FabHacks: Transform Everyday Objects into Home Hacks Leveraging a Solver-aided DSL

Yuxuan Mei
ym2552@cs.washington.edu
University of Washington
Seattle, USA

Benjamin Jones
benjones@cs.washington.edu
University of Washington
Seattle, USA

Dan Cascaval
cascaval@cs.washington.edu
University of Washington
Seattle, USA

Jennifer Mankoff
jmankoff@cs.washington.edu
University of Washington
Seattle, USA

Etienne Vouga
evouga@cs.utexas.edu
The University of Texas at Austin
Austin, USA

Adriana Schulz
adriana@cs.washington.edu
University of Washington
Seattle, USA

Figure 1: We created FabHacks, a design system for "home hacks" built from repurposed everyday objects. The system is built on FabHaL, our domain-specific language for representing rigid fixture hacks. This solver-aided DSL is equipped with verification and solving functionality to help the user finalize their designs. Here we show two hacks, each with the set of everyday items to build it, the solved configuration from our system, and the design fabricated in the real world. Left: the birdfeeder hanging hack made of S-hooks, eyehooks, sticky hooks and a hanger. Right: the reading nook hack made of obstacle rings, toy ring links, S-hooks, turnbuckles and a hula hoop; the environment for the reading nook hack was scanned and calibrated with the PolyCam mobile application.

ABSTRACT

Storage, organizing, and decorating are important aspects of home design. Buying commercial items for many of these tasks, this can be costly, and reuse is more sustainable. An alternative is a "home hack," i.e., a functional assembly constructed from existing household items. However, coming up with such hacks requires combining objects to make a physically valid design, which might be difficult to test if they are large, require nailing or screwing to the wall, or if the designer has mobility limitations.

We present a design and visualization system, *FabHacks*, for creating workable functional assemblies. The system is based on a new solver-aided domain-specific language (S-DSL) called *FabHaL*. By analyzing existing home hacks shared online, we create a design abstraction for connecting household items using predefined connection types. We also provide a UI for designing hack assemblies that fulfill a given specification. FabHacks leverages a physics-based solver that finds the expected physical configuration of an assembly design. Our validation includes a user study with our UI, which shows that users can easily create assemblies and explore a range of designs.

SCF '24, July 7–10, 2024, Aarhus, Denmark

ACM ISBN 979-8-4007-0496-3/24/07.
https://doi.org/10.1145/3639473.3665788

CCS CONCEPTS

• **Computing methodologies** → **Graphics systems and interfaces**; • **Human-centered computing** → *Interaction design.*

KEYWORDS

domain-specific languages, fabrication, sustainability

ACM Reference Format:
Yuxuan Mei, Benjamin Jones, Dan Cascaval, Jennifer Mankoff, Etienne Vouga, and Adriana Schulz. 2024. FabHacks: Transform Everyday Objects into Home Hacks Leveraging a Solver-aided DSL. In *ACM Symposium on Computational Fabrication (SCF '24), July 7–10, 2024, Aarhus, Denmark.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3639473.3665788

1 INTRODUCTION

> In nature nothing is lost, nothing is created, everything is transformed. —*Antoine Laurent de Lavoisier*

Everyday life presents many challenges regarding our physical environment that we are constantly trying to solve, from common wear and tear (such as stovetop stains) to cluttered spaces (such as a messy desk). It is tempting to purchase the latest cleaning or organizational tools in a world of next-day delivery that bombards us with advertisements. However, buying still more products is wasteful, costly, unsustainable, and often unnecessary.

Instead, a thriving subculture is growing on the Internet of sharing "home hacks" that repurpose common household items into cost-effective and environment-friendly solutions. We analyzed the space of home hacks (full analysis in Appendix A) and found that we can divide them into two categories based on their functionalities. One category, like a blinds-cleaning tool made from binding tissues on tongs with rubber bands, makes creative reuse of a *single item* to change the shape or feel of an existing object, enabling better grasping or easier interaction. The other, like hangers linked with soda can tabs to make effective use of closet space, involves *assembling multiple items into a structure* that holds objects at a specific location and orientation relative to the environment. We term the latter "fixture hacks" because their goal is to build an assembly that holds a *target object* in a fixed *environment*. Our analysis found that rigid undeformed fixtures (i.e., composed of rigid parts combined but not deformed or modified destructively) are typically used in fixture hacks. Thus, our work focuses on this well-scoped subset.

Replicating existing fixture hacks at home might be straightforward, but inventing new hacks requires knowledge, insight, creativity, experimentation, and access to all parts. Furthermore, fixture hacks often involve multiple objects that interact mechanically, and gravity can affect a design's stability, making physical prototyping necessary to design a hack. However, physical prototyping is not always possible, not only for people with limited mobility, but also in situations where not all parts are available or prototyping would be costly or permanently alter one's home.

Our main insight is that despite the variety of objects used in rigid fixture hacks, these objects attach via eight common types of *connector primitives* (Figure 2). For example, the handle on a mug, the top hook on a hanger, and the handle on a basket can all be represented using a "hook" primitive (Figure 4). The connector primitive is thus a key concept in our system: these primitives abstract away the complex low-level geometry that is irrelevant to how users combine objects or to the overall assembly functionality.

This insight informed the design of *FabHaL* (FabHacks Language), the key contribution of our work. FabHaL is a solver-aided domain-specific language (S-DSL) for representing fixture hacks. By embedding the connection behavior and compatibility constraints for each pair of connectors into the solver (Section 4), we help users more easily explore hack designs within the domain's constraints.

On top of FabHaL, we build a novel design system and UI, *FabHacks*, for designing home hacks. The system lets users experiment virtually and simulate their designs under gravity. We validate our system through a user study, with results showing that users find FabHacks intuitive to use and is useful for exploring hack designs.

2 RELATED WORK

This work proposes FabHacks based on the FabHaL DSL that addresses the specific challenges posed by the domain of fixture hack design. We survey tools and recent work related to our approach.

CAD Tools for Assembly Design. Assembly design is important in manufacturing industries. Various tools have been developed for this task, including computer-aided design tools [Onshape 2023; SOLIDWORKS 2023]. We can use CAD tools to construct assemblies of parts using *mate constraints*, which define the relative orientation of two entities (part or surface) and the constraints on their degrees of freedom. However, modeling complex assemblies with existing CAD software requires a high degree of expertise.

First, mates are tricky to work with despite recent research [Jones et al. 2021] on providing mating suggestions. Multiple different mate types between two parts could appear to encode the same kinematics, only to be shown different later in the design process when another part is added that further constrains the existing degrees of freedom. Mate constraints are also not made for representing fixture home hacks. The everyday hacks that inspired our system (Figure 14) consist of many loose connections, such as a hook dangling over a rod, or a ring with a much greater radius than the hook it is attached to. Mates, usually single-origin coordinate systems with limited degrees of freedom, are more suitable for representing a mechanical assembly where parts fit snugly together, leaving only a few degrees of freedom for the overall assembly motion.

Second, if the object geometry comes in other formats (such as point clouds from scans, voxels, or inaccurate STLs), CAD users must create B-rep models from these inputs before they can specify mate connectors. FabHacks can accept any format and simply requires the geometry to be tagged with connector primitives. For example, for the reading nook hack in Figure 1, right, we scanned the room and used it as the geometry for the environment. We created an OnShape plugin (Figure 4, top) for tagging the connector primitives on geometry that comes in various formats.

Finally, performance analysis is also important during assembly design. Existing CAD tools are primarily concerned with analyzing the kinematics of mechanical assemblies and evaluating whether they achieve the desired concerted motion. In contrast, evaluating the performance of rigid fixture hacks that we focus on means measuring their stability as a hanging assembly under gravity. This type of simulation-based analysis is either completely separate from

Figure 2: We analyzed 24 rigid undeformed fixture hacks and extracted eight connector primitive types found on objects in those hacks; Table 4 in Appendix A documents the shape parameters we use to parametrize a primitive's geometry. We show each connector next to an example hack where it appears. The eight example hacks (left to right, top to bottom) are cup hanger (No.24), scarf organizer (No.22), toothbrush holder (No.7), charger holder (No.18), bathroom organizer (No.11), nonslip hanger (No.3), pants hanger (No.8), and soap bottle bag (No.1) as numbered in Table 2 in Appendix A.

current CAD design tools or exists with the CAD tool as part of a software suite that requires additional expertise to use.

Solver-Aided DSLs. DSLs have proven effective at abstracting expert knowledge and allowing non-experts to create valid designs, but they are, by definition, designed for a specific domain of applications. Several works [Jones et al. 2020; Zhao et al. 2020] have used DSLs for geometric modeling in specific domains, like simulated terrestrial robots and cuboid-based 3D shapes; they define a DSL and try to synthesize programs in it given specific objectives. DSLs can also be used to specify designs and fabrication plans for carpentry [Wu et al. 2019; Zhao et al. 2022], where we can use program synthesis techniques for design generation and optimization.

In this work, we propose a DSL (FabHaL) specifically for fixture hacks. FabHaL imitates the paradigm of *solver-aided languages*, where a user can partially specify a program (vastly reducing the search space) while leaving certain sections abstract (such as expressions or parameters) [Torlak and Bodik 2013]. An external solver is then invoked to concretize the partially specified program into a complete one, which can then be executed to verify the result. A FabHaL program is essentially a partial specification of a home hack design: a sequence of instructions to attach a specific connector primitive on one part to a specific connector primitive on another.

This paradigm has proven useful in domains in the programming languages community, such as program deobfuscation [Jha et al. 2010], synthesizing GPU kernels [Phothilimthana et al. 2019], and validating and planning biology experiments [Fisher et al. 2014]. Other applications include user interface designs [Hottelier et al. 2014] for resolving conflicts in the constraints of a layout design and mathematical diagrams [Ye et al. 2020] for automatically placing visual elements given user specifications. In our case, users can specify the skeleton of connections between primitives while leaving the precise placements of parts for a solver to fill in, and the solver could also provide feedback to users, such as informing them of whether a connection is valid.

Generative Design of Connectors. Existing works on modeling or creating connections involve generating new connection geometry. Koyama et al. [2015] propose a tool for automatically generating 3D printed structures given a user specification to hold or connect two objects. Hofmann et al. [2018] also generate connections between objects and support the specification of assembly information and constraints affecting the assembly, but they do not automate solving for those constraints. In addition, both works focus on manufacturing new parts, in contrast to our focus on exclusively reusing existing objects.

Sustainability in Design and Fabrication. Sustainability considerations have become increasingly prevalent in our everyday lives and in fabrication research [Yan et al. 2023]. Our work explores the general question of how to fabricate more sustainably. In this space, prior work explored how fabrication can reduce waste by using 3D printing to fix broken objects [Lamb et al. 2019; Teibrich et al. 2015] and reusing materials, such as plastic bags [Choi and Ishii 2021] and yarns [Wu and Devendorf 2020]. Other work augments existing objects with fabrication for repurposing [Davidoff et al. 2011; Guo et al. 2017; Ramakers et al. 2016], such as by generating structures for re-interfacing with robot arms, legacy physical interfaces, or appliances. Chen et al. [2018; 2015; 2016] use 3D printing to augment existing objects with additional functionality (some involving mechanisms), while Arabi et al. [2022; 2022] and Li et al. [2020; 2019; 2022] focus more on augmenting robots using everyday objects or mechanisms to help them manipulate objects.

Our research examines how to use rigid everyday objects of any shape without modifications to build a hanging fixture. Our work is distinctive in that we consider how *multiple* objects fit together into an assembly; the preceding work instead augments one specific object to allow robotic manipulation or to create a mechanism. (For example, none of the preceding work could be used to design the hanging birdfeeder in Figure 1, which uses several different parts.)

3 SYSTEM OVERVIEW

Consider as an example a novice user designing a birdfeeder to hang between two hooks using FabHacks (Figure 3).

Annotated Object Library. First, the user selects the parts they would like to repurpose into their home hack from the *Annotated Object Library*. The Library contains 3D models of a variety of rigid everyday objects, each annotated with the eight types of connector primitives we currently support. We call these annotated Library objects "parts." In addition to labeling regions of a part with a

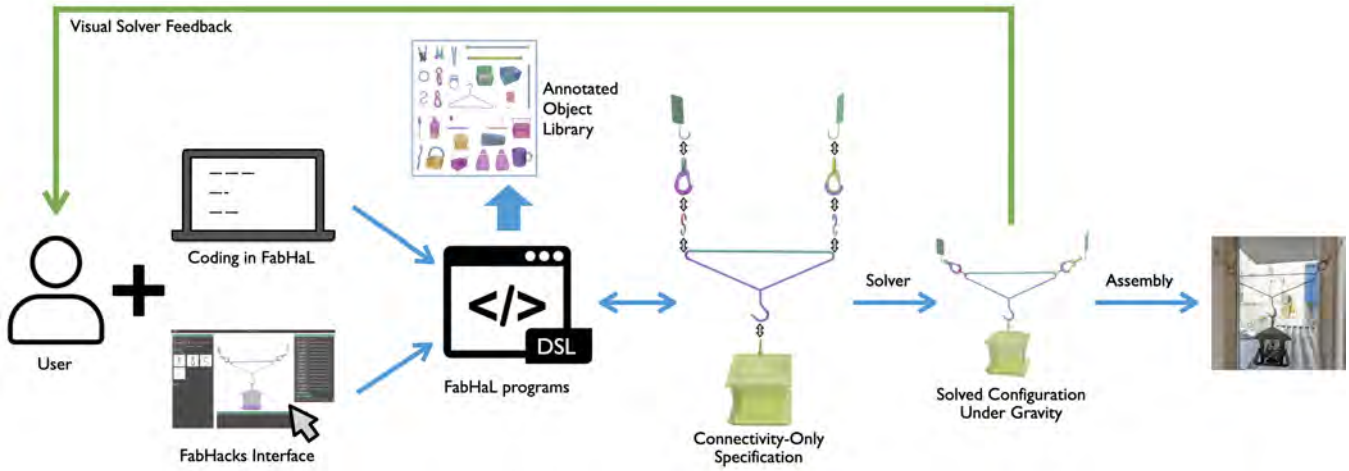


Figure 3: Overview of the FabHacks system. The user can either directly code in FabHaL or use the UI to create programs. FabHaL programs build on top of an annotated object library. The programs are connectivity-only specifications of a hack design, and the 3D configurations of the parts are completed by the automatic solver. Users can then get visual feedback from the program viewer and use the feedback to iterate on the design. When satisfied with the design, they can fabricate the hack in the real world.

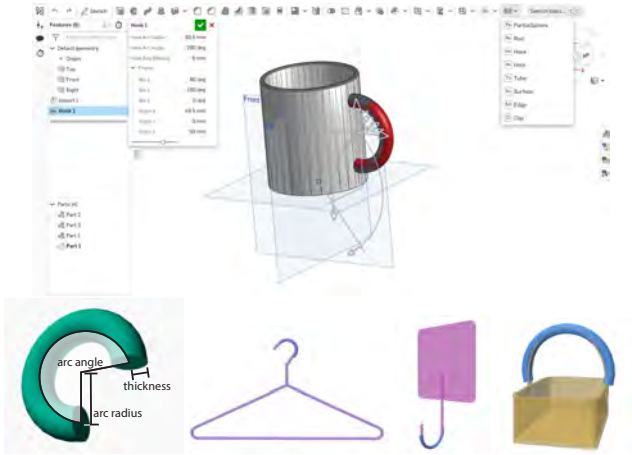


Figure 4: Top: the OnShape plugin for tagging 3D models with the eight connector primitives. Bottom: (left) an example showing how we defined the hook shape parametrically with its arc angle, arc radius, and thickness, and (right) three parts tagged with a hook primitive, each with different parameters.

connector primitive type (such as "hook"), the annotations include type-specific parameters needed to define the geometry of that primitive. For example, we show in Figure 4, bottom right, three example parts that have been annotated with a hook primitive, each parametrized to match the exact radius and thickness of the hook geometry in that part (bottom left).

We stress that the user does *not* typically need to do any 3D modeling or annotation themselves but rather can select parts from the predefined library. All examples in this work use a proof-of-concept library of 47 parts: 22 parts for to modeling the fixed environment or the target object to be held fixed in place by the hack, and 25 everyday objects rich in connector primitives for use as components of a home hack. To build this database, we extended the OnShape CAD modeling system's API to support part annotation. Our plugin (see Figure 4, top) lets users import a 3D model of a part and add connector primitives. When a primitive is added to a part, parameters are set interactively to ensure the connector aligns with the part. The Annotated Object Library can be extended to include custom parts that users want to include in their hack design, and we envision this to be part of a future community effort.

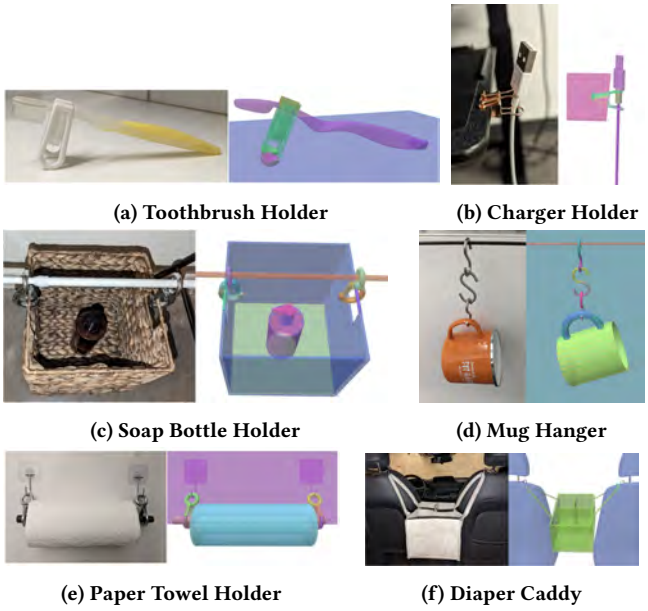
FabHaL. Next, the user assembles parts into a hack design using FabHaL, our solver-aided DSL. Users have two ways to interact with FabHaL to create hack designs: either directly writing programs in the FabHaL language, or using the FabHacks graphical interface to click on two connectors of two parts to connect them. When programming in FabHaL, users can also *parameterize* the programs; for example, they can specify that a hack should include a chain with an unknown number N of links and search over N for valid hack designs with the help of the solver (Section 6.2). In either case, note that the user need not write any kinematic constraints: these are inferred automatically by FabHacks from the part annotations. See Section 4 for more information on the FabHaL language.

Solver-aided Evaluation. Finally, the user asks FabHacks to realize the hack design in 3D space using a constrained optimization solver (Section 4.3). Our solver checks whether the part connections are feasible and, if so, relaxes the 3D positions of the parts under gravity and presents the final, solved configuration visually to the user. The solver also reports problems with the design to the user (such as infeasible connections or parts that would fall off the assembly if relaxed under gravity). Given this feedback, the user can iteratively improve the FabHaL program and solve again.

4 AN S-DSL FOR FABHACKS

We now introduce the S-DSL FabHaL for representing rigid fixture hacks. Figure 5 shows six example designs represented in the DSL.

The design of FabHaL was motivated by two factors. First, our analysis of home hacks (see Appendix A) influenced the language



(a) Toothbrush Holder (b) Charger Holder

(c) Soap Bottle Holder (d) Mug Hanger

(e) Paper Towel Holder (f) Diaper Caddy

Figure 5: Six hacks created by directly programming in FabHaL, with photos of fabricated designs and renderings of the corresponding programs in our viewer (see Appendix B).

design. We found that objects in home "fixture hacks" are typically connected via eight common shapes, which we term *connector primitives*, and define in Section 4.1.

Second, the language design is guided by our goal of using the DSL as a vessel for domain knowledge. We intend for this DSL to help users without prior experience in modeling or simulation to design fixture hacks. Therefore, being straightforward and succinct is an important desideratum. To achieve this, we choose to introduce a solver to complete a connectivity-only partial specification of the hack design, so the user need only specify (1) the configuration for a target object and its environment, and (2) which connector primitives to connect. We introduce the simple syntax and example usage in Section 4.2 and the solver in Section 4.3.

4.1 Connector Primitives

FabHaL includes eight primitive types: hook, rod, ring, tube, hemisphere, clip, edge, and surface, which can be assigned to a wide variety of objects (Section 3). We summarize the connectivity between these primitives in Table 1. Next, we explain how we model the connection behavior between pairs of connector primitives and the information associated with each primitive that the solver uses to verify and finalize the configuration of a hack design.

Figure 6: Three examples of rod-hook connections. Image on the right © Matt Kingston.

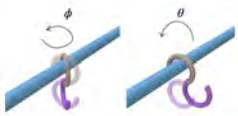
Connector Frames. Our analysis of home hacks (Appendix A) found that the connection behavior between parts is local to the pair of primitives that form the connection. Take as an example the rod-hook connection (see inset): a

Table 1: Pairs of primitives that can be connected. A checkmark means that connection is currently allowed by the DSL, and a light grey cell means it is not. We ignore the lower-triangular region (dark grey) since it is redundant with the upper-triangular one.

	hook	ring	hemi.	edge	rod	tube	clip	surf.
hook	✓	✓			✓	✓		
ring					✓	✓		
hemi.								✓
edge							✓	
rod						✓	✓	
tube						✓	✓	
clip								
surf.								✓

hook can slide along a rod and flex around it regardless of whether this rod is in a closet, a shower, or an ironing board. To represent such behavior mathematically so that we can formulate it as part of the solver's constrained optimization, we must first establish the concept of a `Frame`.

In FabHaL, `Frame` consists of a position vector (x, y, z) and yaw-pitch-roll intrinsic Euler angles. Frames can represent a single-origin coordinate system (similar to mates in CAD) or the 3D configuration of a primitive or a part. When frames are used to represent the connection points on primitives, we call them *connector frames*. The connector frames of a primitive can be computed from its base frame and shape parameters (obtained from the part annotations) and additional degrees of freedom specific to its type. For example, a hook primitive has two additional DoFs, θ and ϕ, parameterizing the location and orientation of the point of contact (see inset). In FabHaL, the DoFs and the information on how to use them to compute the parametric connector frames are associated directly with each connector primitive.



Alignment Offsets. When two primitives are connected, their connector frames need to be coincident in position, but the orientation may have some offset. Based on our analysis, this orientation offset is common to a pair of connectable primitives. For example, as shown in the inset figure, when a rod and a hook connect, their connector frames are offset by a rotation of $[180°, 0°, 90°]$ in yaw-pitch-roll intrinsic Euler angles. (Here the frames are intentionally placed to be not coincident at their origins to better display the orientation offset.) We call the offset rotation between two primitives' connector frames an *alignment offset*.

With the connector frames and alignment offsets defined for each pair of connector primitives, we can represent the connection behavior precisely with respect to the degrees of freedom associated with each primitive. Even with a small number of categories of connector primitives, we can capture a wide range of possible connections that appear in hacks. This set of primitives and associated alignment offsets is also easily extensible.

In addition to the theoretically allowed connectivity between primitives (Table 1), two primitives must be physically compatible before they can be connected. We encode two pieces of additional

information in connector primitives so that users need not reason about this lower-level detail.

Closed Primitives. Two primitives with no openings cannot connect because there is no valid motion path to create the connection. Among the eight connector primitives, the ring primitive is always closed. In addition, primitives that are not generally closed could be inaccessible in the context of the geometry of the part containing it. For example, the handle of the basket in Figure 4 (bottom-right) is tagged as a hook, which can connect to a ring according to Table 1. But as an integral part of the basket, it is part of closed geometry; thus, a ring primitive without an opening cannot connect to this hook. We allow tagging of individual primitives as *closed primitives* (e.g., the basket handle) when annotating parts for the Annotated Object Library, and our solver checks that designs do not attempt to connect two closed primitives.

Critical Dimensions. Primitives might not have sufficient physical space for a connection. For example, a one-to-one connection between a rod and a hook is possible only if the hook's hoop radius exceeds the rod's radius. For a multi-to-one connection between several hooks and tubes and a single rod, the hooks and tubes might fully occupy the length of the rod. In this case, no new connection could be made with the rod because there is insufficient space.

To track available physical space on primitives, we specify a *critical dimension* for connector primitives that can have multiple connections (i.e., the eight primitives except hemisphere and clip). The *available critical dimension* refers to the dimension of a primitive that gets occupied when a new connection is made between itself and another primitive. For example, the critical dimension of a rod is its length; when a hook connects to this rod, we reduce its available length by the hook's width. The hook's critical dimension—the hoop radius—is also reduced by the rod's radius.

4.2 Language Constructs and Hack Construction

To represent a hack, we must connect *parts* (annotated objects from the Annotated Object Library) using their connector *primitives*. These connected parts form a graph (Figure 7) that we call an *assembly* (i.e., a hack).

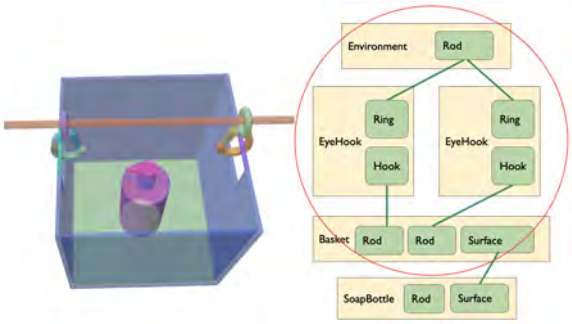


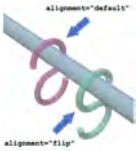
Figure 7: An assembly with a cycle: a basket is connected to a rod via two eyehooks, forming a cycle (in the red circle) between the basket and the environment. Yellow rectangles represent parts, and green ones represent primitives.

Two special parts in an `Assembly` are assumed to be fixed in place: the part representing the *environment* the assembly is attached to, and the *target part*, a part meant to be fixed relative to the environment and whose configuration is used as a target for the solver. For example, the clip in Figure 5a is rests on a table, supporting a toothbrush. The table is the environment, represented as a surface primitive with a fixed position and orientation. The toothbrush is the target part to be fixed above the table.

Our DSL exposes three operations needed to create an `Assembly`:

- `start_with(part, frame)`
- `end_with(part, frame)`
- `connect(part1.primitive, part2.primitive)`

`start_with` is used to specify the environment part with a fixed configuration (frame), and `end_with` specifies the target part's configuration (frame). `connect` takes two primitives as arguments regardless of order and determines whether each `Part` is already part of the `Assembly` or is newly introduced. It has two optional parameters: (1) `alignment` (either "flip" or "default") to indicate an orientation flip, e.g., a hook can hang on a rod coming from both sides of the rod, as shown in the inset; (2) `is_fixed`, a boolean value that indicates that the connection is formed by static friction and thus the degrees of freedom involved should be held constant during solver-aided evaluation (e.g., the design requires taping connectors together).



If both connected parts are already part of the assembly, this connection creates a cycle in the graph representation of the connected parts (see Figure 7). Not all `connect` operations will be physically realizable, and we discuss how the solver verifies whether a connection can be made in Section 4.3.

Figure 8 shows an example program in our DSL. This fixture hack hangs a basket with a round handle between two rods. In this program, we first initialize an `Assembly` and then the environment. Next, we use `connect` to add two eyehooks to the two rods by connecting the eyehook's eye to the rod. Finally, we initialize the target part and connect the hook part of the eyehooks to the handle of the basket.

4.3 Solver-aided Evaluation

The core advantage of FabHaL is its ability to simplify the representation of an `Assembly` to a graph of connected `Parts`, leaving to the solver the work of calculating the placement of parts.

In our solver, we model an `Assembly` using a reduced representation of a kinematic rigid body chain, a common approach in fields like rigid body mechanics and robotics [Featherstone 1983]. Except for the environment part configuration, which takes 6 DoFs (degrees of freedom), the remaining parts are represented with only the connection parameters (usually $1 \sim 3$D).

The solver handles both the simulation (4.3.3) and the pre-checks (4.3.1, 4.3.2) that check whether the `connect()` operations can be physically realized. We describe the pre-checks before we discussing the simulation of the assembly under gravity.

4.3.1 Verify Connect(). Two potential issues can arise when a connection is being made between two parts. First, a connection cannot be made between two primitives that cannot be joined according

```
assembly = Assembly()

# adding the starting environment
ENV_start = Environment({"part": TowelHangingEnv()})
start_frame = Frame([0,0,200], [0,0,-90])
assembly.start_with(ENV_start, start_frame)

# connecting the eyehooks to the starting environment
eyehook1 = HookEyeConnector()
eyehook2 = HookEyeConnector()
assembly.connect(eyehook1.hole, ENV_start.rod1, alignment="flip")
assembly.connect(eyehook2.hole, ENV_start.rod2)

# adding the target part and connecting the eyehooks to it
ENV_end = Environment({"part": RoundHandleBasket()})
assembly.connect(eyehook1.hook, ENV_end.hook)
assembly.connect(eyehook2.hook, ENV_end.hook)
assembly.end_with(ENV_end.hook, Frame([0,0,100], [-75,0,90]))
```

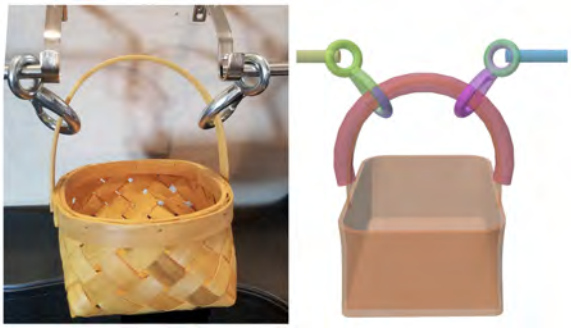


Figure 8: An example program in FabHaL (top), with the corresponding assembly solved for and physically reproduced (bottom left) and rendered by our system (bottom right).

to the connectivity table (Table 1), such as a rod to another rod, or when they are two *closed primitives*.

Second, the solver must check whether the *available critical dimension* of a primitive is sufficient for what is needed for a new connection. Based on the primitives' critical dimensions, we add constraints to the parameters of the connector primitive that has multiple connections. For example, when two hooks connect to the same rod, two sets of parameters that decide where along the rod the hooks connect are created. Suppose that the hooks each have widths w_1, w_2, the rod has length l, and the two connection parameters indicating the position of the hooks along the length of the rod are $t_1, t_2 \in [0, 1]$. Then, this "multi-connection" constraint $|t_1 - t_2| \cdot l \geq \frac{w_1+w_2}{2}$ is created and included in the solving process. We represent this constraint as a soft penalty, as follows:

$$C_{m,f} = 0 \text{ if } f \geq 0,\ C_{m,f} = f^2 \text{ if } f < 0,$$

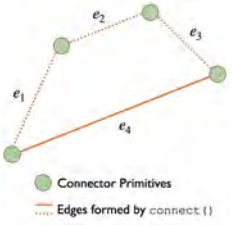
where $f = |t_1 - t_2| \cdot l - \frac{w_1+w_2}{2}$. We use the symbol C_m to represent the sum of all multi-connection constraint penalties.

4.3.2 Additionally Verify Connect() that Creates Cycles. A `connect` operation creates at least one cycle in the graph representation of the assembly if it is between two parts that are already part of the assembly (Figure 7). Such cycles require explicitly modeling constraints over the configurations of the parts being connected. Thus, we must also check whether we can find a set of values for the degrees of freedom that satisfy these constraints.

For a cycle, we model six constraints measuring the failure of the connector frames on the two `connected` primitives to match each other. An assembly with n cycles is feasible if valid values of the connection parameters exist along the cycles that satisfy $6n$ equality constraints of the form $f_i(\mathbf{x}) = \vec{0}, i \in [1..n]$, where $\mathbf{x}$ is a vector of all the DoFs in the assembly and $f_i(\mathbf{x}) \in \mathbb{R}^6$ measures the failure of the ith cycle to close up. We minimize the sum of constraint residuals $C(\mathbf{x}) = \sum_{i=1}^{n} \|f_i(\mathbf{x})\|^2$ subject to bound constraints on the DoFs, $\mathbf{x}_{\min} \leq \mathbf{x} \leq \mathbf{x}_{\max}$. We use the Powell method [2020] to minimize C and, if the solver succeeds in finding parameters with $C(\mathbf{x}) \leq 10^{-6}$, the assembly is considered feasible (and thus the `connect` successful). Since the success of the minimization depends on parameter initializations and can get stuck in local minima, we repeat the optimization T times starting from different random initial guesses. We terminate early if a solution is found. We observed that $T = 16$ works well in practice.

Geometric Quick Reject. Before we run a full optimization to find a system configuration that satisfies the connection constraints, we also utilize some precomputed information about the parts and primitives to perform a quick geometric check.

Our geometric check uses the triangle inequality: k line segments of length $\ell_1 \geq \ell_2 \geq \cdots \geq \ell_k$ cannot be arranged into a closed loop in 3D unless $\ell_1 \leq \sum_{i=2}^{k} \ell_i$. To apply this principle to our problem, we note that since each part i in a part cycle is rigid, we can bound the Euclidean distance $e_i \in [e_i^-, e_i^+]$ between the point where part i connects to parts $i-1$ and $i+1$. Because $[e_i^-, e_i^+]$ depend only on the geometry of the part and its two connectors involved in the cycle, not on the connection parameters, we can precompute these bounds for all the parts defined in our Annotated Object Library.



Consider the inset figure representing a design that has a cycle of 4 parts. To close the cycle, the following linear program over the distances e_i between connectors must be feasible:

$$\min_{e_i} 1 \quad \text{s.t.} \quad \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \geq \vec{0} \qquad (1)$$
$$e_i^- \leq e_i \leq e_i^+, \quad i \in [1..4].$$

Checking the existence of a set of distances e_i satisfying the bound constraints and triangle inequality then amounts to checking the feasibility of a set of linear inequality constraints. This can be solved in milliseconds by standard Python libraries, quickly rejecting impossible `connect` operations.

Stall Prevention. When we run the optimization, we put in measures for stall prevention. To halt optimization of C when the solver stalls, we pass a custom callback function to `scipy.optimize` that performs linear regression on $C(\mathbf{x}_i)$ for a sliding window of the last ten DoF iterates $\mathbf{x}_i$. We abort the optimization in failure if the slope of the fit line is less than 0.1 (i.e., the optimizer is not making much progress). This strategy gained us an additional 1.4x speedup on average for examples with cycles in Figure 5.

4.3.3 Solving the Assembly. After a valid assembly is constructed in FabHaL, the user can invoke the solver to find the values for the degrees of freedom in the system that bring the target part

as close as possible to its specified configuration while being in static equilibrium under gravity and respecting all cycle-closure and critical-dimension constraints.

This is a constrained optimization: we want to minimize the user objective subject to the balance of forces and torques on each non-environment part. Early experiments revealed that black-box nonlinear optimization was prohibitively slow at solving this problem and often failed to converge to a feasible local minimum. Therefore, we propose instead a two-step solver that first minimizes the user objective subject to all constraints being satisfied and then uses the optimized configuration as an initial guess for a simulation that relaxes the assembly to static equilibrium.

Step 1: Minimizing the User Objective. We use the Powell [2020] method to find a feasible configuration of the assembly that minimizes the user objective:

$$\mathbf{x}_{\text{feas}} = \arg\min_{\mathbf{x}} f_{\text{obj}}(\mathbf{x}) + \sigma(C_m(\mathbf{x}) + C(\mathbf{x})) \quad \text{s.t.} \quad \mathbf{x}_{\min} \le \mathbf{x} \le \mathbf{x}_{\max},$$

where $C_m(\mathbf{x})$ are the multi-connection constraints described in Section 4.3.1, $C(\mathbf{x})$ are the cycle-closure constraints in Section 4.3.2, and σ is a penalty parameter starting from $\sigma = 100$. If after optimization the constraint residual is not below 10^{-6}, we double σ and repeat the optimization, using $\mathbf{x}_{\text{feas}}$ as the initial guess. We repeat this process up to 5 times, which is usually sufficient to find $\mathbf{x}_{\text{feas}}$. If the constraint residual is still not below 10^{-6} after 5 times, we pass the best configuration found to the second step.

Step 2: Relaxing under Gravity. We use a physics solver to relax the assembly to an equilibrium state under its self-load, starting from the guess $\mathbf{x}_{\text{feas}}$. Let $q_i \in SE(3)$ represent the configuration of the ith part and $\mathbf{q} = \{q_i\}_{i=1}^{n}$ represent the configuration vector of the entire assembly. For an assembly with c total pairs of primitives connected, let $g_j(\mathbf{q}, \mathbf{x}) \in \mathbb{R}^6$ for $j = 1, \ldots, c$ be constraint functions encoding that each pair of primitives are connected with connection parameters $\mathbf{x}$.

To relax the assembly under gravity, we solve

$$\arg\min_{\mathbf{q},\mathbf{x}} E(\mathbf{q}, \mathbf{x}) \quad \text{s.t.} \quad \mathbf{x}_{\min} \le \mathbf{x} \le \mathbf{x}_{\max} \tag{2}$$

$$E(\mathbf{q}, \mathbf{x}) = \sum_i P_i(\mathbf{q}) + \sigma \sum_{j=1}^{c} \|g_j(\mathbf{q}, \mathbf{x})\|^2,$$

where $P_i(\mathbf{q})$ measures part i's gravitational potential energy and σ is a penalty parameter enforcing that connectors stay attached: we use $\sigma = 100$. We optimize Equation (2) using an active-set Newton's method [Nocedal and Wright 2006].

To demonstrate the two-step process, we take the hack design from Figure 7 as an example, which hangs a soap bottle from a rod using eyehooks and a basket. Both programs visualized in Figure 9 use the same target configuration specification for the soap bottle; thus, after the first step, the soap bottle is in a configuration that is closest to the target configuration. However, after the second step of relaxing under gravity, without a second eyehook to balance, the top row's design falls under gravity into a less desirable configuration compared to the bottom row's design.

During the physics relaxation, we also predict whether the assembly will fall apart due to connectors slipping off each other. To perform this analysis, we annotate each connection parameter for each primitive in our library with one of three tags:

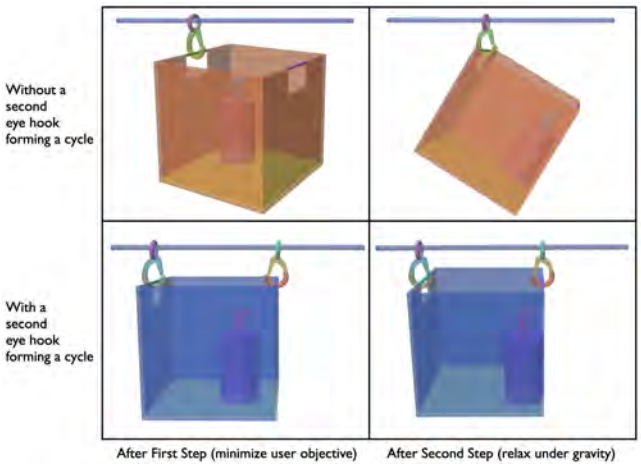


Figure 9: The top row shows the hack design without a second eyehook to balance the basket, and the bottom row shows the hack design with the second eyehook. The first column shows the intermediate results after first running the user objective minimization, and the second column shows the resulting configuration after the second step is run.

- `UNBOUNDED` parameters are periodic and should be allowed to "wrap around" from $x_{\max}$ to $x_{\min}$ during optimization. For example, for a ring that can rotate 360 degrees, the angle parameter specifying the rotation of the ring about its central axis is `UNBOUNDED`.
- `BOUNDED_AND_CLAMPED` parameters are used if the geometry of the primitive prevents the parameter from ever leaving the interval $[x_{\min}, x_{\max}]$. The position parameter of a rod along the bottom of a clothes hanger is an example of this parameter.
- `BOUNDED_AND_OPEN` parameters are used if exceeding the bounds of the parameter would cause the assembly to fall apart. The position parameter of a dowel rod, for example, is `BOUNDED_AND_OPEN`: hooks or rings that slide past the end of the dowel rod fall off the assembly.

At the end of optimization, for each `BOUNDED_AND_OPEN` parameter i, we check whether x_i is in the inequality constraint active set, i.e., whether x_i is equal to its maximum or minimum allowed value, and if so, whether $\nabla_{x_i} E$ points away from x_i's feasible interval. If so, we report to the user that the assembly falls apart.

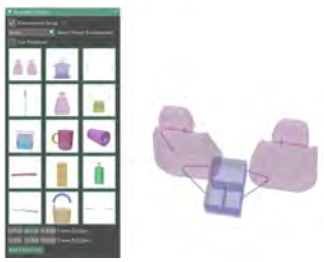
A hack assembly might have many different equilibrium states under gravity; our method above finds just one of them. For instance in the "Demo" assembly (inset Figure 10), the S-hook and ring could slide to either end of the hanger's rod depending on which end $\mathbf{x}_{\text{feas}}$ encodes they are closer to.

Figure 10: "Demo".

5 THE FABHACKS INTERFACE AND USER WORKFLOW

This section describes how users create FabHaL programs with the FabHacks UI. As introduced in Section 4.2, to construct a hack design, users first specify a starting environment with `start_with` and a target part's configuration relative to the environment with `end_with`, and then connect connector primitives on two parts with `connect`. Then, they can use `solve` to check design validity and solve for the configuration of their design under gravity. Based on feedback about whether a connection is valid and the visual feedback shown in the UI, users can choose to iterate on their design, as needed.

(a) The user interface.



(b) Step 1: the UI in the process of setting up the target part (a diaper caddy) relative to the environment (car seats).

(c) Step 3: The user clicks on "Run Optimization" and the inset shows the solved configuration.

Figure 11: Top: a screenshot of the user interface. Bottom: example interactions for Steps 1 (left) and 3 (right).

The UI consists of the *workspace region* and *three menus* (see Figure 11a). The left menu helps users during environment setup and for selecting parts to use in assembly design; the parts shown here are all from the "Annotated Object Library." The bottom menu helps users choose connector primitives of the selected part, and this is where the buttons for constructing the assembly appear. The right menu helps users solve for the assembly's final configuration.

We design the UI interactions to roughly correspond to the program construction process.

Step 1: Environment Setup. Users start by setting up the environments where this hack will be situated. Taking the diaper caddy hanging hack (Figure 5f) as an example, in Figure 11b, we (as users) have already added the car seats as the starting environment and are in the process of specifying the configuration of the target part (diaper caddy) relative to the environment. The desired configurations (position and orientation) of the environment and target part can be changed with sliders. This completes the environment setup, which corresponds to `start_with` and `end_with` in the program.

Step 2: Assembly Design. Next, users construct the assembly by specifying which connections to make. They can either select a part from the left menu to connect it to the assembly or select two connector primitives already in the assembly and specify that they should be connected. As defined in our DSL, `connect` might introduce unsatisfiable constraints and thus need to be verified. We provide two-way filtering based on the connectivity table (Table 1) to skip some pre-checks. For example, if a hook is selected in the menu, then only hook, ring, rod, and tube primitives will be enabled for selection in the workspace, and vice versa for a hook clicked on in the workspace. If a connection cannot be made because of failed pre-checks or because the solver cannot find a valid set of parameters that satisfy the constraints, specific feedback is provided to users. For example, if the connection to be made introduces a cycle but the cycle cannot be formed according to the geometric quick reject, the feedback will remind users that the two primitives might be too far away to be connected.

Step 3: Solving. After the environment and the target part are fully connected, the assembly is considered "valid." Users can then invoke `solve` with the button "Run Optimization," and the solver positions the parts such that changes in the configuration of the target part are minimized and the assembly is stable under gravity subject to any constraints in the design (Figure 11c). We note that this is not an interactive-rate step because the full physics-based solving can take up to a few minutes for complicated assemblies. More details on the runtime can be found in Table 5 in Appendix C.

After users see the visual or textual feedback on their design, they can choose to continue modifying it either with some backtracking via undo and redo buttons or by simply adding more parts. They then re-run the solve to view the updated design. For example, in the case of an unstable design, the solver would return feedback that one connection will fall apart under gravity, and the user might choose to modify that specific connection. In case the solver fails to solve (which happens rarely, as shown in Appendix C), the number of random initial guesses to try could be increased in the UI.

6 EVALUATION OF FABHACKS

We now discuss implementation details and how we evaluated our system on both direct programming with examples and programming via an interface with a user study.

6.1 Implementation

We implemented FabHaL as a shallowly embedded DSL with Python, i.e., it is embedded in the host language Python without its own abstract syntax tree. This allows the DSL to be used as a Python library and have access to common control structures from the host language for straightforward programmatic design generation. Python as the host language also facilitates easy integration with existing optimization and geometry processing libraries in

Figure 12: The parametrized assembly design and its four variations that most closely match the desired target part configuration given different ring sizes, with the photo (left) and simulated result (right). The parameter combination is indicated below the simulated result.

our solver implementation [Jacobson et al. 2018; Sharp et al. 2019; Virtanen et al. 2020]. The UI is also implemented in Python using polyscope [2019] with extended features from imgui.

6.2 Examples from Direct Programming and Programmatic Generation

When the user has an overall idea of the hack they want to create, they can directly code their design and virtually test it. Figures 1 and 5 show some example hacks created via direct programming.

As a DSL, FabHaL also lends itself well to programmatic generation of families of programs. This is helpful when the user knows roughly what parts to use but is not sure of how many. They can generate parametrized designs using the host language features, such as conditionals and loops, and use the solver to find the set of parameters from hundreds of variations that let the program best satisfy the given target part configuration.

As an example, suppose we are preparing for a trip to a summer camp with bunk beds. We would like to hang a clippable reading light at a certain distance from a hook on the top bunk bed so that it is sufficiently far away to not affect others in the same space and we can also reach the light's switch easily. We have a rough idea to use a hanger, extendable M4 turnbuckles, and rings of different sizes and chain them together into a fixture hack for this scenario. The parametrized design consists of a chain of n turnbuckle-ring pairs, with each turnbuckle extended by l millimeters and each ring of radius X (see Figure 12, leftmost). With this parametrization, we can programmatically generate a family of programs. If we already know the desired length and the size of rings that we have, we can use the solver to find the best parameters of n and l for a given ring size. Figure 12 shows four designs that match the target configuration, each corresponding to the four ring sizes ($X \in \{XS, S, M, L\}$) and selected from the 80 program variations with $n \in [1..4]$, $l \in [0, 45.7]$ (discretized into 20 values).

6.3 User Study with FabHacks Interface

Users with minimal coding experience can create designs more interactively with our UI. We evaluate how useful our system is for hack designs through a user study with ten participants. Participant ages ranged from 18 to 34, with CAD experience ranging from none to greater than 5 years. Participants reported their gender as Male (5), Female (3), Non-Binary (1) and N/A (1). We conducted the study in our lab with a laptop we provided, and each session took about an hour. Audio and screen capture were recorded during the study.

6.3.1 Method. After obtaining informed consent, we showed participants a tutorial on how to construct the "Demo" assembly (inset Figure 10) in FabHacks, and participants repeated the same steps.

Participants completed an **open-ended design task** using FabHacks where they were asked **to hang a bird feeder from two hooks and think aloud during the process**. The wall hooks (environment) and the bird feeder (target) were given. They had up to 30 minutes to create a design and were asked to come up with additional designs if time remained. We coded all designs as *feasible* or *infeasible* and then grouped them into categories based on similarity.

After the study, we asked participants to **answer three questions**: (1) "Can you tell us up to three things you would like to see us **keep** in the FabHacks tool?" (2) "Can you tell us up to three things you would like to **change** in the FabHacks tool?" (3) "Can you think of a **real-world change you would like to make** to your space in the office or at home that the FabHacks tool could help you with?" We grouped the responses into categories and discussed them until we reached a consensus.

6.3.2 Results and Discussion. Overall, our study shows that FabHacks is an efficient and intuitive way to construct hacks.

The 10 participants created 25 feasible designs, each unique though many used similar strategies. Twenty-three of these belong to one of four common strategies: (A) constructing symmetric

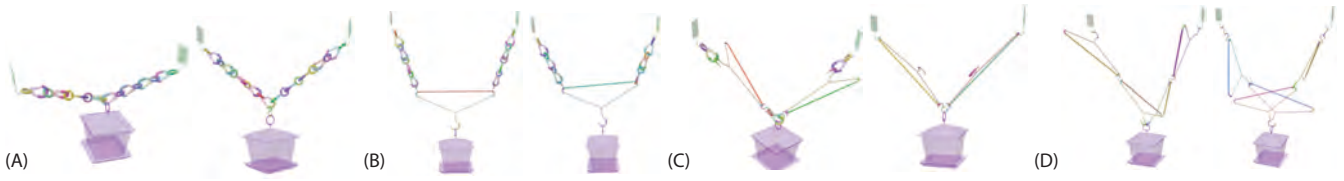


Figure 13: Examples of each of the 4 common design strategies (A-D) found by participants.

chains of small objects to anchor the birdfeeder between the two hooks (7 instances), (B) constructing two short chains, hanging a coat hanger upside-down between them, and dangling the birdfeeder from the hook of the coat hanger (6 instances), (C) hanging a coat hanger from each wall hook and anchoring the bird feeder where they meet (8 instances), and (D) chaining two coat hangers from each wall hook and connecting the birdfeeder in the middle of them (2 instances). While several participants discovered each pattern, no two were identical; they chose different types or numbers of parts to achieve similar construction or connected the same parts in different ways.

Looking at participants' answers to our three questions, we saw several important themes arise.

Question 1: FabHacks Keepers. First, multiple participants liked how "intuitive" the FabHacks interface was and praised its physics solver. One participant praised the "real-time realistic feedback" on connections, and another praised the "simplicity" of making connections in FabHacks.

Question 2: FabHacks Changes. At the same time, participants noted areas for improvement. For example, multiple users mentioned that "not knowing the reason a [connection] is failing [when validation is run] can be frustrating" and asked for a wider variety of undo and delete operations (a simple feature to add). Participants also made suggestions such as: having a constraint on the number of available pieces; better support for orienting, panning, and zooming; a tree diagram showing the connections; and better feedback about what is selected.

These critiques generally represent opportunities for improved user experience design rather than fundamental flaws with the mental model required to use our tool. For example, it would be possible to tell the user more specifically which part had a geometric flaw (e.g., being not long enough) and caused a connection to fail, or to visualize the configuration found by the solver with the failed connections highlighted.

Question 3: Real-world Use Scenarios. We also found that 6/10 participants had concrete ideas for how they would use FabHacks in their everyday lives, from a tree swing to outdoor lights to wall hangers to hang decorations or photos. Of four participants who did not see a use for FabHacks, one felt that the library needed to be expanded and account for things like weight because otherwise they would prefer testing the design directly; one felt they could more easily make a plan in their head; and two did not have an idea for how to use it.

Although participant comments suggest that there is room for improvement, the preceding feedback mostly focuses on things that can be solved with a larger library and iteration on the user experience. Future work could explore adding physical properties like weight and center of mass to our physics solver or letting users choose other materials for their parts.

7 LIMITATIONS AND FUTURE WORK

A limitation of our system is that it only considers rigid unmodified parts and makes simplifying assumptions on the part interactions. An important future direction is to extend the proposed abstractions to handle any hacks using soft parts, more complicated parts (e.g., with shifting centers of mass), or examples where the part shapes can be altered during assembly, such as a piece of wood that can be cut to size. For example, our parametric connections can be expanded to accommodate additional degrees of freedom, allowing for the representation of deformable objects or items that can change dimensions when cut.

More physical solvers could also be incorporated to handle deformable shapes and more complex part-part interactions. For example, more advanced methods could be employed to determine the physical compatibility of parts instead of relying on approximations through closed primitives and critical dimensions.

Our user study was designed to verify the usability of our system for creating valid hack designs without physical prototyping. Although we did not ask participants to build the hacks they designed, we retrospectively built five designs for which we had enough parts and verified their physical stability. In fact, it is evident from one participant's response on preferring testing directly that physical assembly and experimentation remains important despite our system's goal. In the future, understanding how users physically prototype—as well as how hacks get assembled, disassembled, and actually used—would greatly inform how the current system could evolve to become more usable in real life.

Our system also presents opportunities for automating how parts that make up the library are created. For example, it would be interesting to explore automated recognition and fitting of connection primitives given a 3D model of a part. A step further would be to automatically add a part to the library from LIDAR data or multiple images of an object, which would enrich the modeling power of the system and help bridge the reality gap.

Another promising opportunity is the complete automation of assembly design. By abstracting out eight common connector primitives and rules on their connection behaviors, our proposed DSL not only supports interactive design but has the potential to facilitate the generation of optimal designs under various objectives because it fundamentally reduces the search space. Automating the design of home hacks is a challenging task because it involves searching through discrete combinations of parts and finding suitable continuous parameters that meet the specifications. Our abstractions enable us to decouple this problem into a program synthesis task

nested with continuous optimization, which is performed by our solver. How to make program synthesis techniques usable in this context poses an interesting research problem.

FabHaL as a DSL could also benefit from the recent advances in large language models. Recent experiments that use LLMs for generating [Jain et al. 2023; Skreta et al. 2023] or completing programs [Piereder et al. 2024] in various DSLs show promising results. In preliminary experiments, we prompted GPT-4 [OpenAI 2024] to design a hack for hanging the birdfeeder with eyehooks, S-hooks, and hangers. While most attempts did not lead to a desirable design, GPT-4 was able to propose valid and near-valid designs (see inset). The inset-left shows a design created by GPT-4 that is very close to our design (Figure 1), and the inset-right shows a design that is not physically valid when evaluated with our solver (the S-hook will disconnect and fall off) but resembles the participant-created designs using strategy C (Figure 13). Exploring how to enable LLMs to create physically valid hack designs with FabHaL is an exciting research direction: the underlying solver could become useful in generating feedback based on the solve results to prompt LLMs to fix issues in invalid or undesired designs.

8 CONCLUSION

This work introduces a design system that helps users create fixture hacks built out of household items. Our solution features a new solver-aided DSL, FabHaL, which was inspired by our analysis of a collection of hacks. Using the solver-aided paradigm, FabHaL lets users create connectivity-only partial specifications of a hack, which simplifies the design of hacks to connecting primitives between parts. Our study showed that FabHacks can support end-user construction of hack assemblies and is intuitive to use. Participants identified potential opportunities for using FabHacks in everyday life, suggesting that the ideas presented in this work could inspire a new age of sustainable DIY design.

ACKNOWLEDGMENTS

We would like to thank our colleagues who pilot tested early versions of the tool. We also thank Sitong Zhou for helping with assembling the reading nook hack. Finally, we thank the anonymous reviewers for their thoughtful feedback that has greatly improved the work and Sandy Kaplan for her writing support. This work was funded by NSF 2017927, 2212049, and 2327136.

REFERENCES

Abul Al Arabi and Jeeeun Kim. 2022. Augmenting Everyday Objects into Personal Robotic Devices. In *SIGGRAPH Asia 2022 Emerging Technologies*. ACM, Daegu Republic of Korea, 1–2. https://doi.org/10.1145/3550471.3564763

Abul Al Arabi, Jiahao Li, Xiang 'Anthony Chen, and Jeeeun Kim. 2022. Mobiot: Augmenting Everyday Objects into Moving IoT Devices Using 3D Printed Attachments Generated by Demonstration. In *CHI Conference on Human Factors in Computing Systems*. ACM, New Orleans LA USA, 1–14. https://doi.org/10.1145/3491102.3517645

Marilyn Caylor. 2019. 75 super easy ways to organize your entire home. https://homehacks.co/easy-home-organizational-tips/

Xiang 'Anthony' Chen, Stelian Coros, and Scott E. Hudson. 2018. Medley: A Library of Embeddables to Explore Rich Material Properties for 3D Printed Objects. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–12. https://doi.org/10.1145/3173574.3173736

Xiang 'Anthony' Chen, Stelian Coros, Jennifer Mankoff, and Scott E. Hudson. 2015. Encore: 3D Printed Augmentation of Everyday Objects with Printed-Over, Affixed and Interlocked Attachments. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/2807442.2807498 event-place: Charlotte, NC, USA.

Xiang 'Anthony' Chen, Jeeeun Kim, Jennifer Mankoff, Tovi Grossman, Stelian Coros, and Scott E. Hudson. 2016. Reprise: A Design Tool for Specifying, Generating, and Customizing 3D Printable Adaptations on Everyday Objects. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 29–39. https://doi.org/10.1145/2984511.2984512 event-place: Tokyo, Japan.

Kyung Yun Choi and Hiroshi Ishii. 2021. Therms-Up!: DIY Inflatables and Interactive Materials by Upcycling Wasted Thermoplastic Bags. In *Proceedings of the Fifteenth International Conference on Tangible, Embedded, and Embodied Interaction*. ACM, Salzburg Austria, 1–8. https://doi.org/10.1145/3430524.3442457

5-Minute Crafts. 2022. 5-Minute Crafts — Learn. Create. Improve. https://5minutecrafts.site/

Scott Davidoff, Nicolas Villar, Alex S. Taylor, and Shahram Izadi. 2011. Mechanical hijacking: how robots can accelerate UbiComp deployments. In *Proceedings of the 13th international conference on Ubiquitous computing*. ACM, Beijing China, 267–270. https://doi.org/10.1145/2030112.2030148

R. Featherstone. 1983. The Calculation of Robot Dynamics Using Articulated-Body Inertias. *The International Journal of Robotics Research* 2, 1 (March 1983), 13–30. https://doi.org/10.1177/027836498300200102

Jasmin Fisher, Nir Piterman, and Rastislav Bodik. 2014. Toward Synthesizing Executable Models in Biology. *Frontiers in Bioengineering and Biotechnology* 2 (2014), 1–8. https://doi.org/10.3389/fbioe.2014.00075

Fiyaa. 2013. 15 Cord Management Life Hacks for No More Tangled Wires. https://www.amazinginteriordesign.com/15-cord-management-life-hacks-for-no-more-tangled-wires/

Anhong Guo, Jeeeun Kim, Xiang 'Anthony' Chen, Tom Yeh, Scott E. Hudson, Jennifer Mankoff, and Jeffrey P. Bigham. 2017. Facade: Auto-generating Tactile Interfaces to Appliances. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, Denver Colorado USA, 5826–5838. https://doi.org/10.1145/3025453.3025845

Megan Hofmann, Gabriella Hann, Scott E. Hudson, and Jennifer Mankoff. 2018. Greater than the Sum of its PARTs: Expressing and Reusing Design Intent in 3D Models. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–12. https://doi.org/10.1145/3173574.3173875

Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by manipulation for layout. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, Honolulu Hawaii USA, 231–241. https://doi.org/10.1145/2642918.2647378

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

Rijul Jain, Wode Ni, and Joshua Sunshine. 2023. Generating Domain-Specific Programs for Diagram Authoring with Large Language Models. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH 2023)*. Association for Computing Machinery, New York, NY, USA, 70–71. https://doi.org/10.1145/3618305.3623612

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ACM, Cape Town South Africa, 215–224. https://doi.org/10.1145/1806799.1806833

Benjamin Jones, Dalton Hildreth, Duowen Chen, Ilya Baran, Vladimir G. Kim, and Adriana Schulz. 2021. AutoMate: a dataset and learning approach for automatic mating of CAD assemblies. *ACM Transactions on Graphics* 40, 6 (Dec. 2021), 1–18. https://doi.org/10.1145/3478513.3480562

R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2020. ShapeAssembly: learning to generate programs for 3D shape structure synthesis. *ACM Transactions on Graphics* 39, 6 (Dec. 2020), 1–20. https://doi.org/10.1145/3414685.3417812

Karo. 2019. 25 IKEA Hacks to Keep Things Organized. https://craftsyhacks.com/ikea-organizing/

Yuki Koyama, Shinjiro Sueda, Emma Steinhardt, Takeo Igarashi, Ariel Shamir, and Wojciech Matusik. 2015. AutoConnect: computational design of 3D-printable connectors. *ACM Transactions on Graphics* 34, 6 (Nov. 2015), 1–11. https://doi.org/10.1145/2816795.2818060

Nikolas Lamb, Sean Banerjee, and Natasha Kholgade Banerjee. 2019. Automated Reconstruction of Smoothly Joining 3D Printed Restorations to Fix Broken Objects. In *Proceedings of the ACM Symposium on Computational Fabrication (SCF '19)*. ACM, New York, NY, USA, 3:1–3:12. https://doi.org/10.1145/3328939.3329005 event-place: Pittsburgh, Pennsylvania.

Jiahao Li, Meilin Cui, Jeeeun Kim, and Xiang 'Anthony' Chen. 2020. Romeo: A Design Tool for Embedding Transformable Parts in 3D Models to Robotically Augment Default Functionalities. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, Virtual Event USA, 897–911. https://doi.org/10.1145/3379337.3415826

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

Jiahao Li, Alexis Samoylov, Jeeeun Kim, and Xiang 'Anthony' Chen. 2022. Roman: Making Everyday Objects Robotically Manipulable with 3D-Printable Add-on Mechanisms. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 272, 17 pages. https://doi.org/10.1145/3491102.3501818

Jorge Nocedal and Stephen J. Wright. 2006. *Numerical Optimization* (2e ed.). Springer, New York, NY, USA.

Onshape. 2023. Onshape | Product Development Platform. https://www.onshape.com/en/

OpenAI. 2024. ChatGPT. https://chat.openai.com

Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence RI USA, 65–78. https://doi.org/10.1145/3297858.3304059

Christina Piereder, Günter Fleck, Verena Geist, Michael Moser, and Josef Pichler. 2024. Using AI-Based Code Completion for Domain-Specific Languages. In *Product-Focused Software Process Improvement (Lecture Notes in Computer Science)*, Regine Kadgien, Andreas Jedlitschka, Andrea Janes, Valentina Lenarduzzi, and Xiaozhou Li (Eds.). Springer Nature Switzerland, Cham, 227–242. https://doi.org/10.1007/978-3-031-49266-2_16

Lauren Piro. 2015. 8 Clutter Problems Solved by Shower Rings. https://www.goodhousekeeping.com/home/decorating-ideas/shower-curtain-rings-organizing

Raf Ramakers, Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2016. RetroFab: A Design Tool for Retrofitting Physical Interfaces using Actuators, Sensors and 3D Printing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, San Jose California USA, 409–419. https://doi.org/10.1145/2858036.2858485

Nicholas Sharp et al. 2019. Polyscope. www.polyscope.run.

Marta Skreta, Naruki Yoshikawa, Sebastian Arellano-Rubach, Zhi Ji, Lasse Bjørn Kristensen, Kourosh Darvish, Alán Aspuru-Guzik, Florian Shkurti, and Animesh Garg. 2023. Errors are Useful Prompts: Instruction Guided Task Programming with Verifier-Assisted Iterative Prompting. https://doi.org/10.48550/arXiv.2303.14100 arXiv:2303.14100 [cs].

SOLIDWORKS. 2023. 3D CAD Design Software | SOLIDWORKS. https://www.solidworks.com/

Jenny Stanley. 2021. 33 Brilliant Home Hacks Using Our 3 Favorite Items. https://www.familyhandyman.com/list/20-home-hacks-hangers-rubber-bands-and-cardboard-tubes/

Karen Sullivan and Jim Heumann. 2019. Karen and Jim's Excellent Adventure: Fiddly Bits: Making life on a small boat safer and more comfortable. http://karenandjimsexcellentadventure.blogspot.com/p/fiddly-bits.html

Alexander Teibrich, Stefanie Mueller, FranÃ§ois GuimbretiÃšre, Robert Kovacs, Stefan Neubert, and Patrick Baudisch. 2015. Patching Physical Objects. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 83–91. https://doi.org/10.1145/2807442.2807467 event-place: Charlotte, NC, USA.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry compiler. *ACM Transactions on Graphics* 38, 6 (Dec. 2019), 1–14. https://doi.org/10.1145/3355089.3356518

Shanel Wu and Laura Devendorf. 2020. Unfabricate: Designing Smart Textiles for Disassembly. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376227

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 (UIST '23 Adjunct)*. Association for Computing Machinery, New York, NY, USA, 1–3. https://doi.org/10.1145/3586182.3617433

Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Transactions on Graphics* 39, 4 (Aug. 2020), 144:144:1–144:144:16. https://doi.org/10.1145/3386569.3392375

Allan Zhao, Jie Xu, Mina Konaković-Luković, Josephine Hughes, Andrew Spielberg, Daniela Rus, and Wojciech Matusik. 2020. RoboGrammar: graph grammar for terrain-optimized robot design. *ACM Transactions on Graphics* 39, 6 (Dec. 2020), 1–16. https://doi.org/10.1145/3414685.3417831

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 Transactions on Graphics* 41, 3 (March 2022), 32:1–32:13. https://doi.org/10.1145/3508499

A ANALYSIS OF THE DESIGN SPACE OF HOME HACKS

The concept of "home hacking" covers a variety of topics. To better understand this design space, we first analyzed a collection of hacks and defined our problem domain, which informed our DSL design. Next, we go over research work relevant to home hacks design and existing tools that can be used to model home hacks.

We first gathered over 400 examples of hacks across 17 sources (including DIY blogs, videos, and individual designs from colleagues). After eliminating hacks that are repeated, or essentially the same but used under different scenarios, we selected 48 distinct hacks for further analysis.

Our analysis started with identifying each hack's functionality. This gives us two main categories: hacks that hold or fix some objects in a specific location and orientation – or *fixtures* (27 out of 48), and hacks which typically make creative reuse of a single item to change the shape or feel (material property) of an existing object to allow for better grasping or easier interaction, e.g., a pool noodle used to organize wires and hide them, cover sharp saw edges, or create padding on furniture corners for a baby-proof environment. In contrast to the latter category, fixture hacks usually involve multiple parts, e.g., several wire baskets chained together with S-hooks to organize items above a kitchen sink. And since their goal is to hold a part at a specific location and orientation relative to its environment, gravity will affect the design's stability. Thus, fixture design can be hard to reason with intuition and is well-suited as a computational design problem. Since deformable objects are difficult to model and simulate efficiently compared to rigid bodies, and it is unclear how end users can accurately specify manual modifications, we further limit the domain to rigid fixtures with only undeformed constituting objects because they are the majority of fixtures and present a well-scoped subset (24 out of 27). We show the complete set of 24 rigid (or can be seen as rigid) fixture hacks in Figure 14.

We then analyzed how the individual objects, which we call "parts", were connected in the subset of rigid undeformed fixtures. Although many different parts are involved in the hack examples, connections typically form between common types of *connector primitives*. These connector primitives connect in ways that are independent of the objects that they are part of. From our analysis, we extract the following categories of connector primitives (Figure 2): *rod*, *hook*, *ring*, *tube*, *clip*, *edge*, *surface*, and *hemisphere*.

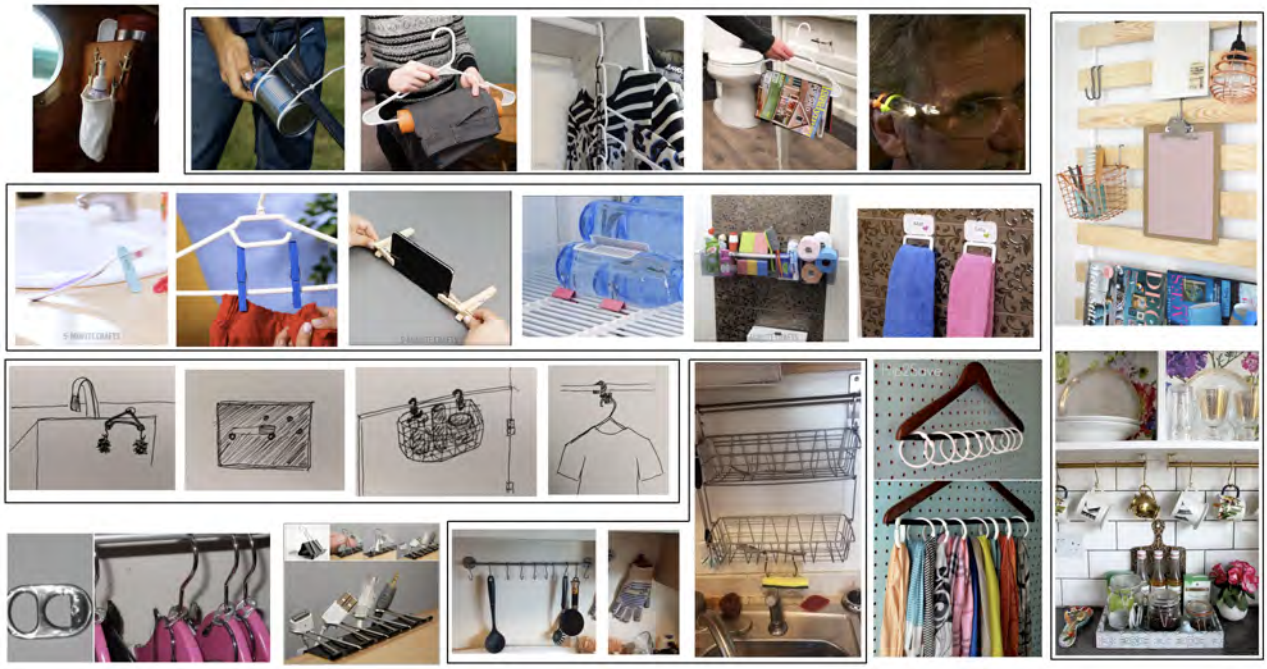


Figure 14: As ordered in the image, the hacks or hack groups in black boxes are from [Crafts 2022; Stanley 2021; Sullivan and Heumann 2019], one of the colleagues, [Caylor 2019; Fiyaa 2013], another one of the colleagues, [Karo 2019; Piro 2015]. We provide a short description for each hack and its source in Table 2.

Table 2: Hacks and their sources.

No.	Hack Source	Short Description
1	[Sullivan and Heumann 2019]	soap bottle bag
2	[Stanley 2021]	bottle holder on mower
3		nonslip hanger
4		hangers chained with rings
5		magazine on a hanger
6		glass light holder
7	[Crafts 2022]	toothbrush holder
8		pants hanger with clothes clips
9		phone holder from clothes clips
10		binder clips stoppers in fridge
11		bathroom organizer
12		tissue box towel hanger
13	Colleague A	dish sponge hanger
14		pen over pins on a board
15		shower essentials holder
16		hang clothes parallel to wall
17	[Caylor 2019]	hangers chained with soda can tabs
18	[Fiyaa 2013]	charger holder
19	Colleague B	kitchen tools rack
20		oven mitten holder
21		wire baskets chained with S-hooks
22	[Piro 2015]	scarf organizer
23	[Karo 2019]	bed slat as rack
24		cup hanger

Further analyzing the hack designs by looking at the parts and the connecting shapes of the parts, we determined the relationship between these connector primitives on whether they connect and how they align with each other when they connect. We summarized the common design patterns of how objects can be connected in Table 3. The eight connector primitives and their pairwise interactions become the basis for the design of the DSL and the underlying logic for assembly building in our system (see Section 4).

Table 3: Number of appearances in the 24 rigid undeformed fixture hacks (Figure 14) of each connection type. * means this connection type didn't appear but we deduced that it is compatible based on similar connection types. # means this connection type didn't appear in rigid fixture hacks but appeared in a non-rigid fixture hack. Light grey means this connection type did not appear or cannot be deduced. We ignore the lower-triangular region (dark grey) as it is redundant with the upper-triangular one.

	hook	ring	hemi.	edge	rod	tube	clip	surf.
hook	1	5			12	1		
ring					4	*		
hemi.								2
edge							3	
rod						2	5	
tube						#	*	
clip								
surf.								7

B PROGRAMS FOR GALLERY EXAMPLES

B.1 Toothbrush Holder

```
ASSEMBLY_toothbrush_holder = Assembly()

surface = Surface({"width": 400, "length": 400})
ENV_start = Environment({"surface": surface})
ENV_end = Environment({"toothbrush": Toothbrush()})

start_frame = Frame()
```

Table 4: We parametrize each primitive with its corresponding shape parameters, and show an example of the primitive.

Primitive	Shape Parameters	Example
hook	arc angle, arc radius, thickness	
ring	arc radius, thickness	
hemisphere	radius	
edge	width, length, height	
rod	radius, length	
tube	inner radius, thickness, length	
clip	width, height, base distance, open gap, thickness	
surface	width, length	

```
ASSEMBLY_toothbrush_holder.start_with(ENV_start, start_frame)

PART_clip = PlasticClip()
ASSEMBLY_toothbrush_holder.connect(PART_clip.hemisphere1, ENV_start.surface)
ASSEMBLY_toothbrush_holder.connect(PART_clip.hemisphere2, ENV_start.surface)
ASSEMBLY_toothbrush_holder.connect(ENV_end.rod, PART_clip.clip)
ASSEMBLY_toothbrush_holder.connect(ENV_end.hemisphere, ENV_start.surface)

end_frame = Frame([-10, -62.5, 50], [-65,0,0])
ASSEMBLY_toothbrush_holder.end_with(ENV_end, end_frame)
```

B.2 Charger Holder

```
ASSEMBLY_cable_holder = Assembly()

edge = Edge({"width": 100, "length": 200, "height": 1.5})
ENV_start = Environment({"edge": edge})
ENV_end = Environment({"cable": Cable()})

start_frame = Frame([0,0,150],[0,0,0])
ASSEMBLY_cable_holder.start_with(ENV_start.edge, start_frame)

PART_binderclip = BinderClip()
ASSEMBLY_cable_holder.connect(PART_binderclip.clip, ENV_start.edge, is_fixed=True)
ASSEMBLY_cable_holder.connect(ENV_end.rod1, PART_binderclip.ring1)
ASSEMBLY_cable_holder.connect(ENV_end.rod1, PART_binderclip.ring2)

end_frame = Frame([0,57.5,163], [0,0,0])
ASSEMBLY_cable_holder.end_with(ENV_end.rod2, end_frame)
```

B.3 Soap Bottle Holder

```
ASSEMBLY_soapbottle_holder = Assembly()

rod = Rod({"length": 500, "radius": 5})
ENV_start = Environment({"door": rod})
ENV_end = Environment({"soapbottle": SoapBottle()})

start_frame = Frame([0,0,500], [90,0,90])
ASSEMBLY_soapbottle_holder.start_with(ENV_start.door, start_frame)

PART_hookeye1 = HookEyeLeftS()
ASSEMBLY_soapbottle_holder.connect(PART_hookeye1.ring, ENV_start.door)
PART_basket = Basket()
ASSEMBLY_soapbottle_holder.connect(PART_basket.rod1, PART_hookeye1.hook)
PART_hookeye2 = HookEyeLeftS()
ASSEMBLY_soapbottle_holder.connect(PART_hookeye2.ring, ENV_start.door, alignment="flip")
ASSEMBLY_soapbottle_holder.connect(PART_hookeye2.hook, PART_basket.rod2)
ASSEMBLY_soapbottle_holder.connect(ENV_end.surface, PART_basket.surface)

end_frame = Frame([0,0,253], [0,0,180])
ASSEMBLY_soapbottle_holder.end_with(ENV_end, end_frame)
```

B.4 Mug Hanger

```
ASSEMBLY_mug_hanger = Assembly()

rod = Rod({"length": 500, "radius": 2})
ENV_start = Environment({"rod": rod})
surface = Surface({"length": 800, "width": 600})
ENV_wall = Environment({"wall": surface})
ENV_end = Environment({"mug": Mug()})

start_frame = Frame([0,0,200], [90,0,90])
ASSEMBLY_mug_hanger.start_with(ENV_start.rod, start_frame)
wall_frame = Frame([0,50,0], [90,0,0])
ASSEMBLY_mug_hanger.start_with(ENV_wall.wall, wall_frame)

PART_doublehook1 = DoubleHook()
PART_doublehook2 = DoubleHook()
PART_doublehook3 = DoubleHook()
ASSEMBLY_mug_hanger.connect(PART_doublehook1.hook2, ENV_start.rod)
ASSEMBLY_mug_hanger.connect(PART_doublehook2.hook2, PART_doublehook1.hook1)
ASSEMBLY_mug_hanger.connect(PART_doublehook3.hook1, PART_doublehook2.hook1)
ASSEMBLY_mug_hanger.connect(ENV_end.hook, PART_doublehook3.hook2)

end_frame = Frame([0,0,50], [-35,0,-90])
ASSEMBLY_mug_hanger.end_with(ENV_end.hook, end_frame)
```

B.5 Paper Towel Holder

```
ASSEMBLY_paper_towel_holder = Assembly()

ENV_start = Environment({"env": TowelHangingEnv()})
ENV_end = Environment({"paper_towel_roll": PaperTowelRoll()})

wall_frame = Frame([0,0,300], [0,0,0])
ASSEMBLY_paper_towel_holder.start_with(ENV_start, wall_frame)

PART_hookeye1 = HookEyeLeft()
PART_hookeye2 = HookEyeLeft()
PART_broomrod = BroomRod()
ASSEMBLY_paper_towel_holder.connect(PART_hookeye1.ring, ENV_start.hook1)
ASSEMBLY_paper_towel_holder.connect(PART_hookeye2.ring, ENV_start.hook2)
ASSEMBLY_paper_towel_holder.connect(PART_broomrod.tube, PART_hookeye1.hook, is_fixed=True)
ASSEMBLY_paper_towel_holder.connect(ENV_end.tube, PART_broomrod.tube)
ASSEMBLY_paper_towel_holder.connect(PART_hookeye2.hook, PART_broomrod.tube, is_fixed=True)

end_frame = Frame([53,0,160], [-90,-60,0])
ASSEMBLY_paper_towel_holder.end_with(ENV_end.tube, end_frame)
```

B.6 Diaper Caddy

```
ASSEMBLY_diaper_caddy = Assembly()

ENV_start = Environment({"backseat": BackSeats()})
ENV_end = Environment({"diaper_caddy": DiaperCaddy()})

start_frame = Frame([0,0,0], [0,0,0])
ASSEMBLY_diaper_caddy.start_with(ENV_start, start_frame)

PART_doublehook1 = DoubleHook()
PART_doublehook2 = DoubleHook()
PART_doublehook3 = DoubleHook()
PART_doublehook4 = DoubleHook()
ASSEMBLY_diaper_caddy.connect(PART_doublehook1.hook1, ENV_start.rod1)
ASSEMBLY_diaper_caddy.connect(PART_doublehook2.hook2, ENV_start.rod2)
ASSEMBLY_diaper_caddy.connect(PART_doublehook3.hook1, PART_doublehook1.hook2)
ASSEMBLY_diaper_caddy.connect(PART_doublehook4.hook1, PART_doublehook2.hook1)
ASSEMBLY_diaper_caddy.connect(ENV_end.hook2, PART_doublehook3.hook2)
ASSEMBLY_diaper_caddy.connect(ENV_end.hook1, PART_doublehook4.hook2)

end_frame = Frame([124.3,580,717.1], [-135.5,-40,20.5])
ASSEMBLY_diaper_caddy.end_with(ENV_end.hook2, end_frame)
```

C SOLVER RUNTIME

We used the example hacks as test cases and collected runtimes of various solver operations into Table 5 below. The same laptop used for the user study sessions, a MacBook Pro (2020) with M1 Chip and 8GB memory, was used for collecting these data. In the context of UI interactions, step 3's "Run Optimization" corresponds to **Avg. Full Solve Time**, and step 2's pre-checks correspond to **Avg. Quick reject Time** and **Avg. Constraints Checking Time**.

Based on the table, the runtime increases as the complexity (# Parts, # Cycles, # Params) of the example increases.

Table 5: This table shows the runtime averages of solver operations. The first two columns are the example's name and the corresponding figure. The next three columns records the number of parts, the number of cycles, and the number of parameters (i.e., the degrees of freedom of the connections) in the hack design. The runtimes in the next four columns are collected as averages over 10 runs: (1) Avg. Full Solve Time: the average time taken for a full solve of the example; (2) Avg. Quick Reject Time: the average time taken for the *geometric quick reject* pre-check; (3) Avg. # Initial Guesses: the average number of initial guesses needed for checking constraints satisfaction of the example; (4) Avg. Constraints Checking Time: the average time taken for checking constraints satisfaction. The ⋆ next to the reading nook example means that due to its complexity, instead of using randomly generated initial guesses, the initial guesses are computed from optimizing individual chains in the design, which takes around 2-3 minutes.

Example	Figure	# Parts	# Cycles	# Params	Avg. Full Solve Time (sec)	Avg. Quick Reject Time (sec)	Avg. # Initial Guesses	Avg. Constraints Checking Time (sec)
demo	Fig. 10	4	0	10	9.997	-	-	-
toothbrush holder	Fig. 5a	3	2	20	15.551	0.003	1.0	2.027
charger holder	Fig. 5b	3	1	8	2.014	0.001	1.4	0.743
soap bottle holder	Fig. 5c	5	1	17	26.937	0.003	1.0	1.536
mug hanger	Fig. 5d	6	0	11	5.188	-	-	-
bird feeder	Fig. 1 left	10	1	16	45.734	-	-	-
paper towel holder	Fig. 5e	5	1	15	20.479	0.002	2.3	8.251
diaper caddy	Fig. 5f	6	1	18	50.839	0.002	1.3	8.227
bathroom basket	Fig. 8	4	1	14	28.151	-	-	-
clip lights 1	Fig. 12	11	0	25	30.820	-	-	-
clip lights 2	(from	11	0	25	28.293	-	-	-
clip lights 3	left to	9	0	20	19.182	-	-	-
clip lights 4	right)	9	0	20	18.685	-	-	-
reading nook⋆	Fig. 1 right	19	2	65	742.279	0.0146	3.1	336.741