Improving Programming for Exploratory Digital Fabrication with Inline Machine Control and Styled Toolpath Visualizations

Jasper Tran O'Leary
jaspero@cs.washington.edu
University of Washington
Seattle, Washington, USA

Eunice Jun
emjun@cs.washington.edu
University of Washington
Seattle, Washington, USA

Nadya Peek
nadya@uw.edu
University of Washington
Seattle, Washington, USA

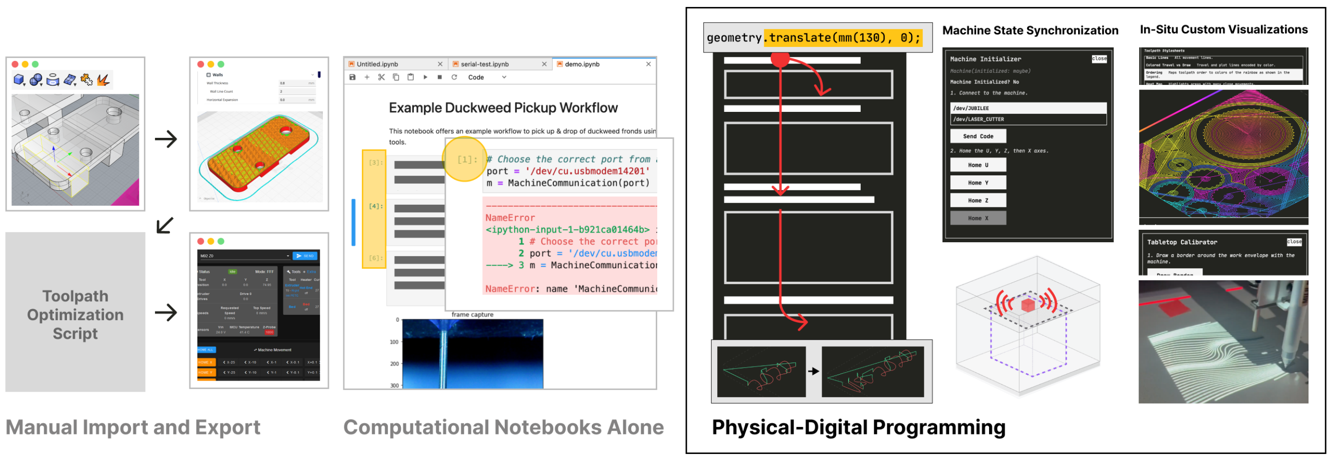

Figure 1: Programming Approaches for Exploratory Digital Fabrication. Left) workflows are commonly constructed by painstakingly importing and exporting files between discrete GUIs and scripts. Middle) computational notebooks allow more flexible composition, but require additional synchronization between code, machine configuration, and real-world outputs. Right) Verso provides a live programming environment (left), within-code GUIs that synchronize machine state with values in code (middle), and generation of custom visualizations projected onto the corresponding physical locations (right).

ABSTRACT

Makers from increasingly diverse backgrounds use digital fabrication machines to explore novel design spaces. However, software tools for fabrication are designed primarily for replication-based tasks; programming machines for bespoke applications while accounting for physical contingencies remains challenging. To better support exploratory fabrication, we present Verso, a proof-of-concept approach that extends concepts from computational notebooks. Verso affords graphical control via *modules* that result in continuous feedback, letting makers fluidly view and iterate on their workflows. Modules also provide controlled synchronization between code and external digital and physical processes while preserving a clear flow of data. *Toolpath stylesheets (TSS)* translate machine instructions into task-specific visualizations that can be projected into the machine's physical work space. To demonstrate our approach, we synthesize three design goals and propose three example exploratory workflows for subtractive manufacturing, materials science, and biology.

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools**; • **Applied computing → Computer-aided manufacturing**; • **Software and its engineering → Domain specific languages**.

KEYWORDS

Exploratory digital fabrication, computational notebooks, physical-digital programming

ACM Reference Format:
Jasper Tran O'Leary, Eunice Jun, and Nadya Peek. 2022. Improving Programming for Exploratory Digital Fabrication with Inline Machine Control and Styled Toolpath Visualizations. In *Symposium on Computational Fabrication (SCF '22), October 26–28, 2022, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3559400.3561998

1 INTRODUCTION

Digital fabrication machines are machines that can be programmed to create physical objects. Increasingly, makers from diverse backgrounds are pushing the limits of where these machines can be applied. Artists explore new sketching techniques with drawing

SCF '22, October 26–28, 2022, Seattle, WA, USA

ACM ISBN 978-1-4503-9872-5/22/10.
https://doi.org/10.1145/3559400.3561998

machines called *plotters* [He and Adar 2020; Yasu 2017], carve molds using computer numerical control (CNC) mills, and even create data-driven ceramics with clay 3D printers [Wasp 2019]. Materials scientists explore novel material structures through controlled hydrogel deposition using CNC machines [Chatterjee et al. 2019; Rivera et al. 2020; Wang et al. 2016]. Biologists pioneer automated high-throughput experimentation using robotic pipetting and imaging tools [Ouyang et al. 2021]. All these scenarios require processing data from heterogeneous sources, fine-grained control and customization of machine movements, and a way to test and improve workflows.

Prior digital fabrication practice and research generally focuses on *replication-based workflows*, whose goal is to faithfully replicate a digital model [Baudisch and Mueller 2017]. However, recent findings show that makers in emerging domains build bespoke, application-specific workflows, often from the ground-up [Twigg-Smith et al. 2021]. They prioritize self-guided exploration over replication. Exploration gives rise to iteration, which requires increased control over machine motion and input data sources. These makers are already *programming* novel workflows, but often do not have programming tools suited for their applications.

For example, an artist milling a mold might want to add features, but must quickly preview any additional machine movements in situ with the existing cuts in the chosen material. A materials scientist may want to author machine movements at the instruction level to control deposition of novel materials with untested physical properties. A plant biologist may need to integrate camera data to help a robot locate and move samples. These scenarios contrast sharply with the present prevailing goal in digital fabrication, i.e., minimizing error and replicating idealized outputs. Instead, they highlight the need for makers to explore cutting-edge design spaces, including the challenges of working with the material (e.g., wood, hydrogel, plant matter).

We term this paradigm *exploratory digital fabrication (EDF)*, which champions programmability, inspectability, and iteration with the end result that the practitioner arrives at a more holistic understanding of the design space of the fabrication task. EDF workflows commonly involve different challenges than those found in replication-based fabrication, including instruction-level manipulation, managing and processing sensor data, and adapting to physical landmarks.

1.1 Why is Exploratory Fabrication Difficult?

Currently, makers create and manage workflows through two main approaches:

(1) Most commonly, **manual export and import** of intermediate files between GUI-based applications for manipulating data (e.g., Cura or machine-specific software suites).
(2) More rarely, centralizing code in **computational notebooks** (e.g., Jupyter notebooks).

Manual export and import works well for replication-based workflows, but poses major issues for EDF. Namely, it is difficult to test and iterate upon workflows as makers explore the design space. Makers changing one part of the workflow—for example, editing a model's geometry in a CAD program—must manually import and export back down the entire chain of software tools to propagate the change. Makers cannot quickly and in real time visualize the result of changes in their workflow before committing to them. They must go through a series of time-consuming steps after each iteration. Similarly, managing workflows across separate programs can lead to inconsistent state. Makers updating and exporting new files must remember to manually import the new file and re-export the remaining intermediate files, which may require re-running and even re-writing postprocessing scripts. The length and complexity of workflows create additional effort a maker must exert to ensure consistency.

On the other hand, computational notebooks let makers coordinate several tasks, for example, connecting to, calibrating, and running the machine, all within a single programming environment. This allows for increased coordination between different steps of a workflow. However, notebooks are designed primarily for digital-only data tasks and pose challenges for working with physical data. Working with fabrication machines calls for persistent recalibration, modifying program behavior as needed to account for physical realities. Unlike many standalone machine-specific GUI tools, code in computational notebooks is divorced from physical state. In addition, many tasks with EDF are best suited for graphical input and direct manipulation, yet such input in notebooks remains largely out of reach. Instead, makers must both visualize machine behavior and control machines with code alone.

1.2 How Can We Improve Programming for EDF Workflows?

We argue for extending concepts from computational notebooks to better accommodate EDF. We propose **three design goals** to build on the foundation that notebooks provide:

(1) synchronize program state with physical realities,
(2) allow graphical input without sacrificing the flexibility of code, and
(3) visualize different views of machine behavior physically in-situ.

To test these goals, we built Verso, a proof-of-concept implementation in a prototypical notebook environment. Verso lets makers edit source code as the programming environment nearly immediately reflects changes in the program's output. Makers can program and share *modules*, which are GUIs embedded inline into the workflow's source code that input and output data from the rest of the program. Verso modules can read and write data to external applications, sensors, and machines in a structured manner; modules thus provide a layer of abstraction between data flow and communication with the physical world.

Verso also facilitates rapid previewing of workflow output via programmable and shareable *toolpath stylesheets* (TSS), which translate a final set of machine instructions into a visualization that is tailored for a particular application. Verso programs, modules, and TSS code can be shared and remixed with the maker community. Using Fogarty [2017]'s language, Verso sketches novel functionality (i.e., unified control over heterogeneous data sources and steps in one live program) through a combination of existing and novel techniques.

To demonstrate Verso, we authored three example workflows based on ongoing collaborations with three makers building experimental digital fabrication workflows. In each example, we contribute initial ideas for workflow code, modules, and TSS that could be helpful to makers in the given scenarios. These examples demonstrate how Verso efficiently represents and permits iteration of experimental workflows across a range of machines and scenarios.

To summarize, we contribute:

- A new workflows-as-live-programs paradigm for experimental digital fabrication
- A proof-of-concept programming environment for authoring workflows using code and embedded user-defined GUIs to encapsulate input/output to physical machines
- An approach for making custom visualizations (TSS) to test workflows
- Three demonstrations that illustrate how the workflows-as-live-programs approach might help makers explore fabrication techniques more efficiently than is currently possible.

2 RELATED WORK

Verso draws upon prior insights about how makers use fabrication machines to explore, challenges they face when digitally controlling physical machines, and ideas from programming environments for other domains.

2.1 Exploratory Digital Fabrication

The notion of exploratory fabrication stems from several related ideas in HCI research. Namely, Kim et al. introduce the concept of fabrication middleware in their work on *compositional fabrication*, where a machine behaves like an audio mixer and reflects changes to parameters in realtime [Kim et al. 2018]. They argue that this extends Willis et al.'s notion of *interactive fabrication*, where an interactive, but fixed, correspondence exists regarding how a machine reacts to maker input [Willis et al. 2011]. We aim to realize this vision of composition at a robust level with visualizable live programming that supports feedback from the physical world.

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 [Ou et al. 2016]. Defextiles leverage under-extrusion of plastic filament ("defects") on unmodified machines to create textile patterns useful for rapidly prototyping fashion designs, for example [Forman et al. 2020]. 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 [Twigg-Smith et al. 2021]. Additionally, Li et al. [Li et al. 2020] 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

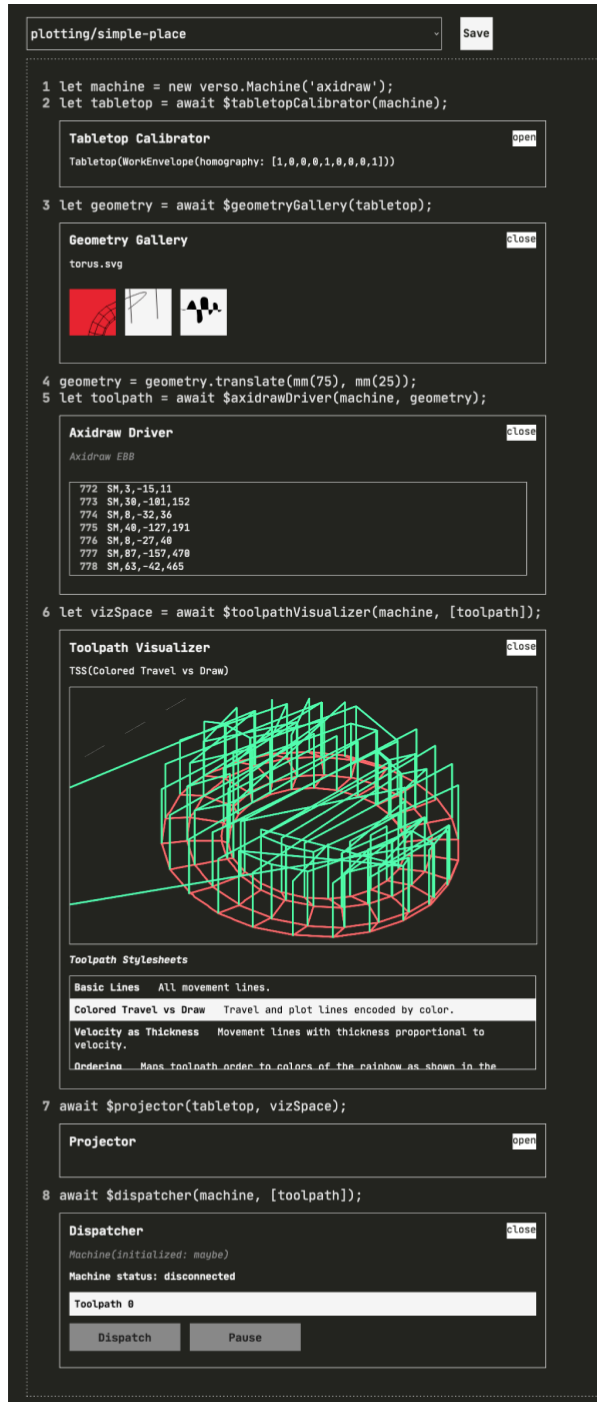

Figure 2: Plotter workflow demonstrates Verso's interface. Makers load an existing workflow from the dropdown menu at at the top or create a new workflow. This workflow shows the plotting of a torus with an Axidraw machine. While writing code in the editor, makers can add calls to module functions that generate associated embedded GUIs. By default, these GUIs are visible, but makers can minimize them to enhance their ability to skim workflow code.

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.

2.2 Challenges with Exploratory Fabrication

A key challenge for professionals [Yildirim et al. 2020], hobbyists [Li et al. 2020; Twigg-Smith et al. 2021], and students [Hudson et al. 2016] who use digital fabrication machines is connecting digital tools to their physical counterparts. For instance, Yildrim et al. [Yildirim et al. 2020] identify that among the current challenges manufacturing professionals face is how the "perfection of software would break down in the translation to real materials and tools," requiring makers to take copious notes of failures, source materials differently, and seek previews of their fabrication intents before committing them to production.

Researchers have introduced techniques to address this challenge. 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 [Dogan et al. 2021]. 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 [Gulay and Lucero 2019] introduce "integrated workflows," which are similar to Tian et al.'s vision of "lucid" fabrication workflows that blend the digital and physical [Tian et al. 2019]. Fossdal et al. extend CAD environments to encourage toolpath and material exploration [Fossdal et al. 2021]. Focusing on machine operations rather than workflows, Taxon provides a way to represent any fabrication machine and reason about valid operations for it [Tran O'Leary et al. 2021]. Additionally, prior work has investigated the role of augmented reality in digital fabrication, for example, copyCAD [Follmer et al. 2010], SPATA [Weichel et al. 2015a], ReForm [Weichel et al. 2015b], MixFab [Weichel et al. 2014], AdapTutAR [Huang et al. 2021], and Mahapatra et al.'s investigation into barriers to augmented fabrication [Mahapatra et al. 2019].

While prior work has focused on supporting specific fabrication interactions and techniques, Verso raises the level of abstraction from a specific machine and workflow to workflows more generally. In this way, Verso complements existing research on ways to more smoothly support transitions between digital and physical control in digital fabrication.

2.3 Programming Exploratory Workflows

We discuss existing techniques beyond manual import and export for programming exploratory workflows. We draw inspiration from graphical tools for programming, as well as improvements to computational notebooks for data science tasks. Verso seeks to combine benefits from each approach to provide a computational environment with increased physical-digital synchronization.

2.3.1 Graphical Programming Tools. Live programming envisions a programming environment that provides coders with continuous feedback as they author a program [Tanimoto 1990]. Programming environments that support liveness can tighten the feedback loop between code and its effects, enabling faster iteration and debugging. Peek and Gershenfeld presented Mods, a live, graphical interface for composing common-case digital fabrication workflows [Peek and Gershenfeld 2018]. Verso extends Mods' concept by embedding graphical interfaces within a larger code-based programming environment and by letting makers customize visualizations.Omar et al. introduced liveness in programs through user-defined GUIs called livelits [Omar et al. 2021]. End-users invoke a livelit they previously defined by calling the function in a program, which introduces a GUI in-line. Similarly, Verso provides *modules* which allow for live graphical input for code, but which mediate input to and output from physical machines.

Moreover, program visualizations that describe a program's state increase code understanding and debugging [Hoffswell et al. 2018]. Prior work has investigated application debugging through targeted inspection [Burg et al. 2013] and always-on visualizations [Kang and Guo 2017; Lieber et al. 2014]; many of these contributions influenced the design of state-of-the-art web developer tools. Verso extends such visualization and interactive debugging techniques to digital-physical workflows.

Purely graphical systems for live programming with sensor input, e.g., Pure Data [Puckette 2022] and LabView [Bitter et al. 2006], let programmers build interactions around sound and instrumented testing respectively within a flow-based programming graphical editor. However, their graphical-only nature precludes building more complex workflows outside of their target applications. Verso seeks to integrate the benefits of graphical control and live sensor control within a textual programming environment.

2.3.2 Computational Notebooks for Data Science. Computational notebooks, like Jupyter Notebook [Kluyver et al. 2016], Databricks [Lakehouse 2022], and Google Colab [Bisong 2019], let programmers interleave code, data, and visualizations as *cells* within a single programming environment. They are intended for tasks that involve purely digital data manipulation, as opposed to the physical-digital data processing required in exploratory digital fabrication. Nonetheless, many computational notebook implementations present significant pain points to data scientists, for example, loading data from multiple (digital) sources and re-running code after tweaking parameters [Chattopadhyay et al. 2020]. Coordinating code with physical machines and materials exacerbates this issues; naively sending and receiving data from the physical world can lead to inconsistent program state and unsafe machine execution.

One particular notebook that addresses some of these issues is Observable [Bostock 2018] which treats notebook cells as a graph and re-computes any child cells when a cell is modified. It also supports rudimentary graphical inputs that modify the value of a cell in real time [Bostock 2022].

Building on this idea, the mage [Kery et al. 2020] system supports graphical input in computational notebooks by directly manipulating visualizations and synthesizing resulting code changes. Like mage, B2 [Wu et al. 2020] supports tighter integration of code and interactive data visualizations in Jupyter notebooks. A key insight in B2 is that manipulations to data frames are queries that can be used to generate visualizations, just as interactions with visualizations are also query operations that can update and impact analysis code. Sketch-and-sketch also supports bidirectional authoring of vector graphics (SVGs) via code and direct manipulation [Hempel et al. 2019]. Glinda combines live programming with multimodal program authoring in a domain-specific language for data science workflows [DeLine 2021]. Symphony provides interactive components for machine learning tasks, such as confusion matrix and and 3D path visualizations, that can be included in notebooks alongside reports and dashboards simultaneously [Bäuerle et al. 2022]. Verso combines live programming with graphical input through modules and applies these techniques to a new domain, digital fabrication. Our goal with Verso is to apply these techniques to previewing, writing to, and reading data from the physical world.

Finally, p5.fab provides low-level 3D printer control from a Javascript environment, letting users experiment with material properties using unconventional printing processes [Subbaraman and Peek 2022]. Verso's aims to build on the programmatic control exemplified in p5.fab by providing abstractions around machine calibration, control, and phyical previewing of execution.

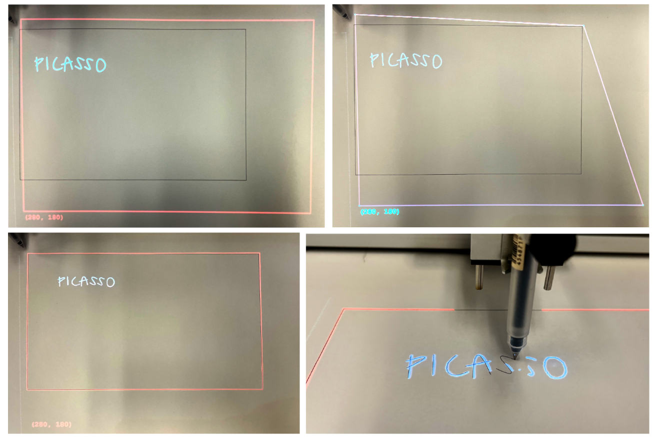

Figure 3: Calibrating visualizations for a physical machine. 2D projections of a visualization do not initially match the machine's coordinate space. Using the TabletopCalibrator module (not pictured), the plotter draws a ground truth bounding box (upper left). The maker uses the mouse to interactively draw the projected bounding box to match the physical one (upper right). Now that 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).

3 VERSO SYSTEM OVERVIEW

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 where makers write workflow code in Javascript, a language accessible to many makers. As the maker types, Verso listens for input events and continuously re-evaluates the maker's code. Verso provides a library of classes for representing common concepts in digital fabrication such as `Machine` and `Geometry`.

In addition to raw code, makers 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 maker types in a module function, Verso generates the module's GUI immediately below the program line containing the module function. The maker can now 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, 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 maker 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 would occur, makers 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. Makers use the toolpathVisualizer module to select which TSS they would like to use for the set of instructions currently passed into the module function. We currently implement the visualizations generated by TSS in Three.js [Cabello 2014]. Altogether, changes to the workflow code and to modules results in near-immediate visual feedback via TSS. TSS are further described in Section 5.

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 worrying about how well they would integrate with an existing notebook system. In the future, we will implement Verso's features in popular computational notebook systems such as Observable [Bostock 2018].

4 MODULES: EMBEDDED GUIS THAT ENCAPSULATE PHYSICAL CONTROL

Representing digital fabrication workflows as live programs affords advantages but 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, makers must constantly communicate with the external digital and physical processes, for example, 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, away 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. Modules' functions optionally accept arguments and return values. Each module's associated GUI is rendered inline with code. When makers 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 makers stage toolpaths to send to the physical machine which are only sent once the maker 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 maker 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 [Paul Hudak et al. 1998; Wadler 1992] and build off of Omar et al. [2021]'s *livelits* (see discussion in subsection 2.3).

4.1 Example Modules

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

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* which maps the virtual space to its true physical location. To calculate the homography, makers 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 makers 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 are understood by the Axidraw plotter. AXIDRAWDRIVER communicates with the open-source Axidraw driver [Mark Oskay 2022] 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 maker 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 `$machineInitializer` function. For example, for the Jubilee CNC machine [Vasquez et al. 2020], the module exposes subroutines for connecting to the machine over a serial port and sends G-Code to the machine to home its axes in the required order. Makers execute these subroutines using graphical input and can debug and edit module subroutines as needed. The `$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.

4.2 Implementing custom modules

Makers 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 provides definitions for 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` which 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 3.
- **`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.
- **`handleInput()`** (multiple): methods that handle maker 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

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: Ordering TSS (G-Code Instruction Set)

needed calculations, such as computing the homography to localize visualizations to visual space.

5 TASK-RELEVANT VIEWS THROUGH TOOLPATH STYLESHEETS (TSS)

When building and iterating on workflows, makers 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, makers 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.

To empower makers to generate bespoke visualizations, Verso uses *toolpath stylesheets* (TSS), which translate instructions into visual primitives. Figure 1 shows an example TSS in the upper right for visualizing toolpath order, corresponding to the pseudocode of Figure 4. Formally, a TSS is an interpreter that interprets each toolpath instruction according to user-defined semantics. For example, consider the G-Code instruction `G1 X50 Y32 Z503 F1500`. Assuming that absolute (and not relative) coordinates are enabled on the machine, this instruction would cause the machine tool to move from its current position to the coordinate $(50, 32, 503)$ mm at a maximum speed of 1500mm/s. A TSS for producing a visualization might render a line from the current position to the new position. Further, a TSS for debugging speeds might color the rendered line according to the instruction's speed, with darker lines corresponding to faster movements. Visualizing speed allows the maker to quickly detect potentially risky high speed motion. This speed TSS provides functionality currently omitted from existing toolpath visualizers. As another example, consider a common issue with using laser cutters: overlapping geometries can produce a toolpath with many laser movements in a small area, which can lead to material ignition. Unfortunately, such issues can be invisible in the geometry; overlapping paths can easily be missed. A TSS would facilitate catching such events by parsing all movement-related instructions into points and binning the points into a 2D histogram to obtain a rough map of the expected energy output (a heat map) of the toolpath. The resultant heat map would immediately highlight potentially problematic areas of the toolpath, which makers can then debug in their workflow. This heat map TSS is detailed in Figure 5 top and Figure 6.

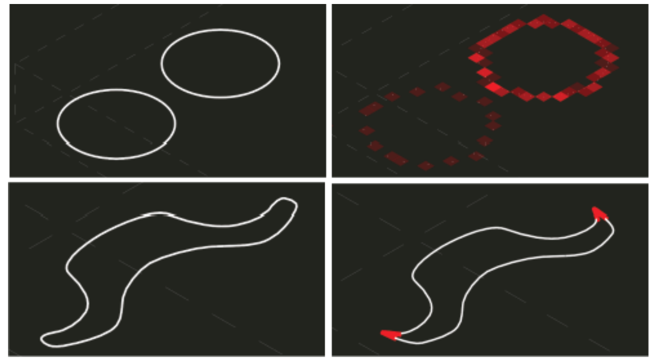

Figure 5: Toolpath Stylesheets (TSS) can highlight salient details of toolpaths. Top: the geometry of a toolpath of two circles is shown on the left. The expected energy from laser cutting is shown on the right. This TSS highlights that the rightmost circle contains duplicate paths; excess heat applied could ignite the material. Bottom: 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 maker.

More formally, a TSS *interprets* machine instructions with alternative visual semantics instead of physical machine action. Rather than attempting to simulate a physical process exactly, it offers a way to programmatically capture important—and possibly otherwise hard to detect—criteria of planned machine actions. Which criteria to include is chosen by the maker, enabling application-

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.}
$grid \leftarrow [\quad][\quad]$
$setZeroAll(grid)$
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

$maxCount \leftarrow max(grid)$
$meshes \leftarrow [\quad]$
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 6: Heat Map TSS

and task-specific visualizations. Similar to how CSS enables different views of underlying HTML, TSS enables multiple views of instructions for visual debugging in different scenarios.

Implementing a custom TSS. To implement a new TSS, makers write a pure function that accepts an instruction as an argument, matches the instruction by its opcode (e.g., `G0` for linear movement), parses the instruction's operands (e.g., `X50 Y32 Z503 F1500`) and returns a visual primitive, such as a line, arrow, or square.

We used the THREE.js library [Cabello 2014] to produce 3D visual primitives. Additional libraries (such as d3.js [Bostock et al. 2011]) are also compatible. Figure 4 and Figure 6 show pseudocode for implementing the ordering TSS (Figure 1, upper right) and the heat map TSS (Figure 5 top), respectively. As with modules and workflows, TSS can be shared with the community so makers need to write new ones only if they need functionality that existing ones lack.

6 EVALUATING VERSO PROGRAMMING FOR BESPOKE WORKFLOWS

We designed Verso to support experimental digital fabrication. Our evaluation assesses Verso's ability to enable explorations that would be difficult to conduct using existing workflow paradigms. Towards this goal, demonstrations of Verso's abilities are an appropriate evaluation method in toolkit research [Ledo et al. 2018].

We recruited three makers from our professional contacts who explore unconventional fabrication workflows: an artist who creates molds for sculptures using CNC milling and laser cutting, a

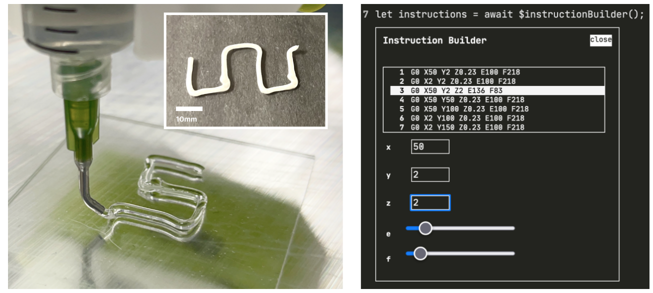

Figure 7: Exploring fabrication behaviors of novel materials using Verso. Shown here, a custom module for exploring salient parameters of extruding compressible hydrogel. Left: The hydrogel extruded in a rectilinear pattern. Right: A module with sliders for adjusting the amount of compressive force and speed of travel. The speed of travel and extrusion force are interdependent parameters; a tool for rapid iteration in this parameter space is valuable for the maker's exploration.

materials scientist who 3D prints structures using a novel hydrogel material, and a plant biologist who automates experiments with millimeter-scale plants. Over several sessions, we observed how each maker constructed their workflows, paying particular attention to parts of the workflow that were difficult to coordinate with existing tools. For each situation, we created a proof-of-concept workflow in Verso that demonstrates how live programming, modules, and TSS address pain points that each maker described.

6.1 Extending Toolpath Localization to Subtractive Manufacturing

Use case. The artist was learning to use subtractive manufacturing tools to create molds for sculptures. These tools cut away material to create a desired form. Here, the artist primarily used a Shopbot large-format CNC mill, a Roland toolchanging CNC mill, and a laser cutter. Although GUIs for CNC mills exist, they require extensive knowledge of milling jargon and parameters, and the generated toolpaths can be difficult to understand and modify. Moreover, subtractive manufacturing machines pose unique challenges. First, because parts are carved out of stock material, there is a much higher potential for waste. Second, if toolpaths are programmed improperly, the rapidly spinning end mills can break, throw material, crash into the machine body, and catch material on fire.

Demonstration. Using Verso, we first addressed the artist's primary concern: knowing where the tool will move *in physical space*, not just in an on-screen simulation, before running the machine and consuming material. We again used the TABLETOPCALIBRATOR and PROJECTOR modules to project visualizations into the machine's physical work envelope. As with with the plotter, the TABLETOPCALIBRATOR prompts the user to drag the corners of the projected work envelope to match the physical ground-truth work envelope. It then uses the empirical positions to calculate the same homography to accurately map the visualizations to the true locations in physical space. Unlike use of the plotter, the maker need only drag projected corners to match the machine's bed size without

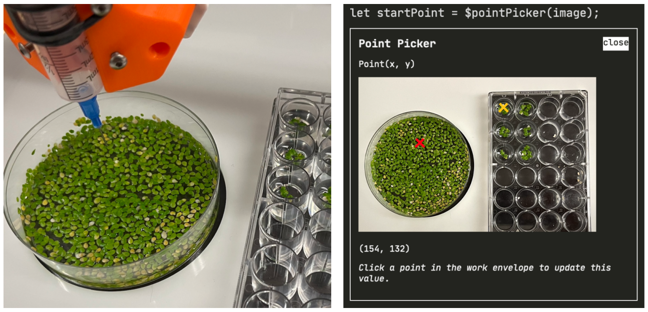

Figure 8: Machine-collected data inform biology workflows. Only healthy green duckweed plants should be transferred from the Petri dish to the well plate (left image). The plant biologist can select the healthy plants that should be transferred from an image of the workspace using the custom PointPicker module (red "x" to the yellow "x" in right image). Verso facilitates gathering data from the physical world that can be processed by subsequent modules.

first drawing landmarks. For Shopbot, which has a relatively large bed, projector lets makers select a smaller subset of the visualization to show. For the Roland mill, which has a moving bed, Verso currently supports visualization only when the bed is stationary before runtime; it could be extended to track bed movement using computer vision and fiducial markers and to update the projected visualization accordingly.

Furthermore, we proposed two new TSS that could help the artist visually recognize and debug issues unique to laser cutters and CNC mills. First, we proposed a TSS for creating a heat map of cuts resulting from the laser cutter (Figure 5). If a toolpath for a laser cutter cuts only a small region for an extended period, it risks warping, scorching, or even igniting the material there, depending on the type of material and the toolpath's power and speed settings. To generate this visualization, the heat map TSS internally plots all points visited by the laser in 2D space, bins them by proximity, space, and time, and generates a rough density visualization that calls attention to parts of the toolpath that have high energy density. This view lets the artist do a quick sanity check to determine if and where there might be a problem in the toolpath. Verso can then be used to edit the toolpath's instructions directly—for example, reducing laser power or increasing speed at dense areas, or removing extraneous paths. Second, we proposed a visualization that colors toolpaths by the order in which the instructions are executed, giving makers a quick view of when each part of the toolpath will be cut (Figure 1, upper right). In both cases, the artist can glean important information at the *instruction* level, which offers more information than just analyzing toolpath geometry alone, as is done in most standalone GUI tools.

6.2 Characterizing Hydrogel Deposition Behavior with Instruction-Level Programming

Use case. The material scientist was exploring the possibilities of 3D printing Pluronic F-127 thermosensitive hydrogel [National Center for Biotechnology Information 2022]. Pluronic gel, a new material, is solid at room temperature but liquid when cooled; research in materials science and engineering has investigated this property for use in drug delivery [Shriky et al. 2020], transdermal therapy [Chatterjee et al. 2019], and experimental methods in biology [Lesanpezeshki et al. 2019]. The ability to fabricate precise gel structures via digital fabrication could yield important future applications of Pluronic gel.

However, digitally fabricating structures is difficult because the gel is highly viscous and compressible. Extrusion requires fine control (see Figure 7). Additionally, unlike 3D printing with a plastic filament, the materials scientist must apply force on the gel in the syringe *before* it emerges from the needle, and vary the force applied depending on the velocity of the tool head, which itself varies with any sharp turns in the toolpath. The height of the needle off the printing bed also impacts whether and how the gel coils as it is extruded due to internal stresses in the extrusion stream. Thus, exploration is key to helping the materials scientist eventually print the desired structure.

Demonstration. To help the materials scientist explore, we authored a Verso program. First, the machineInitializer homed and zeroed a Jubilee machine [Vasquez et al. 2020]. The machineInitializer guides makers through sending G-code to the machine, and the module function does not return a `Machine` object that is marked as `{ initialized: true }` until the steps are completed. To enforce initialization prior to runtime, uninitialized `Machine` objects cannot be used by any modules to send instructions to the physical machine. Next, by calling instructionBuilder, the program invoked a GUI to interactively create G-code instructions with varying extrusion and feed rates. Unsurprisingly, while performing some test prints with dispatcher, we found that the material stalled and then "blobbed" when emerging from the needle if force was applied too suddenly. To address this, we added several instructions of gradually increasing extrusion rates to create a "slow start" for extrusion. The program also significantly reduced the velocity of instruction that moves the tool around corners.

To generalize these findings, we added a function that took a start and end point as inputs and output an array of instructions that implemented the "slow start" property. We added another function for handling velocities around curves. For both functions, output instructions varied the bed height by adjusting the z-coordinates of the instructions to control coiling during deposition. In contrast, most conventional slicers impose deposition layers of a fixed per-layer height.

Finally, to make this material knowledge available in the future, we created a new TSS that outputs a visualization specific to the Pluronic gel. The interpreter parses G-code and outputs gaps and blobs, highlighted in red, wherever the extrusion rates in the instruction increase too quickly. It also renders both the toolpath, whose z-coordinates may vary and a naïve prediction of the deposited gel that may be stretched or coiled. This TSS highlights findings from empirical tests and visually previews the gel's material properties. These functions and the new TSS offer first steps for *programatically capturing* material properties, which can then be readily refined for future iterations by adding them to the live program in Verso.

6.3 Integrating Camera Data for Handling Biological Samples

Use case. The plant biologist cultivates duckweed (*Lemna minor*) for large-scale, robotically automated hypothesis testing of phenotype data. Duckweed, an aquatic plant approximately 5mm in width, consists of tiny floating fronds. The biologist cultivates the duckweed en masse in a Petri dish and was attempting to transfer individual plants to a well plate (Figure 8). The key challenge is that the location of the duckweed plants in the Petri dish is unknown. Thus, off-the-shelf pipetting or pick-and-place software tools do not work well for this task because they typically assume fixed positions to transfer all materials.

Demonstration. To specify this workflow in Verso, we first created a new live program to drive a machine equipped with a camera and syringe. The code uses the IMAGER module, which opens a GUI so the plant biologist can (1) connect to the camera and (2) interactively apply a perspective transform to the camera feed so the resulting image data can precisely capture the machine's work surface. IMAGER's function output is an image with a one-to-one mapping of pixel coordinates to machine coordinates.

Using the POINTPICKER module, we clicked on a duckweed plant location in the image and then clicked on its destination on the well plate, which the module function returned. As a result, Verso generated toolpath instructions to move the syringe to the selected duckweed plant, apply slight suction with the syringe to pick up the plant, move it to the well plate, and deposit the plant in the desired well.

The functional nature of a Verso program helped us build on user-defined abstractions. For example, Verso lets makers write a function that takes an experimental protocol (i.e., a list of well plate locations corresponding to experimental conditions), prompt makers for a list of duckweed locations, and execute a set of duckweed movements at once.

Our demonstrations show how Verso's workflows-as-live-programs paradigm, interactive modules, and TSS could enable hypothetical workflows for experimental digital fabrication. These workflows are based on real-world makers and applications. We achieve what Ledo et al. [Ledo et al. 2018] describe to be the purpose of demonstrations as evaluation methods: "the goal of a demonstration is to use examples and scenarios to clarify how the toolkit's capabilities enable the claimed applications. A demonstration is an existence proof showing that it is feasible to use and combine the toolkit's components into examples that exhibit the toolkit's purpose and design principles."

7 LIMITATIONS AND FUTURE WORK

Verso provides a live programming environment for authoring digital fabrication workflows. To support liveness, it reruns the entire program every time a maker updates a program or a module's values. However, modules that communicate with external processes may experience high latency, disrupting the user experience of immediate, continuous feedback. To improve latency, we could represent incomplete data as program *holes* that we propagate rather than block and wait for, as presented by Omar et al. [Omar et al. 2019]. Once data is returned from the module, a hole would be updated and more computations propagated through the rest of the program. As proposed by Omar et al., Verso would then need some way to represent holes in further computation. Alternatively, Verso could cache computation so that only modified aspects of a workflow must be re-run, not the entire program.

We have shown how to interface with external tools using modules to encapsulate I/O. Tools with programmatic APIs can be integrated seamlessly into module implementations, whereas complex black-box graphical tools—such as software for driving the tool changing Roland CNC mill—are much more difficult to encapsulate. In such difficult scenarios, before a complete module is built, makers could write a preliminary module that displays a list of inputs that they manually input into an external process, as well as a way to import results from that process. Even in this stopgap scenario, Verso is still likely to improve upon the manual import/export paradigm by providing continuous evaluation for the rest of the workflow.

Ongoing collaborations with makers inspired Verso and the demonstrations we chose. To date, our example workflows serve as proofs-of-concept for how framing workflow construction as live programming can facilitate maker exploration, even across disparate domains. In the future, we will assess Verso's usability in studies with makers. As mentioned previously, we will also port Verso's features to more mature computational notebook tools for increased accessibility.

8 CONCLUSION

With Verso, we have demonstrated how the paradigm of workflows as live programs can help makers build and iterate on experimental digital fabrication workflows. The state-of-the-art in programming for fabrication machines presents numerous challenges for makers. We aimed to support three design goals—*synchronize program state with physical realities*, *allow graphical input without sacrificing the flexibility of code, and*, and *visualize different views of machine behavior physically in-situ*—that could address currently known problems. To this end, we contributed: a live programming environment for near-immediate feedback when making changes to workflows, modules for graphically interfacing with external processes and physical machines, and TSS for generating custom toolpath views from a set of machine instructions. We authored three example workflows that show how Verso could solve challenges with the state of the art across disparate domains where digital fabrication tools are beginning to play pivotal roles.

From reasoning about instruction set architectures, to handling asynchronous I/O across distributed machines and systems, to creating bespoke data visualizations, makers—whether they realize it or not—are becoming skilled programmers as they explore the unique design spaces afforded by digital fabrication machines. To this end, Verso celebrates and further encourages this emerging programming practice by championing persistent feedback, integrated control, and visualization of fine-grain machine control. By building workflows as live programs, we move one step closer to a future where novel production processes in art, science, and engineering can be more quickly composed and more widely shared, understood, and extended by a diverse community of makers.

ACKNOWLEDGEMENTS

We thank the Alfred P. Sloan Foundation for supporting this work.

REFERENCES

Patrick Baudisch and Stefanie Mueller. 2017. Personal Fabrication. *Foundations and Trends® in Human–Computer Interaction* 10, 3-4 (2017), 165–293. https://doi.org/10.1561/1100000055

Ekaba Bisong. 2019. Google Colaboratory. In *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Apress, Berkeley, CA, 59–64. https://doi.org/10.1007/978-1-4842-4470-8_7

Rick Bitter, Taqi Mohiuddin, and Matt Nawrocki. 2006. *LabVIEW: Advanced programming techniques*. Crc Press.

Mike Bostock. 2018. A Better Way to Code. https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0

Mike Bostock. 2022. Observable Inputs. https://github.com/observablehq/inputs original-date: 2021-01-25T16:50:46Z.

M. Bostock, V. Ogievetsky, and J. Heer. 2011. D³ Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. *UIST* (2013). https://doi.org/10.1145/2501988.2502050

Alex Bäuerle, Ángel Alexander Cabrera, Fred Hohman, Megan Maher, David Koski, Xavier Suau, Titus Barik, and Dominik Moritz. 2022. Symphony: Composing Interactive Interfaces for Machine Learning. In *CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3491102.3502102

Ricardo Cabello. 2014. three.js - Javascript 3D library. https://threejs.org/

Sudipta Chatterjee, Patrick Chi-leung Hui, Chi-wai Kan, and Wenyi Wang. 2019. Dual-responsive (pH/temperature) Pluronic F-127 hydrogel drug delivery system for textile-based transdermal therapy. *Sci Rep* 9, 1 (Aug. 2019), 11658. https://doi.org/10.1038/s41598-019-48254-6 Number: 1 Publisher: Nature Publishing Group.

Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. 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*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376729

Robert A DeLine. 2021. 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*. Association for Computing Machinery, New York, NY, USA, 1–11. http://doi.org/10.1145/3411764.3445267

Mustafa Doga Dogan, Steven Vidal Acevedo Colon, Varnika Sinha, Kaan Akşit, and Stefanie Mueller. 2021. 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)*. Association for Computing Machinery, New York, NY, USA, 24–38. https://doi.org/10.1145/3472749.3474733

James Fogarty. 2017. Code and contribution in interactive systems research. In *Workshop HCITools: Strategies and Best Practices for Designing, Evaluating and Sharing Technical HCI Toolkits at CHI*.

Sean Follmer, David Carr, Emily Lovell, and Hiroshi Ishii. 2010. 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)*. Association for Computing Machinery, New York, New York, USA, 381–382. https://doi.org/10.1145/1866218.1866230

Jack Forman, Mustafa Doga Dogan, Hamilton Forsythe, and Hiroshi Ishii. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 1222–1233. https://doi.org/10.1145/3379337.3415876

Frikk Fossdal, Rogardt Heldal, and Nadya Peek. 2021. Interactive Digital Fabrication Machine Control Directly Within a CAD Environment. In *Symposium on Computational Fabrication (SCF '21)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3485114.3485120

Emrecan Gulay and Andrés Lucero. 2019. Integrated Workflows: Generating Feedback Between Digital and Physical Realms. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3290605.3300290

Shiqing He and Eytan Adar. 2020. Plotting with Thread: Fabricating Delicate Punch Needle Embroidery with X-Y Plotters. In *Proceedings of the 2020 ACM Designing Interactive Systems Conference (DIS '20)*. Association for Computing Machinery, New York, NY, USA, 1047–1057. https://doi.org/10.1145/3357236.3395540 event-place: Eindhoven, Netherlands.

Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New Orleans, LA, USA, 281–292. https://doi.org/10.1145/3332165.3347925

Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. 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)*. ACM, New York, NY, USA, 532:1–532:12. https://doi.org/10.1145/3173574.3174106 event-place: Montreal QC, Canada.

Gaoping Huang, Xun Qian, Tianyi Wang, Fagun Patel, Maitreya Sreeram, Yuanzhi Cao, Karthik Ramani, and Alexander J. Quinn. 2021. 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)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3411764.3445283

Nathaniel Hudson, Celena Alcock, and Parmit K. Chilana. 2016. 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)*. ACM, New York, NY, USA, 384–396. https://doi.org/10.1145/2858036.2858266

Hyeonsu Kang and Philip J. Guo. 2017. 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)*. Association for Computing Machinery, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842

Jeeeun Kim, Clement Zheng, Haruki Takahashi, Mark D Gross, Daniel Ashbrook, and Tom Yeh. 2018. Compositional 3D Printing: Expanding & Supporting Workflows Towards Continuous Fabrication. In *Proceedings of the 2Nd ACM Symposium on Computational Fabrication (SCF '18)*. ACM, New York, NY, USA, 5:1–5:10. https://doi.org/10.1145/3213512.3213518

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. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, 87–90. https://doi.org/10.3233/978-1-61499-649-1-87

Lakehouse. 2022. Databricks. https://databricks.com/

David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 36:1–36:17. https://doi.org/10.1145/3173574.3173610 event-place: Montreal QC, Canada.

Leila Lesanpezeshki, Jennifer E. Hewitt, Ricardo Laranjeiro, Adam Antebi, Monica Driscoll, Nathaniel J. Szewczyk, Jerzy Blawzdziewicz, Carla M. R. Lacerda, and Siva A. Vanapalli. 2019. Pluronic gel-based burrowing assay for rapid assessment of neuromuscular health in C. elegans. *Sci Rep* 9, 1 (Oct. 2019), 15246. https://doi.org/10.1038/s41598-019-51608-9 Number: 1 Publisher: Nature Publishing Group.

Jingyi Li, Joel Brandt, Radomír Mech, Maneesh Agrawala, and Jennifer Jacobs. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376765

Tom Lieber, Joel Brandt, and Rob Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. *CHI* (2014). https://doi.org/10.1145/2556288.2557409

Chandan Mahapatra, Jonas Kjeldmand Jensen, Michael McQuaid, and Daniel Ashbrook. 2019. Barriers to End-User Designers of Augmented Fabrication. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, Glasgow, Scotland Uk, 1–15. https://doi.org/10.1145/3290605.3300613

Mark Oskay. 2022. AxiDraw. https://github.com/evil-mad/axidraw original-date: 2016-02-01T11:21:13Z.

National Center for Biotechnology Information. 2022. Pluronic F-127. https://pubchem.ncbi.nlm.nih.gov/compound/10154203

Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 511–525. http://doi.org/10.1145/3453483.3454059

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live functional programming with typed holes. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019), 14:1–14:32. https://doi.org/10.1145/3290327

Jifei Ou, Gershon Dublon, Chin-Yi Cheng, Felix Heibeck, Karl Willis, and Hiroshi Ishii. 2016. 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)*. Association for Computing Machinery, New York, NY, USA, 5753–5764. https://doi.org/10.1145/2858036.2858257

Wei Ouyang, Richard W. Bowman, Haoran Wang, Kaspar E. Bumke, Joel T. Collins, Ola Spjuth, Jordi Carreras-Puigvert, and Benedict Diederich. 2021. An Open-Source

Modular Framework for Automated Pipetting and Imaging Applications. *Advanced Biology* n/a, n/a (Oct. 2021), 2101063. https://doi.org/10.1002/adbi.202101063 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/adbi.202101063.

Paul Hudak, John Peterson, and Joseph Fasel. 1998. A Gentle Introduction to Haskell: IO. https://www.haskell.org/tutorial/io.html

Nadya Peek and Neil Gershenfeld. 2018. Mods: Browser-Based Rapid Prototyping Workflow Composition. In *Proceedings of the 38th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*. Mexico City, Mexico, 6. http://papers.cumincad.org/cgi-bin/works/Show?acadia18_66

Miller Puckette. 2022. Pure Data. https://puredata.info/

Michael L. Rivera, Jack Forman, Scott E. Hudson, and Lining Yao. 2020. Hydrogel-Textile Composites: Actuators for Shape-Changing Interfaces. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems (CHI EA '20)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3334480.3382788

Bana Shriky, Adrian Kelly, Mohammad Isreb, Maksims Babenko, Najet Mahmoudi, Sarah Rogers, Olga Shebanova, Tim Snow, and Tim Gough. 2020. Pluronic F127 thermosensitive injectable smart hydrogels for controlled drug delivery system development. *Journal of Colloid and Interface Science* 565 (April 2020), 119–130. https://doi.org/10.1016/j.jcis.2019.12.096

Blair Subbaraman and Nadya Peek. 2022. p5.fab: Direct Control of Digital Fabrication Machines from a Creative Coding Environment. *arXiv:2205.00323 [cs]* (April 2022). http://arxiv.org/abs/2205.00323 arXiv: 2205.00323.

Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (June 1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

Rundong Tian, Vedant Saran, Mareike Kritzler, Florian Michahelles, and Eric Paulos. 2019. Turn-by-Wire: Computationally Mediated Physical Fabrication. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New Orleans, LA, USA, 713–725. https://doi.org/10.1145/3332165.3347918

Jasper Tran O'Leary, Chandrakana Nandi, Khang Lee, and Nadya Peek. 2021. Taxon: a Language for Formal Reasoning with Digital Fabrication Machines. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, New York, NY, USA, 691–709. https://doi.org/10.1145/3472749.3474779

Hannah Twigg-Smith, Jasper Tran O'Leary, and Nadya Peek. 2021. 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)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3411764.3445653

Joshua Vasquez, Hannah Twigg-Smith, Jasper Tran O'Leary, and Nadya Peek. 2020. Jubilee: An Extensible Machine for Multi-tool Fabrication. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. ACM, New York, NY, USA.

Philip Wadler. 1992. Comprehending Monads. In *Mathematical Structures in Computer Science*. 61–78.

Guanyun Wang, Lining Yao, Wen Wang, Jifei Ou, Chin-Yi Cheng, and Hiroshi Ishii. 2016. xPrint: A Modularized Liquid Printer for Smart Materials Deposition. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, San Jose California USA, 5743–5752. https://doi.org/10.1145/2858036.2858281

Wasp. 2019. Delta WASP 2040 Clay. https://www.3dwasp.com/en/clay-3d-printer-delta-wasp-2040-clay/

Christian Weichel, Jason Alexander, Abhijit Karnik, and Hans Gellersen. 2015a. SPATA: Spatio-Tangible Tools for Fabrication-Aware Design. In *Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '15)*. ACM, New York, NY, USA, 189–196. https://doi.org/10.1145/2677199.2680576

Christian Weichel, John Hardy, Jason Alexander, and Hans Gellersen. 2015b. ReForm: Integrating Physical and Digital Design through Bidirectional Fabrication. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, Charlotte, NC, USA, 93–102. https://doi.org/10.1145/2807442.2807451

Christian Weichel, Manfred Lau, David Kim, Nicolas Villar, and Hans W. Gellersen. 2014. MixFab: a mixed-reality environment for personal fabrication. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. Association for Computing Machinery, Toronto, Ontario, Canada, 3855–3864. https://doi.org/10.1145/2556288.2557090

Karl D.D. Willis, Cheng Xu, Kuan-Ju Wu, Golan Levin, and Mark D. Gross. 2011. Interactive Fabrication: New Interfaces for Digital Fabrication. In *Proceedings of the Fifth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '11)*. ACM, New York, NY, USA, 69–72. https://doi.org/10.1145/1935701.1935716

Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 152–165. https://doi.org/10.1145/3379337.3415851

Kentaro Yasu. 2017. Magnetic Plotter: A Macrotexture Design Method Using Magnetic Rubber Sheets. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, Denver Colorado USA, 4983–4993. https://doi.org/10.1145/3025453.3025702

Nur Yildirim, James McCann, and John Zimmerman. 2020. 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)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376621