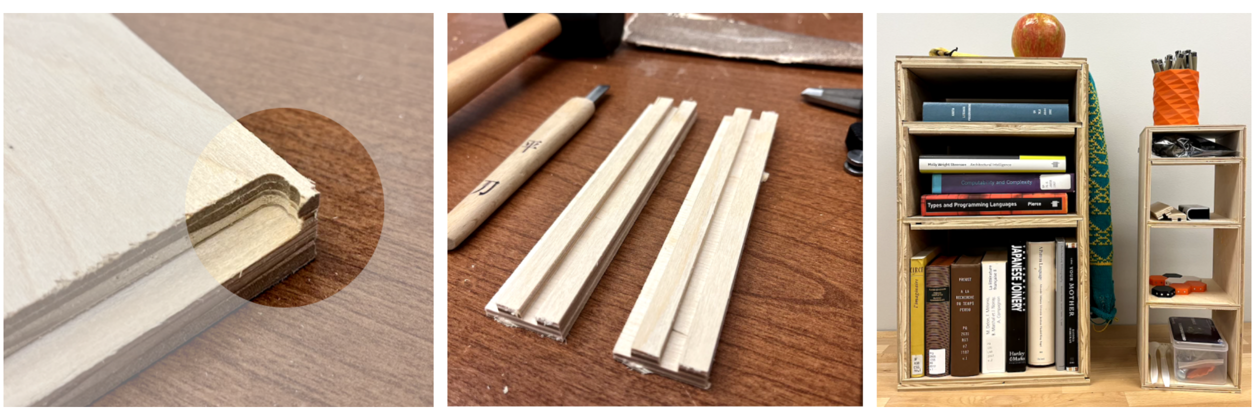


Figure 5.10: Moments from Developing MiniShelf. Left: a miscalculation in generating the rabbet toolpath left artifacts in the corners. Middle: parametrically generated test pieces to check the cut depth of the joints. Right: two shelves each fabricated with MiniShelf.

5.5.3 Negotiating Learning and Making within Notebooks

We noticed that presenting a fabrication workflow in a computational notebook format often uncovered two competing goals for new users: the need to understand how Imprimer and the CNC machine itself work versus the need to actually dispatch jobs and try things out. P1 in particular had a great deal of background in computational notebooks but less in CNC milling, and commented about how CNC-specific practice would need to be introduced and discussed before getting to the "making" parts of the notebook. For example, while physically installing an end mill, P1 remarked:

> **P1** Are you recording what we're doing now? Because that could be helpful for somebody installing the end mill. ... So much information like 80%, 20% of the shaft in the collet—I'm not sure how bad it is if one thing is done wrong versus another.

We discussed with P1 this tension between walking through steps specific to CNC milling versus, in their words, "just getting things done." They suggested that we might structure a series of notebooks in a tutorial-and-reference style akin to programming language tutorials. The tutorial notebooks would then link to reference notebooks on topics for makers might who more information—for example, an entire dedicated notebook to understanding what end mills are, how to install them, and how to program in a notebook to account for their effects. To this end, we extracted the "how to" parts from the **D1: QuickDraw** and **D2: FunctionTile** notebooks and placed them in dedicated tutorial notebooks.

5.5.4 Utility of Custom Views versus Graphical Input Elements

In both the **D1: QuickDraw** and **D2: FunctionTile** notebooks, participants frequently used visualizations to understand how their input would map to toolpaths. We found that, in general, participants preferred to debug their toolpaths by making small adjustments in parameters such as stepover, tool choice, or their choice of sketch and subsequently inspecting the results in a visualization. Some participants found that being able to bridge mathematical renderings with CNC-specific toolpath visualizations in the same visual space unlocked new forms of interaction.

> **P3** This is like when I first got Mathematica, I wanted to see all sorts of things that it would plot. I would just edit the code—is that something I can do here? Okay then! I would divide the denominator with another factor of r … Is that two times symbols in Javascript? Also I might add a phase shift right in the sine function to get more activity in the middle of the material.

Conversely, visual *input* elements often confused participants, who saw code-generated input elements as conceptually separate from other code. This differs from other notebooks like Jupyter Notebook which rarely feature code-defined input elements; in the context of CNC milling, such elements could present more of a cognitive gap than they do with purely digital tasks. Instead, while participants used output visual elements without question, they generally preferred to write raw cell variable values that did not obscure underlying code with graphical controls.

> **P4** How do you even create visual elements, and how do they work? Sliders don't make sense to me. It's difficult to understand the code that generates them. I would want to key in important values.

P3 further remarked that anything that took their attention away from raw code, even though the input elements were themselves defined in code, detracted from their experience of CNC milling through programming.

> **P3** To me, input elements defeat the purpose of the notebook. I want to see the code, and I want to edit things myself. When it comes to widgets, that's something that

> you might want your boss to play around with, but the scientist needs to be working with just the code.

5.5.5 Scaffolding and Sharing Experimental Milling with Others

However, in contrast to other participants, P5 saw great value in graphical input elements because of their potential value for scaffolding and sharing knowledge about novel production processes with others. P5 cited their background as an electrical engineer and engineering educator in how they interacted with the Imprimer notebooks. In particular, they mentioned how computational notebooks such as Jupyter Notebook and Google Colab helped their students collaborate around projects in control theory, and how such code needed to be curated and documented.

> **P5** I really like that you can write code that creates buttons and sliders. This way you could limit the notebook to specific functionalities, like you can adjust and play with these numbers, but not these other ones that might be more dangerous to mill. Like you'd say "I only want you to change this number from 1 to 10." Like, well you could [edit more] if you went into the code, but most students would stick to the controls you give them.

P3 echoed the value of interleaved documentation, saying "*I use Jupyter notebook to teach my classes—having tons of lines of prose for just one line of code is very helpful.*" In addition, P5 noticed which code cells were and were not pinned open to always show their code implementations rather than collapsing the code after editing. While we had originally considered pinning to be a superfluous detail, P5 insisted that proper choice of what code to show and hide was crucial for allowing others to experiment with notebooks without being misled.

5.6 Discussion

Deep engagement with digital fabrication requires both programming and manufacturing knowledge. Fabrication systems in HCI research tend to lower the threshold to fabrication by abstracting away complexity. This approach invites more users in, which is extremely valuable to increase and diversify participation and normalize fabrication practices. Yet, rather than asking how we can facilitate the fabrication process, we were motivated to ask how we can support the acquisition of programming and manufacturing knowledge necessary to transform makers from passive users to active developers of workflows.

Because the computational notebook paradigm offers the ability to weave rich documentation with machine code, it moves the programming practices of machine shops to a wider audience, and with better readability and portability than raw G-code. Instead of espousing a vision of fabrication as more streamlined, with the divide between bits and atoms rendered invisible by "seamless couplings" [IU97], we suggests a different type of interaction with CNC machines rooted in sustained engagement and an attitude of troubleshooting. We argue that a literate programming environment such as computational notebooks best supports this approach, which exposes all the "wiring" while offering scaffolding, support, and flexibility.

For example, P2 demonstrated that their fabrication process was a series negotiations between physical contingencies, the notebook's code, and its visual debugging tools. When they sought to make a profile cut and discovered that this function was not supported in the notebook, they went ahead and wrote their own code for a profile cut. They then adjusted the cut depth by first checking in the notebook's 3D view, then by measuring the stock with calipers. That P2 did so indicated not only their understanding of machine behavior (the CNC machine cutting deeper through the material with each pass, and the importance of defining each pass's depth), but also how their engagement was sustained by the back and forth between programming, digital, and physical verification. Rather than going through the process of "loading computer-aided design (CAD) files into a fabricator" [KZT+18], P2 could directly sketch their design, generate toolpaths, and make adjustments on the fly.

Another tension that arose during the user study was *what* to show in the notebook and the *order* in which to show it. What we initially considered a logical sequence of steps and parameters selection turned out to rely heavily on mental models built on prior fabrication experience. P1 commented several times on the difficulty to know how much precision was required as they progressed through the notebook: "*It's difficult to know how exact I have to be when I measure things, and to know what would go wrong if I didn't measure something right.*" As a result, P1 had to trust the support and indications we provided. This revealed the importance of developing the notebooks to support different levels of expertise: we authored separate notebooks for core workflows versus supplementary skill-building. In parallel, P3 described his experience of using Imprimer as sometimes "confusing" compared to digital-only computational notebooks in which physical machine state is not a concern. In the case of Imprimer, "*mixing state between the notebook and the machine was hard.*" As we work to empower makers to become developers of workflows, these insights became important design considerations for the future development of Imprimer.

Regardless of user background, because they include source code, notebooks can be customized to support various fabrication goals and expertise. We started this project wondering how to write and format the notebooks so that they were legible *and* provided optimal machine control, but this question became less important as more participants used them. Each user came with a different background, programming sensibility and sense of possibilities of what they could make with a CNC mill. The principles of universal design [FO10] paled when faced with the diversity of fabrication experience and contexts, even with such a small sample. For instance, P3 and P4 both mentioned that the graphical elements of the notebooks did not add to the experience of programming the ShopBot. P5, on the other hand, saw great pedagogical value in the ability to constrain the notebooks to specific functionalities and value ranges. While P2 enjoyed the ability to sketch freehand geometry, P3 would have liked to connect the notebook to a drawing API for cutting accurate shapes. The level *and* the locus of complexity varied for each user according to their personal goals and concerns—for some it was at the design level, for others in the machining aspects, for others in the particulars of programming. Complexity was not a fixed variable that one system could address; it was a shifting tension that varied with each maker and with each session.

The entangled challenges and benefits of representing aspects of the machining process through literate programming raise the question of how to appropriately guide makers through a notebook. In particular, we observed a tension between the goals of *getting something done* versus *learning how something works.* Many existing systems will orient makers towards getting things done, hiding complexity and making design decisions that ensure manufacturability and ease of use. Others will offer many functionalities as well as in-app and external support to guide makers through the many features. With Imprimer, we contemplate a third alternative: to show the code underlying each functionality from the start so that makers can develop their own practice of machine control.

5.7 Conclusion and Future Work

Ultimately, Imprimer is only the beginning of an emerging paradigm: digital fabrication as writing literate programs. Leveraging its network of notebooks, a machine, and an augmented reality overlay, we showed how to extend computational notebooks beyond their envisioned use in data science into the physical world. Through demonstrations and a user study, we examined the effectiveness of starting with low-level machine control and abstracting up to more human-centered machine interactions. Future work will be needed to incorporate the vast breadth of existing CNC milling techniques into open source literate code while also adding entirely novel ones. Nevertheless, it is precisely through programming that we aim to anchor further thought and experimentation with CNC milling across physical and digital worlds.

Chapter 6

Making Workflows-as-Programs Reproducible

Despite the rise of experimental digital fabrication in human-computer interaction (HCI) research and beyond, we are missing a robust way to share working implementations of research with others. In contrast, in fields like the basic sciences, reproducibility of past research is fundamental [Per21, BR18, Nat19]. Researchers and practitioners should be able to reproduce, validate, and build upon prior work. In the case of experimental fabrication, those who would benefit from reproducibility are not only novices to digital fabrication, but also experts unfamiliar with a newly invented fabrication workflow.

To illustrate this scenario, consider two personas separated by geographical distance: Florence, a community wood shop manager who is experienced with CNC milling; and Siena, an HCI researcher who is working on new interaction techniques for bringing craft-based techniques to CNC mill control. Florence has developed a workflow for doing *two-sided milling*, where a fabricator carefully mills on two opposite sides of a workpiece using a conventional 3-axis mill. Siena is eager to learn two-sided milling not only to increase the types of objects she can

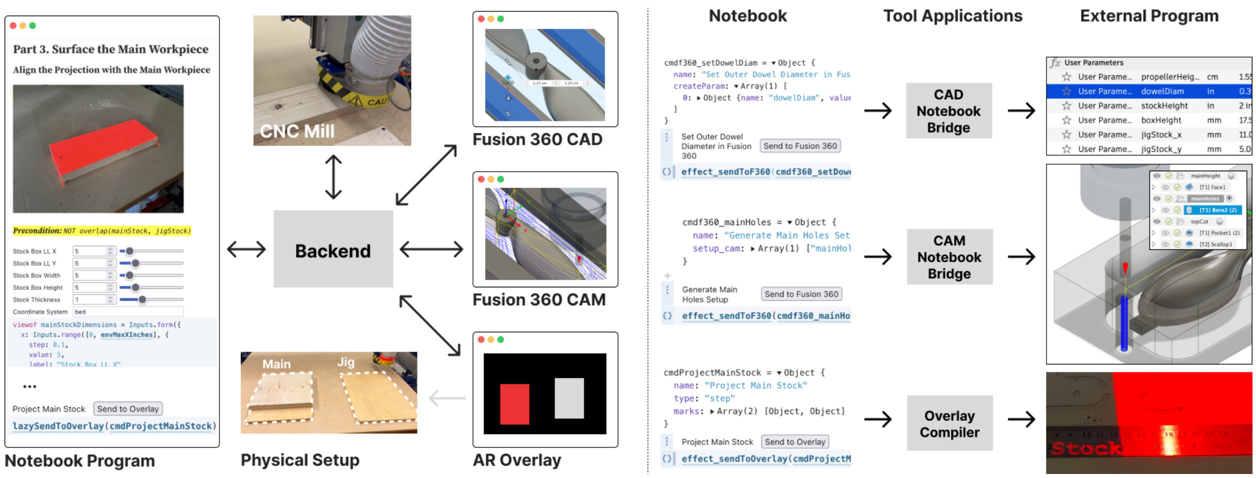


Figure 6.1: Tandem System Architecture. (Left) the notebook interacts with Fusion 360 CAD and CAM, with the AR Overlay, and with the CNC mill via HTTP requests. The notebook makes these requests to a dedicated backend server that forwards the requests to the respective *tool application*, e.g., an add-in we wrote for Fusion 360. The AR overlay (bottom right) prompts the fabricator to make adjustments to the physical setup. (Right) Tandem's API communicates over HTTP with tool applications in the notebook. Top: sending a value calculated in the notebook to be set as a user parameter in Fusion 360 CAD; Middle: automatically generating a toolpath in Fusion 360 CAM to mill alignment holes; Bottom: rendering boxes on the mill's bed at specified positions using the AR overlay.

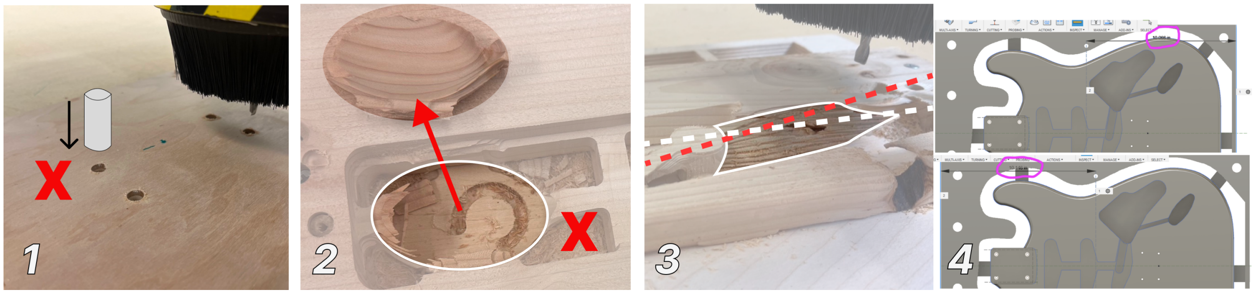


Figure 6.2: Examples of Errors that Arise in Two-Sided Milling. (1) A milled alignment hole is too narrow for the dowel to fit, requiring a second set of holes to be milled. (2) The CAD design of a spoon features a handle that is too narrow and gets torn off by the mill. (3) Flipping the workpiece over an axis that deviates too much from an assumed axis of rotation results in an invalid post-milling geometry. (4) An error taken from an online forum [Ste19] where a fabricator reported misaligned features while milling a guitar. Another forum user responded that the reason was that the alignment holes were asymmetric across the axis of rotation assumed by the original poster.

manufacture on her CNC mill, but also to apply techniques used in Florence's workflow to her own research.

However, there is currently no way for Florence to create reproducible versions of her novel digital fabrication workflows beyond writing a tutorial—a helpful but incomplete representation. When Siena follows text or video tutorials (e.g., [Mar16, Jer20, Vec21, LaC17, Bra23]), she must manually re-implement every step of the workflow using her own software tools and machine hardware. Imprecise re-implementation can result in errors that are time-consuming and difficult to debug because Siena needs to reason about interdependent physical and digital states. Figure 6.2 shows example errors that arise in two-sided milling. In contrast, for her digital-only programming projects, Siena can easily fork repositories, run the code to get an immediate result, read parts of code in depth, and experiment with her own changes.

In particular, three challenges stand out for Siena in replicating Florence's workflow.

1. Florence uses several disparate software tools interdependently, including conventional CAD/CAM software. There is no single environment to present a linear sequence of steps

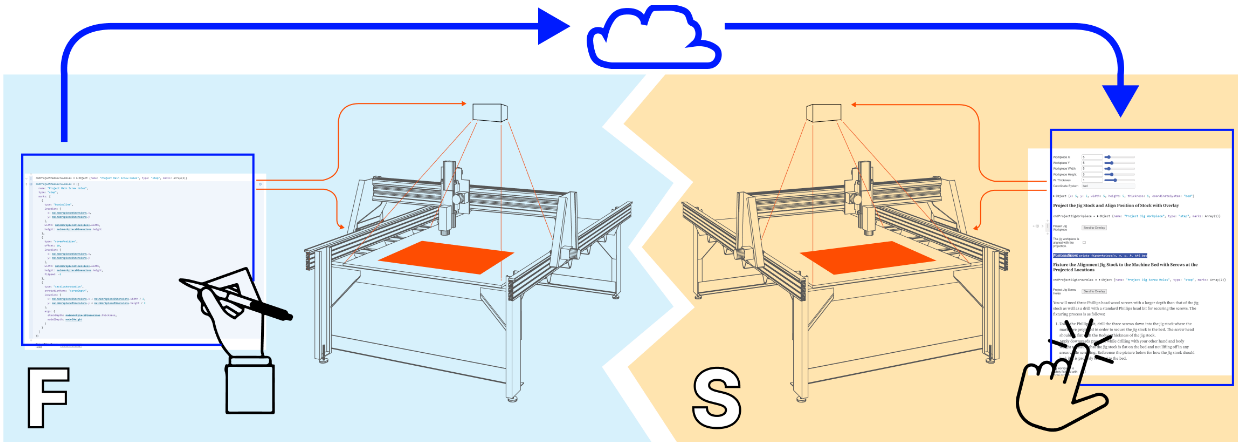


Figure 6.3: Programming and Handing Off a Notebook. Left) Florence refines her two-sided milling process on her own. She then implements the steps of her process in notebook cells by writing source code with Tandem's library calls. Right) Siena navigates to the notebook in the browser, starts the Tandem backend on her laptop, and connects it to the projector and CNC mill. She works through the notebook to replicate Florence's workflow.

for Siena to execute in code.

2. Siena must adjust physical machine and material settings to match assumptions made in digital code, or vice-versa; it is difficult for Siena to know when to manually intervene and how to do so.

3. Each step of Florence's workflow has implicit preconditions about what must be satisfied before running the mill. It is difficult for Florence to explicitly formulate these "gotchas" so that Siena does not repeat the same mistakes Florence encountered.

To provide an infrastructure for implementing reproducible versions of novel fabrication workflows, we present Tandem, a library and backend that lets fabricators like Florence implement their entire workflow as a computational notebook program (Figure 3.1). Florence uses Tandem's API to author her workflow as function calls to CAD and CAM software, to an *augmented reality (AR) overlay* for manual interventions, and to the CNC mill itself (Figure 6.1). Tandem also lets

Florence include *assertions* in her code that check preconditions that she provides before fabricators like Siena execute error-prone milling steps. Fabricators like Siena then reproduce the workflow simply by executing the code, following instructions that the notebook produces, and manually intervening as directed by the AR overlay; Siena may also interact with the code itself if she wishes to adjust parts of the workflow.

We evaluate the feasibility of Tandem by implementing an entire end-to-end two-sided milling workflow as a computational notebook program and producing four artifacts as a result (Section 6.5). We chose two-sided milling as an example novel fabrication process because it is more difficult than many; it involves tool changes, synchronizing machine and material setups in physical and digital states, measurement and manual inspection, and experimenting with milling feed and speed. We elected to explore this single workflow in depth rather than explore multiple digital fabrication workflows at a shallower level. While Tandem's features are most clearly showcased in two-sided milling, they can be applied to many other novel fabrication processes as well, including 3D printing, laser cutting, and laboratory automation (see Section 6.6.3).

The structure of the paper is as follows: after describing related work, we first walk through how a fabricator who reproduces a workflow (Siena) uses a Tandem notebook; then, we walk through how an authoring fabricator (Florence) implements the workflow's CAD, CAM and machine control functionality using Tandem's API (Section 6.2). We then detail how Florence programs physical steps using the AR overlay (Section 6.3), and how Florence guards against a large class of easily preventable errors using assertions (Section 6.4). Finally, we describe the demonstration artifacts fabricated with the notebook alongside artifact-specific assertions (Section 6.5) before concluding with lines of future work that are now possible with a reproducible fabrication workflow (Section 6.6).

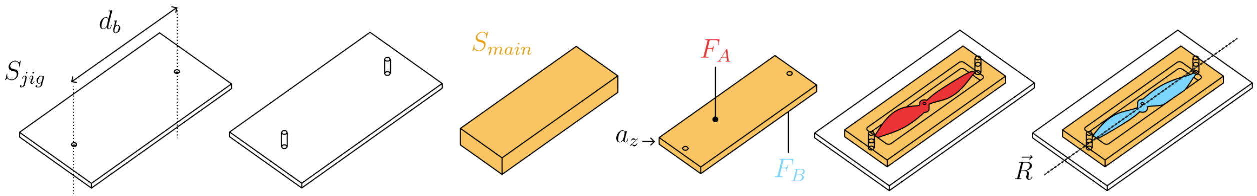


Figure 6.4: Schematic of Two-Sided Milling a Propeller in Tandem. The fabricator, Siena, mills alignment holes in a sheet of wood and inserts two dowels to form the *alignment jig* S_{jig} (white). Moving to the *main workpiece* S_{main} from which the propeller will be milled (yellow), she mills the S_{main} down to exactly a_z in height and mills two additional alignment holes. We denote the top and bottom faces of S_{main} as F_A (red) and F_B (blue), respectively. Next, she fixtures S_{main} on top of S_{jig} and mills F_A in two passes: a roughing pass and a finishing pass (not shown). Finally, flipping S_{main} over the predetermined axis of rotation $\vec{R}$, Siena mills all areas reachable from the bottom of the propeller into F_B, again in two passes.

6.1 Walkthrough: Replicating an Existing Workflow

We now provide a high-level walkthrough of how a fabricator like Siena would follow Florence's notebook to replicate Florence's workflow; her goal is to CNC mill a spoon (Section 6.5). Section 6.2 provides a code-level explanation of how the workflow is implemented in part of the notebook.

To start the walkthrough, Siena would need to have installed Tandem's backend on her computer; then, she opens an Observable computational notebook [Obs23b] in her web browser; the notebook contains all the source code, prose, and input elements that she uses to interact with CAD/CAM tools and an AR overlay for interacting with the physical setup of materials on the mill's bed and with the mill itself. Note that our implementation provides a messaging service for the backend to communicate with the Shopbot PRSAlpha Mill [Sho22] that we use; to use another machine instead, the fabricator would need a service built for that machine or custom built if one does not exist.

Once the notebook is open, Siena proceeds through the 6 notebook sections, which we represent in respective subsections below. We present the high-level progression of these steps in Figure 6.4 and moments corresponding to each step in Figure 6.5.

6.1.1 Prepare the CAD File

Siena opens the Fusion 360 CAD file of the spoon she wants to mill in Fusion 360. While there are several approaches to two-sided milling, Tandem's current approach uses two *alignment dowels* that hold the *main workpiece*, i.e., the piece of material being milled into a spoon, in the same place after Siena flips it over to mill the second side. This means that Siena must add an extra part to her CAD file, which we call the *outer*. The outer, pictured in Figure 6.5 Part 1, contains two precisely positioned holes for the alignment dowels and connects to the spoon itself using tabs. Using the notebook, Siena enters the diameter of the dowels she obtained beforehand, and the notebook automatically generates the outer in Fusion 360 CAD.

6.1.2 Mill the Alignment Jig

Next, Siena mills the *alignment jig*, which holds the main workpiece of the spoon in place while it is being milled (Figure 6.5 Part 2). Using the AR overlay, the notebook guides Siena through cutting down and positioning the *jig workpiece* from which she will mill the jig. The overlay projects a rectangle onto the the CNC mill's bed where the stock material goes. Once the physical stock is synchronized with the projection, the notebook changes the overlay to a new projection that shows Siena where she should drill screws to fixture the stock to the mill's bed. Using the dimensions gathered about the jig workpiece, the notebook then generates a toolpath in Fusion 360 CAM for the machine to mill the alignment holes in the jig workpiece. It then guides Siena through manually setting the zero (the XYZ origin) on the CNC mill to match the zero assumed by the toolpath in CAM. Finally, the notebook generates low level machine instructions for the toolpath and lets Siena dispatch the instructions to the mill directly from the notebook. After the holes have been milled in the jig workpiece, Siena presses the dowels into place to complete the jig.

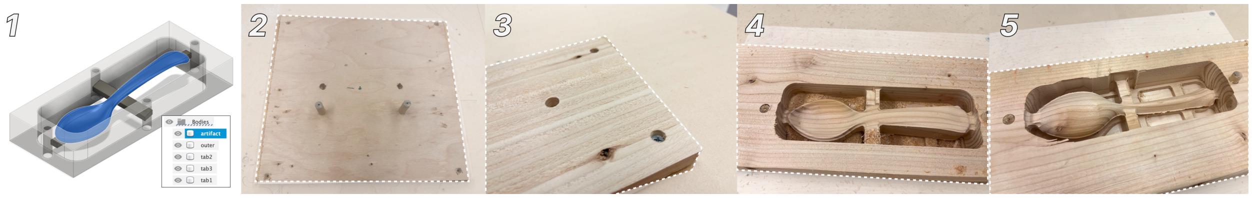


Figure 6.5: Moments from Siena's Steps in the Notebook. (1) Given the already modeled spoon, Tandem generates the outer and tabs that surround the spoon. (2) A completed alignment jig. (3) The main workpiece S_{main} after being milled to the height of the spoon a_z. (4) The top face F_A of the main workpiece S_{main} after milling the top-down cut. (5) The bottom face F_B of the main workpiece S_{main} after milling the bottom-up cut.

6.1.3 Mill the Main Workpiece to the Correct Height

Next, Siena moves to the main workpiece and mills it down to the exact height of the spoon that will be milled. She mostly repeats the same steps she took with milling the alignment jig except for using the main workpiece. However, because the mill will remove all material above a certain height, she must be careful to screw the main workpiece to the bed so that the screws lie below the lowest cut that the mill will make. To do this, Tandem uses *counterbores* that position the top of a screw lower than the top of the material that it is screwed into, as shown in Figure 6.5 Part 3 post-milling. The notebook calculates the depths and locations of the counterbores that Siena needs to make and projects a visual guide using the AR overlay (Figure 6.11 Right).

6.1.4 Mill Alignment Holes in the Main Workpiece

Once the main workpiece is milled to the exact height of the spoon, Siena works through steps in the notebook to mill alignment holes in the main workpiece. This is the same as milling holes in the alignment jig, except applied to the main workpiece.

6.1.5 Mill the Top-Down Cut (Face A)

After the four previous preparation steps, Siena is finally ready to mill the spoon itself. The notebook guides Siena through otherwise error-prone steps. First, the the notebook generates toolpaths in CAM that mill the main workpiece from the "top-down" before Siena flips the workpiece on the alignment jig and mills "bottom-up" from the perspective of the workpiece. Next, the notebook asks Siena to establish a designated axis of rotation $\vec{R}$ around which she will flip the part. the notebook adjusts the origins in CAM to match this axis. Finally, the notebook prompts the AR overlay to preview the cut before milling so that Siena can visually confirm that the cut looks correct before dispatching it from the notebook; the workpiece then looks like that shown in Figure 6.5 Part 4.

6.1.6 Mill the Bottom-Up Cut (Face B)

At this point, Siena has milled the spoon completely from the top downwards, but she must flip it over to mill the parts that were unreachable with the workpiece's first orientation. To this end, the AR overlay prompts Siena to flip the workpiece in this direction according to $\vec{R}$, which was set previously. Siena then mills the bottom-up cut (Figure 6.5 Part 5), removes the main workpiece from the mill, and separates the spoon from the tabs, concluding the workflow.

6.2 Implementing a Workflow using Tandem

To better describe how Florence uses Tandem's API to implement her workflow as a notebook for Siena to use, we walk through the code for the first two sections of the notebook[1] : preparing the CAD model and milling the alignment jig.

[1] The source code for the entire notebook and the Tandem backend can be found at `http://depts.washington.edu/machines/projects/tandem/`.


```
viewof dowelDiam = coeffect_range([0.1, 2.0], {
  label: "Dowel Diameter (in)",
  step: 0.01
})
```

Figure 6.6: Prompting for User Input with a Coeffect Function. Some properties are omitted in the listing for brevity.

6.2.1 Prepare the Model in CAD

In our implementation, the notebook communicates, for both CAD and CAM functionality, with Autodesk Fusion 360. We assume that Siena has already modeled the object she wishes to fabricate in Fusion 360. Then, she uses the notebook to talk directly to Fusion 360 and make necessary adjustments for two-sided milling. For example, the notebook prompts her to measure the diameter of two dowel to be used to hold the workpiece in place during milling. To do this, the notebooks uses a *coeffect function*, which generates a user input element; when Siena types in the diameter or adjusts the slider, that value is captured in the variable `dowelDiam`, as shown in the notebook snippet in Figure 6.6. Since CAD and CAM state is stored in Fusion 360, Siena can still make adjustments in Fusion 360 and re-run any necessary effect functions from the notebook.

For all notebook snippets, the *notebook cell*, which contains the source code, appears below the result of evaluating the code. In this case, evaluating the `coeffect_range` function generates the slider element above and assigns the current value of the slider to `dowelDiam`.[2]

Then, in Figure 6.7, Siena sends the diameter of the dowel to Fusion 360 to be set as a *user parameter*, which is a value that Fusion 360 can reference to set the actual diameter of the hole in CAD, mutating the model accordingly. To do this, she uses an *effect function*, i.e., a function that generates a button that, when pressed, sends the value to Fusion 360. Effect functions send

[2]For those familiar with Observable Notebook's input functions, `coeffect_range` is an alias for `Inputs.range`.

```
cmdf360_setDowelDiam = ▼Object {
  name: "Set Outer Dowel Diameter in Fusion 360"
  createParam: ▼Array(1) [
    0: ▶Object {name: "dowelDiam", value: 0.37, unit: "in"}
  ]
}
```

```
cmdf360_setDowelDiam = ({
  name: "Set Outer Dowel Diameter in Fusion 360",
  createParam: [{
    name: "dowelDiam",
    value: dowelDiam,
    unit: "in"
  }]
})
```

Set Outer Dowel Diameter in Fusion 360 [Send to Fusion 360]

```
effect_f360(cmdf360_setDowelDiam)
```

Figure 6.7: Mutating an External CAD Model with an Effect Function.

messages to Fusion 360 CAD and CAM, to the AR overlay, or to the CNC mill—we refer to these recipients as *tool applications*. They take *commands* as arguments, which contain parameters that dictate what the receiving tool application should do; for example, Figure 6.11 shows commands for changing the AR overlay. We generate a button that must be pressed first, rather than simply sending the value, because Observable Notebook is a live programming environment that frequently re-evaluates cells when related cells change. Thus, eagerly sending the values would result in extraneous messages being sent.

Once Siena presses the button, Tandem knows that the model in Fusion 360 has the correct user parameter for the alignment holes. Afterwards, the notebook walks her through several more steps, including: generating the *outer*, which bounds the artifact to be milled; generating the alignment holes in the outer; and generating tabs that connect the outer to the artifact. At the end


```
viewof jigWorkpieceDimensions = coeffect_form({
  x: coeffect_range([0, envMaxXInches], { ... }),
  y: coeffect_range([0, envMaxYInches], { ... }),
  ...
})
```

```
▼Object {
   x: 13.6
   y: 5
   width: 10
   height: 5
   thickness: 0.7
   coordinateSystem: "bed"
 }
```

```
jigWorkpieceDimensions
```

Figure 6.8: Combining Multiple Coeffect Functions with `coeffect_form`. The function evaluates to an object containing the values set on the sliders as values representing the contingent position and dimensions of a workpiece.

of this notebook section, the CAD model is ready to be transformed through CAM into *toolpaths* that dictate the mill's motion.

6.2.2 Mill the Alignment Jig

The alignment jig consists of a flat sheet of wood (the *jig workpiece* S_{jig}) with two dowels that are press-fit into respective alignment holes. To properly generate the toolpath for the alignment jig, Siena must empirically gather data about the physical setup of the sheet material.

To align the physical dimensions and position of S_{jig} with code in the notebook, Siena uses a coeffect function (sliders) to generate a contingent list of dimensions and position on the mill bed (Figure 6.8).

In Figure 6.9, she then uses the gathered data to construct a *command* for the AR overlay (more information in Section 6.3). Next, she passes the command to an effect function to project a rectangle of light corresponding to the contingent dimensions via the AR overlay, as shown in Figure 6.11 Photo 1.

Siena iterates by changing the dimensions on the sliders and/or repositioning or cutting down the jig workpiece. After this iterative process, the AR projection, representing the dimensions sent from the notebook, matches the top of the physical jig workpiece. Siena then checks a box generated by a coeffect function (Figure 6.10) so that the function now evaluates to true; this result can subsequently be used in an assertion (see Section 6.4).

Once the dimensions and position in the notebook are matched with the physical setup, the rest of the section generates a CAM setup with this information, helps Siena zero the tool according to the CAM setup, fixtures S_{jig}, generates a toolpath from the setup, and dispatches the cutting job.

```
cmdProjectJigWorkpiece = ▼Object {
  name: "Project Jig Workpiece"
  type: "step"
  marks: ▼Array(1) [
    0: ▶Object {type: "box", location: Object, width: 10, height: 5}
  ]
}
```

```
cmdProjectJigWorkpiece = ({
  marks: [{
    type: "box",
    location: {
      x: jigWorkpieceDimensions.x,
      y: jigWorkpieceDimensions.y
    },
    width: jigWorkpieceDimensions.width,
    height: jigWorkpieceDimensions.height
  }],
  ...
})
```

Project Jig Workpiece [Send to Overlay]

```
effect_sendToOverlay(cmdProjectJigWorkpiece)
```

Figure 6.9: Using Position and Dimensional Information to Generate a Command. When sent to the AR overlay, the command generates a rectangular projection on the machine's bed corresponding to the values provided.

The jig workpiece is aligned with the projection. ☐

```
viewof jigWorkpieceCorrect = coeffect_toggle({
  label: "The jig workpiece is aligned with the projection.",
  invalidator: jigWorkpieceDimensions
})
```

Figure 6.10: Coeffect Function for Checking that the AR Overlay's Workpiece Projection Aligns with the Physical Workpiece.

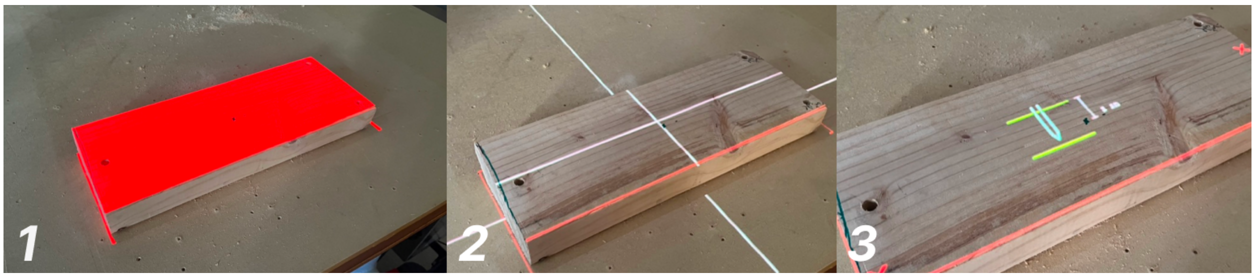
6.3 The AR Overlay

The preceding programming examples show how Florence programs the AR overlay to let Siena "read from" and "write to" the state of the physical setup. Through a declarative syntax, Florence writes code to help Siena position a workpiece, drill fixture screws at designated points, double-check machine zeroes, and more. Specifically, the AR overlay is an interface projected from a projector mounted above the CNC mill onto the mill's bed. We implemented the AR overlay as a standalone web page whose appearance changes based on *overlay commands* (see below) sent from the notebook.

6.3.1 Overlay Grammar

Siena interacts with the AR overlay by sending commands in JSON syntax to Tandem's backend. A command consists of projected *marks*, which help her complete a step in the notebook. A mark could be an outline of where she should place the stock, a crosshair indicating where the center, or zero, is, or a more complex visualization, such as the toolpath of certain steps, like surfacing the stock. For each step in the notebook, Siena can send a group of marks to the projector using the "Send to Overlay" button, which appears after calling the respective effect function. The overlay and notebook together provide visual and text directions on how to accurately complete each step needed for the two-sided milling workflow.

The code snippets included in Figure 6.11 show the syntax fabricators use to create different overlay commands to send to the projector. Florence specifies several properties in the command for Siena, e.g., the type of mark, mark location, and arguments to pass to the overlay. For example, for the `screwDepth` annotation, stock thickness and model height are both needed to calculate a safe drilling depth.




```
projectMainStock = ({
  name: "Main Stock Bounds",
  marks: [{
    type: "box",
    location: {
      x: mainStock.x,
      y: mainStock.y
    },
    width: mainStock.width,
    height: mainStock.height
  }]
})
```

```
projectMainZero = ({
  name: "Preview XY Zero",
  marks: [{
    type: "crosshair",
    location: mainZero
  },
  {
    type: "box",
    outline: true,
    location { ... }
  }]
})
```

```
projectMainScrewHoles = ({
  name: "Main Screw Holes",
  marks: [{
    type: "sectionAnnotation",
    annotation: "screwDepth",
    location: { ... },
    args: {
      stockDepth: m.thickness,
      modelDepth: modelHeight
    }
  }]
})
```

Figure 6.11: Example Overlay Commands on the Physical Setup (top) and the Command Syntax (bottom). (1) Projecting a red rectangle with the same dimensions as the stock helps the user place the stock correctly on the bed before milling. (2) Projecting a crosshair at the center (or zero) of the stock helps the user zero the machine at the center of the stock, as assumed in a corresponding CAM setup. (3) The `screwDepth` annotation generates lines that show the top/bottom surfaces of the stock and drill bit as well as a label that indicates how deeply the user must drill the screws when securing the stock to the bed before milling.

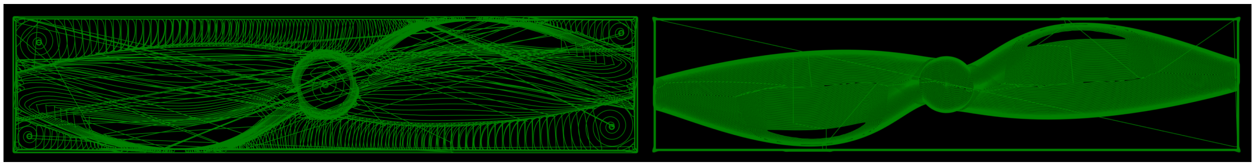


Figure 6.12: Toolpath Visualizations for the Roughing (left) and Finishing (right) Milling Passes for F_A of the Propeller.

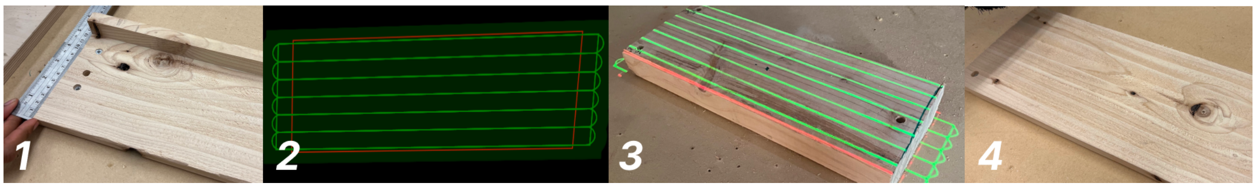


Figure 6.13: Preventing Leftover Stock while Milling the S_{main} to the a_z. (1) Slight deviations in the physical position of S_{main}, versus the ideal position, can cause leftover "slivers" of unmilled stock. (2) A screenshot of the web interface for the AR overlay with a visualization of the toolpath for cutting down to a_z. The red boundary is the calculated boundary of S_{main}, and the green lines represent the path of the end mill. The fabricator adjusts the "stock offset" option in CAM to increase the bounds of the green lines beyond the red boundary. Too little offset results in unmilled stock, while too much results in wasted time spent milling air. The entire rendering in the web interface is warped using an empirically gathered perspective transform to account for the positioning of the projector above the machine. (3) The overlay projected onto S_{main}. (4) S_{main} after being milled with the corrected toolpath.

6.3.2 Toolpath Visualizations

One important mark for overlay commands is `toolpath`, which projects a visualization of the toolpath for a certain step in the milling workflow. For example, Figure 6.13 shows an overlay for milling the main workpiece down to the correct height. We implemented the "toolpath" mark by extending techniques proposed in previous work on AR toolpath visualization [TOJP22]. Figure 6.12 shows two overlays corresponding respectively to the roughing and finishing passes for F_A for the propeller.

6.3.3 Calibrating the Overlay

To calibrate the overlay for her machine-projector setup, at the beginning of the notebook, Siena projects a calibration rectangle onto the CNC mill's bed. She sets the position and dimensions of the rectangle to match a ground truth rectangle on the mill's bed; the rectangle should encompass the area in which she will place workpieces. Siena then drags the distorted projected rectangle's corners to match the ground truth rectangle on the bed. From these four point transforms, Tandem computes a homography that ensures that marks appear at the correct x and y location on the bed at depth $z = 0$. For greater depths, i.e., marks projected on a workpiece, we downscale the projection by a factor of $1 - \frac{\theta}{d_{lens}}$ where θ is the thickness of the workpiece and d_{lens} is the distance from the projector's lens to the machine bed. With this relatively simple calibration technique, we found that test marks projected where precise within 3mm of the specified position. More sophisticated techniques could reduce this margin of error.

6.4 Physical Digital Assertions

When implementing her workflow in Tandem, Florence wants to prevent future fabricators like Siena from encountering the same errors she did. As with any novel fabrication workflow, errors arise from many sources: digital logic, material behavior, artifact geometry (CAD), manufacturing settings (CAM), and setups on the physical machine. Further, many errors arise from mismatched assumptions between these disparate factors. For example, CAM software may assume a given coordinate space origin (the *zero*), but the zero assumed by the fabricator in their physical setup is different (see Figure 6.14). Errors with novel workflows can range from being time-consuming at best to catastrophic at worst.

To solve this problem, Tandem lets workflow implementers like Florence write *physical-*

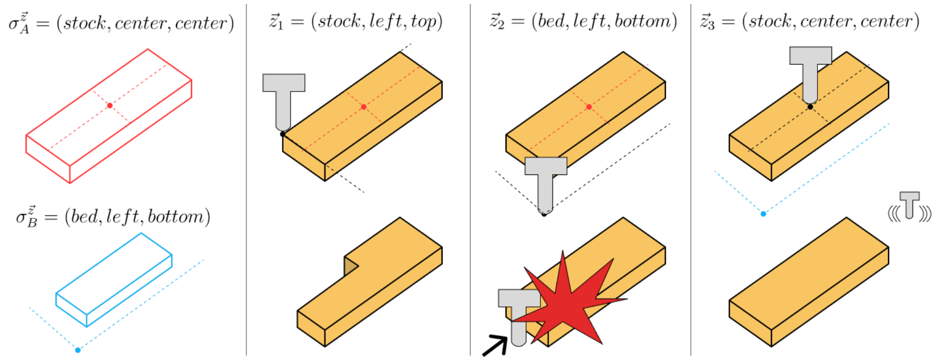
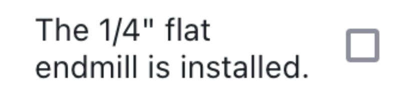


Figure 6.14: Potential errors resulting from differing origins between a CAM setup and a physical setup. (Left) A CAM setup $\sigma_A^{\vec{z}}$ where the origin ("zero") is designated as the center of the stock on the x and y axes (red), and a CAM setup $\sigma_B^{\vec{z}}$ where the origin is designated as a constant location on the machine bed (blue). (Right) assuming $\sigma_A^{\vec{z}}$, if the fabricator sets the physical zero $\vec{z}_1$ at the top left of the stock, the mill will cut through only half of the stock. Similarly, assuming $\sigma_A^{\vec{z}}$, with $\vec{z}_2$ as the physical zero, the mill will crash catastrophically into the stock and the bed. Assuming $\sigma_B^{\vec{z}}$, if the fabricator zeros on the stock at $\vec{z}_3$, then the mill will instead cut air far from the stock.

digital assertions in the workflow's code. In digital-only programs, assertions check whether a *predicate* evaluates to true, and, if not, typically halts program execution; for example the Python statement `assert material_thickness > 0` will raise an exception at runtime if the variable `material_thickness` is less than zero. Assertions are available in most modern programming languages, including some end user programming environments [BCP+03]. The goal of physical-digital assertions is to extend the reach of assertions from digital-only code to now include fabrication workflows.

6.4.1 Example Assertions

We briefly walk through examples of how Florence notices where she can write assertions and how she does so in Tandem. As a simple example, Florence wants to let fabricators know to install a quarter-inch flat end mill before milling the alignment jig. Intuitively, Florence wants to make sure that Siena and any other fabricators using the notebook mill the alignment jig only if



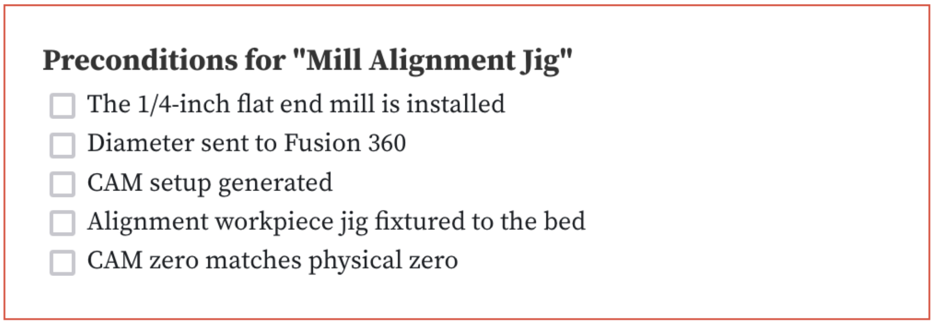

```
viewof mainEndMillInstalled = coeffect_toggle({
  label: ’The 1/4" flat end mill is installed.’
  invalidator: [
    surfacingEndMillInstalled,
    ballEndMillInstalled ]
})
```

Figure 6.15: A basic predicate that prompts the fabricator to manually check that a given end mill is installed.

they have first installed the correct tool. Equivalently, we can say that the code that mills the jig executes only when a prior tool installation assertion evaluates to true.

To implement the most basic form of tool change assertion, Florence writes her precondition as a predicate in Javascript code as shown in Figure 6.15. As with other calls to `coeffect_toggle`, the notebook generates a checkbox with a label instructing Siena to check the box only if she has manually changed the tool to the specified end mill. Because `coeffect_toggle` evaluates to true if the checkbox is checked and false otherwise, `mainEndMillInstalled` is a valid predicate. In an adjacent notebook cell, Florence can also describe in prose why her precondition is necessary, e.g., to prevent leftover material that can result from drilling the jig with a ball end mill.

Florence then passes `mainEndMillInstalled` as an argument alongside other predicates to the `requires` function which evaluates to true if and only if all the predicates evaluate to true. As shown in Figure 6.16, the result of `requires`, assigned to `precondFinishing`, is thus a "checklist" of preconditions that must be fulfilled before Siena can a run the jig milling step on the physical machine. She passes the result to the effect function which sends a job to the mill, `effect_sendToMill(jigInstructions, preCondFinishing)`; if `precondFinishing` evaluates to false, then the button generated by `effect_sendToMill` will be disabled to prevent error-prone




```
viewof preJig = requires("Mill Alignment Jig", [
  ["The 1/4-inch flat end mill is installed", () => jigEndMillInstalled],
  ["Diameter entered into CAD", existsDowelDiam()],
  ["CAM setup generated", jigSetup],
  ["Alignment jig workpiece fixtured", jigWorkpieceFixtured],
  ["CAM zero matches physical zero", checkJigZeros()]
])
```

Figure 6.16: Preconditions for milling the alignment jig.

milling. Altogether, combination of `requires` and `effect_sendToMill` comprises a physical-digital assertion. Florence then repeats the process for each milling step in the notebook, first writing predicates that are specific to each step, then combines them using `requires` whose resulting value she passes to each milling effect function.

As a more complicated example, consider the final predicate in Figure 6.16, “CAM zero matches physical zero,” which the `checkJigZeros` function implements as a predicate. This predicate means that the origin used in the CAM setup—where the point $(0, 0, 0)$ is assumed to be in relation to the stock material—must match the true physical location at which the machine was zeroed (see Figure 6.14). To zero the machine, Siena must first move the tool head to the desired location and then run the zeroing command on the mill, which sets the location to $(0, 0, 0)$. This establishes a local coordinate system relative to the stock material, while the world coordinate system remains relative to the unchanging machine bed.

To implement this predicate, Florence must check that the origin used in the CAM setup visually matches the zero set in hardware using the overlay and user confirmation. To do so, she writes an effect function to safely move the mill's tool to the zero currently in hardware. The notebook then prompts fabricators like Siena to visually compare the current position of the tool in hardware with the zero assumed in the CAM setup; Siena checks the predicate's checkbox to confirm a visual match.

6.4.2 Scope and Limitations of Assertions

The goal of assertions is to allow Florence to prevent a large set of predictable errors from occurring when Siena uses the notebook by explicitly including checks and preconditions in the notebook's code. Just like in digital-only code, assertions cannot prevent all errors. In Tandem's case, assertions focus on errors that are easily preventable as long as Siena is made aware of prerequisite steps such as mismatched CAM zeros, tool changes, and material positioning. While these checks may be ”trivial” to those experienced with a given workflow, they are easily missed as workflows grow more complex or deviate from techniques that Siena is used to.

Assertions can be more or less useful depending on their implementation in code. For example, Florence could improve the simple checkbox that Siena checks to verify that the correct tool is installed. Namely, she could implement a model that uses computer vision to classify the currently installed tool[3]. In addition, rather than using a hard-coded end mill, Florence could query the current CAM setup's designated tool using a coeffect function. Currently, predicates primarily rely on user verification such as checkboxes to “sense” physical state; this suffices for many types of assertions we encountered. Still, in future work, we intend to leverage sensor data, such as video feed, touch sensors, and audio data to automatically satisfy or invalidate assertions based

[3]On CNC mills with automatic tool changers, Florence could design an assertion by querying the machine's hardware for the current tool. However, on more common mills without automatic tool changers, more work is required on the notebook side.

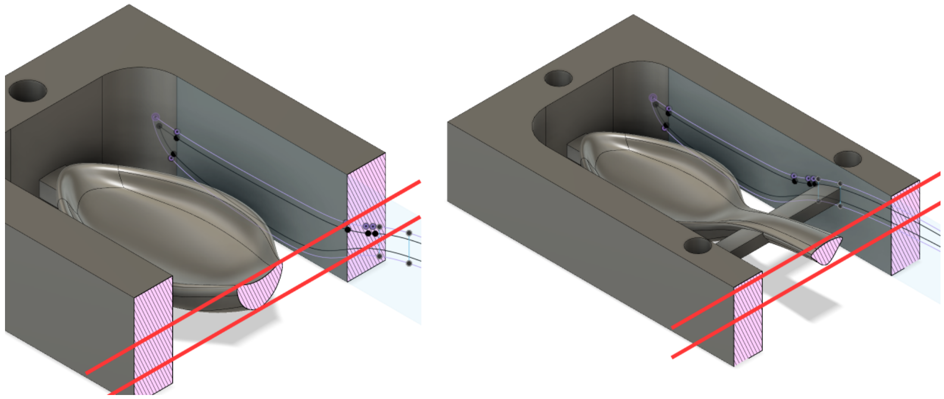


Figure 6.17: Minimum Thickness Assertion. From the notebook, an effect function enables Fusion 360 CAM's section analysis tool and prompts the fabricator to inspect the minimum thicknesses of the XZ cut planes. After inspecting this, the fabricator checks a checkbox in the notebook, which returns true, fulfilling the assertion.

on empirical data (see Section 6.6.2).

Finally, assertions are not meant to be absolutely binding. Like any programmer, Siena can check off, rewrite, or remove an assertion if she finds her own working style is at odds with an assertion that Florence implemented. Even in this case, Florence's assertion still serves as a "heads up" and Siena can add the assertion back in later if needed. Also, we note that physical-digital assertions are *not* the same as design rule checks that occur solely within CAD or CAM, e.g., checking for collisions while simulating a toolpath. Instead, these assertions concern potential errors that cross *between* physical setup and CAD/CAM, such as synchronizing the zero in CAM with the zero set on the machine.

6.5 Demonstrations

To demonstrate Tandem, we (the authors) used our two-sided milling program to mill four artifacts: a propeller, a spoon, a bowl, and a printed circuit board (PCB). As we fabricated each artifact, we documented the challenges we faced, e.g., the spoon snapping during a roughing operation. To prevent other fabricators from encountering these same errors, we formalized assertions for each object or implemented additional notebook functionality to support future notebook users. Again, in all cases, we assume that the fabricator has already designed an initial 3D model or PCB layout of the artifact they wish to produce since Tandem focuses on post-modeling steps of the workflow.

6.5.1 Propeller

We first fabricated a drone propeller with extremely thin blades (Figure 6.18) that are difficult to mill (4.7mm thick, 248mm total length). Because the propeller was relatively long and thin, it was prone to heavy vibration during the bottom-up cuts. Further, we had to ensure that the line through the alignment holes was completely parallel to the machine's y axis to avoid moving the alignment jig post-fabrication. In one instance, having both a misaligned alignment axis and insufficient tabs tore through a blade (Figure 6.2, 3).

To minimize the chance of tear-out occurring while milling the thin blades, we established two assertions: (1) there must be four tabs holding the propeller to the outer, two at the blade tips and two at the center of the propeller, and (2) the alignment holes must be axis aligned. To implement the first assertion, we simply ask in the notebook whether the fabricator has manually created four tabs in CAD; to implement the second, we projected the location of the alignment holes as calculated earlier in the notebook, alongside a line representing the axis of rotation $\vec{R}$. Using a coeffect function, we manually checked whether the true alignment holes corresponded with the projected locations; if excessive deviation is observed, we added code in the notebook bridge to guide the user to re-mill the alignment holes. Finally, using the `precondition` function, we required both checks before a fabricator could begin the top-down and bottom-up cuts.

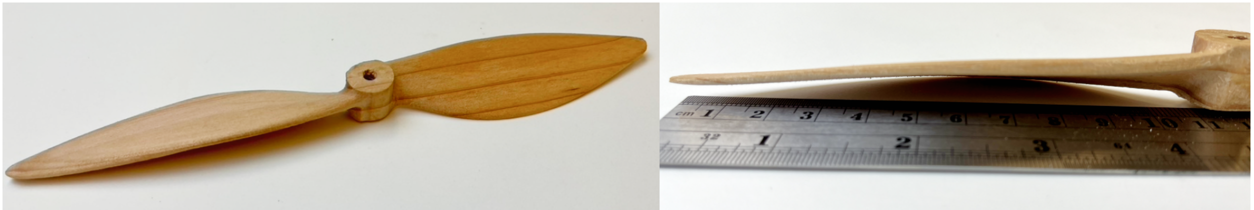


Figure 6.18: The Propeller Milled from Fir.

6.5.2 Spoon

Compared to the propeller, fabricating the spoon raised additional challenges due to its asymmetrical form in the XZ plane; it has a low, shallow bowl with a long, thin handle that rises up vertically (see Figure 6.19). Despite creative placement of tabs, we found that if any part of the spoon's handle was below a minimum thickness, the spoon bowl would tear out during milling.

To counteract this, we developed an assertion that enforces a minimum thickness requirement along the longest axis of the spoon, in our case, the y axis. Formally, we express this assertion as checking the intersection of the XZ cut plane with the spoon at all points on the y axis through the body of the spoon:

$$\forall y_{A_{min}} \leq y_i \leq y_{A_{max}} : min_\theta(A \cap plane(y_i)) \geq \theta_{min},$$

where A is the body denoting the artifact, in this case the spoon, $plane(y_i)$ is the XZ cut plane with its origin at $(0, y_i, 0)$, min_θ is the minimum thickness (dimension in the z direction) of the intersection of the cut plane and the artifact, and θ_{min} is a user-defined minimum thickness value.

To implement this assertion, we added code in the notebook bridge that analyzes the CAD model in Fusion 360. For now, it opens the model in section analysis mode (Figure 6.17) and asks the user to manually verify that the the minimum thickness is met. In the future, we aim to fully automate this process in the tool application. We added this assertion before fabricators mill

Figure 6.19: A Spoon Milled from Fir.

anything at all; if their model is too thin, the notebook will prompt them to edit it in CAD to conform to a minimum thickness requirement.

6.5.3 Plywood Bowl

Next, we milled a bowl out of a glued stack of plywood. The bowl's design features multiple layers of laminated wood in its profile (Figure 6.20). We chose the bowl as a larger artifact to mill than the previous two, too large, in fact, to mill on most conventional mills for metal. It measures 300mm wide, 250mm long and 38mm tall.

Accordingly, we faced challenges of scale and size. First, unlike one-sided milling, the maximum thickness of any millable artifact is limited by the maximum length below the collet of the shortest end mill. To prevent fabricators from realizing this too late during the milling process and then needing to start over, we implemented additional checks in the form of a coeffect function that asks for the maximum depth-below-holder values for all end mills to be used during the process. The coeffect function then checks the height of the model and warns the user if the model is too tall for the given end mills.

Second, the main workpiece must itself be fabricated in advance from pieces of plywood cut from larger stock; we provided additional functionality for generating the CAD and CAM to mill

Figure 6.20: A Bowl Milled from a Stack of Plywood. Because the bowl is ellipsoidal, it cannot be turned on a lathe and must be milled.

stackable pieces to create the main stock. Namely, for a desired artifact thickness θ_a and plywood thickness θ_p, we mill exactly n pieces from the larger stock, where

$$n = \left\lceil \frac{\theta_a}{\theta_p} \right\rceil,$$

which results in the minimal necessary thickness of the workpiece that must be wasted by milling down to artifact height θ_a

$$\theta_{wasted} = n\theta_p - \theta_a.$$

6.5.4 Printed Circuit Board

As a final example, we implemented two-sided milling of a different type of artifact: printed circuit boards (PCBs). We implemented a separate notebook program that works with a PCB layout in Fusion 360 standalone 3D models. Our notebook program takes a 3D model that Fusion 360 generates from the fabricator's PCB layout (Figure 6.21) as input and generates the following CAM operations to dispatch to the mill, in order: cutting traces on the top layer, drilling through-holes, cutting traces on the bottom layer, and cutting out the outline of the entire board.

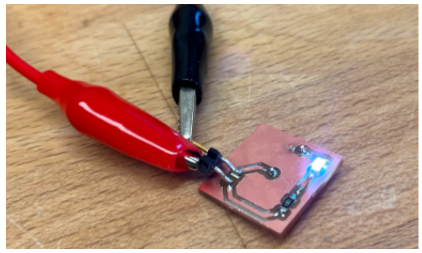


Figure 6.21: A Simple Two-Sided Printed Circuit Board Milled with Tandem Using a Modified Notebook.

For milling PCBs, we used the Clank open-source small-format mill [Rea23], which has a static bed and space above the tool carriage where we mounted a mini laser projector. We controlled the mill with a Duet2 control board [Dav23] connected directly to the fabricator's computer over a single-pair Ethernet connection. To communicate directly with the mill, we added additional functionality to the Tandem backend to forward HTTP requests sent from the notebook directly to the Duet2 control board.

6.6 Limitations and Future Work

6.6.1 Characterizing and Formalizing Material Behavior

Currently, Tandem uses standard techniques for accommodating material constraints. For example, if a fabricator uses Tandem to generate a setup for milling a step, Tandem sets the feed and speed values of the mill to an appropriate default for the given material, such as soft wood like fir, hard wood, or plywood. Thus, some material-specific assertions are possible; for example, a notebook could prompt the user to verify that they are milling wood and not foam for fine detailing work.

However, fabricators often need to experiment with settings like feed and speed depending on the actual results that they obtain with their mill given a certain material. Tandem does not yet support automatic adjustment of parameters based on the observed response of a workpiece being milled. To address this, in future work, we plan to include support for gathering empirical data about material response. For example, for milling a propeller, Tandem could support a fabricator like Florence in helping future fabricators like Siena select an appropriate feed rate for milling the ends of the propeller blades, which are prone to excess vibration. In this case, Tandem would generate several thin test cuts, each milled with a different feed rate. Tandem would prompt Siena to select a rate that does not result in breakage yet is not excessively slow. Her selection would then fulfill an assertion that Siena must select an appropriate feed rate for milling the propeller blades.

6.6.2 Automatically Sensing Manual Interventions

The need for material characterization raises another opportunity for future research: integrating sensing into Tandem programs. Currently, coeffect functions rely on manual verification that a fabricator enters into the notebook by means of user interface elements. This approach offers some benefits, including prompting fabricators to physically intervene and identify the state of a physical setup at a tactile level. Further, in most cases, measuring electrical and thermal properties of a material requires manual inspection with a multimeter or thermometer, respectively. Yet, manual verification can be slow, cumbersome, and difficult to characterize features such as electrical properties or heat.

An important future line of work, then, would be integrating automated sensing into Tandem programs. One important sensor would be a camera; though we initially explored camera-projector interactions for the AR overlay, we decided to focus solely on manual interventions for the scope of this paper. In future work, we plan to leverage camera support to automate some assertions, such as finding the position and geometry of a workpiece on the machine bed. Other techniques like speckle sensing [DACS+21] or motor torque monitoring [LHR+21] could aid novices in setting reasonable initial feed and speed values like in the example above. Sensing would also let fabricators program workflow-specific versions of runtime monitoring, e.g., pausing a CNC mill if a workpiece deviates from its fixturing. We envision a larger body of work that explores implementing closed-loop control for fabrication (e.g., as explored by Piovarči et al. [PFX+22]) in Tandem.

6.6.3 Generalization to Non-Milling Fabrication Techniques

Though we designed and implemented Tandem as a platform for any experimental digital fabrication workflow, this paper focuses only on two-sided CNC milling, a subtractive manufacturing workflow. The single-workflow focus let us describe Tandem's functionality in all the complexity that this single workflow entails. However, other experimental workflows navigate different physical-digital mismatches than those faced here. Nonetheless, this limitation also highlights a strength of Tandem: fabricators can program their own workflows and tests that are specific to additive manufacturing and laser cutting.

As an example, we briefly discuss how Tandem could help a fabricator like Florence implement an experimental 3D printing technique, Wireprint, as a notebook [MIG+14]. Wireprint is a technique for 3D printing a wireframe of an object, rather than an entire solid body; the approach is difficult to replicate and has not yet seen widespread adoption[4]. The approach involves several steps: designing a geometry in CAD that is feasible with the approach, generating wireframe triangles from the geometry, and adjusting the printer's feed, extruder, and pause parameters.

Using Tandem, Florence could write effect functions that bring the CAD model into the notebook as an STL before passing it into the toolpath generation algorithm. She would then write code, possibly using graphical elements from coeffect functions, that help fabricators like Siena

[4] While some forms of the Cura slicer have a wireframe option, the algorithm differs from that proposed by Mueller et al. [MIG+14] and is difficult to tune within Cura alone.

adjust feed, extrusion, and pauses at different points in the toolpath. Florence would design the notebook to progressively print more complex parts of a wireframe, starting with a single triangle, moving to a surface of triangle, and finally to concentric slices of the model. Florence would use assertions to ensure that Siena debugs and fabricates simpler constructions before moving on to more intractable ones. Florence could also use assertions to analyze the model's feasibility for the Wireprint approach; for example, an assertion could reject geometries with overhangs or non-concentric parts and ask the fabricator to redesign those parts.

Finally, while AR functionality would likely not be useful for the Wireprint workflow, it could be useful for novel 3D printing workflows that involve precisely printing on existing geometries [KZT+18, CKM+16].

6.6.4 Notebooks versus External tools

The benefit of the notebook is that Florence can lay out her steps sequentially so that Siena can follow along in one environment without the additional cognitive load of managing many applications at once. Though the notebook uses Tandem's API to read and write to Fusion 360, the actual state of the CAD model and CAM setups still lies outside of the notebook. Siena is free to make adjustments to CAD and CAM in Fusion 360 itself, independently of the notebook's instructions. Still, we could imagine a future where CAD and CAM functionality were part of the notebook itself; for example, a fabricator could simply call a function to open a CAM environment within the notebook. Some tools provide basic CAD functionality for web environments [Ons22] and even for Jupyter [KRKP+16] computational notebooks [Sch23, Jup23]. Unfortunately, two-sided milling called for more mature CAD and CAM functionality than these tools offer. Ideally, future versions of Fusion 360 could support embedded CAD and CAM windows inside notebook environments. Alongside Fusion 360, future fabricators could expand Tandem's backend to interface with 3D printing-specific CAM, such as Cura, Formlabs PreForm, or Slic3r, all of which have respective APIs [Ult21, For23, Ale23].

6.6.5 Encouraging Practice and Debugging Skills

The goal of Tandem is to perform all the major steps of a digital fabrication workflow from a single program and check that prerequisites are satisfied during the workflow. It thus complements, not replaces, the need for craft-specific knowledge and hands-on tool education. The notebook scaffolds Siena's understanding of *why* she must do something and how the workflow works and how to avoid errors. Future iterations of fabrication-as-reproducible notebooks could include extra steps for encouraging debugging [LTY+16] and practice [LJDM23].

6.7 Conclusion

Through sustained attention to an example experimental workflow—two-sided CNC milling—we have shown why reproducing workflows is unexpectedly complicated yet crucial for research and practice. Through Tandem, we addressed this issue by making sharing experimental workflows more like sharing code. Tandem breaks the different parts that must occur into programmable components: changes to the design of 3D models, changes to the movements of machines, control of machines themselves, and the setup of the physical environment as necessitated by decisions in software. Tandem programs explicitly include unspoken preconditions that allow future fabricators to safely carry out and modify workflows, rather than leave this knowledge as informal lore passed down only in machine shops and makerspaces.

Overall, Tandem challenges an unsustainable status quo of a workflow as an abstract concept spread across multiple pieces of software and physical assumptions. Instead, we have provided the first steps towards computational "glue" for concrete implementations of fabrication research.

Through programming, Tandem unites disparate requirements for experimental fabrication in HCI and encourages a holistic, reproducible notation of physical-digital labor.

Chapter 7

Discussion

"The representation of the craftsman is drowned in concreteness, engaged in material manipulation and sensible existence; it is dominated by its object; the representation of the engineer is one of domination; it turns the object into a bundle of measured relations, a product, a set of characteristics. … The prime condition for the incorporation of technical objects would thus be for man to be … capable of approaching and getting to know them through entertaining a relation of equality with them, that is, a reciprocity of exchanges; a social relation of sorts."

— Gilbert Simondon, *On the Mode of Existence of Technical Objects* [Sim16, p. 105]

In the preceding quote, philosopher Gilbert Simondon discusses two different perspectives on technical objects, defined here to be devices and systems that are designed and composed of several parts. He sees the "representation of the craftsman" as a mastery someone develops from using a technical object repeatedly, often beginning when young. This resembles what Michael Polanyi calls *tacit knowledge*: knowledge that is systematic, embodied through practice and difficult to articulate with words. In contrast, the "representation of the engineer" is far more familiar to computer scientists; in this way of understanding, a person breaks a problem into parts that can be translated into formal notation, which can range from numbers to musical notes to circuit diagrams to source code.

Simondon claims that relying too much on only one of these perspectives can cause us to develop an unhealthy relationship with technology. We might see a technical object as something "non-technical," where someone has to "just get it" to understand how the object works. Or, we might see a technical object as completely governed by preconceived equations and plans. In the latter case, there can be little to no room for practical realities and complications that disagree with elegant charts or diagrams, and less room for what Donald Schön calls *reflection-in-action*: roughly, thinking on one's feet and using one's senses in the moment while problem-solving [Sch83].

As I reflect on the research presented thus far, the reader might ask why this tension of perspectives matters for physical-digital programming. In short, I would answer that physical-digital programming is a concrete way to negotiate the representations of both the craftsperson and the engineer as conceived by Simondon. In Chapter 1 and throughout this work, I discussed how fabricators who develop novel fabrication workflows must work across physical contingencies and the constraints of code simultaneously. Using Simondon's perspective, experimental fabricators must think like both craftspeople and engineers. They need to address the practicality of machines and materials while also representing this knowledge through source code. Conversely, they also need to rely on affordances unique to source code, such as precision, repetition, and testing, as they negotiate changes to the physical artifact.

Physical-digital programming guides fabricators towards a *technical mentality* in workflow development, or having "direct knowledge of a technical system" [BP24], which in our case is a workflow, i.e., (1) a composition of interchangeable parts, (2) whose composition is shaped by the feasibility of the entire workflow. Through reasoning, experimentation, and reproducibility of interdependent physical and digital steps in source code, fabricators and eventually society at large will have a more direct, non-threatening understanding of novel fabrication workflows. At one extreme, we live in a throwaway society, where technical objects are easier than ever to purchase, use, and discard without knowledge of their inner workings. At the other, centralization of technological power, particularly massive-scale, proprietary generative AI, is causing people to default to the opposite extreme of anthropomorphizing and fearing complex technical objects. This extreme, in the words of Simondon, "supposes that these objects are also robots and that they represent a permanent danger of aggression and insurrection towards him ... in the belief that reduction to slavery[1] is a sure way to prevent any rebellion" [Sim16, p. 17].

7.1 Quality Control Can Foster Technical Mentality

I argue that the deeper understanding that physical-digital programming can help foster extends beyond digital fabrication and comes at a crucial moment for society where, despite the odds, we have a precious chance to rethink our means of production and our way of thinking about technical objects as a whole. To better understand how we as a society might occupy a middle ground—develop a technical mentality—I draw our attention to *quality control*. Quality control extends the facets of physical-digital programming covered so far, namely, reasoning, experimentation, and reproducibility. It forces us to see fabrication artifacts as neither throwaway, without material quirks and completely characterized by equations, nor as having a "life of their own," unable to be reworked, rethought, and iterated upon.

Quality control is the way in which fabricators ensure that their workflow produces an artifact that meets a fabricator-defined specification. Different fabricators will define varying specifications

[1] As mentioned in Chapter 1, Simondon does not use slavery as a metaphor. He traces the origins of the hylomorphic schema, i.e. thinking of a brick as mute matter placed in a mold, to ancient Athens' reliance on a slave class to support the *polis*. In this class structure, slaves who worked with the physical constraints of the clay were not allowed to influence the manufacturing processes of bricks altogether [BP24].

based on their own use cases. In conventional fabrication practices such as CNC machining with metal, an established example of a specification is size tolerance. For example, a precision hole drilled into a steel workpiece will have a radius that is within a given tolerance of a desired value, say ±0.01″ versus ±0.001″; these values indicate the maximum allowed deviation from a nominal value. For most applications, a tolerance of ±0.01″ from a nominal radius would not pose a problem, and manufacturing to such a tolerance is usually achievable with CNC milling alone. However, other applications might require a tight fit between two parts, requiring a tolerance of ±0.001″. In this case, the workflow to machine this part would involve more steps and specialized equipment, increasing the overall cost of manufacturing [tar21].

In both conventional and experimental workflows, a fabricator must think critically about their own requirements and design a quality control specification accordingly. With conventional fabrication, quality control for dimensional precision is highly standardized and forms part of a machinist's training. Systems such as geometric dimensioning and tolerancing are codified in the design of engineering drawings and specify how fabricators must measure parts post-fabrication to ensure adherence to a specification [tar21, Ame19].

In contrast, experimental fabrication currently has no analogous method of quality control, even though the consequences of fabricating outside a specification can be equally severe. For example, for the case studies presented at the beginning of Chapter 1, the materials engineer 3D printing batteries may need to ensure a minimum volume for interlocking electrodes, the ceramicist may need a constant flow of clay into the printer nozzle within an allowable tolerance, and the lattice research team may need to establish consistency checks between different 3D printed lattices for their experiments.

For quality control in both conventional and experimental fabrication, fabricators must:

1. Write down specifications in language, whether in prose, a graphical language for conven-

tional fabrication, or a programming language,

2. Adjust their workflow to possibly meet the specification, and
3. Measure the artifact post-fabrication to ensure that the artifact actually does meet the specification.

Given this definition of quality control, what is its relationship with technical mentality? When a fabricator approaches workflow development with quality control in mind, they are not overly biased towards either the representation of the craftsman or the engineer. For those who prefer to fabricate objects using only tacit knowledge and gut instinct, quality control forces them to reflect on their process, isolate any steps that might violate a specification, and reformulate the process accordingly. For those with an engineering mindset—especially computer scientists who are researching digital fabrication—quality control forces them to dispense with idealistic, preconceived notions of a workflow and instead grapple with the practicalities of machine and material.

7.2 An Abstraction Stack for Experimental Fabrication

Given the conceptual importance of quality control, we can now synthesize the lessons learned from the four projects presented in this dissertation from a new perspective. As discussed in Chapter 2, much prior work has has adopted either a top-down, simulation-focused approach, or a bottom-up lens of personal craft and expertise. In contrast, quality control challenges experimental fabricators to discover constraints and requirements that arise at the high-level (e.g., artifact design requirements), the low-level (e.g., optimizing tool trajectories), or anywhere in between. With digital-only programming, programmers can formulate requirements that handle potentially disparate concerns; for example, the cause of a non-updating user interface could range from

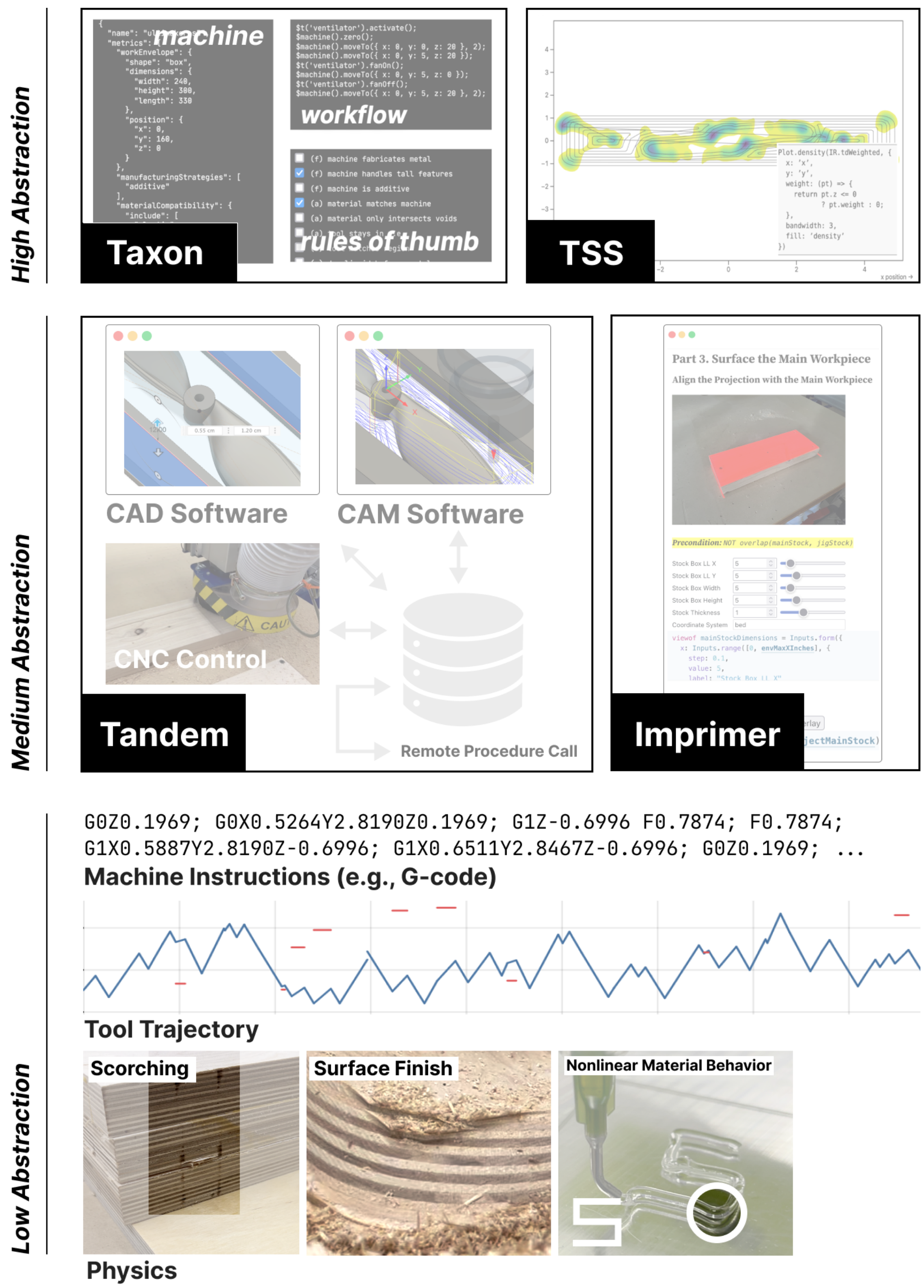


Figure 7.1: Physical-Digital Programming Provides a *Ladder of Abstraction* [Vic11] for Experimental Physical Automation. The four projects (shown with black labels) organize formerly disparate concerns, building from low-level machine control and material behavior, up to workflows-as-programs with an API for communicating with existing software, and finally up to high-level, user-defined static analyses and visualizations for highly specific automation tasks.

buggy client-side code, a database issue, a problem with the network, or many other sources. At a broad level, programmers manage the complexity of these issues using *abstraction*: organizing concerns into discrete layers which have controlled interfaces between them.

Ultimately, the four projects comprising physical-digital programming thus far provide the beginnings of an abstraction stack for experimental physical automation. Figure 7.1 demonstrates such a *ladder of abstraction* [Vic11] which helps experimental fabricators methodically move between layers of abstraction to address issues that arise during workflow development. At the highest layers, Taxon provides static analyses (rules of thumb) that enforce fabricator-defined checks. Similarly, Toolpath Stylesheets provide task-specific visualizations using fabricator-defined transforms and visual renderings of a given toolpath. For a fabricator to actually create a novel toolpath, they descend one abstraction layer and use Imprimer to compose a notebook program that generates a toolpath. In the notebook, they use Tandem's API to interface with existing CAD and CAM software, directly control machines, and generate AR instructions for manual interventions. Going even further, how would a fabricator ensure that their workflow-as-program correctly produces the desired artifact? They would drop down one or more abstraction layers and investigate machine instructions, tool trajectories, and material behavior. Using manual inspection or data gathered from sensors, they propagate information back up the stack, for example, by updating assertion predicates in Tandem, or by modifying a given TSS to more accurately visualize material behavior.

Given the plethora of existing tools for digital fabrication both in conventional usage and in computer science research, what value does a physical-digital abstraction stack provide? Ultimately, physical-digital programming helps fabricators methodically expand the frontier of possible workflows into novel applications spaces. Rather than simply add one-off fabrication toolchains that contribute a single new workflow, physical-digital programming lets fabricators leverage functionality built by others at all abstraction levels.

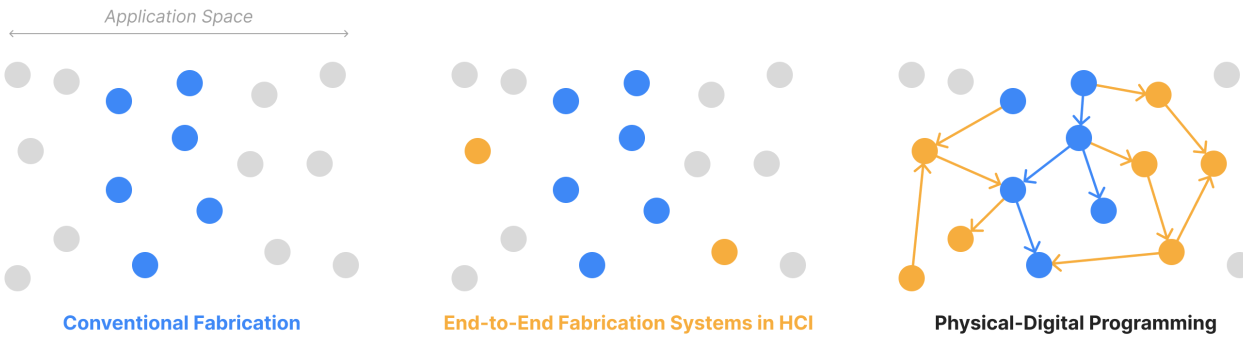


Figure 7.2: Physical-Digital Programming versus One-Off End-to-End Systems. Vertices are steps of a workflow and edges are interoperation between steps. Conventional fabrication tools provide well-established functionality in a narrow application space. Novel end-to-end fabrication systems typically add black box functionality in a single new application area. In contrast, physical-digital programming modularizes both existing and novel functionalities which connect to each other via programming language constructs.

Figure 7.2 illustrates this idea by representing the total space of experimental fabrication as a conceptual graph. Vertices represent individual steps of a workflow and edges represent interoperation between steps. The vertical and horizontal axes roughly map to the spectrum of domains in which experimental fabricators innovate. In the case of conventional fabrication, the frontier of possible workflows, marked by blue vertices, is well-established. At the same time, the application space is narrow, and movements between steps typically require manual export and import of data. In contrast, end-to-end digital fabrication tools—commonly presented as contributions in human-computer interaction research (see Section 2.2)—let fabricators execute workflows in new application spaces. However, such black box systems typically form their own discrete vertices, disjoint from existing tools and difficult to modify.

To provide the best of both approaches, physical-digital programming instead modularizes steps of a workflow as code, either by writing functionality from scratch or wrapping existing functionality in an API as done in Tandem. Together, Imprimer and Tandem let fabricators build workflows by constructing paths from high to low levels of abstraction. Such paths can visit steps that are both conventional (blue) or experimental (gold), and Imprimer and Tandem provide automated interoperation between both kinds of steps. Finally, Taxon and Toolpath Stylesheets help fabricators enforce quality control, that is, finding the best path for their use case, rather than being constrained to a preexisting one.

Extending further from this concept, each facet of physical-digital programming provides a structured way to build new abstractions over machine functionality. Programming, at its core, involves creating abstractions over the hardware of a computing machine. While certain abstractions over hardware, such as operating systems, provide foundational functionality, programmers will always need to create new abstractions as new challenges arise. Similarly, in digital fabrication, experimental fabricators rely on conventional fabrication techniques, but must also build new abstractions over fabrication machine hardware. Often, these abstractions encapsulate functionality that is not at all supported with conventional fabrication software.

To this end, Figure 7.3 shows how each facet of physical-digital programming facilitates full-fledged programming: the structured development and testing of case-specific abstractions over machine hardware. Figure 7.3 illustrates how a given workflow maps to a given subset of the total space of machine functionality, represented as a rectangle at the bottom. The space includes, as subspaces, conventional (blue), novel (gold), invalid (red), and/or unexplored (white) functionalities. Experimentation (Imprimer) empowers fabricators to build meaningful abstractions to achieve novel machine functionality, while reproducibility lets fabricators incorporate functionality from more stable conventional fabrication tools. An aim of reproducibility (Tandem) is to enable the same machine functionality across varying software and hardware setups. To this end, Tandem lets fabricators reproduce the same functionality by enforcing equivalence between workflows with different graph topologies. Finally, formal reasoning helps fabricators avoid errors states, which are represented as invalid subspaces of the total space, using both textual (Taxon) or visual (Toolpath Stylesheets) analyses.

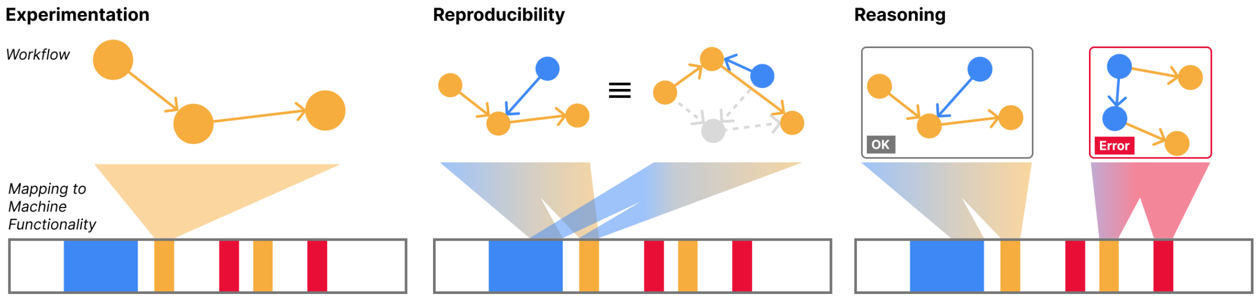


Figure 7.3: Physical-Digital Programming Structures Composition of New Abstractions Over Machine Functionality. Experimentation means that fabricators can adjust their workflow (gold) by adjusting the path between possible steps; different paths map to different subspaces of the total space of machine functionality (bottom). Reproducibility lets fabricators incorporate tools from conventional fabrication (blue) as well as establish equivalent workflows if a given step is unavailable (gray). Reasoning means that fabricators can check their workflow for potential error states (red), defined as invalid machine functionality for a given specification.

7.3 Three Directions for Future Work

Given both the need for quality control and an abstraction stack that engenders it, I discuss three avenues for future work: runtime monitoring and data collection; AI assistance for maintenance and repair; and building open-source fabrication communities.

7.3.1 Runtime Monitoring and Data-Driven Debugging

One notable omission from the projects presented so far is *runtime monitoring*, making sure certain conditions are satisfied when a digital fabrication machine is operating. Many off-the-shelf software tools now offer rudimentary runtime monitoring functionalities, e.g., using a camera to check for poor adhesion of the first layer of a 3D print to the 3D printer bed [Bam24]. However, similar to how toolpath stylesheets allow task-specific visualizations beyond the common case, I postulate language constructs for runtime monitoring as a core feature.

For example, a CNC milling fabricator could initialize a series of sensors, perhaps two cameras with different angles of the work envelope, an accelerometer measuring vibrations of the tool, and a microphone recording data. This suite of sensors could be represented as *streams* in a programming language, where each stream would have a dedicated API. A fabricator could define anomalous conditions in language, for example, by writing code that processes a video feed from a camera to detect a large deviation from an expected form as gauged by computer vision metrics. In this case, a Tandem program could detect and pause the machine if part of the workpiece were torn-out or if the workpiece slips out from its fixturing. For more robust detection, anomalies could be described as a combination of thresholds over several sensor streams, e.g., a sudden noise above *n* decibels accompanied by deviations in the camera feed and accelerometer frequency spectrum.

Determining what would be considered a deviation from typical behavior would require the assembly of a dataset of "successful" machine operations over a variety of machines, materials, and fabrication tasks. Building this dataset would need the cooperation from a community of fabricators who could contribute data from running their own machines. Fabricators would also need to label the data recorded from their sensor suites. Again, designing dedicated constructs in a programming environment could encourage this sharing of data. For example, I could imagine visualizing high-level runtime data from sensors in a Tandem notebook, labeling certain parts of the runtime that were anomalous, and then submitting the data from the notebook to a collective dataset so that other fabricators across the world could benefit.

Why go through the process of curating a dataset in the first place? Though I could extol the value of bringing machine learning techniques to runtime monitoring, I instead shift the focus to the description in language of proper machine behavior. Ultimately, I aim to enable fabricators to describe *specifications* of proper machine runtime rather than merely tweak their workflow to get it to work. An analog in digital-only programming would be how a programmer could not only debug their program, but also add to their programs' function *types* that describe the "shape" of inputs and outputs. The benefit of this is that others can more easily understand and modularize a program, becoming better acquainted with system subassemblies, which empowers programmers to modify and repair parts of a program.

7.3.2 Machine Maintenance and Repair

As we have seen, breakdowns in a fabrication workflow are a pervasive reality that we cannot fully model beforehand. While a more descriptive programming language or type system, or a prior dataset of machine runtime data, can help experimental fabricators recognize errors, it takes additional work to help fabricators recover from them. Because digital fabrication involves physical hardware, knowing the state or condition of the machine—whether it might have a loose timing belt, old 3D printing filament, or a misaligned axis—is crucial to achieve quality artifacts.

This calls into question notions of care, maintenance, and repair, all concepts that prior research in HCI and other fields has studied, but that software tools for digital fabrication have scarcely addressed. Prior work has studied technology repair shops in the Global North and South, observing that "values in repair (and indeed other HCI contexts) are better conceived as processes rather than things: in effect, verbs rather than nouns [HJR+16]." I point out that lines of code in Imprimer and Tandem *are* verbs; they ultimately control a machine to modify the physical world, sense and interpret the current state of the world, or prompt a fabricator to inspect and intervene.

Beyond descriptive studies of repair sites, I draw inspiration for future work from prior literature that examines care, maintenance, and repair in the context of specific technical objects. Other work has examined repair and maintenance in the context of FDM 3D printing, where researchers documented repair practices of the online community r/FixMyPrint [SP23]; the researchers found that fabricators formalized and developed discrete routines that localized and allowed repair for certain error-prone machine states, e.g., tracing 3D print layer shifting back to loose timing belts and under-extruded parts back to a clogged nozzle. As a more conceptual example, Lu and Lopes built a watch that relied on a healthy colony of slime mold (*Physarum polycephalum*) inside the watch to function correctly [LL22]. If the watch user did not feed and water the slime mold regularly, the watch would cease working. Participants in the study reported affective responses to changes in the watch's state: rather than view the device's functionality as guaranteed, they became connected to the contingent nature of the slime mold components. With both the online 3D printing community and the physarum watch, the fragility of a device's subassemblies became front and center.

Extending from the notion of repair as a verb, the next step in computational notebooks for fabrication is developing more language constructs that facilitate calibration. Here, I draw from the formalized maintenance routines described by Subbaraman and Peek [SP23]. For example, one physical-digital assertion could require a CNC mill to cut some reference shape within a margin of allowable error; a camera or other sensor would register the calibration cut and compare it with a ground truth. If the calibration cut deviated too much from a reference, the notebook would recommend and guide repair steps before trying the calibration cut again. By encoding maintenance and repair steps in code, we place these values—and by extension, deeper care for the workings of a machine—on equal footing with code that computes and creates.

As a proof-of-concept, Appendix A describes an AI-assisted prototype for debugging machine breakdowns.

7.3.3 Open-Source Fabrication Communities

The last major limitation of the work discussed in this dissertation is the connection of physical-digital programming tools to communities of experimental fabricators. As I claim in Chapter 1, open-source software communities have grown over the last few decades from being specialized venues to powering technology in all types of applications and in all corners of the earth. Through shared understanding of a programming language, contributors separated by great distances can work together to build crucial infrastructure.

While physical-digital programming could remove barriers to collaboration on fabrication workflow development through the same affordances that programming languages give to open-source software contributors, programming languages are not the only ingredient for success in open-source communities. Indeed, there is a surprising lack of research on what aspects of programming languages help open-source communities thrive, e.g., how different programming language designs facilitate (or hinder) collaboration in different communities. To enable a future of open-source fabrication, more studies on how small groups of fabricators program and iterate together will be necessary.

The bulk of research on open-source communities has been through an economic [Ben06] or anthropological [Egh16, Egh20] lens, focusing on how people interact with one another. If and when future researchers study emerging communities of open-source fabrication, I urge them to analyze using a technical mentality: what machines, what material choices, and what programming constructs make fabrication workflow development *work* at all? Understanding how emerging communities of open-source fabrication encode fragile conditions production into language is a first step towards a society that can more mindfully speak about machines as neither throwaway nor threatening, but rather a crucial part of our humanity.

Chapter 8

Conclusion

"The most powerful cause of alienation in the contemporary world resides in this misunderstanding of the machine, which is not an alienation caused by the machine, but by the non-knowledge of its nature and its essence..."

— Gilbert Simondon, *On the Mode of Existence of Technical Objects* [Sim16, p. 16]

In this dissertation, I focused on practitioners who are developing experimental applications in science, art, and engineering. Such experimental workflow development demands that fabricators reason fluidly and simultaneously across physical and digital constraints. Unlike conventional applications of digital fabrication technology, experimental fabricators must constantly test, diagnose, and iterate upon their very own means of production. Finally, while experimental fabricators may have common goals and approaches, such practitioners in disparate domains nonetheless remain isolated and relatively ill-equipped to borrow, remix, and share workflows from and with others. To this end, Chapter 1 explored how code, despite forming the core of digital fabrication technology, currently lends scant expressive power to experimental workflow development. Instead, fabrication software tools reinforce a batch programming paradigm from the middle of the 20th century and impose undue limits on creative exploration and problem-solving.

This thesis has turned this limitation on its head. I have argued that programming languages, if they appropriately capture the physical-world constraints that guide workflow development, could instead help experimental fabricators flourish and form stronger communities of practice across domains. Chapter 2 discussed the people for whom physical-digital programming could be most transformative, identified gaps in the literature that physical-digital programming could address, and visited existing systems from which I drew inspiration.

The subsequent four chapters addressed three facets crucial to the design of physical-digital computing: reasoning, experimentation, and reproducibility. For reasoning, Chapter 3 presented Taxon, a system for formalizing in language unspoken rules of thumb that guide machine choice and usage; Chapter 4 presented Toolpath Stylesheets, which let fabricators express highly specific aspects of machine behavior as visualizations of a toolpath. For experimentation, Chapter 5 presented Imprimer, which extends computational notebook programming environments to support the development of experimental workflows as single programs. For reproducibility, Chapter 6 introduced Tandem, a library and accompanying backend that augments Imprimer notebooks to make workflows more accessible to others; this involved integrating external CAD and CAM tools, an augmented-reality guided interface for manual steps, and basic error-checking into the notebook.

Though the development of physical-digital programming has thus far targeted experimental fabricators only, its implications affect computer science as a discipline and, more generally, society's relationship to technology as a whole. Chapter 7 raised the notion of computational quality control: ensuring that the state of physical machines, materials, and artifacts meets expectations, and how intervene if not. Physical-digital programming forces fabricators and digital-only programmers to reckon with the precarious nature of the physical world—itself the foundation of all data. This approach to programming places questions of care, maintenance, and repair of physical-digital technologies front and center, rather than banishing them outside the programming environment milieu.

Such centering of physical contingency provides a crucial foil to today's state of computing, which is increasingly dominated by massive-scale modeling that is most accessible to those with power and that consumes data from "elsewhere" [SHRB20]. Borrowing from philosopher Gilbert Simondon's technical mentality, this undue fixation on massive-scale modeling exacerbates a schism in society's view of technical objects: they become either throwaway or anthropomorphized, all-knowing entities.

The farthest-reaching goal of physical-digital computing is to dismantle this unsustainable form of thinking and help experimental fabricators—and society at large—reestablish a healthier relationship with technology through direct engagement with a machine's inner and outer workings. Physical-digital computing reveals the contingent nature of physical reality within digital programming environments, encouraging us to not devolve the making of data or artifacts to others, but rather to accept personal responsibility for the development of physical and digital artifacts.

Ultimately, it is through language itself that we might transcend today's dominant computational paradigm, where no thing is precious, to the computing of tomorrow, where all things are precious.

Bibliography

[AHYD20] Lea Albaugh, Scott E. Hudson, Lining Yao, and Laura Devendorf. Investigating Underdetermination Through Interactive Computational Handweaving. In *Proceedings of the 2020 ACM Designing Interactive Systems Conference*, DIS '20, pages 1033–1046, New York, NY, USA, July 2020. Association for Computing Machinery.

[Ale23] Alessandro Ranellucci. Slic3r - Open source 3D printing toolbox, 2023.

[Ame19] American Society of Mechanical Engineers. *Dimensioning and tolerancing: engineering product definition and related documentation practices.* American Society of Mechanical Engineers, New York, 2019. OCLC: 1091588533.

[And14] Chris Anderson. *Makers: The New Industrial Revolution.* Crown Currency, no-value edition edition, April 2014.

[Ans73] F. J. Anscombe. Graphs in Statistical Analysis. *The American Statistician*, 27(1):17–21, February 1973. Publisher: Taylor & Francis _eprint: https://www.tandfonline.com/doi/pdf/10.1080/00031305.1973.10478966.

[ard05] arduino.cc. Arduino Language Reference, 2005.

[ASS02] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs.* Electrical engineering and computer science series. MIT Press [u.a.], Cambridge, Mass., 2. ed., 7. [pr.] edition, 2002.

[AT10] Vaishnavi Ananthanarayanan and William Thies. Biocoder: A programming language for standardizing and automating biology protocols. *Journal of Biological Engineering*, 4(1):13, November 2010.

[Aut22] Autodesk. Fusion 360, 2022.

[AWDM19] Kristina Andersen, Ron Wakkary, Laura Devendorf, and Alex McLean. Digital crafts-machine-ship: creative collaborations with machines. *interactions*, 27(1):30–35, December 2019.

[BAH+20] Stuart A. Bagley, Jonathan A. Atkinson, Henry Hunt, Michael H. Wilson, Tony P. Pridmore, and Darren M. Wells. Low-cost automated vectors and modular environmental sensors for plant phenotyping. *Sensors (Switzerland)*, 20(11):1–15, 2020.

[Bam24] Bambu Lab Contributors. Spaghetti Detection, January 2024.

[BBKE13] BJ Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. *UIST*, 2013.

[BBPJ21] Gabrielle Benabdallah, Sam Bourgault, Nadya Peek, and Jennifer Jacobs. Remote Learners, Home Makers: How Digital Fabrication Was Taught Online During a Pandemic. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–14, New York, NY, USA, May 2021. Association for Computing Machinery.

[BCH+22] Alex Bäuerle, Ángel Alexander Cabrera, Fred Hohman, Megan Maher, David Koski, Xavier Suau, Titus Barik, and Dominik Moritz. Symphony: Composing Interactive Interfaces for Machine Learning. In *CHI Conference on Human Factors in Computing Systems*, CHI '22, pages 1–14, New York, NY, USA, April 2022. Association for Computing Machinery.

[BCP+03] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 93–103, May 2003. ISSN: 0270-5257.

[BD09] Ana Bove and Peter Dybjer. Dependent Types at Work. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, Lecture Notes in Computer Science, pages 57–99. Springer, Berlin, Heidelberg, 2009.

[BEGK11] Dimosthenis E. Bolanakis, Georgios A. Evangelakis, Euripidis Glavas, and Konstantinos T. Kotsis. A teaching approach for bridging the gap between low-level and high-level programming using assembly language learning for small microcontrollers. *Computer Applications in Engineering Education*, 19(3):525–537, 2011. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.20333.

[Ben06] Yochai Benkler. *The wealth of networks: how social production transforms markets and freedom.* Yale University Press, New Haven London, 2006.

[BHA+22] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma

Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the Opportunities and Risks of Foundation Models, July 2022. arXiv:2108.07258 [cs].

[BM17] Patrick Baudisch and Stefanie Mueller. Personal Fabrication. *Foundations and Trends® in Human–Computer Interaction*, 10(3-4):165–293, 2017.

[BOH11] M. Bostock, V. Ogievetsky, and J. Heer. D³ Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[Bos18] Mike Bostock. A Better Way to Code, February 2018.

[BP24] Gabrielle Benabdallah and Nadya Peek. Technical Mentality: Principles for HCI Research and Practice. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, pages 1–15, New York, NY, USA, May 2024. Association for Computing Machinery.

[BR18] Fabien C. Y. Benureau and Nicolas P. Rougier. Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *Frontiers in Neuroinformatics*, 11, 2018.

[Bra23] Bray Lab Contributors. Flip-Cut (2 Sided Milling) Instructions – Bray Lab, 2023.

[BSH+19] Gytis Bernotas, Livia C.T. Scorza, Mark F. Hansen, Ian J. Hales, Karen J. Halliday, Lyndon N. Smith, Melvyn L. Smith, and Alistair J. McCormick. A photometric stereo-based 3D imaging system using computer vision and deep learning for tracking plant growth. *GigaScience*, 8(5):1–15, 2019.

[Bø06] Susanne Bødker. When second wave HCI meets third wave challenges. In *Proceedings of the 4th Nordic conference on Human-computer interaction: changing roles*, pages 1–8, Oslo Norway, October 2006. ACM.

[Cab14] Ricardo Cabello. three.js - Javascript 3D library, 2014.

[Car20] L. Cardelli. Kaemika app, Integrating protocols and chemical simulation. *undefined*, 2020.

[CGDE20] Emma J Chory, Dana W Gretton, Erika A DeBenedictis, and Kevin M Esvelt. Flexible open-source automation for robotic bioengineering. *bioRxiv*, page 2020.04.14.041368, 2020.

[CGS21] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. PL and HCI: better together. *Commun. ACM*, 64(8):98–106, July 2021.

[CH18] Matthew Conlen and Jeffrey Heer. Idyll: A Markup Language for Authoring and Publishing Interactive Articles on the Web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 977–989, New York, NY, USA, 2018. ACM.

[Cir24] CircuitPython Community. CircuitPython, 2024.

[CKM+16] Xiang 'Anthony' Chen, Jeeeun Kim, Jennifer Mankoff, Tovi Grossman, Stelian Coros, and Scott E. Hudson. Reprise: A Design Tool for Specifying, Generating, and Customizing 3D Printable Adaptations on Everyday Objects. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 29–39, Tokyo Japan, October 2016. ACM.

[CPG+15] David Correa, Athina Papadopoulou, Christophe Guberan, Nynika Jhaveri, Steffen Reichert, Achim Menges, and Skylar Tibbits. 3D-Printed Wood: Programming Hygroscopic Material Transformations. *3D Printing and Additive Manufacturing*, 2(3):106–116, September 2015.

[CPH+20] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, New York, NY, USA, April 2020. Association for Computing Machinery.

[CPT+22] Ishan Chatterjee, Tadeusz Pforte, Aspen Tng, Farshid Salemi Parizi, Chaoran Chen, and Shwetak Patel. ARDW: An Augmented Reality Workbench for Printed Circuit Board Debugging. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, UIST '22, pages 1–16, New York, NY, USA, October 2022. Association for Computing Machinery.

[Cru92] S. Scott Crump. Apparatus and method for creating three-dimensional objects, June 1992.

[Csu68] Charles A. Csuri. Numeric Milling, 1968.

[CVTH21] Matthew Conlen, M. Vo, Alan Tan, and Jeffrey Heer. Idyll Studio: A Structured Editor for Authoring Interactive & Data-Driven Articles. *UIST*, 2021.

[DACS+21] Mustafa Doga Dogan, Steven Vidal Acevedo Colon, Varnika Sinha, Kaan Akşit, and Stefanie Mueller. SensiCut: Material-Aware Laser Cutting Using Speckle Sensing and Deep Learning. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, pages 24–38, New York, NY, USA, October 2021. Association for Computing Machinery.

[Dav23] David Crocker. Duet 3D, 2023.

[DBG+20] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, Honolulu HI USA, April 2020. ACM.

[DDKMR16] Laura Devendorf, Abigail De Kosnik, Kate Mattingly, and Kimiko Ryokai. Probing the Potential of Post-Anthropocentric 3D Printing. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, DIS '16, pages 170–181, New York, NY, USA, 2016. ACM.

[DeL21] Robert A DeLine. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–11, New York, NY, USA, May 2021. Association for Computing Machinery.

[DKM+21] Grigoris Daskalogrigorakis, Salva Kirakosian, Angelos Marinakis, Vaggelis Nikolidakis, Ioanna Pateraki, Aristomenis Antoniadis, and Katerina Mania. G-Code Machina: A Serious Game for G-code and CNC Machine Operation Training. In *2021 IEEE Global Engineering Education Conference (EDUCON)*, pages 1434–1442, Vienna, Austria, April 2021. IEEE.

[DSTM17] Andrei Dobrescu, Livia C.T. Scorza, Sotirios A. Tsaftaris, and Alistair J. McCormick. A "Do-It-Yourself" phenotyping system: Measuring growth and morphology throughout the diel cycle in rosette shaped plants. *Plant Methods*, 13(1):1–12, 2017.

[DT19] Audrey Desjardins and Timea Tihanyi. ListeningCups: A Case of Data Tactility and Data Stories. In *Proceedings of the 2019 on Designing Interactive Systems Conference*, DIS '19, pages 147–160, New York, NY, USA, June 2019. Association for Computing Machinery.

[Egh16] Nadia Eghbal. Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure. Technical report, Ford Foundation, 2016.

[Egh20] Nadia Eghbal. *Working in Public: The Making and Maintenance of Open Source Software*. Stripe Press, San Francisco, August 2020.

[Emi21] Emily Peyton. How to Communicate Between CNC Machinery and a Computer, December 2021.

[EPF+17] Eythor R. Eiriksson, David B. Pedersen, Jeppe R. Frisvad, Linda Skovmand, Valentin Heun, Pattie Maes, and Henrik Aanæs. Augmented Reality Interfaces for Additive Manufacturing. In Puneet Sharma and Filippo Maria Bianchi, editors, *Image Analysis*, volume 10269, pages 515–525, Cham, 2017. Springer International Publishing. Book Title: Image Analysis Series Title: Lecture Notes in Computer Science.

[FCLI10] Sean Follmer, David Carr, Emily Lovell, and Hiroshi Ishii. CopyCAD: remixing physical objects with copy and paste from the real world. In *Adjunct proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 381–382, New York, New York, USA, October 2010. Association for Computing Machinery.

[FDFI20] Jack Forman, Mustafa Doga Dogan, Hamilton Forsythe, and Hiroshi Ishii. DefeXtiles: 3D Printing Quasi-Woven Fabric via Under-Extrusion. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 1222–1233, New York, NY, USA, October 2020. Association for Computing Machinery.

[FHP21] Frikk Fossdal, Rogardt Heldal, and Nadya Peek. Interactive Digital Fabrication Machine Control Directly Within a CAD Environment. In *Symposium on Computational Fabrication*, SCF '21, pages 1–15, New York, NY, USA, October 2021. Association for Computing Machinery.

[FIM+21] Jeremy L. Fredricks, Hareesh Iyer, Robin McDonald, Jeffrey Hsu, Andrew M. Jimenez, and Eleftheria Roumeli. Spirulina-based composites for

3D-printing. *Journal of Polymer Science*, 59(22):2878–2894, 2021. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/pol.20210683.

[FMS93] Steven Feiner, Blair Macintyre, and Dorée Seligmann. Knowledge-based augmented reality. *Commun. ACM*, 36(7):53–62, July 1993.

[FNH+23] Frikk H Fossdal, Vinh Nguyen, Rogardt Heldal, Corie L. Cobb, and Nadya Peek. Vespidae: A Programming Framework for Developing Digital Fabrication Workflows. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference*, DIS '23, pages 2034–2049, New York, NY, USA, July 2023. Association for Computing Machinery.

[FO10] Christian Fuchs and Marianna Obrist. HCI and Society: Towards a Typology of Universal Design Principles. *International Journal of Human-Computer Interaction*, 26(6):638–656, May 2010.

[For23] Formlabs. PreForm, 2023.

[FTD+14] S. Felton, M. Tolley, E. Demaine, D. Rus, and R. Wood. A method for building self-folding machines. *Science*, 345(6197):644–646, August 2014.

[Fut83] Yoshihiko Futamura. Partial Computation of Programs. *RIMS Kôkyûroku*, 482:255–295, March 1983.

[GB08] Saul Greenberg and Bill Buxton. Usability Evaluation Considered Harmful (Some of the Time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 111–120, New York, NY, USA, 2008. ACM.

[GBT19] Elia A. Guzzi, Giovanni Bovone, and Mark W. Tibbitt. Universal Nanocarrier Ink Platform for Biomaterials Additive Manufacturing. *Small*, 15(51):1905421, 2019. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smll.201905421.

[Geo14] Damian George. MicroPython - Python for Microcontrollers, 2014.

[Ger07] Neil A. Gershenfeld. *Fab: the coming revolution on your desktop - from personal computers to personal fabrication.* Basic Books, New York, NY, paperback publ edition, 2007.

[Gin22] Gina Häußge. OctoPrint.org, 2022.

[GL19] Emrecan Gulay and Andrés Lucero. Integrated Workflows: Generating Feedback Between Digital and Physical Realms. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 1–15, New York, NY, USA, May 2019. Association for Computing Machinery.

[HAC16] Nathaniel Hudson, Celena Alcock, and Parmit K. Chilana. Understanding Newcomers to 3D Printing: Motivations, Workflows, and Barriers of Casual Makers. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 384–396, New York, NY, USA, 2016. ACM.

[Had07] Hadley Wickham. ggplot2, 2007. original-date: 2008-05-25T01:21:32Z.

[Har22] Brendan Harmon. CNC Surface Milling, 2022.

[HBJP23] Mare Hirsch, Gabrielle Benabdallah, Jennifer Jacobs, and Nadya Peek. Nothing Like Compilation: How Professional Digital Fabrication Workflows Go Beyond Extruding, Milling, and Machines. *ACM Trans. Comput.-Hum. Interact.*, July 2023. Just Accepted.

[HCA+18] Sehoon Ha, Stelian Coros, Alexander Alspach, Joohyung Kim, and Katsu Yamane. Computational co-optimization of design parameters and motion trajectories for robotic systems. *The International Journal of Robotics Research*, 37(13-14):1521–1536, December 2018. Publisher: SAGE Publications Ltd STM.

[Her03] Hernando Barragán. Wiring, 2003.

[HHB+19] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 270:1–270:12, New York, NY, USA, 2019. ACM.

[HHHM16] Megan Hofmann, Jeffrey Harris, Scott E. Hudson, and Jennifer Mankoff. Helping Hands: Requirements for a Prototyping Methodology for Upper-limb Prosthetics Users. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1769–1780, New York, NY, USA, May 2016. Association for Computing Machinery.

[HHHM18] Megan Hofmann, Gabriella Hann, Scott E. Hudson, and Jennifer Mankoff. Greater Than the Sum of Its PARTs: Expressing and Reusing Design Intent in 3D Models. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 301:1–301:12, New York, NY, USA, 2018. ACM.

[HJR+16] Lara Houston, Steven J. Jackson, Daniela K. Rosner, Syed Ishtiaque Ahmed, Meg Young, and Laewoo Kang. Values in Repair. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1403–1414, New York, NY, USA, May 2016. Association for Computing Machinery.

[HJS+20] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–12, New York, NY, USA, April 2020. Association for Computing Machinery.

[HMH20] Megan Hofmann, Jennifer Mankoff, and Scott E. Hudson. KnitGIST: A Programming Synthesis Toolkit for Generating Functional Machine-Knitting Textures. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 1234–1247, New York, NY, USA, October 2020. Association for Computing Machinery.

[HQW+21] Gaoping Huang, Xun Qian, Tianyi Wang, Fagun Patel, Maitreya Sreeram, Yuanzhi Cao, Karthik Ramani, and Alexander J. Quinn. AdapTutAR: An Adaptive Tutoring System for Machine Tasks in Augmented Reality. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–15, New York, NY, USA, May 2021. Association for Computing Machinery.

[HSH18] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 532:1–532:12, New York, NY, USA, 2018. ACM.

[HT19a] Ammar Hattab and Gabriel Taubin. Interactive Fabrication of CSG Models with Assisted Carving. In *Proceedings of the Thirteenth International Conference on Tangible, Embedded, and Embodied Interaction*, TEI '19, pages 677–682, New York, NY, USA, March 2019. Association for Computing Machinery.

[HT19b] Ammar Hattab and Gabriel Taubin. Rough carving of 3D models with spatial augmented reality. In *Proceedings of the ACM Symposium on Computational Fabrication*, SCF '19, pages 1–10, Pittsburgh, Pennsylvania, June 2019. Association for Computing Machinery.

[Hus06] David Huson. Digital fabrication techniques in art/craft and designer/maker ceramics. *UWE Bristol Technical Reports*, Volume, September 2006.

[IA08] James Inglese and Douglas S. Auld. High Throughput Screening (HTS) Techniques: Applications in Chemical Biology. In *Wiley Encyclopedia of Chemical Biology*, page wecb223. John Wiley & Sons, Inc., Hoboken, NJ, USA, May 2008.

[IFW+16] Alexandra Ion, Johannes Frohnhofen, Ludwig Wall, Robert Kovacs, Mirela Alistar, Jack Lindsay, Pedro Lopes, Hsiang-Ting Chen, and Patrick Baudisch. Metamaterial Mechanisms. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 529–539, New York, NY, USA, 2016. ACM.

[IKS+18] Alexandra Ion, Robert Kovacs, Oliver S. Schneider, Pedro Lopes, and Patrick Baudisch. Metamaterial Textures. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, New York, NY, USA, April 2018. Association for Computing Machinery.

[Ing13] Tim Ingold. *Making: anthropology, archaeology, art and architecture*. Routledge, Milton Park, Abingdon, Oxon, 2013.

[Ion20] Ionos. Imperative programming: Overview of the oldest programming paradigm, 2020.

[IPPA15] John L. Irwin, Douglas E. Oppliger P.e, Joshua M. Pearce, and Gerald Anzalone. Evaluation of RepRap 3D Printer Workshops in K-12 STEM. In *Proceedings of 2015 ASEE Annual Conference & Exposition*, pages 26.696.1–26.696.18, June 2015.

[IU97] Hiroshi Ishii and Brygg Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, pages 234–241, Atlanta Georgia USA, March 1997. ACM.

[IWKB17] Alexandra Ion, Ludwig Wall, Robert Kovacs, and Patrick Baudisch. Digital Mechanical Metamaterials. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 977–988, New York, NY, USA, 2017. ACM.

[JDR+19] Eunice Jun, Maureen Daum, Jared Roesch, Sarah Chasins, Emery Berger, Rene Just, and Katharina Reinecke. Tea: A High-level Language and Runtime System for Automating Statistical Analysis. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pages 591–603, New York, NY, USA, October 2019. Association for Computing Machinery.

[Jer20] Jeremiah Langston. Milling a 2 side contoured part using dowel pin indexing features (Fusion 360), March 2020.

[JFP+20] Trevor G. Johnston, Jacob P. Fillman, Hans Priks, Tobias Butelmann, Tarmo Tamm, Rahul Kumar, Petri-Jaan Lahtvee, and Alshakim Nelson. Cell-Laden Hydrogels for Multikingdom 3D Printing. *Macromolecular Bioscience*, 20(8):2000121, 2020. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/mabi.202000121.

[Juh21] Juha Kuusama. LitePlacer | The Pick and Place Machine for Your Lab, 2021.

[Jup23] JupyterCAD Development Team. JupyterCAD — JupyterCAD documentation, 2023.

[JZ15] Jennifer Jacobs and Amit Zoran. Hybrid Practice in the Kalahari: Design Collaboration through Digital Tools and Hunter-Gatherer Craft. In *Proceedings of the 33rd*

Annual ACM Conference on Human Factors in Computing Systems, CHI '15, pages 619–628, New York, NY, USA, 2015. Association for Computing Machinery.

[KG17] Hyeonsu Kang and Philip J. Guo. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pages 737–745, New York, NY, USA, October 2017. Association for Computing Machinery.

[KGF15] Jarrod Knibbe, Tovi Grossman, and George Fitzmaurice. Smart Makerspace: An Immersive Instructional Space for Physical Tasks. In *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces*, ITS '15, pages 83–92, New York, NY, USA, 2015. ACM.

[KHM17] Mary Beth Kery, Amber Horvath, and Brad Myers. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 1265–1276, Denver Colorado USA, May 2017. ACM.

[Kib22] Hassan Kibirige. plotnine, May 2022. original-date: 2017-04-24T19:00:44Z.

[KLP+20] Yoonji Kim, Hyein Lee, Ramkrishna Prasad, Seungwoo Je, Youngkyung Choi, Daniel Ashbrook, Ian Oakley, and Andrea Bianchi. SchemaBoard: Supporting Correct Assembly of Schematic Circuits using Dynamic In-Situ Visualization. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 987–998, New York, NY, USA, October 2020. Association for Computing Machinery.

[KM10] Shigeru Kondo and Takashi Miura. Reaction-diffusion model as a framework for understanding biological pattern formation. *Science*, 329(5999):1616–1620, September 2010.

[Knu92] Donald Ervin Knuth. *Literate programming*. Number no. 27 in CSLI lecture notes. Center for the Study of Language and Information, Stanford, Calif., 1992.

[KP10] Stacey Kuznetsov and Eric Paulos. Rise of the expert amateur: DIY projects, communities, and cultures. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries*, NordiCHI '10, pages 295–304, New York, NY, USA, October 2010. Association for Computing Machinery.

[KRH+20] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium* *on User Interface Software and Technology*, UIST '20, pages 140–151, New York, NY, USA, October 2020. Association for Computing Machinery.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. Jupyter Notebooks – a publishing format for reproducible computational workflows. In Fernando Loizides and Birgit Scmidt, editors, *Proceedings of the 20th International Conference on Electronic Publishing*, pages 87–90. IOS Press, 2016.

[KZT+18] Jeeeun Kim, Clement Zheng, Haruki Takahashi, Mark D Gross, Daniel Ashbrook, and Tom Yeh. Compositional 3D Printing: Expanding & Supporting Workflows Towards Continuous Fabrication. In *Proceedings of the 2Nd ACM Symposium on Computational Fabrication*, SCF '18, pages 5:1–5:10, New York, NY, USA, 2018. ACM.

[LaC17] Don LaCourse. 2-Sided (Flip) Machining Explored, July 2017.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, October 1973.

[Lan24] LangChain. Run LLMs locally, February 2024.

[LBM14] Tom Lieber, Joel Brandt, and Rob Miller. Addressing misconceptions about code with always-on programming visualizations. *CHI*, 2014.

[LBM+20] Jingyi Li, Joel Brandt, Radomír Mech, Maneesh Agrawala, and Jennifer Jacobs. Supporting Visual Artists in Programming through Direct Inspection and Control of Program Execution. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–12, New York, NY, USA, April 2020. Association for Computing Machinery.

[Lei14] Daan Leijen. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science*, 153, June 2014.

[LHR+21] Benjamin Lutz, Philip Howell, Daniel Regulin, Bastian Engelmann, and Jörg Franke. Towards Material-Batch-Aware Tool Condition Monitoring. *Journal of Manufacturing and Materials Processing*, 5(4):103, December 2021. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute.

[LHV+18] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 36:1–36:17, New York, NY, USA, 2018. ACM. event-place: Montreal QC, Canada.

[LJCH17] Jingyi Li, Jennifer Jacobs, Michelle Chang, and Björn Hartmann. Direct and Immediate Drawing with CNC Machines. In *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication*, SCF '17, pages 11:1–11:2, New York, NY, USA, 2017. ACM.

[LJDM23] Mackenzie Leake, Kathryn Jin, Abe Davis, and Stefanie Mueller. InStitches: Augmenting Sewing Patterns with Personalized Material-Efficient Practice. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, pages 1–14, New York, NY, USA, April 2023. Association for Computing Machinery.

[LL22] Jasmine Lu and Pedro Lopes. Integrating Living Organisms in Devices to Implement Care-based Interactions. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, UIST '22, pages 1–13, New York, NY, USA, October 2022. Association for Computing Machinery.

[LLWL23] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual Instruction Tuning, December 2023. arXiv:2304.08485 [cs].

[LMM+18] Jeffrey Ian Lipton, Robert MacCurdy, Zachary Manchester, Lillian Chin, Daniel Cellucci, and Daniela Rus. Handedness in shearing auxetics creates rigid and compliant structures. *Science*, 360(6389):632–635, May 2018. Publisher: American Association for the Advancement of Science.

[LRR+23] Jingyi Li, Eric Rawn, Jacob Ritchie, Jasper Tran O'Leary, and Sean Follmer. Beyond the Artifact: Power as a Lens for Creativity Support Tools. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, pages 1–15, New York, NY, USA, October 2023. Association for Computing Machinery.

[LTY+16] Joanne Lo, Cesar Torres, Isabel Yang, Jasper O'Leary, Danny Kaufman, Wilmot Li, Mira Dontcheva, and Eric Paulos. Aesthetic Electronics: Designing, Sketching, and Fabricating Circuits Through Digital Exploration. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 665–676, New York, NY, USA, 2016. ACM.

[LWK+20] Nic A. Lee, Ramon E. Weber, Joseph H. Kennedy, Josh J. Van Zak, Miana Smith, Jorge Duro-Royo, and Neri Oxman. Sequential Multimaterial Additive Manufacturing of Functionally Graded Biopolymer Composites. *3D Printing and Additive Manufacturing*, 7(5):205–215, October 2020. Publisher: Mary Ann Liebert, Inc., publishers.

[LZB+17] Min Liu, Yunbo Zhang, Jing Bai, Yuanzhi Cao, Jeffrey M. Alperovich, and Karthik Ramani. WireFab: Mix-Dimensional Modeling and Fabrication for 3D Mesh Models. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 965–976. ACM, May 2017.

[Mar16] Martin Barfoed. Two Sided 3D Milling using Pin Method. CAD and CAM in Fusion 360, October 2016.

[MAYZ+20] Ali Mahdavi-Amiri, Fenggen Yu, Haisen Zhao, Adriana Schulz, and Hao Zhang. VDAC: volume decompose-and-carve for subtractive manufacturing. *ACM Trans. Graph.*, 39(6):203:1–203:15, November 2020.

[MBF+18] Florian Müller, Maximilian Barnikol, Markus Funk, Martin Schmitz, and Max Mühlhäuser. CaMea: Camera-Supported Workpiece Measurement for CNC Milling Machines. In *Proceedings of the 11th PErvasive Technologies Related to Assistive Environments Conference*, PETRA '18, pages 345–350, New York, NY, USA, June 2018. Association for Computing Machinery.

[McN23] McNeel. Grasshopper, 2023.

[MEJV+22] Daniela Mitterberger, Selen Ercan Jenny, Lauren Vasey, Ena Lloret-Fritschi, Petrus Aejmelaeus-Lindström, Fabio Gramazio, and Matthias Kohler. Interactive Robotic Plastering: Augmented Interactive Design and Fabrication for On-site Robotic Plastering. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, pages 1–18, New York, NY, USA, April 2022. Association for Computing Machinery.

[MF17] Justin Matejka and George Fitzmaurice. Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 1290–1294, New York, NY, USA, May 2017. Association for Computing Machinery.

[MIG+14] Stefanie Mueller, Sangha Im, Serafima Gurevich, Alexander Teibrich, Lisa Pfisterer, François Guimbretière, and Patrick Baudisch. WirePrint: 3D printed previews for fast prototyping. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, UIST '14, pages 273–280, New York, NY, USA, October 2014. Association for Computing Machinery.

[MJMA19] Chandan Mahapatra, Jonas Kjeldmand Jensen, Michael McQuaid, and Daniel Ashbrook. Barriers to End-User Designers of Augmented Fabrication. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 1–15, Glasgow, Scotland Uk, May 2019. Association for Computing Machinery.

[MJS+20] Karthick Manikandan, Xuepeng Jiang, Amit A. Singh, Beiwen Li, and Hantang Qin. Effects of Nozzle Geometries on 3D Printing of Clay Constructs: Quantifying Contour Deviation and Mechanical Properties. *Procedia Manufacturing*, 48:678–683, January 2020.

[MLB12] Stefanie Mueller, Pedro Lopes, and Patrick Baudisch. Interactive construction: interactive fabrication of functional mechanical devices. *Proceedings of the 25th annual ACM symposium on User interface software and technology*, pages 599–606, October 2012. Conference Name: UIST '12: The 25th Annual ACM Symposium on User Interface Software and Technology ISBN: 9781450315807 Place: Cambridge Massachusetts USA Publisher: ACM.

[MS05] Molga, M and Smutnicki, C. Test functions for optimization needs, 2005.

[MSDRO14] Laia Mogas-Soldevila, Jorge Duro-Royo, and Neri Oxman. Water-Based Robotic Fabrication: Large-Scale Additive Manufacturing of Functionally Graded Hydrogel Composites via Multichamber Extrusion. *3D Printing and Additive Manufacturing*, 1(3):141–151, September 2014. Publisher: Mary Ann Liebert, Inc., publishers.

[MTP08] Deborah Morgenthal, Suzanne J. E Tourtillott, and Penland School of Crafts. *The Penland book of ceramics: master classes in ceramic techniques.* Union Square & Co, 2008. OCLC: 195713911.

[MWN+19] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):438–448, January 2019. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[Mø95] A. Mørch. Three levels of end-user tailoring: customization, integration, and extension. In *Proceedings of the Third Decennial Aarhus Conference*, August 1995.

[NAH+18] Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James Mccann. Automatic Machine Knitting of 3D Meshes. *ACM Trans. Graph.*, 37(3):35:1–35:15, August 2018.

[Nan21] Chandrakana Nandi. *Programming Language Tools and Techniques for Computational Fabrication.* Thesis, University of Washington, 2021. Accepted: 2021-10-29T16:20:06Z.

[Nar93] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing.* The MIT Press, 1993.

[Nat19] National Academies. *Reproducibility and Replicability in Science.* National Academies Press, September 2019.

[Nic17] Nicolas Padfield. More elegant CNC dogbones, March 2017.

[NN20] Shukantu Dev Nath and Sabrina Nilufar. An Overview of Additive Manufacturing of Polymers and Associated Composites. *Polymers*, 12(11):2719, November 2020. Number: 11 Publisher: Multidisciplinary Digital Publishing Institute.

[NWA+20] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 31–44, New York, NY, USA, June 2020. Association for Computing Machinery.

[OAA+23] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie

Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, C. J. Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. GPT-4 Technical Report, December 2023. arXiv:2303.08774 [cs].

[Obs23a] Observable. Observable Plot, 2023.

[Obs23b] Observable Notebook. Observable Notebook, 2023.

[Obs24] Observable. Observable Framework, March 2024. original-date: 2023-09-27T21:06:30Z.

[ODC+16] Jifei Ou, Gershon Dublon, Chin-Yi Cheng, Felix Heibeck, Karl Willis, and Hiroshi Ishii. Cilllia: 3D Printed Micro-Pillar Structures for Surface Texture, Actuation and Sensing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 5753–5764, New York, NY, USA, May 2016. Association for Computing Machinery.

[OGL08] Alex Olwal, Jonny Gustafsson, and Christoffer Lindfors. Spatial augmented reality on industrial CNC-machines. In Ian E. McDowall and Margaret Dolinsky, editors, *Proceedings of Electronic Imaging 2008*, page 680409, San Jose, CA, February 2008.

[OI04] Dan O'Sullivan and Tom Igoe. *Physical computing: sensing and controlling the physical world with computers*. Thomson, Boston, 2004. OCLC: ocm56122209.

[Ols07] Dan R. Olsen, Jr. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 251–258, New York, NY, USA, 2007. ACM. event-place: Newport, Rhode Island, USA.

[OMB+21] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 511–525, New York, NY, USA, June 2021. Association for Computing Machinery.

[Ong92] Walter J. Ong. Writing is a technology that restructures thought. In Pamela A. Downing, Susan D. Lima, and Michael Noonan, editors, *The Linguistics of Literacy*, Typological Studies in Language, page 293. John Benjamins Publishing Company, July 1992.

[Ons22] Onshape. Onshape, 2022.

[PBW+18] Huaishu Peng, Jimmy Briggs, Cheng-Yao Wang, Kevin Guo, Joseph Kider, Stefanie Mueller, Patrick Baudisch, and François Guimbretière. RoMA: Interactive Fabrication with Augmented Reality and a Robotic 3D Printer. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 579:1–579:12, New York, NY, USA, 2018. ACM.

[Pee16] Nadya Meile Peek. *Making machines that make : object-oriented hardware meets object-oriented software*. Thesis, Massachusetts Institute of Technology, 2016.

[Per21] Jeffrey M. Perkel. Reactive, reproducible, collaborative: computational notebooks evolve. *Nature*, 593(7857):156–157, May 2021.

[PFX+22] Michal Piovarči, Michael Foshey, Jie Xu, Timmothy Erps, Vahid Babaei, Piotr Didyk, Szymon Rusinkiewicz, Wojciech Matusik, and Bernd Bickel. Closed-loop control of direct ink writing via reinforcement learning. *ACM Trans. Graph.*, 41(4):112:1–112:10, July 2022.

[PG18] Nadya Peek and Neil Gershenfeld. Mods: Browser-Based Rapid Prototyping Workflow Composition. In *Proceedings of the 38th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, page 6, Mexico City, Mexico, 2018.

[PJJ98] Paul Hudak, John Peterson, and Joseph Fasel. A Gentle Introduction to Haskell: IO, 1998.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 71–84, New York, NY, USA, March 1993. Association for Computing Machinery.

[PK20] Xiaoying Pu and Matthew Kay. A Probabilistic Grammar of Graphics. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, New York, NY, USA, April 2020. Association for Computing Machinery.

[Pol67] Michael Polanyi. *The tacit dimension.* Anchor books philosophy. Doubleday, Garden City, New York, 1967.

[POM14] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. *SIGPLAN Not.*, 49(9):123–135, August 2014.

[PPSV17] Alexander Y. Piggott, Jan Petykiewicz, Logan Su, and Jelena Vučković. Fabrication-constrained nanophotonic inverse design. *Scientific Reports*, 7(1):1786, May 2017. Number: 1 Publisher: Nature Publishing Group.

[Pru12] Josef Prusa. Prusa3D Printer, 2012.

[Qui60] Willard Van Orman Quine. *Word and Object.* MIT Press, Cambridge, MA, 1960.

[R P93] R Project. R: The R Project for Statistical Computing, August 1993.

[Rea23] Jake Read. Clank: Fabricatable, Modular CNC, 2023.

[RLR18] Jessica Rosenkrantz and Jesse Louis-Rosenberg. Coral Cup, November 2018.

[RM18] Troels A Rasmussen and Timothy Merritt. ProjecTables: Augmented CNC tools for sustainable creative practices. *International Journal of Architectural Computing*, 16(3):227–242, September 2018. Publisher: SAGE Publications.

[Rob02] Rush D. Robinett, editor. *Flexible robot dynamics and controls.* Number 19 in IFSR international series on systems science and engineering. Kluwer Academic/Plenum Publishers, New York, 2002.

[Rob22] Robert McNeel & Associates. Rhinoceros 3D, 2022.

[RTH18] Adam Rule, Aurélien Tabard, and James D. Hollan. Exploration and Explanation in Computational Notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 1–12, New York, NY, USA, April 2018. Association for Computing Machinery.

[Rya22] Matthew Ryan. Immersive Virtual Reality Error Management Training for CNC Machining Setup Procedures. *Theses*, April 2022.

[SBP21] Nischal Shrestha, Titus Barik, and Chris Parnin. Unravel: A Fluent Code Explorer for Data Wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, pages 198–207, New York, NY, USA, October 2021. Association for Computing Machinery.

[SBR+15] Christian Schumacher, Bernd Bickel, Jan Rys, Steve Marschner, Chiara Daraio, and Markus Gross. Microstructures to Control Elasticity in 3D Printing. *ACM Trans. Graph.*, 34(4):136:1–136:13, July 2015.

[SBS+20] Rachel Soo Hoo Smith, Christoph Bader, Sunanda Sharma, Dominik Kolb, Tzu-Chieh Tang, Ahmed Hosny, Felix Moser, James C. Weaver, Christopher A. Voigt, and Neri Oxman. Hybrid Living Materials: Digital Design and Fabrication of 3D Multimaterial Structures with Programmable Biohybrid Surfaces. *Advanced Functional Materials*, 30(7):1907401, 2020. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/adfm.201907401.

[Sch83] Donald A. Schön. *The reflective practitioner: how professionals think in action*. Basic Books, New York, 1983.

[Sch23] Phil Schatzmann. OpenSCAD Kernel for Jupyter, September 2023. original-date: 2020-02-26T09:42:36Z.

[SG08] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):8:1–8:29, April 2008.

[Sho22] Shopbot Tools. ShopBot, 2022.

[SHRB20] Stephen C Slota, Andrew S Hoffman, David Ribes, and Geoffrey C Bowker. Prospecting (in) the data sciences. *Big Data & Society*, 7(1):2053951720906849, January 2020. Publisher: SAGE Publications Ltd.

[Sim16] Gilbert Simondon. *On the mode of existence of technical objects*. Univocal Pub, Minneapolis, MN, 2016.

[SJS+18] Abhijit Saha, Trevor G. Johnston, Ryan T. Shafranek, Cassandra J. Goodman, Jesse G. Zalatan, Duane W. Storti, Mark A. Ganter, and Alshakim Nelson. Additive Manufacturing of Catalytically Active Living Materials. *ACS Appl. Mater. Interfaces*, 10(16):13373–13380, April 2018. Publisher: American Chemical Society.

[SKX+22] Ryo Suzuki, Adnan Karim, Tian Xia, Hooman Hedayati, and Nicolai Marquardt. Augmented Reality and Robotics: A Survey and Taxonomy for AR-enhanced Human-Robot Interaction and Robotic Interfaces. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, pages 1–33, New York, NY, USA, April 2022. Association for Computing Machinery.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA, 2006. ACM. event-place: San Jose, California, USA.

[Smi07] Peter Smid. *CNC Programming Handbook, Third Edition*. Industrial Press, Inc., New York, N.Y, third edition edition, November 2007.

[SMWH17] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, January 2017.

[SNL+16] Eldon Schoop, Michelle Nguyen, Daniel Lim, Valkyrie Savage, Sean Follmer, and Björn Hartmann. Drill Sergeant: Supporting Physical Construction Projects Through an Ecosystem of Augmented Tools. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '16, pages 1607–1614, New York, NY, USA, 2016. ACM.

[SP22] Blair Subbaraman and Nadya Peek. p5.fab: Direct Control of Digital Fabrication Machines from a Creative Coding Environment. In *Proceedings of the 2022 ACM Designing Interactive Systems Conference*, DIS '22, pages 1148–1161, New York, NY, USA, June 2022. Association for Computing Machinery.

[SP23] Blair Subbaraman and Nadya Peek. 3D Printers Don't Fix Themselves: How Maintenance is Part of Digital Fabrication. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference*, DIS '23, pages 2050–2065, New York, NY, USA, July 2023. Association for Computing Machinery.

[SRHH16] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, January 2016. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[SRPX13] Greg Saul, Tiago Rorke, Huaishu Peng, and Cheng Xu. Make Your Own Piccolo. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction*, TEI '13, pages 439–442, New York, NY, USA, 2013. ACM.

[Ste19] Thomas Stevens. Rotation error when flipping stock - CNC Machines / Shapeoko, July 2019.

[SWH14] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, UIST ’14, pages 669–678, New York, NY, USA, October 2014. Association for Computing Machinery.

[Tan90] Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, June 1990.

[Tan13] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34, May 2013.

[tar21] tarkka. Holier than Thou: Precision Holes by Drilling, Boring, and Reaming, February 2021.

[TFRSB17] Vasiliki Tsaknaki, Ylva Fernaeus, Emma Rapp, and Jordi Solsona Belenguer. Articulating Challenges of Hybrid Crafting for the Case of Interactive Silversmith Practice. In *Proceedings of the 2017 Conference on Designing Interactive Systems*, DIS ’17, pages 1187–1200, New York, NY, USA, June 2017. Association for Computing Machinery.

[THL+18] Jose C. Tovar, J. Steen Hoyer, Andy Lin, Allison Tielking, Steven T. Callen, S. Elizabeth Castillo, Michael Miller, Monica Tessman, Noah Fahlgren, James C. Carrington, Dmitri A. Nusinow, and Malia A. Gehan. Raspberry Pi–powered imaging for plant phenotyping. *Applications in Plant Sciences*, 6(3):1–12, 2018.

[Tih18] Timea Tihanyi. Burst and Follow, 2018.

[TLP16] Cesar Torres, Wilmot Li, and Eric Paulos. ProxyPrint: Supporting Crafting Practice Through Physical Computational Proxies. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, DIS ’16, pages 158–169, New York, NY, USA, 2016. ACM.

[TMG+15] Alexander Teibrich, Stefanie Mueller, François Guimbretière, Robert Kovacs, Stefan Neubert, and Patrick Baudisch. Patching Physical Objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST ’15, pages 83–91, New York, NY, USA, November 2015. Association for Computing Machinery.

[TOBP23] Jasper Tran O’Leary, Gabrielle Benabdallah, and Nadya Peek. Imprimer: Computational Notebooks for CNC Milling. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI ’23, pages 1–15, New York, NY, USA, April 2023. Association for Computing Machinery.

[TOJP22] Jasper Tran O'Leary, Eunice Jun, and Nadya Peek. Improving Programming for Exploratory Digital Fabrication with Inline Machine Control and Styled Toolpath Visualizations. In *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication*, SCF '22, pages 1–12, New York, NY, USA, October 2022. Association for Computing Machinery.

[TONLP21] Jasper Tran O'Leary, Chandrakana Nandi, Khang Lee, and Nadya Peek. Taxon: a Language for Formal Reasoning with Digital Fabrication Machines. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, pages 691–709, New York, NY, USA, October 2021. Association for Computing Machinery.

[TORZP24] Jasper Tran O'Leary, Thrisha Ramesh, Octi Zhang, and Nadya Peek. Tandem: Reproducible Digital Fabrication Workflows as Multimodal Programs. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, pages 1–15, New York, NY, USA, May 2024. Association for Computing Machinery.

[TP21] Rundong Tian and Eric Paulos. Adroid: Augmenting Hands-on Making with a Collaborative Robot. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, pages 270–281, New York, NY, USA, October 2021. Association for Computing Machinery.

[TPBM21] Iremnur Tokac, J. Philips, H. Bruyninckx, and A. Vande Moere. Fabrication grammars: bridging design and robotics to control emergent material expressions. *Construction Robotics*, 2021.

[TSBS+23] Sławomir K. Tadeja, Luca O. Solari Bozzi, Kerr D. G. Samson, Sebastian W. Pattinson, and Thomas Bohné. Exploring the repair process of a 3D printer using augmented reality-based guidance. *Computers & Graphics*, 117:134–144, December 2023.

[TSK+19] Rundong Tian, Vedant Saran, Mareike Kritzler, Florian Michahelles, and Eric Paulos. Turn-by-Wire: Computationally Mediated Physical Fabrication. In *Proceedings of the 32Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pages 713–725, New York, NY, USA, 2019. ACM.

[TSN+18] Cesar Torres, Sarah Sterman, Molly Nicholas, Richard Lin, Eric Pai, and Eric Paulos. Guardians of Practice: A Contextual Inquiry of Failure-Mitigation Strategies within Creative Practices. In *Proceedings of the 2018 Designing Interactive Systems Conference*, DIS '18, pages 1259–1267, New York, NY, USA, June 2018. Association for Computing Machinery.

[TSP24] Hannah Twigg-Smith and Nadya Peek. Dynamic Toolchains: Software Infrastructure for Digital Fabrication Workflows. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–20, 2024.

[TSTOP21] Hannah Twigg-Smith, Jasper Tran O'Leary, and Nadya Peek. Tools, Tricks, and Hacks: Exploring Novel Digital Fabrication Workflows on #PlotterTwitter. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–15, New York, NY, USA, May 2021. Association for Computing Machinery.

[Tuk77] John Wilder Tukey. *Exploratory data analysis*. Addison-Wesley series in behavioral science. Addison-Wesley Pub. Co, Reading, Mass, 1977.

[ULSF10] Alexander S. Urban, Andrey A. Lutich, Fenando D. Stefani, and Jochen Feldmann. Laser Printing Single Gold Nanoparticles. *Nano Lett.*, 10(12):4794–4798, December 2010. Publisher: American Chemical Society.

[Ult21] Ultimaker. Ultimaker/Cura Printer Definitions, 2021.

[Vec] Vectric. VCarve Pro Product Page.

[Vec21] Vectric Ltd. Introduction to Two Sided Machining | Vectric V11 Tutorials, July 2021.

[VG19] Priyan Vaithilingam and Philip J. Guo. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, pages 563–576, New York, NY, USA, 2019. ACM. event-place: New Orleans, LA, USA.

[Vic11] Bret Victor. Up and Down the Ladder of Abstraction: A systematic approach to interactive visualization., 2011.

[Vic12] Bret Victor. Learnable Programming, September 2012.

[VTSTOP20] Joshua Vasquez, Hannah Twigg-Smith, Jasper Tran O'Leary, and Nadya Peek. Jubilee: An Extensible Machine for Multi-tool Fabrication. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, New York, NY, USA, 2020. ACM.

[Wad92] Philip Wadler. Comprehending Monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[Was19] Wasp. Delta WASP 2040 Clay, 2019.

[WDBD21] Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–12, New York, NY, USA, May 2021. Association for Computing Machinery.

[Wei99] Mark Weiser. The Computer for the 21st Century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, July 1999.

[WFB+21] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, pages 1–15, New York, NY, USA, May 2021. Association for Computing Machinery.

[WGD+19] Jitkanya Wong, Alex T. Gong, Peter A. Defnet, Leire Meabe, Bruce Beauchamp, Robert M. Sweet, Haritz Sardon, Corie L. Cobb, and Alshakim Nelson. 3D Printing Ionogel Auxetic Frameworks for Stretchable Sensors. *Advanced Materials Technologies*, 4(9):1900452, 2019. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/admt.201900452.

[WHAG15] Christian Weichel, John Hardy, Jason Alexander, and Hans Gellersen. ReForm: Integrating Physical and Digital Design through Bidirectional Fabrication. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, pages 93–102, Charlotte, NC, USA, November 2015. Association for Computing Machinery.

[WHS20] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 152–165, New York, NY, USA, October 2020. Association for Computing Machinery.

[Win22] Windell Oskay. AxiDraw, March 2022. original-date: 2016-02-01T11:21:13Z.

[WMA+16] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, January 2016. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[WST+19] Max Willsey, Ashley P. Stephenson, Chris Takahashi, Pranav Vaid, Bichlien H. Nguyen, Michal Piszczek, Christine Betts, Sharon Newman, Sarang Joshi, Karin Strauss, and Luis Ceze. Puddle: A Dynamic, Error-Correcting, Full-Stack Microfluidics Platform. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 183–197, Providence, RI, USA, April 2019. Association for Computing Machinery.

[WVMR18] D. Wood, C. Vailati, A. Menges, and M. Rüggeberg. Hygroscopically actuated wood elements for weather responsive and self-forming building parts – Facilitating upscaling and complex shape changes. *Construction and Building Materials*, 165:782–791, March 2018.

[WW05] Leland Wilkinson and Graham Wills. *The grammar of graphics*. Statistics and computing. Springer, New York, 2nd ed edition, 2005.

[WXW+11] Karl D.D. Willis, Cheng Xu, Kuan-Ju Wu, Golan Levin, and Mark D. Gross. Interactive fabrication: new interfaces for digital fabrication. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction - TEI '11*, page 69, Funchal, Portugal, 2011. ACM Press.

[WYW+16] Guanyun Wang, Lining Yao, Wen Wang, Jifei Ou, Chin-Yi Cheng, and Hiroshi Ishii. xPrint: A Modularized Liquid Printer for Smart Materials Deposition. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 5743–5752, New York, NY, USA, 2016. ACM.

[XNB+21] Lirong Xiang, Trevor M. Nolan, Yin Bao, Mitch Elmore, Taylor Tuel, Jingyao Gai, Dylan Shah, Ping Wang, Nicole M. Huser, Ashley M. Hurd, Sean A. McLaughlin, Stephen H. Howell, Justin W. Walley, Yanhai Yin, and Lie Tang. Robotic Assay for Drought (RoAD): an automated phenotyping system for brassinosteroid and drought responses. *Plant Journal*, 107(6):1837–1853, 2021.

[YCC+20] Chih-Kai Yang, Yu-Hsi Chen, Tung-Jui Chuang, Kalpana Shankhwar, and Shana Smith. An augmented reality-based training system with a natural user interface for manual milling operations. *Virtual Reality*, 24(3):527–539, September 2020.

[YIO+15] Hironori Yoshida, Takeo Igarashi, Yusuke Obuchi, Yosuke Takami, Jun Sato, Mika Araki, Masaaki Miki, Kosuke Nagata, Kazuhide Sakai, and Syunsuke Igarashi. Architecture-scale human-assisted additive manufacturing. *ACM Trans. Graph.*, 34(4):88:1–88:8, July 2015.

[YK16] Junichi Yamaoka and Yasuaki Kakehi. MiragePrinter: interactive fabrication on a 3D printer with a mid-air display. In *ACM SIGGRAPH 2016 Studio*, SIGGRAPH '16, pages 1–2, New York, NY, USA, July 2016. Association for Computing Machinery.

[YMZ20] Nur Yildirim, James McCann, and John Zimmerman. Digital Fabrication Tools at Work: Probing Professionals' Current Needs and Desired Futures. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–13, New York, NY, USA, April 2020. Association for Computing Machinery.

[ZHB20] Amy X. Zhang, Grant Hugh, and Michael S. Bernstein. PolicyKit: Building Governance in Online Communities. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, pages 365–378, New York, NY, USA, October 2020. Association for Computing Machinery.

[Zor16] Amit Zoran. A manifest for digital imperfection. *XRDS*, 22(3):22–27, April 2016.

[ZP13] Amit Zoran and Joseph A. Paradiso. FreeD: A Freehand Digital Sculpting Tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2613–2616, New York, NY, USA, 2013. ACM.

Appendix A

Work In Progress: AI Assistance for Machine Repair

In addition to more straightforward calibration routines, I am eager to explore the potential of AI-assisted tools for encouraging care, maintenance, and repair. I have thus far mentioned the use of generative AI tools (particularly large language models (LLMs) that incorporate text, vision, and video) as being associated with one end of Simondon's extremes, that is, the propensity to view technical objects as having a mind of their own, unable to be reworked and redesigned. This is partly because many of these *foundation models*[1] are closed-source, although researchers have been investigating open-source models along with ways to run models locally [Lan24]. However, though LLM's subassemblies can often be difficult to work with, I believe an LLM-based tool could nonetheless encourage people to think about the subassemblies of *other systems*, in our case, digital fabrication machines and the code used to program them.

[1]This term has been used to describe machine learning (ML) that are "general" and massive in scale, with the effect that somebody who wants to use an ML model can fine-tune an existing foundation model, rather than create a new model from scratch [BHA+22]. This paradigm shift carries both opportunities, namely, a much lower threshold to creating working models for a given task, and risks, such as centralizing data and power over data into the hands of a few.

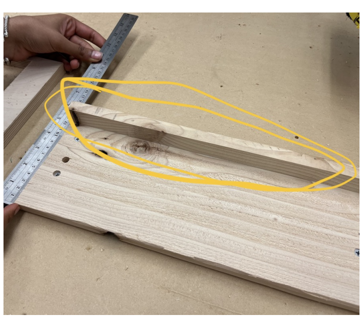
Conversation Input	*Coxswain Example Response*
I am trying to mill the first side of a two-sided milling job of a spoon. However, it appears that the milling went deeper than expected. What went wrong?	To provide an accurate diagnosis, I need to see the result of your milling process. Please send me the photo of your workpiece after milling so I can compare it with the expectation and help identify the issue.
	It appears you have sent the same image twice, showing your expectation for the project. To accurately diagnose the problem, I'll need to see the result after milling. Could you please provide a photo of the actual result of the milling process?
	It appears that the issue with the milling might be due to tool slip. The result shows that the cut is deeper than expected, particularly at the circled area where it has gone completely through the stock, which indicates that the tool may have slipped. This could have caused the tool to cut deeper into the material than programmed. Could you please check the torque on the tool's collet and confirm if the tool has slipped during the milling operation?
	Based on the second image where the milling process missed part of the stock, it looks like the issue may be related to incorrect work coordinate system (WCS) setup or the toolpath boundary settings. The surfacing operation should have covered the entire top surface, but if the toolpath wasn't set to include the edge or if the machine was zeroed incorrectly, it could result in this type of incomplete milling. Additionally, have you double-checked the zeroing of the tool before starting the cut?

Table A.1: Example Conversation Responses. The fourth row shows Coxswain's response to a separate issue than that addressed in the first three rows.

To envision AI for technical mentality—care, maintenance, and repair for fabrication workflows—my collaborator, Ștefania Drugă, and I are building a prototype of a foundation model-based debugging tool for machine breakdowns. This tool, called **Coxswain**, is a conversational assistant that prompts a fabricator to describe the problem in the event of a breakdown; Coxswain then suggests possible issues, calls repair processes for correcting the issues, and prompts the fabricator to investigate further. To support this functionality, Coxswain takes as input data in several modalities from the current state of a machine and its workpiece or material when the breakdown occurs. These data modalities include: image data from one or more cameras, vibration and audio data from accelerometers and microphones, and, most importantly, text input from the fabricator.

As an example, Table A.1 shows how a fabricator (i.e., Siena) interacts with Coxswain when, halfway through milling part of a wooden spoon as described in Section 6.5.2, the CNC mill makes a sudden noise, prompting the fabricator to immediately halt machine operation by pressing the emergency stop button. As Siena inspects the workpiece, she notices a C-shaped gash in the spoon around where the machine was milling tight curves in the toolpath. She loads Coxswain on her computer along with her Tandem notebook and uploads an image of the gash to Coxswain with a text description of what she was doing when the problem occurred (Figure A.1, right). It also provides Siena with buttons; the third, when clicked, opens up Autodesk Fusion 360 and lets her select the AAM toolpath from which the issue arose.

Coxswain analyzes the data that Siena provides, which includes both what she would like to have happen and diagnostic data of what went wrong. It then provides several potential causes for the issue, which in our case it identifies as *tool slip*. Coxswain then explains that tool slip could result from one or more causes: under-tightening the end mill; a Z-axis origin that does not match the physical workpiece, causing the mill to cut too deeply; and overly fast feed rate or speed; a worn end mill; or other factors. Starting with the most likely cause, under-tightening of the end mill, Coxswain calls an overlay function from Tandem's API to highlight where on

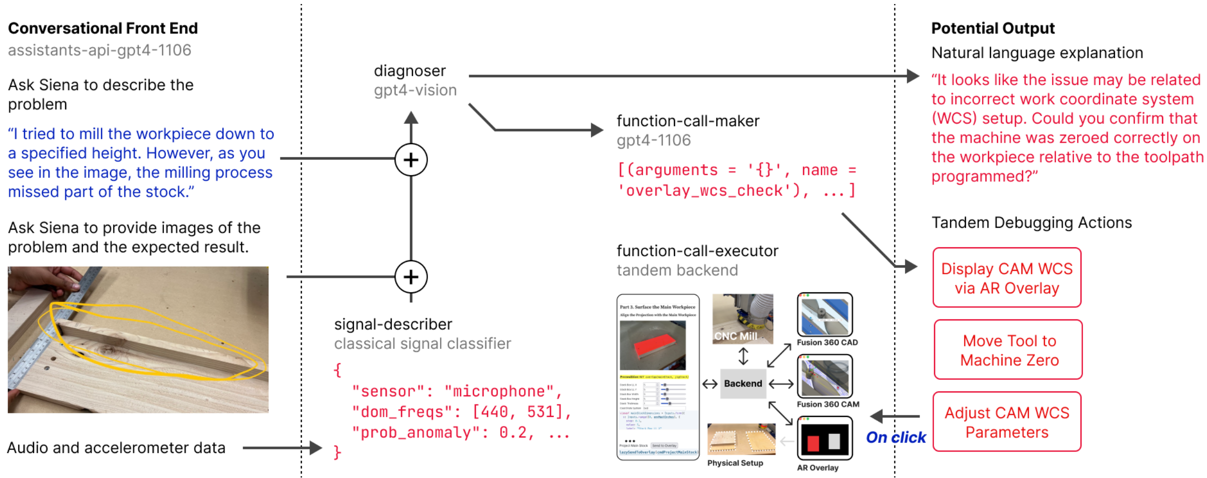


Figure A.1: Coxswain's System Architecture, as a Work-In-Progress. On the left, Coxswain receives text, image, and sensor data that document a breakdown from Siena, an example fabricator. The `signal-describer` (not yet implemented) translates signal data into a JSON description with salient features. The JSON, text, and image data are passed to the `diagnoser`, which outputs a natural language explanation of the issue. The explanation is also passed to the `function-call-maker`, which generates Tandem API calls that automate repair processes for Siena. On clicking these repair options in the conversation window, Tandem executes the action in CAD/CAM, with the AR overlay, or on the CNC mill.

the machine Siena needs to tighten the collet using a pair of wrenches. After making the change, Siena runs her cutting job again; if the problem recurs, she tries other suggestions that Coxswain provided. In future iterations of Coxswain, Ștefania and I plan to build functionality that would let Coxswain use information from multiple failed attempts at repair, as well as data from others' successful repair interventions, to help diagnose and fix the issue.

Beyond diagnosing and repairing errors in the workflow and physically on the machine, Coxswain helps Siena gain greater insight into how breakdowns can culminate from multiple, often obscure causes. Because the system is still an early prototype, I discuss below some salient components and challenges that could evolve over time rather than present an exhaustive description of the system (Figure A.1).

Conversational Interaction. Using a conversational model, fabricators like Siena would have more flexibility to ask for more information, and Coxswain would be able to seek more information or provide multiple approaches. Compared to a fixed set of rules for repair, a conversational style can adapt more easily to unknown situations.

Chaining Multiple Models. For the current implementation, Ștefania and I composed Coxswain from multiple GPT-4 [OAA+23] models that each implement specialized functionality. One major constraint we face is that, at the time of writing, GPT-4 with Vision does not allow API users to fine-tune using images. Thus, though we can fine-tune the `diagnoser` model with text-only input-output pairs, we cannot show the `diagnoser` an image of a machine breakdown alongside a ground-truth diagnosis of the problem. Briefly, comparative visual question-answering (VQA) models such as LlaVa [LLWL23] that do allow fine-tuning with images unfortunately do not have sufficiently high performance to provide sensible outputs on our few-shot dataset.

Fabrication-Specific Knowledge. Besides bona fide fine-tuning, at this early stage of prototyping, we provided Coxswain with a knowledge base of common CNC debugging techniques that we wrote by hand. Specifically, for the `diagnoser` and `function-caller` models, we included contextual information for diagnosing breakdowns and for mapping breakdowns to Tandem calls, respectively. Table A.2 shows part of our small fabrication-specific knowledge base fed to GPT-4 as a system prompt.

Adapting Tandem's API for Repair tasks. Using GPT-4's function calling capability, the `function-caller` translate certain queries into API calls. The Tandem API interacts with CAD/-CAM, the overlay, and the machine directly, guiding Siena.

Once we further develop Coxswain, we intend to run a user study to explore whether and how Coxswain could improve fabricators' attention to repair and maintenance. For an example use case of 3D printing, we would run a between-subjects experiment where participants would debug a buggy machine and associated 3D printing workflow. Participant groups would include a control group (no assistance), a group using current methods (e.g., advice found online, such as r/FixMyPrint), and the experimental group using Coxswain. The experiment would aim to identify whether Coxswain improves repair success rates and whether fabricators who use it know more about the machine. As a more difficult question to ascertain, we would also like information about whether fabricators using Coxswain care more about their machine and workflow; this would require some measure that corresponds to care as an experimental construct.

Beyond reducing the number of errors that occur when following a workflow—through Coxswain or otherwise—bringing maintenance and repair to programming languages marks the beginning of an ethical shift. This approach fundamentally changes the concept of program execution, from "doing things" to instead "working with" the machines, material, and code that we or others have maintained. Understanding this contingency, this hidden semantics of machines, lets fabricator-programmers "support … the *conditions* of production and not just the realization of computational designs" [BP24].

CNC Milling Concept	*Context Prompt*
Surfacing	Surfacing is a toolpath operation where the end mill passes back and forth completely over a surface. Stock material is removed to the depth of the end mill wherever the end mill passes.
Tool Slip	If the tool is not installed in the collet with enough torque, the tool can slip mid cut. This looks like a resulting deep cut in part of the workpiece. It also generates a lot of noise. Tool slip can also result in all cuts being too deep into the workpiece, even though it looks correct otherwise.
Stuck Machine	While the machine could get stuck, it is not likely this will happen without the user noticing noise or visible signs of getting stuck.
Zeroing in the Wrong Place	If a cut starts in an unexpected location, the location at which it was zeroed physically might not match the CAM setup.

Table A.2: Sample from Coxswain's Knowledge Base. We pass a plain text list of CNC milling concepts to the `diagnoser` to form a rudimentary milling-specific knowledge base. Pairs of milling concepts (left column) and elaborations (right) are passed as a message to the model, with the message parameter `role` set to `system` in OpenAI's chat completions API.

Appendix B

Tandem Notebook Implementation

To give an example of a physical-digital program, I include a printout of the two-sided milling notebook described in Chapter 6. The printout shows only the graphical elements that remain after hiding all of the source code in the notebook cells. To view the live version of the notebook, please visit Tandem's project page at: `http://depts.washington.edu/machines/projects/tandem/`. This page contains the most up-to-date links to the notebook and the Tandem backend required to use it.

Two-Sided Milling

Part 0: Setup

Follow the steps below to set up and test all the necessary connections in order to run the notebook:

1. Open the Tandem GitHub Repository on another tab: https://github.com/machineagency/tandem
2. Navigate to the README.md file and follow the steps for installing the backend, AR overlay, and notebook bridge
3. Connect to the physical CNC mill by following the setup instructions for "Physical CNC mill"
4. Turn on the projector and set it up by following the setup instructions for "Projector Setup"
5. Test that the servers are correctly working using the buttons below before proceeding to initial preparations

Test the backend server

Test Backend Server

`null`

Test the overlay projection

```
cmdTestBox0 = ▸Object {name: "Test Box", type: "step", marks: Array(1)}
```

Send a Test Box

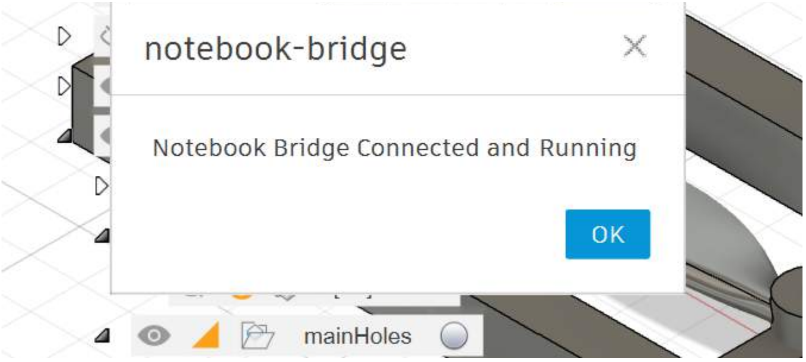
Test the notebook bridge

```
cmdf360_testConnection = ▸Object {name: "Test Notebook Bridge Connection", test_c
```

Test Notebook Bridge Connection

Send to Fusion 360

If you see a message pop up in Fusion 360 as pictured below, then your notebook bridge is running properly! If not, ensure that all steps for setting up the notebook bridge from the README.md were completed.



Only proceed to the remaining parts once you have completed the above steps, or else the notebook will not work as intended.

Part 1: Initial Preparations

Manually Set the XY Zero to the Bed Origin Point

Find the crosshair marked on the bed in the lower left corner. This should be the point that the projector extends the furthest in the lower left corner, near the absolute corner of the bed.

Zero XY

Record and Keep Track of Zeros

```
type CoordinateSystem = 'bed' | 'jig' | 'mainWorkpiece'

currentCoordinateSystem = "mainWorkpiece"
```

Set the Away Position

Record the position where the gantry won't block the projector. Use the button pad, ask user to move it away. Once it's away, query the server for the shopbot's position and record it. Note that this poisition is in bed coordinates and needs to be transformed to a local coordinate system.

Record Current Position as Away

```
null
```

Calibrate the Projector

Follow the steps below to calibrate the projector:

1. Ensure your projector is turned on and setup according to the instructions in Part 0
2. Make sure that your projection is bigger than the work area you want to mill on

2. Make sure that your projection is bigger than the work area you want to mill on
3. Mark the corners of the projected area on your bed
4. Record the width, height, and x and y of the lower left corner of the projected area below using the sliders
5. Click the "Make Calibration Square Appear" box below
6. Move the corners with your mouse so that the corners of the box line up with the corners of the projection
7. Test that your projection works by sending a test square and checking that the dimensions match with the ones you set with a ruler

LL X

5

LL Y

5

Width

5

Height

5

```
cmdCalibrateOverlay = ▸ Object {name: "Calibrate Overlay", type: "calibration", ma
```

Make Calibration Square Appear

```
null
```

```
cmdTestBox = ▸ Object {name: "Test Box", type: "step", marks: Array(1)}
```

Send a Test Box

```
null
```

Set up the Model in F360

1. move the model such that it's entirety is above xy plane.
2. name main model body as **artifact**. This is the body will be milled

The main body is named 'artifact' in F360. ☐

Record the Diameter of the Dowels

Using calipers, measure the diameter of your alignment dowels.

Dowel Diameter (in)

0.35

```
cmdf360_setDowelDiam = ▸Object {name: "Set Outer Dowel Diameter in Fusion 360", c
```

```
cmdf360_createOuter = ▸Object {name: "Create Outer", create_outer: true}
```

Set Outer Dowel Diameter in Fusion 360

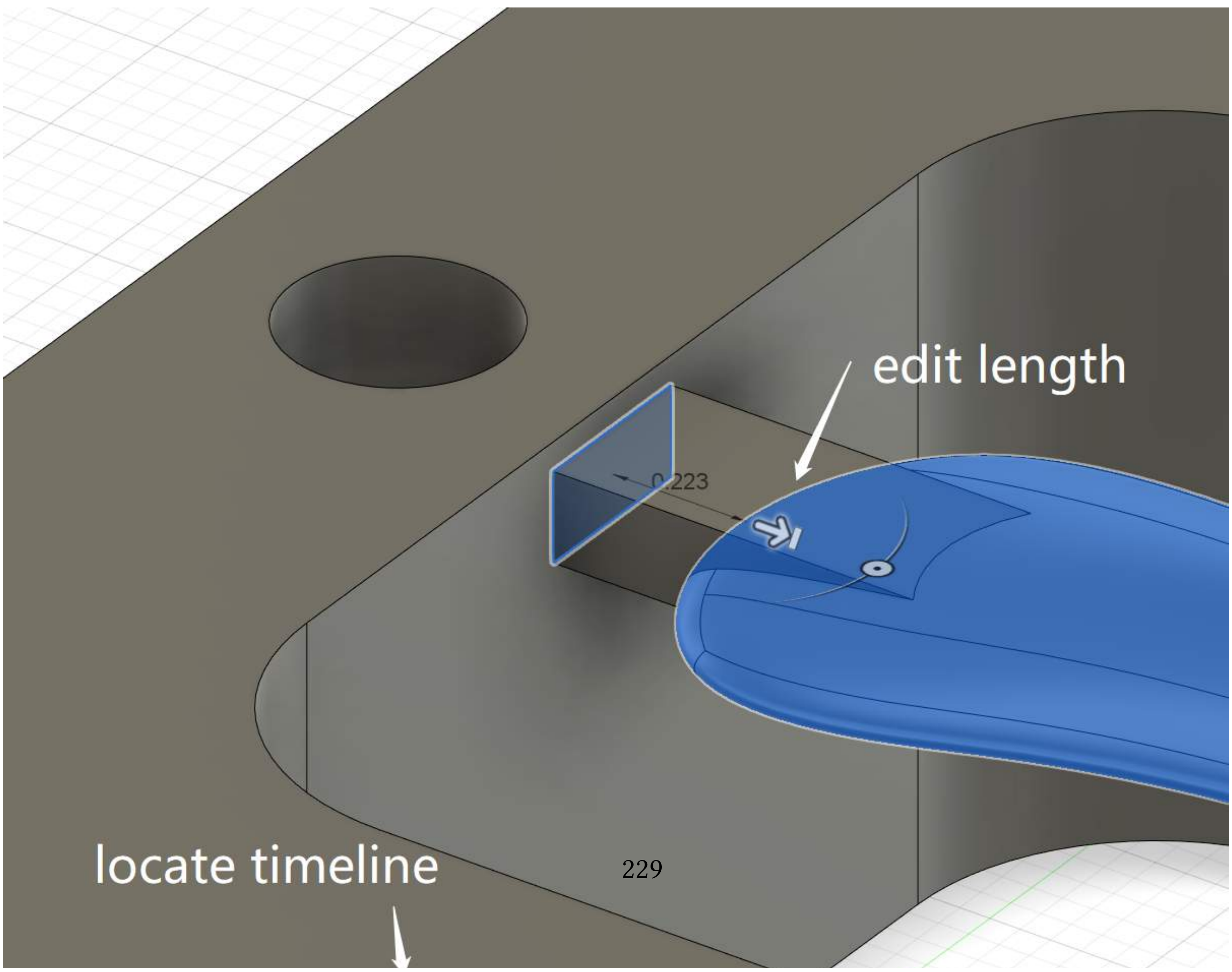
Send to Fusion 360

Create Outer

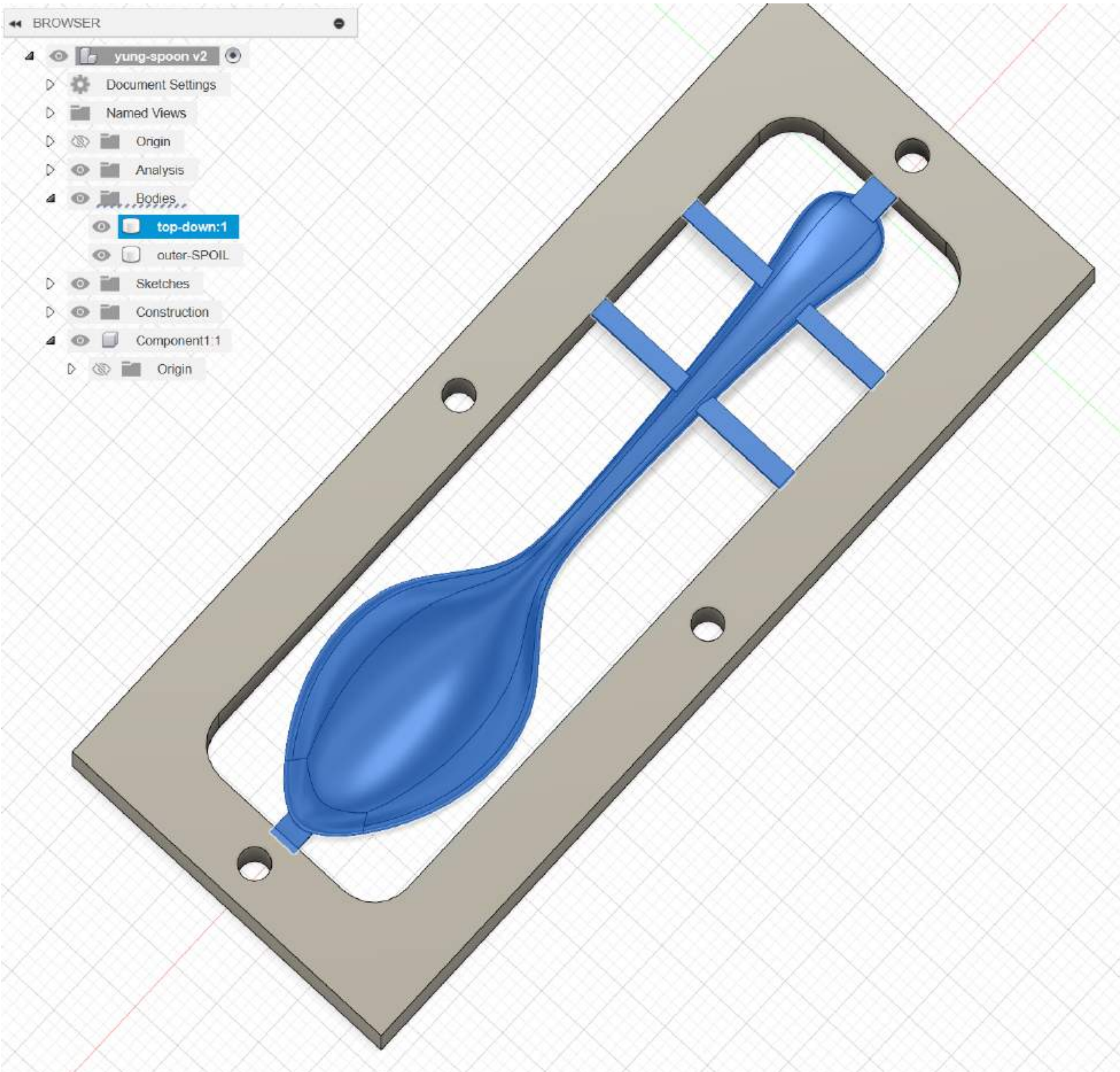
Send to Fusion 360

Manually Adjusting Tabs

At this step, tab supports are generated at a specific distance.Now we ask you to adjust the tab to the specific length, location, so the tabs supports the model you are going to mill. We provide you four default tabs, but feel free to add or remove tabs to suits your need. At the end, please combine all the tabs into body called: "artifact". Be careful that when creating sketch, create from a construction plane instead of from outer, this will make sure that the tab successfully combines with the spoon

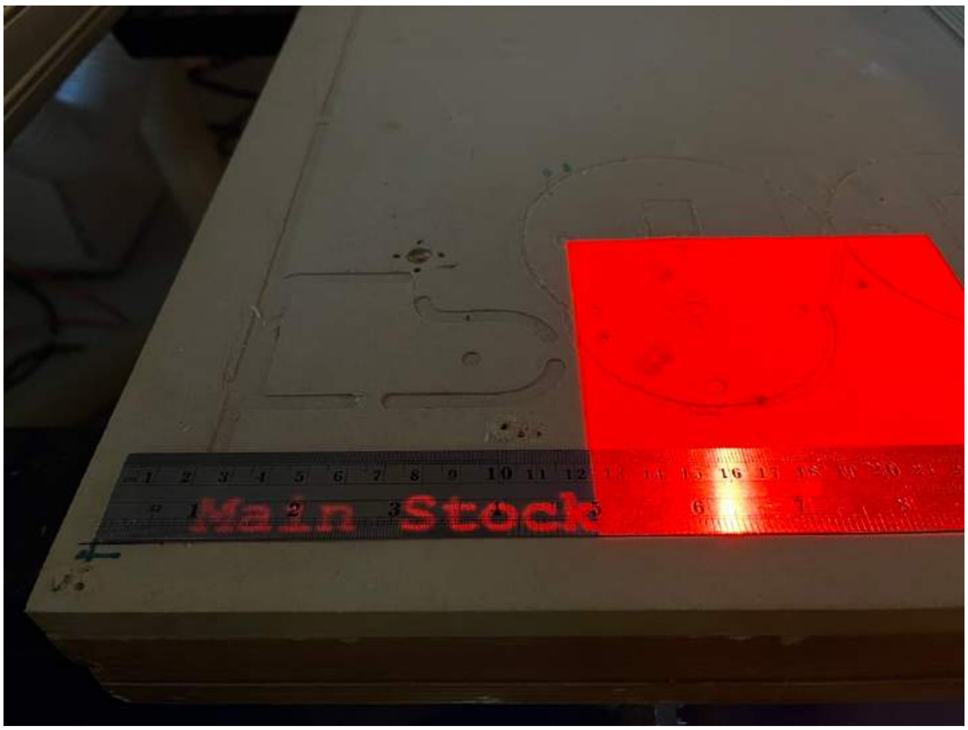


A demonstration of a properly tabbed, and combined model



Part 2: Mill the Alignment Jig

Set the Dimensions for the Alignment Jig Workpiece



Adjust the sliders below based on the dimension of the Alignment Jig Workpiece

Workpiece X

5

Workpiece Y

5

Workpiece Width

5

Workpiece Height

5

W. Thickness

1

231

Coordinate System

bed

```
▶ Object {x: 5, y: 5, width: 5, height: 5, thickness: 1, coordinateSystem: "bed"}
```

Project the Jig Stock and Align Position of Stock with Overlay

```
cmdProjectJigWorkpiece = ▶ Object {name: "Project Jig Workpiece", type: "step", ma
```

Project Jig Workpiece

Send to Overlay

Ensure that the jig workpiece is aligned with the projected box, or adjust the x and y coordinates of the workpiece as necessary.

The jig workpiece is aligned with the projection. ☐

Fixture the Alignment Jig Stock to the Machine Bed with Screws at the Projected Locations

```
cmdProjectJigScrewHoles = ▶ Object {name: "Project Jig Screw Holes", type: "step",
```

Project Jig Screw Holes

Send to Overlay

You will need three Phillips head wood screws with a larger depth than that of the jig stock as well as a drill with a standard Phillips head bit for securing the screws. The fixturing process is as follows:

1. Using the Phillips bit, drill the three screws down into the jig stock where the marks are projected in order to secure the jig stock to the bed. The screw head should be flat with the ReduceThickness of the jig stock.
2. Apply downwards pressure while drilling with your other hand and body weight to ensure that the jig stock is flat on the bed and not lifting off in any areas while screwing. Reference the picture below for how the jig stock should look if it is properly secured to the bed.

Jig workpiece is safely fixtured with three screws. ☐

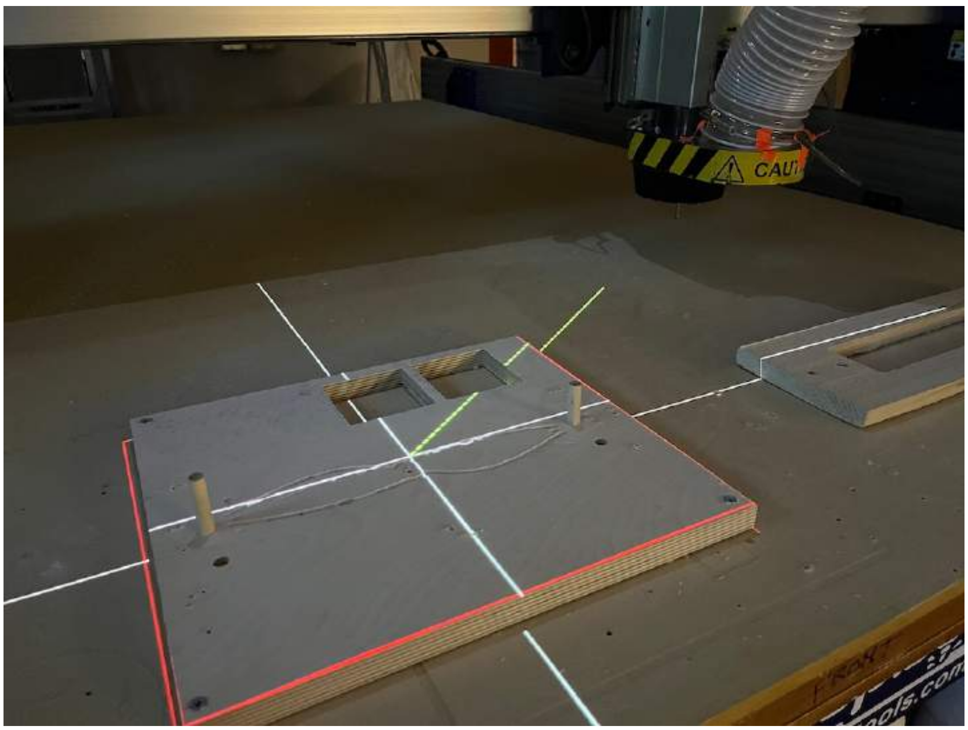
Important: do not move the alignment jig after it's been synchronized with the projection. This is because we'll use the position of the alignment jig to XY zero the mill later on. If you must move the jig after milling, then repeat the projection synchronizing step.

Tool Change: Install the the Non-Surfacing Bit

Check the box below only if you have installed the correct end mill.

The 1/4" flat endmill is installed. ☐

Automatically Set the XY Zero Based on the Projected Box



```
jigZero = ▸Object {x: 7.5, y: 7.5}
```

```
cmdProjectJigZero = ▸Object {name: "Preview Jig XY Zero", type: "step", marks: Ar
```

Preview Jig XY Zero

[Send to Overlay]

Move the bit to the crosshair and zero

[Send to Mill]

Zero the Z Axis

[Send to Mill]

```
tVec_BedToJig = ƒ(v)
```

```
tVec_JigToBed = ƒ(v)
```

Generate the Bore Setup and Instructions in F360 using the recorded Jig Stock Dimensions

```
cmdf360_setjigWorkpieceDimensions = ▸ Object {name: "Set Jig Stock Dimensions in F
```

Set Jig Stock Dimensions in F360

Send to Fusion 360

```
cmdf360_generateJigSetup = ▸ Object {name: "Make Alignment Jig Setup", setupCam: A
```

Make Alignment Jig Setup

Send to Fusion 360

The zero in CAM matches the physical zero. ☐

Generate Machine Instructions and Dispatch them to the Mill

Maybe todo: generate a profile cut to cut out the entire jig if the user needs that.

```
cmdf360_instructionsJig = ▸ Object {name: "Generate Jig Machine Code", exportSbp:
```

Generate Jig Machine Code

Send to Fusion 360

Wait 5 seconds after generating the code, then press the following button to refresh.

Refresh Jig Bore Machine Code

No code yet.

Preview the Jig Dowel Locations

```
cmdProjectJigBoreToolpath = ▸ Object {name: "Preview Jig Bore Locations", type: "s
```

No Away Position

Preview Jig Bore Locations

Send to Overlay

Preconditions for "Mill Alignment Jig"

☐ The 1/4 inch flat end mill is installed

- [ ] The 1/4-inch flat end mill is installed
- [ ] Diameter sent to Fusion 360
- [ ] CAM setup generated
- [ ] Alignment workpiece jig fixtured to the bed
- [ ] CAM zero matches physical zero

Send to Mill

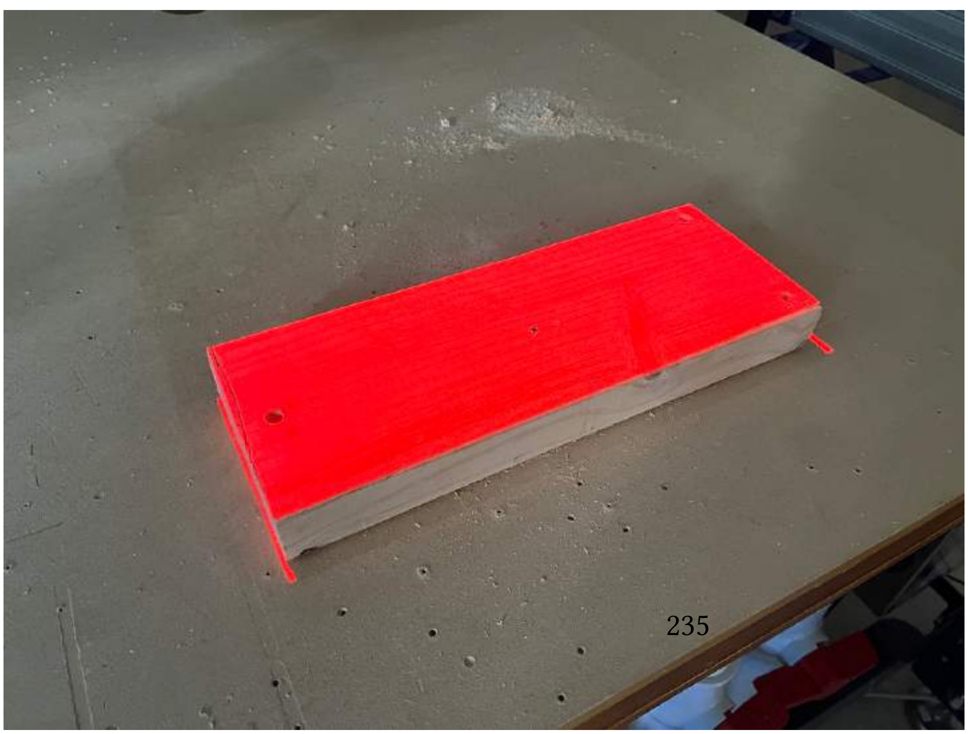
Nothing sent to the mill yet.

Check the Fit of the Bores with the Dowels and Correct if Needed

1. Ensure that the dowels fit tightly into the holes and do not come out easily if pulled by hand. You may need to use a mallet in order to get the dowels in securely.
2. If the holes are too small for the dowel, remeasure the diameter of the dowels and follow the steps above to re-bore the holes to the correct size.
3. If the holes are too large for the dowel, zero the jig-y to be slightly lower or higher than the previous holes. Then, remeasure the diameter of the dowels and follow the steps above again in order to re-bore the holes to the correct size.
4. Once you are finished with this step, the dowels should be tightly secured to the jig stock

Part 3. Reduce the Thickness of the Main Workpiece

Align the Projection with the Main Workpiece



Stock Box LL X

5

Stock Box LL Y

5

Stock Box Width

5

Stock Box Height

5

Stock Thickness

1

Coordinate System

bed

▸ Object {x: 5, y: 5, width: 5, height: 5, thickness: 1, coordinateSystem: "bed"}

cmdProjectmainWorkpiece = ▸ Object {name: "Project Main Stock", type: "step", mark

Project Main Stock

Send to Overlay

Ensure that the main workpiece is aligned with the projected box, or adjust the x and y coordinates of the workpiece as necessary.

The main workpiece is aligned with the projection. ☐

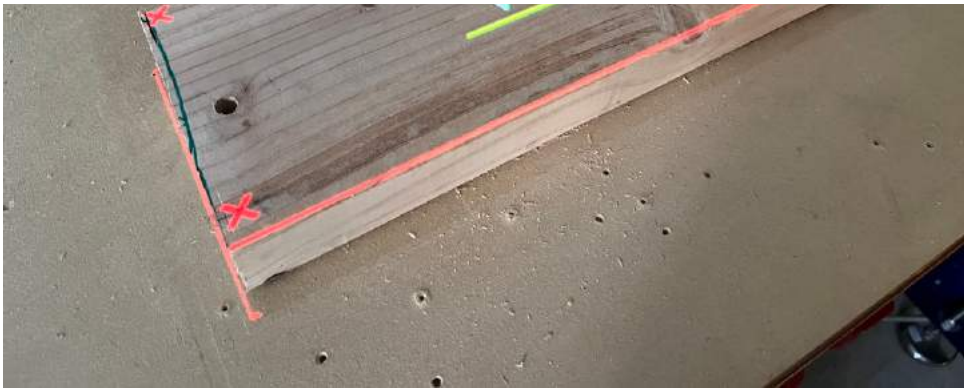
Send Main Stock Info to Fusion 360

cmdf360_setmainWorkpieceDimensions = ▸ Object {name: "Set Main Stock Dimensions ir

Set Main Stock Dimensions in F360

Send to Fusion 360

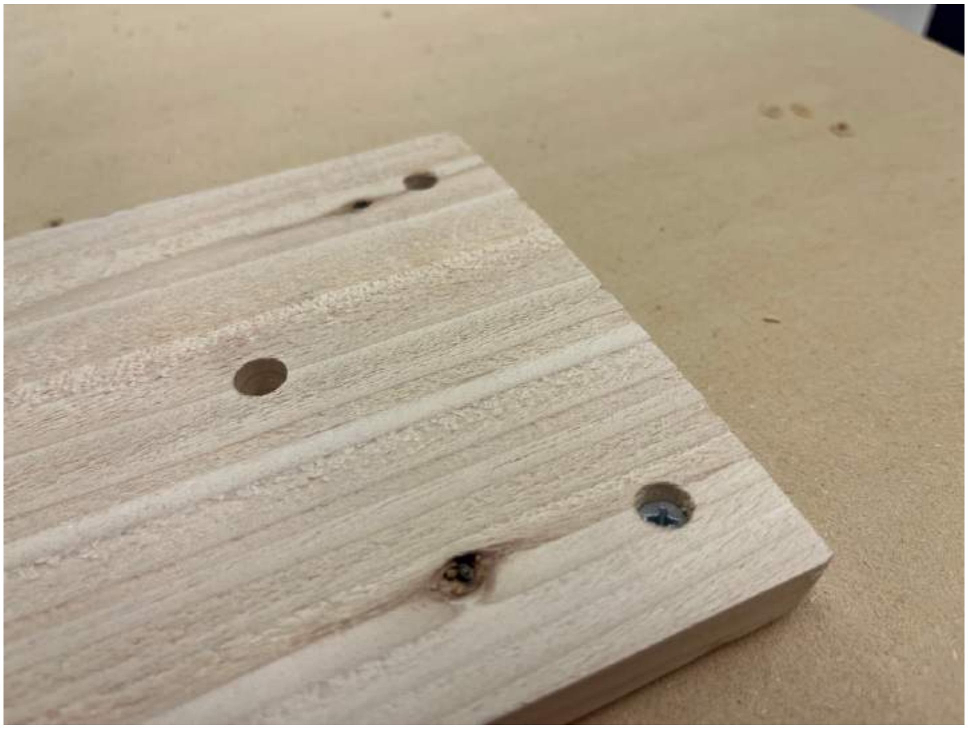
Fixture the Main Workpiece with Screws at a Safe Depth



```
cmdProjectMainScrewHoles = ▸Object {name: "Project Main Screw Holes", type: "step
```

Project Main Screw Holes
Send to Overlay

```
modelHeight = 0.5
```



You will need a drill with three bits: a standard Phillips head bit for securing the screw, a countersink bit that matches the screw, and a boring bit that is at least as wide as the thickest part of the countersink bit. The fixturing process is as follows:

1. Using the boring bit, drill a hole that is at least **0.5 in** inches deep. This is calculated as follows: stock thickness (**1 in**) minus model height (**0.5 in**).
2. Manually check all bore depths with the depth probe of a pair of calipers. Ensure the number reads **0.5 in** or greater.
3. Change to the countersink bit and countersink all the bores.

4. Change to the Phillips bit and screw in all the screws.
5. Check again with the calipers that the depth reading of all bores reads **0.5 in** or greater.

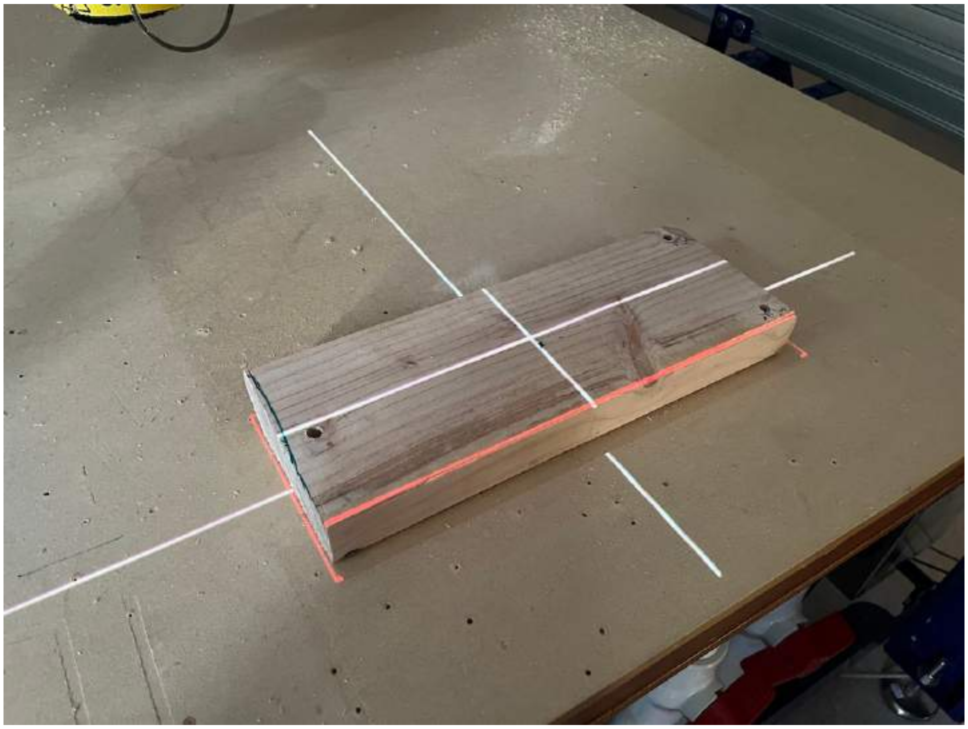
```
safeFixtureDepth = 0.5
```

Main workpiece is safely fixtured with screws at 0.5 or more. ☐

Tool Change: Install the Surfacing Bit

The surfacing bit is now installed. ☐

XY Zero the Mill on the Main Workpiece




```
mainWorkpieceZero = ▸Object {x: 7.5, y: 7.5, coordinateSystem: "bed"}
```

```
cmdProjectmainWorkpieceZero = ▸Object {name: "Preview Main Zero", type: "step", m
```

Preview Main Zero

Send to Overlay

Move to Main Stock Center and Set XY Zero

Send to Mill

```
▸Object {x: 0, y: 0, coordinateSystem: "mainWorkpiece"}
```

```
tVec_BedTomainWorkpiece = ƒ(v)
```

```
tVec_mainWorkpieceToBed = ƒ(v)
```

Z Zero the Mill

Zero the Z-Axis with the Z-Plate

Send to Mill

Generate the Surfacing Setup in F360

```
cmdf360_generateSurfacingSetup = ▸Object {name: "Make Surfacing Setup", setupCam:
```

Make Surfacing Setup

Send to Fusion 360

Generate the Surfacing Toolpath in F360

```
cmdf360_instructionsReduceThickness = ▸Object {name: "Generate Surfacing Machine
```

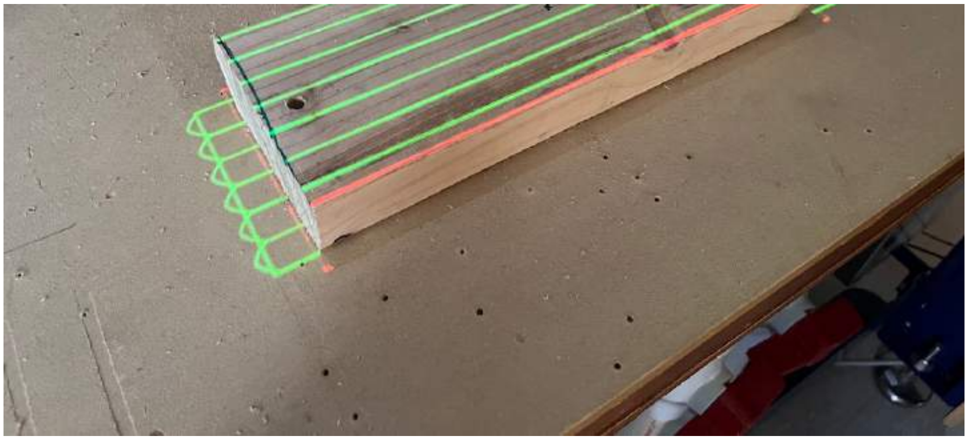
Generate Surfacing Machine Code

Send to Fusion 360

Fetch instructions for reduceThickness

No code yet.

Project the Surfacing Toolpath with Respect to the Main Stock



```
cmdProjectSurfacingToolpath = ▸Object {name: "Preview Surfacing Toolpath", type:
```

Preview Surfacing Toolpath
Send to Overlay

Check that the surfacing fully covers the workpiece. If not you might end up with the result in the image below which is a result of the surfacing toolpath being zeroed at the wrong place, or of the stock setup being wrong in F360. If you change the latter, you need to repeat the main stock localization step.

Dispatch the Surfacing Cut

Preconditions for "Surface Main Workpiece"

- [x] Main workpiece exists and has non-zero dimensions
- [ ] Main stock dimensions sent to Fusion 360
- [x] Z axis zeroed with z plate
- [x] Main stock xy zero set on mill
- [ ] Main stock safely fixtured to bed with appropriate screw depth
- [ ] CAM setup generated

Dispatch Surfacing Cut

Send to Mill

Part 4. Mill Alignment Holes in the Main Workpiece

Tool Change: Install the Main Bit

Z Zero the Mill

Note: do not XY-zero again, unless something was wrong with the last XY zero for surfacing.

Zero the Z-Axis with the Z-Plate

Send to Mill

Generate the Bore Setup and Instructions in F360

```
cmdf360_mainHoles = ▸ Object {name: "Generate Main Holes Setup", setupCam: Array(1
```

Generate Main Holes Setup

Send to Fusion 360

```
cmdf360_instructionsMainBore = ▸ Object {name: "Generate Main Bore Machine Code",
```


Generate Main Bore Machine Code

Send to Fusion 360

Fetch instructions for mainHoles

No code yet.

Preview the Bore Locations

```
cmdProjectMainBoreToolpath = ▸Object {name: "Preview Main Bore Locations", type:
```

No Away Position

Preview Main Bore Locations

Send to Overlay

Dispatch the Bore Cut

Preconditions for "Bore Main Workpiece"

- [x] Z axis zeroed with z plate
- [ ] CAM setup generated

Dispatch Main Bore Cut

Send to Mill

Check the Depth and Fit of the Bores

```
cmdProjectBoreDepth = ▸Object {name: "Project Bore Depth", type: "step", marks: A
```

Project Bore Depth

Send to Overlay

1. Ensure that the depth of the bores is equal to the depth of the stock before removing and unscrewing the stock from the bed. This means that the bore should have drilled holes all the way through the stock. You can measure both the depth of the holes and the depth of the stock with calipers in order to ensure this.
2. If the holes are not deep enough, adjust the depth of the stock and redo the steps above to re-bore the holes.

Part 5. Mill the Top-Down Cut

Fixture the Main Stock to the Jig with Screws at the Projected Locations

```
cmdProjectTopCutScrewHoles = ▸Object {name: "Project Top Cut Screw Holes", type:
```

Project Top Cut Screw Holes

Send to Overlay

1. Align the stock holes with the dowels and secure so that the stock is flat against the top of the jig
2. Using a drill bit with a standard Phillips head and 3 wood screws that are longer than the depth of the stock, secure the stock to the jig at the projected locations

Main workpiece is safely fixtured with screws ☐

Generate the TopCut Setup and Instructions in F360

```
cmdf360_topDown = ▸Object {name: "Generate Top Cut Setup ", setupCam: Array(1)}
```

Generate Top Cut Setup

Send to Fusion 360

```
cmdf360_exportTopCutsbp = ▸Object {name: "Generate Top Cut Machine Code", exportS
```

Generate Top Cut Machine Code

Send to Fusion 360

Fetch instructions for topDown

No code yet.


```
cmdProjectTopCutToolpath = ▸Object {name: "Preview Top Cut Toolpath", type: "step
```

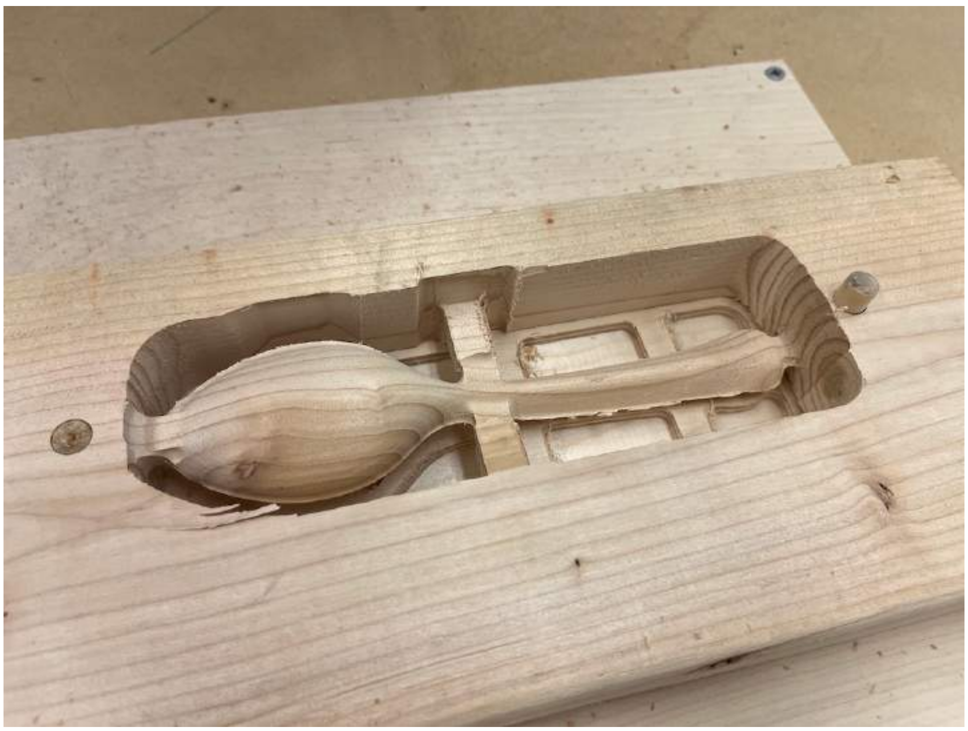
Preview Top Cut Toolpath

Send to Overlay

Preconditions for "Mill Top-Down Cut"

- [ ] Main workpiece safely fixtured to jig
- [ ] CAM setup generated

Part 6. Mill the Bottom-Up Cut



Flip the Main Stock Over the X-Axis To Prepare for the Bottom-Up Cut

```
cmdProjectFlipStock = ▸Object {name: "Project Flip Stock", type: "step", marks: A
```

Project Flip Stock

Send to Overlay

Instructions for flipping the stock....

```
cmdProjectBottomCutScrewHoles = ▸Object {name: "Project Bottom Cut Screw Holes",
```

Project Bottom Cut Screw Holes

Send to Overlay


```
cmdf360_bottomDown = ▸Object {name: "Generate Bottom Cut Setup", setupCam: Array(
```

Generate Bottom Cut Setup

Send to Fusion 360

```
cmdf360_exportBottomCutsbp = ▸Object {name: "Generate Bottom Cut Machine Code", e
```

Generate Bottom Cut Machine Code

Send to Fusion 360

Utility Functions

```
envMaxXInches = 50

envMaxYInches = 24

maxThickness = 3

zSafeHeight = 2

eagerSendToOverlay = ƒ(command)

effect_sendToOverlay = ƒ(command)

effect_sendToF360 = ƒ(command)

effect_sendToMill = ƒ(command)

moveAway = ƒ()

tVec_current = ƒ(coords)

fetchInstructions = ƒ(setupName)

__sendInstructionsToMill = ƒ(instructions)

highlightPrecondition = ƒ(pc)

highlightPostcondition = ƒ(pc)

highlightPictureNeeded = ƒ(pc)
```



```
fetchSvg = async ƒ()
```

```
requires = ƒ(commandName, textPreconditionPairs)
```