Machine-o-Matic: a Programming Environment for Prototyping Digital Fabrication Workflows

Jasper Tran O'Leary
University of Washington, Seattle, WA, USA
jaspero@uw.edu

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

Abstract

Digital fabrication tools for Makers have increased access to manufacturing processes such as 3D printing and computer-controlled laser cutting or milling. However, these machines and their associated software tools are difficult to modify and adapt beyond common case tasks. How can we enable Makers to design and operate machines with other applications? To facilitate custom machine design and control, we propose a domain-specific language for formalizing fabrication workflows as programs. This language, called Machine-o-Matic, provides an interface for authoring workflow and for defining machine configurations in software. Programs in the language compile to custom firmware for controlling physical machines. We demonstrate key features of Machine-o-Matic and highlight the future possibilities for verifiable fabrication using a programming languages approach.

2012 ACM Subject Classification Human-centered computing → Interactive systems and tools; Applied computing → Computer-aided manufacturing; Software and its engineering → Domain specific languages

Keywords and phrases Digital fabrication, programming languages, user interfaces, prototyping

Digital Object Identifier 10.4230/OASIcs.CVIT.2016.23

1 Introduction

At a global scale, the rise of the Maker movement and academic makerspaces has engaged more people in using digital fabrication tools than ever before. Tools for digital fabrication include CNC machines, which we use to to refer to any computer-controlled machine that users can program through computer software. Common examples of CNC machines include general-purpose machines such as laser cutters, 3D printers, and CNC mills, as well as machines for niche use cases. In addition to physical CNC machines, there is a growing ecosystem of open-source software tools to support specific parts of the fabrication pipeline, for example: optimizing 3D model meshes for fabrication [8], slicing 3D model meshes into toolpaths [2], and designing printed circuit boards [1]. With the increased availability of affordable CNC machines comes the promise of diverse applications of digital fabrication, where individuals who are not expert machine users can adapt CNC machines and software to their own workflows.

1.1 Workflow: a Fabrication Task Made up of Digital and Physical Steps

Let us define a *workflow* as going from a concept, through various stages of design and physical fabrication, to a completed prototype for product. Any digital fabrication workflow will incorporate various machines, materials, software tools, hardware modifications, file types, etc. that are strung together. For example, a workflow for something as simple as 3D printing a metal figurine, a model is made in CAD, exported as an STL, sliced in

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:11
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

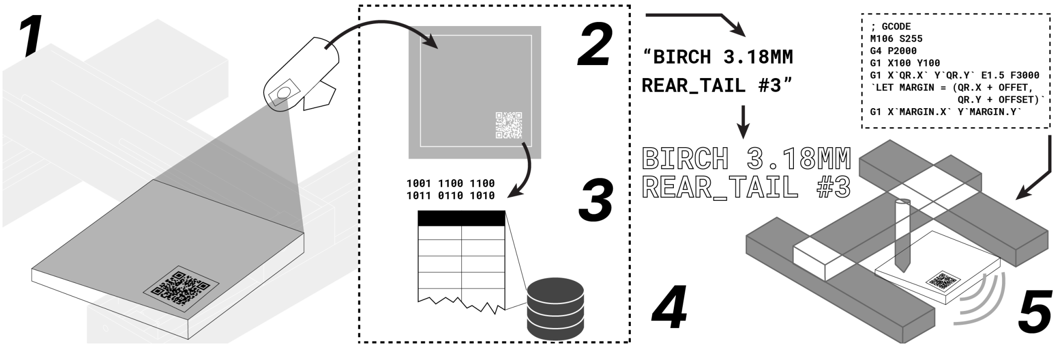

Figure 1 *Example Workflow. A user positions sheet material in a plotter's work envelope (1) and images the material to extract a QR code (2). They then look up the code in a database to retrieve an annotation for the given sheet (3). Finally, they create a toolpath for drawing the annotation (4) and generate G-Code using quasiquoted variable values for moving the machine (5).*

a printer-specific slicer, exported as G-code, transferred to the machine, interpreted by controllers into motor moves of a motion platform and extrusion head, removed from the bed of the printer, sintered in an oven according to the material's temperature specs, then cleaned and polished for final use. Other applications will require a different assembly of steps in software, hardware, and material handling.

However, digital fabrication infrastructure is static—difficult to modify and adapt. There is no formalization for connecting different parts of a workflow. At the machine level, modifying CNC machine typically involves reprogramming *controller boards* that are hard-wired to support machine controls for engineering use cases. Even for users with technical expertise in machine building, modifying controllers to change kinematics, or to add functionality, is "hacky" and involves rewriting firmware. With software tools, it can be difficult to reason about inputs and outputs for different parts of the pipeline. For example, a user who is 3D printing will often need to tinker with the 3D model's design, the conversion of the model to a mesh, the slicing of that model, and the machine instructions (G-Code) generated from the slices—all at the same time. This static infrastructure poses a prohibitively high barrier to a diverse set of users who need to do tasks not commonly supported by software tools, but who do not have prior technical background in fabrication.

In particular, there are few, if any, ways to *formally verify* that output from one part of the workflow will work as input for another part of the workflow; for example, ensuring that a GCode file will not cause a spindle to exit the work envelope, or ensuring that a 3D printer extruder never revisits a place with material already deposited. Even if we implemented these simple safety checks ad-hoc, they might not cover other workflows that are developed by different users in the future. With current tools, if the user has modified the printer, or wishes to generate machine instructions from sources besides a 3D model, they must tinker with machine instructions, export parameters, and model data all at once. All too often, the solution to these issues is for users to "just know" if and when hacks to machines and software tools will work.

1.2 Reimagining Fabrication Workflows as Formal Programs

In this paper, we define *users* as people who are using digital fabrication machines and software in hobbyist, academic, or professional contexts besides mass manufacturing settings. These users may wish to apply the precision of fabrication machines in contexts such as art, biology laboratory work, cooking, packaging, and many others. We envision these users as tinkerers who are comfortable designing for themselves and with learning and using software. However these users need not be comfortable with understanding or designing machines, or with established practices for fabrication workflows.

We now ask: what are the needs of these new fabrication practitioners? What are the tools that can meet these needs? We hypothesize that it is not making mass-manufacturing machines more efficient, but developing novel machines and ways of interacting with those machines. These machines and their workflows need to be robust, reliable, reconfigurable, and easy to learn.

Rather than proceed with further optimizations to static fabrication infrastructure, we envision fundamentally changing how people create fabrication workflows. We propose representing fabrication workflows as *programs*, where machines, materials, data, and controls are all **first-class citizens in an interactive programming environment**. To facilitate programming, modifying, and controlling parts of the workflow, we propose developing Machine-o-Matic: a domain-specific programming language for integrating disparate tools into a cohesive setting. Critically, users would be able to define machines in software based on the criteria they need, adding in sensors, data, and other features within the context of a programming environment that affords static checking, programming by example, etc. The language, Machine-o-Matic, would be embedded within Javascript to enable users to use Machine-o-Matic's features alongside existing capabilities of the popular language.

2 System Architecture

Machine-o-Matic comprises three parts (see Figure 2):

- **Machine-o-Matic Language:** a domain-specific language for formally describing a *machine configuration* of motors, sensors, tools, and instrumentation, as well as support for debugging and verifying machine behavior before runtime. The DSL would be embedded within a larger host language such as Javascript so that users can take advantage of general purpose computation and data processing that the host language affords.
- **Controller Firmware Compilation:** a means of compiling machine configurations in the DSL into firmware to upload to the machine's control board. This firmware translates movement commands into physical motor pulses for the given machine configuration.
- **Graphical Front End:** a browser-based visual tool for quickly assembling and simulating machines, for synthesizing parts of programs in the DSL using graphical techniques, and for inspecting and visualizing stages of the workflow.

Machine-o-Matic Language

To conceptualize how a machine configuration would be represented in the Machine-o-Matic language, we use a concrete example of a CNC plotter (see Figure 2 for a visual representation of a plotter) instrumented with a web camera to label sheet material in an specified location. Our example workflow, illustrated in Figure 1 is as follows:

1. Place a sheet of material with a QR code sticker on it in the plotter's work envelope.
2. Use a web camera mounted above the plotter to image the material.
3. Read the QR code and look up the appropriate annotation for the current sheet of material.
4. Create a toolpath for writing the annotation with the machine's writing tool.

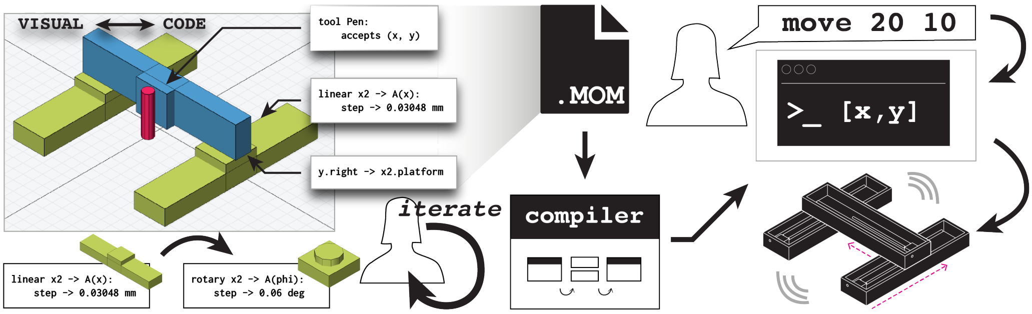

Figure 2 *Machine-o-Matic System Architecture. A user programs a machine configuration in the DSL. Code in the DSL corresponds to physical implementation. The system compiles the configuration to controller firmware which then actuates the physical machine.*

Listing 1 Defining a machine configuration in the DSL

```
let plotter : Machine = new Machine({
    "linear Axis(x)" : "Motor(x1), Motor(x2) @ step -> 0.03048 mm",
    "linear Axis(y)" : "Motor(y) @ step -> ??? mm",
    "binary ToolUpDown" : "Motor(t)",
});
```

5. Position the writing tool 100mm to the right of the sticker and plot the annotation onto the sheet.

In the Machine-o-Matic DSL, the user first defines a machine configuration as shown in Listing 2. With this code, the user defines a plotter machine configuration as the variable `plotter`. To instantiate the machine, the user provides information about the machine's motors, for example `"linear Axis(x)" : "Motor(x1), Motor(x2) @ step -> 0.03048 mm"`. This statement indicates that there are two motors named `x1` and `x2` which will move in parallel to drive the plotter's toolhead along the x-axis. The machine configuration will eventually be used to generate controller board firmware, so the user must indicate how many millimeters of displacement result from one step of the motor (`step -> 0.03048 mm`) to aid with kinematic calculations. If the user does not know this information, they can leave a hole in the program (`???`) and the system will probe the motor's movement at runtime, prompt the user to measure the displacement, and synthesize a correct replacement to the hole based on empirical measurement. The user also declares a non-axis degree of freedom `ToolUpDown`, which simply uses two positions on the motor named `t` to extend and retract the writing instrument.

Listing 2 Declaring objects for sensor input, material data, and computer-aided manufacturing (CAM)

```
let camera : WebCamera = new WebCamera({
    port: "/dev/tty.usbserial1402"
});
plotter.addSensor(camera);
let materialTable : Table = loadTableFromDatabase();
let profileCAM : CAM = new CAM({ pathType: "profile" });
```

Listing 3 Defining a machine action that can be called during runtime

```
plotter.action("locateAndPlotAnnotation", () => {
    let image : Image = camera.readImage();
    let annotationPoint : Vector3 = image.findQRPoint()
        .translateX(100);
    let annotationForSheet : String = materialTable
        .query(image.decodeQR());
    let toolpath : Toolpath = profileCAM
        .generateToolpath(annotationForSheet);

    this.moveTo(annotationPoint);
    this.ToolUpDown.down();
    this.plot(toolpath);
    this.ToolUpDown.up();
});
```

In addition to defining a machine configuration in software, the user can add and integrate sources of data. In Listing 2, the user declares a variable `camera` as an interface to a web camera mounted above the plotter, and connects that as a sensor to the plotter. The user also imports a database that maps the QR codes on the stickers to the appropriate text to be plotted on the corresponding sheet of material. They then instantiate a `CAM`, or computer-aided manufacturing object to transform text into movement paths for the plotter. Each of these variables is declared with a type, for example `let toolpath : Toolpath`, which affords static type checking at compile time.

Next, as shown in Listing 2 the user defines an *action*, `locateAndPlotAnnotation` that the machine can perform. The machine can perform actions at any time, similar to calling a function in software. Critically, this action integrates image data, database lookups, toolpath generation, and custom motor movements within a single function call.

In the above code, the machine images the material sheet and processes image data. Using the data, the machine (`this`) moves the tool to the correct location, actuates the motor to lower the writing tool, plots the toolpath, and re-raises the tool. For every step, the language employs a type system to check for common compile-time errors.

Using a programming language exposes formerly black boxed tools, grants machines open access to data, and allows users to verify high-level constraints before machines begin running, all in a clear syntax that can easily be shared and modified. Critically, a language allows us to use standard program analysis techniques to verify the behavior of the workflow. For example, we can check to make sure the machine instructions produced are compatible with the machine that will run them. We can also enforce invariants such as requiring that the tool never be moved outside the machine's work envelope, or that all machine instructions generated from a data source contain no null values. Finally, Machine-o-Matic provides a graphical front end for composing programs in the language, including designing machine configurations (see Figure 3, left) and visualizing stages of the workflow (see Figure 3, right).

Controller Firmware Compilation

CNC machines have controller boards that translate machine instructions such as GCode into electrical pulses that actuate the motors and move the tool head. Typically, machine kinematics are "baked in" the physical controller board and are difficult to modify. Seemingly simple modifications like adding another motor or adding another machine instruction usually

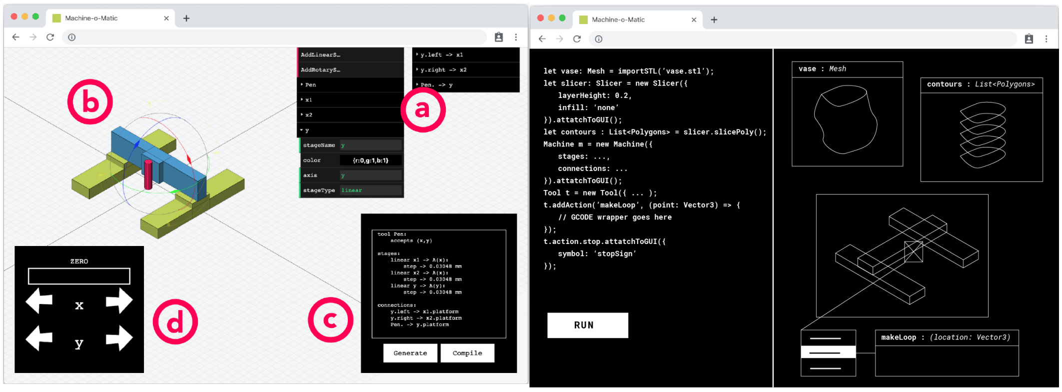

Figure 3 *Machine-o-Matic Front End. Left: direct manipulation interface for machine creation. a) menu-based editor for motor parameters, b) machine simulation, c) corresponding* `.mom` *program, d) GUI controller for moving the physical and simulated machine. Right: High-level scripting interface with code (left side) and visual traces (right side).*

require rewriting low-level firmware, along with purchasing specialized hardware.

With Machine-o-Matic, a user can instead specify high-level machine configurations details in the DSL, and then the system compiles toe specification down to low-level firmware code to upload to a non-machine specific controller board such as an Arduino. For example, given the plotter from the code above, assume that `annotationPoint` is (50mm, 30mm), that is, 50mm on the x-axis and 30mm on the y-axis. For the machine to move the tool to this location given its current position, the machine's controller needs to know the machine's kinematics, which motors control which axes, and the motor step rates, how much displacement along the axis results from one step of the motor. Because the user provides this information when instantiating `plotter`, Machine-o-Matic compiles the information to firmware that the user can upload to the controller board. Now, whenever the user wishes to modify their machine, they need only change the machine configuration in the DSL, recompile, and reupload, rather than spending hours or days reconfiguring machine firmware.

As opposed to low-level configuration files common in CNC control frameworks e.g. [12, 13, 30, 31], a programming language enables portable software-defined hardware, as opposed to hardware-defined software, which is currently the norm with machine making. We draw inspiration from other hardware description languages such as Verilog for electronics [34], ROS Unified Robot Description Format [27], and openFrameworks for cross-platform graphics [25]. These languages allow designers to build at various levels of abstraction before diving into the implementation details. Our goal is not to supplant existing control frameworks, but rather to provide a more robust way to design and deploy control firmware, including compiling to existing configuration files, rather than expecting users to write them by hand.

3 Related Work

We draw from literature in robotics, programming languages, and HCI for fabrication. Our work adds to concepts in fabrication literature such as interactive [35], mobile [28], and personal [7] fabrication. At the same time, we acknowledge lineages of making that lie outside of Western and technology solutionist views of fabrication [6, 18, 5] Our goal is to use techniques from existing fabrication, robotics, and graphics literature to empower a wider group of people to build their own fabrication infrastructure.

3.1 Component-Based Design

One thread of work examines computational tools that let users create complex objects through structured assembly of primitives. We particularly note Peek et al., who built a system for rapidly building fabrication machines of various sizes using modular cardboard parts [26]. As part of this system, Moyer contributed the idea of a virtual CNC machine where physical machine modules could be programmed in software in an object-oriented manner [23]. These parts can be controlled by constructing virtual machine representations and controlling the modules in software. This kit could further the concept of mobile fabrication by allowing people to build machines on-the-fly. However, the kit has relatively high user overhead for defining and using the controls, which limits the build machines' ability to work specifically for a user's environment. We wish to faciliate the creation of machines for specific scenarios, which futhers this vision of mobile fabrication towards ubiquitous fabrication.

Work primarily in computer graphics has explored how users can build prototypes while ensuring that they can be assembled. In particular, Koo et al. contributed a system for building works-like prototypes, where users can specify high-level functional relationships between components, such as hinges [15]. Lau et al. draw inspiration from programming language techniques to build a system that allows users to create generative furniture [16]. In their technique, they define a grammar for fabrication rules, and add lexical analysis for checking the feasibility of assemblies. Ureta et al. define a vocabulary of fixed motions for 3D assemblies [32].

In robotics, researchers have applied these techniques for quickly building robots. Mehta and Rus propose ways of making robots from printable cardboard parts [20, 21, 22]. Schultz et al. build on these components for cardboard robots while optimizing for motion [29]. Desai et al. look at how to quickly make robots out of expert-defined parts, both for non-articulated [9] and articulated robots [10]. They promote the idea of *computational abstractions*, where a user designs with abstractions e.g. robot arm, robot body, but the implementations can vary without invalidating the design.

Others in industry have applied component-based techniques to part of machine design. In particular, Vention offers a commercial platform for creating machines from drag-and-drop components, and then the company ships the parts to the user for assembly [33]. However, the target audience is for technical users in existing manufacturing settings, which differs from our goal of rapid iteration and control by novice users. We now wish to apply similar component-based techniques within the context of a programming language for machines.

3.2 Programming Language Techniques for Fabrication

Finally, we turn towards an emerging thread of work that uses techniques from programming language research to reconceptualize the fabrication process. Nandi et al. designed a functional programming language for representing constructive solid geometries (CSG) commonly used in CAD modeling for fabrication [24]. By representing CSGs as a programming language, users can verify that their designs are fabricatable, compile to mesh models, and even decompile meshes into CSGs. Du et al. similarly propose a system for reverse engineering CSG representations of static meshes [11]. The Tool Path Language project proposes a replacement for G-COde for machine instructions, and employs a clearer syntax and integration with Javascript [3].

Other systems use programming language techniques to drive interaction. We draw particular inspiration from Mayer et al., who feature a language that lets users directly manipulate artwork, or the source code that generated it, while having both representations synchronized to new changes [19]. Lerner at all feature program construction through assembling *polymorphic blocks* that fit together only with other blocks of appropriate types in the language [17]. Jacobs and Buechley represent fabricatable *objects* as programs, [14], while Agrawal et al. contribute a visual Scratch-based programming environment for creating 3D models [4]. We wish to expand upon these areas by extending these techniques for both representing machine configurations, as well as in framing novices' thinking about machine building.

4 Next Steps and Open Questions

As we further develop Machine-o-Matic, we would like to solicit feedback from the research community on the following challenges:

- **Co-Designing a Language with Practitioners.** How can we best design the constructs in a language that make sense to potential users. Having already redesigned the language once, we recognize the value of iterative prototyping and feedback from users, but also must also must also take a stance on what should and should not be included.
- **Making Use of Techniques in Programming Languages.** How can we leverage contemporary ideas in programming languages literature to empower Machine-o-Matic? Given our recasting of fabrication workflows as programs, we would like to leverage existing techniques for analyzing programs. In other words, what will the "smarts" for this language be?
- **Advocating for Common Infrastructure in Fabrication Research.** In HCI, fabrication research tends to highlight new interaction techniques with machines, rather than look back and tie existing developments together. How can we develop Machine-o-Matic in a way that appeals to the research community?

5 Conclusion

In this paper, we demonstrated for the need for formalizing digital fabrication workflows. We argue that integrating disparate parts of a workflow—including computer-aided design, geometry processing, toolpathing, sensors, and machine design—into a common environment would enable emerging groups of users to leverage fabrication technology. Through authoring workflows as programs, we introduce clearer syntax and replicatabilty as end users are able to share and modify existing workflows. Programs also afford static analysis, checking for errors beofre machines run and possibly waste material, which is particularly important for composing novel workflows. Finally, software-defined fabrication allows for quicker prototyping and debugging of workflows while reducing the amount of time spent working with low-level machine firmware. We aim to further develop and test Machine-o-Matic to encourage a broader community of users to build fabrication workflows that work for their own contexts.

References

1 KiCad. URL: `http://kicad-pcb.org/`.
2 Slic3r - Open source 3d printing toolbox. URL: `https://slic3r.org/`.

3 Tool Path Language. URL: https://tplang.org/.

4 Harshit Agrawal, Rishika Jain, Prabhat Kumar, and Pradeep Yammiyavar. FabCode: Visual Programming Environment for Digital Fabrication. In *Proceedings of the 2014 Conference on Interaction Design and Children*, IDC '14, pages 353–356, New York, NY, USA, 2014. ACM. event-place: Aarhus, Denmark. URL: http://doi.acm.org/10.1145/2593968.2610490, doi:10.1145/2593968.2610490.

5 Shaowen Bardzell. Feminist HCI: Taking Stock and Outlining an Agenda for Design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1301–1310, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1753326.1753521, doi:10.1145/1753326.1753521.

6 Shaowen Bardzell, Jeffrey Bardzell, and Sarah Ng. Supporting Cultures of Making: Technology, Policy, Visions, and Myths. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 6523–6535, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3025453.3025975, doi:10.1145/3025453.3025975.

7 Patrick Baudisch and Stefanie Mueller. Personal Fabrication. *Foundations and Trends® in Human–Computer Interaction*, 10(3-4):165–293, 2017. URL: http://www.nowpublishers.com/article/Details/HCI-055, doi:10.1561/1100000055.

8 Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. *Eurographics Italian Chapter Conference*, page 8 pages, 2008. URL: http://diglib.eg.org/handle/10.2312/LocalChapterEvents.ItalChap.ItalianChapConf2008.129-136, doi:10.2312/localchapterevents/italchap/italianchapconf2008/129-136.

9 Ruta Desai, James McCann, and Stelian Coros. Assembly-aware Design of Printable Electromechanical Devices. pages 457–472. ACM, October 2018. URL: http://dl.acm.org/citation.cfm?id=3242587.3242655, doi:10.1145/3242587.3242655.

10 Ruta Desai, Ye Yuan, and Stelian Coros. Computational abstractions for interactive design of robotic devices. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1196–1203, May 2017. doi:10.1109/ICRA.2017.7989143.

11 Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. InverseCSG: Automatic Conversion of 3d Models to CSG Trees. In *SIGGRAPH Asia 2018 Technical Papers*, SIGGRAPH Asia '18, pages 213:1–213:16, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3272127.3275006, doi:10.1145/3272127.3275006.

12 Gecko Drive. Gecko drive systems, 2019. https://www.geckodrive.com/stepper-motor-controls.html, accessed Aug 2019.

13 Gnea. Grbl v1.1 Configuration, April 2019. https://github.com/gnea/grbl, accessed Aug 2019.

14 Jennifer Jacobs and Leah Buechley. Codeable objects: computational design and digital fabrication for novice programmers. pages 1589–1598. ACM, April 2013. URL: http://dl.acm.org/citation.cfm?id=2470654.2466211, doi:10.1145/2470654.2466211.

15 Bongjin Koo, Wilmot Li, JiaXian Yao, Maneesh Agrawala, and Niloy J. Mitra. Creating Works-like Prototypes of Mechanical Objects. *ACM Trans. Graph.*, 33(6):217:1–217:9, November 2014. URL: http://doi.acm.org/10.1145/2661229.2661289, doi:10.1145/2661229.2661289.

16 Manfred Lau, Akira Ohgawara, Jun Mitani, and Takeo Igarashi. Converting 3d Furniture Models to Fabricatable Parts and Connectors. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 85:1–85:6, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1964921.1964980, doi:10.1145/1964921.1964980.

17 Sorin Lerner, Stephen R. Foster, and William G. Griswold. Polymorphic Blocks: Formalism-Inspired UI for Structured Connectors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3063–3072, New York, NY, USA, 2015. ACM. event-place: Seoul, Republic of Korea. URL: http://doi.acm.org/10.1145/2702123.2702302, doi:10.1145/2702123.2702302.

18 Silvia Lindtner, Shaowen Bardzell, and Jeffrey Bardzell. Reconstituting the Utopian Vision of Making: HCI After Technosolutionism. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 1390–1402, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2858036.2858506, doi:10.1145/2858036.2858506.

19 Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.*, 2(OOPSLA):127:1–127:28, October 2018. URL: http://doi.acm.org/10.1145/3276497, doi:10.1145/3276497.

20 A. M. Mehta and D. Rus. An end-to-end system for designing mechanical structures for print-and-fold robots. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1460–1465, May 2014. doi:10.1109/ICRA.2014.6907044.

21 Ankur Mehta, Joseph DelPreto, and Daniela Rus. Integrated Codesign of Printable Robots. *Journal of Mechanisms and Robotics*, 7(2):021015–021015–10, May 2015. URL: http://dx.doi.org/10.1115/1.4029496, doi:10.1115/1.4029496.

22 Ankur M. Mehta, Joseph DelPreto, Kai Weng Wong, Scott Hamill, Hadas Kress-Gazit, and Daniela Rus. Robot Creation from Functional Specifications. In Antonio Bicchi and Wolfram Burgard, editors, *Robotics Research*, volume 3, pages 631–648. Springer International Publishing, Cham, 2018. URL: http://link.springer.com/10.1007/978-3-319-60916-4_36, doi:10.1007/978-3-319-60916-4_36.

23 Ilan Ellison Moyer. *A gestalt framework for virtual machine control of automated tools*. Thesis, Massachusetts Institute of Technology, 2013. URL: http://dspace.mit.edu/handle/1721.1/85479.

24 Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. Functional Programming for Compiling and Decompiling Computer-aided Design. *Proc. ACM Program. Lang.*, 2(ICFP):99:1–99:31, July 2018. URL: http://doi.acm.org/10.1145/3236794, doi:10.1145/3236794.

25 openFrameworks, 2019. https://openframeworks.cc/, accessed Aug 2019.

26 Nadya Peek, James Coleman, Ilan Moyer, and Neil Gershenfeld. Cardboard Machine Kit: Modules for the Rapid Prototyping of Rapid Prototyping Machines. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 3657–3668, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3025453.3025491, doi:10.1145/3025453.3025491.

27 ROS. Unified robot description format, 2019. URL: http://wiki.ros.org/urdf.

28 Thijs Roumen, Bastian Kruck, Tobias Dürschmid, Tobias Nack, and Patrick Baudisch. Mobile Fabrication. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 3–14, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2984511.2984586, doi:10.1145/2984511.2984586.

29 Adriana Schulz, Cynthia Sung, Andrew Spielberg, Wei Zhao, Robin Cheng, Eitan Grinspun, Daniela Rus, and Wojciech Matusik. Interactive robogami: An end-to-end system for design of robots with ground locomotion. *The International Journal of Robotics Research*, 36(10):1131–1147, September 2017. URL: https://doi.org/10.1177/0278364917723465, doi:10.1177/0278364917723465.

30 Smoothieware. Modular, Open Source, High Performance G-code Interpreter and CNC Controller., 2019. http://smoothieware.org/configuring-smoothie, accessed Aug 2019.

31 Synthetos. TinyG Configuration, April 2019. https://github.com/synthetos/TinyG, accessed Aug 2019.

32 Francisca Gil Ureta, Chelsea Tymms, and Denis Zorin. Interactive Modeling of Mechanical Objects. In *Proceedings of the Symposium on Geometry Processing*, SGP '16, pages 145–155, Goslar Germany, Germany, 2016. Eurographics Association. URL: https://doi.org/10.1111/cgf.12971, doi:10.1111/cgf.12971.

33 Vention, 2019. URL: vention.io.

34 Verilog, 2019. URL: http://www.verilog.com/.

35 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, pages 69–72, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1935701.1935716, doi:10.1145/1935701.1935716.