



Constraint Synthesis for Parametric CAD

Aman Mathur  and Damien Zufferey 

MPI-SWS, Germany

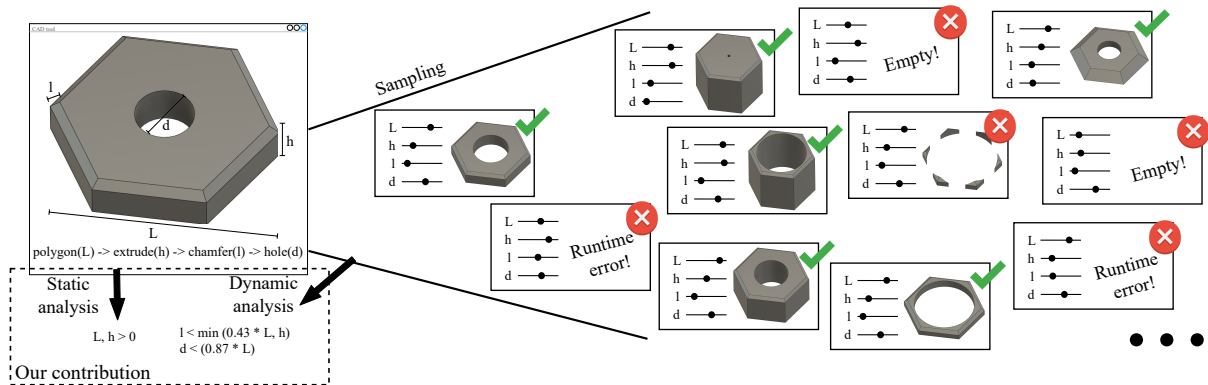


Figure 1: We present a technique for the automatic synthesis of constraints to CAD parameters. Using a mix of static and dynamic program analysis, we restrict the parameter space of designs to only those configurations that produce valid final objects.

Abstract

Parametric CAD, in conjunction with 3D-printing, is democratizing design and production pipelines. End-users can easily change parameters of publicly available designs, and 3D-print the customized objects. In research and industry, parametric designs are being used to find optimal, or unique final objects. Unfortunately, for most designs, many combinations of parameter values are invalid. Restricting the parameter space of designs to only the valid configurations is a difficult problem. Most publicly available designs do not contain this information. Using ideas from program analysis, we synthesize constraints on parameters of parametric designs. Some constraints are synthesized statically, by exploiting implicit assumptions of the design process. Several others are inferred by evaluating the design on many different samples, and then constructing and solving hypotheses. Our approach is effective at finding constraints on parameter values for a wide variety of parametric designs, with a very small runtime cost, in the order of seconds.

CCS Concepts

- *Computing methodologies* → *Shape analysis*;

1. Introduction

Parametric design is a popular design methodology in which designers encode their *design intent* by specifying the sequence of operations in the design. This enables end-users to change parameters of the design, which after a re-evaluation of the sequence of operations, results in different variations of final objects. Parametric design tools such as AUTODESK FUSION 360, SOLIDWORKS, PTC CREO, OPENSCAD, and FREECAD are ubiquitous in the industry and ‘maker’ community. Moreover, websites such as THINGIVERSE, where users share, remix, and customize parametric designs, are extremely popular.

A serious limitation of current customization pipelines is that the relationship between CAD parameters is often unknown. In practice, only a small subset of parameter values lead to valid final objects. Unfortunately, this information is usually not conveyed in shared designs. Currently, there is also no good way of inferring this information automatically. This is a hard problem because designs can have many parameters, each of which influences the validity of the final object. This complexity grows as designs involve more CAD operations and parameters. At the same time, constraints on design parameters are extremely valuable to end-users. It provides them a high-level perspective on how the parameters

interact, and guides them towards valid final objects. Customizable designs that include information on parameter constraints are much more popular on THINGIVERSE. Still, most new designs do not have this information. In many research and industrial use cases, parameters of designs are sampled to find optimal (according to metrics like weight, strength, stability, etc.) or otherwise unique final objects [SWG*18]. Depending on the design under test, a significant proportion of these samples lead to invalid final objects, and are therefore wasted. This can be drastically improved if information on the constraints to design parameters is available.

In this paper, we present an approach for synthesizing CAD parameter constraints automatically. The central intuition behind our technique is treating parametric designs as traditional programs, and applying ideas from program analysis for synthesizing parameter constraints. Each design can be broken down into its constituent CAD operations, and each operation provides us with some information on constraints to its parameters. Some of this information can be collected *statically*, i.e., without evaluating the design on any concrete parameter value. For example, when constructing a circle, we can be sure that its diameter is > 0 ; when making a counter-bore hole, we can be sure that the inner hole depth is \geq outer hole depth, and that this depth is $<$ the diagonal of the bounding box of the intermediate object on which the operation is performed.

After a static analysis pass, if there are still parameters without known constraints, we move to *dynamic analysis*, i.e., we try to infer constraints based on evidence from evaluating the design on many concrete parameter values. Such a two-pronged approach (static & dynamic analysis) has been successful at finding bugs and program invariants in traditional computer programs (for example, see [ECH*01]). We adapt these ideas for parametric CAD, first by defining what it means for a CAD operation to *fail*, or for an object to be *invalid*. Then, we describe a few inference rules that help synthesize some parameter constraints statically. Finally, we present a novel guess-and-check algorithm for dynamically synthesizing constraints.

Let us present a concrete example of our technique in action. Consider the design in Figure 1. The design consists of first making a hexagon of diameter L . Then, this is extruded to a height h . Next, the 6 top-most edges are chamfered using a length of l . Finally, a hole of diameter d is drilled on the top-most face.

Now, our technique can statically infer the following: (i) as L and h create new geometry, they should be > 0 , (ii) the chamfer operation cannot succeed (run-time failure) if the value of l is very high, (iii) if the hole diameter d is very large, then the final object may be empty, or fractured (segmented into unconnected components). Additionally, we have rough constraints for l and d . They should both be less than the bounding box diagonal of the intermediate object on which they operate. Precise constraints for l and d need to be found, for which we move to dynamic analysis.

Our dynamic analysis algorithm samples the design over many parameter values, and tries to construct and fit hypotheses based on the observed runs. Our hypothesis generator proposes hypotheses of increasing complexity. To check whether the hypothesis fits, and to synthesize a precise constraint, we use mixed integer linear programming. For our current example, this technique synthesizes correct constraints for both, l and d : $l < \min(0.43 * L, h)$,

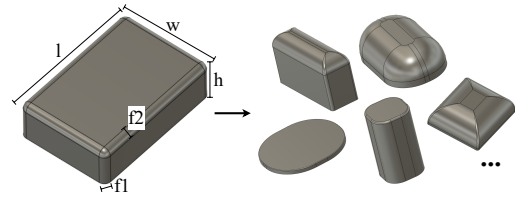


Figure 2: A design with 5 parameters, and some valid variations.

and $d < 0.87 * L$. Our automated technique finds all constraints in just 48 seconds. Clearly, coming up with these constraints by hand would be challenging.

This is hardly a toy example. Another design taken from the same dataset [LWJ*21] is presented in Figure 2. Just 5 parameters here encapsulate a wide-variety of different final objects. To add to this, sampling this design randomly yields only 3% of the samples to produce valid final objects (many configurations fail with runtime errors). In general, designs may have many more parameters, and much sparser space of valid designs. An automated technique for finding parameter constraints, therefore, can be quite useful. We have evaluated our technique on designs with up to 12 parameters, and as little as 0.1% of configurations valid (see Section 4).

The rest of the paper is organized as follows. In Section 2, we describe some related work. Then, in Section 3, we present details about our proposed approach. In Section 4, we evaluate our technique on a public dataset, and present our accuracy and runtime characteristics. Finally, in Section 5, we conclude, and discuss opportunities for future work.

2. Related Work

Constraints on design parameters. We are not the first to propose the idea of synthesizing CAD parameter constraints. FAB FORMS [SSM15] employs user-specified validity conditions to synthesize constraints for design parameters. These constraints are then embedded in an interactive design explorer that enables a quick preview of the various (valid) final objects. FAB FORMS, however, takes between several hours to several weeks for pre-computation. Part of this is because they, like other similar techniques [YYPM11, TSG*14], work (albeit indirectly) with meshes. Moreover, many of FAB FORMS validity checks are costly (e.g. finite element method). Our work uses the higher-level, and widely popular Boundary Representation (B-rep) [Str06]. We perform validity checks within the representation, which is more efficient than checks on meshes. The use of B-rep and its operations also enables us to support more designs, and more design workflows than specialized constrained editing tools [BWSK12, SSL*14]. Our validity conditions come from implicit assumptions of CAD operations and design methodology. We can therefore quickly eliminate many designs that are generally accepted to be invalid. Then, if more complicated checks are required, these can be performed on a much smaller subset of final objects.

Generative design in the context of 3D CAD has gained widespread prominence in research and industry. The idea is to generate a large number of objects based on some abstract metric,

such as user choice [XZCOC12,FRS*12], or physical properties by sampling parametric designs [SWG*18]. These techniques are expensive, and can benefit greatly by a constrained parameter space, as can be provided by our technique.

Program analysis/synthesis. The idea of applying program language techniques to design is not new. Some work already exists on synthesizing programs for vector graphics [HLC19,CHSA16]. Recently, this has been extended to parametric CAD [MPZ20]. Our work shares a similar intuition. We analyze CAD representations as *traditional* programs [NNH04]. We collect and use implicit assumptions [ECH*01] of CAD operations and learn constraints on the design parameters that avoid invalid configurations. This is inspired by the work on synthesizing program invariants that are understandable by programmers [ECGN00,FL01,GLMN14]. Our synthesis approach is based on enumeratively proposing and checking hypotheses of increasing complexity, as is standard in Syntax Guided Synthesis (SyGuS) [ABD*15].

3. Framework

We now provide details of our constraint synthesis technique. We first introduce our CAD representation, and define what it means for CAD objects to be *valid* or *invalid*. Then, we present some rules for statically inferring constraints. Finally, we discuss our dynamic constraint synthesis algorithm.

3.1. CAD and Validity

Though many different representations for parametric CAD exist, Boundary Representation (B-rep) [Str06] is by far the most popular. B-rep offers a rich collection of high-level operations to create and modify 3D shapes. Our CAD interface (CADQUERY) is based on top of the OPEN CASCADE B-rep kernel. Most other open-source B-rep tools also use OPEN CASCADE, and therefore support similar operations and internal representations.

We now concretely define our notion of invalidity of CAD objects. We define objects, and the parameter configurations that evaluate them as *invalid*, if either: (i) an error during evaluation occurs, (ii) the final (or an intermediary) object is empty, (iii) the final (or an intermediary) object is fractured (unconnected components), or (iv) an operation specific assumption is not satisfied. The first criterion is a clear flag for invalidity. The next two capture implicit assumptions of designers, i.e., not to have an empty object, or an object with a broken topology. The final criterion captures operation specific assumptions. For example, if a vertex is chosen for drilling a hole, the implicit assumption is that this vertex must lie on one of the faces of the object.

In standard programming, it is common to have a set of pre-conditions, say *pre*, and a set of postconditions, say *post*, such that for a given operation *op*, if the pre-conditions hold, then after the operation the post-conditions also hold. This is usually written in the form of a Hoare triple [Hoa69]: $\{pre\}op\{post\}$. In our setting, the post-conditions are the validity conditions as presented before. Our aim is to synthesize the *weakest* pre-condition *pre*, which means, $\forall pre'. \{pre'\}op\{post\} \Rightarrow (pre' \Rightarrow pre)$. All parameter configurations that satisfy our constraints lead to valid designs, and those that do not, result in invalid designs.

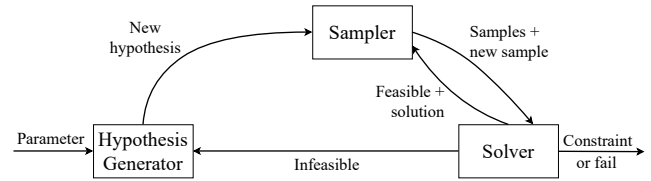


Figure 3: Overview of our dynamic synthesis algorithm. We take the parameters for which constraints need to be synthesized. The Hypothesis Generator (Section 3.3.1) generates hypotheses of increasing complexity for these. The Sampler (Section 3.3.2) samples the parameter space, and evaluates the design/operations. The Solver (Section 3.3.3) uses samples from the Sampler to formulate each hypothesis into a linear program. If a solution to this is feasible, then more samples are collected to check and refine the constraint derived from the hypothesis (until a fixed threshold of samples is reached). Else, the hypothesis is rejected, and the Hypothesis Generator generates a new one.

3.2. Static Analysis

The static analysis part of our technique looks at the sequence of operations in the design, and comes up with an initial set of parameter constraints. The following inference rules capture most of our statically inferred constraints:

$$\frac{P_i, P_{ii}, P_{iii}, \dots > 0}{\text{CreateGeometry}(p_i, p_{ii}, p_{iii}, \dots)}$$

$$\frac{d_{in} \leq d_{out}, \quad h_{out} \leq h_{in}}{\text{CounterSink/BoreHole}(d_{in}, d_{out}, h_{out}, h_{in})}$$

$$\frac{p < BB().Diagonal}{\text{Fillet/Chamfer/Hole}(p)} \quad \frac{t > -BB().Diagonal}{\text{Shell}(t)}$$

The first rule captures CAD operations that are responsible for creating new geometry (excluding geometry that is later used for a cut or difference operation). Operations such as creating a `circle`, `box`, `extrude`, etc. are constrained using this rule, so are intermediary operations such as `offsets`, which later create geometry using `lofts`, for example. The second rule captures easily encoded constraints that must hold for these operations to succeed. The last two rules capture rough constraints. These rough constraints can be used later by the dynamic analysis to more effectively find precise constraints. $BB().Diagonal$ stands for the bounding box diagonal of the intermediate object on which these operations are applied. Though the exact value of this expression would require evaluating the design on concrete parameter values, it is an over-approximated bound that would surely hold for any valid evaluation.

3.3. Dynamic Analysis

An overview of our dynamic analysis approach is presented in Figure 3. We now discuss each component individually.

3.3.1. Hypothesis Generator

We follow the approach of enumerative program synthesis [ABD*15], and propose expressions of increasing complexity. The

following is the grammar of a hypothesis h in our system:

$$\begin{aligned} \langle H \rangle &\models \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \mid \langle H \rangle \wedge \langle H \rangle \\ \langle \text{expr} \rangle &\models \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \min(\langle \text{atom} \rangle, \langle \text{atom} \rangle) \\ &\quad \mid \max(\langle \text{atom} \rangle, \langle \text{atom} \rangle) \mid \langle \text{atom} \rangle \\ \langle \text{atom} \rangle &\models \langle c \rangle \mid \langle c \rangle \cdot \langle p \rangle \\ \langle \text{op} \rangle &\models < \mid \leq \end{aligned}$$

where c denotes a constant in \mathbb{R} , and p denotes a parameter of the design. Our hypotheses are essentially linear expressions augmented with min and max.

3.3.2. Sampler

The Sampler samples different parameter values (until a threshold maximum number of samples is reached) and evaluates the design on these. If an evaluation is invalid, we also track which operation is responsible. Each sample generated by the Sampler respects already known constraints. This is done by sampling randomly until such a sample is found. If a hypothesis is found to be feasible, then we sample at the boundary of this hypothesis so as to refine the values of constants in the hypothesis.

3.3.3. Solver

Each proposed hypothesis needs to be checked, and the missing coefficients need to be found. For this, we use mixed integer linear programming. We have a set of samples from the Sampler, and we also know, for each operation, if these samples are valid or invalid. Our hypotheses can be converted into a conjunction of n linear inequalities over the parameters P . Then for a valid sample s_+ , we simply substitute the sample values:

$$\bigwedge_{i=1}^n \left(\sum_{p \in P} c_{i,p} \cdot s_+[p] \geq d_i \right)$$

where $c_{i,q}$ and d_i are the constants whose values need to be determined. For an invalid sample s_- , we need the following inequality:

$$\bigvee_{i=1}^n \left(\sum_{p \in P} c_{i,p} \cdot s_-[p] < d_i \right)$$

However, this introduces disjunctions, which are not natively supported in linear programming. To get rid of these, we use a variation of the Big-M method [GNS09] to generate the following inequalities:

$$\begin{aligned} \bigwedge_{i=1}^n \left(\sum_{p \in P} c_{i,p} \cdot s_-[p] < d_i - M \cdot m_i \right) \\ \bigwedge_{i=1}^n m_i \in \{0, 1\}, \quad 0 \leq \sum_{i=1}^n m_i < n \end{aligned}$$

where M is a sufficiently large constant that overpowers the rest of the inequality, and the m_i act as switches that ensure at least one disjunct holds.

Hypotheses that use `min` and `max` can be turned into linear inequalities by:

$$\begin{aligned} e \leq \min(e_1, e_2) &\Leftrightarrow (e \leq e_1 \wedge e \leq e_2) \\ e \leq \max(e_1, e_2) &\Leftrightarrow (e \leq e_1 \vee e \leq e_2) \end{aligned}$$

and removing the disjunction as explained above.

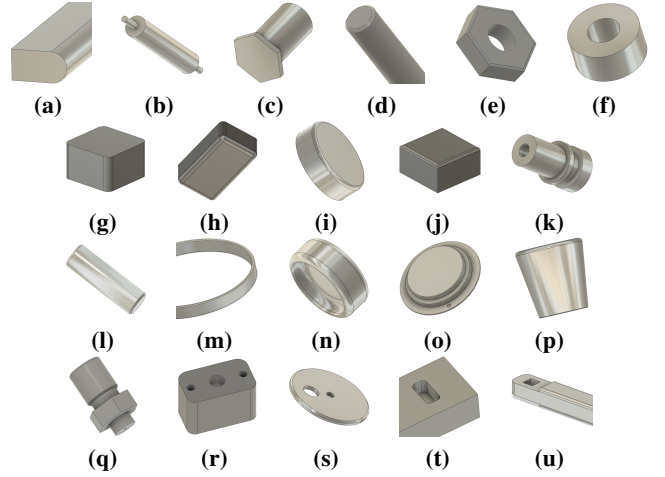


Figure 4: Objects from the Fusion 360 Segmentation dataset used in the experiment (some views clipped to show prominent features).

4. Evaluation

We now provide some implementation details, and present experimental evidence on the efficacy of our technique for synthesizing CAD parameter constraints.

4.1. Implementation

Our implementation consists of approximately 2000 lines of Python code. The source code of our project, as well as the experiments in the evaluation are available at: <https://gitlab.mpi-sws.org/mathur/constraints-cad>. We use CADQUERY (version 2.0) as our CAD interface. CAD validity checks are performed using PYTHONOCC, which provides access to most of the underlying OPEN CASCADE structures. The synthesis procedure uses PULP as the mixed integer linear programming library, and GLPK as the solver.

4.2. Experiments

We now evaluate our technique on some real-world designs from the recently released, and publicly accessible Fusion 360 Segmentation Dataset [LWJ*21].

4.2.1. Experimental Procedure

We re-design objects in the dataset to an equivalent B-rep sequence of operations in CADQUERY. Due to practical considerations, we consider the first 21 compatible designs (excluding duplicates, trivial objects, and designs involving sketching or non-straightforward mappings to CADQUERY syntax). The designs we use for our evaluation are presented in Fig. 4, labelled (a) - (u). The number of parameters in these designs range from 3 to 12, with an average of 5.5. For our dynamic analysis, we set the threshold of maximum number of samples for each constraint to 320 samples; the Hypothesis Generator generates hypotheses with at most 3 atomic predicates. The experiments are performed on a computer with an Intel Core i7-7820HK processor, 32 GB of RAM, and running on Windows 10.

Table 1: Synthesis of constraints, and runtimes for the various designs under test. For each design, we report the total number of parameters, and for how many we can synthesize constraints through static or dynamic analysis. The Success corresponds to how many random samples (out of 1000) lead to valid designs without constraints (Before) or with the synthesized constraints (After).

Id.	Parameters			Runtime	Success	
	Total	Static	Dynamic		Before	After
Fully solved; statically (14%)						
(a)	3	3	0	0.1 s	100 %	100 %
(b)	6	6	0	0 s	100 %	100 %
(c)	4	4	0	0 s	100 %	100 %
Fully solved; statically & dynamically (52%)						
(d)	3	2	1	7.6 s	19.6 %	99.4 %
(e)	4	2	2	48.0 s	10.6 %	98.7 %
(f)	3	2	1	7.4 s	46.2 %	100 %
(g)	4	3	1	6.8 s	15.5 %	98.2 %
(h)	5	3	2	45.6 s	3.1 %	98.6 %
(i)	3	2	1	5.9 s	19.8 %	99.2 %
(j)	4	3	1	176.4 s	14.8 %	97.7 %
(k)	8	6	2	48.8 s	17.6 %	83.9 %
(l)	3	2	1	7.3 s	15.7 %	98.4 %
(m)	4	2	2	22.4 s	4.2 %	83.1 %
(n)	4	3	1	8.4 s	16.3 %	98.2 %
Partially solved (33%)						
(o)	12	7	3	121.3 s	4.8 %	86.3 %
(p)	4	3	0	3.0 s	22.6 %	40.4 %
(q)	11	9	1	44.8 s	18.2 %	79.8 %
(r)	7	3	3	54.9 s	1.3 %	9.4 %
(s)	6	2	2	44.2 s	10.4 %	28.0 %
(t)	8	3	2	21.2 s	4.2 %	86.0 %
(u)	10	6	4	67.5 s	0.1 %	71.3 %
Averages						
	5.5	3.6	1.4	36.1 s	26 %	83.6 %

4.2.2. Results and Discussion

The results of our evaluation are summarized in Table 1. We are able to synthesize constraints for almost all the designs under test. For a significant majority of the designs, we synthesize correct constraints for all the parameters involved, thereby segmenting the space of valid designs with a high degree of accuracy. There are also some designs for which we cannot synthesize constraints for all design parameters. In such cases, we synthesize constraints for some parameters, and are still able to segment the space of valid designs fairly well. There is just one design for which we cannot synthesize any constraints. For this design, we still do better than naive sampling of the parameter space because we can identify samples that have no chance of passing (restricting fillet radius to $<$ bounding box diagonal). Our technique is reasonably fast (median: 21.2 s, average: 36.1 s, max: 176.4 s). The constraints we synthesize improve sampling efficiency from 26% to 83.6% on average. We organize further discussion by how the constraints are generated.

Fully solved statically. There are 3 designs where the static analysis finds *all* the constraints. Our method has a negligible runtime for these cases. The results may seem a bit uninteresting, as all

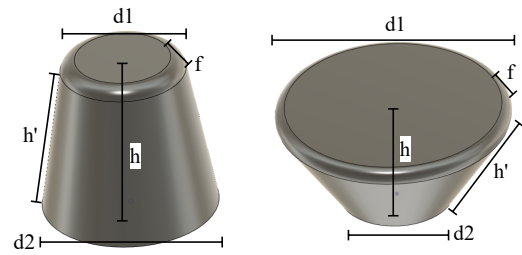


Figure 5: Two variations of the design (p). The parameter f depends on $d1$ and $d2$, as well as h' . However, h' is not an exposed parameter; but h (the offset between the two circles) is. We know that f must be less than the bounding box diagonal. However, the precise constraint cannot be synthesized, as doing so would require non-linear hypotheses with geometric functions.

points in the parameter space are valid, but the synthesized constraints can still be useful to end-users to get a general overview of the design, and how the parameters interact.

Fully solved statically and dynamically. There are 11 designs that can be solved fully via a combination of static and dynamic analysis. The solved constraints fall shy of 100% accuracy, but are in-fact correct (verified manually). The small errors noticed here primarily show up when working with curved shapes, and are due to numerical instabilities in the underlying CAD kernel. As depicted in Table 1, the designs here have a uniformly low success rate when sampling naively. With just a small runtime overhead, we are able to synthesize accurate constraints for *all* parameters of these designs.

We already discussed (e) from this segment in Section 1. Let us now look at our solution for the design (h), as depicted in Figure 4. A detailed view of its parameters and variations of final objects was presented earlier, in Figure 2. The design consists of the following sequence of operations: (i) a box with dimensions l , w , and h is created, (ii) the 4 edges parallel to the Y-axis are filleted with radius $f1$, (iii) the 4 edges on top in the Y-axis are filleted with radius $f2$. Our static analyzer quickly comes up with constraints for l , w , and h . Then, our dynamic analyzer is asked to find constraints for $f1$ and $f2$. They cannot take any arbitrary values, and *must* take values less than the bounding box diagonal of their respective intermediate objects. We enumeratively build hypotheses for both, $f1$, and $f2$. The simplest inequalities do not work out. We first find a solution for $f1$ of the form: $f1 < c * \min(l, h)$, with $c = 0.478$ initially. This is later refined to $c = 0.499$. Similarly, we find a constraint for $f2$ as: $f2 < 0.5 * \min(l, w, h)$.

Partially Solved. For 7 designs in the experiment, we are only able to synthesize constraints for some of the parameters involved. For the rest, our hypothesis generator cannot construct correctly fitting hypotheses. As shown in the final segment of Table 1, our technique still significantly improves the sampling success rate vis-à-vis random sampling. This is because in addition to finding correct constraints for at least some of the parameters involved, we can often eliminate configurations that fail for sure (for example, see Figure 5), or find an approximate constraint (see Figure 6).

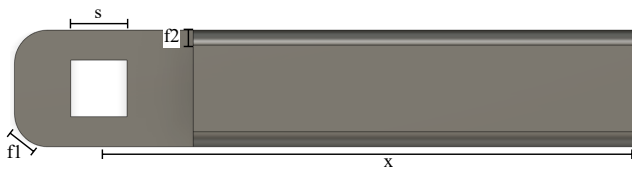


Figure 6: Incomplete top-view of the design (u). We synthesize correct constraints for $f1$, $f2$, and x . For s , we obtain $s < 3.31 * \max(1, w, h)$, which is an approximation.

5. Conclusion and Future Work

The power of parameterization is foundational to many modern applications of CAD. Our proposed technique is successful at synthesizing constraints for CAD parameters accurately and efficiently for a wide-variety of designs. However, designs can get arbitrarily complex, and we cannot directly synthesize many non-linear and geometric constraints. Supporting these would be an obvious next step, but this would also make the synthesis procedure significantly more complex. We may need to settle for *predictors*, rather than accurate constraints. In initial experiments, as is typical with machine learning, we also found an obvious trade-off between readability and accuracy. If knowledge about the valid space of designs is not human readable, it becomes useless for many end-users. Though such predictors may still be useful for optimization and generative techniques, misclassifications would be difficult to debug.

Acknowledgements

This research was funded in part by the Deutsche Forschungsgemeinschaft project 389792660-TRR 248 and by the European Research Council under the Grant Agreement 610150 (ERC Syn-ergy Grant ImPACT).

References

- [ABD*15] ALUR R., BODÍK R., DALLAL E., FISMAN D., GARG P., JUNIWAŁ G., KRESS-GAZIT H., MADHUSUDAN P., MARTIN M. K., RAGHOTHAMAN M., SAHA S., SESHIA S. A., SINGH R., SOLAR-LEZAMA A., TORLAK E., UDUPA A.: Syntax-guided synthesis. In *Dependable Software Systems Engineering*, Irlbeck M., Peled D. A., Pretschner A., (Eds.), vol. 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*. IOS Press, 2015, pp. 1–25. URL: <https://doi.org/10.3233/978-1-61499-495-4-1>. 3
- [BWSK12] BOKELOH M., WAND M., SEIDEL H.-P., KOLTUN V.: An algebraic model for parameterized shape editing. *ACM Trans. Graph.* 31, 4 (July 2012). URL: <https://doi.org/10.1145/2185520.2185574>. 2
- [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* (2016), pp. 341–354. 3
- [ECGN00] ERNST M. D., CZEISLER A., GRISWOLD W. G., NOTKIN D.: Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000* (2000), Ghezzi C., Jazayeri M., Wolf A. L., (Eds.), ACM, pp. 449–458. URL: <https://doi.org/10.1145/337180.337240>. 3
- [ECH*01] ENGLER D., CHEN D. Y., HALLEM S., CHOU A., CHELF B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), SOSP '01, Association for Computing Machinery, pp. 57–72. URL: <https://doi.org/10.1145/502034.502041>. 2, 3
- [FL01] FLANAGAN C., LEINO K. R. M.: Houdini, an annotation assistant for `esccjava`. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings* (2001), Oliveira J. N., Zave P., (Eds.), vol. 2021 of *Lecture Notes in Computer Science*, Springer, pp. 500–517. URL: https://doi.org/10.1007/3-540-45251-6_29. 3
- [FRS*12] FISHER M., RITCHIE D., SAVVA M., FUNKHOUSER T., HANRAHAN P.: Example-based synthesis of 3D object arrangements. *ACM Trans. Graph.* 31, 6 (Nov. 2012). URL: <https://doi.org/10.1145/2366145.2366154>. 3
- [GLMN14] GARG P., LÖDING C., MADHUSUDAN P., NEIDER D.: Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification* (2014), Springer, pp. 69–87. 3
- [GNS09] GRIVA I., NASH S. G., SOFER A.: *Linear and nonlinear optimization*, vol. 108. SIAM, 2009. 4
- [HLC19] HEMPEL B., LUBIN J., CHUGH R.: Sketch-n-Sketch: Output-directed programming for `svg`. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (2019), pp. 281–292. 3
- [Hoa69] HOARE C. A. R.: An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. URL: <https://doi.org/10.1145/363235.363259>. 3
- [LWJ*21] LAMBOURNE J. G., WILLIS K. D., JAYARAMAN P. K., SANGHI A., MELTZER P., SHAYANI H.: Brepnet: A topological message passing system for solid models. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2021). [arXiv:2104.00706](https://arxiv.org/abs/2104.00706). 2, 4
- [MPZ20] MATHUR A., PIRRON M., ZUFFEREY D.: Interactive programming for parametric CAD. In *Computer Graphics Forum* (2020), vol. 39, Wiley Online Library, pp. 408–425. 3
- [NNH04] NIELSON F., NIELSON H. R., HANKIN C.: *Principles of program analysis*. Springer Science & Business Media, 2004. 3
- [SSL*14] SCHULZ A., SHAMIR A., LEVIN D. I. W., SITTHI-AMORN P., MATUSIK W.: Design and fabrication by example. *ACM Trans. Graph.* 33, 4 (July 2014). URL: <https://doi.org/10.1145/2601097.2601127>. 2
- [SSM15] SHUGRINA M., SHAMIR A., MATUSIK W.: Fab forms: Customizable objects for fabrication with validity and geometry caching. *ACM Trans. Graph.* 34, 4 (July 2015). URL: <https://doi.org/10.1145/2766994.2>
- [Str06] STROUD I.: *Boundary representation modelling techniques*. Springer Science & Business Media, 2006. 2, 3
- [SWG*18] SCHULZ A., WANG H., GRINSPUN E., SOLOMON J., MATUSIK W.: Interactive exploration of design trade-offs. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–14. 2, 3
- [TSG*14] TANG C., SUN X., GOMES A., WALLNER J., POTTMANN H.: Form-finding with polyhedral meshes made simple. *ACM Trans. Graph.* 33, 4 (July 2014). URL: <https://doi.org/10.1145/2601097.2601213>. 2
- [XZCOC12] XU K., ZHANG H., COHEN-OR D., CHEN B.: Fit and diverse: Set evolution for inspiring 3D shape galleries. *ACM Trans. Graph.* 31, 4 (July 2012). URL: <https://doi.org/10.1145/2185520.2185553>. 3
- [YYPM11] YANG Y.-L., YANG Y.-J., POTTMANN H., MITRA N. J.: Shape space exploration of constrained meshes. *ACM Trans. Graph.* 30, 6 (2011), 124. 2