Taxon: a Language for Formal Reasoning with Digital Fabrication Machines

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

Chandrakana Nandi
cnandi@cs.washington.edu
University of Washington
Seattle, Washington, USA

Khang Lee
leekg97@uw.edu
University of Washington
Seattle, Washington, USA

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

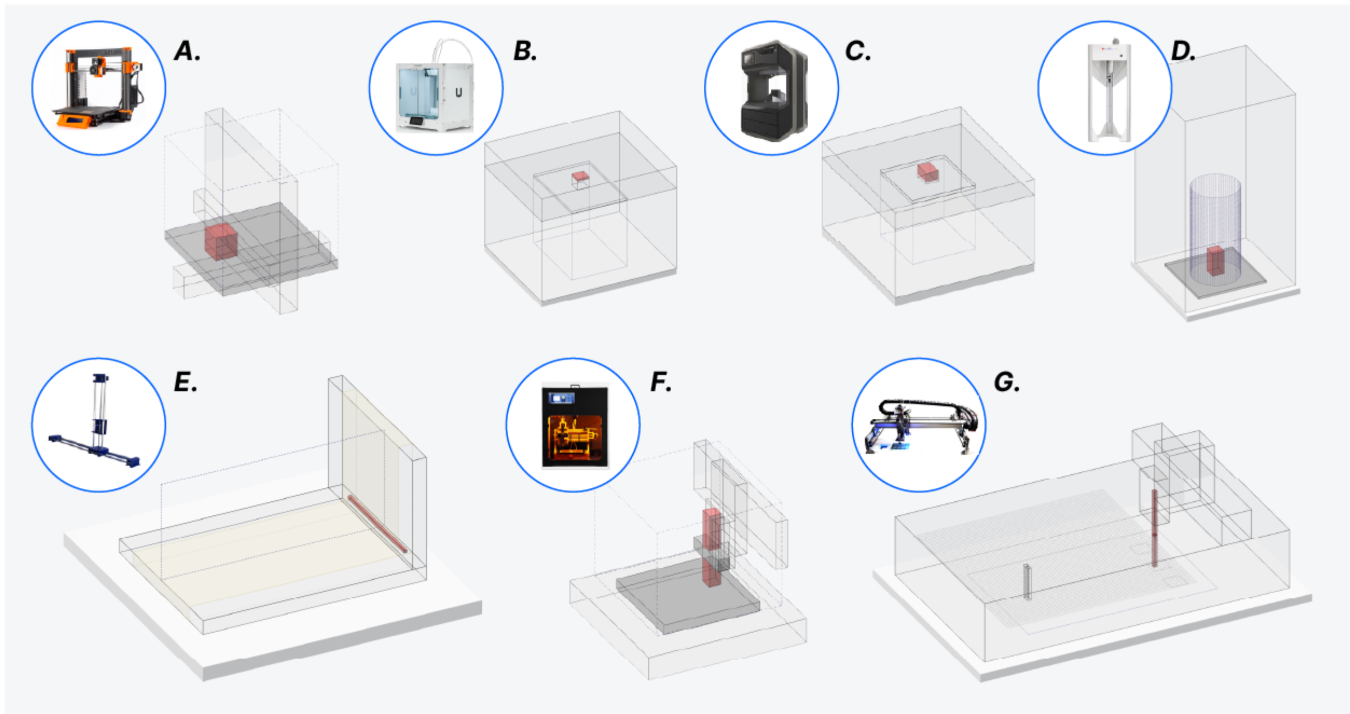

Figure 1: Taxon Concept. The Taxon language can represent a breadth of digital fabrication machines as programs that compile to abstracted machine simulations. Shown are Taxon implementations of A. Prusa i3-mk3, B. Ultimaker S5, C. Makerbot Method, D. Delta WASP 2040 Clay printer, E. hot wire cutter, F. xPrint modular liquid printer, and G. LitePlacer pick and place machine.

ABSTRACT

Digital fabrication machines for makers have expanded access to manufacturing processes such as 3D printing, laser cutting, and milling. While digital models encode the data necessary for a machine to manufacture an object, understanding the trade-offs and limitations of the machines themselves is crucial for successful production. Yet, this knowledge is not codified and must be gained through experience, which limits both adoption of and creative exploration with digital fabrication tools. To formally represent machines, we present Taxon, a language that encodes a machine's high-level characteristics, physical composition, and performable actions. With this programmatic foundation, makers can develop *rules of thumb* that filter for appropriate machines for a given job and verify that actions are feasible and safe. We integrate the language with a browser-based system for simulating and experimenting with machine workflows. The system lets makers engage with rules of thumb and enrich their understanding of machines. We evaluate Taxon by representing several machines from both common practice and digital fabrication research. We find that while Taxon does not exhaustively describe all machines, it provides a starting point for makers and HCI researchers to develop tools for reasoning about and making decisions with machines.

This work is licensed under a Creative Commons Attribution International 4.0 License.

UIST '21, October 10–14, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8635-7/21/10.
https://doi.org/10.1145/3472749.3474779

CCS CONCEPTS

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

KEYWORDS

Digital fabrication, programming languages, user interfaces, prototyping

ACM Reference Format:
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), October 10–14, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3472749.3474779

1 INTRODUCTION

Digital fabrication machines, like 3D printers, laser cutters, CNC mills, drawing machines, and laboratory pipetting machines allow a broadening base of users to program machines to create and modify physical objects. An extensive infrastructure of software and machine tools along with increasing research in human-computer interaction have unlocked more possibilities for making. Research in digital fabrication has contributed novel machines (e.g., [21, 41, 58]) and computational pipelines for creating objects with machines (e.g., [10, 27–29, 59]). Practitioners have also expanded the use of precision machines to new applications, such as ceramics [13, 46, 48], engineering education [4, 15, 39], and even fabricating personal protective equipment for hospitals in a crisis [32]. In all of these contexts, understanding how a machine works, the high-level pros and cons of a machine, and how to integrate the machine with digital and physical materials are crucial for selecting and using a machine safely and appropriately.

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

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

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

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

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

This paper's contributions include:

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

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

2 RELATED WORK

Unlike most prior work, which seeks to explain and solve specific parts of the digital fabrication pipeline, Taxon aims to formalize programmatically: machine characteristics and architecture (as metrics and blocks), manufacturing steps (as actions), and best practices (as rules of thumb). It offers a structured way to represent previously explored interactions between machine plans, actions, digital models, and materials.

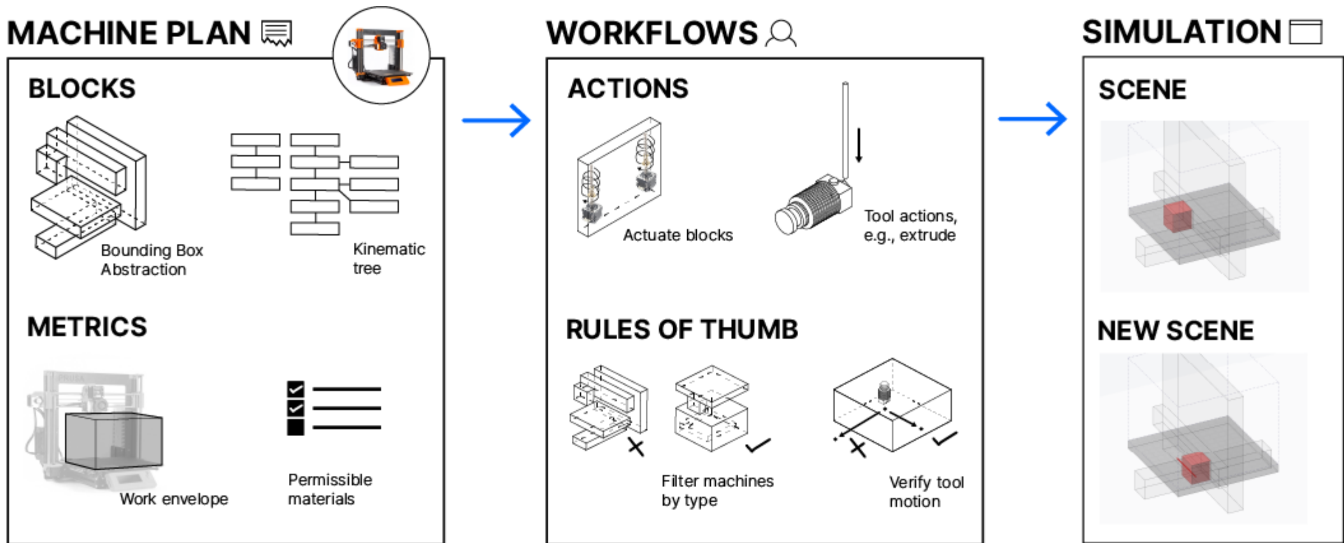

Figure 2: System Architecture. Makers represent a physical machine as a Taxon Machine Plan program. The program contains: *blocks*, which provide an abstracted composition of the machine; and *metrics*, which describe high-level machine characteristics, such as the volume that its tool can access (work envelope) and material compatibility. Taxon Machine Plans can be used in workflows containing *actions*, which simulate motion and fabrication. *Rules of thumb* are a database of community-contributed rules that can be used when analyzing machine plans and actions for appropriate machine choice for a manufacturing task or safety of actions. Executing an action updates the machine in the simulation, here, actuating the blocks such that the print head (in red) moves to the directed position.

2.1 Understanding and Building for Challenges in Fabrication

Prior research in HCI and graphics has proposed new paradigms for making with digital fabrication tools [24, 36, 68], while a smaller subset of work has critically investigated challenges that arise during the process. For example, Yildirim et al. interviewed professionals about their needs and challenges using these tools, noting that "negotiation of [fabrication] trade-offs" was a crucial part of a maker's skill set, yet was not facilitated by most tools [71]. Hudson et al. studied challenges to 3D printing for newcomers, noting that one way to reduce barriers would be "to 'weave in' expert tips, advice, and explanations throughout the printing workflow" [20]. Torres et al. examined strategies or rituals that makers adopted to account for the possibilities of failure with fabrication machines [60]. Some design researchers, such as Andersen et al. [3], Albaugh et al. [2], and Devendorf et al. [14] have foregrounded the experience of blending human craft with machine precision, along with developing mindsets that account for tensions between the two. From a high level, these studies highlight that the capabilities and limitations of a given machine must be accounted for at all parts of the making process. Schoop et al. [53] and Knibbe et al. [25] built prototypes of makerspace equipment augmented with sensors and projectors to monitor equipment use and provide alerts and recommendations to the maker. In the same vein, rather than relying solely on a maker's expertise at choosing and building applications with machines, Taxon formalizes these trade-offs and characteristics explicitly as programs, thus enforcing machine constraints throughout the process. Another example is Hofmann et al.'s PARTS plug-in for the Fusion 360 CAD program, which lets designers of 3D models embed additional information on design intent in the 3D model [19]. We build on this concept of encoding intent and constraints, but apply it to machines and their associated workflows rather than to digital models.

Currently, most checks about what is "safe" to do with a machine are implemented using computer-aided manufacturing (CAM) software. For example, a slicer is a common CAM software package that converts a digital 3D model into a tool path for a 3D printer to follow to print the modeled object, a process is called slicing. During slicing, the software can enforce conditions like ensuring the tool path fits within the printer's work envelope, or the printer does not print over thin air without having support material laid down first. However, these checks are baked into CAM software and are not portable to other contexts; in this case, they are accessible only to the maker during slicing, not before or after. We note that the popular open source slicer Cura has detailed internal representations of machine configurations [62], and the open source machine motion control firmware Marlin permits some reconfiguration of specific machine parameters [49]. Again, these representations are accessible only to each respective software tool and within its respective part of the fabrication pipeline. We argue that logic around what is safe to do with a given machine and task should not be sequestered to CAM software or machine controllers, but rather it should be available to makers at all steps of the manufacturing process.

2.2 Languages for Interactive Exploration

To design Taxon, we drew from prior work in programming languages research and information visualization research. In particular, we adopted the strategy used by the Vega and Vega-Lite grammars, where all information required to generate lower-level code for a visualization (in D3.js) is represented declaratively in JSON format [6, 50, 51]. Formalizing visualization properties in this way enables automated recommendation of visualization chart types [70] and formalizing design best practices as constraints [35]. Building on interactivity, Hempel, Mayer, Chugh, and colleagues built a system for authoring Scalable Vector Graphics (SVG) that synchronized programmatic representations of the SVG with direct manipulation edits performed on the graphic [9, 18, 34]. In the realm of digital fabrication, Nandi et al. contributed formalized languages for CAD models and triangle meshes that enabled decompilation of mesh models to CAD programs [37] along with a system for simplifying complex CAD programs into simpler ones using equality saturation techniques [38].

We also build on prior work that uses programming languages to teach programming to novices. Namely, Scratch provides a block-based syntax for teaching programming [44]. Makeblock's mBlock language builds on Scratch to help children program small robots [30]. Andersen et al. contributed an interface for programming Arduino circuits that translates high-level descriptions to lower-level circuit firmware and components [3]. Jacobs and Buechley built a language that enables designers to program fabricatable garments in the Processing creative coding tool [22]. In contrast, Taxon's focus is not necessarily to provide a simpler interface for using digital fabrication tools; rather, it aims to make machine trade-offs and safety checks elements of a programming interface. Currently, Taxon is meant as a tool for makers to investigate prior to physical fabrication, similar to existing tools like model analyzers for 3D printing [31]. Our goal is not to replace such tools, but rather to explicitly represent trade-offs and caveats in a common language. In addition, some logic from analyzer tools could be ported to Taxon as rules of thumb.

2.3 Programmatic Representations of Digital-Physical Systems

We draw inspiration from how hardware description languages for VLSI design have enabled robust program verification of desired properties in digital circuits prior to their fabrication [54]. To develop abstractions for representing machines programatically, we drew on machine design principles as formulated by Slocum [55], and abstractions about robot composition and motion from research [1, 5] and practice, as used in the common robotics middleware ROS [47]. In particular, we leveraged Mason's 1981 idea of formally modeling robotic mechanisms in a programming language, so that "the formalism serves as a simple interface between the manipulator and the programmer, isolating the programmer from the fundamental complexity of low-level manipulator control" [33]. Compared to the level of granularity common in robotics models, which typically account for torques, forces, and trajectories, Taxon's formalisms are simpler and optimized to have a lower amount of information required of the programmer. However, our goal is not immediately to provide highly accurate simulation; rather we aim to encode when and how to use machines given a fabrication task—a task that most robotics middleware assumes users have already done on their own.

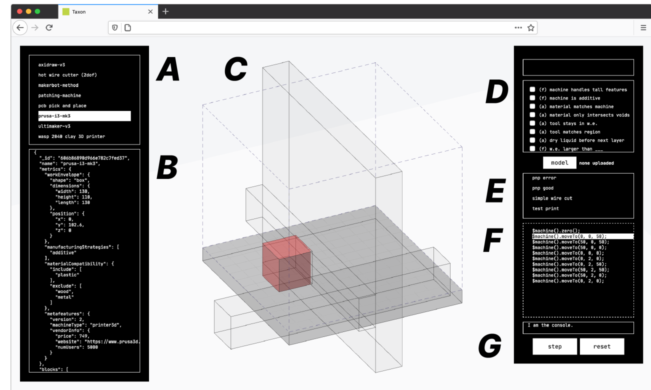

Figure 3: Taxon User Interface. The interface runs in a browser-based environment and contains the following parts: A) a database of machine plans, B) the selected machine plan containing metrics and blocks, C) the scene rendered from the current plan, D) rules of thumb—each can be double-clicked to show its implementation, E) a database of workflows, F) the current workflow containing actions that operate on the scene, model and material, executed one line at a time via G) the execution controls.

We also extend Tran O'Leary and Peek's initial exploration into representing a machine as a program [61], by Willsey et al.'s full-stack abstraction for programming microfluidics [69], and by Desai et al.'s abstractions for interactive robot design [11, 12]. Similarly, industrial tools like Vention let consumers purchase custom machines specified from a library of parts [65]. We diverge from these techniques by not prescribing modular machine components; instead, we create an abstraction that can capture a breadth of machine designs. In general, we aim to leverage prior techniques to formalize abstractions as grammars and apply these techniques to the foundational component of digital fabrication: the machine itself.

3 SYSTEM ARCHITECTURE

The system we propose has the following design goals:

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

To address these goals, we implement[1] the Taxon language to (1) represent high-level features, constraints, and composition of machines, and (2) help users compare and simulate basic machine tasks. Figure 6 shows the core components of Taxon's grammar in Backus-Naur form — a Taxon program consists of a machine-plan and a workflow :

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

- **Machine Plan:** describes the composition of the machine from a functional level and contains two parts:
  - **Metrics:** a collection of various high-level properties (*metrics*) about a machine, such as which materials the machine can use and structural considerations of a machine for fabricating certain types of models
  - **Blocks:** a collection of abstracting bounding boxes for functional parts of a machine—each known as a *block*—where each block represents volumetric and kinematic properties the given region of the machine
- **Machine Workflow:** composed of *actions*—statements that are executed by the Taxon interpreter which is specialized [16] for the current machine plan. Actions are valid Javascript statements and are a strict subset of Javascript. Actions simulate various tasks with part or all of the machine, with materials, and with uploaded 2D and 3D models to both visualize the machine and to check for potential errors using codified *rules of thumb*. Rules of thumb are user-selected static and run-time checks that filter machines (*filtering* rules of thumb) and raise warnings for actions that are risky or incompatible with a given machine (*action* rules of thumb).

Taxon also consists of a browser-based user interface that supports browsing and filtering *machine plans* (Figure 3, left) and simulating and checking *machine actions* (Figure 2 and Figure 3, right). To frame the design of Taxon, we envision and refer to two users throughout the paper:

- **The Programmer/Contributor:** a machine manufacturer or community expert who writes machine plan programs to describe real-world machines. We envision machine enthusiasts, e.g., from online hobbyist communities, contributing (1) machine plans for machines on which they have expertise, (2) Rules of Thumb that govern proper machine use as checked by the Taxon interpreter, and (3) extensions to the Taxon language.
- **The Maker:** a fabricator with a rough idea of what they would like to manufacture. These users have some background in digital fabrication but would like to explore a breadth of machines and fabrication tasks beyond their current familiarity. They are not necessarily a programmer and are not interested in modifying machine plans. They want to learn how possible machines and workflows can fabricate their ideas, compare machines based on metrics, and visualize machine performance and limitations.

These user types are not mutually exclusive; people who contribute to Taxon's infrastructure may also use it for their own manufacturing processes, and vice versa. Real-life makers exist on a spectrum between these archetypes each with varying levels of expertise and domain knowledge. We use rules of thumb avoid prescribing universal ways of using machines and also to give more flexibility to expert users. For now, we adopt the makers' perspective as we step through the user interface (shown in Figure 3) to explain how the each of the languages work to support this user.

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

Once makers select a machine, the machine plan is loaded into the plan viewer Figure 3b. The scene compiler then compiles and renders the plan into an interactive *scene* in 3D simulation Figure 3c. The scene is implemented in Javascript so the plan compiler's output is a Javascript object containing a complete THREE.js scene and animation system [8]. Makers can then double-click machine blocks in text in the plan viewer to highlight the corresponding block in the simulation, or double-click blocks in the simulation to jump to the definition in the plan viewer.

Given a selected machine plan and corresponding simulated scene, makers can now program the machine using actions Figure 3f. Pre-made action workflows from a community database are also provided Figure 3e. Actions animate individual blocks from the selected machine as well as the entire selected machine. They then select action rules of thumb from the database Figure 3d, where each rule enforces a constraint on actions in the workflow. For example, the rule *"material must match machine"* requires that any material used in the program be contained in the machine's `acceptableMaterials` metric. Once the machine, model, material, and rules of thumb are selected, the maker writes or modifies actions in the editor and steps through the program using execution controls Figure 3g. If any actions violate rules of thumb, the action pane explains the error and proposes alternative machine, model, or material choices.

4 MACHINE PLANS: BLOCKS

The core of Taxon is its representation of machines as a group of bounding boxes called *blocks*. (syntax shown in Figure 6). A block is an abstraction of subassemblies of physical machines. Each block encodes information about the machine's physical size, kinematics, and tool functionality. The goal of the blocks part of a machine plan is to provide one standard representation for many different types of digital fabrication machines; this allows us to access a diverse range of machines in one programmatic representation.

Blocks are JSON objects that are rendered as block-shaped bounding boxes in the scene. Conceptually, a block represents one functional moving unit of a machine. For example, the Prusa i3 3D printer example shown in Figure 3 is divided into three blocks each representing one axis of movement, and additional blocks for its build plate and extruder. The organization of a machine is hierarchical in that blocks have *connections* to their children, and connections are one-directional and parent-child. For example, in many 3D printers, the filament extruder is attached to a timing belt; in Taxon parlance, we say that the belt connects to the extruder, and the extruder is the belt's child.

4.1 Block Syntax

Figure 5 shows an example of using Taxon's block feature to specify parts of a Prusa-i3 3D printer [43]. Blocks both present information (e.g. the volume of various parts of the machine) and model how the machine moves in space. We define *moving* to mean that the position of a block changes; we define *actuating* to mean that an

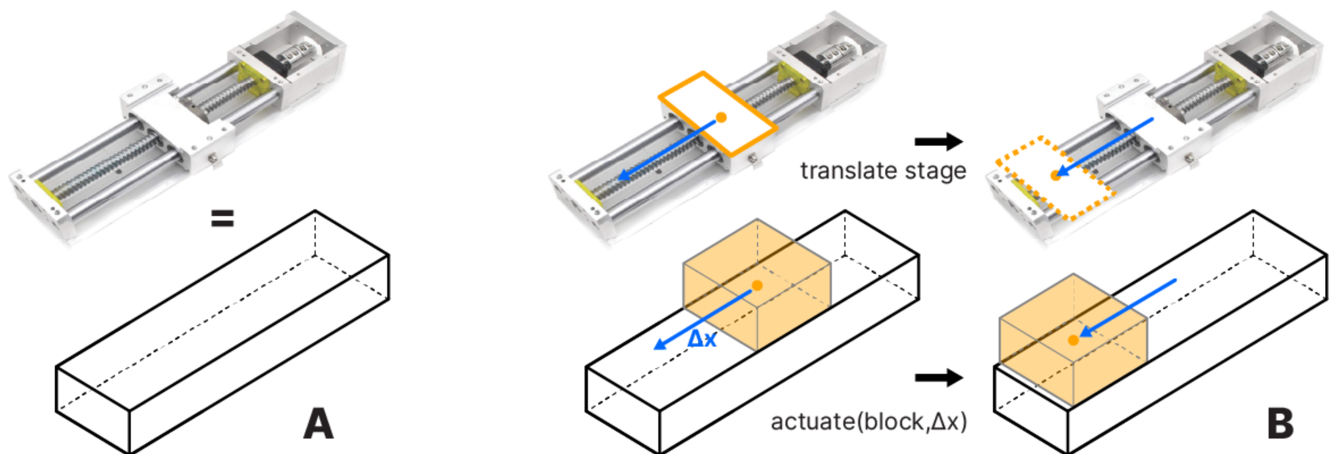

Figure 4: Principle of Actuation. In this example, a block (A, bottom) represents a linear actuator (A, top) made up of a driving motor, a lead screw, and a translating stage. When the driving motor steps, the lead screw turns, moving the stage along its actuation axis (B, top). When our block abstraction is actuated, it does not itself move, rather it moves all of its descendants (B, bottom, shown in orange) in the kinematic tree. Each block connection makes up an edge in the kinematic tree. Details such as the step-displacement ratio of a block are listed in the `attributes` property.

actuating block does not move, but all of its descendent blocks move on the actuating block's `actuatingAxes` (see Figure 4). Note that the y-axis is the "up" direction in our convention. All blocks contain a `dimensions` field denoting the size of its bounding box, while a block's *position* is coordinates of the box's centroid. A block's position is defined *explicitly* through the `position` property or *implicitly*, where the block is a child to another block and its position snaps to its parent.

Each block has a `blockType` depending on the type of actuation it supports and the number of motors it contains. The bounding box of a block is assumed to contain the motors which e.g., *step* (turn a fixed angular amount) to drive its actuation. For example, the most common type of block is a *linear block* which is driven by one underlying stepper motor and has exactly one actuation axis; a linear block provides one degree of freedom of movement. Examples of linear blocks include each of the actuating blocks in the Prusa 3D printer machine plan. In contrast, a *cross block* represents a more complex assembly with parallel kinematics, where multiple underlying motors interdependently control multiple axes. Examples of cross blocks include the block representing the XZ axes in the Jubilee machine plan, which are driven by CoreXY kinematics [64]. *Redundant linear blocks* are driven by more than one motor but move in a single degree of freedom; such blocks typically represent linear synchronous motion, for example, the two vertical lead screws on the Prusa 3D printer. Not all blocks actuate. An example of such a *non-actuating* block is the build plate of a 3D printer; while the build plate itself moves, it neither actuates nor provides additional movement to any other blocks. Finally, a tool is a special class of block whose movement can change the scene—for example, extruding 3D printer filament, carving material out of stock, or picking and placing an object. A block is designated as a tool by setting `''isTool'': true`.

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

4.1.1 Declaring Motors Explicitly.

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

4.2 Connections

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

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

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

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

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

Figure 5: (Top Left) Top level properties of a machine plan: name, metrics, and blocks. (Top Right) Example of an actuating block. The block type is `redundantLinear`, and the actuation axis is the y-axis. Attributes are additional properties specific to the block. (Bottom Left) Example metric. The `work envelope` is the volume within which the machine's tools can move. The metric also describes material compatibility, the manufacturing strategy, and machine features like resolution, and travel speed. (Bottom Right) Example of a (non-actuating) tool block. Not all of this machine's blocks are shown. Note that y-axis is the "up" direction in our convention.

⟨*name*⟩	::=	string width $\in \mathbb{R}$ height $\in \mathbb{R}$ length $\in \mathbb{R}$
⟨*position*⟩	::=	$(\mathbb{R}, \mathbb{R}, \mathbb{R})$
⟨*dimensions*⟩	::=	width height length
⟨*connection*⟩	::=	block offset offset $\in (\mathbb{R}, \mathbb{R}, \mathbb{R})$
⟨*actuation-axis*⟩	::=	X | Y | Z
⟨*attribute*⟩	::=	is-tool
	|	is-platform
	|	drive-type
	|	step-displacement-ratio
	|	nozzle-count | ...
⟨*block-type*⟩	::=	non-actuating
	|	linear
	|	redundant-linear
	|	cross
	|	delta-bot
⟨*block*⟩	::=	name block-type dimensions [position] actuation-axis* attribute* connection*
⟨*work-envelope*⟩	::=	shape dimensions position
⟨*metric-value*⟩	::=	work-envelope | $\mathbb{R}$ | string | object | ...
⟨*metric*⟩	::=	name metric-value
⟨*shape*⟩	::=	box | cylinder | rectangle | ...
⟨*machine-plan*⟩	::=	name metric* block*
⟨*method*⟩	::=	MoveTo (machine-plan, position, $\mathbb{R}$)
	|	Actuate (block, $\mathbb{R}$) | ...
⟨*constructor*⟩	::=	SelectBlock
	|	SelectMachine
	|	SelectTool
	|	SelectMaterial
	|	SelectModel | ...
⟨*selector*⟩	::=	constructor name
⟨*action*⟩	::=	selector method
⟨*workflow*⟩	::=	action*
⟨*program*⟩	::=	machine-plan workflow

Figure 6: Syntax of core components of the Taxon Language. Reading bottom-up, a Taxon program consists of a machine-plan and a workflow. A workflow is a list of zero or more actions. An action consists of a selector and a method to be called on the component returned by the selector. The selector returns a component of the machine-plan (which could be a block or the entire machine plan). Examples of methods are `MoveTo` which moves a component to a position (possibly extruding material at a rate indicated by the third argument) and `Actuate` which actuates a block by some amount given as the second argument. A machine plan consists of metrics and blocks. Metrics are a collection of innate properties of a machine, and blocks represent volumetric and kinematic properties of machine components. Sections 4 and 5 describe these components in detail.

defined in relationship to the parent and moving when the parent actuates.

For example, in Figure 7, `verticalLeadScrewFrame` shows a connection with the `crossbarAssembly`. In this case, the crossbar assembly is the child of the vertical lead screw frame, which does two things. First, the position of the child block "snaps" to the position of the parent block such that their centroids occupy the same point, plus the user-defined `offset`. In this example, the crossbar assembly is positioned in the lower front of the vertical leadscrew frame. Second, a connection produces a directed edge in the machine's *kinematic graph*, with the parent and child blocks as nodes. Actuating any node in a machine's kinematic graph actuates its sub-tree along the same axes. For example, if the vertical lead screw frame were actuated, the crossbar assembly and any of its own descendants would move up or down in the parent's sole actuation axis. A block may have more than one child.

5 MACHINE PLANS: METRICS

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

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

As an example, we describe the metrics for the Prusa 3D printer as shown in the bottom right code listing of Figure 5.

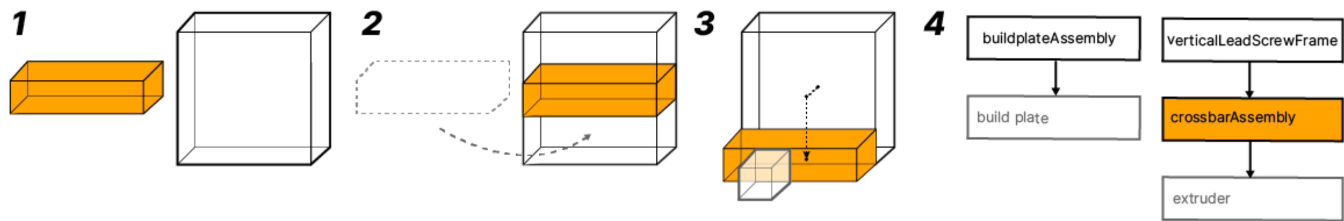

Figure 7: Building a Kinematic Tree with Connections. This example uses two blocks, the crossbar block (1, Left) and the lead screw frame block (1, Right) from the Prusa-i3 3D printer example. 1) A connection involves a child (left) and parent block; 2) the child block is translated such that its centroid is at the same point as the parent's centroid; 3) a user-specified offset translates the child relative to the parent's centroid, resulting in the child's final position. Any descendants of the child block (gray) receive the same total translation. 4) shows the kinematic tree for the entire 3D printer, with black arrows denoting connections.

- **Machine Class:** a broad designation of this machine's class, e.g., 3D printer, laser cutter, mill, etc.
- **Work Envelope:** the bounding box in which the tool can safely move. The size of the machine's work envelope limits the size of the models it can manufacture. Mobile machines such as the Piccolo plotter [52] could be represented as having work envelopes with infinite dimensions on certain axes.
- **Manufacturing Strategies:** can include additive, subtractive, drawing, pick and place, etc.
- **Material Compatibility**: a list of materials that the given machine is known to be able to work with (e.g., plastic for the 3D printer) and a list of materials that the machine should explicitly not work with (e.g., metal for any 3D printer). Materials not contained in these lists have unknown compatibility with the current machine, and users must exercise caution if using them.
- **Resolution:** the minimum movement in millimeters that the machine can support. Machines with lower resolutions cannot support very fine features in models.
- **Max Travel Speed:** the maximum speed in millimeters per second that the machine's tool can move while not extruding or cutting material
- **Metafeatures:** a collection of various features that do not affect the machine's operation, for example, its cost, website, and estimated size of its active user base.

6 WORKFLOW AND ACTIONS

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

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

In addition to controlling the machine, actions let makers load materials and digital models into the scene. Like blocks, materials are represented by a bounding box that is placed in the scene, shown in Figure 8 (right). Each material must have a *material class*, which can be *additive*, i.e., it is deposited in the scene by actions with additive machines, or *subtractive*, i.e., makers must place the material first and then process it using tool actions. Materials also have names, e.g., "wood" or "plastic," which are typically listed in a machine's `materialCompatibility` metric. This high-level description of materials suffices for high-level simulations. Actions also support uploading digital files (e.g., STL files) and placing them in the scene. Taxon's modular design affords easy extensions to the language for slicing STL files into tool paths and processing geometry; we leave these for future work.

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

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

Figure 8: (Left) Workflow testing various movements on the Prusa 3D printer. (Right) Workflow loading sheet material and a 3D model, creating a toolpath from the model, positioning the material and model, and running the mill over the toolpath.

6.1 Machine Action Language Example

To illustrate Taxon, we present the examples in Figure 8 where a novice user tests various actions using the Prusa 3D printer featured in previous examples. The selector evaluates to an object of type `Block`, `Tool`, `Machine`, `Model` or `Material`, where each type supports its own set of methods. For example, the first two lines of Figure 8 select the blocks from the running 3D printer example (type `Block`); then call `.wiggle()` on the crossbar assembly, causing it to actuate back and forth, and `actuate()` on the vertical lead screw frame, causing it to actuate upwards. The next statement selects the printer's extruder (type `Tool`) and extrudes 10 millimeters of filament. The following statement has `$machine()`, which selects the entire `Machine` object and calls `.zero()`, which establishes a Cartesian coordinate system with the machine's tool's current position as the origin. For any actions executing coordinate movement, a coordinate system must be instantiated first by calling `.zero()`. Finally, given the coordinate system, `.moveTo(coords)` moves the machine's tool to the (x, y, z) coordinates specified in `coords`. If a bounds-checking rule of thumb is enabled, the interpreter throws an error if the coordinates are outside the machine's work envelope.

6.2 Checking Machine Actions with Rules of Thumb

The motivation behind simulating machine actions is twofold: first, we want users to be able to visualize how a given machine works and to experiment with it, and second, we want users to be able to formally state what they want to do with a given machine in order to enforce best practices. In particular, the act of formally describing one's steps and using error checking offers a useful way to introduce machine knowledge to users. Rather than merely presenting the user with a list of best practices, users can program various machine actions and learn about potential issues and errors in the context of what they are trying to do.

Our solution to these goals are *rules of thumb*, which analyze machine plans and actions to offer suggestions and enforce safe machine usage. Rules of thumb are modular and stored in a community-contributed database; users can select which rules they want to enforce for a given task. To enable these rules of thumb, the user selects which rules they wish to enforce in the Action pane (Figure 3g). There are two types of rules of thumb:

- ***Filtering* rules of thumb** examine only the machine plan, filtering a list of feasible machines. For example, users planning to use a 3D printer would likely want to require that any 3D model they will print to fit within the printer's work envelope. Or, someone who is 3D printing a model with tall and skinny features would want to select a printer design that is well equipped to print those features without excessive ringing or vibration.
- ***Action* rules of thumb** examine both the machine plan and actions in the workflow that the interpreter executes. For example, milling machines should have their end mills intersect with and carve away stock material only when the spindle is actively spinning the end mill; otherwise, the end mill will break. One rule of thumb might be to check that whenever a move command is executed with a milling machine, its spindle will be turned on if that move would intersect with any stock material.

Figure 9 shows the implementations of two rules of thumb in Javascript, where programmers can access the machine plan as Javascript objects with the same selector syntax used in action programs. Because the machine plan is a JSON object, it can be parsed and traversed in Javascript in the rule of thumb's implementation. The first rule, "model must fit in envelope," is an action rule of thumb that fires whenever the method `Model.placeAt()` is called; here, the rule checks that the model placed in a scene can fit in the current machine's work envelope. The second rule of thumb, "machine handles tall features," is a filtering rule which removes

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

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

Figure 9: Rules of Thumb (Action and Filtering) implemented in Javascript. The first fires when an action attempts to load a digital model into the scene; if an error is thrown during checking, e.g. if a metric is `undefined`, then the implementation reports that it could not complete its checking, but returns `true` as a default. The second checks that a machine does not use a platform moving in non-vertical directions; machines meeting this property are better suited to print tall features.

from the machine plan list any machines unable to manufacture tall and skinny features due to their composition.

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

7 EVALUATION: ADDING TO THE LANGUAGE THOUGH DEMONSTRATION PROGRAMS

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

7.1 Evaluation Method

The goal of our evaluation is to understand where Taxon can—and in some cases, cannot yet—gracefully extend to different fabrication tasks. To evaluate Taxon's expressivity, we investigate six demonstration[2] machines that showcase ways of formalizing existing machines and workflows. We use Taxon to represent the breadth of machine types and workflows used in both common practice and digital fabrication research. Table 1 summarizes the different machines we use for evaluation, which are also shown in Figure 1. Each demonstration is more than just a proof of concept — for each machine and new workflow, we highlight challenges and discuss new rules of thumb and new constructs in the action part of the language. We then implement these new features and add them to Taxon's core. For example, in implementing pick and place machines, we designate parts of the work envelope as *regions*, namely, regions to store parts that the machine will later place, and a region dedicated as a parking location for tool changes. Further, we add rules of thumb based on the regions. For example, we add rules of thumb that enforce that the region where parts are stored cannot overlap with the PCB material, and regions that hold very small parts can be entered only when the appropriate tool for handling small SMD parts is currently in use.

Note that this paper does not assess usability: it focuses instead on flexible infrastructure and language expressivity that let users build new interactions on top of a formal foundation. Furthermore, because the concept of establishing formal checks on fabrication tasks is relatively novel, and therefore lacks precedent, it is premature to compare Taxon's usability to a non-existent baseline [17, 40]. Once the language has developed sufficient new settings, we can evaluate its ease of use. In the meantime, our primary concern is evaluating the expressivity of the language.

7.2 Comparing Off-the-Shelf 3D Printers

As an initial task, we implemented the metrics and blocks for 3 different commercial 3D printers: the Prusa i3-mk3—described in the prior sections—the Makerbot Method, and the Ultimaker S5. In addition, we implemented the metrics only for 20 different 3D printers; all of these machine plans can be found online in our repository. These printers vary significantly in price, material compatibility,

[2] We use Ledo et al.'s definition of a demonstration as an evaluation [26], which Zhang et al. exemplify in evaluating their programming language for online community governance [72].

Table 1: Demonstration Machines and Workflows. Entries in the rightmost column denote the lines of code in the machine plan, the workflow, and associated rules of thumb, respectively. *Average lines of code per machine plan.

Taxon Program				
Machine Plan	*Workflow*	*Rules of Thumb*	*Additions To Core Language*	*Lines of Code*
Three Off-the-Shelf 3D Printers	N/A	"work envelope larger than ___"	dependent rules of thumb	(132*, 0, 15)
Hot Wire Cutter	cuting an airfoil from styrofoam	"material intersects voids only"	work envelope orientation, collision voids	(99, 19, 20)
Wasp Clay 3D Printer [67]	basic clay print	N/A	delta bot block, cylindrical work envelope	(102, 16, 0)
Wang et al. xPrint [66]	controlled deposition of natto cell culture	"dry liquid before next layer"	active tools	(209, 13, 21)
Liteplacer Pick and Place Machine [23]	placing SMD components in footprints on a PCB	"tool matches region"	envelope regions, tool changing	(259, 12, 24)

and construction, and comparing many printers can quickly become tedious. As metrics for each printer, we added the machines' work envelope dimensions, material compatibility, resolution, and other features listed on the printer's website. When implementing machine plans for these printers, our choice of blocks varied based on each printer's physical construction. For example, the Prusa's construction uses a build platform for the in-progress print that moves side-to-side, whereas the Ultimaker's platform only moves vertically. As a result, the in-progress print for the Ultimaker never moves side-to-side since all such motion is accomplished by the uppermost assembly which only moves the extruder head. This construction is useful when printing tall and skinny features (Figure 9); however, the Prusa is cheaper, and, depending on the user's model, such construction might not be necessary. In addition, as attribute properties for each actuated block, we identify its type of drive mechanism (e.g., lead screw, timing belt, or a rack and pinion). This information could be used to implement rules of thumb that optimize machine choices based on choice of drive train, e.g., filtering for machines with all lead screw actuators due to their ability to handle high-force applications.

Makers use filtering rules of thumb to compare different machines. Again, the choice of rules of thumb for filtering rather than built-in sorting features is because the requirements that makers would want to filter around change based on the task as well as over time as machines change. However, in this case, a rule of thumb may need additional information from users; for example, users might not have a particular model in mind, yet they still want to see only 3D printers with work envelopes larger than a certain value. To support this, we implemented *dependent rules of thumb*, which take into account a value from the user when filtering (see Figure 10). For example, the dependent rule of thumb named `''work envelope is larger than ___''` prompts the user to enter a bounding box with a desired width, height, and length. The rule then filters for machines that have a work envelope with a box shape whose dimensions equal or exceed the values listed. The notion of dependent rules of thumb comes from the concept of dependent types in programming languages, which are types (e.g. `String` or `Array`) that depend on a value—for example, a type representing all arrays of length 3 [7]. In this light, this dependent rule of thumb implicitly assigns a dependent type to all machines based on the dimension of their work envelope and filters them accordingly.

7.3 Cutting a Styrofoam Airfoil with a Hot Wire Cutter

Hot wire cutters work by heating a suspended wire with an applied voltage and then moving the wire through a work piece of styrofoam, which cuts the material by melting it. In this example we model a typical CNC (Computer Numerical Controlled) hot wire cutter topology with two degrees of freedom, namely, vertical and horizontal wire movement (see Figure 11). We observed that both redundant linear blocks consume significant space for their bounding boxes that, in reality, are empty space where the tool (the wire) and the workpiece would go. In the hot wire cutter's implementation, blocks in this machine readily intersect with one another. Assume we want to create rules of thumb that distinguish between an intersection that is physically harmless versus an intersection where machine parts are actually crashing into one another. A "harmless" intersection in this case is how the foam work piece's volume might intersect with the vertical machine block's empty space. An unacceptable intersection occurs when the work piece intersects with the vertical block's solid parts. To this end, we implemented the notion of a *void*, an optional property that can be added to a block's attributes property. A void is a bounding box that fits in the block's bounding box that represents empty space within the block. As a result, we can be confident that other blocks intersecting with a block in its void will not cause a collision. We implemented a rule of thumb to check that a machine does not collide with itself, referencing blocks and voids. Though this rule is not strictly necessary for the hot wire cutter, it would likely be

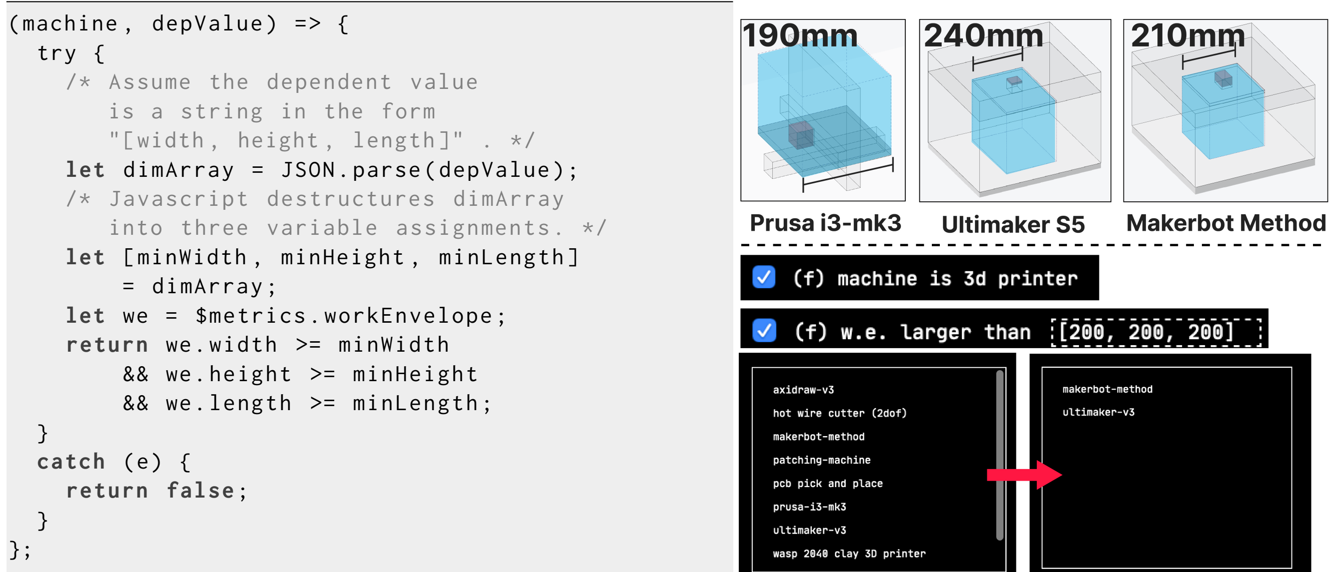

Figure 10: Dependent Filtering Rule of Thumb for Comparing 3D Printers. (Left) Implementation of the rule of thumb that takes an additional argument from the user interface, `depValue`, representing the minimum allowable size of the machine's work envelope. (Right, Top) The three commercial 3D printers in our evaluation rendered in the scene which the work envelope highlighted in blue. (Right, Bottom) The checked rules of thumb filter the machine plan list down to only two satisfactory machines.

useful for more complex machines and workflows where collisions are more likely to happen.

7.4 Printing with Clay

We implemented a Delta Wasp 2040 Clay 3D printer and an example workflow for it that 3D prints a vase out of clay [67]. This printer uses delta bot kinematics, which feature three levers connected with revolute joints to a rigid body. The end of each lever that is not fixed to the rigid body moves independently via a drive mechanism, and the position of the levers' ends determines the location of the rigid body. As a result, the rigid body has three Degrees of Freedom (DOF) and depends on three or more motors to drive the levers, which exceeds the linear (1 DOF) and cross (2 DOF) blocks that we have thus far implemented. Therefore, we implemented a 3DOF `DeltaBotBlock` that encompasses the majority of the printer; in fact, the printer's machine plan has only three blocks: the delta bot block, the tool, and an immobile platform. One feature of delta bot kinematics is a cylindrical work envelope as opposed to a box-shaped one, which we implemented as well.

In addition to the delta bot block and the cylindrical work envelope, we could not feasibly address other important features in the Wasp printer with the current level of abstraction in Taxon. In formative interviews with makers, we learned that a key challenge in 3D printing with clay comes from the material properties of clay itself. In all such 3D printers, the clay must be packed into a container tube and forced evenly through the nozzle. The Wasp printer uses an external compressed air source along with a rotating auger for this. Unlike the plug-and-play nature of plastic 3D printer filament, here the maker must mix their own clay and find the right consistency: too watery and the extruded material may collapse, or too thick and the clay will extrude unevenly. One ceramicist noted that she knew from her own experience with clay what an ideal consistency felt like, how to vary it based on the type of print she was attempting, and when and when not to manually intervene during a print to reshape features by hand. To codify this sort of tacit knowledge, Taxon would need to support materials with enough detail to do basic physics simulation, and, even then, such rules of thumb based on the material properties of clay would be "ballpark" estimates at best. We decided that this level of physics-based material representation is beyond the current scope of Taxon but would be fruitful future work.

7.5 Bio-actuated Textiles

In this demonstration, we replicated a workflow that Wang et al. demonstrated with xPrint, a machine with modular tooling for printing with liquid-based smart materials [66]. They created discrete components—such as a liquid dispenser, a mechanical stirrer, and a ventilator unit—that can be manually attached and removed as necessary for the desired workflow. In addition, they created a plug-in for the Grasshopper/Rhinoceros CAD program that lets makers create custom tool paths for the machine. We focused on one workflow they presented: depositing liquid natto cell culture onto textiles, which makes the textiles curl when exposed to moisture (see Figure 12). Depending on the pattern of culture deposited, makers can control the direction and degree of curl. In Wang et al.'s paper, this workflow involved attaching a solution container

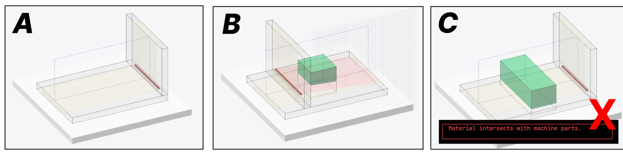

Figure 11: Hot Wire Cutter Example. A) The hot wire cutter rendered in the scene, where the red block is the tool (wire), the dashed line is the work envelope, and the goldenrod regions are user-defined voids within the non-tool blocks (gray). B) A simple workflow consisting of placing a block of foam material, zeroing the machine, and moving the wire to cut through the foam with the cut plane illustrated in pink. C) If the material is placed in such a way where it intersects the blocks outside the voids, i.e. overlapping with machine parts, it triggers an error from a rule of thumb that requires that materials may only intersect blocks in their voids.

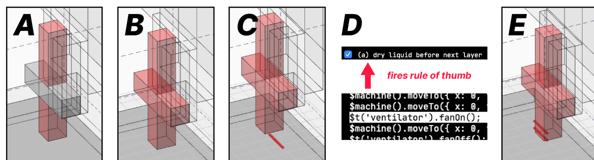

Figure 12: Depositing Natto Cell Culture with xPrint. A) This xPrint configuration contains a dispenser (red) and an inactive ventilator (dark gray). B) An action activates the ventilator (now red), allowing the user to turn its fan on and off. C) Extruding a thin layer of culture with a `moveTo` action. D) Calling `fanOn` dries the current layer, which triggers the relevant rule of thumb "dry liquid before next layer" to update its state. E) Depositing the second layer. If the first layer were not dried with the fan, the rule of thumb would have thrown an error.

equipped with a mechanical stirrer for keeping the natto cells suspended, a dispenser for dispensing the culture, and a combined ventilation unit in addition to a heated bed to evaporate the current layer of culture before the next layer is deposited.

Despite the complex nature of tooling in this workflow, we found that implementing the machine plan and the workflow in Taxon was less challenging than expected. This is largely because much of the complexity comes from the choice of tooling to install, rather than from the process of choosing or controlling the machine. While multiple tool components can be attached to the machine at any given time, once attached, no further automated tool changes are involved. In addition, the machine itself is a fairly straightforward 3-axis machine, which we implemented using three linear blocks: one non-actuating block for the build plate, one non-actuating block for the tool component substrate, and non-actuating tool blocks for the dispenser, solution container, and ventilator. We implemented rules of thumb that govern which tools must be equipped to work with a given material. In our case, if the xPrint machine were selected, and if natto cell culture were used as a material, then the machine would need a dispenser, solution container, stirrer, and ventilator.

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

Figure 13: Implementation of the "dry liquid before next layer" Rule of Thumb. This action rule of thumb accesses the `store` argument which references a state that the program keeps between different executions of the rule of thumb. This allows the implementation to keep state for whether the most recent layer was dried by the fan or not.

To implement the notion of "active," we added a new property to a tool block's attributes property, called `active`, which can be true or false. If false, while the tool is included in the machine's plan, it is not rendered to the scene and cannot be accessed by the action interpreter unless marked with the `.activate` method. We also implemented a rule of thumb (shown in Figure 13) that requires the maker to dry the current layer with the fan before moving on to the next layer.

7.6 Robotic Assembly of PCB Components

We implemented the Liteplacer pick and place machine, an open source, do-it-yourself machine kit created by Juha Kuusama [23]. This machine picks up SMD components for printed circuit boards

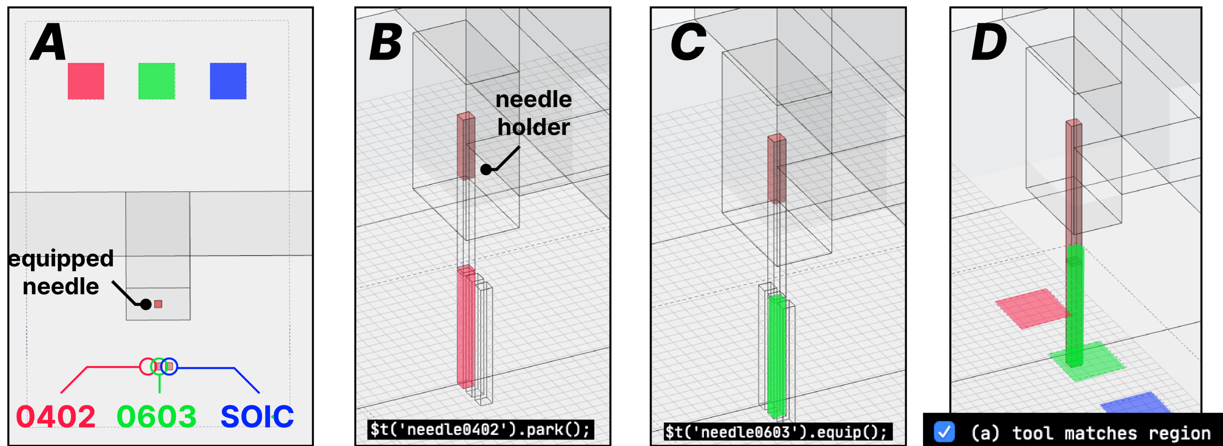

Figure 14: Verifying Correct Tool Configurations in the Pick and Place Machine. A) A bird’s-eye view of the pick and place machine’s setup with three differently sized needles for picking up SMD components of respective sizes (0402, 0603, and SOIC-8) and placing them on a PCB. At the bottom are three regions for parking each needle and at the top are three regions where each type of component is stored before being picked. B) The needle holder travels to the 0402 parking region and parks the needle there. C) The needle holder travels to the 0603 parking region and picks up that needle. D) The needle holder moves to the 0603 storage region successfully because the 0603 needle is currently equipped. If the needle had moved there before swapping needles, the “tool matches region” rule of thumb would have thrown an error because the needle would be incorrectly sized for picking up 0603 components.

and places them precisely within the components’ respective footprints on the PCB board itself. The machine receives as input locations of the footprints on the PCB and uses a camera-based localization routine to zero itself and calculate trajectories for moving components from the supply region to the footprints.

This is a relatively complex machine and workflow. For the machine plan, we implemented the machine to use two linear blocks for both the lower assembly, which moves along the x axis, and the crossbar assembly, which moves on the z axis. The tool assembly was more complicated to model, consisting of a linear block that moves on the y axis, a *rotary block* that rotates any PCBs that are currently picked up, and several suction-cup-tipped needles for actually picking up the SMD components. We chose not to implement rotary blocks for now because rotational motion complicates our model of machine motion; it would require us to solve general-case inverse kinematics rather than the simpler cases with only linear motion. Instead, we focused on the machine’s tool changing feature and on correct tool usage, as shown in Figure 14.

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

In this light, a goal of Taxon is to codify best practices—here, ensuring that the correct needle is attached before attempting to pick up an SMD component. Enforcing this logic necessitated the creation of a new feature, *envelope regions*, which lets the programmer designate and label subspaces within the work envelope. Similar to a work envelope itself, an envelope region can be specified as a box that checks for inclusion within the region based on x, y, and z coordinates or as a rectangle that checks only two coordinates. We created a rectangle XZ region where each needle is parked, a region in which to place stock SMD components corresponding to each size of needle, and a region in which to place the PCB itself. An automated tool changing routine would involves traveling to the region for parking the current tool, having the machine deposit the current tool, traveling to the region to park the new tool, and then attaching it to the tool holding assembly. We then implemented an action rule of thumb (shown in Figure 15) stating that before a tool can enter a region for stock SMD components, it must have the correctly sized needle currently attached.

8 LIMITATIONS

While implementing the six demonstration programs for the evaluation, we encountered several limitations. First, simulating a workflow often requires us to reason about how a material will react when acted upon by the machine. For example, in the case of the hot wire cutter, having the wire pass through a styrofoam workpiece put foam in the path of the wire. Or, for the clay 3D printer, the consistency of the clay greatly affects its behavior when deposited by

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

Figure 15: Implementation of the "tool matches region" Rule of Thumb.

the extruder. However, modeling the physical behavior of material—even in a greatly simplified setting—remains challenging. Because material properties always affect the semantics of actions, they must be implemented in the core Taxon language, as opposed to being added in a modular fashion using rules of thumb. The need to incorporate more physics into Taxon than we initially anticipated created a bottleneck that we must address before more workflows can be fully represented in code.

In addition, nontrivial geometry processing is not yet implemented in Taxon. For the evaluation, we attempted to replicate the workflow proposed in a paper by Teibrich et al., which features a machine that can patch existing 3D printed objects by removing material with a mill and then reprinting new material with an 3D printer extruder [57]. These subroutines depend on access to the 3D model, the tool path, and robust software for detecting collisions. While it is possible to implement these subroutines as rules of thumb, such as "avoid collisions by milling material that blocks access" and "rotate the build plate to minimize milled material", we would need to add substantial geometry processing to implement such collision detection and optimization programatically. Currently in Taxon, we can reason only about collision with bounding boxes, not with the precise geometry of the work piece. We defer such features to an extension in future work.

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

9 FUTURE WORK: AFFORDANCES OF FORMAL REPRESENTATION

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

- **Structured Querying for Machines**. Similar to how the Voyager tool enabled structured exploration of visualizations based on their grammars [70], we could allow for structured exploration of machine options beyond filtering rules of thumb given a user-specified workflow. Makers would then be able to explore trade-offs along speed, precision, cost, and other factors that are influenced by machine choices for the workflow.
- **Optimizing Instructions for Machine Kinematics**. In many machines, dedicated control software translates instructions, e.g., in G-Code, into motor pulses to optimize machine movement based on its physical characteristics. Given a Taxon program that lets users infer volumes, masses (in the future), and kinematics, it would be possible to optimize machine instructions for any machine rather than hard-coding these optimizations for one machine. One example optimization is *input shaping*, which involves generating signals that cancel vibrations associated with moving machine parts and result in higher quality surface finish [45].

- **Online Infrastructure for Sharing Machines and Workflows**. Taxon makes possible a rich online ecosystem of machine and workflow descriptions. Members of online communities could share their custom machine builds and small production tasks as Taxon programs. Other members could run these custom setups in simulation and remix Taxon programs. Taxon could serve as a quasi API for creating Instructables-like tutorials with interactive previews built into each step. In addition, existing open-source specifications of machines such as Cura's 3D printer profiles [62] could be ported to Taxon machine plans.
- **Scheduling High-Throughput Production in Existing Maker Spaces**. Maker space managers could create a list of machine plans that represent the machines they have available in their maker space. Given a large production task, such as producing personal protective equipment requiring multiple machines, programmers could write tools that solve for the optimal allocation of production tasks to machines in the maker space.
- **Program Synthesis for Fabrication.** We can view a digital fabrication pipeline as a compiler that compiles a digital model into a physical object. In computational fabrication, *inverse design problems* solve for digital models given some physical constraints, e.g. the design of nanophotonic devices given nanoscale fabrication limitations [42]. However, inverse problems to solve for machines themselves have not been explored. By providing a formal specification and programmatic description for machines, Taxon could let researchers use program synthesis techniques [56, 63] to infer a machine specification that best suits the fabrication of a digital model under physical constraints.

10 CONCLUSION

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

This work was supported by the Alfred P. Sloan Foundation's Technology Program. We would like to thank: Gabrielle Benabdallah and Blair Subbaraman for their feedback on early versions of Taxon; Rastislav Bodik, Sarah Chasins, Eunice Jun, and Zachary Tatlock for their numerous insights on programming language design; and the anonymous reviewers for their helpful and constructive critique.

REFERENCES

[1] E. Aertbeliën and J. De Schutter. 2014. eTaSL/eTC: A constraint-based task specification language and robot controller using expression graphs. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1540–1546. https://doi.org/10.1109/IROS.2014.6942760 ISSN: 2153-0866.

[2] Lea Albaugh, Scott E. Hudson, Lining Yao, and Laura Devendorf. 2020. Investigating Underdetermination Through Interactive Computational Handweaving. In *Proceedings of the 2020 ACM Designing Interactive Systems Conference (DIS '20)*. Association for Computing Machinery, New York, NY, USA, 1033–1046. https://doi.org/10.1145/3357236.3395538

[3] Kristina Andersen, Ron Wakkary, Laura Devendorf, and Alex McLean. 2019. Digital Crafts-Machine-Ship: Creative Collaborations with Machines. *Interactions* 27, 1 (Dec. 2019), 30–35. https://doi.org/10.1145/3373644 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[4] Camille Andrews. 2017. Learning and Teaching in Library Makerspaces: A Literature Review on Making Literacies. In *Proceedings of the 2nd International Symposium on Academic Makerspaces*. Cleveland, OH, USA.

[5] Gianni Borghesan, Enea Scioni, Abderrahmane Kheddar, and Herman Bruyninckx. 2016. Introducing Geometric Constraint Expressions Into Robot Constrained Motion Specification and Control. *IEEE Robotics and Automation Letters* 1, 2 (July 2016), 1140–1147. https://doi.org/10.1109/LRA.2015.2506119 Publisher: IEEE.

[6] 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.

[7] Ana Bove and Peter Dybjer. 2009. Dependent Types at Work. In *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto (Eds.). Springer, Berlin, Heidelberg, 57–99. https://doi.org/10.1007/978-3-642-03153-3_2

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

[9] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016* (2016), 341–354. https://doi.org/10.1145/2908080.2908103 arXiv: 1507.02988.

[10] Ruta Desai, James McCann, and Stelian Coros. 2018. Assembly-aware Design of Printable Electromechanical Devices. ACM, 457–472. https://doi.org/10.1145/3242587.3242655

[11] Ruta Desai, Margarita Safonova, Katharina Muelling, and Stelian Coros. 2018. Automatic Design of Task-specific Robotic Arms. *arXiv:1806.07419 [cs]* (June 2018). http://arxiv.org/abs/1806.07419 arXiv: 1806.07419.

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

[13] Audrey Desjardins and Timea Tihanyi. 2019. ListeningCups: A Case of Data Tactility and Data Stories. In *Proceedings of the 2019 on Designing Interactive Systems Conference (DIS '19)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/3322276.3323694

[14] Laura Devendorf, Abigail De Kosnik, Kate Mattingly, and Kimiko Ryokai. 2016. Probing the Potential of Post-Anthropocentric 3D Printing. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems (DIS '16)*. ACM, New York, NY, USA, 170–181. https://doi.org/10.1145/2901790.2901879

[15] Daniela Faas, Emily Ferrier, and David Freeman. 2019. Integrative Tool Training Framework for Fabrication and Library Spaces. In *Proceedings of 4th International Symposium on Academic Makerspaces*. New Haven, CT, USA.

[16] Yoshihiko Futamura. 1983. Partial Computation of Programs. 482 (March 1983), 255–295. https://repository.kulib.kyoto-u.ac.jp/dspace/handle/2433/103401

[17] Saul Greenberg and Bill Buxton. 2008. Usability Evaluation Considered Harmful (Some of the Time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1357054.1357074

[18] 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

[19] Megan Hofmann, Gabriella Hann, Scott E. Hudson, and Jennifer Mankoff. 2018. Greater Than the Sum of Its PARTs: Expressing and Reusing Design Intent in 3D Models. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, 301:1–301:12. https://doi.org/10.1145/3173574.3173875

[20] 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

[21] Scott E. Hudson. 2014. Printing Teddy Bears: A Technique for 3D Printing of Soft Interactive Objects. In *Proceedings of the SIGCHI Conference on Human*

Factors in Computing Systems (CHI '14). ACM, New York, NY, USA, 459–468. https://doi.org/10.1145/2556288.2557338

[22] Jennifer Jacobs and Leah Buechley. 2013. Codeable Objects: Computational Design and Digital Fabrication for Novice Programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)*. ACM, New York, NY, USA, 1589–1598. https://doi.org/10.1145/2470654.2466211

[23] Juha Kuusama. 2021. LitePlacer | The Pick and Place Machine for Your Lab. https://liteplacer.com/

[24] 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

[25] Jarrod Knibbe, Tovi Grossman, and George Fitzmaurice. 2015. Smart Makerspace: An Immersive Instructional Space for Physical Tasks. In *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces (ITS '15)*. ACM, New York, NY, USA, 83–92. https://doi.org/10.1145/2817721.2817741

[26] 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.

[27] Danny Leen, Raf Ramakers, and Kris Luyten. 2017. StrutModeling: A Low-Fidelity Construction Kit to Iteratively Model, Test, and Adapt 3D Objects. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 471–479. https://doi.org/10.1145/3126594.3126643

[28] Jiahao Li, Jeeeun Kim, and Xiang 'Anthony' Chen. 2019. Robiot: A Design Tool for Actuating Everyday Objects with Automatically Generated 3D Printable Mechanisms. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New Orleans, LA, USA, 673–685. https://doi.org/10.1145/3332165.3347894

[29] Wallace Lira, Chi-Wing Fu, and Hao Zhang. 2018. Fabricable Eulerian Wires for 3D Shape Abstraction. In *SIGGRAPH Asia 2018 Technical Papers (SIGGRAPH Asia '18)*. ACM, New York, NY, USA, 240:1–240:13. https://doi.org/10.1145/3272127.3275049

[30] Makeblock. 2019. mBlock. http://learn.makeblock.com/en/software/

[31] MakePrintable. 2021. MakePrintable: 3D Printing API. https://makeprintable.com/

[32] Albert Manero, Peter Smith, Amanda Koontz, Matt Dombrowski, John Sparkman, Dominique Courbin, and Albert Chi. 2020. Leveraging 3D Printing Capacity in Times of Crisis: Recommendations for COVID-19 Distributed Manufacturing for Medical Equipment Rapid Response. *Int J Environ Res Public Health* 17, 13 (July 2020). https://doi.org/10.3390/ijerph17134634

[33] M. T. Mason. 1981. Compliance and Force Control for Computer Controlled Manipulators. *IEEE Transactions on Systems, Man, and Cybernetics* 11, 6 (June 1981), 418–432. https://doi.org/10.1109/TSMC.1981.4308708 Conference Name: IEEE Transactions on Systems, Man, and Cybernetics.

[34] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 127:1–127:28. https://doi.org/10.1145/3276497

[35] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (Jan. 2019), 438–448. https://doi.org/10.1109/TVCG.2018.2865240 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[36] Stefanie Mueller, Pedro Lopes, and Patrick Baudisch. 2012. Interactive Construction: Interactive Fabrication of Functional Mechanical Devices. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 599–606. https://doi.org/10.1145/2380116.2380191 event-place: Cambridge, Massachusetts, USA.

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

[38] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing structured CAD models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

[39] Jasper O'Leary and Nadya Peek. 2018. Material Flow in Makerspaces. In *Proceedings of the 3rd International Symposium on Academic Makerspaces*. ACM, Stanford, CA, USA.

[40] Dan R. Olsen. 2007. Evaluating user interface systems research. In *Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST '07)*. Association for Computing Machinery, New York, NY, USA, 251–258. https://doi.org/10.1145/1294211.1294256

[41] Huaishu Peng, Rundong Wu, Steve Marschner, and François Guimbretière. 2016. On-The-Fly Print: Incremental Printing While Modelling. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 887–896. https://doi.org/10.1145/2858036.2858106

[42] Alexander Y. Piggott, Jan Petykiewicz, Logan Su, and Jelena Vučković. 2017. Fabrication-constrained nanophotonic inverse design. *Scientific Reports* 7, 1 (May 2017), 1786. https://doi.org/10.1038/s41598-017-01939-2 Number: 1 Publisher: Nature Publishing Group.

[43] Josef Průša. 2012. Prusa3D Printer. https://www.prusa3d.com/

[44] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

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

[46] Jessica Rosenkrantz and Jesse Louis-Rosenberg. 2018. Coral Cup. https://n-e-r-v-o-u-s.com/blog/?p=8222

[47] ROS.org. 2019. Unified Robot Description Format - ROS Wiki. http://wiki.ros.org/urdf

[48] Hidekazu Saegusa, Thomas Tran, and Daniela K. Rosner. 2016. Mimetic Machines: Collaborative Interventions in Digital Fabrication with Arc. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, New York, NY, USA, 6008–6013. https://doi.org/10.1145/2858036.2858475

[49] Sarf2k4. 2021. Configuring Marlin. https://marlinfw.org/docs/configuration/configuration.html

[50] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[51] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. 2014. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology (UIST '14)*. Association for Computing Machinery, New York, NY, USA, 669–678. https://doi.org/10.1145/2642918.2647360

[52] Greg Saul, Tiago Rorke, Huaishu Peng, and Cheng Xu. 2013. Make Your Own Piccolo. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction (TEI '13)*. ACM, New York, NY, USA, 439–442. https://doi.org/10.1145/2460625.2460723

[53] Eldon Schoop, Michelle Nguyen, Daniel Lim, Valkyrie Savage, Sean Follmer, and Björn Hartmann. 2016. Drill Sergeant: Supporting Physical Construction Projects Through an Ecosystem of Augmented Tools. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '16)*. ACM, New York, NY, USA, 1607–1614. https://doi.org/10.1145/2851581.2892429

[54] Erik Seligman, E. Thomas Schubert, and M. V. Achutha Kiran Kumar. 2015. *Formal verification: an essential toolkit for modern VLSI design*. Elsevier/MK, Morgan Kaufmann is an imprint of Elsevier, Amsterdam ; Boston. OCLC: ocn920376471.

[55] Alexander Slocum. [n.d.]. FUNdaMENTALS of Design. http://pergatory.mit.edu/resources/fundamentals.html

[56] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907 event-place: San Jose, California, USA.

[57] Alexander Teibrich, Stefanie Mueller, François Guimbretière, Robert Kovacs, Stefan Neubert, and Patrick Baudisch. 2015. Patching Physical Objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, New York, NY, USA, 83–91. https://doi.org/10.1145/2807442.2807467

[58] 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)*. ACM, New York, NY, USA, 713–725. https://doi.org/10.1145/3332165.3347918 event-place: New Orleans, LA, USA.

[59] Cesar Torres, Jasper O'Leary, Molly Nicholas, and Eric Paulos. 2017. Illumination Aesthetics: Light As a Creative Material Within Computational Design. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 6111–6122. https://doi.org/10.1145/3025453.3025466

[60] Cesar Torres, Sarah Sterman, Molly Nicholas, Richard Lin, Eric Pai, and Eric Paulos. 2018. Guardians of Practice: A Contextual Inquiry of Failure-Mitigation Strategies within Creative Practices. In *Proceedings of the 2018 Designing Interactive Systems Conference (DIS '18)*. Association for Computing Machinery, New York, NY, USA, 1259–1267. https://doi.org/10.1145/3196709.3196795

[61] Jasper Tran O'Leary and Nadya Peek. 2019. Machine-o-Matic: A Programming Environment for Prototyping Digital Fabrication Workflows. In *The Adjunct Publication of the 32Nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, New York, NY, USA, 134–136. https://doi.org/10.1145/3332167.3356897 event-place: New Orleans, LA, USA.

[62] Ultimaker. 2021. Ultimaker/Cura Printer Definitions. https://github.com/Ultimaker/Cura

[63] Priyan Vaithilingam and Philip J. Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32Nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, New York, NY, USA, 563–576. https://doi.org/10.1145/3332165.3347944 event-place: New Orleans, LA, USA.

[64] 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.

[65] Vention. 2019. Vention. vention.io

[66] 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

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

[68] 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

[69] Max Willsey, Ashley P. Stephenson, Chris Takahashi, Pranav Vaid, Bichlien H. Nguyen, Michal Piszczek, Christine Betts, Sharon Newman, Sarang Joshi, Karin Strauss, and Luis Ceze. 2019. Puddle: A Dynamic, Error-Correcting, Full-Stack Microfluidics Platform. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, Providence, RI, USA, 183–197. https://doi.org/10.1145/3297858.3304027

[70] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. 2016. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (Jan. 2016), 649–658. https://doi.org/10.1109/TVCG.2015.2467191 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[71] 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

[72] Amy X. Zhang, Grant Hugh, and Michael S. Bernstein. 2020. PolicyKit: Building Governance in Online Communities. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/3379337.3415858