DOI: 10.1111/cgf.14046

COMPUTER GRAPHICS *forum*
Volume 39 (2020), number 6 pp. 408–425

Interactive Programming for Parametric CAD

Aman Mathur, Marcus Pirron and Damien Zufferey

Max Planck Institute for Software Systems, Germany
{mathur, mpirron, zufferey}@mpi-sws.org

Abstract

Parametric computer-aided design (CAD) enables description of a family of objects, wherein each valid combination of parameter values results in a different final form. Although Graphical User Interface (GUI)-based CAD tools are significantly more popular, GUI operations do not carry a semantic description, and are therefore brittle with respect to changes in parameter values. Programmatic interfaces, on the other hand, are more robust due to an exact specification of how the operations are applied. However, programming is unintuitive and has a steep learning curve. In this work, we link the interactivity of GUI with the robustness of programming. Inspired by programme synthesis by example, our technique synthesizes code representative of selections made by users in a GUI interface. Through experiments, we demonstrate that our technique can synthesize relevant and robust sub-programmes in a reasonable amount of time. A user study reveals that our interface offers significant improvements over a programming-only interface.

Keywords: CAD, modelling, human–computer interfaces, interaction, solid modelling

ACM CCS: • Computing methodologies → Model development and analysis; Modelling methodologies; Graphics systems and interfaces

1. Introduction

Parametric computer-aided design (CAD) is a modelling paradigm where objects are represented by the sequence of operations in their design, rather than their final forms. This enables users to change design parameters, which re-executes the sequence of operations, and produces a new final object fitting a different use case. Since its introduction in PTC Pro/ENGINEER in 1988 [Yar13], parametric CAD has become the industry standard for 3D design. Moreover, due to advances in quality and availability of 3D printing, it is no longer exorbitantly more expensive to manufacture niche variants of base objects. Manufacturing is becoming increasingly decentralized, and we can expect people to manufacture different objects fitting specific use cases rather than mass producing the exact same object. This has re-ignited wide interest in parametricity of designs.

There are two interfaces to parametric design: programming and GUI. Though most popular CAD tools such as Autodesk Fusion 360, OnShape, SolidWorks, PTC Creo and FreeCAD are GUI-based, these interfaces are far from perfect. GUI interfaces are interactive: users directly select elements they want to modify, and then apply operations to these. This interactivity, however, is a double-edged sword. The specification of which elements users select is 'latent', i.e. not explicitly known. In GUI, this specification is never made explicit. For many computer applications, this is an acceptable compromise. However, for parametric CAD, this is a severe limitation. For instance, what happens when a designer modifies 1 of 18 edges in a design, and then due to a parameter change, there are 36 edges? CAD interfaces need to recompute the design and present the final object reflecting the 'design intent'. However, due to the under-specification of GUI, this is an impossible task. Therefore, GUI-based CAD interfaces often end up with modifications that do not match the design intent. This problem is well known and many heuristics have been proposed in prior work [Che95, CH95, CCH96, Kri95, AMP00]. In our observations, we find that robustness issues are still quite ubiquitous.

Before we proceed, let us point out that a (near) perfect solution to the robustness problem already exists: programming. Programmes are essentially precise specifications. Programmes clearly identify which elements need to be modified, for all valid combinations of parameter values. Depending on the Application Programming Interface (API), this is done by semantically selecting elements that possess explicit features, using loops (OpenSCAD [Kin19] and Open CASCADE [OPE19]), or declarative queries (CadQuery [Par19a], FeatureScript [Fea20] and Scadla [Zuf19]).

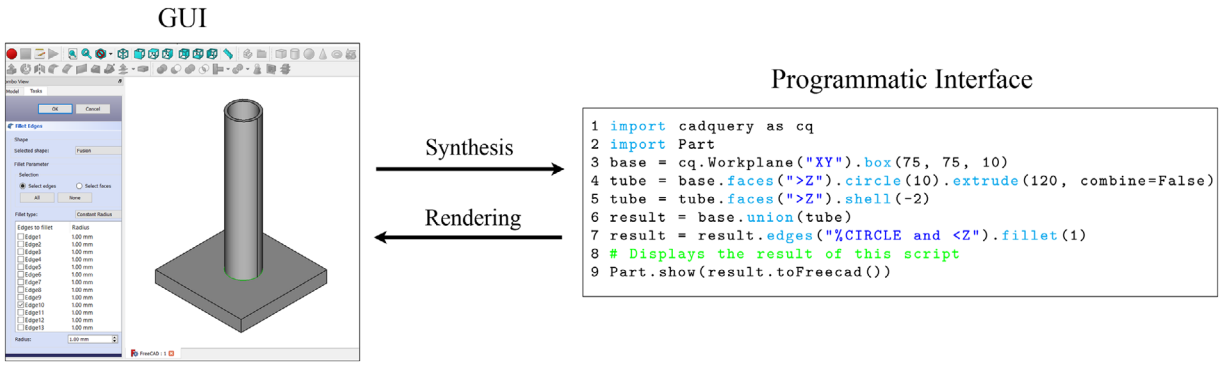
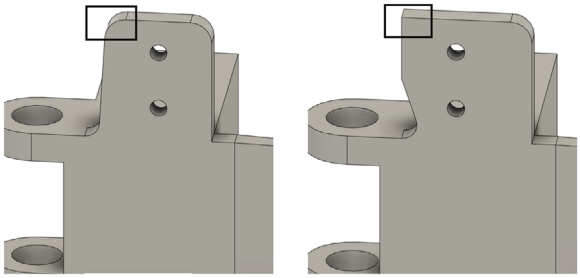


Figure 1: *We present a technique to bridge GUI and programmatic interfaces for CAD. With the insight that GUI is intuitive but brittle, and programming is robust but difficult, we link the two by synthesizing programme segments representative of GUI-based operations.*



(a) *Initial design.* **(b)** *After perturbation.*

Figure 2: *The side bracket design fails to fillet the intended edges (see black box) after the length of the top edges is increased.*

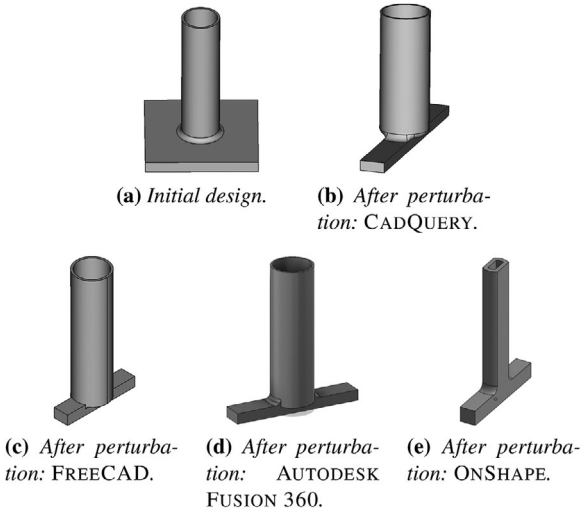
However, writing complicated semantics representative of simple GUI interactions has proven to be an unfortunate barrier-to-entry. As a result, programmatic interfaces are much less popular than their GUI-based peers. In this work, we propose bringing the ease of use of GUI with the robustness of programming, and present a system that uses GUI interactions to automatically synthesize code.

1.1. Motivation

We now motivate towards the need for a better resolution to GUI-based CAD's robustness problem, and why our technique offers a tangible solution.

Design intent. GUI-based CAD tools often fail in capturing design intent. This is true even for curated designs on professional-grade CAD software. We present an example from PROJECT EGRESS [Bar19], a large collaborative project for building a replica of the Apollo 11 space hatch, designed in AUTODESK FUSION 360 . Figure 2 shows a simple perturbation on the capsule side bracket module. The perturbation causes more edges to appear in the resulting object as before, which, in turn, causes the fillet (rounding) operation to fail. Clearly, the CAD tool fails to capture design intent (the top-left edge is not filleted any more).

Different and unknown heuristics. Different tools use different heuristics to try and resolve ambiguity. Unfortunately, these heuristics are usually not known *a priori*, and designers often observe different unexpected results on different tools. Consider the simple design in Figure 3 (inspired by a dowel-end cap design [Bri19] on THINGIVERSE, an online repository of several CAD projects). We re-designed this example on CADQUERY, a programmatic interface for a baseline specification of what we want. Using the same steps, we also designed this on FREECAD, AUTODESK FUSION 360 and ONSHAPE, all of which are GUI-based. On the initial design, we change the dimensions of the rectangular base. All tools under test yield different results. FREECAD fillets one edge on the right of the connection between the hollow rod and the base. AUTODESK FUSION 360 fillets two opposite sides of the connection. ONSHAPE changes the shape of the hollow rod. Reducing the width of the rectangular base leads to there being more edges than before while performing the fillet operation. FREECAD does not use complex heuristics. Internally, it uses names for all elements in the design, and operations are applied to elements by name. So, when the design changes, the names are re-evaluated and the fillet is applied to the element(s) with the same name. As is evident, this often does not



(a) *Initial design.* **(b)** *After perturbation:* CADQUERY. **(c)** *After perturbation:* FREECAD. **(d)** *After perturbation:* AUTODESK FUSION 360. **(e)** *After perturbation:* ONSHAPE.

Figure 3: *A simple dowel end-cap design and results of changing dimensions of the base in various popular CAD tools.*

work well. Unfortunately, due to the closed-source nature of Autodesk Fusion 360 and OnShape, we cannot be certain of their ambiguity resolution heuristics.

Programming and GUI. In the context of the two examples provided above, writing a programmatic specification for each step in the design process has two advantages:

(i) it makes the design intent explicit, and
(ii) it removes the need for ambiguity resolution heuristics.

Current GUI-based CAD tools already recognize some other advantages of programming. For example, most tools support Macros, which encapsulate GUI operations into executable chunks. Additionally, the popularity of procedural design has introduced dataflow programming to CAD (examples are Grasshopper3D and Dynamo by Autodesk), where GUI-based designs are connected to dataflow components. However, most major CAD tools today offer an exclusively GUI-based design interface, backed by opaque heuristics for capturing design intent and for ambiguity resolution.

Inspired by recent work on integration of programming and direct manipulation for vector graphics [CHSA16], we propose bridging programmatic and GUI-based CAD. We do this by synthesizing programmatic queries representative of designers' GUI selections and operations. We use a modified decision tree algorithm for this, which prefers short queries that *generalize*. Moreover, we use techniques from programme analysis to synthesize queries at the relevant line number, using the relevant intermediary object, and when applicable, using programme variables and scope. Based on experimental evidence, we find that our technique synthesizes queries that are robust, and it does so fairly quickly (taking at most a tenth of a second) for various samples. A user study reveals that our interface is faster, more accurate and preferable to a programming-alone interface.

1.2. Our contributions

The main contributions of this work are as follows:

(i) We identify bridging programming and direct manipulation interfaces as a possible solution to brittleness of GUI-based interfaces for parametric CAD.
(ii) We propose an algorithm for automatic synthesis of relevant selection queries from GUI-based interactions.
(iii) We have implemented our algorithm in a prototype built on top of CadQuery and FreeCAD .
(iv) We validate our approach using various designs, application scenarios and a user study.

2. Related Work

Robustness of parametric CAD. GUI-based CAD tools initially suffered from a severe robustness challenge of not being able to persistently and uniquely identify geometric entities of a design. This led to research on heuristics to resolve this, both, during design time [Che95, CH95, CCH96] and during re-evaluation (under changed parameter values) [Kri95, AMP00]. Robustness in design has been explored using geometric constraints [BH11], and explicit user input [Gir01, PPG96]. These works successfully provide persistent and unique names to geometric entities (termed the 'naming problem'). However, ambiguity resolution is done differently on different CAD tools, and because this information is usually opaque to users, interchange of designs between different CAD tools is difficult [MH05]. Prior work has focussed on providing unique names to geometric entities, and mapping these names to different geometric entities when there are parameter changes. To the best of our knowledge, no prior work has looked into linking GUI with programming.

Parametricity in CAD. The importance of parametric CAD, in the context of the modern design and fabrication landscape, has greatly increased. There has been work on systematic exploration of large parametric design spaces [LSL*19, SWG*18, SSM15], using parameters for quality control [WDRAJ16], and interactive modification of designs [OLFB18]. There are extensions to the original idea of modelling by example [FKS*04] to ensure that the resulting design is fabricable [SSL*14], and on generation of fabrication constraints at design time [LVLR19]. Interestingly, research focussed on exploring parametricity of designs has either chosen programmatic back-ends, or highly curated designs. This is also true for the industry. Thingiverse [Mak19], for example, only supports designs with a programmatic back-end for its customizable designs section.

CAD synthesis. For modelling based on Constructive Solid Geometry (CSG), which involves creation of shapes using binary operators on solids, there has been recent work on synthesizing the underlying CSG-tree or programmatic representation [DIP*18, NWP*18]. Although these techniques can synthesize CSG-trees of various 3D models, CSG representation is less expressive than Boundary Representation (B-Rep), the representation we and several modern CAD tools use. We view these works as complimentary to ours in that these try to uncover the underlying semantic representation of a rendered 3D model, whereas we try to uncover the semantic representation of user activity *during* the design process.

Interactive programming. Our work shares the same motivation as recent work on synthesizing programme repairs from GUI-based edits on the programme output [CHSA16, Vic12, MKC18]. Prior works focus primarily on small programme repairs achieved either through constraint solving, or some well-defined transformation rules. There is also work on synthesizing simple programmes from GUI-based target outputs [HC16]. The synthesized programmes, however, are very simple transformations of GUI operations to a textual representation.

Programme synthesis. Programme synthesis is already a mature field, from the first major use cases in data extraction and synthesis of spreadsheet macros [LG14, BGHZ15], to more recent work such as the synthesis of SQL queries [WCB17]. Our use case for CAD shares many of the same goals as other synthesis approaches, i.e. we want to synthesize small programmes and we want to do this fairly quickly. A unique challenge we tackle, unlike some other programming-by-example [Gul16] based approaches, is that we work with only one *example* (i.e. a user's direct manipulation action).

Our underlying synthesis approach needs to be fast enough to be unobtrusive to the GUI-based interaction. In this context, we borrow ideas from syntax-guided synthesis (SyGuS) [SL08, ABJ*13], as our abstract grammar restricts the syntax of the programmes we synthesize. In SyGuS terminology, we use a compositional technique on top of an enumerative approach, i.e. we enumerate small

programmes and combine them to create more complex programmes. However, unlike the SyGuS setting, we do not have a complete specification of what the synthesized programme should do.

Our synthesis approach is based on a modified decision tree algorithm. Decision trees have already been used in the context of programme analysis and synthesis. For example, these have been applied to the learning of programme invariants [GNMR16] and for tying-together small programmes in a divide-and-conquer synthesis approach [NSM18, ARU17]

3. Preliminaries and Overview

In Section 1.1, we discussed robustness issues in GUI-based CAD and proposed bridging GUI with programming to remedy this brittleness. We now provide a quick overview of modern CAD interfaces and representations, and how our proposed system addresses the limitations of both, GUI-based and programmatic CAD.

3.1. CAD representation and operations

B-rep is a versatile and widespread representation that keeps track of the *features* in a shape, as well the topology and geometry of each element. A vertex is described by its x, y and z coordinates in 3D Cartesian space. An edge is a curve bounded by two vertices. A curve can be a straight line, a circle or even something complicated like a Bezier curve or B-spline curve. A face is a list of edges with an enclosed surface. The surface can be planar, conical, toroidal or even a B-spline surface. Finally, a solid consists of a closed list of faces. Once an object is roughly built, users can modify sub-parts of the object by selecting features (edges, faces, etc.) and applying operations on them. The following is an abstract view of some common operations available in CAD tools:

⟨Solid⟩ ⊨ ⟨Primitive⟩ | ⟨Affine⟩ | ⟨Boolean⟩ | ⟨PointOp⟩ | ⟨EdgeOp⟩ | ⟨FaceOp⟩

⟨Affine⟩ ⊨ (`Translate` | `Rotate` | `Scale` | `Mirror`) ⟨Solid⟩

⟨Boolean⟩ ⊨ (`Union` | `Intersection` | `Difference`) ⟨Solid⟩ ⟨Solid⟩

⟨PointOp⟩ ⊨ (`Hole` | `CounterSink` | `CounterBore`) ⟨Solid⟩ ⟨Face⟩ ⟨(x,y,z)⟩⋆

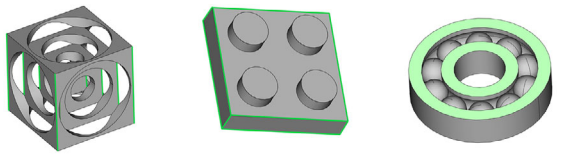
⟨EdgeOp⟩ ⊨ (`Fillet` | `Chamfer`) ⟨Solid⟩ ⟨Edge⟩⋆

⟨FaceOp⟩ ⊨ (`Shell`) ⟨Solid⟩ ⟨Face⟩⋆

The choice of a B-rep implementation fixes the set of primitive operations which can be used in the language. Most open-source projects use OPEN CASCADE, and therefore, support similar operations.

CadQuery *programmatic interface*. CADQUERY is a flexible and high-level domain-specific language based on OPEN CASCADE. CADQUERY is implemented as a shallow embedding in PYTHON, and therefore, inherits its control structure and module system. There are two types of domain specific operations in CAD:

(i) algebraic and
(ii) *query* operations.



(a) *Edges parallel to the Z-axis (some selections hidden due to occlusion).* **(b)** *Non-circular edges (some selections hidden due to occlusion).* **(c)** *Faces with the maximum Z-coordinate.*

Figure 4: *Some selection predicates supported by* CadQuery .

Algebraic operations have an algebraic structure, for example, affine transformations and boolean operations. These operations map directly to operators or methods in the underlying language. A distinct feature of algebraic operations is that these are often total, i.e. they are well defined for all possible inputs, and are therefore robust. *Query* operations, on the other hand, modify specific features of objects. For example, a chamfer is applied to a specific edge. Therefore, to apply such operations, there is a need of identifying features on which the operation applies (recall that GUI-based tools use name). Programmatic interfaces such as CADQUERY (as well as FEATURESCRIPT and SCADLA) provide a small query language to perform such selections. A query on an object first specifies a type, and then a property. The query returns elements of the specific type which satisfy the property. Coming back to the dowel end-cap example presented in Section 1, the following code segment (in CADQUERY) generates a design that is robust to parameter changes:

```
# Make the base
2 base = box(base_l, base_w, base_h)
3 # Make the tube
4 tube = base.faces(''>Z'').circle(tube_r).extrude
(tube_h, combine=False)
5 tube = tube.faces(''>Z'').shell(tube_shell)
6 # Union the base and the tube
7 result = base.union(tube)
8 # Fillet relevant edge(s)
9 result = result.edges(''%CIRCLE'').
edges(''<Z'').fillet(fillet_r)
```

Line 7 performs an algebraic operation, which is robust, even when done via GUI. Lines 4, 5 and 9 perform *query* operations, which cannot be robustly specified via GUI. Line 4 creates a solid cylinder on the maximal face in the Z-axis of the base. Line 5 transforms the solid cylinder into a shell of thickness `tube_shell` by removing the top-most face. Line 9 first selects all circular edges in the design, and then fillets the minimal edges in the Z-axis.

The selection API in CADQUERY includes several selection predicates (see Figure 4 for some examples). These predicates can be categorized on the basis of whether they depend on intrinsic properties or relative properties over multiple elements, and whether the predicates take parameters. For instance, an intrinsic predicate can select all the edges parallel to the Z-axis (Figure 4a), or, non-circular edges (see Figure 4b). A relational predicate can select the face(s) with the maximal Z-coordinate (see Figure 4c). Note that in addition to the standard axes, the selection predicates can be defined over any arbitrary vector. Another parametric predicate is a bounding

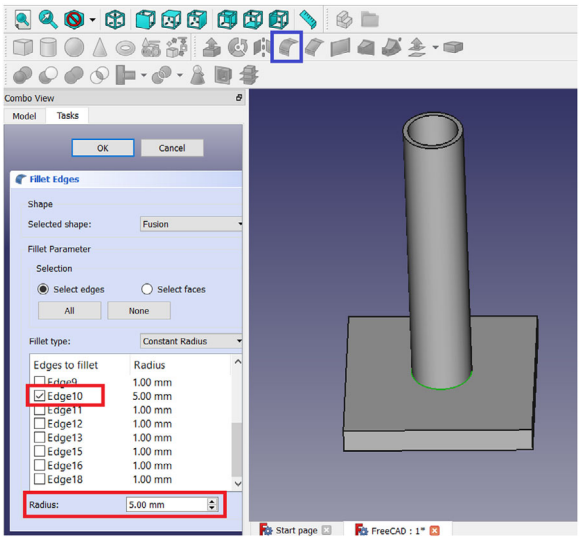
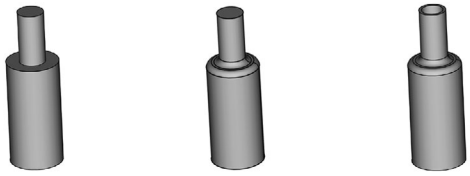


Figure 5: *Filleting the edge connecting the rectangular base and the cylindrical tube in* FreeCAD. *The relevant edge is selected in the interactive view. Then, the 'Fillet' option in the toolbar is chosen. The boxes on the menu on the left show the corresponding edge reference and the fillet parameter.*



(a) *Union of two offsetted cylinders.* **(b)** *Sharp edges at the neck smoothed.* **(c)** *Shell created (final design).*

Figure 6: *Some intermediary steps for designing a bottle in CAD.*

box, which selects all the elements within it. Predicates can also be chained (using `.`), and combined as boolean formulae (using `and`, `or`, and `not`).

In addition to the predicates we have already seen, there are also predicates for parallelism (`''|Z''`), orthogonality (`''#Z''`), and other special geometry (`''%CYLINDER''`). Common predicates are written as strings. Predicates can also be objects in the programming language. This allows them to take expressions as parameters. For instance, `BoundingBox((0,0,0), (s,s,s))` selects all elements inside a box, whose dimensions depend on the variable `s`.

Semantic queries versus direct manipulation. As an alternative mode of selection, writing queries requires thinking semantically. While it is usually harder to write a query than select elements in the GUI, a query carries more meaning. For instance, consider line 9 of the dowel end-cap code presented before. To do the same operation in GUI, users need to select the relevant edge, and choose the fillet operation, as demonstrated in Figure 5. We show FreeCAD here, but other popular CAD applications also have similar interfaces. The selection mechanism in GUI is intuitive and quick. However, notice

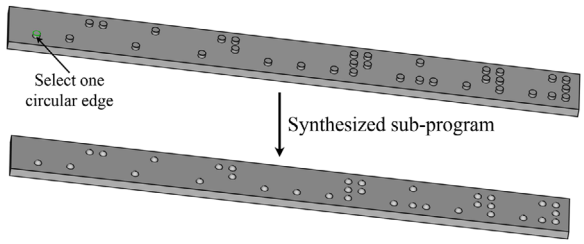


Figure 7: *Using the programme's structure to synthesize more relevant queries. A programme generates a Braille plate with several extruded circles that are created together in a collection. Selecting any one (or more) of these circles would generate a formula for all the circles in this collection. Therefore, the rounding is applied to all the extruded circles to get the final design.*

that the GUI in Figure 5 uses a name identifier for the selected edge (`Edge10`). This is the source of brittleness. If this identifier changes because the shape is modified, the fillet operation would either fail, or worse, modify the object in some other way (due to unknown ambiguity resolution heuristics).

3.2. Towards interactive programming for CAD

Programming languages offer variables for parameter management, control flow structures, precision, modularity and re-usability. On the other hand, GUI interfaces shine when it comes to selection mechanisms and getting immediate feedback on operations. Though writing selection queries can be a challenging task, due to the implicit structure to most parametric designs, it is clearly meaningful to do so. We show that a tight integration of direct manipulation and programming can help designers get the best of both worlds. The focus of our work is on achieving this by enabling designers to use GUI interfaces for easy selection, and automatically synthesizing sub-programmes that represent their actions.

We now present a simple design example to demonstrate the capabilities of our approach. Suppose we want to design a bottle as in Figure 6c. Designers start with a blank sketch and programme. They fill in some environment variables, such as the radius and height of the bottle:

```
radius = 5.0
height = 20.0
```

They then create a cylinder, which serves as the body of the bottle. This operation can be directly translated to code, and is robust:

```
cyl1 = circle(radius).extrude(height)
```

They then need to create another cylinder for the neck of the bottle. This cylinder is to be created on top of the existing one. In the GUI, users select the face on top of which they want to create the new cylinder. Our system automatically synthesizes the relevant query:

```
cyl2 = cyl1.faces(''>Z'').circle(radius/2).extrude
(height/2, combine=False)
```

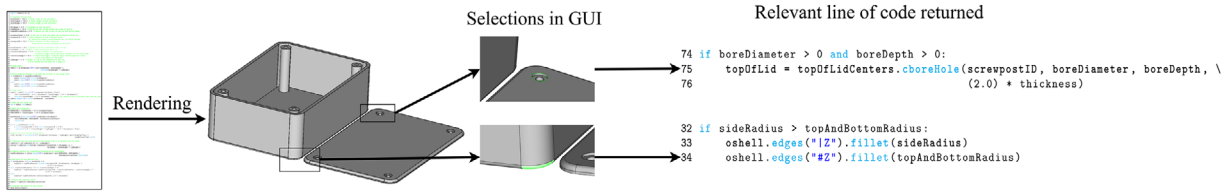


Figure 8: *Modification/debugging of a complicated programme. As we track intermediary states of the design, selecting an element or feature in the direct manipulation interface takes us to the relevant line of code responsible for it. This aids in local modifications and debugging.*

Designers now union the two cylinders together. Again, this operation can be directly translated to code:

```
bottle = cyl1.union(cyl2)
```

So far, we have a design as in Figure 6a. Next, the designers want to smooth the edges at the neck of the bottle (as in Figure 6b). They perform this operation in GUI, and the relevant selection query is automatically synthesized:

```
bottle = bottle.edges(''%CIRCLE and (not >Z) and
(not <Z)'').fillet(0.2 * radius)
```

Finally, they create an empty shell by removing the top most face of the bottle and specifying a thickness. The query is automatically synthesized:

```
bottle = bottle.faces(''>Z'').shell(-0.1 * r)
```

This completes the design process and we have the final design as in Figure 6c.

Notice that this workflow shields designers from the complexity of the semantics of their GUI selections, and at the same time, benefits them due to a generalizable underlying programme. Such workflows are enabled by our system. In addition to such interactive synthesis of queries, we also keep track of environment variables and the programme structure (loops, if-else blocks, and collections). This helps us synthesize more relevant queries and support interactive local modifications and debugging. We now present two examples based on samples from CadQuery 's public repository to demonstrate this.

Using programme structure. Consider the example of making a plate of Braille text (Figure 7). The following code segment generates this with cylindrical bumps:

```
1 # Make the base plate
2 base = box(get_length(), get_width(), height)
3 # Get points of the braille and extrude them from
the base plate
4 braille = base.faces(''>Z'').get_points
(uvCoords)
5 .circle(radius)
6 .extrude(bump, combine=False)
```

Now, let us say we want to round the cylinders created in line 6 so as to resemble hemispheres. In a traditional GUI-based interface, this would entail selecting each of the extruded edges and rounding them. In our system, however, selecting any one of these edges helps us identify the programmatic context in which the edge was created. In this example, all of these edges were created together in line 6. Therefore, our system generates a query for all the edges created in the same context:

```
result = braille.edges(''>Z'').fillet(radius)
```

3.2.1. *Local modifications and debugging*

The way designers typically debug and perform local modifications on their design is by selecting the specific feature of the design in GUI, and analysing/modifying its attributes. However, this can be considerably more difficult in a programmatic interface. For example, consider a storage box design (Figure 8). The programmatic representation (48 lines of code) is parametric, and uses a loop and several if-else blocks. It is difficult to follow the programme logic due to interdependencies between several operations. Such use cases can be drastically simplified by our system. As we track how the design changes in each successive line of code, given a specific feature in the design, using reflection, we can identify the line of code responsible for it. Users can then analyse attributes or make modifications there. Figure 8 shows two examples of user selections in the GUI and the relevant line of code returned by our system.

In the context of CAD, two important parts we do not cover in this work are 2D sketching, and algebraic operations and affine transformations. Both of these have been covered in the sketch-n-sketch framework for vector graphics [HC16, CHSA16], and these techniques can be directly applied to CAD. Our focus is on the synthesis of *queries* from user selections in direct manipulation interfaces. Our implementation is based on the FreeCAD GUI and CadQuery programmatic back-end. We synthesize selection queries using a modified decision tree algorithm. Decision trees are quite efficient at building such formulae. Moreover, due to their white-box nature, it is easy to understand why the synthesis procedure comes up with a certain formula rather than another one (in cases where more than one formula is possible). Our decision tree procedure essentially chooses a predicate to add to the formula in a greedy fashion. Doing so incrementally gives us the complete and correct-by-construction formula. Although the decision tree procedure cannot guarantee synthesis of the shortest formula, we find that, in practice, these formulae are quite small, readable and quickly computed.

4. The Synthesis Framework

Abstractly, our method learns code from examples. Given an example (a shape and a direct manipulation operation), the goal is

to learn code that produces the same result as the direct manipulation when run on the shape. Furthermore, we want the code to generalize to other shapes obtained by changing parameters of the design. Since we look at selection, our problem is also a classification problem, i.e. learning a classifier where the selected elements are the positive instances. However, as we generate code corresponding to the classifier, we use white-box learning, which provides models that can be interpreted by humans. Furthermore, our algorithm needs to be fast and complete (i.e. able to generate a selector for any direct manipulation operation) for a predictable user experience. Finally, our method needs to already work on a single example.

We use decision trees as they satisfy the constraints stated above. We can generate code by traversing learned trees and we can have enough predicates for selection so that our method is complete. To generalize from a single example, we rely on Occam's razor. We search for small decision trees and expect them to generalize better. However, our method can be easily extended to search according to other cost functions.

The core of our method is a decision tree algorithm modified in two important ways. Firstly, decision trees usually make decisions using unary predicates (*intrinsic* properties of elements). However, we also include *relational* predicates that depend on context. For instance, element(s) with the maximal X-coordinate in a shape may change if we first filter this shape with another selector. Secondly, in our case, the set of predicates available to the algorithm is not fixed in advance. We can also select elements based on values in the programme. For instance, we can select elements located within a range, where the values for this range can be constant literals, or come from variables in scope.

We now describe the general framework of our technique. Given the programme, we analyse objects at each line of code to decide which line number and object to synthesize a query for, i.e. for $\bar{O}$ as the ordered set of objects at each successive line of code, we find: $\min(\{i \mid O_i \in \bar{O} \wedge T \subseteq O_i\})$, where T is the target set of elements. If O is the object at this line of code, our objective is to synthesize a sub-programme that when applied to O gives T.

4.1. Syntax of Synthesized Programmes

The following is the abstract syntax of the programmes we synthesize:

$$\begin{aligned}\langle\text{Selection Query}\rangle &\models \neg\langle\text{Predicate}\rangle \mid \langle\text{Primitive Predicate}\rangle \mid \\ &\quad\ \langle\text{Predicate}\rangle\ \langle\text{Binary\_Op}\rangle\ \langle\text{Predicate}\rangle \\ \langle\text{Binary\_Op}\rangle &\models \wedge \mid \vee \mid .\end{aligned}$$

A *selection query* is essentially a combination of primitive predicates selecting elements in a shape (i.e. vertices, edges or faces). We have two modalities for combining predicates: boolean operations and sequence ('.'). Boolean combinations behave as set intersection, union and complement over sets of elements. The sequence operation is related to predicates that work on groups of features. For instance, minimum and maximum fall into this category. Sequencing involves re-evaluating these predicates on the current set of features. Primitive predicates are predicates directly supported by the underlying language (CadQuery in our case).

Algorithm 1. Modified decision tree algorithm for synthesizing a query representative of a GUI selection

SynthQuery(C, T, S_{curr}, S)
Parameters: t: threshold for information gain.
Input: C: Current set of elements, initially all elements,
T: Target set, i.e., the selected elements,
S_{curr}: Pre-computed set of predicates,
S: All available selection predicates.
Output: Decision Tree,
Flag for recomputation, ignore for last return

1 **if** $C \subseteq T$ **then**
2 **return** True
3 **else if** $C \cap T = \emptyset$ **then**
4 **return** False
5 **else**
6 Evaluate the relational selection predicates in S on C and store the result in S_{new}.
7 Calculate information gain for each selection predicate in S and S_{new}.
8 Let $s \in S_{new} \cup S_{curr}$ be the selection predicate with the highest information gain, and $\emptyset \subset s \cap C \subset C$.
9 **if** s *information gain* $< t$ **then**
10 Use C and program context to generate a new selection predicate (Section 4.2.1).
11 **return** SynthQuery(C, T, S_{curr}, $S \cup \{s\}$)
12 **else**
13 **if** $s \in S_{new}$ **then**
14 $S_{curr} := \left\{ x \mid \begin{matrix}(x \text{ intrinsic } \wedge x \in S_{curr}) \vee \\ (x \text{ relational } \wedge x \in S_{new})\end{matrix} \right\}$
15 $L_T, L_C :=$ SynthQuery($C \cap s$, T, $S_{curr} \setminus \{s\}$, S)
16 $R_T, R_C :=$ SynthQuery($C \setminus s$, T, $S_{curr} \setminus \{s\}$, S)
17 $L := L_C\,?\,(s.L_T)\ :\ (s \wedge L_T)$
18 $R := R_C\,?\,(\neg s.R_T)\ :\ (\neg s \wedge R_T)$
19 **return** $L \vee R, s \in S_{new}$

4.2. Synthesis algorithm

Decision trees are popular in machine learning for solving classification and regression tasks. Unlike other techniques like neural networks, decision tree learning is a white-box approach. It is possible to understand the logic behind decision procedures of these trees. Moreover, due to their simple design, this logic is also human readable. We use this feature of decision trees to synthesize code snippets for selection queries. For the sake of simplicity, we use a selection predicate and the set of elements it selects interchangeably.

Description of the algorithm. Our synthesis algorithm is based on the popular ID3 Decision Tree learning algorithm [Qui86]. Given the relevant top-level object O, we start with all the elements that can possibly be selected, $O = \{o_1, o_2, o_3, \ldots\}$. We also have the set of selected elements we want to derive a selection query for, $T = \{t_1, t_2, t_3, \ldots\} \subseteq O$. We maintain the notion of a current set, C, which is the set we are currently working with in the decision tree procedure. In the beginning of the procedure, $C = O$. We then follow the algorithm as in Algorithm 1. Firstly, we check if the current set C is a *base case*. Set C is a base case if $C \cap T = \emptyset$ or if $C \cap T = C$. In either case, we do not need further decision steps,

as C contains only positive or negative examples. If C is not a base case, we perform further decision steps until we reach a base case. At each decision step, we choose the selection predicate with the highest information gain (and that does not lead to selecting C or $\emptyset$). When the rate of progress decreases, we add a new candidate selection predicate by looking at the programme context. The selection predicate can either be from a pre-calculated selector set S_{curr} or from a newly calculated predicate set S_{new}. Re-calculating means we combine the predicate with '.' and using a pre-calculated predicate means we combine it with '$\wedge$'. The recursive call of the algorithm returns a flag to indicate whether to use '.' or '$\wedge$' when connecting the sub-tree to its parent.

The formulation of entropy and information gain is the same as in the standard ID3 Decision Tree algorithm: $H(C) = -\sum_{x \in X} p(x) \log_2 p(x)$, where $H(C)$ is the entropy of the current set C. X is the set of classes in C. In our case, X has two classes, elements that are in the target set (positive examples) and elements that are not (negative examples). $p(x)$ is the proportion of elements in a class x to the total number of elements in C. The information gain for the predicate s when applied to C is $IG(C, s) = H(C) - H(C|s) = H(C) - \sum_{k \in \{C \cap s, C \setminus s\}} p(k)\, H(k)$. The selection predicate s partitions C into two subsets, one for s and the other for $\neg s$. $p(k)$ is the proportion of elements in subset k to the total number of elements in C.

4.2.1. *Correctness and completeness*

Our algorithm is correct-by-construction. On the other hand, completeness critically relies on the availability of selection predicates, and the threshold t on the information gain to be low enough for all nodes in the decision tree to have an information gain larger than t. Our algorithm is relatively complete (depending on t).

For trivial completeness, we can examine each $o \in O$ and generate a predicate for each one (structural equality). However, this would slow down the algorithm and likely degrade the quality of the synthesized queries (over-fitting). Therefore, our algorithm starts with fewer, more general predicates, and if these are not enough, it adds more specialized predicates lazily. This process is bound to complete (given a low enough threshold) as there are finitely many properties that elements possess, and in each successive decision step, the size of the current set C reduces monotonically until a base case is reached.

4.2.2. *Practical adjustments for efficiency*

The aim of our technique is to provide an interactive programming environment that can work with arbitrarily complex shapes and GUI selections. The algorithm already handles intrinsic and relational predicates differently to avoid needless re-evaluation of intrinsic predicates. We now suggest some further adjustments to the algorithm.

(i) To re-evaluate selection predicates that depend on the set of features (like maximum and minimum), we need to create a temporary object O_{new} and evaluate the predicates on this to generate S_{new}. This is an expensive process, especially when the set of selection predicates and elements in the object is large. Therefore, we propose only calculating this if the maximum information gain from predicates in S_{curr} is less than a certain threshold.

(ii) The calculation of S_{new} can be done in a smart way. Predicates such as largest or smallest in a particular coordinate axis depend on the elements in our current set C. However, there are certain predicates which do not need to be explicitly re-calculated, such as orthogonality and parallelism to a coordinate axis. These can be directly inferred from S_{curr}.

4.2.3. *Generating selection predicates*

Selection predicates, typically the intrinsic ones, may depend on parameters. This enables the generation of new predicates on-the-fly.

4.2.4. *Non-parametric predicates*

The simplest predicates are non-parametric. There are a finite number of them and they capture the most common use cases. Our algorithm starts with these predicates. These predicates include maximal or minimal elements in each coordinate axis, elements parallel or orthogonal to each coordinate axis, types of geometry, etc. Extremal elements are quite intuitive for humans to understand and use as there is a direct mapping from these to natural language ('top-most', 'left-most', etc). This is also the case for parallelism and orthogonality predicates. Predicates based on geometry enable selections such as round edges, planar faces, etc.

4.2.5. *Parametric predicates*

The second category of predicates take parameters. By giving different values to these parameters, we get different selections. To generate these selection predicates, we use values from the elements selected, and variables in scope where the query is to be generated. We prioritize use of variables as they are more likely to be robust to programme changes.

We have implemented selection based on bounding boxes. Although the bounds can be generated using constant literals, we try to fit variables in scope to the constraints so as to have more readable and likely-to-generalize queries. If $V = \{v_1, v_2, v_3 \ldots\}$ are environment variables in the programme, and $[a, b]$ is the bounding constraint for a particular selection set, we try to find $v_i \in V$ with minimum distance to a and $v_i \leq a$. Similarly, for the upper bound, we try to find v_i with minimum distance to b and $v_i \geq b$.

However, there are many more parametric selectors which can be added. For instance, the length of edges, the area of faces or the volume of solids can be used. Our algorithm is extensible and it is easy to add more selection predicates. More predicates may lead to synthesis of shorter queries. However, in order to understand what these queries do, designers would need to know a larger list of predicates. A decision on which direction is better in this trade-off requires further study.

4.3. Querying objects in a loop or collection

Very often, programmes operate on collections of objects. An obvious example is the map function, which takes as input a collection of objects and returns a collection with some transformation

Algorithm 2. Synthesizing queries on collections

SynthQuery$_{\text{Collection}}$ $([O_1, \ldots, O_n], T)$
Input: $[O_1, \ldots, O_n]$: Collection of objects, T: Target set.
Output: Query.
1 $L := [O_1, \ldots, O_n]$.filter($\lambda o.\ T \subseteq o$)
2 $C := |L| = 1\ ?\ \text{head}(L) : \bigcup_i O_i$
3 Evaluate the predicates in S on C, store results in S_{curr}.
4 **return** SynthQuery(C, T, S_{curr}, S)

applied to each element. In CAD, this is fairly common as well (an example was presented in Section 3.2). As we maintain a snapshot of programme state at each line number, given a set of elements selected using direct manipulation, we check if the selected element(s) are part of a collection. If this is the case, we generate a query fitting the whole collection. Algorithm 2 shows how this is done.

5. Evaluation

We now present implementation details, and provide experimental evidence to demonstrate the applicability of our approach to modern parametric CAD workflows. Section 5.1 provides implementation details. Section 5.2 provides evidence of our approach working well in practice. In Section 5.3, we present a user study. In Section 5.4, we show that even when dealing with complex designs that do not have a programmatic representation, our technique can synthesize selection queries fairly quickly.

Wherever we report number of lines of code, we exclude blank lines and comments. Wherever we report running time, the experiments are done on a machine with an Intel Core i3-8100T processor, 8GB RAM and an Intel UHD Graphics 630 graphics card. In the Appendix, we report additional details and more experiments.

5.1. Implementation

Our implementation is available at https://gitlab.mpi-sws.org/mathur/ipcad (around 1100 lines of PYTHON code). We build on top of FREECAD (version 0.17), a popular open-source GUI-based CAD application, and CADQUERY (version 1.2.0), an open-source programmatic interface. These two interfaces are bridged together. Although CADQUERY offers a set of tools for integration with FREECAD, this is restricted to displaying the programme's output on the FREECAD GUI. There is no interactivity during the design process and no programming-specific debug features. Our implementation brings this.

GUI interface. The FREECAD API enables listening to GUI events. Our implementation listens to events that correspond to selection of elements in the design and performing of operations. Once a selection in the GUI is made, we map the selected elements in the GUI to elements in the design's programmatic representation.

Programming interface. The programming interface is unaware of the GUI interface except for its use as the output device (as is usually the case). However, we instrument the programme so as to track intermediate states of the design with the help of PYTHON's reflection API (`inspect` module). Tracking intermediary states helps us determine which line of code led to a particular selected element being modified. This enables interactive local modifications and debug features. Moreover, tracking intermediate states helps understanding the control structure of the programme, as well as identifying which elements were created together, for example, in a loop or in a collection. In addition to tracking the state of the design and control flow, we also maintain a list of variables in scope. This is done so as to include these as parameters in the synthesized queries, especially range-based queries.

5.2. Synthesis robustness and runtime

We now evaluate our technique on several important metrics like robustness of the synthesized queries and runtime of the algorithm. The evaluation is based on a wide variety of designs, both simple and complex, and encompasses a large variety of application areas. There is a need for ground truth to appropriately assess the quality of our synthesized queries. Our examples are therefore based on designs whose source code is also available. Here, we discuss experiments on CADQUERY samples, where we use example programmes from their public repository [Par19b]. In Appendix E, we provide additional experiments on THINGIVERSE samples, where we use some parametric designs available on THINGIVERSE 's customizable section [Mak19].

Experimental procedure and results. There are 22 designs in CADQUERY 's repository that use selection queries. We include all of them (see Figure 9 for a snapshot of the designs) for this experiment. The aim is to evaluate the runtime of our system's synthesis procedure, as well as examine whether the synthesized queries are correct (or, as intended by the original authors of the design). The experimental procedure is as follows:

(i) For each CADQUERY example, we start with a blank programme, and copy the example until a selection query occurs.
(ii) We use the FREECAD GUI to display the output until this line.
(iii) In the FREECAD GUI, we manually select the elements selected by the ground truth query.
(iv) We append the query returned by our automatic synthesis procedure to the programme, and carry on until the next selection query, or the end of the design.

In Table 1, we report runtime and query size (number of predicates in the query) for these examples. The number of vertices, edges and faces in the examples, as well as lines of code (LOC) are also reported to get a better sense of the complexity of the underlying designs. To evaluate the correctness of the synthesized queries, we compare the synthesized queries to the ground truth queries. Two corresponding queries can either be equal, logically equivalent or different. Equality and logical equivalence ensure correctness of the synthesized query. However, if the synthesized query is different from the ground truth query, it does not necessarily mean that it is incorrect (existence of multiple semantically equivalent queries). For synthesized queries that are different from the ground truth query, we randomly sample over the programmes' parameter space and compare the resulting meshes. We sample parameters randomly as

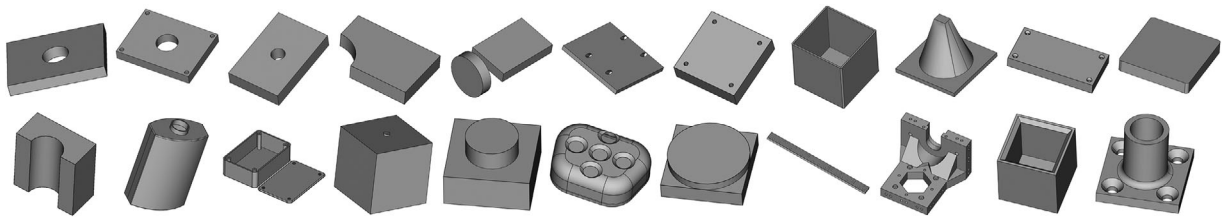
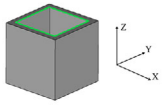
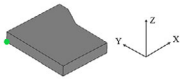


Figure 9: CadQuery *designs (default parameters).*



(a) *Logically equivalent: synthesized "(>Z and (not >X and (not >Y and (not <X and (not <Y))))) vs. ground truth "(>Z) . (not (<X or >X or <Y or >Y))".*



(b) *Experimentally equivalent: synthesized "(>Y and (not >X and (not >Z)))" vs. ground truth "(<Z) . (>Y) . (<X)".*

Figure 10: *Examples of logically and experimentally equivalent queries (elements in green are selected).*

Table 1: *Analysis of query size, synthesis runtime and robustness.*

CadQuery examples			
	Min.	Avg.	Max.
# LOC	3	23.40	127
# Vertices	8	33.27	172
# Edges	12	51.72	252
# Faces	6	24.81	113
# Queries	1	2.41	13
Query size	1	1.64	7
Time (s.)	0.001	0.008	0.089
Robustness of 55 synthesized queries			
# Equal			43
# Logically equivalent			4
# Experimentally equivalent			8

automatically finding 'intended' parameter values is known to be difficult [HK01]. We compare meshes by calculating the Hausdorff distance [RW09] between them using MeshLab [CCC*08]. If the resulting meshes are the same over 50 random samples, we mark these queries as *experimentally equivalent*. Figure 10 provides a visual example of the difference between logically and experimentally equivalent queries. We report the results on our synthesized queries in Table 1. In Appendix D, Table D1 gives specific details on each example in the experiment.

On the basis of these results, we find that our technique is useful in a practical interactive programming setting. The synthesis procedure is quick to synthesize queries. In fact, the longest running time for these examples is less than a tenth of a second. The synthesized queries are also successful at capturing design intent.

Table 2: *User performance on Programmatic only versus Programmatic + GUI (our) interface. We report the percentage of queries attempted and correctness over all participants.*

	Programmatic only			Programmatic + GUI		
	Min.	Avg.	Max.	Min.	Avg.	Max.
% Attempted	48	77.5	100	64	94	100
% Correct	58.3	84.5	95.8	93.8	98.3	100

5.3. User Study

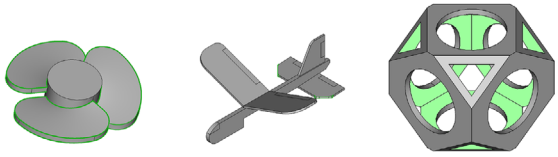
We conducted a user study with six participants, inclusive of beginner, intermediate, and expert CAD users. Each study was approximately 60 min long: 20 min for a short tutorial and feedback, 20 min using a Programmatic system, and 20 min using Programming + GUI (our system). Users were asked to write queries for 14 selected designs (total 49 queries), which contained CadQuery examples after removing redundant designs (designs with only one query, which occurs in a similar way in other designs), and adding the dowel-end cap and bottle examples presented earlier in the paper. The designs were partitioned into two groups. Participants interchangeably used the programming only interface for one group, while using our interface for the other. The aim of the study was to collect quantifiable user data on efficiency and accuracy of our interface in comparison to baseline (programming alone). Appendix F provides more details on the user study.

In Table 2, we report how fast and how accurate the participants were in the two interfaces. Using the GUI to synthesize programmatic queries yielded significant improvements in speed and accuracy of design. There were just two instances when the GUI + Programmatic interface did not give the correct result: firstly, where the user gave up because they needed to select several edges, whereas the query was relatively straightforward, and secondly, where the user selected the wrong element in a symmetric design.

Participants also filled a post study questionnaire. Table 3 aggregates their answers to questions based on a Likert scale. The questionnaire also asked which interface the participants preferred. Everyone preferred the GUI + Programmatic interface. When asked why they preferred this interface, the most recurring opinion was

Table 3: *A summary of qualitative opinions in the user study, reported on the Likert scale (1—Strongly agree, 5—Strongly disagree).*

Opinion	Min.	Max.	Median	Avg.
Writing selection queries yourself (without aid from GUI) is difficult.	1	4	3	2.8
Generating selection queries using the GUI simplifies the process of writing selection queries.	1	2	1	1.3
Queries generated by the GUI are what users/programmers would write themselves.	1	3	2	2.0



(a) *A propeller with edges of the blades selected.* **(b)** *A glider with rear wings' front edges selected.* **(c)** *A whiffle ball with inner faces selected.*

Figure 11: *Some complex designs without a programmatic representation. We synthesized queries for the elements coloured in green.*

Table 4: *Analysis of query size and synthesis runtime on models without a programmatic representation.*

Model	Vertices	Edges	Faces	Query size	Time (s.)
Propeller	42	61	27	106	1.130
Glider	136	219	88	54	5.133
Whiffle ball	60	90	26	14	1.355

that using the GUI was faster. The participants especially preferred GUI when the selections were non-standard (more than one selection predicate, or when the object was not parallelepiped).

5.4. Synthesis scalability

The queries generated in the previous experiment are usually small as they utilize the underlying programmatic representation of the design. We now show that our algorithm can also scale to more complicated and artistic designs that do not have an underlying programmatic representation. A typical use case of this is when designers only have access to the final object, and want to make robust modifications on these. The models for this case study are taken from the public repository of FreeCAD 's official tutorials [BPL16]. The models we used and the selections for which we synthesize queries are depicted in Figure 11. The results of the corresponding synthesis procedure are summarized in Table 4. Though it is not surprising that the query sizes are quite large for these examples, we find that our algorithm can still cope with this in a reasonable amount of time.

6. Future Work

We are exploring two complementary directions as future work. The first is extending the system's capabilities for modifying code. The second is related to longer term deployment and user studies.

Code modifications. We can already track geometric features through code. However, our implementation currently only generates new code snippets and does not modify existing code. For an even tighter integration of direct manipulation and programming for CAD, we need to merge our technique with prior work such as value trace equation solving [CHSA16] and lenses [MKC18].

Richer parametric selectors. Currently, parametric selectors use programme variables in scope directly. However, existing work on programme synthesis [ARU17] can be used to generate arithmetic expressions over programme variables for use in these selectors. This would enable finding parametric selectors with values not readily stored in variables. For instance, the coordinate of the side of a cube can be the sum of one corner's position and the width.

Tuning the algorithm's selection. By default, the algorithm uses information gain to decide which selector to use to expand the decision tree. However, we could use other heuristics as long as they avoid non-trivial predicates that do not split the feature set. In particular, we could modify the information gain to weigh predicates differently. As predicates are not unique, the weight function can be tailored to users' preferences. The algorithm can generate multiple trees by expanding multiple alternatives with roughly similar information gain and let users select which is best. Selections can be recorded, and over time, one can learn a weight function.

Effect of the interface choice on design quality. Related to the previous point, which requires user participation, we would like to study the impact of interface choice on robustness of designs thus created. For example, studying programming versus direct manipulation versus a tight integration of the two, and evaluating user productivity and quality of resulting designs. Indeed, the question of productivity and quality of designs in CAD interfaces is quite old [BJ96], and still very relevant.

7. Conclusion

We identified bridging direct manipulation and programming as a possible solution to getting the best of both worlds for parametric CAD, i.e. intuitiveness and ease of use of direct manipulation, and robustness, generalizability and modularity of programming. To this end, we presented a decision-tree-based approach that synthesizes semantics of selections made in a direct manipulation interface. We demonstrated how the formulae and sub-programmes thus generated can be used for interactive programming of CAD and that our technique scales to complex designs. Our system can also use the programme structure to generate design specific queries and aid in programme modifications and debugging.

References

[ABJ*13] Alur R., Bodik R., Juniwal G., Martin M. M., Raghothaman M., Seshia S. A., Singh R., Solar-Lezama A., Torlak E., Udupa A.: Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design* (2013), IEEE, pp. 1–8.

[AMP00] Agbodan D., Marcheix D., Pierra G.: Persistent naming for parametric models. *Proceedings of the eighth International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media' 2000* (Czech Republic, 2000), University of West Bohemia, Campus Bory, Plzen — Bory.

[ARU17] Alur R., Radhakrishna A., Udupa A.: Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2017), A. Legayand, T. Margaria (Eds.), Springer, Berlin, Heidelberg, pp. 319–336.

[Bar19] Barth A.: Project Egress, 2019. URL: https://dpo.si.edu/blog/project-egress. Accessed October 2019.

[bel19] belliveaul: Air mattress plug, 2019. URL: https://www.thingiverse.com/thing:3818734. Accessed October 2019.

[BGHZ15] Barowy D. W, Gulwani S., Hart T., Zorn B.: FlashRelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, ACM, pp. 218–228. URL: http://doi.acm.org/10.1145/2737924.2737952.

[BH11] Bettig B., Hoffmann C. M: Geometric constraint solving in parametric computer-aided design. *Journal of Computing and Information Science in Engineering 11*, 2 (2011). URL: https://doi.org/10.1115/1.3593408.

[BJ96] Bhavnani S. K, John B. E: Exploring the unrealized potential of computer-aided drafting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1996), CHI '96, ACM, pp. 332–339. URL: http://doi.acm.org/10.1145/238386.238538.

[BPL16] BPLRFE: Youtube-Tutorial-Models, 2016. URL: https://github.com/BPLRFE/Youtube-Tutorial-Models. Accessed October 2019.

[br15] bpl rfe: Turners Cube, 2015. URL: https://www.thingiverse.com/thing:1072045. Accessed October 2019.

[Bri19] Britt A.: Filament storage dowel end cap, 2019. URL: https://www.thingiverse.com/thing:3324110. Accessed October 2019.

[CCC*08] Cignoni P., Callieri M., Corsini M., Dellepiane M., Ganovelli F., Ranzuglia G.: MeshLab: An open-source mesh processing tool. In *Eurographics Italian Chapter Conference* (2008), V. Scarano, R. D. Chiara and U. Erra (Eds.), The Eurographics Association. http://doi.org/10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[CCH96] Capoyleas V., Chen X., Hoffmann C. M.: Generic naming in generative, constraint-based design. *Computer-Aided Design 28*, 1 (1996), 17–26.

[CH95] Chen X., Hoffmann C. M.: Towards feature attachment. *Computer-Aided Design 27*, 9 (1995), 695–702.

[Che95] Chen X.: *Representation, Evaluation and Editing of Feature-Based and Constraint-Based Design*. PhD thesis, Purdue University, 1995.

[CHSA16] Chugh R., Hempel B., Spradlin M., Albers J.: Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2016), PLDI '16, ACM, pp. 341–354. URL: http://doi.acm.org/10.1145/2908080.2908103.

[Chu19] Chunkmunk01: Parametric Setup Block, 2019. URL: https://www.thingiverse.com/thing:3749463. Accessed October 2019.

[dan19a] danieln: Parametric Hose or Pipe Adapter, 2019. URL: https://www.thingiverse.com/thing:3819146. Accessed October 2019.

[dan19b] dannyprince: Configurable Electronics bay for Arduino Nano, 2019. URL: https://www.thingiverse.com/thing:3859829. Accessed October 2019.

[DIP*18] Du T., Inala J. P., Pu Y., Spielberg A., Schulz A., Rus D., Solar-Lezama A., Matusik W.: InverseCSG: Automatic conversion of 3D models to CSG trees. In *SIGGRAPH Asia 2018 Technical Papers* (2018), ACM, p. 213.

[DiP19] DiPi92: Simple customizable speaker grill, 2019. URL: https://www.thingiverse.com/thing:3834348. Accessed October 2019.

[Fea20] FeatureScript: Welcome to FeatureScript, 2020. URL: https://cad.onshape.com/FsDoc/. Accessed October 2019.

[FKS*04] Funkhouser T., Kazhdan M., Shilane P., Min P., Kiefer W., Tal A., Rusinkiewicz S., Dobkin D.: Modeling by example. In *ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), SIGGRAPH '04, ACM, pp. 652–663. URL: http://doi.acm.org/10.1145/1186562.1015775.

[Gir01] Girard P.: Bringing programming by demonstration to CAD users. In *Your Wish Is My Command*. H. Lieberman (Ed.), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 135–162. URL: http://dl.acm.org/citation.cfm?id=369505.369514.

[GNMR16] Garg P., Neider D., Madhusudan P., Roth D.: Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New York, NY, USA, 2016), POPL '16, ACM, pp. 499–512. URL: http://doi.acm.org/10.1145/2837614.2837664.

[Gul16] Gulwani S.: Programming by examples: Applications, algorithms, and ambiguity resolution. In *Automated Reasoning* (Cham, 2016), N. Olivetti and A. Tiwari (Eds.), Springer International Publishing, pp. 9–14.

[HC16] Hempel B., Chugh R.: Semi-automated SVG programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (New York, NY, USA, 2016), UIST '16, ACM, pp. 379–390. URL: http://doi.acm.org/10.1145/2984511.2984575.

[HK01] Hoffmann C. M., Kim K.-J.: Towards valid parametric CAD models. *Computer-Aided Design 33*, 1 (2001), 81–90.

[itz18] itzco: Customizable funnel, 2018. URL: https://www.thingiverse.com/thing:3243733. Accessed October 2019.

[jam19] james_lan: Meade ETX80 replacement eyepiece holder, 2019. URL: https://www.thingiverse.com/thing:3811422. Accessed October 2019.

[jmc19] jmcbassett: Meter box lock shaft, 2019. URL: https://www.thingiverse.com/thing:3841621. Accessed October 2019.

[Kin19] Kintel M.: OpenSCAD: the programmers solid 3D CAD modeller, 2019. URL: http://www.openscad.org/. Accessed October 2019.

[Kri95] Kripac J.: A mechanism for persistently naming topological entities in history-based parametric solid models. In *Proceedings of the 3rd ACM Symposium on Solid Modeling and Applications* (New York, NY, USA, 1995), SMA '95, Association for Computing Machinery, pp. 21–30. URL: https://doi.org/10.1145/218013.218024.

[LG14] Le V., Gulwani S.: FlashExtract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 542–553. URL: https://doi.acm.org/10.1145/2594291.2594333.

[LSL*19] Lipp M., Specht M., Lau C., Wonka P., Müller P.: Local editing of procedural models. *Computer Graphics Forum 38*, 2 (2019), 13–25. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13615.

[LVLR19] Leen D., Veuskens T., Luyten K., Ramakers R.: JigFab: Computational fabrication of constraints to facilitate woodworking with power tools. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2019), CHI '19, ACM, pp. 156:1–156:12. URL: http://doi.acm.org/10.1145/3290605.3300386.

[Mak19] MakerBot Industries, LLC: Thingiverse: Digital designs for physical objects, 2019. URL: https://www.thingiverse.com/.

[MH05] Mun D., Han S.: Identification of topological entities and naming mapping for parametric cad model exchanges. *International Journal of CAD/CAM 5*, 1 (2005), 69–82.

[MKC18] Mayer M., Kuncak V., Chugh R.: Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages 2* (October 2018), 127:1–127:28. URL: http://doi.acm.org/10.1145/3276497.

[Nat19] naturaltangent: Parametric bolt cap, 2019. URL: https://www.thingiverse.com/thing:3814727. Accessed October 2019.

[Not19] notace: Parametric Setup Block, 2019. URL: https://www.thingiverse.com/thing:3870946. Accessed October 2019.

[NSM18] Neider D., Saha S., Madhusudan P.: Compositional synthesis of piece-wise functions by learning classifiers. *ACM Transactions on Computational Logic 19*, 2 (May 2018), 10:1–10:23. URL: http://doi.acm.org/10.1145/3173545.

[NWP*18] Nandi C., Wilcox J. R., Panchekha P., Blau T., Grossman D., Tatlock Z.: Functional programming for compiling and decompiling computer-aided design. *Proceedings of the ACM on Programming Languages 2* (2018), 99.

[OLFB18] Okuya Y., Ladeveze N., Fleury C., Bourdot P.: ShapeGuide: Shape-based 3D interaction for parameter modification of native CAD data. *Frontiers in Robotics and AI 5* (2018), 118. URL: https://www.frontiersin.org/article/10.3389/frobt.2018.00118.

[OPE19] OPEN CASCADE SAS: Open cascade technology, 2019. URL: https://dev.opencascade.org. Accessed October 2019.

[Par19a] Parametric Products Intellectual Holdings, LLC: CadQuery: a parametric cad script framework, 2019. URL: https://github.com/dcowden/cadquery. Accessed October 2019.

[Par19b] Parametric Products Intellectual Holdings, LLC: CadQuery examples, 2019. URL: https://github.com/CadQuery/cadquery/tree/master/examples. Accessed October 2019.

[PPG96] Pierra G., Potier J.-C., Girard P.: The EBP system: Example based programming system for parametric design. In *Modelling and Graphics in Science and Technology* (Berlin, Heidelberg, 1996), J. C. Teixeira and J. Rix (Eds.), Springer, Berlin, Heidelberg, pp. 124–140.

[Qui86] Quinlan J. R.: Induction of decision trees. *Machine Learning 1*, 1 (March 1986), 81–106. URL: http://doi.org/10.1023/A:1022643204877.

[RW09] Rockafellar R. T, Wets R. J.-B.: *Variational Analysis*, Vol. *317*. Springer Science & Business Media, Berlin, 2009.

[SL08] Solar-Lezama A.: *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[SSL*14] Schulz A., Shamir A., Levin D. I. W., Sitthi-Amorn P., Matusik W.: Design and fabrication by example. *ACM*

Transactions on Graphics (Proceedings SIGGRAPH 2014) 33, 4 (2014).

[SSM15] SHUGRINA M., SHAMIR A., MATUSIK W.: Fab forms: customizable objects for fabrication with validity and geometry caching. *ACM Transactions on Graphics 34*, 4 (2015), 100:1–100:12. URL: https://doi.org/10.1145/2766994.

[SWG*18] SCHULZ A., WANG H., GRINSPUN E., SOLOMON J., MATUSIK W.: Interactive exploration of design trade-offs. *ACM Transactions on Graphics 37*, 4 (July 2018), 131:1–131:14. URL: http://doi.acm.org/10.1145/3197517.3201385.

[Vic12] VICTOR B.: Learnable Programming, 2012. URL: http://worrydream.com/LearnableProgramming/. Accessed October 2019.

[WCB17] WANG C., CHEUNG A., BODIK R.: Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 452–466. URL: http://doi.acm.org/10.1145/3062341.3062365.

[WDRAJ16] WAN DIN W. I., ROBINSON T. T., ARMSTRONG C. G., JACKSON R.: Using CAD parameter sensitivities for stack-up tolerance allocation. *International Journal on Interactive Design and Manufacturing (IJIDeM) 10*, 2 (May 2016), 139–151. URL: https://doi.org/10.1007/s12008-014-0235-2.

[xyt16] XYTHOBUZ: Customizable long GoPro screw knob, 2016. URL: https://www.thingiverse.com/thing:1643650. Accessed October 2019.

[Yar13] YARES E.: The failed promise of parametric CAD part 1: From the beginning, 2013. URL: https://www.3dcadworld.com/the-failed-promise-of-parametric-cad/. Accessed October 2019.

[Zuf19] ZUFFEREY D.: Scadla rendering backend based on open cascade, 2019. URL: https://github.com/dzufferey/scadla-oce-backend. Accessed October 2019.

Appendix A: Initial Selection Predicates

The choice of initial set of selection predicates is important for generating quick and human-readable queries. We now present the set of initial selection predicates we used in our experiments and that we use as default:

(i) Intrinsic: parallelism and orthogonality to each of the three coordinate axes, centre in the positive or negative direction of each of the three coordinate axes, line, circle or arc geometry for edges, and plane, cylindrical or spherical geometry for faces.
(ii) Relative: Maximal and minimal elements in each of the three coordinate axes.

All of these predicates are available in CADQUERY, and the synthesized queries can therefore directly be used in a CADQUERY programme. The initial set of selection predicates was kept the same throughout the various case studies. Moreover, the threshold on the information gain is set to 0 for our case studies. This is to guarantee synthesis of a selection query for any possible user selection. However, we see later, in Appendix B how this can lead to rather long selection queries.

Appendix B: Range Queries

We now provide a specific example where a parametric bounding box selector would make sense. Consider the model of a Turner's cube as in Figure B1. For the sake of this example, let us assume that this model does not have a programmatic representation. If we start with the usual selection predicates and 0 threshold on the information gain, we get an extremely large selection query with 56 selection predicates. Due to the layered structure of the design, our algorithm has a hard time coming up with a selector for the inner faces. It tries to select the inner faces by removing faces from the top-level object layer-by-layer. A remedy to this can be a concise parametric selector. For example, if the inner-most cube's edge length, `s` is made available to the algorithm, the synthesized query is:

```
faces(BoxSelector((0,0,0), (s,s,s)))
```

This is an example of a range query in which all faces inside the bounding box defined by the range are selected. Not only is this query shorter, but it is also synthesized quicker: it takes only 0.05 s to synthesize, as compared to 23.29 s for the larger query.

Appendix C: Local Design Modifications

In Section 3.2, we discussed how our system can be used to discover which line of code is responsible for a particular feature in the design. We now demonstrate a slightly different experiment, based on the same example, wherein instead of just debugging or changing of some parameters, we change the design itself. We use the same example as before, i.e. a storage box. Figure 8 shows our storage box. Let us remove the fillet from the bottom part of this box. To do this, we can select any of the rounded faces (or edges) that we wish to modify (see Figure 8), and ask our system to return the line of code responsible for this. Our system returns the following line of code:

```
oshell = oshell.edges(''#Z'').
fillet(topAndBottomRadius)
```

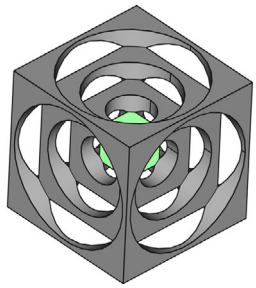


Figure B1: *A Turner's Cube with the innermost faces selected.*

Table D1: *A summary of various* CadQuery *examples run on our system. We report the complexity of the model, the number of queries in the example, the number which were equal/experimentally equal, the average size of each synthesized query and the time it took to synthesize it.*

Model	Vertices	Edges	Faces	# Queries	# Equal	# Exp. Eq.	Avg. size	Avg. time (s.)
Block with Bored Center Hole	10	15	7	1	1	0	1	0,001
Pillow Block with Counterbored Holes	26	39	19	2	2	0	1	0.001
Creating Workplanes on Faces	10	15	7	1	1	0	1	0.001
Locating a Workplane on a Vertex	10	15	7	1	0	1	3	0.001
Offset Workplanes	10	15	9	1	1	0	1	0.001
Rotated Workplanes	20	30	10	1	1	0	1	0.001
Using Construction Geometry	16	24	10	1	1	0	1	0.001
Shelling to Create Thin Features	8	12	6	1	0	1	1	0.001
Lofts	18	27	12	1	1	0	6	0.049
Counter Sunk Holes	20	32	14	1	1	0	2	0.001
Rounding Corners with Fillets	8	12	6	1	1	0	3	0.001
Splitting an Object	12	18	8	2	2	0	1	0.001
Classic OCC Bottle	14	21	10	2	2	0	3	0.001
Parametric Enclosure Filleting	124	192	90	7	6	1	1	0.089
FreeCAD Solids as CQ Objects	10	15	8	1	1	0	1	0.001
Lego Brick	16	24	11	3	3	0	1	0.001
Remote Enclosure	64	112	51	5	4	1	1	0.001
Numpy	13	22	11	1	1	0	1	0.001
Braille	110	182	108	2	2	0	1	0.001
3D Printer Extruder Support	172	252	113	13	7	5	1	0.014
Shelled Cube Inside Chamfer	16	24	11	2	2	0	2	0.001
Reinforce Junction Using Fillet	25	40	18	3	3	0	1	0.015

Table D2: *A summary of various* Thingiverse *examples run on our system. We report the complexity of the model, the number of queries required to be synthesized, the average size of each synthesized query, the time it took to synthesize it and error metrics on the meshes generated using random sampling.*

Model	Vertices	Edges	Faces	# Queries	Avg. size	Avg. time (s.)	Max. error	Avg. error
Air Mattress Plug [bel19]	18	27	11	1	1	0.001	0%	0%
Bolt cap [Nat19]	26	43	20	4	1	0.001	3.39%	0.55%
Electronics Bay [dan19b]	50	90	24	2	1	0.001	0%	0%
Eyepiece Holder [jam19]	35	53	19	3	1	0.001	0.32%	0.13%
Funnel [itz18]	11	7	7	2	1	0.001	0%	0%
GoPro Screw [xyt16]	36	54	23	3	1	0.001	2.88%	0.41%
Hose Adapter [dan19a]	8	14	8	3	1	0.001	0%	0%
Lock Shaft [jmc19]	4	6	6	5	1	0.001	1.33%	0.43%
Setup block [Chu19]	234	339	83	3	1	0.001	1.79%	0.01%
Speaker Grill [DiP19]	124	186	52	2	1	0.001	3.78%	1.79%
Turner's cube [br15]	116	210	42	19	1	0.001	0.01%	0%
Wire End Clamp [Not19]	220	34	14	2	1	0.001	0%	0%

During the process of creating our storage box, we round all edges that are orthogonal to the *Z*-axis. We can now delete this operation and ask our system to generate a selector for only the upper edges. This gives us the following sub-programme:

```
oshell = oshell.edges(''>Z'').
fillet(topAndBottomRadius)
```

Replacing this with the previous line of code gives us the requisite modification.

Appendix D: Detailed Results on CadQuery examples

In Section 5.2, we evaluated the utility of our technique for an interactive programming interface for CAD. We provide detailed results of the CadQuery examples in Table D1.

Appendix E: Experiments on Thingiverse Examples

The experiment presented here is an extension to the one presented in Section 5.2. The set of ground truth examples we use here is obtained from Thingiverse 's section of customizable designs. We chose 12 customizable designs representative of different application areas and of varying complexity. Figure E1 provides a snapshot of the chosen designs. The examples in Thingiverse 's customizable section are constructed using OpenSCAD [Kin19] and have a programmatic representation in CSG. For each example, we start with a blank programme, and re-construct the design in our system. Note that a direct translation of the ground truth programme is not possible due to a difference in the underlying CAD representation and available operations. We then use the same procedure as in Section 5.2, and synthesize a selection query wherever possible.

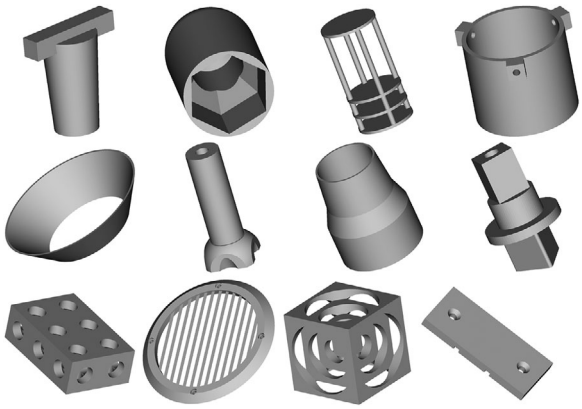


Figure E1: Thingiverse *designs (default parameters).*

Table E1: *Analysis of query size and synthesis runtime.*

Thingiverse examples			
	Min.	Avg.	Max.
# Vertices	4	74	234
# Edges	6	89	339
# Faces	6	26	83
# Queries	1	4	19
Query size	1	1	1
Time (s.)	0.001	0.001	0.001

We report runtimes of the synthesis process along with the complexity of the designs in Table E1. As OpenSCAD does not have a query language, an analysis of correctness of the synthesized queries is done using random sampling of the parameter space of the ground truth design and comparing the resulting mesh to the one generated using synthesized selectors (50 random samples). We compare meshes by calculating the Hausdorff distance between them using MeshLab. As parameter values often change the size of the overall design, we divide the Hausdorff distance by the length of the diagonal of the bounding box of the ground truth model. This is our mesh error metric. We plot errors for each example in Figure E2. The small error values indicate that the synthesized selectors are robust, and there are no unexpected side effects. Indeed, most of the errors here are caused due to the difference in the underlying CAD libraries. This can be confirmed by visually inspecting the meshes with the most error (>1%) in Figures E3–E7.

Appendix F: User Study Details

The user study was done by six participants (all male, 20–35 years of age). They had varying degrees of experience with CAD, with some being beginners with less than 1 year of experience, some with more than 3 years of experience, and some who fell in-between. They got 20 min on each, Programmatic and Programmatic + GUI interface. They were asked to write selection queries to complete some selected designs. The interface they were presented with is depicted in Figure F1. Participants were counter-balanced between doing the Programmatic interface and the Programatic + GUI interface first. All the designs that they had to complete were also counter-balanced and shuffled randomly. Participants were free to do the designs in any order or skip some if they did not want to finish them. Participants were not warned when their selection in the GUI or the query they wrote was incorrect, but were generally aided through questions about the interface and any technical queries they had. In the end, they filled out a post-study questionnaire, where they were asked questions about their experience on the two interfaces. The results of this questionnaire have been discussed in Section 5.3.

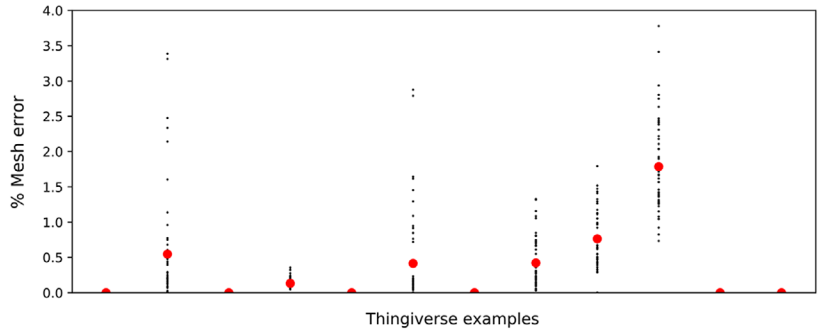


Figure E2: *% Mesh error on the* Thingiverse *examples (total 51 synthesized queries), enumerated on the* X*-axis (ordered as in Figure E1). Black dots represent mesh errors of each random sample. Red dots represent the average mesh error of each example.*

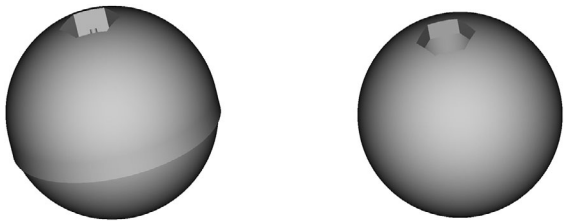


Figure E3: *Bolt cap (ground truth on the right). The mesh error is 3.39%.*

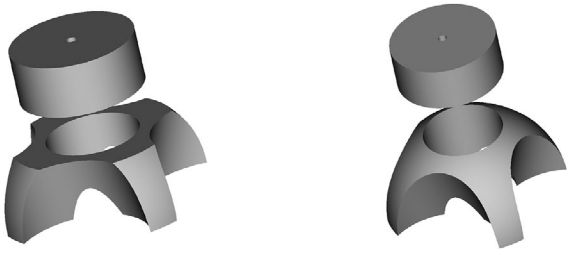


Figure E4: *GoPro Screw (ground truth on the right). The mesh error is 2.88%.*

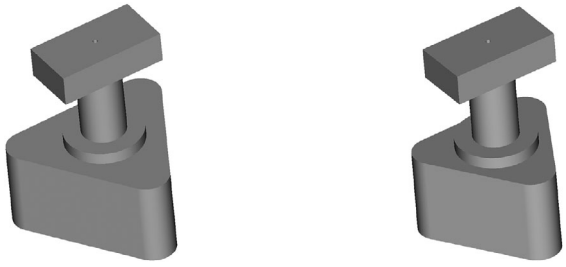


Figure E5: *Lock Shaft (ground truth on the right). The mesh error is 1.33%.*

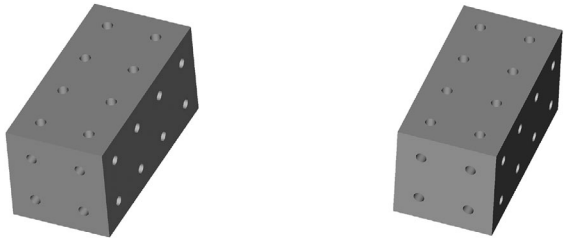
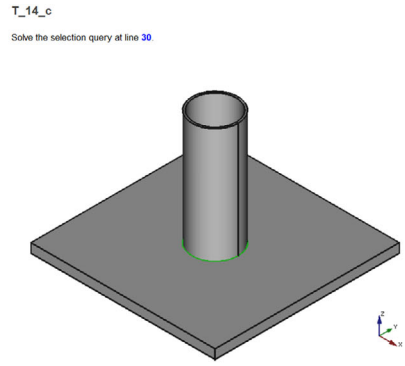


Figure E6: *Setup block (ground truth on the right). The mesh error is 1.79%.*

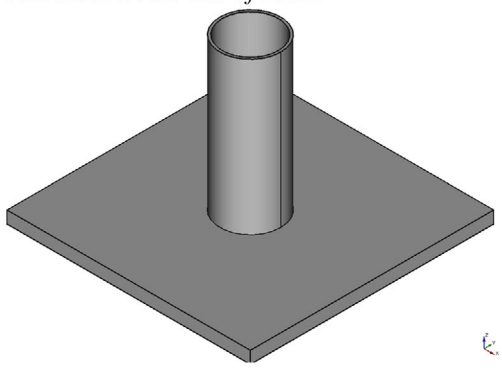
Figure E7: *Speaker Grill (ground truth on the right). The mesh error is 3.78%.*



(a) *Screenshot of the element(s) to be selected and the relevant line number for the query.*

```python
# ---------------------------------
# Trial 14
# Fill up the query corresponding  to selection(s) in the screenshot(s)
# ---------------------------------
import cadquery as cq
import Part

# Program variables
base_l = 200.0
base_w = 200.0
base_h = 10.0
base_r = 40.0
tube_r = 25.0
tube_h = 125.0
tube_shell_r = -2.0
fillet_r = 6.0

# Make the base
base = cq.Workplane("XY").box(base_l, base_w, base_h)

# See Trial 14 (a) selection
tube = base.faces(">Z").circle(tube_r).extrude(tube_h, combine=False)

# See Trial 14 (b) selection
tube = tube.faces("TODO").shell(tube_shell_r)

result = base.union(tube)

# See Trial 14 (c) selection
result = result.edges("TODO").fillet(fillet_r)

# Displays the result of this script
Part.show(result.toFreecad())

# ---------------------------------
# For experimenter use only: this example corresponds to FilamentStorage
```

(b) *The program representation of the design with the relevant line of code.*



(c) *GUI representation (when applicable), where the relevant element(s) can be directly selected and the query automatically synthesized.*

Figure F1: *Participants were presented two or three tabs depending on whether they were using the Programmatic or the Programmatic + GUI interface.*

[bel19] belliveaul: Air mattress plug, 2019. URL: https://www.thingiverse.com/thing:3818734. Accessed October 2019.

[br15] bpl rfe: Turners Cube, 2015. URL: https://www.thingiverse.com/thing:1072045. Accessed October 2019.

[Chu19] Chunkmunk01: Parametric Setup Block, 2019. URL: https://www.thingiverse.com/thing:3749463. Accessed October 2019.

[dan19a] danieln: Parametric Hose or Pipe Adapter, 2019. URL: https://www.thingiverse.com/thing:3819146. Accessed October 2019.

[dan19b] dannyprince: Configurable Electronics bay for Arduino Nano, 2019. URL: https://www.thingiverse.com/thing:3859829. Accessed October 2019.

[DiP19] DiPi92: Simple customizable speaker grill, 2019. URL: https://www.thingiverse.com/thing:3834348. Accessed October 2019.

[itz18] itzco: Customizable funnel, 2018. URL: https://www.thingiverse.com/thing:3243733. Accessed October 2019.

[jam19] james_lan: Meade ETX80 replacement eyepiece holder, 2019. URL: https://www.thingiverse.com/thing:3811422. Accessed October 2019.

[jmc19] jmcbassett: Meter box lock shaft, 2019. URL: https://www.thingiverse.com/thing:3841621. Accessed October 2019.

[Nat19] Naturaltangent: Parametric bolt cap, 2019. URL: https://www.thingiverse.com/thing:3814727. Accessed October 2019.

[Not19] Notace: Parametric Setup Block, 2019. URL: https://www.thingiverse.com/thing:3870946. Accessed October 2019.

[xyt16] xythobuz: Customizable long GoPro screw knob, 2016. URL: https://www.thingiverse.com/thing:1643650. Accessed October 2019.