FEDT: Supporting experiment design and execution in HCI fabrication research

Valkyrie Savage
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
vasa@di.ku.dk

Harrison Goldstein
Computer and Information Science
University of Pennsylvania
Philadelphia, Pennsylvania, USA
hgo@seas.upenn.edu

Nóra Püsök
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
pusoknora@yahoo.com

Jia Yi Ren
University of Calgary
Calgary, Alberta, Canada
jiayi.ren@ucalgary.ca

Bhaskar Dutt
Department of Computer Science
University of Copenhagen
Copenhagen, Denmark
bd@di.ku.dk

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

Lora Oehlberg
University of Calgary
Calgary, Alberta, Canada
lora.oehlberg@ucalgary.ca

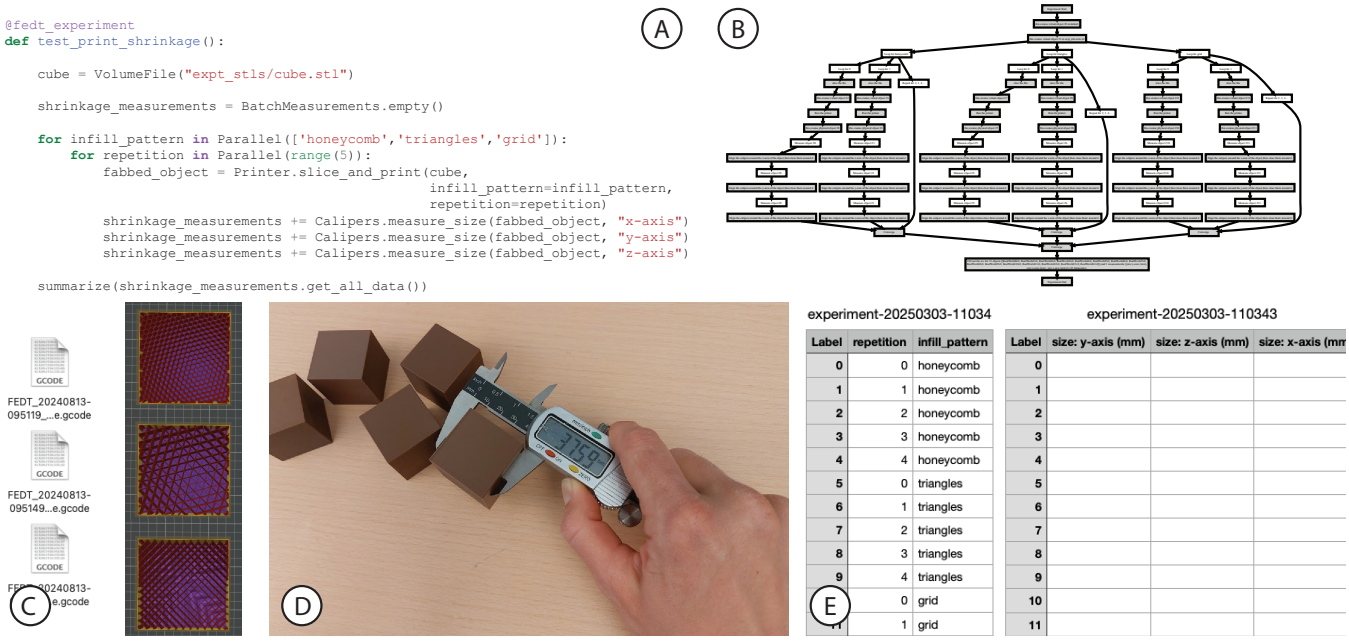


Figure 1: The Fabrication Experiment Design Tool (FEDT) helps define and run experiments that involve modifying parameters of CAD, CAM, fabrication, and post-processing, as well as differing post-fabrication usage. Experiment definitions are created in code (A), which can be run to create flowcharts (B) that capture either their high-level design or their actual execution. FEDT can help generate machine files (e.g., gcode) (C) for fabrication (D), as well as measurement support files (e.g., CSVs) to help execute experiments (E). FEDT is available at https://osf.io/ewuxb/.

UIST '25, Busan, Republic of Korea

ACM ISBN 979-8-4007-2037-6/25/09
https://doi.org/10.1145/3746059.3747739

Abstract

Fabrication research in HCI relies on diverse experiments to inform and assess research contributions. However, performance and reporting of these experiments is inconsistent, not only reducing transparency that reassures reviewers and readers of a project's

rigour but also challenging methods' replicability by future researchers. We analyze recent fabrication publications to extract a unified experimental workflow, which we develop into a domain-specific language, and into the openly-available Fabrication Experiment Design Tool (FEDT). FEDT facilitates designing and executing HCI fabrication experiments. We demonstrate its comprehensiveness by using FEDT to model 42 fabrication experiments from 10 papers, which leverage varied fabrication technologies and techniques, including requiring human intervention in their steps. We discuss FEDT and our modeled experiments with the papers' original authors to evaluate its precision and utility in real workflows, and we demonstrate functionality with end-to-end replications of two published experiments.

CCS Concepts

• **Human-centered computing → HCI design and evaluation methods**; **Interactive systems and tools**; • **Software and its engineering → Domain specific languages**.

Keywords

digital fabrication, experiments, domain-specific language, interactive object fabrication workflow

ACM Reference Format:

Valkyrie Savage, Harrison Goldstein, Nóra Püsök, Jia Yi Ren, Bhaskar Dutt, Chandrakana Nandi, and Lora Oehlberg. 2025. FEDT: Supporting experiment design and execution in HCI fabrication research. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28–October 01, 2025, Busan, Republic of Korea.* ACM, New York, NY, USA, 69 pages. https://doi.org/10.1145/3746059.3747739

1 Introduction

Since the first UIST session on "Fabrication and Hardware" in 2012, digital fabrication has become a booming area in Human-Computer Interaction (HCI) research, with seven paper sessions and two late-breaking work sessions devoted to it at CHI '24, and three paper sessions at UIST '24, alongside panels [19], journal articles [8], and workshops [86]. These publications cover a broad range of techniques and applications for digital fabrication, like low-cost 3D displays made on a laser cutter [25] and graph-based manipulations of knitting patterns [35]. A core part of this type of technical research is *experiments*, in which researchers create fabricated artifacts under different conditions and subject them to controlled tests to measure their behaviour. This stands in contrast to "experimentation," a less-structured method by which authors try out different designs or settings to find a combination that works at all: we focus only on experiments here. Experiments are key to exploring underlying concepts, generalizability, and replicability of technical contributions and form part of the ever-growing knowledge base in the field: e.g., will it work with my laser cutter? What if I rotate 90° before 3D printing? Could I reliably build my next research paper based on what this paper claims to achieve?

As we will see, experiments are incredibly diverse in their structure and purpose. However, there are universal assumptions and challenges in experimental design, some of which have been explored by prior work. For example, researchers in HCI outside of fabrication have built systems that structure hypotheses for user studies [73], scaffold pre-registrations [58], or infer appropriate analyses from datasets [40]. In fields like molecular biology that require rigorous process standardization, end-to-end design and management frameworks[1] support scientists in defining, replicating, and sharing processes. These techniques lead to better quality science and a greater sense of openness and transparency in methods. In software, toolmakers have created platforms[2] to capture data, metadata, analysis code, computational environments, and more all in one location: this enables straightforward re-execution and replication of virtual experiments. However, it is not obvious how to facilitate similar transparency for interactive object fabrication workflows that are increasingly common across HCI fabrication research: merely uploading design files to share repositories[3], generating user study designs [58, 73], and maintaining data and experimental analysis scripts[4] is not sufficient—transparency requires maintaining a choreographed series of executions by humans, machines, and software, mediated through physical and virtual objects. Our work lays foundations for making fabrication work replicable and more transparent by leveraging the common structure of interactive object fabrication workflows to clearly describe and automate best practices for researchers (Figure 1).

Based on an examination of prior fabrication research, we describe a common computational fabrication workflow (Figure 2) that is the starting point for all types of technical validation experiments in fabrication. Researchers digitally manipulate geometry, specify materials, adjust machine settings, then fabricate required objects from digital files, using digitally-controlled machinery. Then, once the object is completed, it can be post-processed, manipulated by users and subjected to tests. Data from tests are recorded, quantitatively analyzed, transformed into digital graphs, and reported in digital conference proceedings along with supplementary materials.

As HCI researchers, we are drawn to the interfaces between the human and these digital processes during this research practice. Humans participate at several points: to plug gaps in machinery or infrastructure (e.g., to assemble ruffles [76]), as a comment on mechanization (e.g., by being the machine [18]), or as a source of noise in the experiments themselves (e.g., due to different users' capacitive properties [71]). Supporting and cataloging the rich, diverse experiments of the fabrication community while harnessing their essential digital nature benefits replicability, and therefore the growth of this vibrant research area. At present, there is no tool or vocabulary that explores designing, executing, and sharing experiments in this area. One opportunity for doing so is to leverage a Domain-Specific Language (DSL).

It is in this spirit that we create FEDT: the Fabrication Experiment Design Tool. FEDT is based on a DSL embedded in Python which allows researchers to specify experiments they aim to perform. This is conceptually similar to Open Science pre-registration tools—as the researcher specifies exactly which variables will be controlled, what will be done, and what will be measured—but FEDT supports designing and actually executing the experiment process. Using our

[1] e.g., SnapGene https://snapgene.com or LabGuru https://labguru.com
[2] e.g., Whole Tale https://wholetale.org, Renku https://renkulab.io/, and ReproZip https://reprozip.org/
[3] e.g., Thingiverse https://thingiverse.com
[4] e.g., with the Open Science Foundation https://osf.io/

language, researchers can define experiment workflows that combine user instructions with commands for modeling or toolpathing software and machines into varied loop structures. The language maintains relationships between fabricated objects and references in the code, helping users keep track of the moving parts of their experiment. The FEDT library prototype focuses on workflows with 3D printers and laser cutters available in our labs, but it is designed to be extensible to support new modeling programs and machines.

We demonstrate and reflect on modeling 42 experiments from 10 papers using FEDT, and discuss the results of presenting these to researchers behind the papers. At a high level, researchers strongly appreciated the FEDT concept, with concrete ideas on how they would use it for clearer explanation in future papers, for creating "contracts" with students running experiments, and for speeding up their own workflows. We also describe the types of mistakes that we made in modeling, based on these author conversations, and reflect on what this means for FEDT. We further show end-to-end functionality by manufacturing objects supporting two of the experiments using FEDT in our lab. Finally, we discuss a broad opportunity for future work in DSLs for fabrication tools and Open Science, and the social processes that need to accompany the technology.

Our contributions are thus:

- FEDT: a modular system leveraging a domain-specific language to support researchers in designing and implementing fabrication experiments
- An evaluation of FEDT, including modeling and executing published experiments through the tool, and reflections from expert fabrication researchers

FEDT is openly available at https://osf.io/ewuxb/. We are eager to see its application in future research papers, and for the UIST community to expand upon it.

2 Interactive Object Fabrication Workflow

Despite a breadth of approaches and techniques for both manual and digital fabrication, there is a similar fabrication workflow that people and machines follow to go from concept to object. This workflow becomes particularly salient in the context of experiments, as it becomes clear that each step is an opportunity for innovating new systems, or performing a technical validation.

To understand the variety of ways that fabrication researchers work, we informally surveyed a wide variety of fabrication papers from CHI, UIST, and TEI, which we used to adapt an existing fabrication workflow model [52] to interactive object fabrication(Figure 2, top left). This workflow encapsulates what we observed in the sample and our collective experience as fabrication and engineering design researchers, and helps us understand the space of possible sites for fabrication operations, interventions, and experimental measurements that we can support using a tool.

To ensure the validity of the workflow, we then applied it to a purposive sample of 19 fabrication research publications— intentionally selecting representative papers with unique workflows and diverse fabrication techniques (3D printing [9, 20, 31, 50, 69, 71, 83, 87], CNC milling [46], plotter cutting [44], inkjet printing [43], laser cutting [9, 25, 39, 76, 85], knitting machines [35], moulding [31, 85]); we also selected three papers where the contribution itself was a new fabrication platform [38, 55, 77] (see Figure 2). This sample is not intended to be representative of fabrication research in HCI, but instead curated to challenge assumptions and test at the fringes of emerging research.

2.1 Workflow Phases

We define each workflow phase by the user-provided input as well as the machine-provided artifacts. "User-provided input" may be manually specified, or could come from a selected sensor or other external process or system which reflects the user's needs.

- *Design.* The user specifies the geometry of the object(s) using Computer-Aided Design (CAD) software, such that the system can generate and export digital geometries (e.g., .stl, .svg, knitout).
- *Configuration.* The user provides material specifications to Computer-Aided Manufacturing (CAM) software; the system provides configuration settings (e.g., .config, .dat).
- *Toolpathing.* The user provides machine-specific parameters to CAM software; the system slices a model and/or generates toolpaths (e.g., .gcode, .000).
- *Fabrication.* The user physically configures the fabrication hardware so it is ready to create a part. The system runs configuration and toolpathing files on the machine, resulting in one or several parts of an artifact.
- *Post-Processing*. The user removes the fabricated object from the machine and performs any necessary post-processing. This may include cleaning up parts from the machine, assembling multiple fabricated or off-the-shelf parts, or performing finishing or surface treatments. This results in the final artifact or object.
- *User Interaction.* Users interact with the completed object, and evaluate it according to performance criteria.

2.2 Research Intervention in the Fabrication Workflow

We found that research innovation can come at any stage in the pipeline, and that to support evaluation of these contributions, researchers also introduced experimental variables at every stage in the pipeline. That is, some experiments manipulate geometry (Knit-Picking Textures measures perceived similarity of varied-width knitted star patterns [35]), while others manipulate materials (Soft Inkjet Circuits explores several components for inks [43]), machines (Kerf-Cancelling Mechanisms fabricate their designs on both laser cutters and CNC machines [64]), CAD configurations (A-Line manipulates slicing angle [83]), post-process treatments (spaceR examines the effect of washing on their textile sensors [3]), or interactions (Truscillator perform several kinds of rocking on completed playground equipment generated by their tool [45]).

2.3 Identifying Workflow Variations via Descriptive Model

We offer this workflow as a *descriptive* model, not a prescriptive one. We present visualisations of 9 papers' described processes [9, 31, 35, 44, 46, 48, 50, 55, 77] (Figure 2)—we selected papers which used varying fabrication technologies and particularly unusual workflows, which could challenge the model and show unique

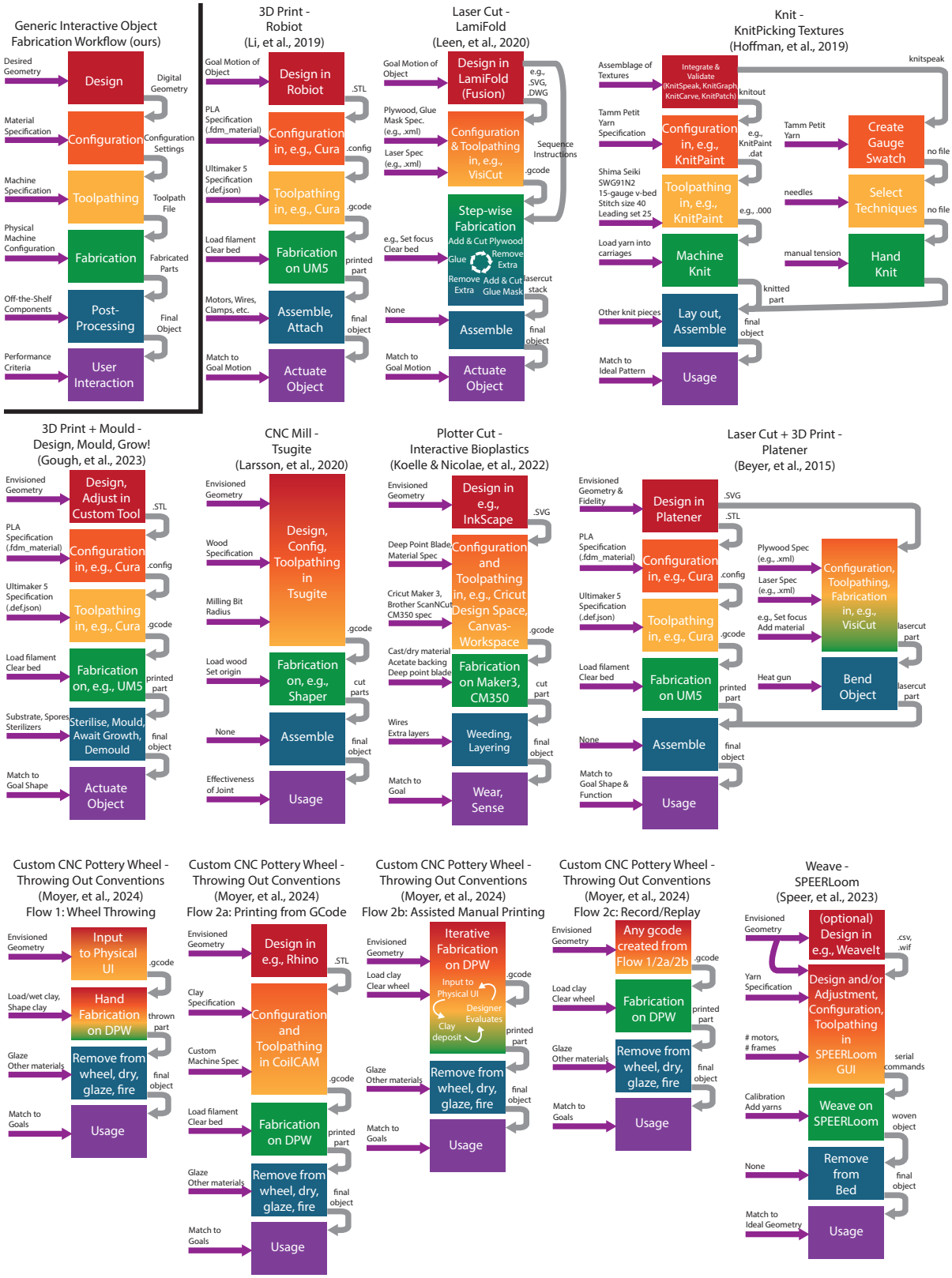


Figure 2: Our proposed workflow for fabrication of interactive objects (upper left) can be applied to diverse fabrication papers. This highlights how messy the workflow can become in reality. The presence of "e.g." means the paper did not specify the exact program or filetype.

paths through it. In doing so, we observed intriguing variability, especially where workflow steps are not clearly separated.

Transitions between workflow steps may be an "implicit" action. For example, when a hand knitter selects a knitspeak pattern to fabricate [35], they implicitly (or, rarely, explicitly) perform configuration (e.g., adjusting the chosen needles or the tension at which to knit) and toolpathing (i.e., choosing which techniques to use to achieve the pattern). There are usually no digital file handoffs between these steps, although some knitters do make notes.

As a result, *software tools may merge multiple workflow steps, without explicitly handing off intermediate files.* Particularly when proprietary software or drivers control fabrication machines, steps in the fabrication process are concealed from the end-user. For example, many laser cutters perform configuration, toolpathing, and initialization of fabrication within a single software driver. Other example include older open-source tools (e.g., Slic3r[5]) that do not follow best practices, or when researchers develop custom software tools tuned to a specific machine or workflow like Tsugite [46], where the geometric design is tied closely to the possible manufactured tolerances with a given CNC bit.

Workflow steps may be iterative. In many cases, user interaction takes place as a part of a workflow step, or as part of an iterative handoff between steps. For example, in LamiFold [48], users repeatedly call fabrication steps while following generated instructions to add new material to the laser cutter bed: a blend of "fabrication" and "post-processing" where humans plug a gap in what a theoretical machine could do. A more extreme case of this is in Throwing Out Conventions [55], where there is a "Choose Your Own Adventure"[6] flavour to how the workflow steps are accessed, allowing users to make or remake parts of an object by returning to earlier workflow steps.

Workflows can fork. In some cases, fabrication is carried out by multiple machines in parallel, as a design is broken up and assembled later. This happens, for example, in Platener [9], where a 3D model is split into some parts for laser cutting and some parts for 3D printing in order to speed up overall fabrication.

Workflow steps may be replaced by simulation. In some projects, part of the research contribution is to create a system that predicts what will happen when, e.g., a slicer is run with particular settings [83], or that compares many more objects than could practically be fabricated [20]. In these cases, some physical workflow steps are replaced with virtual workflow simulations.

Workflow steps are not all described in publication texts. To keep focus on the parts of the workflow that are critical to their contribution, we found many authors neglected to add details of specific tools used or workflows followed. We posit that this is related to a high base-level knowledge in the community of how these workflow steps are executed "in general," which allows readers to follow along without all details. However, this may mean that, as programs and technologies evolve, actually precisely replicating artifacts becomes impossible, particularly when they unwittingly rely on defaults or algorithms hidden in specific programs.

These flows and their variations, which we do not believe are exhaustive, strongly suggest that the generic workflow is present in and adequate to describe interactive fabrication in published works, but that there is significant flexibility in how it is performed.

[5]https://slic3r.org/about/

[6]https://en.wikipedia.org/wiki/Choose_Your_Own_Adventure

2.4 Design Goals for a Tool to Support the Workflow

Based on this exploration, we set out to create a tool for fabrication researchers to design, perform, and communicate their varied technical experiments. We designed the tool with the following key goals in mind:

- *Precision*: the tool should enable researchers and their colleagues to understand exactly what should be or was performed in experiments at design time, at execution time, and in reporting.
- *Comprehensiveness*: the tool should be able to capture any experiment that fits into the digital fabrication workflow—including those that mix human and machine actions, and that have partial or unclear step separation.
- *Extensibility*: the tool should not be locked to specific fabrication technologies; it should have a core set of operators and structures that make it sufficiently expressive while facilitating the addition of extra features required for novel machines and tasks.
- *Analysability*: the tool should enable analysis of experiment structures, including identifying potential failures early and leveraging common structure in related experiments.

Given these goals, we settled on an embedded domain-specific language (DSL) as an appropriate base. A programmatic view of fabrication experiments facilitates *composability*—programs in any DSL let users string together many statements in various ways that best match their mental models or particular setups. *Comprehensiveness* is also supported by using a language: the sample workflows blend and iterate on stages of the generic workflow, and a language (more so than a rigid tool) supports researchers in expressing the workflow needed for their software, machine, and task. DSLs also offer formal benefits: programs are easy to *analyze* statically by traversing their abstract syntax tree or their control flow.

Embedding a DSL within a host language also allows researchers to use existing toolchains (e.g., code editors, source and library managers) to interact with or design in it and to *extend* its capabilities, as well as to track provenance (e.g., through github forking and other metadata). For our host language, we chose Python. Previous (small) studies have found Python is an especially readable language, even for non-programmers [2], which supports our goal of *precision*. A text-based DSL offers the further convenience of being easily-included in textual reporting documents (e.g., publications).

3 FEDT

FEDT is a language for orchestrating fabrication experiments that helps researchers capture subtle details of their experimental designs, ensuring that the execution of the experiment agrees with the way that it is communicated to other researchers. It is implemented as an embedded DSL (eDSL) in Python. Below, we give an overview of the different parts of FEDT with an extended example, explore the core features of FEDT, and describe key implementation details.

3.1 Extended Example

The FEDT language lets researchers program experiments the same way they would program any other step-by-step process. The intention is that a researcher can write the experimental method, then evaluate whether it is adequate for the test they want to perform. After evaluation, the researcher (or a colleague) can execute the method using the same code. The program can later be published as part of a paper, documenting what has been done in the experiment to facilitate replication by future researchers. To demonstrate the features of FEDT, we show a simple experiment to measure the shrinkage of 3D printed cubes when using different infill patterns.

The simplest step of a FEDT program is an `instruction`, which communicates critical steps of the experimental execution. For example, the following line instructs the researcher to ensure that a design file for the experiment is in the correct location:

```
instruction("Create an STL file of a cube at `cube.stl`.")
```

Instructions can be listed one after another, in sequence:

```
instruction("Create an STL file of a cube at `cube.stl`.")
instruction("Print the object...")
```

And they can be interleaved with actual computation that is necessary for the experiment:

```
instruction("Create an STL file of a cube at `cube.stl`.")
cube = GeometryFile("cube.stl")
instruction("Print the object...")
```

The new line above actually looks on the researcher's computer for the `cube.stl` file and prepares it for use in a later stage of the experiment. In essence, instructions in FEDT programs can give direction to humans (either the researcher performing the experiment, or a user taking part in it), or to machines and systems (in this case, the filesystem, or in other cases a laser cutter, knitting machine, slicer, etc.).

Programs can also contain loops. For example, we can add nested for loops that interface with a 3D printer and print a collection of cube objects we want to measure:

```
instruction("Create an STL file of a cube at `cube.stl`.")
cube = GeometryFile("cube.stl")
instruction("Print the objects...")
objects = []
for infill_pattern in Parallel(['concentric','line','rectilinear']):
 for print_rep in Parallel(range(5)):
   fabbed_object = Printer.fab(cube,
                     infill_pattern=infill_pattern,
                     repetition=print_rep)
   objects.append(fabbed_object)
```

These for loops work just as they would in standard Python, but FEDT offers `Parallel` wrappers so the researcher can specify that these steps can be done in any order. (If order matters, the `Series` wrapper can be used instead.) Each loop iteration can also issue instructions to the researcher; for example, the `Printer.fab` function might instruct them to ensure the printer is set up in a particular way.

Next we highlight the `objects` list: the result of running a 3D printer is not bits in a computer, it is a "thing" in the physical world that you can see and touch! Indeed, this is part of what makes fabrication experiments special—they regularly involve turning data into physical objects and vice versa. The FEDT language bridges the gap between virtual and physical by maintaining virtual objects mirroring each physical one that is fabricated. In this case, `fab` returns a `RealWorldObject` each time it is called; this virtual object can be passed around and used in later experiment stages:

```
instruction("Create an STL file of a cube at `cube.stl`.")
cube = GeometryFile("cube.stl")
instruction("Print the objects...")
objects = []
for pat in Parallel(['concentric','line','rectilinear']):
  for print_rep in Parallel(range(5)):
    fabbed_object = Printer.fab(cube,
                      infill_pattern=pat,
                      repetition=print_rep)
    objects.append(fabbed_object)
shrink_measures = BatchMeasurements.empty()
for obj in Parallel(objects):
  for meas_rep in Series(range(5)):
    shrink_measures += Calipers.measure_size(obj, "x-axis")
    shrink_measures += Calipers.measure_size(obj, "y-axis")
    shrink_measures += Calipers.measure_size(obj, "z-axis")
analyze(shrink_measures.get_all_data())
```

We will discuss how `BatchMeasurements` works in the next section, but at a high level these new lines express that the researcher should measure each axis of each printed cube (referenced by an `obj`) several times. After all measurements are logged (`shrink_measures` represents shrinkage measurements), an externally-defined Python script can analyze the final results.

With their full experiment drafted, the researcher can run it. To begin with, they can create a flowchart by running in `Design` mode (Figure 1b). This creates an incomplete but instructive visualization of the procedure they will follow, and calculates how many objects they will create and how many measurements they will perform (or a floor for these, in the case of a **`while`** loop that will repeat an unknown number of times during execution). When they are satisfied with the structure, they can switch to `Execute` mode, which actually creates and manipulates files (in this case, gcode files for printing and a CSV file for data collection), creates prompts on the command line to instruct them to set up prints or measure particular objects, and makes calls to machines (e.g., the 3D printer) (Figure 1D). A complete flowchart is constructed during execute mode, with additional details capturing precisely what was done. `analyze` is an abstraction researchers can implement to further study data gathered, including using other statistical analysis tools [40, 41].

3.2 The FEDT Language

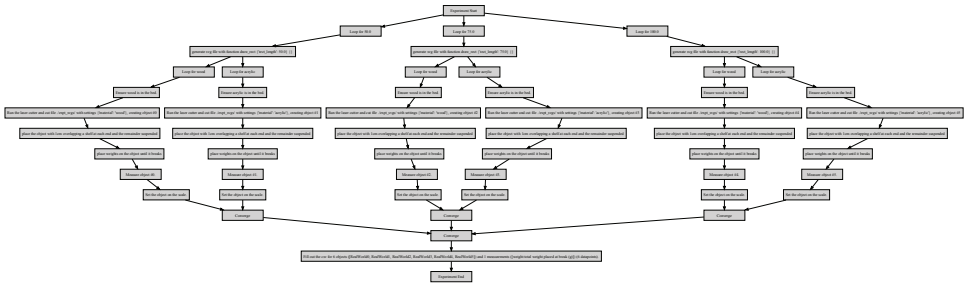
FEDT's several components each support a different aspect of the user experience.

3.2.1 Runtime. FEDT is implemented as a Python decorator that wraps and rewrites experiment code written in Python. The decorator instruments the code with calls into the FEDT *runtime* (the background process of the language) that track the experiment's steps as they are executed. Instructions, loops, and other language features are all tracked as elements crucial to later replication.

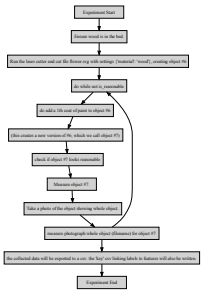
After execution, the runtime outputs a *flowchart* that communicates the instructions and control flow comprising the experiment. This can be visualized directly (e.g., put in a paper's appendix) or otherwise used to aid replicability.

The FEDT language combines experimental *design* and experimental *execution* into a single language. While we have discussed many ways in which this is advantageous, it can make the design

```python
@fedt_experiment
def test_force_at_break():
    breakage_points = BatchMeasurements.empty()

    for rect_length in Parallel(arange(50,100+include_last,25)):
        svg = SvgEditor.build_geometry(draw_rect, CAD_vars={'rect_length':rect_length})
        for material in Parallel(['wood','acrylic']):
            fabbed_object = Laser.fab(svg, material=material)
            instruction("place the object with 1cm overlapping a shelf at each end and the remainder suspended")
            instruction("place weights on the object until it breaks")
            breakage_points += Scale.measure_weight(fabbed_object,"total weight placed at break")

    analyze(breakage_points.get_all_data())
```



```python
@fedt_experiment
def test_paint_layers():
    flower = Laser.fab(LineFile('flower.svg'), material='wood')

    photos = ImmediateMeasurements.empty()
    is_reasonable = False
    coats_of_paint = 0
    while not is_reasonable:
        flower = Human.is_reasonable(flower)
        photos += Camera.take_picture(flower)
        is_reasonable = flower.metadata["human reasonableness check"]
        if not is_reasonable:
            coats_of_paint = coats_of_paint + 1
            flower = Human.post_process(flower, f"add a {coats_of_paint}th coat of paint")

    analyze(photos.dump_to_csv())
```



```python
@fedt_experiment
def test_user_assembly_time():
    simple = VolumeFile("simple_assembly.stl")
    complex = VolumeFile("complex_assembly.stl")

    timings = ImmediateMeasurements.empty()

    treatments = shuffle(["simple_first","complex_first"])

    for (user, treatment) in Parallel(enumerate(treatments)):
        simple_assembly = Printer.slice_and_print(simple)
        complex_assembly = Printer.slice_and_print(complex)
        order = []
        if treatment == "simple_first":
            order = [simple_assembly, complex_assembly]
        else:
            order = [complex_assembly, simple_assembly]
        for assembly in Series(order):
            assembly = User.do(assembly, "solve the assembly", user)
            timings += Stopwatch.measure_time(assembly, "time to solve the assembly")

    analyze(timings.dump_to_csv())
```

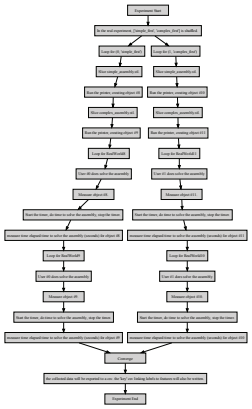


Figure 3: Each program defines a flowchart based on the structures within. `Parallel` structures (top) spread out and can be executed in any order. `while` structures (middle) create loops to be executed as long as a condition is true. `Series` structures (bottom) require that steps be executed in a particular linear sequence.

and testing process fairly tedious. FEDT addresses this with `Design`, a mode of operation where execution engines are disabled but the runtime (and other simple computations) continues to execute. For many experiments, this process can produce a faithful representation of the experiment flowchart *without actually running the experiment*, giving the user insights into how their experiment will execute and allowing them to catch potential problems (e.g., missing steps or incorrect setup) before running it.

In the parlance of the cognitive dimensions of notations [10], design mode increases progressive evaluation and reduces error-proneness. In a few cases, for example with **`while`** loops where the number of executions depends on a measurement, `Design` mode must approximate the experiment execution, but even that approximation has information that can help users avoid design flaws.

`Design` mode is implemented as a global flag in Python; executors are expected to obey this flag and produce no-op results in design mode, whereas the rest of the experiment code (including the runtime) can continue to operate.

3.2.2 Looping, branching, and randomness. FEDT supports several kinds of loops that we observed in our dataset. `Parallel` loops represent operations which can be executed in any order; for example, in the shrinkage study, it does not especially matter whether the cube with `rectilinear` infill is printed before or after the one with `concentric` infill. `Series` loops, on the other hand, describe operations which must be executed in a specific order but are perhaps also based on variables; for example, in a user study (Figure 3, bottom), users may be required to first assemble one object and then another. In cases where the researcher chooses to counterbalance or randomize ordering of these types of operations, FEDT also offers a `shuffle` operator. The `shuffle` operator knows about `Design` mode, so it is able to produce a deterministic order while debugging. Finally, FEDT supports **`while`** loops, which execute until some condition is fulfilled (Figure 3, middle). This type of structure allows for arbitrary numbers of repetitions as researchers explore optimization of parameters in designs.

In `Execute` mode, a researcher interacts with these loops on the commandline one step at a time, completing a full branch each time. While in theory they could jump around and, e.g., run parts of `Parallel` loops in parallel, we have left this to future work.

To render flowcharts visually, we use `graphviz`. We reduce the horizontal and vertical size of large experiment flows by collapsing and eliding long or wide repetitive loops, and further collapsing arithmetic and geometric sequences detected in loop variables.

3.2.3 Object Tracking. Another core goal of FEDT is to reduce the potential of researcher error when executing (and replicating) experiments. One potential source of error is losing track of the relationship between objects in the experiment code, which dictate the way analysis is carried out, and objects in the real world or on the file system. For example, in the previous section we describe an experiment in which the researcher has to print many visually identical (yet different!) cubes; if they get those cubes confused, they may confuse their measurements and get incorrect results.

To address this issue, we introduce Python data for tracking physical items (using the `RealWorldObject` class) and virtual "renderings" of information like output geometry from a modeling tool (using the `VirtualWorldObject` class). Each of these objects has a unique identification number and stores information about how the object has been manipulated over time.

To connect the software concept of a `RealWorldObject` to the actual real-world object it refers to, FEDT's library introduces optional pre-fabrication geometry manipulation functions for *automatic labeling*. FEDT executors can automatically add a `RealWorldObject`'s unique id number to the object itself (e.g., etched into the object with a laser cutter) or as a mark somewhere nearby the object (e.g., printed onto a 3D printer bed next to the object's model). When the researcher post-processes or measures an object, they can double check the id number to ensure that they are interacting with the correct object. During a run, FEDT outputs a CSV that researchers can use to connect a `RealWorldObject`'s provenance to a real world object's id (Figure 1E).

3.2.4 Measurements. FEDT also provides researchers with tools that streamline measurement collection in batches or singly. The `ImmediateMeasurements` class has researchers input one measurement at a time on the commandline, and when they are done it collates the measurements and their associated objects into a `dict` object for further analysis. The `BatchMeasurements` class (used in the shrinkage example) allows more flexibility. Instead of performing sequential, single measurements as objects are created, this enables researchers to create many objects and then input measurements en masse to a generated CSV file. The CSV file contains rows for each `RealWorldObject` (or `VirtualWorldObject`) and columns for each measurement. The measurement file is "blinded" and refers to each object by its unique id, so a second CSV which connects ids to provenance is also generated (1E). When the researcher is done filling out the CSV file, it is read in and again turned into a `dict` that can be analyzed. Multiple, repeated measurements can be recorded for a single object and feature for later averaging.

3.3 The FEDT Library

While FEDT and its runtime offers sufficient flexibility for expressing experiments, the FEDT language does not actually implement the machinery to operate fabrication machines, interact with CAD programs, and carry out other more automatic steps of the experiment workflows. These tasks are handled by external code and called via the FEDT library.

3.3.1 Executors. Automation for interfacing with workflow software and machines is implemented as *executors*; these are functions in the host language—Python—that call external functionality. Currently, FEDT provides a small library of executors that can either be used directly or adapted to work with a user's particular lab setup. The design of executors in FEDT means that any Python-accessible API already provided by researchers' machines, software, and tools can be re-used as part of FEDT experiments. Since the executors are designed to be separate from the language, it is possible to run an experiment with *no* executors, and just a sequence of manual `instruction` calls for a human to perform manual actions. We detail some executors implemented for our own lab in Section 4.1.2.

3.3.2 Encoding Workflow Steps in FEDT Executors. Our library provides a collection of structural abstractions that researchers can use to ensure workflow steps are followed, e.g., to explicitly hand a `GeometryFile` to a `ToolpathSoftware` and get a `CAMFile`, or a

`CAMFile` to a `FabricationMachine` and get a `RealWorldObject`. These structural abstractions are implemented as `class`es, and Python's multi-inheritance capabilities mean that custom researcher code can serve as `DesignSoftware`, `ConfigurationSoftware`, and `ToolpathSoftware` all at once, as in SpeerLoom's custom GUI [77] (Appendix A.10.3). This can also help researchers clarify what function(s) are fulfilled by their custom code or hardware.

3.3.3 Measuring. Measurements are handled by classes instantiated to describe a measurement tool. These classes describe the `Measurement` that the tool can provide, including its name, units, procedure, and the feature on the object that it describes. The class must also implement a `measure` function that returns a structured `Measurement`. Some tools can provide more than one type of measurement (for example, a multimeter can provide resistance, capacitance, and other values).

3.3.4 The Host Language. Embedding FEDT in Python lets researchers use normal Python code as executors to aid in the experiment process, or integrate libraries to reduce the amount of code they must write themselves. For example, an experiment can use basic Python or `psychopy` [59] to generate a Latin square for counterbalancing (e.g., in ./trilaterate [71], Appendix A.1). Integration with the host language can also be used to call out to external processes or languages for data analysis, to interface with a running experiment in a notebook environment, or to use existing tools to generate documentation for their experiments from comments.

4 Evaluating FEDT

To evaluate FEDT's capabilities, we modeled all 42 fabrication experiments from 10 purposively-sampled papers in the tool (full code, citations, and details appear in Appendix A). We then contacted the first and last authors of these papers using information listed on their websites, and discussed the FEDT tool and our replications of their experiments with authors from all 10 papers. We highlight features of these papers and lessons learned from the exercise of modeling; we further discuss three results of our conversations with authors: corrections on our modeled experiments, reflections on what experiments "are," and ideas for using and extending FEDT. The 10 papers we selected are from highly-respected HCI conferences (CHI, UIST, TEI), and cover a variety of fabrication techniques and technologies. Here, we refer to the authors of these 10 works "researchers" to distinguish them from the authors of *this* paper.

4.1 Modeling Existing Experiments

To demonstrate FEDT's *comprehensiveness* (ability to capture a variety of experiments that include interventions in various workflow stages), we modeled all experiments involving digital fabrication from the purposive papers. We exclude one experiment from Design, Mould, Grow! [31] that did not use digital fabrication machines and one from SPEERLoom [77] that did not fabricate any objects. By "model", we mean writing functions in FEDT that can be executed to attain results comparable to those reported in the papers: we only performed actual fabrication steps for two papers' experiments (Section 4.1.2).

The purposive papers include variables introduced in every workflow stage (Figure 2); they use 3D printers, custom weaving machines, laser cutters, knitting machines, vinyl cutters, and scissors; and they explore shape-change, logic gates, electrostatics, tagging, and bio-fabrication. Table 1 summarizes the papers; it also lists the key FEDT features used for each, which we extracted from the experiment programs after researcher-driven correction.

Working from papers' original text, the first author (who has >10 years of experience in publishing HCI fabrication papers) attempted to model each experiment in FEDT. Despite our goal of a purposive sample with well-described experiments, we still found some of them difficult to implement given the descriptions in the paper, with challenges understanding exactly how many objects were fabricated, if and when objects were reused between experiments, what machine was used for manufacture, as well as other missing details. We made our best guess of the experimental procedure using our own knowledge, and noted it as a question for our later conversations with the original researchers.

We were able to model all experiments in our language, and they are presented in full in Appendix A. In total, 42 experiments were modeled (2–8 per paper, 4.2 on average, column 5 in Table 1).

4.1.1 Reflections on Modeling Existing Experiments. For certain experiments in our dataset, we were uncertain how to express them according to the workflow (Figure 2). For example, in AirLogic [69], one experiment involves orienting objects in different ways relative to the print bed, then printing them and forcing air through to determine how printed layer lines at differing orientations affect airflow. Even after discussing with the original researchers, there was no consensus on whether this was a geometry/CAD variable (since the geometry was rotated) or a manufacture/CAM variable (since the relative angle of the slicing was changed). Since FEDT is based on a language, and languages can flexibly capture concepts in varying ways, the upshot is that *it does not matter* whether these align with the workflow in a particular way: FEDT is able to capture *either* method of modeling, and still precisely describe what the researcher did.

An interesting point of flexibility we had not not previously considered is that FEDT allows specifying measurements of both physical and *virtual* objects. In some experiments that we modeled, the original researchers compared outputs of simulations to outputs of physical manufacturing processes by looking at their final geometries. FEDT is currently built such that geometric measurements of virtual objects can be specified and prompted in the same way as those of physical objects (e.g., Appendix A.8). We also found that capturing non-standard fabrication flows (e.g., where 2D geometry is defined by a computer and manually cut with scissors, Appendix A.6) and those on novel fabrication machines (Appendix A.10.3) was possible. These highlight FEDT's extensibility.

4.1.2 End-to-end replications of two experiments. To confirm the connection of the pipeline to our lab's infrastructure, we also used FEDT to create all the objects for end-to-end replications of one experiment from G-ID [20]—which uses 3D printers and variables generated through slicing—and one from CircWood [39]—which uses a laser cutter and variables introduced through materials, fabrication, and post-processing (Figure 6). In particular, we fabricated objects for a CircWood experiment that measured the changes in

Paper Title (Year)	Manufacturing Tech	Author Keywords	Sources of Variability	# Expts	instruction	Parallel	Series	while	shuffle	Batch	Immediate
./trilaterate [71] ('19)	FFF Printer	capacitive sensing	CAD, user interaction	2	✓	✓	✓		✓	✓	
AirLogic [69] ('22)	FFF Printer	fluidics, logic gates	CAD, CAM, actuation	3	✓	✓	✓			✓	
BlowFab [85] ('17)	Laser Cutter, Inflation	prototyping, creativity support	CAD	3	✓	✓	✓			✓	
CircWood [39] ('22)	Laser Cutter	laser beam machining	CAD, CAM, material, post-processing, time	6	✓	✓	✓			✓	✓
Design, Mould, Grow! [31] ('23)	FFF Printer, Moulding	bio-fabrication	CAD, material, post-processing, actuation	3	✓	✓	✓	✓		✓	
ElectriPop [25] ('22)	Laser Cutter, Vinyl Cutter, Scissors	electrostatic inflation	CAD, simulation, time	8	✓	✓	✓	✓		✓	✓
FabHydro [87] ('21)	SLA Printer	interaction, design	CAD, material, time	6	✓	✓	✓	✓		✓	✓
G-ID [20] ('20)	FFF Printer, 3D Rendering	identification, tags	CAD, CAM, machine, environment, actuation	6	✓	✓		✓		✓	✓
KnitPicking [35] ('19)	Machine/Hand Knitting	machine knitting, soft fabrication	CAD	2	✓	✓	✓		✓	✓	✓
SPEERLoom [77] ('23)	Weaving	weaving, mathematics engineering	CAD, machine	3	✓	✓	✓	✓		✓	✓

Table 1: Features of papers modeled for the evaluation. The second column represents the manufacturing technologies/techniques used in each paper. We use `Batch` for `BatchMeasurements`, `Immediate` for `ImmediateMeasurements`. `VirtualWorldObjects` and `RealWorldObjects` are workflow features used in all papers, so they are not in the table.

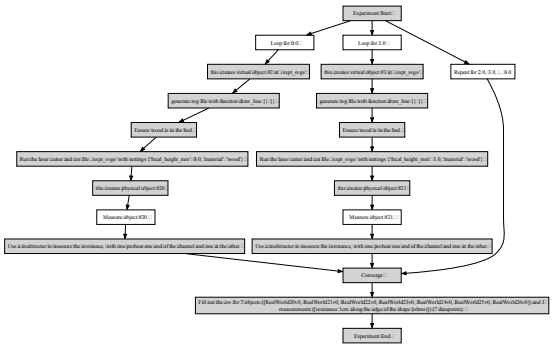


Figure 4: The flowchart of our replicated CircWood experiment, exploring 7 different focal lengths for laser cutting.

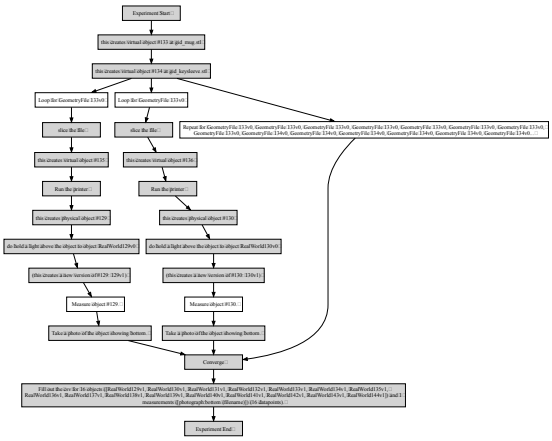


Figure 5: The flowchart of the G-ID experiment we replicated, which generates 14 sets of random slicer settings (10 for large objects and 4 for medium objects) within set ranges and 3D prints them.

resistance from scanning the laser over a piece of wood at different focal heights (Appendix A.4.2), and for a G-ID experiment that manufactured objects with a series of random parameters to see how accurately they can be distinguished (Appendix A.8.2). For the CircWood experiment, we also generated ids to be cut alongside the tests (Figure 6, left). It was necessary to modify the parameters used by the G-ID experiment as our slicer software (Prusa slicer) did not expose identical ones (we substituted "honeycomb" infill for the "trihexagon" described, and converted "infill width" measurements reported into percentage infills). This suggests a future opportunity for formalizing and sharing adaptations of experiments to various physical lab setups' executors, and perhaps even automated suggestions for such translations; such an extension could include, e.g., models of machine capabilities from Taxon [80], but would necessarily rely on some researcher expertise. This is beyond the scope of our work here, though it may come with adoption of FEDT.

On a practical level, to achieve these end-to-end replications we have also implemented executors that connect with software and machines available in our labs. These are described here briefly to give the reader an idea of what is possible, but full implementation is available at https://osf.io/ewuxb/:

Laser Cutter We use the Python drawsvg[7] library to programmatically generate SVG files. Our physical laser is an Epilog Helix 24, with a bed size of 24" x 18", which we drive using the command-line functionality of VisiCut[8]. For this, we programmatically generate

[7]https://pypi.org/project/drawsvg/
[8]https://visicut.org/

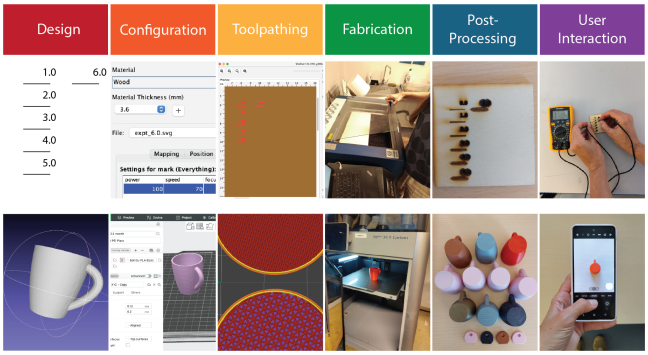


Figure 6: We replicated objects using experiment workflows from CircWood (top) and G-ID (bottom) using FEDT and the backend library functions implemented for our lab.

a settings file at the beginning of a laser experiment which contains all combinations of settings to be tested: the user must then manually import this file into VisiCut, as it cannot be done programmatically. During the run, we generate mappings files using VisiCut's XML templating language to access required settings. Our laser does not support remote execution, so the researcher manually presses the execute button when prompted.

3D Printer We use the Python FreeCAD[9] library to programmatically generate and manipulate STL files. We also implemented instructions that guide the user to set variables in parametric models in Autodesk Fusion, without making use of their Python bindings. We use various 3D printers: Creality Ender 3 Pro, Prusa MK4, Ultimaker 3, and Bambu Labs Carbon X1. We use PrusaSlicer 2.8.0 for all command-line slicing. Our lab's print management system does not offer an API, so we have implemented instructions that ask the user to load gcode files onto SD cards, set the printer according to the experiment setup, and begin prints manually.

Calipers and Multimeters We implemented `Calipers` and `Multimeter` classes that lead a user through manually capturing `Measurements` (e.g., dimensions, resistances, etc.) for immediate or batch processing.

Human Although automated humans are not available in our area, we created a `Human` class to carry out various workflow tasks through `instructions` to the researcher. This explicitly clarifies that a human is performing a task, but functionally such tasks could just as easily be carried out with `instructions`.

4.2 Walkthroughs with Fabrication Researchers

We emailed the first and last researcher listed on all papers whose experiments we modeled, and met with researchers from all 10 papers. We spoke with 14 researchers via Zoom: in 4 cases, both the first– and last listed spoke with us in the same interview. We asked each researcher a series of demographic questions, demonstrated the FEDT prototype for them, walked through our reproduction of their experiments, and had a semi-structured interview to understand how FEDT might fit in their work process.

Demographically, our 14 researchers had 3–27 (average 8.57) years of experience in fabrication research and -3–40 (average 6.93) years since completion of their Ph.D. (in three cases, the researchers had not yet completed their Ph.D.; we recorded these as negative years "since completion"). They had published 1–46 (average 10.07) papers each that they described as fabrication papers (this included papers in HCI, graphics, and additive manufacturing journals). Seven identified as female and seven as male. These researchers' current positions were at nine universities and two companies worldwide, with one saying they were retired and one describing themself as an independent researcher. Researchers were currently based in six countries.

During each conversation, we demonstrated the tool by briefly explaining our method for extracting the DSL, then showing them a collection of toy examples demonstrating its features (`instructions`; flowchart, `Design` mode, and `Execute` mode; `Series`, `Parallel`, `while`, and `shuffle` loops; CAD manipulation; material description; CAM manipulation; post-processing instructions; user study tasks; `BatchMeasurements` and `ImmediateMeasurements`) before walking them through the code we had written to model their experiments. We asked for clarification on any challenges we had in modeling, and we invited them to ask questions throughout. This took approximately 60 minutes per researcher or researcher pair.

4.2.1 Correction of our Modeled Experiments. In some cases, researchers informed us that the models we made were incorrect. Researcher-driven correction of our 42 replications is reproduced in the Appendix, but at a high level we reproduced 13 experiments correctly and made mistakes with the rest (Table 2). The first author went through the correction notes to classify them into categories, and these were refined through discussion with other authors. Note that some experiments were affected by more than one type or subtype of mistake, so counts do not sum to 42. There were four kinds of mistakes that we made. In the first type, papers lacked adequate detail, and we made incorrect assumptions in our reproductions regarding the number of objects made, how randomness was applied, which machines were used, etc.: this happened in 24 experiments. In the second type of mistake, which affected 4 experiments, we did not reproduce enough detail in the experiments to satisfy the original researchers, so more was added.

In 7 cases, we modeled what was written in the paper, but it turned out that the paper's description of an experiment was inconsistent with what was actually performed. This was especially interesting, as it suggests that the precision required to actually perform an experiment is not always identical to what must be communicated about it. This same concept was reflected in some researchers' comments on the gap between reviewing and actually replicating fabrication papers, for example, one researcher described that, in reality, papers do not include all the "know-how" required to replicate them—which became clear when they tried to build on another person's project, but that when "I read [a paper], I *feel* it would be easy to replicate it."

The final type of mistake, affecting 6 experiments, was that we modeled "experiments" which the researchers felt were intended to be "demonstrations."

It was possible to rectify all mistakes within FEDT. In addition, we noted that researchers were frequently able to spot issues with our implementations of their experiments during our screenshare,

[9] https://www.freecad.org/

Error Type	# Occurrences - *expts*
Wrong # unique objects	10 - *1.1, 1.2, 3.1, 4.2, 4.4, 6.5, 6.8, 7.4, 8.1, 10.1*
Wrong # repeated objects	1 - *3.3*
Wrong # repeated steps	5 - *5.2, 6.2, 6.3, 7.2, 8.1*
Wrong use of randomness	1 - *1.2*
Wrong machine or tool used	4 - *3.1, 3.2, 3.3, 9.1*
Wrong measurement method	3 - *6.4, 6.7, 10.2*
Wrong method, same outcome	2 - *4.5, 5.2, 10.1*
Wrong loop type	5 - *5.1, 5.2, 5.3, 8.3, 10.1*
Total incorrect assumptions	24
Insufficient detail	4 - *5.1, 5.2, 5.3, 8.1*
Simplification in paper	7 - *3.1, 3.2, 3.3, 7.2, 7.7, 9.1, 9.2*
Not an experiment	6 - *6.5, 6.6, 6.7, 7.1, 7.6, 8.5*
No corrections	13 - *2.1, 2.2, 2.3, 4.1, 4.3, 4.6, 6.1, 7.3, 7.5, 8.2, 8.4, 8.6, 10.3*

Table 2: Types of corrections made, and which experiments they applied to. Experiments are referred to by their Appendix section, which is written as PAPER.EXPERIMENT, i.e., 8.3 is the third experiment reported in paper #8.

even before we verbally described the experiments for them. We take this as evidence toward FEDT's *precision*.

Our Reflections on Mistakes and Communication. These mistakes provide an intriguing glimpse of the language's power. Once the experiments were translated from communicative prose to executable formalisms, it was clear both to us the authors and to the researchers what was being done correctly or incorrectly. Errors caused by our incorrect assumptions or lack of sufficient detail in modeling, and authors' disagreement with us about what constitutes an experiment versus a demonstration would have been avoided by their initially writing the FEDT versions of the protocols. The "simplification in paper" errors, on the other hand, require more consideration, as they step towards the opportunities and pitfalls of simulation. Future work might explore analysis tools that can simplify or confirm equivalence of programs, but this requires some definition of what equivalence *is*, which becomes quite messy in the real world. FEDT enables these conversations to happen on a technical level: where they go will be a social process to be navigated at the community level.

One especially interesting mistake came from experiment 10.1 in the SPEERLoom paper: the authors, in attempting to calibrate and evaluate their custom-built machine, treat the machine both as a site of fabrication *and* as a fabricated object being tested [77]. This required exercising the language in a different way to other experiments, but was possible to do.

4.2.2 Researchers' Reflections on What an Experiment Is. In some cases, we struggled with defining whether a particular portion of a paper was an "experiment" or a "demonstration," but erred on the side of modeling and following up with the researchers. They reflected on these questions in our studies, with one saying that "in fabrication papers, most of the experiments are 'demonstrative.' We did this, and this is how it performs, and keep that in mind when you design it," and others highlighting that for some of their studies, the purpose was "more anecdotal than scientific" or "intended to give intuition." This aligns with Ledo, et al.,'s description of a demonstration for toolkit evaluation [47]. There were other questions around whether experiments involved "digital fabrication" or not; certain experiments use the basic machinery of digital fabrication but render objects in a different way, e.g., virtually (for example G-ID's experiment to assess camera angle's effect on detection, which was performed using simulated renders of gcode in Blender (Appendix A.8)). What qualifies an artifact as "digitally-fabricated" was not directly discussed with the original researchers, although the question of difference in rendering has been explored in panels [19].

4.2.3 Researchers' Ideas for Uses of FEDT. Towards our goal of *precision*, many researchers felt that FEDT would be useful for giving an unambiguous specification of an experiment, likely through sharing the Python code as they in general felt that the flowcharts were too large and confusing (we added flowchart reduction after our studies). They agreed that it would help them manage their workflows and reconstruct what they had done when coming back to recorded data, models, or objects years later, particularly in concert with object labeling. They also had ideas on other ways they would use it; e.g., to create "contracts" with student researchers for how an experiment was to be performed, or to find and reuse experimental methodologies as "standards." Most researchers expressed an interest in using FEDT right away.

4.2.4 Researchers' Ideas for Extensions to FEDT. Researchers had many suggestions for how FEDT could be expanded to support other types of workflows (such as by mixing textual and video instructions to communicate precise timing or tacit knowledge in physical crafting performances, or by adding more modules that could support "pure" psychology-style user tests), to enable richer end-to-end evaluation of experiments (such as by adding simulation), to record qualitative occurrences during execution (such as jigs breaking and being replaced), to have periodic checks (such as to ensure a needle remains sharp), or to empower more researcher groups (such as by adding visual coding modes, explicit integration with Jupyter Notebooks, or interaction via the flowchart instead of the coding-and-command line paradigm we currently use). Researchers also requested further functionality that would help them manage the most time-consuming portions of an experimental workflow, such as print time or sample growing times. We elaborate on some of these in Section 6, but notably none of the suggested extensions were related to limitations of the *language* we created: only to limitations of the *interface* or *executors*.

5 Related Work

5.1 Overviews of Fabrication Research and Evaluation

Prior work has discussed fabrication's essential Analog → Digital/Digital → Analog workflow, which links it to other processes

like page layouting [8]; we extract a workflow of fabrication *experiments*, which shares this high-level flow but is more detailed. Other categorizations of fabrication literature also exist, for example organizing works into a design space of 3D printable interactivity [5] or defining functional links between fabrication and sensing [70], but these focus on *what* researchers are exploring, while we focus here on *how* it is being explored through experiments, and how that process can be supported. Fabrication machine benchmarking tools and techniques also exist [42, 49], but focus on tools only instead of workflows. Findings of such processes could be integrated into our experimental reporting. Somewhat closer to experiments, Ledo, et al., explore evaluation techniques for toolkits [47]: they describe demonstration, usage, performance, and heuristics as strategies employed by researchers. Here, we examine a different slice of papers (fabrication-related rather than toolkits) and more deeply explore experiments beyond benchmarking; this complements Ledo, et al.,'s definition of technical evaluations.

5.2 Formal Grammars for HCI, for Fabrication, and for Workflows

The study of formal grammars began as a way to capture the structure of human language [13], but in modern computer science they capture commonalities between a wide variety of structures. The Vega family of grammars [53, 67, 68], for example, capture the common structure of graphical visualizations, while Taxon [80] captures commonalities between fabrication machines. In this vein, the language underlying FEDT captures the common structure of fabrication experiments.

But grammars like Vega's and Taxon's do not simply capture commonalities between structures, they act as user interfaces for constructing those structures; they are fully-fledged *domain-specific languages*. These kinds of user interfaces abound in the fabrication literature. O'Leary, et al., have published several systems leveraging programmatic interfaces to fabrication machines, e.g., Imprimer [79], work which was expanded by Dynamic Toolchains [81]; Fossdal, et al.,'s Vespidae likewise explores path planning and visualization for fabrication hardware [28]. This work is highly relevant to our own and informs it on a conceptual level, but examines fabrication devices and their functionality rather than explicit, formal experiments performed with them, as is our focus here.

Nandi, et al., have likewise built formal, grammar-based representations of 3D printed and printable models, enabling decompilation of low-level geometry into high-level DSLs [56, 57], while OPTIMISM focuses on an optimisation language for creating fabricated objects through collaboration between programmers and domain experts [36]. These are inspiring to our exploration, but focus on geometry instead of a full pipeline. Drill Sergeant [72], HELM [84], and ICEE [88] use DSLs for carpentry, enabling users to create parametric fabrication plans and modify them while adhering to physical constraints and available manufacturing processes. Our focus is on precise definition of methodology, rather than enabling improvisation, and FEDT interacts with all stages of an experiment-focused fabrication pipeline rather than only the actual manufacturing step for a pre-defined single object.

Domain specific languages are also emerging as tools for describing parts of the experimental process, including data *collection* and data *analysis*. SureVeyor, like FEDT, targets data collection. It allows users to define high-quality online surveys, with a suite of tools for de-biasing question presentation [75]. While FEDT supports a very different flavor of data collection, both languages provide an unambiguous way to describe—and potentially replicate—a given data collection procedure. When it comes to data analysis, there are two main lines of work. Tea [40] and Tisane [41] support small to medium scale experimental analysis: Tea helps users to choose statistical tests based on their data assumptions, and Tisane supports specification and execution of generalized linear mixed-effect models. Another option for larger data analysis is multiverse analysis, embodied in DSLs like Boba [51] and multiverse [66]. These allow the user to clearly communicate how their data analysis was chosen from space of possible analyses. Our language is orthogonal to these data analysis concerns: FEDT could describe the data collection part of a fabrication experiment and Tea, Tisane, Boba, or multiverse could express the analysis portion.

5.3 Open Science in HCI

HCI is supposed to have a replication crisis [23, 26], with accusations of sloppy statistics and validity slippage leveled against the field at least as early as 1998 [32]. Meanwhile, Ben Shneiderman called out hypothesis generation for scientists as an area to focus on with creativity support tools in 2007 [74], as a chance to help "users to rapidly generate multiple alternatives, explore their implications, or revert to earlier stages when needed." This north star has inspired various kinds of research in HCI. One prominent area builds tools to help scientists with their work. Schwind, et al., build a system to help structure user studies and their analyses based on goals [73], while Aperitif [58] scaffolds pre-registrations and auto-generates analysis code: these are features we would like to integrate into FEDT in the future. Howell and Bateman [37] discuss how sharing making instructions with scientists and industrial colleagues can help increase work's impact—with FEDT, we perform additional analyses and build tools to actually facilitate this process.

A meta area of HCI has explored how open we are as a community. This analysis has been performed via crawling journals [6], authors' self-reports [82], or post-hoc analysis of publications [65]. Echtler and Häußler found that open-sourcing is rare in HCI for several reasons, "e.g. anxiety that one's own code is 'not good enough' for the public [7], not wanting to provide future support for the software, [or] social barriers when contributing to an existing project [78]" [23]. Recent findings have highlighted that many in HCI do not share their artifacts (software or hardware) due to a combination of misunderstanding reasons or methods for sharing [82]: we hope that codifying experimental practices into short, procedure-focused methods and abstracting execution to a second layer of code will help ease this for future researchers.

Somewhat adjacent to HCI, conferences like ACM REP [1] have opened up new opportunities for studying tools intended for replicability of scientific experiments [17]. However, fabrication experiments have so far not been the focus of such tools.

5.4 Experiments and Replicability in Other Fields

Other fields have their own structures and styles of experiments and documentation, which we can gain inspiration from moving forward. Prior work has compared reproducibility and replicability efforts across tools [17] and fields [62]. In particular, the analysis by Rahal, et al. [62] showed that while transparency is relevant in a wide range of areas like social sciences, humanities, life sciences, engineering, and psychology, the stage at which transparency become relevant depends on the field—while in basic research, it becomes relevant as the experiment is being planned and remains relevant till publication, in applied research often the focus is on the overall method rather than reproducing exact experimental results. The question of what components are most important to replicability in HCI fabrication research remains one for community discussion.

Fabrication experiments are complex, involving both tools and human intervention, interleaved through the steps of the workflow. Replicability of these experiments therefore requires documenting the exact steps in the right order, capturing both the design and manufacturing components, all of which is enabled by FEDT.

Researchers in "laboratory sciences" like chemistry and physics maintain physical lab notebooks (in some cases, as a legal requirement to comply with audits and/or to support intellectual property dating) documenting their experiments: this tradition developed into digital lab notebooks, like Jupyter[10]. For pure computation, tools like WholeTale[11], RenkuLab[12], and ReproZip[13] facilitate reproducibility of scientific experiments by bundling data, code, and environment variables and offering a robust platform that reduces experimental setup time. Some projects take digitization even further, with physical and virtual integration of plans and data with machines: LabGuru[14] enables the recording of procedures, tracking of supplies, time-sharing of equipment, and running of experiments through web-enabled sensing and actuation devices and ID tags. These tools serve as inspiration for many more features that could be implemented in future FEDT versions.

6 Discussion

FEDT's groundwork, design, and implementation have all been informed by the authors' academic backgrounds (HCI, fabrication, engineering design, and programming languages).

HCI has a rich tradition of being inspired by and heavily citing work from adjacent disciplines, such as materials science [60, 61], mechanical engineering [4], robotics [50] and architecture [46]. Indeed, researchers we spoke with remarked that researchers in other disciplines tend to report more formal experiments than we do in HCI: in collaboration with such researchers, the domain-specific activities captured in our DSL could help reveal and formalize inter-area differences and assumptions.

Here, we reflect on some of the key points we see as left unsolved and develop visions for where else FEDT could go.

[10]https://jupyter.org
[11]https://wholetale.org/
[12]https://renkulab.io/
[13]https://www.reprozip.org/
[14]https://labguru.com

6.1 Limitations

Research is not without limitations: the choices we made here are certainly not the only valid ones. Most importantly, we acknowledge limitation in our sampling, in the choice and extent of our implementation, and in our selected evaluation methods.

6.1.1 Sampling. The papers we sampled for evaluating both our workflow and FEDT are limited in number, and we selected them purposively rather than systematically or exhaustively. Future samples may show ways in which FEDT must expand to describe all possible fabrication publications.

6.1.2 Overall Approach. Our choice to implement FEDT as a DSL does not come without trade-offs. DSLs often have a significant learning curve and may pose a barrier to entry for those who are not familiar with programming; they are also traditionally text-based and therefore "unimodal," whereas experimental designs often include images, sounds, and other modalities.

We mitigated these downsides with FEDT by basing it on Python: as languages go, Python is fairly beginner-friendly, and it also has extensive support for multi-modal programs in literate environments like notebooks. The upsides of DSLs are foundational to FEDT: the complexity of this space requires an interface that is flexible, compositional, and extensible—all things DSLs excel at.

At the end of the day, a DSL may not be the user interface that the community chooses to fill this niche. FEDT may become the "backend" for a more accessible interface (e.g., a drag-and-drop editor) or it may simply serve as inspiration and guidance for a new generation of approaches. Balancing the needs of a highly-interdisciplinary research community like ours is not the work of one paper, but we see FEDT as a first step towards a future tool for researchers, supervisors, reviewers, and others to use to build, execute, and communicate experiments. It can already serve as a part of a balanced experiment documentation package that includes pre-registrations, photos, videos, and prose.

6.1.3 Implementation. FEDT is intended as a research probe, rather than a fully-engineered system: we have yet to explore its use in non-command-line environments, and the library implementation we have released is confined to our own labs' machines. FEDT has not been engineered to robustly help users recover from errors, crashes, broken machinery, etc., in the process of running experiments. However, each workflow stage results in an object (whether virtual or physical), which encapsulates the choices made up to that point: these could provide natural points from which to resume.

6.1.4 Evaluation. Finally, for this paper, we have focused on evaluating utility rather than usability: we hope that future work will explore FEDT's applicability to the creation of novel experiments and the difficulties that may be encountered in using it by others than the authors of this paper.

6.2 Handling Randomness and Simulations

While FEDT is intended to help researchers clearly specify executed experiments' methods and what they intentionally vary, there are two main sources of randomness that cannot be captured: the environment and humans-in-the-loop. A simulation can only capture the environment as it is programmed to, and the real world will

always be messier: one could replace experiment steps with simulations (supported in FEDT on a technical level), but in some cases this belies their complexity. For example, filament age or a room's humidity on a given day can affect the quality of a 3D print, the precise bend in a sheet of acrylic from the way it was stored can affect laser-cuttability, and non-homogeneities in natural materials (e.g., wood knots) can require changing spindle speeds for CNC machining. Including users and their bodies, actions, or judgment in the process exacerbates this issue. Academics have argued about whether digital fabrication processes can or should be treated like compilation [34], but *no real-world experiment description can contain all the uncontrolled variables present.* Repeating experimental conditions is a best-practice intended to mitigate such variables, however it is not a perfect solution. Replicating identical methods under varied conditions is one way to discover important uncontrolled variables, and this is one way in which FEDT can assist. Our goal with FEDT is to enable completing an experiment in a particular way, *not* to ensure getting the same results.

6.3 Experiments and Experimentation Going Forward

In this work we have distinguished between "experiments"—in which variables are introduced and formal measurements are taken—and "experimentation"—where settings and configurations are tried without formal procedures. This was echoed by our user study participants in their descriptions of "demonstrations" and "giving intuition." It may also be necessary to further define and formalize what constitutes a fabrication experiment: as noted, one of the experiments we modeled (where a machine was treated both as a site of fabrication and as a fabricated object to be tested) required some awkward implementation. Experimentation has been formally examined by, e.g., O'Leary, et al. [79], and we do not explore it here. That said, some interviewed experts wished they had been more diligent in tracking their experimentation: we do believe that using our language for this task, especially with its ability to generate labeled geometry, would be possible.

Going forward, we hope FEDT finds a home with fabrication researchers and students, and that its *extensible* nature will enable this. While we do not here call out specific papers we analyzed with errors in their textual reporting, methods, or graphs, we did encounter a non-trivial number of these which had made it through the peer review process to publication. It is also the case that not every fabrication paper described an experiment.

Making something easier can make it more common. Formalizing methods and making them more portable may also enable benchmarking, or give researchers a simpler way to find experiments appropriate to their studies. Future work should explore creating a searchable database of experiments—perhaps even by previously-published researchers themselves during a workshop or a Wikipedia-style edit-a-thon—and possible integration with initiatives in the IEEE VIS community [27], the Graphics Replicability Stamp Initiative (GRSI)[15], or The Many Labs project [16].

On the flipside, making something easier can make it more formulaic: FEDT could invite future conversations about which

```
@fedt_experiment
def benchy_PrinterA():
  benchy_stl = GeometryFile('benchy.stl')
  results = BatchMeasurements.empty()
  for repetition in Parallel(range(3)):
    fabbed_object = Printer.slice_and_print(benchy,
                      printer='Formlabs Form 2',
                      repetition=repetition)
    results += Calipers.measure_size(fabbed_object, "x-axis")
  analyze(results.get_all_data())

@fedt_experiment
def benchy_PrinterB():
  benchy_stl = GeometryFile('benchy.stl')
  results = BatchMeasurements.empty()
  for repetition in Parallel(range(3)):
    fabbed_object = Printer.slice_and_print(benchy,
                      printer='Formlabs Form 2',
                      repetition=repetition)
    results += Calipers.measure_size(fabbed_object, "x-axis")
    results += Calipers.measure_size(fabbed_object, "y-axis")
    results += Calipers.measure_size(fabbed_object, "z-axis")
  analyze(results.get_all_data())
```

Figure 7: Two FEDT programs for 3D printing a model. Highlights show the difference between the two programs—`benchy_PrinterA` only measures a single dimension of the printed model whereas `benchy_PrinterB` measures three different dimensions. An automated group program analyzer can be used to detect these differences and suggest additional experiments or steps to FEDT users.

benchmarks are important for our community, and where non-experiment-based technical evaluations are more appropriate [47].

6.4 FEDT's Analysability and Group Analysis of Experimental Programs

FEDT's DSL approach opens up additional research directions like *group program analysis*, which statically analyzes a set of programs to automatically infer a library of reusable components [11, 22, 24]. Group program analysis can also be leveraged to automatically suggest new or extended experiments to users of FEDT. For example, consider two FEDT programs, both of which 3D print the popular Benchy model[16]. One of them measures the length, width, and height of the model whereas the other only measures the length (Figure 7). Group program analysis could detect these differences and suggest measuring the additional dimensions. We envision a future automated group program analyzer that, given two (or more) input programs, can detect differences and similarities to (1) learn a general, reusable library of functions, and (2) suggest missing steps.

6.5 Applicability to Other Fabrication Processes

Given that the workflow in Section 2 forms the foundation of FEDT, and that it can capture varied processes, we expect our tool is useful for a variety of fabrication techniques beyond what we test here.

While we only created executors for laser cutting and FFF 3D printing, as we had access to these in our labs, we modeled experiments using other manufacturing techniques and believe their executors could be added due to FEDT's *extensible* nature. For now, it is possible to capture any fabrication description which can be

[15] https://www.replicabilitystamp.org/

[16] https://www.3dbenchy.com

communicated in text and/or code: this invites some flexibility to describe hands-on processes (like the moulding required for Design, Mould, Grow, Appendix A.5), to include sensor input in fabrication processes (as in CircWood, Appendix A.4), or to write new executors that connect with other or novel API-driven machines (as in SPEERLoom, Appendix A.10.3).

6.6 Further Extensions

As an open-source library, FEDT can be extended to interface with different physical measurement or fabrication devices or connect with software-based statistical and plotting tools. It could also integrate with other machine specification and workflow development tools like Taxon [80].

Programming via direct manipulation of code or outputs (as in Scratch [63], Gliimpse [21], i-Latex [30], or Sketch-n-Sketch [14]) could be an interesting direction, and future work should explore what the right interface for interacting with an executing experiment might be. Web-based flowcharts[17], or gesture– [12] or voice-controlled[18] agents are all possibilities.

Power-users of FEDT might want to express even more complex experiments. We envision extensions based on advances in the programming languages literature. For example, ownership types [15] (like those in Rust[19]) could allow tracking that fabricated objects are not destroyed before all necessary measurements have been made. Choreographic programming [54] may allow users to automatically align printing or growing times. Partial evaluation [29] could allow users to perform experiments in "phases"; creating certain objects in one run of an FEDT program and re-using those same objects to continue follow-on experiments at a later time. Challenges await in integrating such software paradigms with the messy real world: for example, pausing execution of an algorithm allows it to trivially resume, but pausing a print for an extended period during partial evaluation may make new layers stick less effectively.

Encoding intention into FEDT programs, as highlighted by some study participants, could be done manually by adding comments and running existing documentation tools like doxygen[20], or automatically through analysis. For example, looking at `git` or Jupyter Notebook [33] logs of experiments as they are being authored or experimentation plans as they are run, can help understand how researchers go about selecting and narrowing down variables. How researchers fork or remix others' FEDT experiments could also give interesting provenance information, and hook into social processes around peer review and validation.

FEDT makes it possible to specify the execution of an experiment, and to render it in a flowchart, but with some additional work the Python code or flowchart could be rendered in human-readable language, possibly at several different levels of detail (to suit an abstract, introduction, methods section, appendix, or IRB filing). We experimented with using LLMs to generate readable experiment descriptions from FEDT programs, but struggled with their tendency to hallucinate. Pure programmatic analysis may be a better fit for this.

[17] SnapGene https://www.snapgene.com
[18] LabTwin https://www.labtwin.com/
[19] https://www.rust-lang.org/
[20] https://www.doxygen.nl/

7 Conclusion

We analyzed a collection of recent fabrication papers from top-tier HCI conferences to better understand their experimental structures. From this, we created a workflow and a domain-specific language to capture existing practices. We implemented a tool and DSL called FEDT, which helps researchers plan and execute fabrication experiments. Using FEDT, we modeled 42 experiments from 10 purposively-sampled papers, and present our reflections from this task as well as responses and corrections from the original papers' authors. We report expert researchers' thoughts about the tool and its possibilities for bringing HCI fabrication research into the future.

Acknowledgments

The authors would like to thank the anonymous reviewers and shepherd for their input and enthusiasm, as well as Eunice Jun for interesting discussions about modeling for science. Sarah Chasins helped bring some of the authors together that led to this collaboration. This work was partially supported by a Novo Nordisk Fonden Starting Grant under grant number NNF21OC0072716, and NSERC RGPIN-2023-03533.

References

[1] 2024. *ACM REP '24: Proceedings of the 2nd ACM Conference on Reproducibility and Replicability* (Rennes, France). Association for Computing Machinery, New York, NY, USA.

[2] Zahin Ahmed, Farishta Jayas Kinjol, and Ishrat Jahan Ananya. 2021. Comparative Analysis of Six Programming Languages Based on Readability, Writability, and Reliability. In *2021 24th International Conference on Computer and Information Technology (ICCIT)*. 1–6. https://doi.org/10.1109/ICCIT54785.2021.9689813

[3] Roland Aigner, Mira Alida Haberfellner, and Michael Haller. 2022. SpaceR: Knitting Ready-Made, Tactile, and Highly Responsive Spacer-Fabric Force Sensors for Continuous Input. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3526113.3545694 event-place: Bend, OR, USA.

[4] Marwa Alalawi, Noah Pacik-Nelson, Junyi Zhu, Ben Greenspan, Andrew Doan, Brandon M Wong, Benjamin Owen-Block, Shanti Kaylene Mickens, Wilhelm Jacobus Schoeman, Michael Wessely, Andreea Danielescu, and Stefanie Mueller. 2023. MechSense: A Design and Fabrication Pipeline for Integrating Rotary Encoders into 3D Printed Mechanisms. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3544548.3581361 event-place: Hamburg, Germany.

[5] Rafael Ballagas, Sarthak Ghosh, and James Landay. 2018. The Design Space of 3D Printable Interactivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2, 2 (July 2018), 61:1–61:21. https://doi.org/10.1145/3214264

[6] Nick Ballou, Vivek R. Warriar, and Sebastian Deterding. 2021. Are You Open? A Content Analysis of Transparency and Openness Guidelines in HCI Journals. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3411764.3445584

[7] Nick Barnes. 2010. Publish your computer code: it is good enough. *Nature* 467, 7317 (Oct. 2010), 753–753. https://doi.org/10.1038/467753a Number: 7317 Publisher: Nature Publishing Group.

[8] Patrick Baudisch and Stefanie Mueller. 2016. Personal Fabrication: State of the Art and Future Research. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '16)*. Association for Computing Machinery, New York, NY, USA, 936–939. https://doi.org/10.1145/2851581.2856664 event-place: San Jose, California, USA.

[9] Dustin Beyer, Serafima Gurevich, Stefanie Mueller, Hsiang-Ting Chen, and Patrick Baudisch. 2015. Platener: Low-Fidelity Fabrication of 3D Objects by Substituting 3D Print with Laser-Cut Plates. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. Association for Computing Machinery, New York, NY, USA, 1799–1806. https://doi.org/10.1145/2702123.2702225 event-place: Seoul, Republic of Korea.

[10] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology: Instruments of Mind*, Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn (Eds.). Springer, Berlin, Heidelberg, 325–341. https://doi.org/10.1007/3-540-44617-6_31

[11] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 396–424. https://doi.org/10.1145/3571207

[12] Derrick Cheng, Pei-Yu Chi, Taeil Kwak, Björn Hartmann, and Paul Wright. 2013. Body-tracking camera control for demonstration videos. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems* (Paris, France) *(CHI EA '13)*. Association for Computing Machinery, New York, NY, USA, 1185–1190. https://doi.org/10.1145/2468356.2468568

[13] Noam Chomsky. 1959. On certain formal properties of grammars. *Information and Control* 2, 2 (June 1959), 137–167. https://doi.org/10.1016/S0019-9958(59)90362-6

[14] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/2908080.2908103

[15] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Springer, Berlin, Heidelberg, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3

[16] Open Science Collaboration. 2015. Estimating the reproducibility of psychological science. *Science* 349, 6251 (2015), aac4716. https://doi.org/10.1126/science.aac4716 arXiv:https://www.science.org/doi/pdf/10.1126/science.aac4716

[17] Lázaro Costa, Susana Barbosa, and Jácome Cunha. 2024. Evaluating Tools for Enhancing Reproducibility in Computational Scientific Experiments. In *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability* (Rennes, France) *(ACM REP '24)*. Association for Computing Machinery, New York, NY, USA, 46–51. https://doi.org/10.1145/3641525.3663623

[18] Laura Devendorf and Kimiko Ryokai. 2015. Being the Machine: Reconfiguring Agency and Control in Hybrid Fabrication. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. Association for Computing Machinery, New York, NY, USA, 2477–2486. https://doi.org/10.1145/2702123.2702547 event-place: Seoul, Republic of Korea.

[19] Mustafa Doga Dogan, Patrick Baudisch, Hrvoje Benko, Michael Nebeling, Huaishu Peng, Valkyrie Savage, and Stefanie Mueller. 2022. Fabricate It or Render It? Digital Fabrication vs. Virtual Reality for Creating Objects Instantly. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3491101.3516510 event-place: New Orleans, LA, USA.

[20] Mustafa Doga Dogan, Faraz Faruqi, Andrew Day Churchill, Kenneth Friedman, Leon Cheng, Sriram Subramanian, and Stefanie Mueller. 2020. G-ID: Identifying 3D Prints Using Slicing Parameters. 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/10.1145/3313831.3376202 event-place: Honolulu, HI, USA.

[21] Pierre Dragicevic, Stéphane Huot, and Fanny Chevalier. 2011. Gliimpse: Animating from markup code to rendered documents and vice versa. In *Proceedings of the 24th annual ACM symposium on User interface software and technology (UIST '11)*. Association for Computing Machinery, New York, NY, USA, 257–262. https://doi.org/10.1145/2047196.2047229

[22] Sebastijan Dumancic, Tias Guns, and Andrew Cropper. 2021. Knowledge Refactoring for Inductive Program Synthesis. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 8 (May 2021), 7271–7278. https://doi.org/10.1609/aaai.v35i8.16893

[23] Florian Echtler and Maximilian Häußler. 2018. Open Source, Open Science, and the Replication Crisis in HCI. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems (CHI EA '18)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3170427.3188395

[24] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 835–850. https://doi.org/10.1145/3453483.3454080

[25] Cathy Mengying Fang, Jianzhe Gu, Lining Yao, and Chris Harrison. 2022. ElectriPop: Low-Cost, Shape-Changing Displays Using Electrostatically Inflated Mylar Sheets. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3491102.3501837 event-place: New Orleans, LA, USA.

[26] Sebastian S. Feger, Sünje Dallmeier-Tiessen, Paweł W. Woźniak, and Albrecht Schmidt. 2019. The Role of HCI in Reproducible Science: Understanding, Supporting and Motivating Core Practices. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems (CHI EA '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3290607.3312905

[27] Jean-Daniel Fekete and Juliana Freire. 2020. Exploring Reproducibility in Visualization. *IEEE Computer Graphics and Applications* 40, 5 (Sept. 2020), 108–119. https://doi.org/10.1109/MCG.2020.3006412 Conference Name: IEEE Computer Graphics and Applications.

[28] Frikk H Fossdal, Vinh Nguyen, Rogardt Heldal, Corie L. Cobb, and Nadya Peek. 2023. Vespidae: A Programming Framework for Developing Digital Fabrication Workflows. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference (DIS '23)*. Association for Computing Machinery, New York, NY, USA, 2034–2049. https://doi.org/10.1145/3563657.3596106

[29] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher Order Symbol. Comput.* 12, 4 (dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496

[30] Camille Gobert and Michel Beaudouin-Lafon. 2022. i-LaTeX : Manipulating Transitional Representations between LaTeX Code and Generated Documents. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3491102.3517494

[31] Phillip Gough, Praneeth Bimsara Perera, Michael A. Kertesz, and Anusha Withana. 2023. Design, Mould, Grow!: A Fabrication Pipeline for Growing 3D Designs Using Myco-Materials. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3544548.3580958 event-place: Hamburg, Germany.

[32] Wayne D. Gray and Marilyn C. Salzman. 1998. Damaged Merchandise? A Review of Experiments That Compare Usability Evaluation Methods. *Human–Computer Interaction* 13, 3 (Sept. 1998), 203–261. https://doi.org/10.1207/s15327051hci1303_2 Publisher: Taylor & Francis _eprint: https://doi.org/10.1207/s15327051hci1303_2.

[33] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

[34] Mare Hirsch, Gabrielle Benabdallah, Jennifer Jacobs, and Nadya Peek. 2023. Nothing Like Compilation: How Professional Digital Fabrication Workflows Go Beyond Extruding, Milling, and Machines. *ACM Trans. Comput.-Hum. Interact.* 31, 1 (Nov. 2023), 13:1–13:45. https://doi.org/10.1145/3609328

[35] Megan Hofmann, Lea Albaugh, Ticha Sethapakadi, Jessica Hodgins, Scott E. Hudson, James McCann, and Jennifer Mankoff. 2019. KnitPicking Textures: Programming and Modifying Complex Knitted Textures for Machine and Hand Knitting. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 5–16. https://doi.org/10.1145/3332165.3347886 event-place: New Orleans, LA, USA.

[36] Megan Hofmann, Nayha Auradkar, Jessica Birchfield, Jerry Cao, Autumn G Hughes, Gene S-H Kim, Shriya Kurpad, Kathryn J Lum, Kelly Mack, Anisha Nilakantan, Margaret Ellen Seehorn, Emily Warnock, Jennifer Mankoff, and Scott E Hudson. 2023. OPTIMISM: Enabling Collaborative Implementation of Domain Specific Metaheuristic Optimization. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/3544548.3580904

[37] Larry L. Howell and Terri Bateman. 2023. Extending research impact by sharing maker information. *Nature Communications* 14, 1 (Oct. 2023), 6170. https://doi.org/10.1038/s41467-023-41886-3

[38] 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* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 459–468. https://doi.org/10.1145/2556288.2557338

[39] Ayaka Ishii, Kunihiro Kato, Kaori Ikematsu, Yoshihiro Kawahara, and Itiro Siio. 2022. CircWood: Laser Printed Circuit Boards and Sensors for Affordable DIY Woodworking. In *Sixteenth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3490149.3501317 event-place: Daejeon, Republic of Korea.

[40] Eunice Jun, Maureen Daum, Jared Roesch, Sarah Chasins, Emery Berger, Rene Just, and Katharina Reinecke. 2019. Tea: A High-level Language and Runtime System for Automating Statistical Analysis. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 591–603. https://doi.org/10.1145/3332165.3347940

[41] Eunice Jun, Audrey Seo, Jeffrey Heer, and René Just. 2022. Tisane: Authoring Statistical Models via Formal Reasoning from Conceptual and Data Relationships. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3491102.3501888

[42] Shohei Katakura, Martin Taraz, Muhammad Abdullah, Paul Methfessel, Lukas Rambold, Robert Kovacs, and Patrick Baudisch. 2023. Kerfmeter: Automatic Kerf Calibration for Laser Cutting. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3544548.3580914 event-place: Hamburg, Germany.

[43] Arshad Khan, Joan Sol Roo, Tobias Kraus, and Jürgen Steimle. 2019. Soft Inkjet Circuits: Rapid Multi-Material Fabrication of Soft Circuits Using a Commodity Inkjet Printer. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/3332165.3347892 event-place: New Orleans, LA, USA.

[44] Marion Koelle, Madalina Nicolae, Aditya Shekhar Nittala, Marc Teyssier, and Jürgen Steimle. 2022. Prototyping Soft Devices with Interactive Bioplastics. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 19, 16 pages. https://doi.org/10.1145/3526113.3545623

[45] Robert Kovacs, Lukas Rambold, Lukas Fritzsche, Dominik Meier, Jotaro Shigeyama, Shohei Katakura, Ran Zhang, and Patrick Baudisch. 2021. Trusscillator: A System for Fabricating Human-Scale Human-Powered Oscillating Devices. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, New York, NY, USA, 1074–1088. https://doi.org/10.1145/3472749.3474807 event-place: Virtual Event, USA.

[46] Maria Larsson, Hironori Yoshida, Nobuyuki Umetani, and Takeo Igarashi. 2020. Tsugite: Interactive Design and Fabrication of Wood Joints. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 317–327. https://doi.org/10.1145/3379337.3415899

[47] 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)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3173574.3173610

[48] Danny Leen, Nadya Peek, and Raf Ramakers. 2020. LamiFold: Fabricating Objects with Integrated Mechanisms Using a Laser Cutter Lamination Workflow. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 304–316. https://doi.org/10.1145/3379337.3415885 event-place: Virtual Event, USA.

[49] H. Li, L. Verdi, S. Chang, L. Verdi, and S. Chang. 2018. Benchmarking 3D Printers’ Resolutions with Geometric Element Test Targets. *Journal of Imaging Science and Technology* 62 (Jan. 2018), 1–8. https://doi.org/10.2352/J.ImagingSci.Technol.2018.62.1.010504 Publisher: Society for Imaging Science and Technology.

[50] 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 York, NY, USA, 673–685. https://doi.org/10.1145/3332165.3347894

[51] Yang Liu, Alex Kale, Tim Althoff, and Jeffrey Heer. 2021. Boba: Authoring and Visualizing Multiverse Analyses. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (Feb. 2021), 1753–1763. https://doi.org/10.1109/TVCG.2020.3028985 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[52] Marco Livesu, Stefano Ellero, Jonàs Martínez, Sylvain Lefebvre, and Marco Attene. 2017. From 3D models to 3D prints: an overview of the processing pipeline. *Computer Graphics Forum* 36, 2 (2017), 537–564. https://doi.org/10.1111/cgf.13147 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13147.

[53] Andrew M McNutt and Ravi Chugh. 2021. Integrated Visualization Editing via Parameterized Declarative Templates. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (, Yokohama, Japan,) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 17, 14 pages. https://doi.org/10.1145/3411764.3445356

[54] Fabrizio Montesi. 2013. *"Choreographic Programming"*. PhD thesis. IT-Universitetet i København, Copenhagen, Denmark. Available at https://portal.findresearcher.sdu.dk/en/publications/choreographic-programming.

[55] Ilan E Moyer, Samuelle Bourgault, Devon Frost, and Jennifer Jacobs. 2024. Throwing Out Conventions: Reimagining Craft-Centered CNC Tool Design through the Digital Pottery Wheel. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 347, 22 pages. https://doi.org/10.1145/3613904.3642361

[56] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional programming for compiling and decompiling computer-aided design. *Proceedings of the ACM on Programming Languages* 2, ICFP (July 2018), 99:1–99:31. https://doi.org/10.1145/3236794

[57] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

[58] Yuren Pang, Katharina Reinecke, and René Just. 2022. Apéritif: Scaffolding Preregistrations to Automatically Generate Analysis Code and Methods Descriptions. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3491102.3517707

[59] Jonathan Peirce, Jeremy R. Gray, Sol Simpson, Michael MacAskill, Richard Höchenberger, Hiroyuki Sogo, Erik Kastman, and Jonas Kristoffer Lindeløv. 2019. PsychoPy2: Experiments in behavior made easy. *Behavior Research Methods* 51, 1 (Feb. 2019), 195–203. https://doi.org/10.3758/s13428-018-01193-y

[60] Isabel P. S. Qamar, Sabina W Chen, Dimitri Tskhovrebadze, Paolo Boni, Faraz Faruqi, Michael Wessely, and Stefanie Mueller. 2022. ChromoPrint: A Multi-Color 3D Printer Based on a Reprogrammable Photochromic Resin. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems (CHI EA '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3491101.3519784 event-place: New Orleans, LA, USA.

[61] Isabel P. S. Qamar, Rainer Groh, David Holman, and Anne Roudaut. 2018. HCI meets Material Science: A Literature Review of Morphing Materials for the Design of Shape-Changing Interfaces. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–23. https://doi.org/10.1145/3173574.3173948

[62] Rima-Maria Rahal, Hanjo Hamann, Hilmar Brohmer, and Florian Pethig. 2022. Sharing the Recipe: Reproducibility and Replicability in Research Across Disciplines. *Research Ideas and Outcomes* 8 (Nov. 2022). https://doi.org/10.3897/rio.8.e89980

[63] 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

[64] Thijs Roumen, Ingo Apel, Jotaro Shigeyama, Abdullah Muhammad, and Patrick Baudisch. 2020. Kerf-Canceling Mechanisms: Making Laser-Cut Mechanisms Operate across Different Laser Cutters. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Association for Computing Machinery, New York, NY, USA, 293–303. https://doi.org/10.1145/3379337.3415895

[65] Kavous Salehzadeh Niksirat, Lahari Goswami, Pooja S. B. Rao, James Tyler, Alessandro Silacci, Sadiq Aliyu, Annika Aebli, Chat Wacharamanotham, and Mauro Cherubini. 2023. Changes in Research Ethics, Openness, and Transparency in Empirical Studies between CHI 2017 and CHI 2022. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–23. https://doi.org/10.1145/3544548.3580848

[66] Abhraneel Sarma, Alex Kale, Michael Jongho Moon, Nathan Taback, Fanny Chevalier, Jessica Hullman, and Matthew Kay. 2023. multiverse: Multiplexing Alternative Data Analyses in R Notebooks. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3544548.3580726

[67] 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 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[68] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (Jan. 2016), 659–668. https://doi.org/10.1109/TVCG.2015.2467091 Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[69] Valkyrie Savage, Carlos E. Tejada, and Mengyu Zhong. 2022. AirLogic: Embedding Pneumatic Computation and I/O in 3D Models to Fabricate Electronics-Free Interactive Objects. In *Proceedings of the 35th annual ACM symposium on User interface software and technology (UIST '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3526113.3545642

[70] Valkyrie Arline Savage. 2016. *Fabbed to Sense: Integrated Design of Geometry and Sensing Algorithms for Interactive Objects*. Doctoral thesis. University of California, Berkeley.

[71] Martin Schmitz, Martin Stitz, Florian Müller, Markus Funk, and Max Mühlhäuser. 2019. .../Trilaterate: A Fabrication Pipeline to Design and 3D Print Hover-, Touch-, and Force-Sensitive Objects. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3290605.3300684 event-place: Glasgow, Scotland Uk.

[72] 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)*. Association for Computing Machinery, New York, NY, USA, 1607–1614. https://doi.org/10.1145/2851581.2892429

[73] Valentin Schwind, Stefan Resch, and Jessica Sehrt. 2023. The HCI User Studies Toolkit: Supporting Study Designing and Planning for Undergraduates and Novice Researchers in Human-Computer Interaction. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3544549.3585890

[74] Ben Shneiderman. 2007. Creativity support tools: accelerating discovery and innovation. *Commun. ACM* 50, 12 (Dec. 2007), 20–32. https://doi.org/10.1145/1323688.1323689

[75] Ye Shu, Emmie Hine, Hugo Hua, Emery D. Berger, and Daniel W. Barowy. PLATEAU 2024. SureVeyor: A Language for High-Quality Online Surveys.

[76] Madlaina Signer, Alexandra Ion, and Olga Sorkine-Hornung. 2021. Developable Metamaterials: Mass-Fabricable Metamaterials by Laser-Cutting Elastic Structures. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3411764.3445666 event-place: Yokohama, Japan.

[77] Samantha Speer, Ana P Garcia-Alonzo, Joey Huang, Nickolina Yankova, Carolyn Rosé, Kylie A Peppler, James Mccann, and Melisa Orta Martinez. 2023. SPEER-Loom: An Open-Source Loom Kit for Interdisciplinary Engagement in Math, Engineering, and Textiles. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 93, 15 pages. https://doi.org/10.1145/3586183.3606724

[78] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. 2015. Social Barriers Faced by Newcomers Placing Their First Contribution in Open Source Software Projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '15)*. Association for Computing Machinery, New York, NY, USA, 1379–1392. https://doi.org/10.1145/2675133.2675215

[79] Jasper Tran O'Leary, Gabrielle Benabdallah, and Nadya Peek. 2023. Imprimer: Computational Notebooks for CNC Milling. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3544548.3581334 event-place: Hamburg, Germany.

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

[81] Hannah Twigg-Smith, Jasper Tran O'Leary, and Nadya Peek. 2021. Tools, Tricks, and Hacks: Exploring Novel Digital Fabrication Workflows on #PlotterTwitter. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3411764.3445653 event-place: Yokohama, Japan.

[82] Chat Wacharamanotham, Lukas Eisenring, Steve Haroz, and Florian Echtler. 2020. Transparency of CHI Research Artifacts: Results of a Self-Reported Survey. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376448

[83] Guanyun Wang, Ye Tao, Ozguc Bertug Capunaman, Humphrey Yang, and Lining Yao. 2019. A-line: 4D Printing Morphing Linear Composite Structures. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300656

[84] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry compiler. *ACM Trans. Graph.* 38, 6, Article 195 (nov 2019), 14 pages. https://doi.org/10.1145/3355089.3356518

[85] Junichi Yamaoka, Ryuma Niiyama, and Yasuaki Kakehi. 2017. BlowFab: Rapid Prototyping for Rigid and Reusable Objects Using Inflation of Laser-Cut Surfaces. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 461–469. https://doi.org/10.1145/3126594.3126624 event-place: Québec City, QC, Canada.

[86] Zeyu Yan, Tingyu Cheng, Jasmine Lu, Pedro Lopes, and Huaishu Peng. 2023. Future Paradigms for Sustainable Making. In *Adjunct Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23 Adjunct)*. Association for Computing Machinery, New York, NY, USA, Article 109, 3 pages. https://doi.org/10.1145/3586182.3617433

[87] Zeyu Yan and Huaishu Peng. 2021. FabHydro: Printing Interactive Hydraulic Devices with an Affordable SLA 3D Printer. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, New York, NY, USA, 298–311. https://doi.org/10.1145/3472749.3474751 event-place: Virtual Event, USA.

[88] Haisen Zhao, Max Willsey, Amy Zhu, Chandrakana Nandi, Zachary Tatlock, Justin Solomon, and Adriana Schulz. 2022. Co-Optimization of Design and Fabrication Plans for Carpentry. *ACM Trans. Graph.* 41, 3, Article 32 (mar 2022), 13 pages. https://doi.org/10.1145/3508499

A Modeling 10 Exemplar Papers' Experiments

To demonstrate our language's flexibility, we have replicated all described experiments from 10 exemplar papers. These are fully-compileable examples, although we have only done end-to-end replication with the 2 described in Section 4.1.2. For each of these papers and each experiment therein, we present (1) the text of the original paper, (2) a replication in FEDT code that we created on our own, and (3) a replication after discussion with the original author (where applicable), along with notes on changes made. For the original text, we have transcribed it to the best of our ability without having access to source; we focus on reproducing only text and not figures or references, i.e., reference numbers are from the original papers and do not correspond to our bibliography. Some text formatting has been adapted to fit within the appendix structure, e.g., section titles have been changed to italic text. The samples are also available in our public repository (Section 4.1.2). Note that for margin considerations, we have added line breaks in the programs when necessary, but they are otherwise identical to what is in our repository.

A.1 Paper 1 : ./trilaterate

Adapted from Schmitz, et al., 2019 [71].

A.1.1 Experiment: Touch Location Identification. This experiment uses a single printed object with several interaction points, which users touch in a given sequence after a calibration task.

Original Paper Description. To evaluate our approach, we conducted quantitative evaluations on the sensing of 3D position and force. While research frequently utilizes mechanical apparatuses to evaluate capacitive sensing (cf. [17, 66]), we opted out for a user study with 12 participants (9m, 3f, mean age 27.3) to account for inter-individual differences in users' capacitive responses.

3D Position Since a proper 3D position estimate is crucial for the quality of touch and hover input, we evaluated both in a single study. We were interested in the effects of the following two factors on 3D-printed capacitive trilateration: (1) the coverage, i.e. the number of electrodes that cover a surface point without being shielded by another trace or shield, and (2) the distance, i.e the mean distance of all electrodes to the respective surface point. To that end, we generated a pyramid (length 8 cm, height 6 cm, ground to earth) with eight electrodes and distributed seven target positions on the surface (see Figure 6) such that they cover all combinations of the independent variables coverage (high vs. low) and distance (near vs. far): Positions 2, 3 and 4 are near electrodes (mean distance < 11.24 mm) and highly covered (up to four electrodes). Position 1 is also highly covered, but the electrodes are far distant (mean distance 14.34 mm). Positions 5, 6, and 7 are lowly covered (only two to three electrodes) because the electrodes are partially blocked by traces and shields from the viewpoint of the position. In contrast to position 5 and 7, position 6 is nearer to the electrodes (mean distance 15.01 mm vs. 20 mm).

While the shape of the object is commonly an important factor for capacitive sensing, the trilateration approach is independent of the non-interactive, insulating surface that covers the electrodes, as the differences in measurements are accounted for through the calibration. Since an informal test confirmed this assumption, we did not investigate the effect of object shape and opted out for the pyramid shape in order to make it easier for the participants to localize the target positions.

Setup & Task To be able to perform tests with multiple participants in a repeatable and comparable way, the object was fixed on a wooden plate (see Figure 6). Participants received an introduction to the system before exploring it freely until they felt comfortable. Then, they calibrated the system by touching randomly-generated positions on the pyramid surface. Between tests performed by the same participant, the system was not recalibrated. Each participant was instructed to repeatedly touch the seven target positions ten times per position (counterbalanced using Balanced Latin Square), leading to a total of 840 samples. All target positions were marked on the object to give the participants an exact reference (see Figure 6B). Further, the position to be touched next was highlighted on a virtual model shown on screen. While touching the position, the participant triggered the data recording with a key press.

Results & Discussion As the dependent variable, we analyzed the measurement error, i.e. the Euclidean distance between a target position and the measured 3D point reported by the system, using a two-way repeated measures ANOVA. When significant effects were revealed, we used Bonferroni corrected pairwise t-tests for post-hoc analysis. We report and classify the effect size η^2 according to [9].

The analysis revealed that the distance of electrodes had a highly significant ($F_{1,11}$ = 105.44, p < .001, η^2 = .587) influence on the 3D position estimate with a large effect size. Post-hoc tests confirmed significantly smaller errors for the near distance conditions compared to the far conditions (see Figure 7). Moreover, we found that the coverage of electrodes had a significant ($F_{1,11}$ = 11.23, p < .05, η^2 = .071) influence on the 3D position estimate with a medium effect size. Posthoc tests confirmed significantly smaller errors for the high coverage condition compared to the low coverage condition. Further, we found an interaction effect with a small effect size between the factors ($F_{1,11}$ = 7.4, p < .05, η^2 = .044).

The analysis shows that electrodes should be placed as near as possible to the surface of the object. Also, coverage of at least four electrodes is crucial for a lower error. Hence, the routing algorithm should avoid voxels that are in line-of-sight between a surface point and other electrodes.

Replication in FEDT. To replicate this experiment in FEDT, we created a new `Measurement` tool to represent the custom capacitance-measuring system that was used for the paper. We also created a `latinsquare` function that generates Latin Squares.

```
class CustomCapacitanceSystem:
    capacitances = [Measurement(
```

```
        name="capacitance",
        description="The capacitance of the object on a specific electrode.",
        procedure="""
            Have the user touch the object, then capture the output.
            """,
        units="pF",
        feature=f"electrode {i}") for i in range(7)]

    @staticmethod
    def measure_capacitances(obj: RealWorldObject, feature: str='') -> BatchMeasurements:
        instruction(f"Measure object #{obj.uid}.", header=True)
        return BatchMeasurements.multiple(obj, set(CustomCapacitanceSystem.capacitances))

def latinsquare(n):
    k = n + 1
    sq = []
    for i in range(1, n + 1, 1):
        sqrw = []
        temp = k
        while (temp <= n) :
            sqrw.append(temp)
            temp += 1
        for j in range(1, k):
            sqrw.append(j)
        k -= 1
        sq.append(sqrw)
    return sq

@fedt_experiment
def placement_response():
    # geometry_file = VolumeFile("pyramid.stl")

    # for coverage_condition in ['high','low']:
    #     for distance_condition in ['near','far']:
    #         for mirror_condition in range(1,2):
    #             StlEditor.edit(geometry_file, f"add an electrode with coverage {coverage_condition}
    #                                 and distance {distance_condition}")
    # unsure if this was generated specifically in this way, or if the pyramid was just taken as given
    # designed geometry to cover the conditions

    electrodes = list(range(7))

    fabbed_object = Printer.fab(GeometryFile("pyramid.stl"))
    raw_results = BatchMeasurements.empty()
    test_values = BatchMeasurements.empty()

    for participant in Parallel(range(12)):
        for repetition in Series(range(1)): # not sure how many random touches were required
            User.do(fabbed_object, f"touch electrode #{random.choice(electrodes)}, rep #{repetition}",
                    participant)
        for touch in Series(latinsquare(len(electrodes))[participant % len(electrodes)]): # how truncated?
            User.do(fabbed_object, f"touch electrode #{touch}", participant)
            raw_results += CustomCapacitanceSystem.measure_capacitances(fabbed_object)
            test_values += CustomCapacitanceSystem.measure_capacitances(fabbed_object)

    analyze(raw_results.get_all_data())
```

```
    # some processing to the test values...
    analyze(test_values.get_all_data())
```

Corrected Experiment. The major correction we had to make was to use the same physical object for this experiment and the next. Our other assumptions (e.g., about Latin Square truncation) were correct.

```
PYRAMID = None

def get_pyramid():
    global PYRAMID
    fabbed_object = PYRAMID
    if PYRAMID is None:
        geometry_file = GeometryFile("pyramid.stl")

        for coverage_condition in Parallel(['high','low']):
            for distance_condition in Parallel(['near','far']):
                for mirror_condition in Parallel(range(1,2)):
                    StlEditor.modify_design(geometry_file, "electrode",
                                            "coverage {coverage_condition} and distance {distance_condition}")

        fabbed_object = Printer.fab(geometry_file)

    PYRAMID = fabbed_object
    return PYRAMID

@fedt_experiment
def placement_response():
    electrodes = list(range(7))

    fabbed_object = get_pyramid()

    raw_results = BatchMeasurements.empty()
    test_values = BatchMeasurements.empty()

    for participant in Parallel(range(12)):
        for repetition in Series(range(1)):
            # the same number of repetitions per participant, but not sure of the number
            User.do(fabbed_object, f"touch electrode #{random.choice(electrodes)}, rep #{repetition}", participant)
        for touch in Series(latinsquare(len(electrodes))[participant % len(electrodes)]):
            User.do(fabbed_object, f"touch electrode #{touch}", participant)
            raw_results += CustomCapacitanceSystem.measure_capacitances(fabbed_object)
            test_values += CustomCapacitanceSystem.measure_capacitances(fabbed_object)

    analyze(raw_results.get_all_data())
    # some processing to the test values...
    analyze(test_values.get_all_data())
```

A.1.2 Experiment: Touch Force Identification. This experiment uses an identical object to experiment 1, and users press it with varying levels of force several times, after a calibration task.

Original Paper Description. The same participants continued the previous study to assess force accuracy (without recalibration). As we wanted to evaluate the force accuracy without interference by non-linear deformation effects of the flexible structure, we opted out for position 3 (see Figure 6B). The central location of this position allows for a linear and continuous deformation.

The object allows for a maximal deformation at target position 3 of approx. 5 mm with moderate force. However, in contrast to the position tests, there is no fixed reference scale for the subjective force exerted by a participant. Each participant can perceive the applied force differently or is not at all able to exert the maximum force of another participant. Therefore, each participant sets an individual force scale by first applying a maximum and then a minimum force, confirming each by pressing a key. The test samples are then taken by asking the participant to set a specific target force level (10 % to 90 % in 20 % steps) displayed on a screen and confirm it by pressing a key. The current force applied by the participant is displayed as a reference. Each participant was instructed to set each force level seven times.

The results of the force tests across all participants are depicted in Figure 8. For 10 % target force, the system reported in average 13.12 % (SD 3.17 %). The average for 30 % target force was 31.25 % (SD 4.7 %) and 48.41 % (SD 3.77 %) for 50 % target force. For higher target forces of 70 % and 90 %, the system reported 68.36 % (SD 10.15 %) and 88.61 % (SD 10.78 %). Of note is, that the mean distance between target and measured force level is 1.8 % (SD 0.76 %). That is, the participants hit the desired force level in average within 1.8 %. On a scale of 5 mm (0 % – 100 %), 1.8 % corresponds to a distance resolution of 0.09 mm (5 mm * 0.018).

In summary, our results show that for highly covered, near distance conditions, multiple 3D positions and levels of force applied by users can be reliably distinguished.

Replication in FEDT.

```
@fedt_experiment
def force_response():
    fabbed_object = Printer.fab(GeometryFile("pyramid.stl")) # same single object for both experiments?

    ground_truth = BatchMeasurements.empty()
    test_values = BatchMeasurements.empty()

    forces = list(arange(10,90+include_last,20))

    for participant in Parallel(range(10)):
        for force_level in Parallel([0,100]):
            User.do(fabbed_object, f"touch the top with {force_level}% force", participant)
            ground_truth += CustomCapacitanceSystem.measure_capacitances(fabbed_object)
        for force in Series(7*latinsquare(len(forces))[participant % len(forces)]):
            User.do(fabbed_object, f"touch with force {force}", participant)
            test_values += CustomCapacitanceSystem.measure_capacitances(fabbed_object)

    analyze(ground_truth.get_all_data())
    # some processing to the test_values...
    analyze(test_values.get_all_data())
```

Corrected Experiment. The major correction to this experiment was changing the way in which we ordered touches to participants: we had assumed that the touches were counterbalanced as in the first experiment, but the author indicated that they were fully random across seven repetitions.

```
@fedt_experiment
def force_response():
    fabbed_object = get_pyramid()

    ground_truth = BatchMeasurements.empty()
    test_values = BatchMeasurements.empty()

    forces = list(arange(10,90+include_last,20))

    for participant in Parallel(range(12)):
        for force_level in Parallel([0,100]):
            User.do(fabbed_object, f"touch electrode #3 with {force_level}% force", participant)
            ground_truth += CustomCapacitanceSystem.measure_capacitances(fabbed_object)
        seven_repetitions = forces*7
        for force in Series(shuffle(seven_repetitions)):
            User.do(fabbed_object, f"touch with force {force}", participant)
            test_values += CustomCapacitanceSystem.measure_capacitances(fabbed_object)

    analyze(ground_truth.get_all_data())
    # some processing to the test_values...
    analyze(test_values.get_all_data())
```

A.2 Paper 2 : AirLogic

Adapted from Savage and Tejada, et al., 2022 [69].

A.2.1 Experiment: Air Loss in Widgets. This experiment prints several widgets and pushes air through them at different pressures, measuring preservation of airflow for each.

Original Paper Description. We empirically evaluated widget air operating requirements and losses, as well as the effects of print orientation and internal air channel bed angles. Using an anemometer and pressure sensor, we recorded airspeed into and out of our input and logic widgets at various input levels, airspeed through a single widget type with various printing parameters, and optimal activation pressures for our output widgets.

These experiments highlight two qualities of our designs: airflow needs and printing requirements. Taken together, the findings can inform mass flow needed for a given device, how widgets are best arranged for printing, and widget chaining possibilities.

We used a JunAir 2000-40PD air compressor, a Festo MS4-LR1/4-D5-AS valve, an analog Panasonic PS-A (ADP5151) barometric sensor, and a Kestrel 3500 NV Pocket Weather Anemometer. We printed encapsulated versions of our widgets, connecting their output channels to our barometric pressure sensor and anemometer with of-the-shelf rubber tubes, OD 6 mm, ID 5 mm. Using the measured airspeed, cross-sectional area of our tubes, and density of air, we calculate and report mass flow rate [40] in kg/s. For the printing parameters test, we printed twenty-five copies of our OR widget: four sets of five with the internal tubes angled 0–90 degrees from the airflow direction, and five total with printed internal pipes of bend radiuses from 0–20 mm before the gates.

Air loss in widgets Our tests showed that in general we lose proportionally less airflow when powering our systems with lower airflow (see Figure 7). On average, when powered with 5e-5 kg/s of air, our logic and input widgets lost 33.1% of the airflow, but at 18e-5 kg/s they lost 48.0%. This is likely related to laminarity: some widgets may work best when airflow is closer to laminar, such that it "sticks" properly to the interior walls of the printed tubes. Our logic widgets tended to lose less airflow than our input widgets (26.8% vs. 47.6%), with the XOR widget performing exceptionally well (average loss for non-XOR logic gates: 34.9% vs. 7.38% for XOR). We hypothesize this is related to XOR's relative insensitivity to turbulence (AND is sensitive due to the two-jet interaction, OR is sensitive due to its escape valve geometry). The touch input widget also performed very well, likely because it can be arbitrarily well-sealed at the top—a squishy finger can close an air escape more completely than a rigid piece of plastic. We did not evaluate the pressure losses of our output widgets, as they are intended to be the last element in our transits. While we did not formally measure the escapes out the "wrong" holes, we experienced that with the tested gates there was very little "erroneous" air signal. We did experience more erroneous signal with the multi-way AND gate in our demo application (see Discussion). The button seemed to pass the most signal of our inputs while in the un-pressed configuration, likely since the cap is only slightly above the hole.

We thus recommend powering AirLogic devices with as little airflow as possible, given the constraints of downstream widgets.

Replication in FEDT.

```
@fedt_experiment
def airflow_gatetypes():
    logic_gate_files = ['and.stl', 'or.stl', 'not.stl', 'xor.stl']
    input_files = ['touch.stl', 'button.stl', 'rocker.stl',
                               'dial.stl', 'slider.stl']
    all_widget_files = logic_gate_files + input_files

    results = BatchMeasurements.empty()
    for stl in Parallel(all_widget_files):
        fabbed_object = Printer.fab(GeometryFile(stl))
        for input_massflow in Series(['5e^-5','9.5e^-5','14e^-5','18.5e^-5']):
            instruction(f"set the air compressor to {input_massflow} and connect the object")
            results += Anemometer.measure_airflow(fabbed_object, f"input massflow at {input_massflow}")

    # it's possible this is two experiments?
    # analyze([result in results if result.related_object in logic_gate_files])
    # analyze([result in results if result.related_object in input_files])
    analyze(results.get_all_data())
```

Corrected Experiment. No corrections were required.

A.2.2 Experiment: Air Loss from Print and Pipe Curvature. This experiment evaluates different geometry parameters to determine how they affect air loss.

Original Paper Description. Air loss from print orientation and printed pipe curvature Overall, more gradual printed curvatures led to better preservation of airflow in our widget designs, with losses ranging from 21.9% at 20 mm radius to 43.5% at 0 mm radius (see Figure 8). We saw no distinguishable pattern from printing angle (see Figure 9), in spite of having many datapoints; loss pattern were fairly consistent across all printings with the same orientation, but there was no progression tied to the specific angle. We suspect internal printing artifacts, e.g.,

how the layers of the printer met up with the internal geometry, affected jet formation at some orientations. The gates printed at 0, 22.5, and 90 degrees performed uniformly well (losing 27.0–30.5% of airflow on average).

We thus suggest including gradually-curving pipes where possible; more work is needed to understand the best printing orientation given devices' complex internal geometry.

Replication in FEDT. While this is written as a single experiment, we broke it into two functions for simplicity.

```
@fedt_experiment
def or_orientations():
    or_gate = GeometryFile('or.stl')

    fabbed_objects = []

    results = BatchMeasurements.empty()
    for print_angle in Parallel(arange(0,90+include_last,22.5)):
        rotated_gate = StlEditor.rotate(or_gate, print_angle)
        for repetition in Parallel(range(4)):
            fabbed_object = Printer.fab(rotated_gate, repetition=repetition)
            fabbed_objects.append(fabbed_object)
    for fabbed_object in Parallel(fabbed_objects):
        for input_massflow in Series(['5e^-5','9.5e^-5','14e^-5','18.5e^-5']):
            instruction(f"set the air compressor to {input_massflow} and connect the object")
            results += Anemometer.measure_airflow(fabbed_object, f"input massflow at {input_massflow}")

    analyze(results.get_all_data())

@fedt_experiment
def or_bendradius():
    or_gate = GeometryFile('or.stl')

    results = BatchMeasurements.empty()
    for bend_radius in Parallel(arange(0,20+include_last,5)):
        gate_with_bend = StlEditor.modify_design(or_gate, "bent inlet", bend_radius)
        fabbed_object = Printer.fab(gate_with_bend)
        for input_massflow in Series(['5e^-5','9.5e^-5','14e^-5','18.5e^-5']):
                instruction(f"set the air compressor to {input_massflow} and connect the object")
                results += Anemometer.measure_airflow(fabbed_object, f"input massflow at {input_massflow}")

    analyze(results.get_all_data())
```

Corrected Experiment. No corrections were required.

A.2.3 Experiment: Optimal Output Pressures. This experiment prints several output widgets and determines the minimum and maximum airflow that activates them.

Original Paper Description. Optimal output pressure Most of our output widgets operate best when actuated with pressures from 5+ kPa. They can likely be tuned for particular pressure systems (e.g., by adjusting counterweights or pressure-exposed surface area and shape), but in the particular configurations we tested we found the pin display works at 13.5+ kPa, the vibration motor works from 13.5–35 kPa, the whistle works from 5–7 kPa, and the wiggler wiggles at .3 kPa, working as a permanently-activated pin display above that.

Given the results of our previous investigations, designers can calculate the amount of input pressure required in their designs in order to optimally actuate their desired outputs.

Replication in FEDT.

```
@fedt_experiment
def output_airneeds():
    output_widgets = ['pin_display.stl','vibration_motor.stl', 'whistle.stl','wiggler.stl']

    epsilon = '.1kPa' # ?

    airflow = BatchMeasurements.empty()
```

```
    for widget in Parallel(output_widgets):
        fabbed_object = Printer.fab(GeometryFile(widget))
        instruction(f"connect object #{fabbed_object.uid} to the air compressor", header=True)
        instruction(f"increase the air pressure by {epsilon} at a time until the object starts to work")
        airflow += Anemometer.measure_airflow(fabbed_object, 'minimum working pressure')
        instruction(f"increase the air pressure by {epsilon} at a time until the object stops working")
        airflow += Anemometer.measure_airflow(fabbed_object, 'maximum working pressure')

    analyze(airflow.get_all_data())
```

Corrected Experiment. No corrections were required.

A.3 Paper 3 : BlowFab

Adapted from Yamaoka, et al., 2017 [85].

A.3.1 Experiment: Effect of B in Horizontal Bend. This experiment changes a parameter of the SVG file and determines the effect on bending after inflation.

Original Paper Description. Horizontal Bending Mechanism The bending direction is vertical and horizontal to the sheet. The user can generate intricate shapes by combining these methods. It is necessary to make a cut in the side of the sheet, as shown in Figure 13, to bend the object horizontally (i.e., along the plane of the sheet). The curvature of the object was investigated by changing the ratio of A to C in Figure 13. First, C was fixed at 10 mm and D was fixed at 14 mm, and the number of incisions made was 6. The objects were inflated by changing the ratio of B in the range of 0.09 to 0.66 (this means 4 mm to 27.5 mm) when A was 1 (41 mm). We attempted to bend 5 times on each object. As a result, as shown in Figure 14 a and c, the average angle of curvature was changed from 5.4°(SD = 0.8) to 38.2°(SD = 0.9).

Replication in FEDT.

```
@fedt_experiment
def effect_of_b_horizontal():

    A = 41
    C = 10
    D = 14

    delta = (27.5-4)/3 # ? -> there are 3 values in the figure

    results = BatchMeasurements.empty()
    for B in Parallel(arange(4, 27.5+include_last, delta)):
        linefile = SvgEditor.design(vars = {'A':A, 'B':B, 'C':C, 'D':D})
        fabbed_object = Laser.fab(linefile)
        for repetition in Series(range(5)):
            fabbed_object = Human.post_process(fabbed_object, "heat with the heat gun")
            # ? -> not sure if it is heat gun or plate
            fabbed_object = Human.post_process(fabbed_object, "inject air to inflate the object")
            # ? -> probably human does this, based on the video
            results += Protractor.measure_angle(fabbed_object,"overall bend")
            fabbed_object = Human.post_process(fabbed_object, "heat with the heat gun")
            # ? -> not sure if it is heat gun or plate
            fabbed_object = Human.post_process(fabbed_object, "flatten the object")
            # ? -> it's not specified that they are flattened, but they must be

    analyze(results.get_all_data())
```

Corrected Experiment. The number of objects fabricated was different from our guess based on the figures. Additionally, we have clarified which tools were used to heat and inflate the devices, and that bend angle was extracted from photographs instead of directly from the devices.

```
@fedt_experiment
def effect_of_b_horizontal():

    A = 41
```

```
    C = 10
    D = 14

    delta = (27.5-4)/5 # only 3 shown in image, but 5 were tested

    results = BatchMeasurements.empty()
    for B in Parallel(arange(4, 27.5+include_last, delta)):
        linefile = SvgEditor.design(vars = {'A':A, 'B':B, 'C':C, 'D':D})
        fabbed_object = Laser.fab(linefile, material="PET")
        for repetition in Series(range(5)):
            fabbed_object = Human.post_process(fabbed_object, "heat with the heat plate")
            fabbed_object = Human.post_process(fabbed_object,
                                               "inject air with the air compressor to inflate the object")
            instruction("slowly turn the valve on the air compressor to inflate just to the point of fullness")
            results += Camera.take_picture(fabbed_object, "overall bend")
            instruction("extract the bend angle from the photograph")
            fabbed_object = Human.post_process(fabbed_object, "heat with the heat plate")
            fabbed_object = Human.post_process(fabbed_object, "flatten the object")

    analyze(results.get_all_data())
```

A.3.2 Experiment: Effect of Incision Number. This experiment edited the SVG files to have additional length and incisions, to understand how this might affect bend angle

Original Paper Description. Moreover, when A is 41 mm, B is 14 mm and the number of incisions is 6, the object bent 22.1°on an average (SD = 0.8). When we changed the number of incisions to 12, the object bent 70.6°on an average (SD = 1.6).

From this experiment, the ratio of incisions and the curvature of the object can be described by the following equation (equation 1). In the equation, $\theta 1$ is the bent angle of the final product of horizontal bending, A is the length (mm) as shown in Figure 13A, B is the length (mm) in Figure 13B, and i means the number of incisions.

$$\theta 1 = (((A/B) - .75l)/-0.017) + 8.08i \tag{1}$$

Replication in FEDT.

```
@fedt_experiment
def effect_of_incision_number():
    A = 41
    B = 14
    C = 10
    D = 14

    results = BatchMeasurements.empty()
    for num_incisions in Parallel([6,12]):
        linefile = SvgEditor.design(vars = {'A':A, 'B':B, 'C':C, 'D':D, 'num_incisions':num_incisions})
        fabbed_object = Laser.fab(linefile)
        fabbed_object = Human.post_process(fabbed_object, "heat with the heat gun")
        # ? -> not sure if it is heat gun or plate
        fabbed_object = Human.post_process(fabbed_object, "inject air to inflate the object")
        # ? -> probably human does this, based on the video
        results += Protractor.measure_angle(fabbed_object,"overall bend")

    analyze(results.get_all_data())
```

Corrected Experiment. We confirmed which equipment was used to heat the objects and inject air, and that the measurements were taken from photographs instead of directly from the objects.

```
@fedt_experiment
def effect_of_incision_number():
    A = 41
    B = 14
```

```
C = 10
D = 14

results = BatchMeasurements.empty()
for num_incisions in Parallel([6,12]):
    linefile = SvgEditor.design(vars = {'A':A, 'B':B, 'C':C, 'D':D, 'num_incisions':num_incisions})
    fabbed_object = Laser.fab(linefile, material="PET")
    fabbed_object = Human.post_process(fabbed_object, "heat with the heat plate")
    fabbed_object = Human.post_process(fabbed_object, "inject air with the air compressor to inflate the object")
    instruction("slowly turn the valve on the air compressor to inflate just to the point of fullness")
    results += Camera.take_picture(fabbed_object, "overall bend")
    instruction("extract the bend angle from the photograph")

analyze(results.get_all_data())
```

A.3.3 Experiment: Interplate Distance for Vertical Bend. This experiment explored adding rigid plate materials to inflating airbags to control bend angle.

Original Paper Description. Vertical Bending Mechanism The user can bend an object automatically in a vertical direction (i.e., angular to the plane of the sheet) by overlapping materials that have different softening points. In this experiment, polycarbonate was selected instead of a kapton film. The softening point of polycarbonate is 120-140 °C. Therefore, when the sheet is heated to a temperature less than 120 °C, only the PET layer becomes soft enough to bend. For this method, polycarbonate plates are placed as shown in Figure 16. The user makes dents on the polycarbonate using a laser cutter. It is assumed that the two sheets bond mechanically when the PET material softens and enters into the dents on the surface of the polycarbonate. The user designs an area of polycarbonate at both ends of an airbag.

The user can specify the bending direction depending on the placement of the polycarbonate: it is either on the front side or on the backside. As shown in Figure 17, the curvature of objects was examined by changing the distance between the two polycarbonates. The size of the molded object is 100 mm × 40 mm, the size of the airbag is 45 mm × 28 mm and the length of the polycarbonate is 8 mm. As a result, when the distance (Figure 16 D) between two sheets was changed from 0 mm to 30 mm, the average angle changed from 100.3°(SD = 1.07) to 140.4°(SD = 1.01). We repeated the bending procedure 5 times on each object. From the results of this experiment, the distance between the polycarbonates and the curvature of the object can be described by the following equation (equation 2). $\theta 2$ is also the bent angle of the final product of vertical bending and D is the length (mm) between the layers as shown in Figure 16.

$$\theta 2 = (D - 105.02)/-0.948 \tag{2}$$

Replication in FEDT.

```
@fedt_experiment
def effect_of_interplate_distance_vertical():
    outer_size = (100, 40)
    airbag_size = (45, 28)

    results = BatchMeasurements.empty()
    for distance in Parallel([0,30]):
        linefile = SvgEditor.design(vars =
           {'outer size': outer_size, 'airbag size': airbag_size, 'distance': distance})
        instruction(f"place polycarbonate in the laser bed with spacing {distance}")
        fabbed_object = Laser.fab(linefile)
        for repetition in Parallel(range(5)):
            fabbed_object = Human.post_process(fabbed_object, "heat with the heat gun")
            # ? -> not sure if it is heat gun or plate
            fabbed_object = Human.post_process(fabbed_object, "inject air to inflate the object")
            # ? -> probably human does this, based on the video
            results += Protractor.measure_angle(fabbed_object,"overall bend")
            fabbed_object = Human.post_process(fabbed_object, "heat with the heat gun")
            # ? -> not sure if it is heat gun or plate
            fabbed_object = Human.post_process(fabbed_object, "flatten the object")
            # ? -> it's not specified that they are flattened, but they must be
    analyze(results.get_all_data())
```

Corrected Experiment. We confirmed which tools were used for heating and inflation, and also that bend angle was again extracted from a photograph. The authors also informed us that there were multiple copies of the objects manufactured.

```
@fedt_experiment
def effect_of_interplate_distance_vertical():
    outer_size = (100, 40)
    airbag_size = (45, 28)

    results = BatchMeasurements.empty()
    for distance in Parallel([0,30]):
        for repetition in Parallel(range(2)):
            linefile = SvgEditor.design(vars =
                    {'outer size': outer_size, 'airbag size': airbag_size, 'distance': distance})
            instruction(f"place polycarbonate in the laser bed with spacing {distance}")
            fabbed_object = Laser.fab(linefile, material="PET")
            for repetition in Parallel(range(5)):
                fabbed_object = Human.post_process(fabbed_object, "heat with the heat plate")
                fabbed_object = Human.post_process(fabbed_object,
                                                   "inject air with the air compressor to inflate the object")
                instruction("slowly turn the valve on the air compressor to inflate just to the point of fullness")
                results += Camera.take_picture(fabbed_object, "overall bend")
                instruction("extract the bend angle from the photograph")
                fabbed_object = Human.post_process(fabbed_object, "heat with the heat plate")
                fabbed_object = Human.post_process(fabbed_object, "flatten the object")
    analyze(results.get_all_data())
```

A.4 Paper 4 : CircWood

Adapted from Ishii, et al., 2022 [39].

A.4.1 Experiment: Materials. This experiment examined whether or not different materials and coatings could produce carbonized paths in the lasercutter.

Original Paper Description. Materials To identify suitable wood materials on which carbon paths can be created with sufficient low resistance to be used as electrical wiring, we tested various types of wood. We tested several types of lauan plywood and various solid woods (lauan, Japanese cypress, paulownia, Magnolia obovata, Japanese cedar, basswood, beech, oak, and walnut). We also examined the effect of coating the wood surface with a fire retardant that prevents burning and generates carbon. Ingredients of the fire retardant are ammonium polyphosphate and ammonium sulfate. These are food additives and harmless both to the environment and to humans.

Lauan plywoods and solid woods are suitable materials for our method because the carbon paths formed on these materials have a high conductivity. We can create carbon paths with usable conductivity on Japanese cypress solid woods by coating them with a fire retardant. The use of a fire retardant also improves the conductivity of carbon paths on lauan solid wood. The application of a fire retardant to plywood, on the other hand, reduces the conductivity of carbon paths. We assume this is due to a reaction that is caused by the glue between the wood layers. As a result, the optimal materials are lauan solid woods coated by the fire retardant, followed by bare lauan plywoods and fire-retardant coated solid woods of Japanese cypress. Since these materials differ in texture and various other characteristics, we can select suitable materials for the project.

Figure 5 shows some examples of carbon paths that were produced on the tested wood surfaces. The conductive carbon paths are entirely black and filled with charcoal grains, as shown in Figure 5 (a). In contrast, the non-conductive paths are not as black, and there are spaces between the charcoal grains, as shown in Figure 5 (b). These non-conductive paths were either burned at high temperatures or were the result of failure to create carbon at low temperatures, which hindered the production of sufficient conductive carbon. Non-conductive carbon paths tend to be generated from hardwoods, such as beech, oak, and walnut, and very softwoods, such as cedar. In contrast, usable conductive carbon tends to be generated from woods of moderate hardness, such as lauan and Japanese cypress.

Replication in FEDT.

```
@fedt_experiment
def test_materials():
    materials = ["lauan solid wood", "lauan plywood", "Japanese cypress",
                         "paulownia", "Magnolia obovata", "Japanese cedar", "basswood",
                         "beech", "oak", "walnut"]
    coatings = ['fire retardant', 'no coating']
```

```
    line_file = SvgEditor.build_geometry(SvgEditor.draw_circle)

    fabbed_objects: list[RealWorldObject] = []
    for material in Parallel(materials):
        for coating in Parallel(coatings):
            instruction(f"get a piece of {material} with coating {coating}")
            fabbed_objects.append(Laser.fab(line_file, material=material, coating=coating))
    results = BatchMeasurements.empty()
    for fabbed_object in Parallel(fabbed_objects):
        results += Multimeter.measure_resistance(fabbed_object)
    data = results.get_all_data()
    return analyze(data)
```

Corrected Experiment. No corrections were required.

A.4.2 Experiment: Laser Offset. This experiment looked at laser focal point offsets and number of scans to minimize resistance.

Original Paper Description. Laser parameters We investigated the laser cutter's setting parameters for efficient carbon production using lauan plywood of 4 mm thickness. The thicknesses of the surface and the second layer of the plywood were approximately 0.5 mm and 3 mm, respectively. The laser-induced carbon was generated mainly on the surface and the second layer. This material generated the highest conductivity carbon among the tested lauan plywoods. No fire retardant was applied in this examination because it would impair the conductivity of the plywood, as discussed in the previous section. We used VD7050-60W (COMMAX Co., Ltd.) laser cutter as described in Section 3.2.

First, we investigated the optimal wooden board position and found that the resistance of the generated carbon path was lowest at a distance of 5 mm below the focal point. We also found that repeating the laser scan at high speed yielded better results than running the laser once at low speed. When the laser cutter was operated at a low speed, adjusting the laser power became difficult, namely, either higher power would burn the wood, or lower power would not generate any carbon.

Replication in FEDT.

```
@fedt_experiment
def test_height_vs_focal_point():
    line_file = SvgEditor.build_geometry(SvgEditor.draw_circle)
    results = BatchMeasurements.empty()

    for focal_height_mm in Parallel(arange(0, 5+include_last)):
        fabbed_object = Laser.fab(line_file, focal_height_mm=focal_height_mm, material='wood')
        results += Multimeter.measure_resistance(fabbed_object)
    data = results.get_all_data()
    return analyze(data)
```

Corrected Experiment. We added a 6mm offset condition that was tested.

```
@fedt_experiment
def test_height_vs_focal_point():

    results = BatchMeasurements.empty()

    for focal_height_mm in Parallel(arange(0, 6+include_last)):
        line_file = SvgEditor.build_geometry(SvgEditor.draw_line,
                                             label_function=SvgEditor.labelcentre)
        fabbed_object = Laser.fab(line_file, focal_height_mm=focal_height_mm, material='wood')
        results += Multimeter.measure_resistance(fabbed_object)
    data = results.get_all_data()
    return analyze(data)
```

A.4.3 Experiment: Optimal Number of Scans. This experiment repeatedly scanned a laser over a fabricated object to find the best number of scans for carbon generation.

Original Paper Text. We confirmed the optimal number of scans by repeating the laser scans consecutively. The resistance continued to decrease until the fifteenth scan, but at the sixteenth scan, the carbon was burned off, and the resistance increased.

Replication in FEDT.

```
@fedt_experiment
def test_optimal_number_of_scans():
    line_file = SvgEditor.build_geometry(SvgEditor.draw_circle)
    results = ImmediateMeasurements.empty()
    resistance = 9999999999
    best_result = resistance
    num_scans = 0
    while resistance == best_result:
        num_scans = num_scans + 1
        fabbed_object = Laser.fab(line_file, num_scans=num_scans, material='wood')
        instruction("leave the material in the bed in between")
        resistance = results.do_measure(fabbed_object, Multimeter.resistance)
        resistance = float(resistance) if resistance else best_result
        if not best_result or resistance < best_result:
            best_result = resistance

    data = results.dump_to_csv()
    return analyze(data)
```

Corrected Experiment. No corrections were required.

A.4.4 Experiment: Laser Power and Speed. This experiment explored pairings of power and speed to find the easiest combination for generating carbon on a particular material.

Original Paper Description. By fixing the material position and the number of laser scans to the optimal conditions (5 mm below the focus and 15 scans), we investigated the generated carbon resistance by changing the laser power and the scan speed. We made four carbon paths for the same power/speed and measured the resistance of each of them. Table 2 shows the average resistance value for each power/speed setting value. Overall, the resistance was the lowest when the power was 50–70 % and the ratio of power to speed was 2:1. The lowest resistance we measured was 30 Ω/sq, which was realized when the power was 50 % (30 W), and the speed was 25 % (381 mm/s). We identified these setting values as the optimal parameters and used them when implementing the applications.

When the wood type was changed, the parameters also changed slightly. For the thicker lauan plywood with three 1 mm and two 3 mm thick sheets stacked alternately, and for lauan solid wood, the optimal parameters of power and speed were the same, but the optimal focal distance was 6 mm below the focal point. The lowest resistance value of the carbon on this type of lauan plywood panel was 75 Ω/sq. The resistance of the lauan solid wood was further reduced when a fire retardant was applied, with a minimum value of 25 Ω/sq. Japanese cypress coated with the fire retardant burned when the same parameters as for the lauan wood were used; hence, we further investigated the optimal parameters. The results demonstrate that the optimal focal distance was 5 mm below the focal point and that running the laser eight times at 30 % power (18 W) and 30 % speed (457 mm/s) was optimal. The lowest resistance value in this scenario was 76 Ω/sq. Based on the investigation above, Table 3 shows the final recommended settings of our process.

Replication in FEDT.

```
@fedt_experiment
def test_laser_power_and_speed():
    speeds = arange(20,80+include_last,10)
    powers = arange(10,50+include_last,5)
    config_file = Laser.create_config(cut_speeds=speeds, cut_powers=powers)

    line_file = SvgEditor.build_geometry(SvgEditor.draw_circle)
    results = BatchMeasurements.empty()

    for cut_speed in Parallel(speeds):
      for cut_power in Parallel(powers):
          for repetition in Parallel(range(4)):
            fabbed_object = Laser.fab(line_file, config_file=config_file,
                                      cut_speed=cut_speed, cut_power=cut_power,
                                      color_to_setting=Laser.SvgColor.GREEN,
                                      repetition=repetition,
                                      material='wood')
```

```
            resistance = Multimeter.measure_resistance(fabbed_object)
            results += resistance
    analyze(results.get_all_data())
```

Corrected Experiment. We pared down the tested conditions to those represented in the table; the authors confirmed that only these "likely" settings were tried, rather than all settings.

```
@fedt_experiment
def test_laser_power_and_speed():
    speeds_powers = [(50,50),
                     (45,50),(45,60),(45,70),(45,80),
                     (40,50),(40,60),(40,70),(40,80),
                     (35,50),(35,60),(35,70),(35,80),
                     (30,50),(30,60),(30,70),
                     (25,50),(25,60),
                     (20,40),
                     (15,40),
                     (10,20),(10,30)]
    config_file = Laser.create_config(cut_speeds=[speed for speed,power in speeds_powers],
                                      cut_powers=[power for speed,power in speeds_powers])

    line_file = SvgEditor.build_geometry(SvgEditor.draw_line)
    results = BatchMeasurements.empty()

    for cut_speed, cut_power in Parallel(speeds_powers):
        for repetition in Parallel(range(4)):
            fabbed_object = Laser.fab(line_file, config_file=config_file,
                                      cut_speed=cut_speed, cut_power=cut_power,
                                      color_to_setting=Laser.SvgColor.GREEN,
                                      repetition=repetition,
                                      material='wood')
            resistance = Multimeter.measure_resistance(fabbed_object)
            results += resistance
    analyze(results.get_all_data())
```

A.4.5 Experiment: Grain Direction. This experiment looked at carbon generation across versus along the grain of the plywood.

Original Paper Description. Effect of the grain direction The previous experiments were conducted by producing carbon paths along the grain of the wood. In this experiment, we used thicker lauan plywood and fabricated a carbon path orthogonal to the wood grain using the optimal parameters obtained in the previous section. The result showed a significant increase in resistance compared to the carbon made along the grain (27300 Ω/sq). This problem can be solved by arranging the carbon paths side by side to increase the area. We demonstrated that the carbon paths conduct when wide/multiple paths are produced, namely, the conductivity of three side-by-side paths is almost equivalent to the conductivity (105 Ω/sq) under the along-the-grain condition.

Replication in FEDT.

```
@fedt_experiment
def test_grain_direction():
    line_file = SvgEditor.build_geometry(SvgEditor.draw_circle)
    fabbed_objects: list[RealWorldObject] = []
    for orientation in Parallel(["orthogonal","along grain"]):
        instruction("orient the wood {}".format(orientation))
        fabbed_objects.append(
            Laser.fab(line_file, orientation=orientation))
    results = BatchMeasurements.empty()
    for fabbed_object in Parallel(fabbed_objects):
        results += Multimeter.measure_resistance(fabbed_object)
    data = results.get_all_data()
    return analyze(data)
```

Corrected Experiment. The authors told us that, instead of rotating the piece of wood in the bed, they instead used a file with two orthogonal lines.

```
@fedt_experiment
def test_grain_direction():
    line_file = SvgEditor.design("a file with two perpendicular lines")
    fabbed_objects: list[RealWorldObject] = []
    for orientation in Parallel(["orthogonal","along grain"]):
        instruction("orient the wood {}".format(orientation))
        fabbed_objects.append(
            Laser.fab(line_file, orientation=orientation))
    results = BatchMeasurements.empty()
    for fabbed_object in Parallel(fabbed_objects):
        results += Multimeter.measure_resistance(fabbed_object)
    data = results.get_all_data()
    return analyze(data)
```

A.4.6 Experiment: Change Over Time. This experiment looked at changes in resistance over time.

Original Paper Description. Resistance change over time We investigated the resistance change of the carbon path that was fabricated using CircWood over time. The resistance change after coating with varnish was also investigated because we assumed that the varnish, which is frequently used to protect surfaces in woodworking DIY, would prevent the resistance of the carbon path from increasing. We used oil-based varnish spray. We created four carbon paths for each of the two conditions, varnished or unvarnished, on lauan plywood with a surface thickness of 0.5 mm and an inner pile thickness of 3 mm. Laser parameters used to create the carbon were optimal values described in Section 5.1.2. We then measured the resistance of each sample over six months.

Figure 6 shows the transition of the average resistance increase rate. After six months, the resistance of the unvarnished samples increased by 1.45 times, while the varnished samples increased by 1.3 times. Judging from the graph, the resistance value seems to be stable around after three months. Although the coating had no significant effect on the resistance value when the carbon paths were first created, the graph shows that varnish coating reduces the increase in resistance in the long term. Moreover, the coating is useful as it increases the physical strength of the circuit and prevents finger contamination by the carbonized touch sensor.

Replication in FEDT.

```
@fedt_experiment
def test_change_over_time():
    line_file = SvgEditor.build_geometry(SvgEditor.draw_circle)
    fabbed_objects = []
    for post_process_condition in Parallel(['varnished','unvarnished']):
        for repetition in Parallel(range(4)):
            fabbed_object = Laser.fab(line_file, repetition=repetition)
            fabbed_objects.append(Human.post_process(fabbed_object, post_process_condition))
    results = BatchMeasurements.empty()
    for wait_months in Series(range(0, 6)):
        fabbed_objects = Environment.wait_up_to_time_multiple(fabbed_objects, num_months=wait_months)
        for fabbed_object in Parallel(fabbed_objects):
            results += Multimeter.measure_resistance(fabbed_object)
    analyze(results.get_all_data())
```

Corrected Experiment. No corrections were required.

A.5 Paper 5 : Design, Mould, Grow!

Adapted from Gough, et al., 2023 [31]. We ignore an experiment reported in Section 3 relating to mushroom species and substrates, as there was no digital fabrication component involved.

A.5.1 Experiment: Geometric Features. This experiment printed several trays and moulded mycomaterial from them to determine how well different geometric features could be preserved.

Original Paper Description. To evaluate how different geometric features are translated into the grown object, we designed three 3D models, first with a variety of features, the ramps and patterns trays shown in Figure 3 A and B respectively, and a tray with circular step features of varying heights shown in Figure 3 C. Each tray design was used to grow two 3D models, and with two different conditions (with and without inclusions) as shown in Figure 3-Table. Grown models were scanned and cross-sections of the scans were compared to the

original design. Figure 4 A to D shows a circular tray with step structure of varying heights (1mm to 5mm) as the design, 3D print, a grown model and a scanned model. Figures 3 A and B further show the details of the 3D printed trays with different geometric features such as spherical, corners, ramps, repeating lines and different kinds of textures. The radius of the spherical geometries ranged from 2mm to 7mm and the smallest cuboid features were 5 × 5mm. The features we compare included convex and concave shapes. For each tray two variants were grown, first using only the substrate in the grow kit and another that includes waste coffee grounds at a ratio of 10:3 by weight. From the circular design, an third object was grown and treated with glycerine, which has been shown to alter the properties of the material by retaining moisture [15].

Geometric Properties

To further analyse geometric changes after the drying process, samples were 3D scanned using a Schining EinScan Pro 2X with models were captured using EXScan Pro software (Figure 4 D). We extracted a series of profiles (cross-sections) from each 3D scan and quantitatively compared them to the corresponding profile of the original design. For the circular tray, 11 slices of profiles were extracted at rotational displacement of 7.5° as shown in Figure 4 E. For the patterns tray, 16 slices of profiles were extracted at linear displacement of 0.1mm between each other showing concave and convex patterns visible in Figure 4 F.

For the circular tray, the 3D scans of models with different treatments were separately analysed. The three materials samples are named for the substrate materials: Plain Substrate (PS), Coffee-inclusion Substrate (CS), and the Glycerine-plain Substrate (GS). For the patterns tray, only the outline of the sample using coffee inclusions was analysed for Concave and Convex (CC) sections, as it showed noticeably superior performance to the sample with no coffee inclusion.

The evaluation was performed by overlapping the profiles of final samples over the original 3D design. Along the outline of the grown material, the absolute error with respect to the original design (in mm) is averaged along evenly spaced intervals across the cross-section. Fig 4 G, H, and I show the profiles of cross-sections with CS, GS and CC conditions respectively. The outlines in each comparison show the mean pattern generated by all the profiles (purple), and the corresponding 95% confidence interval at each point (dashed line). These outlines indicate the ability of each material to replicate the intended pattern. The average mean absolute error (MAE) is shown in Figure 4 J. It is evident that including coffee grounds improves the ability of the material to replicate sharp edges and flat surfaces. We further compared the MAE on different curvature radii and types (convex, protruding out vs. concave, denting in) along the condition CC. CX represents convex and CV1 to CV3 represents three concave radii 7mm, 5mm, and 3mm. Results show that convex features have higher conformity with the original design than the concave, and we recommend a minimum feature size of 5mm (Fig 4 K).

Changes in Dimension

To measure changes in volume during the drying process we compared the final dimensions of the dried myco-material to the dimensions of the 3D printed tray. For the ramps tray in Figure 3 A and its outcome Fig. 5 A Left, the tray was measured to be 128 × 127.5mm, and the myco-material showed a reduction of 93.0% × 92.5% and 92.2% × 91.8% for no inclusion and coffee inclusion respectively. The patterns tray shown in Figure 3 B (outcome in Figure 5 A Right) was measured to be 128 × 123mm, with the myco-material showing a reduction in volume of 92.2% × 91.9% and 90.6% × 91.1% for PS and CS respectively. The circular tray shown in Figure 3 C (outcome Fig. 4 C and E) has a radius of 118mm. All three myco-material samples were measured to be 109 × 109mm, a 92.4% reduction in length along the two straight edges.

Qualitative Observations

Two grown designs are shown in Figure 5 A. We observed slight warping occurred during the drying phase. Previous studies have compared the results when the material is pressed as it is drying [2], which would reduce warping. However, as the object shrinks during drying, this would put strain on some fine details on the material if it were to be dried inside the 3D printed mould, which may also start to soften at drying temperatures, depending on the filament used. It's also worth noting that the 3D prints itself had some geometric imperfections, which occurred during printing that were rendered in the material.

Samples with coffee ground inclusions showed significantly more detail in the final product. The walls between the square indentations, measuring 1.33mm width, can be clearly seen in Figure 5 A-I, but are not visible in the sample grown without the coffee ground inclusion. Additionally, the layers from the 3D prints are clearly visible in both types (layer height was set to 0.30mm during slicing), but more pronounced in the form with inclusions as seen in Figure 5 A-II. The Patterns sample in Figure 5 A-III & -IV shows larger protrusions being rendered clearly by the material, as well as surface textures. Fine details were possible, but can be difficult to remove from a 3D printed mould as mycelia may find its way between the layers of the mould. It may be possible to use other methods to improve fine details by using a higher resolution 3D print technology (e.g. SLA resin). We also noted that if pieces of the designed object are stuck or broken during the demoulding stage, they can be repaired by simply putting them back in place, and the mycelial network will fuse the pieces together during the open growth stage. We did not evaluate the strength of this bond.

Replication in FEDT.

```
class CustomModellingTool(DesignSoftware):
    # call the custom modelling tool that they created
    @staticmethod
    def sphere(radius: float=10.) -> GeometryFile:
        return design('custom.stl', GeometryFile,
                      {'target_radius':radius},
                      f"use the custom tool to model a sphere with radius {radius}")
```

```
    @staticmethod
    def create_design(features: dict[str,object]) -> GeometryFile:
        return design('custom.stl', GeometryFile,
                      features,
                      f"use the custom tool to model {features}")

    @staticmethod
    def modify_design(design: GeometryFile,
                      feature_name: str, feature_value: str|int) -> GeometryFile:
        from control import MODE, Execute
        file_location = design.file_location
        if isinstance(MODE, Execute):
            file_location = input(f"What is the location of the modified geometry file?")

        design.updateVersion(feature_name, feature_value)
        design.file_location = file_location
        return design

def prep_materials():
    instruction("remove pathogens by pouring boiling water through strainer")
    instruction("bulk growth - 1 week")
    instruction("break up for forming growth - 3 days")

@fedt_experiment
def geometric_features():
    shrinkage_results = BatchMeasurements.empty()
    scanning_results = BatchMeasurements.empty()

    geometries = [GeometryFile(f) for f in ['ramps.stl', 'circular.stl', 'patterns.stl']]

    prep_materials() # how was this timed? just constant prepping in rotation, or?

    for geometry_file in Parallel(geometries):
        mould = Printer.fab(geometry_file)
        for myco_material in Series(['30% coffee inclusions', 'no inclusions']):
            fabbed_object = Human.post_process(mould, f"mould mycomaterial {myco_material}")
            Environment.wait_up_to_time_single(fabbed_object, num_weeks=1)
            instruction("remove the material from the mould")
            Environment.wait_up_to_time_single(fabbed_object, num_days=3)
            instruction("allow the material to dry in a 50-80C oven")
            Environment.wait_up_to_time_single(fabbed_object, num_weeks=7) # 1 day to 1 week
            shrinkage_results += Calipers.measure_size(fabbed_object, "important dimension")
            if geometry_file.file_location != 'ramps.stl':
                scan = Scanner.scan(fabbed_object)
                scanning_results += scan
            if geometry_file.file_location == 'circular.stl':
                oneoff_object = Human.post_process(mould, f"mould mycomaterial {myco_material}")
                oneoff_object = Human.post_process(oneoff_object, 'glycerine treatment')
                shrinkage_results += Calipers.measure_size(oneoff_object, "important dimension")
                scanning_results += Scanner.scan(oneoff_object)

    # several derivative measurement operations are ignored here for simplicity;
    # e.g., the extraction of 2D profiles from the scans

    shrinkage_results = shrinkage_results.get_all_data()
    scanning_results = scanning_results.get_all_data()
```

```
    analyze(shrinkage_results)
    analyze(scanning_results)
```

Corrected Experiment. We had misunderstood that there are in fact several check loops at work in the myco-moulding process, and that the timeline presented in the paper is intended to be inspirational rather than precise (we had coded it as a for loop). We adjusted to have several while loops in a helper function for moulding. We also added additional sterilization instructions, in order to remain more true to what was performed by the author.

```
class CustomModellingTool(DesignSoftware):
    # call the custom modelling tool that they created
    @staticmethod
    def sphere(radius: float=10.) -> GeometryFile:
        return design('custom.stl', GeometryFile,
                      {'target_radius':radius},
                      f"use the custom tool to model a sphere with radius {radius}")

    @staticmethod
    def create_design(features: dict[str,object]) -> GeometryFile:
        return design('custom.stl', GeometryFile,
                      features,
                      f"use the custom tool to model {features}")

    @staticmethod
    def modify_design(design: GeometryFile,
                      feature_name: str, feature_value: str|int) -> GeometryFile:
        from control import MODE, Execute
        file_location = design.file_location
        if isinstance(MODE, Execute):
            file_location = input(f"What is the location of the modified geometry file?")

        design.updateVersion(feature_name, feature_value)
        design.file_location = file_location
        return design

def prep_materials():
    instruction("remove pathogens by pouring boiling water through materials in strainer")
    instruction("bulk growth - 1 week")
    instruction("break up for forming growth - 3 days")
    instruction("keep in the fridge as prepared material")
    instruction("remove from fridge to reactivate 1 day before beginning experiments")
    instruction("ensure you wear gloves")
    instruction("sterilize the prep space")

@fedt_experiment
def geometric_features():
    def grow_mycomaterial_from_mould(mould, myco_material):
        instruction("sterilize the prep area")
        instruction("sterlize the mould with isopropyl alcohol")
        fabbed_object = Human.post_process(mould, f"mould mycomaterial {myco_material}")
        is_reasonable = False
        while not is_reasonable:
            Environment.wait_up_to_time_single(fabbed_object, num_days=1)
            instruction("look at or poke the sample")
            fabbed_object = Human.is_reasonable(fabbed_object)
            is_reasonable = fabbed_object.metadata["human reasonableness check"]
        instruction("remove the material from the mould for open growth")
        is_reasonable = False
```

```
        while not is_reasonable:
            Environment.wait_up_to_time_single(fabbed_object, num_days=1)
            instruction("look at or poke the sample")
            fabbed_object = Human.is_reasonable(fabbed_object)
            is_reasonable = fabbed_object.metadata["human reasonableness check"]
        instruction("allow the material to dry in a 50-80C oven")
        is_reasonable = False
        while not is_reasonable:
            Environment.wait_up_to_time_single(fabbed_object, num_days=1)
            instruction("look at or poke the sample")
            fabbed_object = Human.is_reasonable(fabbed_object)
            is_reasonable = fabbed_object.metadata["human reasonableness check"]
        instruction("material is ready when it is dry")
        print(fabbed_object)
        return fabbed_object

    shrinkage_results = BatchMeasurements.empty()
    scanning_results = BatchMeasurements.empty()

    geometries = [VolumeFile(f) for f in ['ramps.stl', 'circular.stl', 'patterns.stl']]

    prep_materials()
    instruction("ensure you wear gloves")
    instruction("sterilize the prep space")

    for geometry_file in Parallel(geometries):
        mould = Printer.slice_and_print(geometry_file)
        for myco_material in Series(['30% coffee inclusions', 'no inclusions']):
            fabbed_object = grow_mycomaterial_from_mould(mould, myco_material)
            shrinkage_results += Calipers.measure_size(fabbed_object, "important dimension")
            if geometry_file.stl_location != 'ramps.stl':
                scan = Scanner.scan(fabbed_object)
                scanning_results += scan
            if geometry_file.stl_location == 'circular.stl':
                oneoff_object = grow_mycomaterial_from_mould(mould, myco_material)
                oneoff_object = Human.post_process(oneoff_object, 'glycerine treatment')
                shrinkage_results += Calipers.measure_size(oneoff_object, "important dimension")
                scanning_results += Scanner.scan(oneoff_object)

    # several derivative measurement operations are ignored here for simplicity;
    # e.g., extraction of 2D profiles from the scans

    shrinkage_results = shrinkage_results.get_all_data()
    scanning_results = scanning_results.get_all_data()

    analyze(shrinkage_results)
    analyze(scanning_results)
```

A.5.2 Experiment: Mechanical and Shrinkage Features. This experiment created a set of cubes and subjected them to tests to understand how they shrunk during drying and how they behaved under compression and release.

Original Paper Description. After observing the reduction of dimensions during the drying process, our next goal was to further evaluate if an object can be designed to achieve the desired output dimensions after drying. To test the effectiveness of dimension-corrected designs, we created a series of rectangular cuboids, as shown in Figure 3 D. The target design was a cuboid 30 × 60 × 16mm. To achieve this, the 3D printed model was scaled up by 1/0.92, based on the average amount of shrinkage (92.0%) measured in the tray designs. The growth stages were the same as for the trays to produce samples with and without coffee ground inclusions (also at 10:3 ratio of substrate to inclusion by weight), producing 8 cuboids with with inclusion, samples A-H, and 8 samples with without inclusion I-P, (see Figure 5 B). These samples were also used for load testing.

The final 3D model dimensions shown in Table 2 differed slightly from the 3D printed dimensions. The grown blocks were surprisingly consistent as shown in Figure 5 B. Samples A - H (Figure 5 B Top) included 30% coffee grounds by weight of the grow kit substrate used to fill the mould. These had consistent shrinkage across the samples, with the greatest variation being in height, with the final blocks being an average of 14.36mm tall, or 83.93% (σ = 2.65) of the target size. The width and length of the blocks were much closer to the target, with the average dimensions being 30.19 × 60.24mm (σ = 0.17 and 0.18). Samples I - P (Figure 5 B bottom) did not include any coffee ground with the substrate, and the change in size was also close to the target, at 29.84 × 59.05 × 15.57mm. Sample L was an outlier, at 57.63mm length, 89.1% of the mould size, while all other samples were between 91.0% and 92.2% of the mould size.

Surface Quality

Samples A - H had formed a white layer of mycelia around the outside of the sample, which is often seen in myco-material examples (see [15]), however Samples I - P had a darker appearance, and the shape of the underlying substrate is more easily visible. This is explained by natural variations of the fungus to the specific conditions of the two growth runs. Under mushroom lab conditions where we would be able to precisely control ambient humidity, temperature and CO2, this may be able to be controlled, but in our HCI lab, some variation was inevitable. It is also observable that samples I - P also had some evidence that they were starting to try to produce fruit, with small white points visible in samples I - P in Figure 5 B (Bottom). These points feel dense or hard when dried, whereas the overall white coating seen on samples A - D in Figure 5 B (Top) feels softer and similar to paper.

Mechanical Properties

We conducted a compression and relaxation study under an external load to test the elasticity of the material using the cuboids. The load testing was performed using a Starr FDG-1000 digital force gauge, loaded on a manual testing stand and digital distance reading. Samples were placed on a plastic block, and the myco-material blocks were pressed using a flat push attachment and the force from the material measured at 0.5mm increments to a maximum depth of 5mm. The reading was taken using a custom Python script. Compression and relaxation cycles were repeated 5 times on each cuboid.

Figure 6 A shows the results of the mechanical test. Subplots a) and b) show samples A - H (without inclusion), c) and d) show samples I - P (coffee ground inclusion). The two curves represent compression (loading, blue) and relaxation (unloading, orange), as a typical elastic material, requiring less force to achieve the same displacement when relaxing. Comparing Figure 6 A subplots a vs c and b vs d, we can see that the coffee inclusion increased the slope or the modulus of elasticity. Comparing Figure 6 A subplots a vs b and c vs d reveals that treating with glycerine has a negligible effect on the elastic properties of the material without coffee inclusion. However, the glycerine treated coffee mixture has better elasticity, since the compression and relaxation cycle almost overlap with low hysteresis loss in a complete cycle. This result indicates that careful selection and tuning of inclusion material can be used to manipulate mechanical properties. This may be relevant to a tangible application where the elastic modulus of the object needs to be adjusted, for example, in squeezable interfaces or pressure sensors.

Replication in FEDT.

```
@fedt_experiment
def mechanical_and_shrinkage_features():
    target_cube = StlEditor.cube((30,60,16))
    scaled_mould = StlEditor.cube((30, 60, 16), scale=1/.92)

    shrinkage_results = BatchMeasurements.empty()
    mechanical_results = BatchMeasurements.empty()
    mould = Printer.fab(scaled_mould)

    fabbed_objects = []

    prep_materials()

    for myco_material in Series(['30% coffee inclusions', 'no inclusions']):
        for repetition in Series(range(4)):
            fabbed_object = Human.post_process(mould, f"mould mycomaterial {myco_material} ({repetition}th copy)")
            Environment.wait_up_to_time_single(fabbed_object, num_weeks=1)
            instruction("remove the material from the mould")
            Environment.wait_up_to_time_single(fabbed_object, num_days=3)
            instruction("allow the material to dry in a 50-80C oven")
            Environment.wait_up_to_time_single(fabbed_object, num_weeks=7) # 1 day to 1 week
            instruction("remove the object from the mould")
            shrinkage_results += Calipers.measure_size(fabbed_object, "x-axis")
            shrinkage_results += Calipers.measure_size(fabbed_object, "y-axis")
            shrinkage_results += Calipers.measure_size(fabbed_object, "z-axis")
```

```
            fabbed_objects.append(fabbed_object)

    for fabbed_object in Parallel(fabbed_objects):
        for repetition_mechanical in Series(range(5)):
            for depth in Series(arange(0,5+include_last,.5)):
                instruction("ensure fabbed object is appropriately arranged on testing stand")
                fabbed_object = Human.post_process(fabbed_object,
                                f"compress the object to {str(depth)} ({repetition_mechanical}th time)")
                # not clear who/what does the compressing?
                mechanical_results += ForceGauge.measure_force(fabbed_object)

    analyze(shrinkage_results.get_all_data())
    analyze(mechanical_results.get_all_data())
```

Corrected Experiment. We corrected our model to have, instead of 4 repetitions of moulding from a one-cube mould, a single repetition of moulding on a 4-cube mould. we also confirmed that the researcher manually compressed the object, and added the release cycle measurements, which we had initially missed. We also, as above, added the more complex helper function to do moulding.

```
@fedt_experiment
def mechanical_and_shrinkage_features():
    def grow_mycomaterial_from_mould(mould, myco_material):
        # helper as above
        return fabricate({'mould':mould, 'mycomaterial':myco_material},
                         f'prepare {myco_material} and mould as before')

    target_cube = StlEditor.cube((30,60,16))
    scaled_mould = StlEditor.cube((30, 60, 16), scale=1/.92)
    scaled_mould_all = StlEditor.modify_design(scaled_mould, 'number of cubes in file', '4')

    shrinkage_results = BatchMeasurements.empty()
    mechanical_results = BatchMeasurements.empty()
    mould = Printer.fab(scaled_mould_all)

    fabbed_objects = []

    prep_materials()

    for myco_material in Series(['30% coffee inclusions', 'no inclusions']):
        fabbed_object = grow_mycomaterial_from_mould(mould, myco_material)
        shrinkage_results += Calipers.measure_size(fabbed_object, "x-axis")
        shrinkage_results += Calipers.measure_size(fabbed_object, "y-axis")
        shrinkage_results += Calipers.measure_size(fabbed_object, "z-axis")
        fabbed_objects.append(fabbed_object)

    for fabbed_object in Parallel(fabbed_objects):
        for repetition_mechanical in Series(range(5)):
            instruction("ensure fabbed object is appropriately arranged on testing stand")
            for depth in Series(arange(0,5+include_last,.5)):
                fabbed_object = Human.post_process(fabbed_object,
                                    f"compress the object to {str(depth)} ({repetition_mechanical}th time)")
                mechanical_results += ForceGauge.measure_force(fabbed_object)
            for depth in Series(arange(5,0-include_last,-.5)):
                fabbed_object = Human.post_process(fabbed_object,
                                    f"release the object to {str(depth)} ({repetition_mechanical}th time)")
                mechanical_results += ForceGauge.measure_force(fabbed_object)

    analyze(shrinkage_results.get_all_data())
    analyze(mechanical_results.get_all_data())
```

A.5.3 Experiment: Software Tool Testing. This experiment used a custom software tool to manufacture a series of objects and observe their shrinkage in different directions.

Original Paper Description. To evaluate the functionality of the tool, we used it to generate a series of moulds for a sphere with radius 20mm and followed the myco-material fabrication process to grow three myco-material spheres. The sphere was chosen because it would reduce in size uniformly in each direction and helps us evaluate performance from different angles. We measured each of the sphere across the diameter at six different locations (See Figure 8 E). The 18 measurements (M = 39.92mm, σ = 0.74) show that the final form was within 0.2% of the target size on average.

Replication in FEDT.

```
@fedt_experiment
def test_software_tool():
    target_sphere = StlEditor.sphere(radius=20.)
    software_generated_mould = CustomModellingTool.sphere(radius=20.)

    prep_materials()

    results = BatchMeasurements.empty()
    mould = Printer.fab(software_generated_mould)
    myco_material = "30% coffee inclusions"
    for repetition in Parallel(arange(1,3+include_last)):
        fabbed_object = Human.post_process(mould, f"mould mycomaterial {myco_material}")
        fabbed_object.metadata.update({'repetition':repetition})
        Environment.wait_up_to_time_single(fabbed_object, num_weeks=1)
        instruction("remove the material from the mould")
        Environment.wait_up_to_time_single(fabbed_object, num_days=3)
        instruction("allow the material to dry in a 50-80C oven")
        Environment.wait_up_to_time_single(fabbed_object, num_weeks=7) # 1 day to 1 week
        measurement_points = arange(0,90+include_last,45)
        for x_axis_point in Parallel(measurement_points):
            results += Calipers.measure_size(fabbed_object, f"{x_axis_point} degrees along the x axis")
        for y_axis_point in Parallel(measurement_points):
            results += Calipers.measure_size(fabbed_object, f"{y_axis_point} degrees along the y axis")
        for z_axis_point in Parallel(measurement_points):
            results += Calipers.measure_size(fabbed_object, f"{z_axis_point} degrees along the z axis")

    analyze(results.get_all_data())
```

Corrected Experiment. The only correction was, as above, adding the relatively more complex helper function describing growth.

```
@fedt_experiment
def test_software_tool():
    def grow_mycomaterial_from_mould(mould, myco_material):
        # helper as above
        return fabricate({'mould':mould, 'mycomaterial':myco_material},
                         f'prepare {myco_material} and mould as before')

    target_sphere = StlEditor.sphere(radius=20)
    software_generated_mould = CustomModellingTool.sphere(radius=20)

    prep_materials()

    results = BatchMeasurements.empty()
    mould = Printer.fab(software_generated_mould)
    myco_material = "30% coffee inclusions"
    for repetition in Parallel(arange(1,3+include_last)):
        fabbed_object = grow_mycomaterial_from_mould(mould, myco_material)
        fabbed_object.metadata.update({'repetition': repetition})
        measurement_points = arange(0,90+include_last,45)
```

```
        for x_axis_point in Parallel(measurement_points):
            results += Calipers.measure_size(fabbed_object, f"{x_axis_point} degrees along the x axis")
        for y_axis_point in Parallel(measurement_points):
            results += Calipers.measure_size(fabbed_object, f"{y_axis_point} degrees along the y axis")
        for z_axis_point in Parallel(measurement_points):
            results += Calipers.measure_size(fabbed_object, f"{z_axis_point} degrees along the z axis")

    analyze(results.get_all_data())
```

A.6 Paper 6 : ElectriPop

Adapted from Fang, et al., 2022 [25].

A.6.1 Experiment: Simulation Optimization. This experiment looked at different simulation parameters to best match captured ground-truths of fabricated and actuated objects.

Original Paper Description. Parameter Search and Optimization We initialized the mechanical parameters of our model using material specifications of mylar [53]. To further refne these parameters, we ran a parameter search using our simulation tool. More specifically, we physically inflated three designs — compound slits (Figure 8), nested flaps (Figure 11) and dragonfly (Figure 12) — and photographed their 3D forms. In 3D CAD software (Blender), we replicated their 3D shape to serve as a ground truth. While using a 3D scanning tool (e.g., laser scanning, photogrammetry) would have been faster, we did not find any that would robustly capture our thin and reflective mylar forms, and so hand posing was necessary.

We then ran our simulation tool over a grid of possible parameters (see previous section). In pilot tests, we found that mylar had essentially zero stretch (at least at the forces we consider), and so we dropped this from our search. This leaves two main parameters of interest: bending energy and electrostatic energy. For each pair of values, the simulation was run on an input SVG, with output compared to the previously prepared ground truth 3D model. We compare matched pairs of vertices to calculate the mean euclidean error. The results of our grid search can be seen in Figure 5. The simulation found only a slight modification of values was needed achieve an optimal matching between simulated and real-world results (darkest blue cell in the center of the grid; minimum error: 1.03%). We adopted these found parameters for all subsequent studies.

Replication in FEDT.

```
electripop_laser_defaults = {
    Laser.MATERIAL: 'mylar',
    Laser.CUT_SPEED: 100,
    Laser.CUT_POWER: 5
}

class ManualGeometryScanner:
    geometry_scan = Measurement(
        name="manual geometry scan",
        description="A full copy of the object's geometry, stored in a file.",
        procedure="""
            Take a photograph of the object from 3 angles,
            and extract the profiles from it into Blender.
            """,
        units="filename",
        feature="full object")

    @staticmethod
    def scan(obj: RealWorldObject) -> BatchMeasurements:
        instruction(f"Scan object #{obj.uid}.", header=True)
        instruction(ManualGeometryScanner.geometry_scan.procedure)
        return BatchMeasurements.single(obj, ManualGeometryScanner.geometry_scan)

class CustomSimulator:
    def runsimulation(*args, **kwargs):
        pass

@fedt_experiment
```

```
def optimize_simulation():

    test_files = [GeometryFile(fname) for fname in ['compound_slits.svg', 'nested_flaps.svg', 'dragonfly.svg']]

    ground_truths = BatchMeasurements.empty()
    simmed = BatchMeasurements.empty()

    for f in Parallel(test_files):
        fabbed_object = Laser.fab(f, default_settings=electripop_laser_defaults)
        # laser, vinyl cutter, scissors?
        ground_truths += ManualGeometryScanner.scan(fabbed_object)
        for weight_of_be_exp in Parallel(arange(-3,6+include_last)):
            for weight_of_ee_exp in Parallel(arange(-1, 1+include_last, .2)):
                sim = VirtualWorldObject('file.sim',
                                         {'weight of bending energy': math.pow(10, weight_of_be_exp),
                                             'weight of electrical energy': math.pow(10, weight_of_ee_exp),
                                             'file': CustomSimulator.runsimulation(f,
                                                                                   weight_of_be_exp, weight_of_ee_exp)})
                simmed += ManualGeometryScanner.scan(sim)

    analyze(ground_truths.get_all_data())
    analyze(simmed.get_all_data())
```

Corrected Experiment. No corrections were required.

A.6.2 Experiment: Electrical Inflation. This experiment examined how long it takes to electrically saturate the mylar.

Original Paper Description. In order to compare against other shape changing methods, as well as more deeply understand the physics of our technique, it was important to characterize ElectriPop's behavior. For these experiments, we used our transformer base (Figure 6), which has an output voltage of 9kV when powered at 12V.

Electrical inflation We measured the time and power required to flood the mylar film with charge (which is different from the time needed to physically inflate, which we describe next). Using our snowman design (Figure 3), we measured the amount of current required to electrically "inflate" our base and snowman design using a floating (non-grounded) multimeter with a known resistance of 9MΩ. The average steady-state current is V /R =11.5V/9MΩ=1.28muA, where V is the measured voltage, R is the resistance of the multimeter. Using a high voltage probe with a resistance of 1GΩ, we also measured the amount of current needed for the voltage converter, which is V /R =6.80kV/1GΩ=6.80μA, where V again is the measured voltage, R is the resistance of the high voltage probe. According to International Electrotechnical Commission's IEC 62368 standard, both current inputs are below the ES1 current limit [1]. In other words, the power source is not capable of supplying dangerous steady-state current, and the system does not require a significant amount of current to be inflated. In terms of duration, it takes on the order of 10 milliseconds to fully saturate the mylar with charge (it acts like a capacitor being charged with a current-limited source).

Replication in FEDT.

```
@fedt_experiment
def electrical_inflation():
    # it takes "on the order of 10ms to fully saturate the mylar with charge"
    # but unclear if this was tested on more than one object or more than one repetition
    pass
```

Corrected Experiment. The authors confirmed that they had done multiple repetitions with the snowman object and averaged the timings with the oscilloscope.

```
@fedt_experiment
def electrical_inflation():
    fabbed_object = Laser.fab(GeometryFile('snowman.svg'), default_settings=electripop_laser_defaults)

    elapsed_times = BatchMeasurements.empty()
    for repetition in Series(range(10)):
        instruction("attach the oscilloscope probe")
        instruction("monitor the oscilloscope to determine full saturation")
        elapsed_times += Stopwatch.measure_time(fabbed_object, f"inflate to full for {repetition}th time")
```

```
    analyze(elapsed_times.get_all_data())
```

A.6.3 Experiment: Physical Inflation. This experiment looked at how long it takes to physically inflate the mylar objects.

Original Paper Description. Physical Inflation As the mylar charges, it begins to repel and physically inflate. The repulsion force has to overcome gravity and bend mylar, which wishes to stay fat. Air resistance also works to retard physical inflation. On average, our snowman design takes around 100ms to fully inflate, and inflation times were remarkably consistent. In contrast, our largest and heaviest design, the Christmas tree, takes 750ms to inflate.

Replication in FEDT.

```
@fedt_experiment
def physical_inflation():
    fabbed_objects =
      [Laser.fab(GeometryFile(fname), default_settings=electripop_laser_defaults)
            for fname in ['snowman.svg','christmas_tree.svg']]

    elapsed_times = BatchMeasurements.empty()
    for obj in Parallel(fabbed_objects):
        # were there repetitions? were there other objects?
        # mention of "remarkably consistent", and implication that some others were tested
        elapsed_times += Stopwatch.measure_time(obj, "inflate to full")

    analyze(elapsed_times.get_all_data())
```

Corrected Experiment. The authors confirmed that there were repetitions on the inflations.

```
@fedt_experiment
def physical_inflation():
    fabbed_objects = [Laser.fab(GeometryFile(fname), default_settings=electripop_laser_defaults)
                                for fname in ['snowman.svg','christmas_tree.svg']]

    elapsed_times = BatchMeasurements.empty()
    for obj in Parallel(fabbed_objects):
        for repetition in Series(range(10)):
            elapsed_times += Stopwatch.measure_time(obj, f"inflate to full for {repetition}th time")

    analyze(elapsed_times.get_all_data())
```

A.6.4 Experiment: Electrical Deflation. This experiment measured how long it takes to deflate the mylar electrically when shunted.

Original Paper Description. Electrical Deflation We also measured how our mylar forms electrically deflate when grounded. For this, we used a 1kΩ shunt resistor to simulate the resistance of a finger. We used an oscilloscope probe attached to the resistor and recorded the voltage across the resistor to measure the discharge current (Figure 23, blue line) when the resistor contacts the system (in this case, an inflated snowman design), and we integrated the current to get the initial charge of the system (Figure 23, green line). The total discharge is 16.3nC. The time constant τ, which is the time it takes to reach 63% of steady-state is 39.5ns (Figure 23 red dash line). Since τ = R × C, where R and C are the resistance and capacitance of the system respectively, we can derive the capacitance of the system, which is 39.5pF. This is an insignificant amount of charge and the system's capacitance is extremely small compared to the IEC 62368 standard [1]. As discharge is not current limited (unlike inflation), electrical discharge times are very fast — 160ns on average. This also allowed us to estimate the initial voltage before discharge V =412V using V = Q/C, and the stored energy before discharge E =6.68μJ, using E = 0.5 × Q × V , where Q in both equations is the total discharge. There is also natural electrical deflation over time due to e.g., water molecules in air slowly bleeding charge. The rate of discharge depends on factors such as object surface area, ambient humidity and airflow. However, in general, the discharge rate is low and was not a significant hindrance in our designs. For example, without periodic recharging, our flower deflates in 25 minutes (at 22.3 °C, 19% relative humidity). Thus, to keep forms fully inflated, we suggest momentary charge boosts every five minutes or so.

Replication in FEDT.

```
@fedt_experiment
def electrical_deflation():
    snowman = Laser.fab(GeometryFile('snowman.svg'), default_settings=electripop_laser_defaults)

    current_measures = ImmediateMeasurements.empty()
```

```
    instruction('connect a 1kOhm resistor to the plate')
    current = 9999999999
    while current > 0:
        current = current_measures.do_measure(snowman, Multimeter.current)
        current = float(current) if current else 99999999
        current_measures += Timestamper.get_ts(snowman)

    analyze(current_measures.dump_to_csv())
```

Corrected Experiment. The authors confirmed the basic setup, but said that the speed of the loop didn't matter and they didn't take continuous measurements; the curves shown in their figures were theoretically derived rather than empirically derived.

```
@fedt_experiment
def electrical_deflation():
    snowman = Laser.fab(GeometryFile('snowman.svg'), default_settings=electripop_laser_defaults)

    current_measures = ImmediateMeasurements.empty()
    instruction('connect a 1kOhm resistor to the plate')
    current = 9999999999
    while current > 0:
        current = current_measures.do_measure(snowman, Multimeter.current)
        current = float(current) if current else 99999999
        current_measures += Timestamper.get_ts(snowman)
        # the speed of the loop isn't so important, because the final result was theoretically derived

    analyze(current_measures.dump_to_csv())
```

A.6.5 Experiment: Physical Deflation. This experiment looked at how long it takes for an object to physically deflate.

Original Paper Description. Physical Deflation Our snowman example takes 67ms to physically fall once static electricity is discharged and the repulsion force becomes absent. Note that this is faster than inflation, as deflation is assisted by gravity and stored spring forces that want to pull the mylar back to its original flat form. Working against deflation is air resistance, the magnitude of which is highly dependent on object geometry (i.e., in some designs, the form can act like a parachute, slowing deflation).

Replication in FEDT.

```
@fedt_experiment
def physical_deflation():
    # discuss whether this is an experiment, or a measurement/characterization
    snowman = Laser.fab(GeometryFile('snowman.svg'), default_settings=electripop_laser_defaults)
    elapsed_times = BatchMeasurements.empty()
    elapsed_times += Stopwatch.measure_time(snowman, "deflate fully")
    analyze(elapsed_times.get_all_data())
```

Corrected Experiment. The authors confirmed that the method was correct, but described that they did not view this as an experiment: rather as a *demonstration* and a way to provide intuition to readers.

```
@fedt_experiment
def physical_deflation_demo():
    snowman = Laser.fab(GeometryFile('snowman.svg'), default_settings=electripop_laser_defaults)
    elapsed_times = BatchMeasurements.empty()
    elapsed_times += Stopwatch.measure_time(snowman, "deflate fully")
    analyze(elapsed_times.get_all_data())
```

A.6.6 Experiment: Volumetric Change. This experiment measured how much size change is possible between the uninflated and inflated states of a shape.

Original Paper Description. Volumetric change
Using our simulation tool, we characterized the volume of our snowman cutout (2.5 microns thin with a surface area of 19.20cm^2) and its total volume after inflation (104.49 cm^3). Thus the total percentage increase in volume is roughly 200,000%.

Replication in FEDT.

```
@fedt_experiment
def volumetric_change():
    configure_for_electripop()
    instruction("set up the snowman linefile and programmatically inflate it")
    snowman_virt = LineFile('snowman.svg')
    snowman_phys = Laser.fab(snowman_virt, material="mylar")
    volumes = BatchMeasurements.empty()
    volumes += ManualGeometryScanner.scan(snowman_phys)
    volumes += ManualGeometryScanner.scan(snowman_virt)
    analyze(volumes.get_all_data())
```

Corrected Experiment. Similar to above, the authors described this as a demonstration rather than an experiment.

```
@fedt_experiment
def volumetric_change_demo():
    instruction("set up the snowman linefile and programmatically inflate it")
    snowman_virt = GeometryFile('snowman.svg')
    snowman_phys = Laser.fab(snowman_virt, default_settings=electripop_laser_defaults)
    volumes = BatchMeasurements.empty()
    volumes += ManualGeometryScanner.scan(snowman_phys)
    volumes += ManualGeometryScanner.scan(snowman_virt)
    analyze(volumes.get_all_data())
```

A.6.7 Experiment: Fabrication Time. This experiment measured how long it takes to fabricate objects for ElectriPop on a laser cutter.

Original Paper Description. Fabrication Time As noted earlier, our designs can be cut by many different technologies that range from slow and laborious (e.g., scissors — one output after several minutes) to fast and precise (e.g., die cutting — many outputs per second). Laser cutting lies somewhere in the middle in terms of speed, and it is also a tool available in many fabrication spaces. In the case of our snowman example, the cut time is 30 seconds on our ULS 30W CO2 laser cutter (100% speed, 5% power).

Replication in FEDT.

```
@fedt_experiment
def fabrication_time():
    # discuss whether this is an experiment, or a measurement/characterization
    snowman = GeometryFile('snowman.svg')
    elapsed_times = BatchMeasurements.empty()
    elapsed_times += Stopwatch.measure_time(snowman, "fabricate on the laser")
    analyze(elapsed_times.get_all_data())
```

Corrected Experiment. As above, authors described this as a demonstration rather than an experiment. They also noted that they did not time the cuts; they used the laser's time estimation.

```
@fedt_experiment
def fabrication_time_demo():
    snowman = GeometryFile('snowman.svg')
    elapsed_times = BatchMeasurements.empty()
    elapsed_times += Stopwatch.measure_time(snowman, "fabricate on the laser")
    analyze(elapsed_times.get_all_data())
```

A.6.8 Experiment: Geometric Accuracy. This experiment explored the accuracy of the full simulation to fabricated objects.

Original Paper Description. To measure the geometric accuracy of our simulation, we compared the 3D model of our simulated result with its real-world output. As noted in section 5.0.1, it was not possible to use a 3D scanning tool (e.g., laser scanning, photogrammetry) to automatically create a 3D model of our physical output, due to the thin and reflective nature of mylar. Instead, we manually created a 3D model that matched the physical output, creating a ground truth. We did this for our rose, snowman, and Christmas tree designs (Figure 1B, 3, 22 respectively). Now, with both a 3D model of the physical output and a 3D model of the simulated output, we can iterate through all vertices to compute the distance error between the two models. To remove rotation as a factor, the base vertices in both models are aligned. We ran our simulations 100 times on a computer with an X3900 RYZEN CPU. Table 1 also reports the average simulation convergence time for each design. Simulation for most designs takes around ten seconds to fully converge, and the user is able to watch the simulation play out in real time.

Across all simulation trials, we found a mean 3D euclidean error of 1.55mm. Broken out by design, our rose, Christmas tree and snowman simulation results had errors of 3.43mm (SD=0.34), 0.75mm (SD=1.30), and 0.46mm (SD=0.15) respectively (Table 1). Qualitatively, we found

that the models largely matched in gestalt. While there were small variations, often accessory details, these were more often than not "correct" answers, and the output depended on how the form was inflated or perturbed from airflow. Put simply: the simulated result was not incorrect, but rather a different correct state. This can be seen in Figure 24, which visualizes the error distribution for our worst performing output: the flower. Note that most of the form is in dark blue (accurate), with a few errorful regions (shades of red) that are still plausibly posed. Overall, our empirical results match our observations that our simulation tool could be relied upon to provide a preview of the 3D form.

Replication in FEDT.

```
@fedt_experiment
def geometric_accuracy():

    test_files = [GeometryFile(fname) for fname in ['rose.svg', 'snowman.svg', 'christmas_tree.svg']]

    ground_truths = BatchMeasurements.empty()
    simmed = BatchMeasurements.empty()

    for f in Parallel(test_files):
        fabbed_object = Laser.fab(f, default_settings=electripop_laser_defaults)
        # laser? vinyl cutter? scissors?
        ground_truths += ManualGeometryScanner.scan(fabbed_object)
        for sim_repetition in Parallel(range(100)):
            sim = VirtualWorldObject({'file': CustomSimulator.runsimulation(f)})
            simmed += ManualGeometryScanner.scan(sim)
            simmed += Stopwatch.measure_time(sim, "converge simulation")

    analyze(ground_truths.get_all_data())
    analyze(simmed.get_all_data())
```

Corrected Experiment. The authors confirmed that they had used a laser cutter to create the objects. They also remarked that they had re-used the same fabricated objects for several experiments, but that these were replaced as they broke. We have added a function that takes care of this.

```
SNOWMAN = None
ROSE = None
TREE = None
def get_objects():
    global SNOWMAN, ROSE, TREE

    if SNOWMAN == None or Human.do_and_respond("check if snowman is broken", "is it broken?") == 'yes':
        SNOWMAN = Laser.fab(GeometryFile("snowman.svg"))
    if ROSE == None or Human.do_and_respond("check if rose is broken", "is it broken?") == 'yes':
        ROSE = Laser.fab(GeometryFile("rose.svg"))
    if TREE == None or Human.do_and_respond("check if tree is broken", "is it broken?") == 'yes':
        TREE = Laser.fab(GeometryFile("christmastree.svg"))

    return [SNOWMAN, ROSE, TREE]

@fedt_experiment
def geometric_accuracy():

    test_objects = get_objects()

    ground_truths = BatchMeasurements.empty()
    simmed = BatchMeasurements.empty()

    for fabbed_object in Parallel(test_objects):
        instruction("inflate the object")
        ground_truths += ManualGeometryScanner.scan(fabbed_object)
```

```
        for sim_repetition in Parallel(range(100)):
            sim = VirtualWorldObject('file.sim', {'file': CustomSimulator.runsimulation(
                        fabbed_object.metadata['line_file']), 'repetition': sim_repetition})
            simmed += ManualGeometryScanner.scan(sim)
            simmed += Stopwatch.measure_time(sim, "converge simulation")

    analyze(ground_truths.get_all_data())
    analyze(simmed.get_all_data())
```

A.7 Paper 7 : FabHydro

Adapted from Yan, et al., 2021 [87].

A.7.1 Experiment: Resins. This experiment compares behaviours of a single object fabricated with different types of resins.

Original Paper Description. Printing Material We experimented with three different types of resin: the standard photosensitive resin [9], the Siraya Tech Tenacious resin [45] (one of the most accessible flexible resins on the market), and the F39/F69 resin from RESIONE [35]. The goal is to find the resin that can be used to print deformable structures, i.e., structures that are both compressible and stretchable.

To understand the flexibility of the selected materials, we printed a set of hollow bellows with 1 mm wall thickness with all three materials (Figure 3a). The one printed with the standard resin cracked under loads before any noticeable deformation (Figure 3c), suggesting that it will not fulfill our goal despite being widely available. The Tenacious resin is performing better in compression than the standard resin, as the model could be compressed with obvious deformation. However, the rebound speed is slow due to the stiffness, and the elongation of the model is limited by the elongation at the break ratio of the material (Figure 3b). Thus, this impact-resistant resin is not ideal as hydraulic systems require faster rebound and better stretchability. With the same testing model, the sample printed with F39/F69 resin can be compressed to half the length and extended to 1.5 times the length without breaking (Figure 3d and e), making it an ideal candidate for printing hydraulic devices. See Table 1 for the summary of the material properties and behaviors.

Replication in FEDT.

```
@fedt_experiment
def resin_types():
    stl = GeometryFile('bellows_1mm.stl')

    compression_results = BatchMeasurements.empty()
    tension_results = BatchMeasurements.empty()
    for resin in Parallel(['standard','tenacious','f39/f69']):
        fabbed_object = Printer.fab(stl, material=resin)
        instruction(f"compress object #{fabbed_object.uid} as much as possible (?)") # with what? how much?
        compression_results += Calipers.measure_size(fabbed_object,"height after compression")
        instruction(f"extend object #{fabbed_object.uid} as much as possible (?)") # with what? how much?
        tension_results += Calipers.measure_size(fabbed_object,"height after stretching")

    analyze(compression_results.get_all_data())
    analyze(tension_results.get_all_data())
```

Corrected Experiment. This experiment was somewhat less formal than we had thought, and mainly involved human intuition for deciding if something was reasonable. The authors also noted that they found the bellows file in prior work, and that they had to recompile it to be the correct size for their project.

```
@fedt_experiment
def resin_types():
    stl = GeometryFile('bellows_1mm_priorwork_recompiled.stl')

    compression_results = BatchMeasurements.empty()
    tension_results = BatchMeasurements.empty()
    for resin in Parallel(['standard','tenacious','f39/f69']):
        fabbed_object = Printer.fab(stl, material=resin)
        instruction(f"compress object #{fabbed_object.uid} as much as possible")
        compression_results += Human.judge_something(fabbed_object, "height after squishing is good")
        compression_results += Human.judge_something(fabbed_object, "speed of squshing is good")
        instruction(f"extend object #{fabbed_object.uid} as much as possible")
```

```
        compression_results += Human.judge_something(fabbed_object, "height after stretching is good")
        compression_results += Human.judge_something(fabbed_object, "speed of stretching is good")

    analyze(compression_results.get_all_data())
    analyze(tension_results.get_all_data())
```

A.7.2 Experiment: Bend vs. Thickness. This experiment explored flexibility of structures with different thicknesses in the chosen resin.

Original Paper Description. Besides flexible structures, a working hydraulic system also needs certain spots to remain strong and rigid to serve as structural supports. As the Poisson effect suggests, the thicker the material gets, the stronger it is to resist deformation. We thus further explored how the deformation is affected by the structure thickness of models printed with the F39/F69 resin.

Figure 4a shows nine testing samples with a rectangular cross-section (20 mm × 30 mm) and a thickness ranges from 1 mm to 5.5 mm, with a 0.5 mm increment. We fixed one end of the samples and engaged a load of 0.49 N at the other end. Figure 4b shows the bending angle versus the thickness in the given dimension of the printed samples. We conclude that a printed wall with a thickness of 3 mm can serve the purpose of a stiff structure, while a wall with less than 2 mm can create a hinge with great flexibility. All models in the rest of the paper are printed with the F39/F69 flexible resin unless otherwise noted.

Replication in FEDT.

```
@fedt_experiment
def bend_vs_thickness():

    bend_results = BatchMeasurements.empty()

    for thickness in Parallel(arange(1,5.5+include_last,.5)):
        stl = StlEditor.cube((20,30,thickness))
        fabbed_object = Printer.fab(stl)
        instruction(f"fix object #{fabbed_object.uid} at one end and hang a load of 0.49N at the other end")
        bend_results += Protractor.measure_angle(fabbed_object,"angle of tip below 90 degrees")

    # what is the source of the error bars?

    analyze(bend_results.get_all_data())
```

Corrected Experiment. The authors described that they did repeat the experiment several times per object, but that it was not predetermined how many times that would be. They also informed us that they did not measure with a protractor on the object, but from a photograph.

```
@fedt_experiment
def bend_vs_thickness():

    bend_results = BatchMeasurements.empty()

    for thickness in Parallel(arange(1,5.5+include_last,.5)):
        stl = StlEditor.cube((20,30,thickness))
        fabbed_object = Printer.fab(stl)
        for repetition in Series(arange(1,3+include_last)): # somewhat unofficial how many repetitions
            instruction(f"fix object #{fabbed_object.uid} at one end and hang a load of 0.49N at the other end")
            instruction("take a photo of the object and use a protractor on the image")
            bend_results += Protractor.measure_angle(fabbed_object,"angle of tip below 90 degrees")

    analyze(bend_results.get_all_data())
```

A.7.3 Experiment: Thin Wall Thickness. This experiment looked at the pressure that can be held by walls of different thickness.

Original Paper Description. Minimum Thin Wall Thickness We report that the thin wall can be reliably printed with a minimum thickness of 0.7 mm. To understand the maximum pressure it can take, we further experimented with two sets of small chambers, one side printed with the 0.7 mm wall thickness and the rest with 5 mm (Figure 16a). Note that the two sets of chambers were also printed with two orientations: one with the printed layers visible across the surface (set 1) and the other with the layers parallel to the surface with no visible seam (set 2). For testing, we ran three samples for each set by pumping water into these chambers until they rapture. The pressure was measured through a regulator. We confirm that the inner walls, even with a 0.7 mm thickness, can undertake the pressure up to 32 psi when printed in the orientation of set 1 and 33 psi for the set 2 samples.

Replication in FEDT.

```
@fedt_experiment
def min_wall_thickness():

    pressure_results = BatchMeasurements.empty()

    base_stl = GeometryFile("wall_thickness_test.stl")
    for orientation in Parallel([0,90]):
        stl = StlEditor.rotate(base_stl, orientation)
        for repetition in Parallel(range(3)):
            fabbed_object = Printer.fab(stl, repeitition=repetition)
            instruction(f"inflate object #{fabbed_object.uid} until it ruptures")
            pressure_results += PressureSensor.measure_pressure(fabbed_object,"pressure at rupture")

    analyze(pressure_results.get_all_data())
```

Corrected Experiment. No changes were required.

A.7.4 Experiment: Minimum Wall Spacing. This experiment looked at how far apart walls could be printed without joining together.

Replication in FEDT.

```
@fedt_experiment
def min_wall_spacing():
    # not clear what the experiment was? they say "Based on our printing test, ..."
    analyze("1.4mm")
```

Corrected Experiment. The authors clarified that they had done a stepped method of testing to find where walls would cease to separate.

```
@fedt_experiment
def min_wall_spacing():
    acceptability_results = ImmediateMeasurements.empty()
    base_stl = GeometryFile("thin_wall.stl")
    neighbours_separated = True
    separation = 2.5
    while neighbours_separated:
        printing_stl = StlEditor.modify_design(base_stl, "spacing between copies", separation)
        fabbed_object = Printer.fab(printing_stl)
        neighbours_separated = acceptability_results.do_measure(fabbed_object,
                                                                 TrueFalser.truefalse.set_feature(
                                                                         "did the neighbouring cubes separate?"))
        if neighbours_separated:
            separation -= .1

    analyze(acceptability_results)
```

A.7.5 Experiment: Thin Wall Area. This experiment looked at deformation of thin spots under pressure, and how this relates to their surface area.

Original Paper Description. Minimum Thin Wall Area Another factor that affects the deformation is the expansion distance generated from the thin wall area. A longer expansion distance will push the adjacent units further away from each other, resulting in a more aggressive overall deformation. In this experiment, we altered the thin wall area to observe the changes of the expansion distance (Figure 16b). Specifically, we printed a series of chambers with one wall with 0.7 mm thick and the rest with 5 mm, the same as in the previous experiment. The cross-section area is a square shape, thus the size of the area is controlled by the edge length, which ranges from 6 mm to 15 mm, with 1 mm increments (Figure 16c). Pressurized water was pumped into the chambers with 20 psi (Figure 16b). The vertical expansion height is shown in the graph (Figure 16d). According to the result, we conclude that the maximum expansion height is approximately linear against the edge length of the inner wall. When considering together with the wall spacing (see above), we can conclude that the thin wall should have edges not smaller than 6 mm.

The above design parameters are summarized in Table 2.

Replication in FEDT.

```
@fedt_experiment
def min_thin_wall_area():
    expansion_results = BatchMeasurements.empty()

    for edge_length in Parallel(arange(6,15+include_last)):
        stl_loc = Human.do_and_respond(
                          f"create an STL with edge length {edge_length}, thin wall 0.7mm, other walls 5mm",
                          "where is the STL?")
        stl = GeometryFile(stl_loc)
        fabbed_object = Printer.fab(stl)
        instruction("pump 20psi into the fabricated object")
        expansion_results += Calipers.measure_size(fabbed_object,"vertical expansion height")

    analyze(expansion_results.get_all_data())
```

Corrected Experiment. No corrections were required.

A.7.6 Experiment: Hydraulic versus Pneumatic. This experiment looked at the relative behaviour of objects made with internal air versus internal resin fluid.

Original Paper Description. In this work, we adopted hydraulic transmission over pneumatic for two reasons. First, a hydraulic mechanism is more energy effcient. We compared both methods' transmission efficiency by printing two sets of single-actuator-single-generator systems with an identical design. We filled one system with water and sealed the other with air. Upon full compression of the generator, the hydraulic-driven system's actuation has a more significant displacement (figure 24). This is caused by the non-linearity of the volume-pressure relationship in the pneumatic systems. Similar effects can be seen in balloons where the air volume increases, but the pressure tends to remain stable.

Replication in FEDT.

```
@fedt_experiment
def pneumatic_vs_hydraulic():
    # demonstration or experiment?
    expansion_results = BatchMeasurements.empty()

    stl = GeometryFile("single_actuator_single_generator.stl")
    for fill in Parallel(['water','air']):
        fabbed_object = Printer.fab(stl)
        filled_object = Human.post_process(fabbed_object,f"fill object with {fill}")
        expansion_results += Calipers.measure_size(filled_object,"displacement of tip")

    analyze(expansion_results.get_all_data())
```

Corrected Experiment. The authors confirmed that this was a demonstration rather than a formal experiment, and that they were uninterested in reporting specific numbers, but rather that they wanted to give intuition about behaviours.

A.7.7 Experiment: Lasting. This experiment printed a large collection of cubes and allowed them to sit in natural sunlight, cutting a few open each day to see how much the sunlight had cured the resin inside.

Original Paper Description. The UV-sensitive resin will cure when exposed to UV light. One of the main concerns we have for a printed hydraulic device, especially with the Submerged Printing approach, is that the liquid resin may get cured after the object is printed.

To understand how long the printed object will still be functional under normal indoor usage, we ran an exposure experiment for a series of printed cubes that all have locked-in resin. Specifically, we printed 48 testing cubes with the side length of 11 mm and wall thickness of 1 mm using the Submerged Printing process. All samples were placed in an indoor environment with a light intensity of 400 to 600 lux. Every 24 hours, we cut open three cubes to measure how much of the resin was still in the liquid phase by weighting the cube before and after the cut, as shown in Figure 25a. Figure 25b shows the result of the 16-day experiment. Over time, we observe that the liquid resin does get cured gradually, but mainly close to the wall. The amount of cured resin reaches stability at day eight, with still more than 75% of resin remains in the liquid phase. Thus, our printing method is proper for rapid prototyping. Increasing the wall thickness can further reduce the fluid loss, but at the cost of losing flexibility.

Replication in FEDT.

```
@fedt_experiment
def lasting():
    stl = GeometryFile("side_11mm_wall_1mm.stl")
    fabbed_objects = []
    for repetition in Parallel(range(48)):
        fabbed_objects.append(Printer.fab(stl, repetition=repetition))

    pre_weights = BatchMeasurements.empty()
    post_weights = BatchMeasurements.empty()

    cubes_per_day = 3
    for day in Series(range(16)):
        Environment.wait_up_to_time_multiple(fabbed_objects, num_days=day)
        cubes = fabbed_objects[day*cubes_per_day:(day+1)*cubes_per_day]
        for cube in Parallel(cubes):
            pre_weights += Scale.measure_weight(cube)
            leaked_cube = Human.post_process(cube, f"cut open the cube and let the fluid leak out")
            post_weights += Scale.measure_weight(leaked_cube)

    analyze([pre_weights.get_all_data(),post_weights.get_all_data()])
```

Corrected Experiment. The authors confirmed that the procedure we had written was correct according to their paper, but that in reality they had printed cubes "as possible" given other use of their printer. Each cube's age was individually tracked, and it was cut open at the appropriate time. This type of experiment pragmatics is challenging to encode in FEDT, and is discussed in Future Work.

A.8 Paper 8 : G-ID

Adapted from Dogan, et al., 2020 [20].

A.8.1 Experiment: Parameter Spacings. This experiment explores different parameter spacings to determine which are distinguishable for the algorithm.

Original Paper Description. We conducted an experiment to determine which parameter spacing can be reliably identified using our detection method. For the experiment, we printed a number of instances with different slicing settings and used our detection method to identify each pattern. Table 1 summarizes the results of this experiment under regular office light conditions. We focus this analysis on the parameters related to bottom surface and infill, which are seen from the object's base, and do not consider those related to the side (intermediate layers).

...

To select these objects, we downloaded the top 50 3D models from Thingiverse [48] and ranked them by their bottom surface area (i.e., the largest square one may inscribe in the contour of the first slice, determined by an automated MATLAB script). We found that 25 models had a large surface area (>6cm^2), 7 models had a medium surface area (1.2-6cm^2), and 18 models had small surface areas (<1.2cm^2). These ranges were determined empirically based on our initial tests. We randomly picked one object representing each of these three categories and printed multiple instances using the parameters below.

Determining the Range for Each Slicer Setting Before slicing each of the models with different settings, we first determined the min and max values for each setting. *Bottom*: For the bottom angle, we can use 0°-180°. Going beyond 180°would cause instances to be non-distinguishable (i.e., 90°looks the same as 270°). We took the min value for initial bottom line width as 0.35mm, the default value recommended in the slicer Cura. Although this parameter can be as large as twice the nozzle size (2*0.4mm), we limit the max value to 0.6mm to avoid disconnected lines. For the pattern settings, which do not have min and max values, we considered the "line" pattern for the bottom surface.

Infill: As for the infill angle, we can use a range of 0°-60° for the trihexagon and triangular patterns, and 0°-90°for the grid pattern, as their layouts are periodic with period 60°and 90°, respectively. For infill line distance (density), we determined that having infill units smaller than 2.6mm makes the pattern unrecognizable — we thus used it as the min value. The max value is 3.2mm for objects with medium base area but may go up to 8.0mm for larger objects. Going beyond this value would imply an infill density of less than 10%, and thus fragile, less stable objects. The three infill pattern (type) settings do not have min or max values.

Slicing with Different Spacings and Capturing Photos Next, we used our three selected objects (small, medium, large), and printed them with different slicing settings. Our goal was to determine how finely we can subdivide the given parameter ranges for accurate detection. To find the optimal spacing for each parameter, we made pairwise comparisons of two values (e.g., for angles, instance #1: 8°- instance #2: 5°; difference: 3°), while keeping all other parameters constant. Based on 16 pictures taken for each pairwise comparison, we report the accuracy at which we can distinguish the two instances. For the prints with infill variations, we held a small light source (Nitecore Tini [34]) against the side of the 3D printed object before taking the image.

Results of the Experiments As expected, objects from the "small" category did not have sufficient base area to fit in enough infill units and thus not give satisfactory results. We therefore conclude that G-ID cannot be used for very small bases and excluded them from the rest of the analysis. We next discuss the results for each slicing parameter for the medium and large object category. *Bottom Line Angle & Width* The dashed lines in Figure 11a,b indicate that a spacing of 5°and 0.05 mm provides a classification accuracy of 100for both the medium and large base area categories, respectively. Thus, a range of 0°-180°would give us 36 variations for the angle. However, we exclude the two degrees 0°and 90°from the bottom line angle range since these are reserved for error checking (as described in section "Detecting Slicing Parameters"), therefore we have 34 variations. A range of 0.35-0.6mm allows 36 variations for the width.

Infill Angle & Line Distance (Width) The dashed lines in Figure 11c,d show that a spacing of 5°and 0.6mm provides a 100% detection accuracy for the two categories, respectively. Thus, for the ranges of 0°-60° and 0°-90°, we can use 12 or 18 variations, respectively. For the width, we can use, for medium objects, a range of 2.6- 3.2mm (2 variations); and for large objects 2.6-8.0mm (10 variations). We report the smaller number in Table 1.

Infill Pattern The confusion matrix in Figure 12 shows that the "grid" and "trihexagon" work for both medium and large classes. For large objects, we can also use all three different patterns.

Replication in FEDT.

```
class Blender:
    def render_gcode(gcode: VirtualWorldObject):
        instruction("render the object in blender")
        return gcode

@fedt_experiment
def find_bottom_spacings():
    large_object = GeometryFile("large from thingiverse.stl")
    medium_object = GeometryFile("medium from thingiverse.stl")
    small_object = GeometryFile("small from thingiverse.stl")

    bottom_angle_results = BatchMeasurements.empty()
    bottom_width_results = BatchMeasurements.empty()
    infill_angle_results = BatchMeasurements.empty()
    infill_density_results = BatchMeasurements.empty()
    infill_pattern_results = BatchMeasurements.empty()

    # these guesses are made based on the labels in the figure at the top of page 8,
    # which has no caption or name, and table 1
    for obj_file in Parallel([large_object,medium_object,small_object]):

        for bottom_angle in Parallel(arange(0,180+include_last,1)): # or was it arange(0,6+include_last,1) ?
            fabbed_object = Printer.fab(obj_file, bottom_angle=bottom_angle)
            bottom_angle_results += Camera.take_picture(fabbed_object, "bottom")

        for bottom_width in Parallel(arange(0.35,0.6+include_last,.01)):
            # or was it arange(0.35,0.35+0.06+include_last,.01) ?
            fabbed_object = Printer.fab(obj_file, bottom_width=bottom_width)
            bottom_width_results += Camera.take_picture(fabbed_object, "bottom")

        for infill_pattern in Parallel(['trihexagon','triangular','grid']):
            infill_angles = None
            if infill_pattern in ['trihexagon', 'triangular']:
                infill_angles = arange(0,60+include_last,1) # or was it arange(0,6+include_last,1) ?
            else: # if it's grid
                infill_angles = arange(0,90+include_last,1) # or was it arange(0,6+include_last,1) ?
            for infill_angle in Parallel(infill_angles):
                fabbed_object = Printer.fab(obj_file,
                                            infill_pattern=infill_pattern,
                                            infill_rotation=infill_angle)
                fabbed_object = Human.post_process(fabbed_object, "hold a light above the object")
                picture = Camera.take_picture(fabbed_object, "bottom")
```

```
                infill_angle_results += picture
                if infill_angle == 0:
                    infill_pattern_results += picture

        for infill_density in Parallel(arange(2.6, 3.2+include_last, 0.1)):
            # or was it arange(2.6, 2.6+0.7+include_last, 0.1) ?
            fabbed_object = Printer.fab(obj_file,
                                        infill_density=infill_density)
            infill_density_results += Camera.take_picture(fabbed_object, "bottom")


    analyze(bottom_angle_results.get_all_data())
    analyze(bottom_width_results.get_all_data())
    analyze(infill_pattern_results.get_all_data())
    analyze(infill_angle_results.get_all_data())
    analyze(infill_density_results.get_all_data())
    # then they do comparative work by examining the relative differences of object features...

    # were other fabrication variables (e.g., filament colour) controlled? ->
    # maybe it was white? this is suggested in a later section
```

Corrected Experiment. The authors corrected a few things about our implementation: filament color was fixed to be white. They only printed parts of the object in some cases (i.e., the bottom several layers), which is not currently supported in our libraries. They did not as aggressively search the parameter space, but did a more conservative search than we originally guessed. In addition, they noted that we had neglected to include 16 photos of each object.

```
class Blender:
    def render_gcode(gcode: VirtualWorldObject):
        instruction("render the object in blender")
        return gcode

class ExternalApp:
    def analyze(photo):
        pass

@fedt_experiment
def find_bottom_spacings():
    large_object = GeometryFile("large from thingiverse.stl")
    medium_object = GeometryFile("medium from thingiverse.stl")
    small_object = GeometryFile("small from thingiverse.stl")

    bottom_angle_results = BatchMeasurements.empty()
    bottom_width_results = BatchMeasurements.empty()
    infill_angle_results = BatchMeasurements.empty()
    infill_density_results = BatchMeasurements.empty()
    infill_pattern_results = BatchMeasurements.empty()

    for obj_file in Parallel([large_object,medium_object,small_object]):
        for bottom_angle in Parallel(arange(0,6+include_last,1)):
            fabbed_object = Printer.fab(obj_file, bottom_angle=bottom_angle, filament_color='white')
            # need to add partial print option
            for photo_rep in Parallel(range(16)):
                bottom_angle_results += Camera.take_picture(fabbed_object, "bottom")

        for bottom_width in Parallel(arange(0.35,0.35+0.06+include_last,.01)):
            fabbed_object = Printer.fab(obj_file, bottom_width=bottom_width, filament_color='white')
            for photo_rep in Parallel(range(16)):
                bottom_width_results += Camera.take_picture(fabbed_object, "bottom")
```

```
        for infill_pattern in Parallel(['trihexagon','triangular','grid']):
            for infill_angle in Parallel(arange(0,6+include_last,1)):
                fabbed_object = Printer.fab(obj_file,
                                            infill_pattern=infill_pattern,
                                            infill_rotation=infill_angle,
                                            filament_color='white')
                fabbed_object = Human.post_process(fabbed_object, "hold a light above the object")
                for photo_rep in Parallel(range(16)):
                    picture = Camera.take_picture(fabbed_object, "bottom")
                infill_angle_results += picture
                if infill_angle == 0:
                    infill_pattern_results += picture

        for infill_density in Parallel(arange(2.6, 2.6+0.7+include_last, 0.1)):
            fabbed_object = Printer.fab(obj_file,
                                                infill_density=infill_density,
                                                filament_color='white')
            for photo_rep in Parallel(range(16)):
                infill_density_results += Camera.take_picture(fabbed_object, "bottom")


    analyze(bottom_angle_results.get_all_data())
    analyze(bottom_width_results.get_all_data())
    analyze(infill_pattern_results.get_all_data())
    analyze(infill_angle_results.get_all_data())
    analyze(infill_density_results.get_all_data())
```

A.8.2 Experiment: Cross-Validation. This experiment creates a few objects with random parameters and attempts to distinguish them.

Original Paper Description. To cross-validate our parameter spacing, we printed another random set of 16 objects (10 large, 6 medium) from the same top 50 model list from Thingiverse (excluding the ones we used in the previous experiment) with white filament. Before printing, we randomly assigned each model a combination of slicing settings from the available parameter space and then tested if G-ID can identify them. Our second goal was to see if the parameter spacing still applies when multiple parameters are varied at the same time (our previous experiment only varied one slicing parameter per instance at a time). Among the 10 objects with large bottom area, all slicing parameters were correctly identified. Among the 6 objects from the medium category, 5 were correctly identified. In total, the detection accuracy is 93.75%. The falsely identified model had the smallest bottom surface area ($1.4cm^2$), which confirms the fact that objects without sufficient surface area cannot be recognized.

Replication in FEDT.

```
def random_param_set():
    infill_pattern = random.choice(['trihexagon','triangular','grid'])
    infill_rotation_range = None
    if infill_pattern in ['trihexagon', 'triangular']:
        infill_rotation_range = arange(0,60,1)
    else: # if it's grid
        infill_rotation_range = arange(0,90,1)
    return (random.choice(arange(0,180,5)), # bottom angle
            random.choice(arange(0.35,0.6+include_last,.05)), # bottom width
            infill_pattern, # infill pattern
            random.choice(infill_rotation_range), # infill rotation
            random.choice(arange(2.6, 3.2+include_last, 0.6))) # infill density / width

@fedt_experiment
def cross_validation():
    large = [GeometryFile("large obj from thingiverse.stl")] * 10
    medium = [GeometryFile("medium obj from thingiverse.stl")] * 6
    objs = large + medium
```

```
    filament = 'white'

    all_object_results = BatchMeasurements.empty()

    for obj in Parallel(objs):
        bottom_angle, bottom_width, infill_pattern, infill_rotation, infill_density = random_param_set()
        fabbed_object = Printer.fab(obj,
                                    infill_pattern=infill_pattern,
                                    infill_rotation=infill_rotation,
                                    bottom_angle=bottom_angle,
                                    bottom_width=bottom_width,
                                    infill_density=infill_density,
                                    filament_color=filament)
        fabbed_object = Human.post_process(fabbed_object,
                                           "hold a light above the object") # I'm assuming this happened
        all_object_results += Camera.take_picture(fabbed_object, "bottom")

    analyze(all_object_results.get_all_data())
```

Corrected Experiment. No corrections were required.

A.8.3 Experiment: Lighting. This experiment subjected six random objects to different lighting conditions to determine how easily they could be distinguished.

Original Paper Description. Surface: To see how the filament's color and different lighting affect the detection of surface parameters, we printed six instances of the key cover with eight different colors of Ultimaker PLA filaments (total of 48 instances) and using a dimmable LED lamp varied the light in the room from 0 – 500 lux (measured with lux meter Tacklife LM01). We picked the filament colors to be the primary and secondary colors of the RYB model, as well as white and black filament. We selected the light intensities to represent different low-light conditions. For slicing, we used surface parameters distributed evenly along bottom line angle and width within the allowed range from Table 1. The results are shown in Figure 14. All colors worked well for lighting conditions above 250 lux. This shows that our approach works well in classrooms (recommended light level: 250 lux) [32] and offices/laboratories (recommended light level: 500 lux). The results also show that the camera needs more light to resolve the lines for lighter colors (i.e., white and yellow) than for darker colors. Since we used white filament for our parameter spacing evaluation, the results in Table 1 work even for worst-case scenarios. This also means that for other filament colors, an even smaller surface area would suffice for correct detection since the surface details can be better resolved.

Infill: We were able to detect the patterns for all of the aforementioned filament colors except for black due to its opaque nature. Further, the brighter the light, the thicker the object base may be: 145 lumens suffice for a base of 1mm (suggested thickness in Cura), 380 lumens suffice for 1.75 mm.

Replication in FEDT.

```
@fedt_experiment
def materials_lighting_thicknesses():
    model = GeometryFile("keycover.stl")
    filament_colors = ['red', 'yellow', 'blue', 'orange', 'green', 'purple', 'black', 'white']
    bottom_line_angles = list(arange(0,180+include_last,(180-0)/6)) # evenly spaced for 6 types of prints?
    bottom_line_widths = list(arange(0.35,0.6+include_last,(.6-.35)/6))
    # evenly spaced for 6 types of prints? : doesn't conform to table
    light_intensities = arange(0,500+include_last,10)
    # unclear what step size was used, or if discrete values were used?

    lighted_photos = BatchMeasurements.empty()

    for color in Parallel(filament_colors):
        for config_id in Parallel(range(len(bottom_line_angles))):
            fabbed_object = Printer.fab(model,
                                        filament_color=color,
                                        bottom_line_angle=bottom_line_angles[config_id],
                                        bottom_line_width=bottom_line_widths[config_id])
```

```
            for light_intensity in Series(light_intensities):
                fabbed_object = Human.post_process(fabbed_object, f"light brightness of {light_intensity}")
                lighted_photos += Camera.take_picture(fabbed_object, "bottom")

    analyze(lighted_photos.get_all_data()) # analysis on minimum detectable illuminances, infill, etc.
```

Corrected Experiment. The authors told us that the experiment was actually carried out more as a **while** loop than a **for** loop. They essentially turned the light from all-dark up to bright until the object started being detected correctly, and recorded the values.

```
@fedt_experiment
def materials_lighting_thicknesses():
    model = GeometryFile("keycover.stl")
    filament_colors = ['red', 'yellow', 'blue', 'orange', 'green', 'purple', 'black', 'white']
    bottom_line_angles = list(arange(0,180+include_last,(180-0)/6))
    bottom_line_widths = list(arange(0.35,0.6+include_last,(.6-.35)/6))
    light_intensity = 0
    intensity_step = 10 # wasn't quite this formal, but this is ok

    lighted_photos = ImmediateMeasurements.empty()

    for color in Parallel(filament_colors):
        for config_id in Parallel(range(len(bottom_line_angles))):
            fabbed_object = Printer.fab(model,
                                        filament_color=color,
                                        bottom_line_angle=bottom_line_angles[config_id],
                                        bottom_line_width=bottom_line_widths[config_id])
            is_detecting = False
            while not is_detecting:
                light_intensity += intensity_step
                fabbed_object = Human.post_process(fabbed_object, f"light brightness of {light_intensity}")
                photo = lighted_photos.do_measure(fabbed_object, Camera.image.set_feature("bottom"))
                is_detecting = ExternalApp.analyze(photo)

    analyze(lighted_photos.get_all_data())
```

A.8.4 Experiment: Different Printers. This experiment used different printers to determine how well settings matched across them.

Original Paper Description. Since G-ID takes as input universal units like millimeters (width) and degrees (angle), our method extends to FDM printers other than the one we used for the experiments. To confirm this, we fabricated six test prints on the 3D printers Prusa i3 MK3S and Creality CR-10S Pro in addition to the Ultimaker 3 that we used for our experiments, and inspected the traces laid down with a microscope. The line widths of the prints had an average deviation of 9.6μm (Prusa) and 10.7μm (Creality) from the Ultimaker 3 prints. The line angles had an average deviation of 0.5°(Prusa) and 0.25°(Creality). These deviations are insignificant for our detection method since the spacing values chosen for the parameters are much larger than these values. To verify this, we used our mobile app to take pictures of these samples and ran our algorithm, which correctly detected the unique identifiers.

Replication in FEDT.

```
@fedt_experiment
def different_printers():
    model = GeometryFile("keycover.stl") # maybe?
    # they are only testing bottom line widths and angles,
    # so I am assuming they use the same 6 configurations as above
    bottom_line_angles = list(arange(0,180+include_last,(180-0)/6)) # evenly spaced for 6 types of prints
    bottom_line_widths = list(arange(0.35,0.6+include_last,(.6-.35)/6)) # evenly spaced for 6 types of prints

    printers = ['Prusa i3 MK3S', 'Creality CR-10S Pro', 'Ultimaker 3']

    angular_deviation_results = BatchMeasurements.empty()
    width_deviation_results = BatchMeasurements.empty()
    photos = BatchMeasurements.empty()
```

```
    for printer in Parallel(printers):
        for config_id in Parallel(range(len(bottom_line_angles))):
            fabbed_object = Printer.fab(model,
                                        printer=printer,
                                        bottom_line_angle = bottom_line_angles[config_id],
                                        bottom_line_width = bottom_line_widths[config_id])
            photos += Camera.take_picture(fabbed_object, "bottom")
            instruction("Use a microscope to make the following measurements")
            width_deviation_results += Calipers.measure_size(fabbed_object, "width of trace")
            angular_deviation_results += Protractor.measure_angle(fabbed_object, "angle of trace to base")

    analyze(angular_deviation_results.get_all_data())
    analyze(width_deviation_results.get_all_data())
    analyze(photos.get_all_data())
```

Corrected Experiment. The authors confirmed that they had used the same configurations as in the previous experiment. No corrections were required.

A.8.5 Experiment: Different Cameras. This experiment explored the distance at which different cameras can reliably detect differences in tagged objects.

Original Paper Description. Distance: If the phone is held too far away from the object, the camera cannot detect the detailed grooves on the surface. To determine how far the camera can be held from the object, we took pictures with smartphones of different camera resolutions. We found that the iPhone 5s (8MP) can resolve the slicing parameter features up to 26cm, Pixel 2 (12.2MP) up to 36cm, and OnePlus 6 (16MP) up to 40cm. Taking into account the cameras' field of view, this means that users can fit in an object with one dimension as large as 29cm, 45cm, and 52cm, respectively. G-ID guides users into the correct camera distance by varying the size of the object outline displayed on the user's phone for alignment.

Replication in FEDT.

```
@fedt_experiment
def camera_distance():
    cameras = ['iPhone 5s', 'Pixel 2', 'OnePlus 6']

    fabbed_objects = [] # what? how many?

    camera_data = BatchMeasurements.empty()

    for camera in Parallel(cameras):
        for fabbed_object in Parallel(fabbed_objects):
            camera_data += Camera.take_picture(fabbed_object, f"bottom with {camera}")

    analyze(camera_data.get_all_data())
```

Corrected Experiment. The authors discussed that they just used objects from above, and emphasized their view that this was more intended to be demonstrative than scientific.

```
@fedt_experiment
def camera_distance():
    cameras = ['iPhone 5s', 'Pixel 2', 'OnePlus 6']

    fabbed_objects = [Printer.fab(x) for x in ['file_from_above.stl']]

    camera_data = BatchMeasurements.empty()

    for camera in Parallel(cameras):
        for fabbed_object in Parallel(fabbed_objects):
            camera_data += Camera.take_picture(fabbed_object, f"bottom with {camera}")

    analyze(camera_data.get_all_data())
```

A.8.6 Experiment: Camera Angle. This experiment simulated different configurations and camera angles of rendered, tagged objects to see how accurately they could be distinguished in theory.

Original Paper Description. Angle: Since our image registration technique uses affine transformation, it is not able to fully remove the perspective distortion if the camera angle varies strongly. However, since our app guides users to align the object, the distortion is negligible. To show our algorithm can robustly read bottom surface parameters on a variety of shapes, we created a virtual dataset similar to [28]. We downloaded the first 600 models from the Thingi10K database [57] that have a rotationally non-symmetric base of appropriate size, sliced them with a random set of slicing settings, and rendered the G-codes using 3D editor Blender. We placed the virtual camera at points located on a spherical cap above the object base, with 8 evenly sampled azimuthal angles for each of the 5 polar angles. The percentage of shapes read correctly for each polar angle value θ is given in Table 2. The spacing values chosen for the parameters act as a buffer to prevent false readings. The objects for which detection failed at small angles had rather rounded bases, which makes alignment challenging.

Replication in FEDT.

```
@fedt_experiment
def camera_angle():
    models = [GeometryFile('thingi10kdb.stl')] * 600
    angles = [f"at {deg} deg, along azimuth {azimuth}" for deg in [4, 6, 8, 10, 12] for azimuth in range(8)]

    images = BatchMeasurements.empty()

    for model in Parallel(models):
        bottom_angle, bottom_width, infill_pattern, infill_rotation, infill_density = random_param_set()
        gcode = Slicer.create_toolpath(model,
                                infill_pattern=infill_pattern,
                                infill_rotation=infill_rotation,
                                bottom_angle=bottom_angle,
                                bottom_width=bottom_width,
                                infill_density=infill_density)
        virtual_render = Blender.render_gcode(gcode)
        for angle in Parallel(angles):
            images += Camera.take_picture(virtual_render, angle)

    analyze(images.get_all_data())
```

Corrected Experiment. No corrections were required.

A.9 Paper 9 : KnitPicking Textures

Adapted from Hofmann, et al., 2019 [35].

A.9.1 Experiment: Texture Testing. This experiment knit every valid texture design that the team could run through their parser and optimizer, and characterized their knitted properties and stretch under load.

Original Paper Description. Sampling Strategy Stanfield and Griffiths curated 300 knitted textures. Of these, the KnitSpeak compiler interpreted 166 (55.3%) into Knit- Graphs which we machine-knitted. All textures in the book section "Bobbles and Leaves" were excluded for using wide increases, decreases, and cables that violate Property 4. All other excluded patterns included annotations that described actions that are not machine-knittable or included multiple yarns. We collected 1454 of the most recent and popular KnitSpeak samples from the 5979 patterns on Stitch-Maps . We excluded 393 (27.0%) samples that were written in the round and 755 (51.9%) because they violated Properties 1 or 2. We observed that these 755 samples were not textures but full patterns (e.g., sweaters) which are beyond the scope of this study. Ultimately, we compiled and knit 306 (21.0%) textures from Stitch-maps.

Swatch Construction and Measuring We constructed and measured textures as follows: (1) machine knit a 60 loop by 60 course swatch with an additional border including eyelets for alignment; (2) weigh the swatch; (3) lay the swatch on fine-grained sandpaper to prevent curling; (4) photograph the swatch in the un-stretched state; (5) measure the un-stretched swatch across the center axes; (6) connect the swatch to a stretching rig and load it with a constant mass; (7) photograph the stretched swatch; (8) measure the stretched swatch. Our measurement station is shown in Figure 6.

We knit our swatches on an Shima Seiki SWG91N2 15-gauge v-bed knitting machine using Tamm Petit, a 2/30NM (8,147 yards per pound) acrylic yarn with moderate twist. We used our machine's digital stitch control system to regulate yarn tension and our stitch size was 40 with leading set 25. We tiled each texture to ft a 60 loop by 60 course swatch, then added knit-stitches to the borders to fill in the gaps. The texture is surrounded by a 12-stitch-wide border of a checkered knit and purl texture to stabilize the swatch edges and normalize their connection to the rig. We placed an eyelet at the center and ends of each edge of the swatch. To stretch a swatch, we hooked each eyelet to

rods that can roll freely in one direction, pulling the swatch linearly along its width and height. These rods were attached to 608g weights. We gently dropped the weights off the edge of the table. We collected four measurements to derive gauge: stretched width (1) and height (2); un-stretched width (3) and height (4). Gauge is the number of loops per unit width and height in a texture. Gauge decreases as a texture is stretched. We calculated opacity as the count of black pixels (matching the sand paper background) shown through the knitted texture and appearing per photograph.

Replication in FEDT.

```
class CustomProgram(DesignSoftware):
    @staticmethod
    def modify_design(knitfile: GeometryFile, feature_name: str, feature_value: str|int):
        knitfile.updateVersion(feature_name, feature_value,
                               f"modify {knitfile.file_location}, set {feature_name} to {feature_value}")
        return knitfile

    @staticmethod
    def knitcarve(knitfile: GeometryFile, specification: str):
        return CustomProgram.modify_design(knitfile, 'knitcarve', specification)

    @staticmethod
    def basicremove(knitfile: GeometryFile, specification: str):
        return CustomProgram.modify_design(knitfile, 'knitcarve', specification)

    @staticmethod
    def showToCrowdworker(obj: RealWorldObject, worker: int):
        # don't increment the version, because this isn't really postprocessing
        pass

@fedt_experiment
def compare_knit_textures_from_dbs():
    textures_from_db = []
    for i in Parallel(range(306)):
        textures_from_db.append(GeometryFile("texture{}.knit".format(i)))

    knitted = []
    weights = BatchMeasurements.empty()
    for texture in Parallel(textures_from_db):
        CustomProgram.modify_design(texture, "tile", "60x60 (fill gaps with knit stitches)") # automatable or manual?
        CustomProgram.modify_design(texture, "edge", "12-stitch checkered knit/purl to edges")
        single_knitted = KnittingMachine.knit(texture)
        Human.post_process(single_knitted, "add eyelets to centre and end of each edge (8 eyelets)")
        knitted.append(single_knitted)
        weights += Scale.measure_weight(single_knitted)

    analyze(weights.get_all_data())

    dims = ImmediateMeasurements.empty()
    photos = ImmediateMeasurements.empty()

    for single_knitted in Parallel(knitted):
        Human.post_process(single_knitted, "lay on sandpaper")
        photos += Camera.take_picture(single_knitted)
        dims += Calipers.measure_size(single_knitted, "unstretched length")
        dims += Calipers.measure_size(single_knitted, "unstreteched width")
        Human.post_process(single_knitted, "hook onto the rig and load with 608g")
        photos += Camera.take_picture(single_knitted)
        # was there a photo of it stretched? I'm not clear on where opacity results are
        dims += Calipers.measure_size(single_knitted, "stretched length")
```

```
        dims += Calipers.measure_size(single_knitted, "stretched width")

    analyze(dims.get_all_data())
    analyze(photos.get_all_data())
```

Corrected Experiment. The authors indicated that the tiling and edging steps were a part of an initial instantiation step, rather than a modification of an existing design. They also corrected our understanding about the eyelets: they are not "hardware" eyelets, but just part of the knitted design (i.e., they are a CAD component, not a post-process component). Regarding loading, the authors informed us that they loaded and unloaded the length and width axes separately, but always in the same order rather than in arbitrary orders. They further clarified that some data whose gathering is described in the paper was not analyzed in the paper, and only included in a database later.

```
@fedt_experiment
def compare_knit_textures_from_dbs():
    textures_from_db = []
    for i in Parallel(range(306)):
        texture = design("texture{}.ks".format(i), GeometryFile, {
            'tile': '60x60 (fill gaps with knit stitches)',
            'edge': '12-stitch checkered knit/purl to edges',
            'eyelets': 'eyelets at centre and end of each edge (8 total)'
        })
        textures_from_db.append(texture)

    knitted = []
    weights = BatchMeasurements.empty()
    for texture in Parallel(textures_from_db):
        single_knitted = KnittingMachine.knit(texture)
        knitted.append(single_knitted)
        weights += Scale.measure_weight(single_knitted)

    analyze(weights.get_all_data())

    dims = ImmediateMeasurements.empty()
    photos = ImmediateMeasurements.empty()

    for single_knitted in Parallel(knitted):
        Human.post_process(single_knitted, "lay on sandpaper")
        photos += Camera.take_picture(single_knitted) # this is the one that went into the opacity measurement
        # (black pixel count). data was collected and used but not reported in the paper, only in a db later.
        dims += Calipers.measure_size(single_knitted, "unstretched length")
        # early samples measured with photos, then automatically with a reference object
        dims += Calipers.measure_size(single_knitted, "unstreteched width")
        for dimension in Series(['length', 'width']): # no concern about its being plastic, but
            # they did the following steps in consistent order anyway
            Human.post_process(single_knitted, f"hook onto the rig and load with 608g on {dimension}")
            photos += Camera.take_picture(single_knitted)
            dims += Calipers.measure_size(single_knitted, f"stretched {dimension}")

    analyze(dims.get_all_data())
    analyze(photos.get_all_data())
```

A.9.2 Experiment: Crowd-sourced KnitCarving Evaluation. This paper took knitted patterns that were made narrower either naïvely or using the KnitCarve algorithm and presented them to crowdworkers to compare with unmodified knit patterns.

Original Paper Description. We evaluated KnitCarving in a study with 200 crowd workers. Each worker rated the similarity of eight sets of two images using a scale from 1 to 10 (10 being the very similar). The first image was a photograph of a texture swatch and the second image was a photograph of swatch where the texture had some number of paths removed. We asked workers to compare the images based on a list of features (i.e. skew, size, number of whole repetitions, stretch, opacity) with simple descriptions of how they apply to knitted textures.

We divided workers into two groups: 100 workers compared swatches that were narrowed with KnitCarving and 100 work- ers compared swatches that were narrowed with a control algorithm that removes the right most loop from each course in a swatch. The control algorithm

may violate our Knittability properties, so some control swatches visibly unraveled (Figure 11). Workers were further grouped based on what portion of the repetition was removed.

Eight textures were selected from our texture data set: two knit/purl patterns, two twist patterns, two cable patterns, and two lace patterns. All eight textures were carved five times by one fifth of their repetition width. Within each texture category, one texture had a large repetition (in the third quartile), and the other had a small repetition (at least five, in the first quartile). Within these constraints, we randomly selected the textures.

We conducted seven independent-samples t-tests to compare the sum-total similarity score across (1) all textures per worker, (2-5) individual texture types, (6) wide repeat textures, and (7) narrow repeat textures. Across all tests, KnitCarving per- formed significantly better than the control algorithm. We summarize the results in Table 3.

Replication in FEDT.

```
@fedt_experiment
def crowdsource_knitcarve_comparison():
    textures = [GeometryFile(f) for f in ['knitpurl_large.knit', 'knitpurl_small.knit',
                                          'twist_large.knit', 'twist_small.knit',
                                          'cable_large.knit', 'cable_small.knit',
                                          'lace_large.knit', 'lace_small.knit']]

    base = {}
    carve = {}
    naive = {}
    for texture in Parallel(textures):
        unaltered = KnittingMachine.knit(texture)
        base[texture] = unaltered
        for carve_level in Series(arange(1,5+include_last)):
            carved = CustomProgram.knitcarve(texture, "remove {}/5ths of a repetition".format(carve_level))
            naive_reduced = CustomProgram.basicremove(texture, "remove {}/5ths of a repetition".format(carve_level))
            carved_phys = KnittingMachine.knit(carved)
            naive_phys = KnittingMachine.knit(naive_reduced)
            if not carve_level in carve:
                carve[carve_level] = {}
            carve[carve_level][texture] = carved_phys
            if not carve_level in naive:
                naive[carve_level] = {}
            naive[carve_level][texture] = naive_phys


    workerpool_size = 200
    ratings = ImmediateMeasurements.empty()
    textures = list(base.keys())
    for worker in Parallel(arange(1,workerpool_size+include_last)):
        carve_level = worker % 5 + 1
        comparator = naive if worker % 2 else carve # do I understand assignments correctly?
        for texture in Parallel(shuffle(textures)):
            CustomProgram.showToCrowdworker(comparator[carve_level][texture], worker)
            CustomProgram.showToCrowdworker(base[texture], worker)
            for aspect in Series(['skew','size','number of whole repetitions','stretch','opacity']):
                ratings += Human.judge_something(comparator[carve_level][texture],
                            "worker {} compare to {} based on {}".format(worker, base[texture], aspect))

    analyze(ratings.get_all_data())
```

Corrected Experiment. The authors informed us of the reasoning between the particular patterns that they selected to test (they corresponded to the chapters of a well-known book on knitting), and confirmed our understanding of worker assignment. They added that they did randomize the order in which swatches were presented to crowd workers: this was not specified in the text, but represented a good practice which they wanted encoded.

```
@fedt_experiment
def crowdsource_knitcarve_comparison():
    textures = [GeometryFile(f) for f in ['knitpurl_large.knit', 'knitpurl_small.knit',
                                          'twist_large.knit', 'twist_small.knit',
                                          'cable_large.knit', 'cable_small.knit',
                                          'lace_large.knit', 'lace_small.knit']]

    base = {}
    carve = {}
    naive = {}
    for texture in Parallel(textures):
        unaltered = KnittingMachine.knit(texture)
        base[texture] = unaltered
        for carve_level in Series(arange(1,5+include_last)):
            carved = CustomProgram.knitcarve(texture, "remove {}/5ths of a repetition".format(carve_level))
            naive_reduced = CustomProgram.basicremove(texture, "remove {}/5ths of a repetition".format(carve_level))
            carved_phys = KnittingMachine.knit(carved)
            naive_phys = KnittingMachine.knit(naive_reduced)
            if not carve_level in carve:
                carve[carve_level] = {}
            carve[carve_level][texture] = carved_phys
            if not carve_level in naive:
                naive[carve_level] = {}
            naive[carve_level][texture] = naive_phys


    workerpool_size = 200
    ratings = ImmediateMeasurements.empty()
    textures = list(base.keys())
    for worker in Parallel(arange(1,workerpool_size+include_last)):
        carve_level = worker % 5 + 1
        comparator = naive if worker % 2 else carve
        for texture in Parallel(shuffle(textures)):
            for knit_object in shuffle([comparator[carve_level][texture], base[texture]]):
                CustomProgram.showToCrowdworker(knit_object, worker)
            for aspect in Series(['skew','size','number of whole repetitions','stretch','opacity']):
                ratings += Human.judge_something(comparator[carve_level][texture],
                                "worker {} compare to {} based on {}".format(worker, base[texture], aspect))

    analyze(ratings.get_all_data())
```

A.10 Paper 10 : SPEERLoom

Adapted from Speer, et al., 2023 [77]. We skip an evaluation section in which the cost of various machines is compared, as no objects were fabricated to assess this.

A.10.1 Experiment: Weaving Quality. This experiment created several designs on different looms and measured characteristics of their final woven forms.

Original Paper Description. We evaluated SPEERLoom on weaving quality (number of warp yarns, EPI, and tension), warping efficiency (winding and threading time), weaving efficiency (shedding time), and cost requirements (Section 3). We compare these results to other Jacquard looms (two commercial looms (the TC2 [44] and the Jacq3G [25]), one DIY loom (Albaugh's loom [1])) and a shaft loom (the Ashford Katie Table Loom [16]). Warp winding and threading time were estimated based on the time taken by non-experts (SPEERLoom, Albaugh's loom, Jacq3g, and Ashford) and estimated by loom experts for the case of the TC2. Shedding time was timed for weaving basic patterns where 50-58% of the warp yarns were raised.

For evaluation of SPEERLoom's tension, we compare the measured tension to the Ashford shaft loom to ensure comparable variance in per-yarn tension to a standard two-warp beam tensioning mechanism. Additionally, we evaluate our tension model through empirical measurements, ensuring the equation estimates the final tension properly. We estimated μ_1 and μ_2 through averaged measurements of

tension at different stages of the system in Figure 4. We first varied N and measured T_1 and T_2 to find u_1 by averaging calculated values. We then varied T_2 and measured T_3 to find u_2 by averaging calculated values. All Values of tension were measured with a tensiometer for different stages of SPEERLoom's tensioning system and the Ashford loom's yarns.

Results

The results of the quantitative measurements taken for various looms are shown in Table 1 and discussed in the following sections.

Weaving Quality As shown in Table 1, SPEERLoom meets or exceeds the cloth properties (warp yarns and EPI) of other DIY looms. While SPEERLoom produces cloth of lower quality than commercial looms, we found that SPEERLoom is able to weave cloth meeting design requirements 3.3.1 and 3.3.2, specifically meeting the definition of quality cloth, as defined in Section 3.3.1 with regards to EPI, number of warp yarns, and tension.

Figure 7 shows different cloths woven on SPEERLoom. SPEERLoom is able to weave basic patterns as well as more complex Jacquard patterns (Req. 3.3.2). These patterns were woven at 12 EPI, giving a sufficient quality of cloth (Req. 3.3.1). While this EPI is not as fine as commercial looms, it is more suitable for classroom use than other DIY options(Table 1). The lower EPI of Albaugh's loom and other DIY looms results in lower fidelity of patterning, reducing the complexity and visibility of patterns produced cloth.

Design requirements in Section 3.3 require SPEERLoom to be suitable for novice and experienced weavers. Student weavers in a class taught with SPEERLoom (see Section 6) had a range of background experience with textiles, but were all able to accomplish weaving cloth on SPEERLoom. The students created custom patterns with matrix multiplication which can be clearly seen in Figure 10.

Figure 8 shows the tension on SPEERLoom and the Ashford loom, demonstrating that SPEERLoom's variance in tension is comparable to that of the Ashford loom. SPEERLoom has an average of ≈199g of tension with a standard deviation of ≈10g (Req. 3.3.1). The Ashford loom has ≈63g of average tension with a standard deviation of ≈28g. While the average tension of the two is different, this can be adjusted on either SPEERLoom (by changing the compression of the spring) or the Ashford loom (by adjusting both warp beams) in order to ft the needs of a specific weaving project. The variance of tension from yarn to yarn is something that cannot be easily adjusted on a two-warp beam style loom, as it would require re-winding the back warp beam. SPEERLoom's yarns however can be individually adjusted to create a more consistent tension across all yarns. This shows SPEERLoom's novel tensioning system is as consistent as an example two-warp beam tensioning mechanism, while saving time when adjusting individual yarn tension (Reqs. 3.3.1 and 3.1.1).

Replication in FEDT.

```
class SPEERLoomGUI(DesignSoftware, ConfigSoftware, ToolpathSoftware):
    def create_design(features: dict[str,object]) -> GeometryFile:
        return design('file.csv', GeometryFile, features, 'create the design in SPEERLoomGUI')

    @staticmethod
    def modify_design(design: GeometryFile,
                      feature_name: str, feature_value: str|int) -> GeometryFile:
        instruction(f"update {feature_name} to {feature_value} in SPEERLoomGUI")
        design.updateVersion(feature_name,feature_value)

    @staticmethod
    def create_config(defaults=dict[str,object]|None, **kwargs) -> ConfigurationFile:
        return design('file.csv', ConfigurationFile, defaults, 'create the configuration in SPEERLoomGUI')

    @staticmethod
    def modify_config(config: ConfigurationFile,
                      feature_name: str, feature_value: str|int) -> ConfigurationFile:
        instruction(f"update {feature_name} to {feature_value} in SPEERLoomGUI")
        config.updateVersion(feature_name,feature_value)

    @staticmethod
    def create_toolpath(design: GeometryFile,
                        config: ConfigurationFile, **kwargs) -> CAMFile:
        instruction("render the toolpath in SPEERLoomGUI")
        return design('file.serial', CAMFile, 'create the serial code in SPEERLoomGUI')

    @staticmethod
    def modify_toolpath(toolpath: CAMFile,
                        feature_name: str, feature_value: str|int) -> CAMFile:
        instruction(f"update {feature_name} to {feature_value} in SPEERLoomGUI")
        toolpath.updateVersion(feature_name,feature_value)
```

```
class SPEERLoom(FabricationDevice, metaclass=NameableDevice):
    uid = 'SPEERLOOM'
    num_loom_threads = 40
    @staticmethod
    def configure(settings: dict[str, object]):
        instruction(f"configure SPEERLoom like {settings}")

    @staticmethod
    @explicit_checker
    def fab(input_geometry: GeometryFile|None=None,
            configuration: ConfigurationFile|None=None,
            toolpath: CAMFile|None=None,
            default_settings: dict[str, object]|None=None,
            **kwargs) -> RealWorldObject:
        return fabricate({'geometry':input_geometry, 'configuration':configuration, 'toolpath':toolpath,
                          'non-default':kwargs},
                          'run the arduino code on the SPEERLoom')

    @staticmethod
    def create_object(non_default_settings: dict[str, object], instr: str|None) -> RealWorldObject:
        return fabricate(non_default_settings, instr)

    @staticmethod
    def describe(default_settings):
        return f"a custom-made SPEERLoom with settings {default_settings}"

class TC2Loom(FabricationDevice, metaclass=NameableDevice):
    uid = 'TC2LOOM'
    num_loom_threads = 440
    @staticmethod
    def configure(settings: dict[str, object]):
        instruction(f"configure loom like {settings}")

    @staticmethod
    @explicit_checker
    def fab(input_geometry: GeometryFile|None=None,
            configuration: ConfigurationFile|None=None,
            toolpath: CAMFile|None=None,
            default_settings: dict[str, object]|None=None,
            **kwargs) -> RealWorldObject:
        return fabricate({'geometry':input_geometry, 'configuration':configuration, 'toolpath':toolpath,
                          'non-default':kwargs},
                          'run the toolpath on the TC2Loom')

    @staticmethod
    def create_object(non_default_settings: dict[str, object], instr: str|None) -> RealWorldObject:
        return fabricate(non_default_settings, instr)

    @staticmethod
    def describe(default_settings):
        return f"an off-the-shelf TC2 Loom with settings {default_settings}"

class Jacq3GLoom(FabricationDevice, metaclass=NameableDevice):
    uid = 'JACQ3GLOOM'
    num_loom_threads = 120
```

```
    @staticmethod
    def configure(settings: dict[str, object]):
        instruction(f"configure loom like {settings}")

    @staticmethod
    @explicit_checker
    def fab(input_geometry: GeometryFile|None=None,
            configuration: ConfigurationFile|None=None,
            toolpath: CAMFile|None=None,
            default_settings: dict[str, object]|None=None,
            **kwargs) -> RealWorldObject:
        return fabricate({'geometry':input_geometry, 'configuration':configuration, 'toolpath':toolpath,
                          'non-default':kwargs},
                          'run the toolpath on the Jacq3G Loom')

    @staticmethod
    def create_object(non_default_settings: dict[str, object], instr: str|None) -> RealWorldObject:
        return fabricate(non_default_settings, instr)

    @staticmethod
    def describe(default_settings):
        return f"an off-the-shelf Jacq3G Loom with settings {default_settings}"

class AlbaughLoom(FabricationDevice, metaclass=NameableDevice):
    uid = 'ALBAUGHLOOM'
    num_loom_threads = 40
    @staticmethod
    def configure(settings: dict[str, object]):
        instruction(f"configure loom like {settings}")

    @staticmethod
    @explicit_checker
    def fab(input_geometry: GeometryFile|None=None,
            configuration: ConfigurationFile|None=None,
            toolpath: CAMFile|None=None,
            default_settings: dict[str, object]|None=None,
            **kwargs) -> RealWorldObject:
        return fabricate({'geometry':input_geometry, 'configuration':configuration, 'toolpath':toolpath,
                          'non-default':kwargs},
                          'run the toolpath on the Albaugh Loom')

    @staticmethod
    def create_object(non_default_settings: dict[str, object], instr: str|None) -> RealWorldObject:
        return fabricate(non_default_settings, instr)

    @staticmethod
    def describe(default_settings):
        return f"a custom-made Albaugh Loom with settings {default_settings}"

class AshfordLoom(FabricationDevice, metaclass=NameableDevice):
    uid = 'ASHFORDLOOM'
    num_loom_threads = 320
    @staticmethod
    def configure(settings: dict[str, object]):
        instruction(f"configure loom like {settings}")

    @staticmethod
```

```
    @explicit_checker
    def fab(input_geometry: GeometryFile|None=None,
            configuration: ConfigurationFile|None=None,
            toolpath: CAMFile|None=None,
            default_settings: dict[str, object]|None=None,
            **kwargs) -> RealWorldObject:
        return fabricate({'geometry':input_geometry, 'configuration':configuration, 'toolpath':toolpath,
                          'non-default':kwargs},
                          'run the toolpath on the Albaugh Loom')

    @staticmethod
    def create_object(non_default_settings: dict[str, object], instr: str|None) -> RealWorldObject:
        return fabricate(non_default_settings, instr)

    @staticmethod
    def describe(default_settings):
        return f"a custom-made Albaugh Loom with settings {default_settings}"

class Tensiometer:
    tension = Measurement(
        name="tension",
        description="The tension of a particular yarn at a location.",
        procedure="""
            Insert the yarn to be measured into the device, read the display.
            """,
        units="grams",
        feature="final stage yarn 0")

    @staticmethod
    def measure_tension(obj: RealWorldObject, feature: str=tension.feature, yarn: int='1') -> BatchMeasurements:
        instruction(f"Measure object #{obj.uid}, {feature}, yarn {yarn}.", header=True)
        instruction(Tensiometer.tension.procedure)
        return BatchMeasurements.single(obj, Tensiometer.tension.set_feature(f"{feature}, yarn {yarn}"))


@fedt_experiment
def evaluate_weaving_quality():
    pattern = GeometryFile('basicpattern.csv') # how many patterns?
    all_objects = []

    # tension
    u1_tensions = BatchMeasurements.empty()
    u2_tensions = BatchMeasurements.empty()
    for loom in Parallel([SPEERLoom, AshfordLoom]):
        woven = loom.fab(pattern)
        all_objects.append(woven)
        for N in Series(range(10)): # what is N? what values can it take? is this series or parallel?
            Human.post_process(woven, f"set N to {N}")
            for woven_thread in Parallel(range(0,loom.num_loom_threads)):
                for stage in Series(["T1", "T2"]):
                    u1_tensions += Tensiometer.measure_tension(woven, f"stage {stage}", woven_thread)
        for T2 in Series(range(10)): # what values did T2 take? is this series or parallel?
            Human.post_process(woven, f"set T2 to {T2}")
            for woven_thread in Parallel(range(0,loom.num_loom_threads)):
                u2_tensions += Tensiometer.measure_tension(woven, f"stage T3", woven_thread)

    analyze(u1_tensions.get_all_data())
```

```
    analyze(u2_tensions.get_all_data())

    # warp yarns and EPI
    quality = BatchMeasurements.empty()
    for loom in Parallel([SPEERLoom, Jacq3GLoom, AlbaughLoom, AshfordLoom, TC2Loom]):
        woven = loom.fab(pattern)
        all_objects.append(woven)
        quality += Human.judge_something(woven, "number of warp yarns")
        quality += Human.judge_something(woven, "ends per inch")

    analyze(quality.get_all_data())
```

Corrected Experiment. We confirmed that our first loop structure was correct, but had to modify the second one (adjusting T2) to reflect that it is a **while** loop structure, with the action performed as long as the thread hasn't slipped. We also got initial conditions and step sizes for the loops. For the weaving quality part, we added other patterns that were used for comparison.

Upon later reflection and a second look at the meeting notes and published paper, we realized that the tension/slipping section of this experiment actually treats the *SPEERLoom itself* as a fabricated object which is being tested, rather than the fabricated weaves being the objects tested. We did not make further corrections after agreement with the authors, but note this here and in the discussion.

```
@fedt_experiment
def evaluate_weaving_quality():
    pattern = GeometryFile('plainweave.csv')
    all_objects = []

    # tension
    u1_tensions = BatchMeasurements.empty()
    u2_tensions = BatchMeasurements.empty()
    for loom in Parallel([SPEERLoom, AshfordLoom]):
        for N in Series(arange(0, 1+include_last, .1)): # could be Parallel, but easier in Series
            loom.configure({"N": N})
            woven = loom.fab(pattern)
            all_objects.append(woven)
            for woven_thread in Parallel(range(0,loom.num_loom_threads)):
                for stage in Series(["T1", "T2"]):
                    u1_tensions += Tensiometer.measure_tension(woven, f"stage {stage}", woven_thread)

        T2 = 75 # g of tension, added with weights
        loom.configure({"T2": T2})
        slipped = False
        while not slipped and T2 <= 200:
            woven = loom.fab(pattern)
            for woven_thread in Parallel(range(0,loom.num_loom_threads)):
                u2_tensions += Tensiometer.measure_tension(woven, f"stage T3", woven_thread)
            T2 += 5
            loom.configure({"T2": T2})
            slipped = (Human.do_and_respond("adjust the weights", "did the thread slip?") == 'yes')

    analyze(u1_tensions.get_all_data())
    analyze(u2_tensions.get_all_data())

    # warp yarns and EPI
    quality_patterns = [pattern, GeometryFile('custompattern.csv'),
                        GeometryFile('overshot.csv'), GeometryFile('12weave.csv')]
    quality = BatchMeasurements.empty()
    for pattern in Parallel(quality_patterns):
        for loom in Parallel([SPEERLoom, Jacq3GLoom, AlbaughLoom, AshfordLoom, TC2Loom]):
            woven = loom.fab(pattern)
            all_objects.append(woven)
```

```
            quality += Human.judge_something(woven, "number of warp yarns")
            quality += Human.judge_something(woven, "ends per inch")
            # in some cases, these judgements came from images of fabricated objects instead of making them locally

    analyze(quality.get_all_data())
```

A.10.2 Experiment: Warping Efficiency. This experiment measured professionals and novices configuring various weaving machines to understand how efficient their warping was.

Original Paper Description. (See above for some setup details)

Warping and Weaving Efficiency

We found that SPEERLoom was more efficient than all other looms with regards to warping efficiency (Req. 3.1.1). Weaving efficiency on SPEERLoom exceeded that of other the other DIY loom (Req. 3.1.2).

SPEERLoom's tension system and creel were designed to eliminate the need for winding a back warp beam to satisfy requirement 3.1.1. This process can take approximately 3-5 hours depending on experience. SPEERLoom's creel was assembled in 20 minutes by a researcher. As shown in Table 1, SPEERLoom's per warp time is quicker than that of other looms satisfying requirement 3.1.1. This saves students hours of warping time for each warp pattern they wish to weave.

To further satisfy requirement 3.1.1, SPEERLoom is more efficient or as efficient as other looms with regards to threading. When measuring threading time, beginners threaded the Ashford loom [16], Jacq3g [25], and SPEERLoom. Threading time for the TC2 [44] was reported by Digital Weaving Norway. All threading time is reported per warp yarn to account for the difference in number of warp threads. An important aspect of the loom threading process for beginners is that a large amount of time is spent correcting mistakes such as threading yarn onto the wrong heddle or in the wrong order. During the threading process for each of the looms, users made several mistakes. The difference we observed was in the time it took to recover from those mistakes. Threading yarn in the wrong heddle for the Ashford loom [16] or Jacq3g [25] meant having to redo most of the threading process.

We observed novice student weavers threading SPEERLoom (see Section 6) and saw that when students made a mistake in threading their loom, it took them on the order of seconds to recover from their mistake. This was due to SPEERLoom's ability to control, place, and tension each warp yarn individually, which enabled the students to swap and re-tension the affected yarns without having to re-thread any other warp yarns (Req. 3.1.1). In this regard SPEERLoom is an improvement over the commercial and DIY alternatives.

Replication in FEDT.

```
@fedt_experiment
def evaluate_warping_efficiency():
    timings = ImmediateMeasurements.empty()
    for loom in Parallel([SPEERLoom, Jacq3GLoom, AlbaughLoom, AshfordLoom, TC2Loom]):
        if loom == TC2Loom:
            timings += Human.judge_something(loom, "how long it takes to set up")
        else:
            timings += Stopwatch.measure_time(loom, "set up warps")
    analyze(timings.get_all_data())
```

Corrected Experiment. While normally a stopwatch would be used to measure time, because of the extremely long setup time and the fact that people were setting up their own looms for this experiment, they just asked people how long it took (the number was in some cases more formal than in others).

```
@fedt_experiment
def evaluate_warping_efficiency():
    timings = ImmediateMeasurements.empty()
    for loom in Parallel([SPEERLoom, Jacq3GLoom, AlbaughLoom, AshfordLoom, TC2Loom]):
        timings += Human.judge_something(loom, "how long it takes to set up")
    analyze(timings.get_all_data())
```

A.10.3 Experiment: Weaving Efficiency. This experiment wove several designs on different weaving looms to characterize their shedding time.

Original Paper Description. (See above for some setup information)

As shown in Table 1, SPEERLoom's shedding time is on par with other looms and, while it is slower than commercial looms, still satisfies requirement 3.1.2. The increase in shedding time over commercial alternatives is a direct result of the reduction in cost by a factor of 30. As compared to a serial mechanism in Albaugh et al.'s loom, SPEERLoom has a much decreased shedding time. This decreased shedding time is a direct result of the increased cost for parallel actuation, but allows students to weave twice as fast. SPEERLoom's shedding time was not detrimental to students' ability to weave quickly. Students weavers in a collegiate class were able to weave the projects shown in Figure 10

over the course of a single week (see Section 6). This duration of weaving is comparable to other looms. Additionally, students commented that they feel as if they saved time weaving on SPEERLoom by having the opportunity to mathematically explore their cloth properties, allowing for faster testing without requiring weaving time.

Replication in FEDT.

```
@fedt_experiment
def evaluate_weaving_efficiency():
    pattern = GeometryFile("basicpattern.csv") # how many patterns? does shedding time depend on pattern?
    timings = BatchMeasurements.empty()
    for loom in Parallel([SPEERLoom, Jacq3GLoom, AlbaughLoom, AshfordLoom, TC2Loom]):
        physical_object = loom.fab(pattern)
        timings += Stopwatch.measure_time(physical_object, "shed the loom")

    analyze(timings.get_all_data())
```

Corrected Experiment. No corrections were required.