InverseCSG: Automatic Conversion of 3D Models to CSG Trees

TAO DU, Massachusetts Institute of Technology
JEEVANA PRIYA INALA, Massachusetts Institute of Technology
YEWEN PU, Massachusetts Institute of Technology
ANDREW SPIELBERG, Massachusetts Institute of Technology
ADRIANA SCHULZ, Massachusetts Institute of Technology
DANIELA RUS, Massachusetts Institute of Technology
ARMANDO SOLAR-LEZAMA, Massachusetts Institute of Technology
WOJCIECH MATUSIK, Massachusetts Institute of Technology

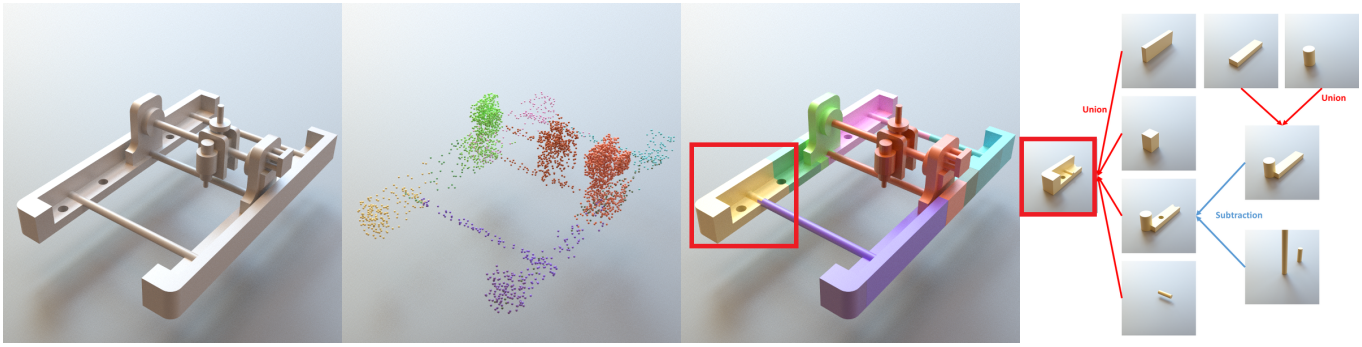

Fig. 1. We provide a system that takes as input a mesh file (left) and outputs its constructive solid geometry (CSG) representation. The three key ideas are a) use of carefully designed point samples to guide a purely discrete search problem (points, middle left), b) a divide-and-conquer algorithm to segment problems to ensure success (colors, middle left), and c) use of program synthesis techniques to solve the hard discrete search problem in each segment. The output CSG structure (middle right) correctly infers over 50 solid primitives and 18 boolean operators. A part of the solution (red box) is extracted for demonstration (right).

While computer-aided design is a major part of many modern manufacturing pipelines, the design files typically generated describe raw geometry. Lost in this representation is the procedure by which these designs were generated. In this paper, we present a method for reverse-engineering the process by which 3D models may have been generated, in the language of constructive solid geometry (CSG). Observing that CSG is a formal grammar, we formulate this inverse CSG problem as a program synthesis problem. Our solution is an algorithm that couples geometric processing with state-of-the-art program synthesis techniques. In this scheme, geometric processing is used to convert the mixed discrete and continuous domain of CSG trees to a pure discrete domain where modern program synthesizers excel. We demonstrate the efficiency and scalability of our algorithm on several different examples, including those with over 100 primitive parts. We show that our algorithm is able to find simple programs which are close to the ground truth, and demonstrate our method's applicability in mesh re-editing. Finally, we compare our method to prior state-of-the-art. We demonstrate that our algorithm dominates previous methods in terms of resulting CSG compactness and runtime, and can handle far more complex input meshes than any previous method.

CCS Concepts: • **Computing methodologies** → **Mesh geometry models**; *Parametric curve and surface models*;

Additional Key Words and Phrases: Procedural Modeling, CSG, CAD

ACM Reference Format:
Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: Automatic Conversion of 3D Models to CSG Trees. *ACM Trans. Graph.* 37, 6, Article 213 (November 2018), 16 pages. https://doi.org/10.1145/3272127.3275006

Authors' addresses: Tao Du, Massachusetts Institute of Technology, taodu@csail.mit.edu; Jeevana Priya Inala, Massachusetts Institute of Technology; Yewen Pu, Massachusetts Institute of Technology; Andrew Spielberg, Massachusetts Institute of Technology; Adriana Schulz, Massachusetts Institute of Technology; Daniela Rus, Massachusetts Institute of Technology; Armando Solar-Lezama, Massachusetts Institute of Technology; Wojciech Matusik, Massachusetts Institute of Technology.

0730-0301/2018/11-ART213
https://doi.org/10.1145/3272127.3275006

1 INTRODUCTION

Computer-aided design (CAD) software has become a key part of many modern mass-manufacturing pipelines over the last few decades. CAD tools are parametric by design, allowing designers to create easily modifiable shapes. This enables engineers to iterate over the design parameters to improve the performance of objects or to adapt existing designs so that they can be reused in new scenarios. For this reason, there has been a growing interest in using parametric CAD representations for fabrication-oriented design exploration and optimization algorithms [Schulz et al. 2017; Shugrina et al. 2015].

Unfortunately, CAD procedures are rarely readily available with released models. The constructive parametric representations are internal to CAD systems, which typically only allow for exporting a design as a 3D mesh or a boundary representation (B-Rep). As a result, there is a vast quantity of important legacy models whose original procedural definitions have been lost in the years since their creation. In addition, many crafted objects were not originally designed in CAD systems at all, and many more products undergo shop floor changes that make the original CAD files inaccurate. For such models, the only available shape representations are 3D scans of the manufactured design that necessarily incorporate noise due to imperfections in the scanning process.

New techniques to reverse engineer CAD models from 3D shapes, such as Monte Carlo sampling methods and genetic algorithms, have attracted the interest of many researchers. Central to any technique is an expressive and concise representation. A natural choice of parametric representation is constructive solid geometry (CSG) as it is a well understood, widely accepted staple in modern CAD systems and is compact in its representation. CSG encodes geometries as trees that are constructed by recursively applying boolean operators to primitive shapes [Requicha and Rossignac 1992]. Theory for the automatic conversion of 3D models to CSG trees has been widely studied for the past 20 years.

However, the current leading techniques either do not scale well for large problems [Fayolle and Pasko 2016] or cannot produce compact representations [Buchele and Crawford 2004]. Furthermore, these methods assume the input shape is an exact B-Rep, as opposed to a 3D model that can come from a low-resolution mesh or a noisy 3D scan. In this paper, we propose a new, scalable approach to reverse engineer CAD models based on the realization that CSG is simply a class of computer programs. This means that generating a CSG tree can be framed as a *program synthesis* problem.

Program synthesis is a process by which computer programs are generated from descriptions of their intended behavior. These techniques seek to generate programs which not only *satisfy* their specifications, but also do so as parsimoniously as possible. Program synthesis has proven effective in a wide range of problem domains, including synthesizing entity matching rules for databases [Singh et al. 2017], inferring excel formulas [Gulwani et al. 2012], and experiment design in biology [Koksal et al. 2013]. Our work builds on *constraint-based synthesis*, a specific class of synthesis algorithms that works by symbolically representing a space of candidate programs and framing the search for a correct program in this space as a constraint satisfaction problem [Solar-Lezama 2008]. Constraint-based methods can scale to large search spaces by leveraging the capabilities of modern constraint-solvers and are easily extended to cover different sets of shape primitives by simply including them in the search space. This is in contrast to purely deductive methods that start from a naive valid solution and apply deductive rules to improve the program's quality (typically measured by its length). These methods can have a hard time finding optimal solutions without the help of carefully crafted rules tailored to each primitive.

There are three main challenges in applying constraint-based program synthesis techniques to the task of reverse engineering CAD models. First, the most scalable constraint-based synthesis systems work by reducing problems to boolean satisfiability (SAT), a purely discrete problem. This is a good fit for discovering the boolean structure of the CSG model, but CAD models also involve continuous parameters (*e.g.* the positions and extents of primitive shapes). Second, the correctness of the program to be synthesized is naturally defined as a geometric constraint, since the 3D shape described by the synthesized CSG program must occupy the same volume in space as the volume contained inside the input mesh. However, this requirement is too complex to be used as a basis for program synthesis, so we need to translate this high-level requirement into a set of constraints on the program behavior that the synthesizer can use to efficiently prune the space of possible programs. Third, noisy inputs (*e.g.* a mesh generated from a scan) generate inconsistencies in the specification with respect to our limited primitive shapes, which could lead to contradicting constraints that cannot be satisfied.

In this paper, we address these challenges to provide a complete and scalable pipeline for generating a compact CSG tree from noisy data that can be approximated at various levels. We address the issue of a mixed continuous and discrete search problem together with the issue of noise by breaking the search into two steps: First, we use robust primitive detection methods based on RANSAC and graph-cut to infer the location and orientation of the primitive shapes, which resolves the continuous search problem in the presence of noisy inputs. After the location and orientation of the primitives are fixed, the remaining search problem is purely discrete, which we solve using Sketch, a state of the art program synthesis tool based on SAT solving [Solar-Lezama 2008]. To generate constraints suitable for program synthesis, we take advantage of canonical intersection terms [Shapiro and Vossler 1991]. We conduct extensive experiments with a CAD library composed of 50 models. The dataset, CAD files, and source code are openly available with this paper.

In this work, we contribute the following:

- A formulation that decouples the continuous aspects of the inverse CSG problem from the combinatorial ones, allowing us to leverage mature program synthesis techniques.
- Implementation of a complete pipeline that generates CSG trees from noisy inputs at varied approximation levels.
- Empirical evidence that our method can synthesize CSG programs for several complex models, demonstrating the efficiency, scalability, and robustness of our algorithm and its immediate application to mesh re-editing.

2 RELATED WORK

Our work draws ideas from previous work on modeling with parametric CAD, inverse procedural modeling, reverse engineering CAD, program synthesis, and primitive detection.

Modeling with Parametric CAD. Parametric CAD systems allow designers to define a geometry as the execution of a list of procedures that depend on a set of parameter values [Farin et al. 2002]. This parametric representation defines and constrains the ways the shape can be modified, allowing it to preserve the structure and other meaningful characteristics, such as manufacturing considerations. Koyama et al. [2015] use CAD to parametrize the space of manufacturable connectors. Similarly, Shugrina et al. [2015] and

Schulz et al. [2017] use parametric CAD models for fabrication-oriented design optimization. These recent works argue that the advantage of directly using CAD models over algorithms for automatic parametrization of an input 3D mesh [Bokeloh et al. 2012; Jacobson et al. 2011; Zheng et al. 2011] include: (1) greater and more meaningful geometric variations (as opposed to some dimensions) and (2) automatic exposure of relational constraints that can be used to understand function and support multiple fabrication methods. In these works, however, the parametric models are taken as an input and assumed to be carefully specified by expert engineers.

There has been some recent work on extracting editable 3D structures or revealing constraints from input that does not contain CAD structure. Chen et al. [2013] allow geometric re-editing from images. Similarly, Xu et al. [2016] propose an interactive method for extracting functional information of mechanisms from multi-view images. In both of these techniques, however, detailed user sketches are used to extract information on the shape components and their relations. In this work, we propose a method for automatically generating a CSG tree from a shape without any additional user assistance. This approach identifies the full structure and provides a plausible means by which the geometry could have been instantiated.

Inverse Procedural Modeling. The problem of recovering the list of geometry procedures that reconstruct a 3D shape is a special case of inverse procedural modeling. Over the years, two primary strategies have been developed for tackling this domain. Recognition-based strategies attempt to discover certain attributes of the underlying shape, such as classified geometry or segmentations [Fish et al. 2014; Kim et al. 2013; Litman et al. 2014; Tulsiani et al. 2017; Valentin et al. 2015; Xie et al. 2014; Zhu et al. 2017] in order to estimate the input's structure. These techniques are typically based off machine learning techniques or geometrically principled rules. While fast, they provide limited insight of a shape's underlying structure and can be difficult to debug. On the other hand, search-based strategies attempt to fully explain the input shape by sampling-based methods [Chaudhuri et al. 2011; Fan and Wonka 2016; Hämäläinen et al. 2014; Kalogerakis et al. 2012; Khungurn et al. 2015; Nishida et al. 2016; Ritchie et al. 2015; Schwarz and Wonka 2014; Talton et al. 2011; Wu et al. 2014] or direct search algorithms [Duncan et al. 2016; Fu et al. 2015; Lau et al. 2011; Peng et al. 2016; Shao et al. 2016]. Since search must be applied to each input geometry, these methods are typically slower; however, the fact that they return the geometry's underlying structure typically makes them more informative and robust. Our approach borrows techniques from both: we semantically segment our input geometry using recognition and then rely on program synthesis methods which take a direct-search approach for solving constraints.

Reverse Engineering CAD. Reverse engineering a CAD model is a classic research problem in the CAD community. The input to this problem is typically a shape represented as a surface mesh or a point cloud, and the output is a solid 3D model that can be used in CAD software for further operations. Central to this problem is the step that converts a boundary representation (B-rep) of a mesh into a CSG model. [Shapiro and Vossler 1991] first proposed a method to solve the general problem of converting a b-rep into CSG models using halfspaces, and lots of improvements have been proposed in later work [Buchele 1999; Buchele and Crawford 2004; Buchele and Roles 2001; Shapiro and Vossler 1993]. At a high level, this line of research attempts to solve the problem in a *deductive* manner: given a set of surface patches on the mesh, the method tries to find the correct combination of them by applying a series of rules, *e.g.*, if a halfplane fully encloses the input shape then the solution can be represented as an intersection of this halfspace and everything else left. As a result, the quality of the solution is greatly shaped by the set of rules one can find to apply. Our method is in sharp contrast to previous work in that we formulate a search problem that allows a modern program synthesizer to explore a much larger space and solve much larger-scale examples than in previous work.

Another family of approaches to reverse engineering a model is evolutionary algorithms. Some attempts have been made to use genetic algorithms to optimize a CSG tree such that certain constraints are satisfied [Hamza and Saitou 2004; Weiss 2009]. More recent work [Fayolle and Pasko 2016] considered the possibility of taking a B-rep as input to the evolutionary algorithm. The output of their method is a CSG tree that closely approximates the surface boundary. One bottleneck of evolutionary algorithms is their long runtime and large consumption of computational resources. Further, their non-deterministic nature makes it hard to control and understand their behavior throughout the optimization. By comparison, although our method also searches a large space, the search is much more directed; the constraint solver is able to take advantage of the structure of the constraints to prune large sections of the space and converge to a solution much faster.

There are two concurrent papers that attempt to solve the similar problem but from different perspectives: the CSGNet paper [Sharma et al. 2018] trains a neural network that takes as input a 2D or 3D shape and outputs a CSG program. Compared to their work, our method does not require a training dataset and we demonstrate our algorithm on 3D shapes of much higher complexity. Wu et al. [2018] reconstruct a CSG tree from raw point clouds by extracting the primitives and inferring CSG tree structures. When building the CSG tree, they divide the bounding box into voxels and label each voxel as inside or outside the point cloud. A CSG tree is then built in a bottom-up manner by solving an energy minimization problem based on the labels of each voxel. Our work shares a similar pipeline but does not require discretizing the inputs into voxels, which allows us to handle inputs with details at various levels.

Program Synthesis. The field of program synthesis also has a long history [Alur et al. 2013; Gulwani et al. 2017]. There have been several approaches to program synthesis including constraint-based search [Solar-Lezama et al. 2006], enumerative search [Udupa et al. 2013], and stochastic search [Schkufza et al. 2014]. In addition, there are program synthesis systems that are specifically targeted to some domains. For example, Gulwani et al. [2011] use a specialized algorithm to learn string transformation programs very efficiently. Deductive synthesis techniques also have a long history in the program synthesis space, including a number of recent success stories [Delaware et al. 2015; Püschel et al. 2004]. These techniques generally scale well because they break the synthesis problem into small local reasoning steps. It can, however, be difficult to engineer

the deductive rules and rule application heuristics to ensure the system efficiently finds a good solution.

For inverse CAD, a constraint-based approach is better suited because the enumerative search does not scale. This is due to the large, high-dimensional search space; stochastic approaches have a low probability of finding the correct program through sampling techniques. Hence, in this paper, we use a constraint-based program synthesis system called Sketch [Solar-Lezama 2008; Solar-Lezama et al. 2006]. Sketch is a general purpose synthesis system in the sense that it allows users to specify the grammar describing the space of possible programs in a high-level language and automatically generates the constraints for SAT solving.

Primitive Detection. Primitive detection is a well studied problem and many solutions have been proposed [Attene et al. 2006; Cohen-Steiner et al. 2004; Le and Duan 2017; Li et al. 2011; Schnabel et al. 2007; Wu Leif Kobbelt 2005; Yan et al. 2012]. In particular, Li et al. [2011] propose a primitive detection method that aims to satisfy CAD constraints like symmetry or perpendicularity. We do not apply their method to our problem directly because CAD models do not always have such assumptions. Instead, we implement our primitive detector based on the RANSAC approach [Schnabel et al. 2007] because of its efficiency and robustness.

3 OVERVIEW

In this section, we state the inverse CSG problem and present an overview of the method. This is a simplified version of the algorithm as described in Sections 4 and 5. In Section 6, we extend this algorithm with a segmentation procedure which allows our method to scale to more complex geometries. Our method's pipeline is illustrated in Figure 2.

We test our method on a benchmark of 50 CAD models. The dataset, CAD files, and source code are openly available with this paper. In terms of the CAD file, We use the format defined by OpenSCAD [OpenSCAD 2018] because it is open-source and freely available. Note that the zero-volume surfaces in some of the generated meshes are due to the numerical instabilities of OpenSCAD. These artifacts are not an indicator of the incorrectness of the resulting CSG tree. As a comparison, we model the same CSG trees in commercial software and get clean results (Figure 3).

3.1 Inverse CSG

Given an input surface mesh $\mathcal{M}$, we wish to find a program which generates an output geometry, such that every point in space is in its interior *if and only if* it is also in the interior of $\mathcal{M}$. Put more simply, this means that the interior of $\mathcal{M}$ and this geometry must occupy the same volumetric region.

In this paper, we focus on the space of CSG programs, which can interchangeably be described as trees. The CSG grammar, as we will describe formally in Section 4, provides a complete language of 3D geometry, making it an expressive choice. At a high level, the CSG grammar is comprised of discrete boolean operators and 3D geometric primitives which are described as a mix of discrete and continuous parameters. To our knowledge, no existing search methods can efficiently search over CSG program structure and its discrete and continuous parameters. Thus, the key to the solution will be reducing this mixed search problem into a purely discrete search.

3.2 Method Overview

Our method takes a manifold surface mesh as input and performs intelligent geometric preprocessing to transform the problem into a compact, discrete form that existing methods in program synthesis can solve efficiently. Below we briefly describe each step in the pipeline (Figure 2).

Primitive Detection. First, we resolve the continuous parameters in the search space by proposing a set of candidate primitives (Section 4.1). This discrete set covers all primitive choices and is generated from the input surface's geometric features and geometric reasoning. This step fixes the values of every continuous parameter and reduces the mixed search problem into a discrete one. This step is also robust to noise and can deal with approximations.

Sampling. Our original problem statement is intractable as a specification for the synthesis process. Luckily, as previous work [Shapiro and Vossler 1991] pointed out, given a finite set of primitives, one can choose a finite subset of point constraints which renders the rest of the constraints redundant. The method thus intelligently samples from the entire set of point constraints, keeping only those which add information to our search (Section 4.2). This step transforms the infinite constraint set into a finite one.

Synthesis. Since the inverse CSG problem reduces to a compact, semantic search over a discrete language, we apply program synthesis techniques in order to efficiently solve the problem (Section 5), feeding in the sampled points as constraints. At the end of this step, the method is guaranteed to find a feasible CSG program whose output matches the input mesh geometry as close as possible.

Post-processing. Finally, we further simplify the output program using deductive rules and then re-parameterize the program for easy end-user editing (Section 7).

4 INVERSE CSG FORMULATION

We state our problem definition as follows:

Definition 4.1. Given a mesh $\mathcal{M}$, the goal is to find a simple CSG tree such that its interior occupies the same volumetric space:

$$\begin{aligned} \min_{\text{CSG}} \quad & \mathit{Complexity}(\text{CSG}) \\ s.t. \quad & \forall p \in \mathbb{R}^3, \text{Inside}(\text{CSG}, p) \Leftrightarrow \text{Inside}(\mathcal{M}, p) \end{aligned} \tag{1}$$

Here *Complexity* is a discrete function that evaluates the complexity of the tree. In our implementation, we limit CSG as a binary tree and define *Complexity* as the number of nodes in the tree. Introducing *Complexity* biases our search towards simpler CSG trees. The constraints check all points and ensure that each point is inside CSG if and only if it is also inside $\mathcal{M}$.

Grammar. Before we describe how to solve the problem from Definition 4.1, we first need to specify the search space for CSG trees. This search space is defined by the grammar shown in Figure 4. In this grammar, each leaf node is a solid primitive, parametrized by variables such that its shape is completely defined in the 3D

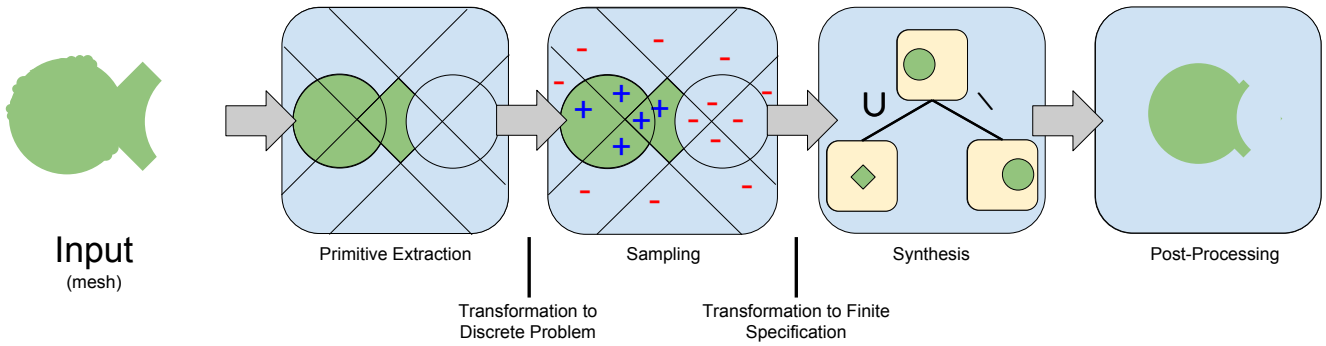

Fig. 2. An overview of our pipeline with a 2D example. As input, our method takes in a potentially noisy surface mesh. First, we extract an over-complete set of possible primitives in the scene (left blue box). In this example, two circles are detected, as well as four hyperplanes. In the second step of preprocessing (center-left blue box), our method attempts to add a labeled sample in each region of the scene, as sectioned by the surface primitives. Samples inside (outside) the input geometry are represented with a blue plus (a red minus). The samples and primitives are fed into our synthesizer (center-right blue box), which produces a correct CSG program as output. The resulting program is post-processed (right blue box); its program structure is simplified (not shown) and its geometric parameters are re-parameterized with metaparameters, allowing it to be interactively edited. Here, the positive and negative space circles are detected as symmetric and metaparameterized with a single radius variable. They are then edited to make the body larger and the tail thinner, while preserving the matching curvature.

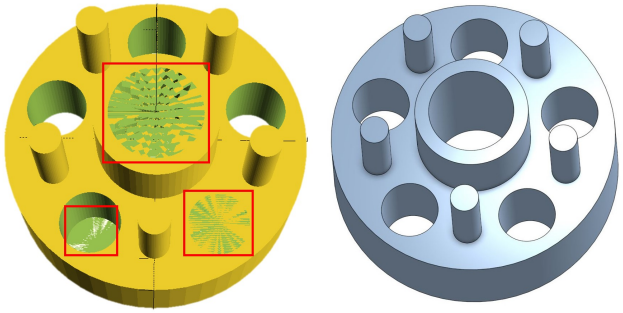

Fig. 3. Here we show one mesh result generated by our pipeline (left). Note that the degenerated planes in red boxes are not resolved by OpenSCAD. We model the same CSG tree in OnShape (right), and all these zero-volume planes are successfully removed.

space. For example, if *Type* is a sphere, then *Parameter* consists of its center and radius. For our experiments, we consider four types of solid primitives — spheres, cylinders, cuboids, and tori — but it is straightforward to add more primitive types. The internal nodes in the grammar are standard boolean operators: union, intersection, and subtraction.

The above problem definition reveals two challenges: first, the search space of CSG programs combines both discrete (boolean operators and primitive selections) and continuous (primitive parameterization) variables, making the optimization problem inherently challenging. Second, this problem specifies an infinite number of constraints to satisfy, since it considers all points in 3D space. Subsections 4.1 and 4.2 discuss our solutions to these challenges.

$$
\begin{aligned}
\text{CSG } C &:= L \mid I \\
\text{Leaf } L &:= \text{SolidPrimitive}_{Type}(Parameter) \\
\text{Internal } I &:= \text{Union}(C_1, C_2) \mid \text{Intersection}(C_1, C_2) \mid \\
&\quad\ \text{Subtraction}(C_1, C_2)
\end{aligned}
$$

Fig. 4. The CSG grammar. The leaf nodes are solid primitives with a geometric type and shape parameters. The internal nodes are boolean operators.

4.1 Detecting Primitives

We first remove the continuous variables by detecting primitives in the mesh. However, notice that it is difficult to infer full solid primitive parameters from the surface especially for primitives such as cuboids. Thus, at this stage, we can only detect *surface primitives* such as spherical surfaces, (infinitely large) planes, and (infinitely long) cylindrical surfaces. Each surface primitive represents a boundary $f(x) = 0, x \in \mathbb{R}^3$ and can be faithfully detected from the mesh itself. We will first discuss our method to detect all surface primitives, then describe how we can extract solid primitives from them.

4.1.1 Detecting Surface Primitives. The goal of surface primitive detection is twofold: locally, for each facet in the mesh we want to find a surface primitive that is as close as possible; globally, the total number of surface primitives should be kept low to avoid oversegmentation. Our surface primitive detector is built on top of the efficient RANSAC method [Schnabel et al. 2007] with a few extensions to improve its robustness and flexibility. First, we run the efficient RANSAC algorithm multiple times and collect surface primitives at various scales. Second, we select the set of surface primitives by running a graph-cut algorithm on a graph $G = (V, E)$ defined on the surface of the mesh, where each facet is a node v_i and each pair of adjacent facets defines an edge e_{ij}. Let $\{f_1, f_2, \cdots\}$

be the set of the surface primitives detected in the first step, we then find a graph-cut in order to assign fs to each facet v_i by minimizing the following energy:

$$E = \sum_{v_i} E_{unary} + \alpha \sum_{e_{ij}} E_{binary} + \beta \sum_{f_i} E_{label} \tag{2}$$

The unary energy for each facet is defined as

$$E_{unary}(v_i, f) = \int_{\mathbf{x} \in v_i} |Dist(\mathbf{x}, f)| d\mathbf{x} \tag{3}$$

where f is the surface primitive assigned to v_i. The binary energy on each edge is

$$E_{binary}(e_{ij}, f_i, f_j) = \begin{cases} 0, & \text{if } f_i = f_j \\ 1, & \text{otherwise} \end{cases} \tag{4}$$

where f_i and f_j are the surface primitives assigned to v_i and v_j. The label energy is defined as

$$E_{label}(f) = \begin{cases} 0, & \text{if } f \text{is not assigned to any node} \\ 1, & \text{otherwise} \end{cases} \tag{5}$$

The label energy can be thought of as a sparsifier on the set of output primitives, preferring assignments where fewer primitives are needed. This term is useful because our multiple invocations of RANSAC can detect a large number of potential surface primitives; the label energy term helps prune unnecessary primitives.

The user-defined scaling factors α and β allow for a tradeoff between favoring small distances to facets and fewer numbers of surface primitives. Combining efficient RANSAC and graph-cut algorithm results in a surface primitive detector that is both robust to noises and flexible to approximate the mesh at various level.

4.1.2 Adding Auxiliary Planes. The above algorithm can detect all visible surface primitives. In order to reconstruct solid primitives, we also need to infer hidden surfaces. For example, if one face of a cuboid is hidden inside another solid primitive, it will be impossible to exactly determine the cuboid's dimensions. We use the method in [Shapiro and Vossler 1991, 1993] to add auxiliary planes. Specifically, we use the point samples to be described shortly to detect whether a new auxiliary plane is needed, and choose to add planes that are at certain anchor points (*e.g.*, the center of a sphere) and parallel to the detected surface primitives.

4.1.3 Building Solid Primitives. We construct solid primitives once all surface primitives are collected. Building the solid primitives can itself be formulated as a discrete search problem: We first locally solidify surface primitives by replacing $f(x) = 0$ with $f(x) \geq 0, x \in \mathbb{R}^3$, and then we build a solid primitive by intersecting the individually solidified surface primitives. For example, a cuboid can be built by searching for three pairs of parallel planes orthogonal to each other and intersecting either themselves or their complements, depending on their signs. Although this solid primitive construction adds one more layer of search below the leaf node in the grammar, the search space is now purely discrete: all continuous parameters have been determined at this point and encoded in as a discrete search.

4.2 Reducing the Number of Constraints

Now that we have reduced the search space to a discrete one, the next step is to handle the infinite number of point constraints. Previous work [Shapiro and Vossler 1991, 1993] has laid the theoretical foundation for tackling this problem, which we briefly state below:

Definition 4.2 (Canonical intersection term). Let $\{f_1, \cdots, f_n\}$ be the set of all surface primitives. A canonical intersection term C is defined as $C = \cap_{i=1}^{n} F_i$ where F_i is either $\{\mathbf{x} | f_i(\mathbf{x}) > 0\}$ or $\{\mathbf{x} | f_i(\mathbf{x}) < 0\}$.

In other words, a canonical intersection term is an intersection of halfspaces induced by each surface primitive. Based on the above definition, [Shapiro and Vossler 1991] proposed the following describability theorem:

Definition 4.3. An input mesh is *describable* by surface primitives $\{f_1, \cdots, f_n\}$ if there exists a CSG tree whose leaves are f_is and it occupies the same volumetric region as the interior of the mesh.

Theorem 4.4. *An input mesh is describable by surface primitives $\{f_1, \cdots, f_n\}$ if and only if every canonical intersection term has the same classification with respect to the mesh.*

The theorem is based on the fact that each canonical intersection term is either fully included or excluded in the CSG tree, so the input mesh can be accurately reconstructed if and only if each canonical intersection term is fully inside or outside the mesh. As a result, instead of checking every point $p \in \mathbb{R}^3$, it is sufficient to check only one representative point from each canonical intersection term.

4.2.1 Sampling-based Method. Although [Shapiro and Vossler 1991] established the theoretical foundation, it assumes all the canonical intersection terms are given beforehand. In practice, however, as the number of canonical intersection terms grows combinatorially, enumerating all canonical intersection terms quickly becomes intractable. This motivates us to use a sampling-based method to find a representative from approximately all canonical intersection terms: we uniformly sample points inside the bounding box of the input mesh, then keep only one representative sample in each canonical intersection term. Then, we divide all resulting representatives into two sets, P^+ and P^-, based on whether they are inside our outside the input mesh, respectively.

4.2.2 Handling Imperfect Input. Sometimes the input mesh cannot be precisely reconstructed because there are missing surface primitives, or because the mesh is noisy or imperfect. This causes ambiguity when we assign labels (positive or negative) to representatives because both positive and negative samples can occur in a single canonical intersection term. In this case, we assign the label based on the majority of samples in that canonical intersection term. This is equivalent to finding a CSG tree such that the volumetric difference between it and the input mesh is minimized.

This step concludes the preprocessing step and leaves us with a more tractable problem with a finite number of constraints:

ALGORITHM 1: A naïve algorithm for finding a feasible solution

Input: Surface Primitives $f_i(x) = 0$; P^+; P^-.
Output: A CSG tree that satisfies all constraints in Problem 4.5.

```
CSG = ∅;
for each p ∈ P+ do
    C_p = ℝ^3;
    for each f_i do
        if f_i(p) > 0 then
            C_p = C_p ∩ {f_i(p) > 0};
        else
            C_p = C_p ∩ {f_i(p) < 0};
        end
    end
    CSG = CSG ∪ C_p;
end
return CSG;
```

Definition 4.5. Problem (1) can be equivalently redefined as the following discrete problem:

$$\begin{aligned} \min_{\text{CSG}} \quad & Complexity(\text{CSG}) \\ s.t. \quad & PrimitiveSet(\text{CSG}) = \{Detected\ Primitives\} \\ & \forall p \in P^+, Inside(\text{CSG}, p) = True \\ & \forall p \in P^-, Inside(\text{CSG}, p) = False \end{aligned} \tag{6}$$

4.3 A Naïve Algorithm

Algorithm 1 describes a naïve algorithm that produces a *feasible* CSG solution to the problem in Definition 4.5 that satisfies all the constraints but does not attempt to minimize its complexity.

Intuitively, Algorithm 1 outputs a long program that generates a flattened CSG tree by merging many small 3D pieces. To get a rough idea of how large the resulting CSG tree could be in the worst case, recall that an arrangement of n general planes partitions the space into $O(n^3)$ nonempty canonical intersection terms, which grows quickly to tens of thousands of regions when we increase n to a number even as small as 30. Even if only one percent of them are occupied by the input mesh, this still leaves hundreds of subtrees for Algorithm 1 to merge. As a result, this algorithm returns a concise CSG program in practice only if $|P^+|$ is very small.

5 ALGORITHM USING PROGRAM SYNTHESIS

Algorithm 1 provides a valid solution to the problem defined in Definition 4.5. However, the resulting CSG program is far from compact. It might be possible to simplify this CSG program to some extent using set theory rules, but piecing together all the small regions in the solution produced by Algorithm 1 is inherently a hard task. In this section, we describe an algorithm that uses program synthesis to directly search for a simple program that satisfies the constraints in Definition 4.5. Since even small-sized programs can induce a significantly large search space, for the algorithm described in this section, we assume that the desired CSG program is small in size. In the next section, we rectify this scalability issue by providing a divide-and-conquer algorithm which operates atop program synthesis.

Program Synthesis. In program synthesis, one defines a space of programs and attempts to search within the space for a program that meets an input specification. For this algorithm, we define the search space as all CSG trees that have at most $k = 32$ nodes within the grammar specified in Section 4. We define the specification that the positive (negative) representative points P^+ (P^-) lie in the interior (exterior) of the CSG program generated mesh. Specifically, among all CSG trees that have no more than k nodes, the synthesizer must find the smallest one that can satisfy the point constraints on P^+ and P^-. As one can see, if there exists a tree of size k that satisfies all the point constraints, the synthesizer, in theory, should be able to find it within the search space. The success of a synthesizer hinges on its ability to quickly search through the colossal search space to find a satisfying solution. To give an example of how big this search space is, given a choice of 10 solid primitives and depth 5, the total number of valid CSG trees is 4.3×10^{25}, making any explicit enumeration of the search space infeasible. In our work, we rely on Sketch [Solar-Lezama et al. 2006], a state-of-the-art program synthesis system that uses constraint-based reasoning using SAT solvers to efficiently search through this space. Sketch gives you the choice of searching for the globally optimal solution or performing only best-effort optimization. In our experiments, we run with best-effort optimization because it is faster and works well in practice.

Counter-Example Guided Inductive Synthesis. One of the key algorithmic components which allows Sketch to scale is the Counter-Example Guided Inductive Synthesis (CEGIS) algorithm. The insight is that while all constraints need to be satisfied for a program to be correct, not all constraints need to be considered by the synthesizer to produce the correct program: Imagine we're synthesizing a linear function $y = mx + b$, there can be thousands of point constraints on the line, but only 2 distinct points are needed. CEGIS employs two sub-routines, a synthesizer and a checker: The synthesizer solves the search problem on a subset of constraints, producing a candidate program. The checker takes the candidate program and, if possible, produces a counter-example — that is, a constraint that invalidates the candidate program. This counter-example is added to the subset of constraints, prompting the synthesizer to find a better candidate program. CEGIS terminates successfully when the checker fails to produce a counter-example and terminates unsuccessfully when the synthesizer fails to produce a candidate program. By iteratively adding counter-examples to the subset, CEGIS limits the number of constraints that need to be considered by the synthesizer, making synthesis scalable.

Our Algorithm. We now have enough information to describe Algorithm 2, which is built on top of Sketch and CEGIS. The algorithm is designed to solve small-scale problems under the assumption that the desired program is in the space of programs given to Sketch. The algorithm returns a simple CSG tree satisfying all point constraints.

Theorem 5.1. *Algorithm 2 finds a feasible solution in finite time if the desired program is in the search space given to the synthesizer.*

The proof can be found in Appendix A.

ALGORITHM 2: An algorithm for small-scale problems.

Input: Surface Primitives $f_i(x) = 0$; P^+; P^-.
Output: A CSG tree that satisfies all constraints in Definition 4.5.
Q^+ = Pick a subset of P^+;
Q^- = Pick a subset of P^-;
CSG = $\emptyset$;
do
  CSG = The result from calling a synthesizer to satisfy Q^+ and Q^-;
  Diff = $\emptyset$;
  for *each* $p \in P^+$ **do**
    if *not Inside(*CSG, *p)* **then**
      Diff = Diff $\cup$ $\{p\}$;
    end
  end
  for *each* $p \in P^-$ **do**
    if *Inside(*CSG, *p)* **then**
      Diff = Diff $\cup$ $\{p\}$;
    end
  end
  Pick a subset of points in Diff and add them to Q^+ and Q^- accordingly;
while Diff $\neq \emptyset$;
return CSG;

ALGORITHM 3: Final algorithm to solve large-scale problems.

Input: Surface Primitives $f_i(x) = 0$; P^+; P^-.
Output: A CSG tree that satisfies all constraints in Definition 4.5.
if $|P^+| = 1$ **then**
  return CSG from Algorithm 1;
end
Compute (CSG, $succeed$) using Algorithm 2;
if $succeed$ **then**
  return CSG;
else
  CSG = $\emptyset$;
  Partition P^+ into $P_1^+, P_2^+, \cdots, P_m^+$;
  for *each* P_i^+ **do**
    T_i = Recursively call Algorithm 3 with P_i^+ and P^-;
    CSG = CSG $\cup$ T_i;
  end
  return CSG;
end

6 ALGORITHM FOR LARGE-SCALE PROBLEMS

We now describe how to scale the algorithm in Section 5 to large problems. The key insight is to split the set of point constraints from Definition 4.5 so that the synthesizer will only have to discover the program for a small portion of the actual mesh. In an extreme case, if we give only one positive point constraint to satisfy, then we can always switch to Algorithm 1 to ensure success. Formally, the idea is stated by Theorem 6.1:

Theorem 6.1. *Let $P_1^+, P_2^+, \cdots, P_m^+$ be a partition of P^+ in Definition 4.5. Let T_i be the solution to the following problem:*

$$\begin{aligned} \min_{T_i} \quad & Complexity(T_i) \\ s.t. \quad & PrimitiveSet(T_i) = \{Detected\ Primitives\} \\ & \forall p \in P_i^+, Inside(T_i, p) = True \\ & \forall p \in P^-, Inside(T_i, p) = False \end{aligned} \tag{7}$$

Then $\cup T_i$ is a feasible solution to Definition 4.5.

The proof can be found in Appendix A.

While Theorem 6.1 does not specify a partition method, in practice semantically meaningful partitions like mesh segmentation or point cloud segmentation techniques produce simpler and more semantically meaningful programs than arbitrary partitions. This is intuitive as geometrically similar points usually have a better chance of being from the same CSG subtree. In our experiments, we tested surface mesh segmentation, spectral clustering, and hierarchical agglomerative clustering (HAC). We have found that HAC provides the best trade-off of algorithmic simplicity and desirable results.

Algorithm 3 shows our final algorithm based on Theorem 6.1. It recursively calls Algorithm 2 on the different partitions; if the synthesizer fails on any partition, then the positive constraints are split and the algorithm is run again. For the base case when $|P_i^+| = 1$, the algorithm reverts back to Algorithm 1, which is guaranteed to produce a solution for that partition. However, reaching the base case is very rare (it never happened in our experiments) since the search space that the synthesizer can handle is typically much bigger than the size of a single partition.

Theorem 6.2. *Algorithm 3 is guaranteed to produce a feasible solution to the problem in Definition 4.5 in finite time.*

The proof can be found in Appendix A.

7 POST-PROCESSING

The post-processing procedure has two stages. First, our method attempts to further simplify the CSG tree returned by Algorithm 3. Second, symmetric patterns are detected and reparameterized for end-user re-editing. We describe each of these processes in turn.

7.1 Simplification

Following Algorithm 3, we begin a procedure to simplify its output. Since the output is a union of the outputs produced by the program synthesis system for the different segments, there might be some unnecessary redundancies across the segments. We eliminate these redundancies by applying a set of equivalence rules to simplify the output CSG tree.

This simplification consists of three steps. In the first step, all extremely similar pairs of solid primitives are identified. Two solid primitives are considered similar if their types are the same and their positions, orientations, and parameterizations are all close in Euclidean distance. These similar solid primitives are replaced with a single common solid primitive.

In the second step, all intersections and unions are flattened with respective nested operations, and redundant expressions are eliminated. For example, $(A \cup B) \cup A$ would be flattened to $(A \cup B \cup A)$, and the redundant $\cup A$ would be eliminated, yielding $A \cup B$. The same would be true if all of the unions were changed to intersections. However, expressions with mixtures of operators, such as $(A \cup B) \cap A$ would not be simplified in any way in this step.

In the third and final step, our algorithm recursively attempts to simplify nested expressions. During each simplification, all combinations of all sub-expressions are compared, and a set of prescribed simplification rules are greedily applied. If a pair of subexpressions is simplified at a certain depth in the expression tree, the entire tree is re-simplified from that node downward, in order to take advantage of potentially newly exposed simplifications.

7.2 Symmetry-Based Re-editing

While the resulting CSG tree is useful in that it can be directly edited, each component of the tree is independent. Parameterizations are isolated to each individual solid primitive. In order to facilitate easy user exploration and editing of the resulting CSG tree, we re-parameterize some of the solid primitives of the resulting CSG tree in order to take advantage of potential symmetries. In particular, we loop over each pair of solid primitives of the same type in our CSG tree and compare their volumetric parameterizations. For each pair, we compare sphere radii, cylinder radii, cylinder heights, and cuboid bounding box extent triplets (disregarding order). If any of these four quantities match, we consider the solid primitive pair to be symmetric in that parameter (or parameter set, in the case of the cuboid), and reparameterize them with a meta-parameter. This meta-parameterization allows all detected symmetric solid primitives to be edited simultaneously by changing a single value, preserving the detected symmetry.

8 RESULTS

In this section, we first present a new dataset for benchmarking the performances of different reverse engineering methods. We then compare our algorithm to two baselines and report their performances on our dataset. For almost every model in the dataset, our method manages to find a much more compact CSG tree compared to the baseline approaches and reconstruct all examples with $< 7\%$ relative error (Figure 8). Next, we demonstrate the robustness of our method by testing it on imperfect meshes. Finally, we discuss the effectiveness of our simplification step and present mesh re-editing examples.

8.1 Benchmark

Although reverse engineering a surface mesh to a CSG tree is a long-studied problem, not a lot of effort has been made to build a test set for evaluating different algorithms. The lack of a high-quality dataset makes it more difficult to fairly compare between different methods. In this work, we attempt to close this gap by presenting a dataset that consists of 50 clean surface meshes of various complexity. These meshes are collected from examples in the previous work [Buchele and Crawford 2004; Fayolle and Pasko 2016] and online CAD libraries [GrabCAD 2018; Thingiverse 2018; Zhou and Jacobson 2016], including objects such as brackets, gears, and knot structures. The simplest of these surface meshes can be constructed by fewer than 10 surface primitives while the most complex model requires over 100 surface primitives. We ask readers to refer to Figure 5, 6, and 7 for more details.

We use this benchmark set to evaluate our algorithm and two baselines [Buchele and Crawford 2004; Fayolle and Pasko 2016].

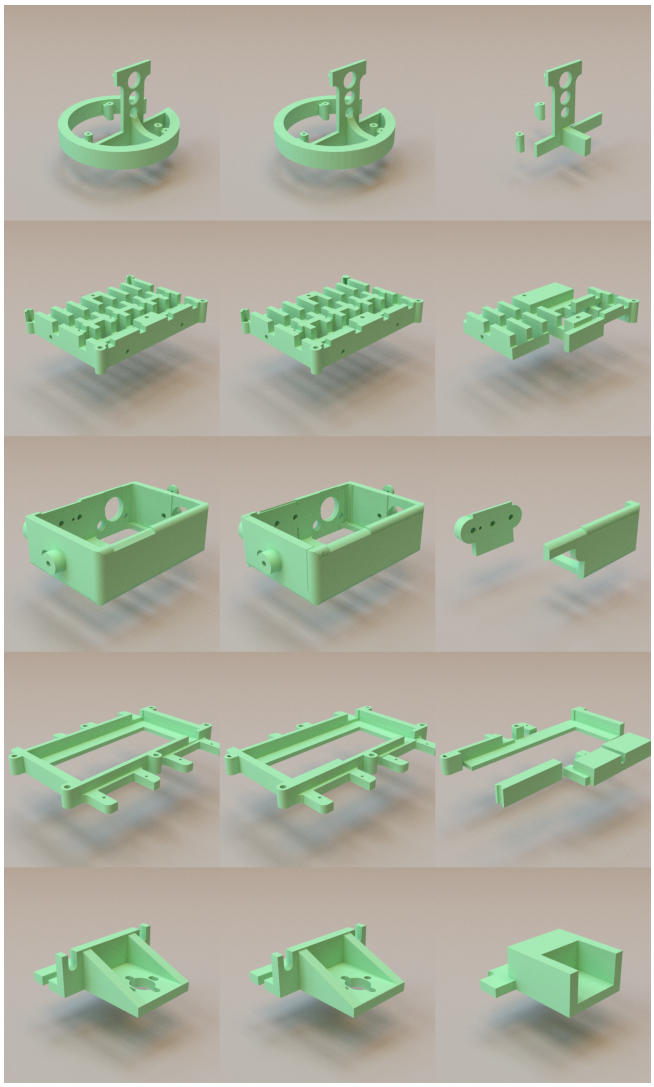

Fig. 5. Here we show five examples from the benchmark and our solutions. Left: input meshes. Middle: output meshes reconstructed from our CSG solutions. Right: intermediate CSG results for visualization purposes. We visualize the CSG trees by doing a post-order tree traversal. For these five examples, the complete visualization can be found in our video.

Each method takes as input a mesh from the dataset and outputs the best CSG tree it can find. We evaluate methods on three metrics: the complexity of the tree (the number of nodes), the volumetric difference between the mesh and the tree, and the runtime.

8.2 Comparison to Deductive Methods

Since Theorem 4.4 was proposed in [Shapiro and Vossler 1991], many follow-up methods have been proposed [Buchele 1999; Buchele and Crawford 2004; Buchele and Roles 2001; Shapiro 2001; Shapiro and Vossler 1993] to try to solve both 2D and 3D meshes. At a high level, this line of research attempts to solve the problem in a deductive manner: knowing the sign of each canonical intersection

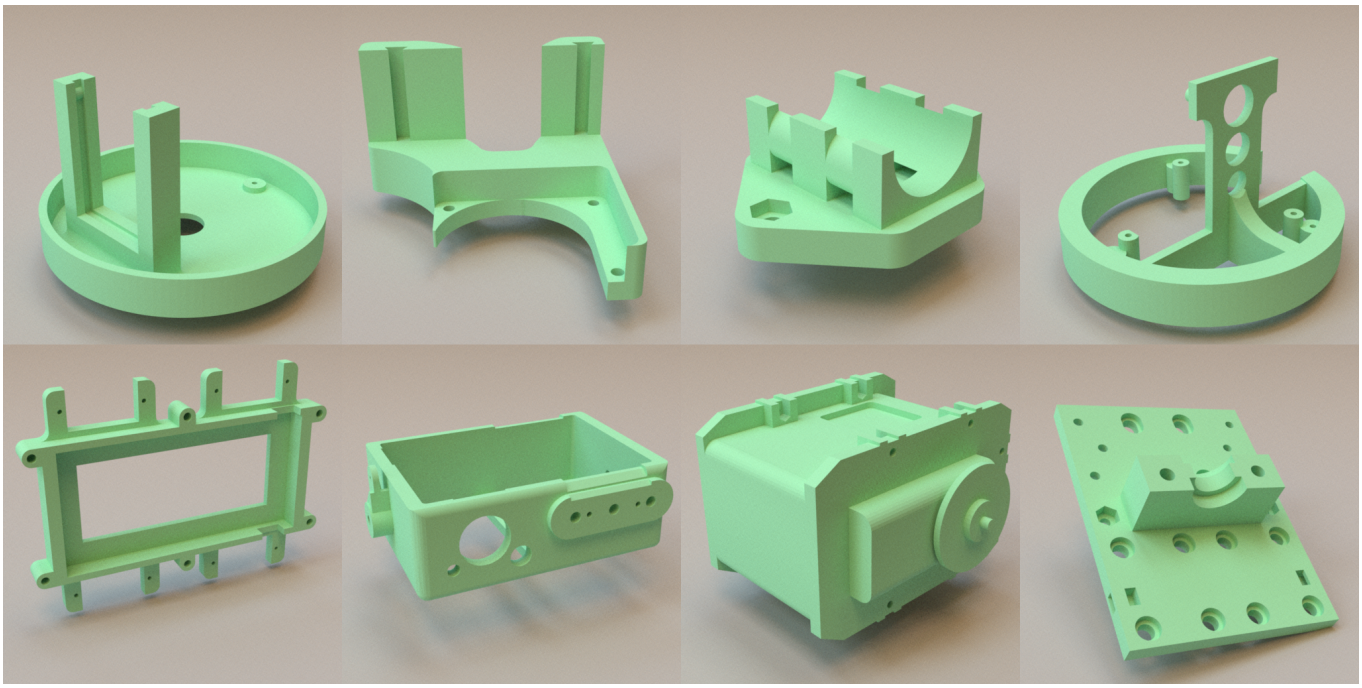

Fig. 6. Eight representative examples out of the 50 models in our benchmark.

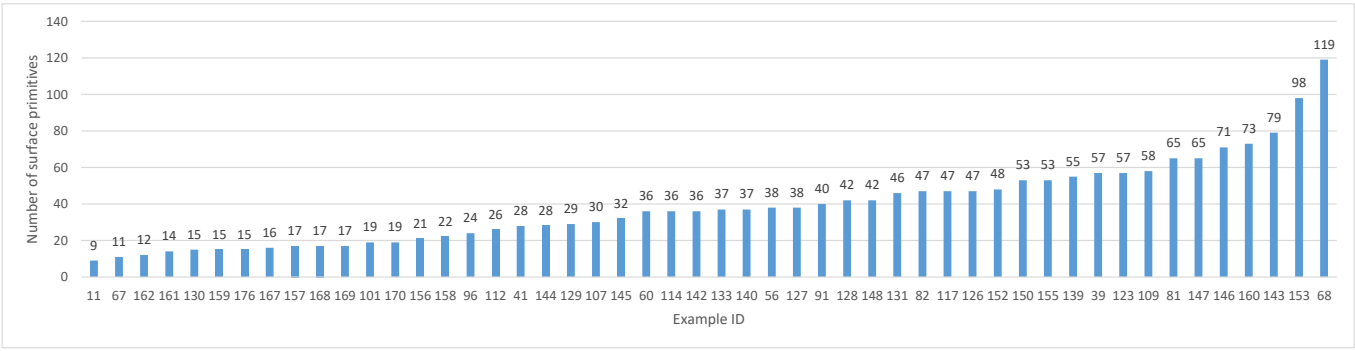

Fig. 7. Our benchmark provides meshes of various complexity (number of surface primitives). This chart shows the number of surface primitives in each mesh in an increasing complexity order.

term, a union of all the positive canonical intersection terms is considered as an initial (valid) solution. Then, a set of equivalence rules are iteratively applied to simplify this solution. In this section, we implemented the BHC algorithm in [Buchele and Crawford 2004] and evaluate its performance on our benchmark. The output is a binary CSG tree saved in an OpenSCAD file and no post-processing was applied. The results are compared to ours and discussed below.

Tree Complexity. We evaluate the tree complexity in terms of the number of nodes (Figure 9). For almost every example, our method generates significantly more concise solutions, and the trend becomes more obvious as the problem size scales up. This is because our synthesizer gives the performance boost in generating a smaller number of leaf and internal nodes.

Volumetric Difference. Since both methods are built on top of Theorem 4.4, the results are guaranteed to have small volumetric difference (Figure 8). The standard Hausdorff distance is also evaluated and provided along with the volumetric error. The relative errors shown in these examples are mostly due to the difference between perfect curved surfaces and the tessellation in the input meshes (Figure 11).

Runtime. Figure 10 shows the runtime of our algorithm for each example in the benchmark. Runtime less than 2 minutes is not displayed in Figure 10 due to the axis range. The most time-consuming

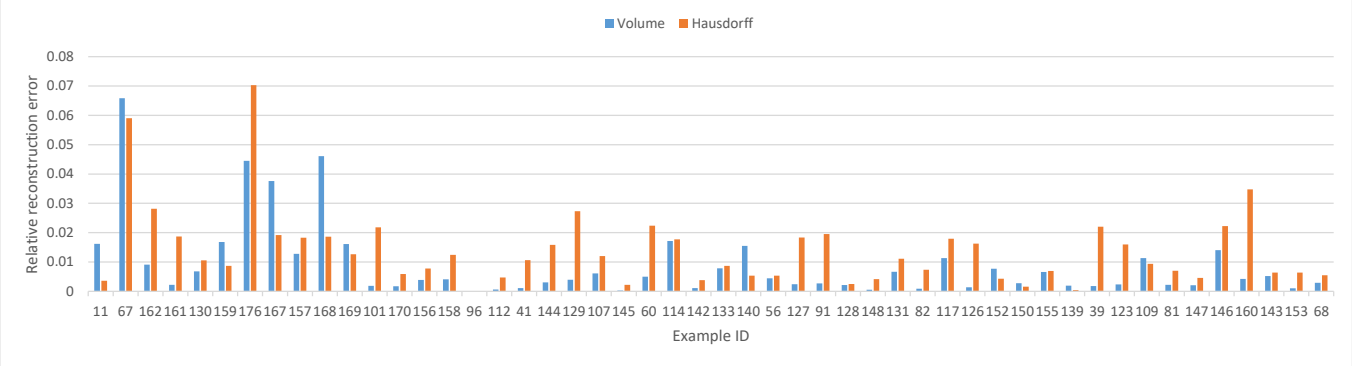

Fig. 8. The relative reconstruction error using our method. Blue: the relative volumetric error, computed by dividing the volumetric difference by the volume of the input mesh. Orange: the relative Hausdorff distance, computed by the Hausdorff distance divided by the size of the bounding box of the input mesh.

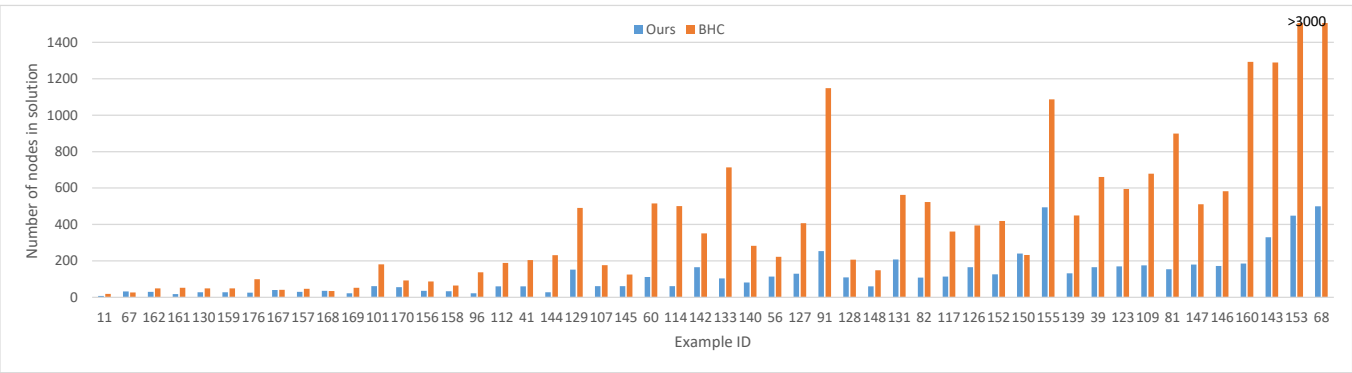

Fig. 9. Comparison between our method and the BHC algorithm in terms of node numbers.

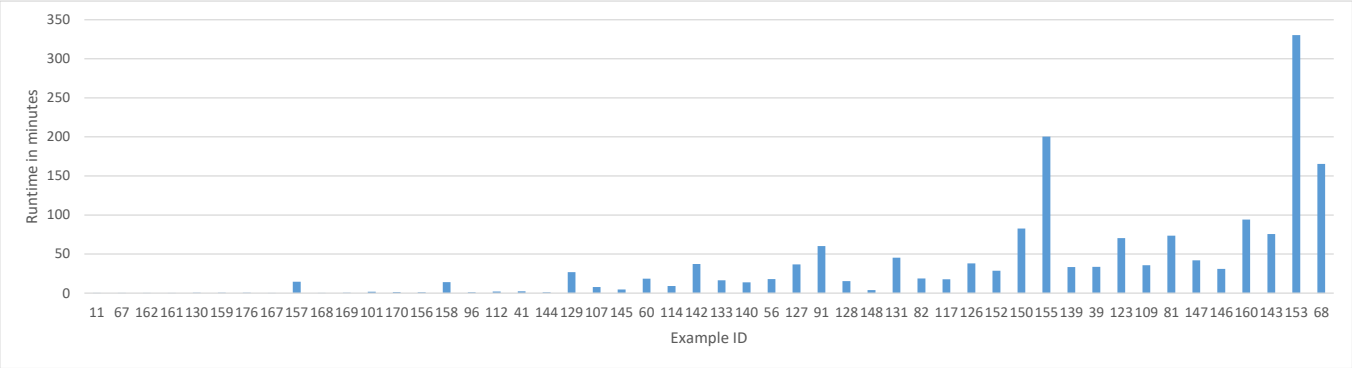

Fig. 10. Time spent on each example in the data set by our method. Missing columns mean the examples were solved in less than 2 minutes.

example was finished in 330.3 minutes, and the average runtime across the whole benchmark is 38.3 minutes. The time reported in Figure 10 was measured by running our algorithm in the sequential mode in order to make a fair comparison to other baselines. In practice, one can parallelize the algorithm in Section 6 because after segmentation, solving each part is completely independent of each other.

Compared to our method, the BHC algorithm has an expected $O(n^3)$ running time where n is the number of surface primitives. For our benchmark where most examples have $n < 100$, we observed the BHC algorithm finished typically in less than a few minutes. Although BHC is a faster algorithm, our method generates much more compact solutions, making it arguably the better choice when the runtime is not a bottleneck.

8.3 Comparison to Genetic Algorithm

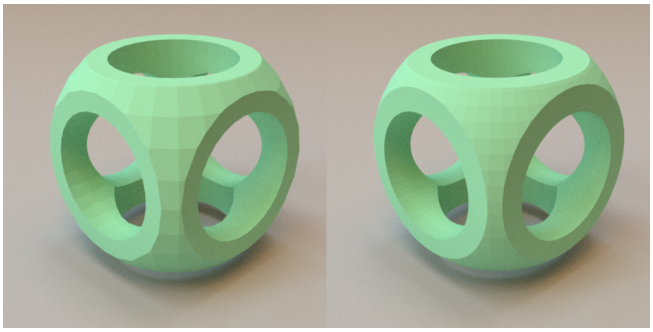

Fig. 11. Here we show example 67, the one with the largest volumetric error in Figure 8. Left: the input mesh which triangulates the spherical surface using 24×12 squares. Right: our synthesizer recognizes the underlying sphere and exports it as a high-resolution mesh, causing the large volumetric difference shown in Figure 8 (6.5%, column 2).

We also compared our algorithm to a genetic algorithmic approach presented in [Fayolle and Pasko 2016]. We ran this algorithm on our benchmark using the parameters as they reported them — 50,000 sample points and a population size of 150.

We compared our algorithm to the genetic algorithm on four different meshes, running on a Xeon E5-1620 3.5 GHz processor. Solving for the CSG of these meshes scaled exceptionally poorly in the number of candidate primitives. Examples 101, 128, and 145 had 19, 42, and 32 surface primitives respectively, and each required at least 2 hours per iteration. Given that they each required over 100 iterations, we terminated the optimization after 200 hours without completion. By comparison, our algorithm was able to solve these problems in 1.6, 15.2, and 4.6 minutes.

We were able to run one example to completion using the genetic algorithm. The genetic algorithm ran on example 96, with 12 solid primitives in approximately 2 hours, and ultimately terminated with 14% error. Figure 12 shows the energy of the elite CSG tree versus the generation. The genetic algorithm's most computationally expensive step — the energy function evaluation — is embarrassingly parallelizable, and speedups scale linearly with the number of CPU cores available up to the population size. Ideally, when we use 150 cores, this approach can be sped up to a runtime 8 minutes and 56 seconds. However, we note that most CPUs have no more than 8 cores (parallelizing on 8 cores would require more than 4 hours for this example), leaving such parallelization infeasible outside of cloud applications. By comparison, our algorithm was able to solve example 96 in only 50 seconds on a single thread — faster than any parallelization of the genetic algorithm, while using significantly fewer resources, and producing only 0.0023% error. For reference, we show the evolution of the genetic algorithm's elite candidate CSG tree energy function and volumetric error in Figure 13.

To further demonstrate the scalability of our algorithm versus the genetic algorithm, we ran the genetic algorithm on a CSG tree comprised of a single leaf cuboid and provided 6 candidate primitives. Single-threaded, the genetic algorithm was able to find the optimal solution in 1 hour 4 mins (and 103 generations). Given 150 cores, this

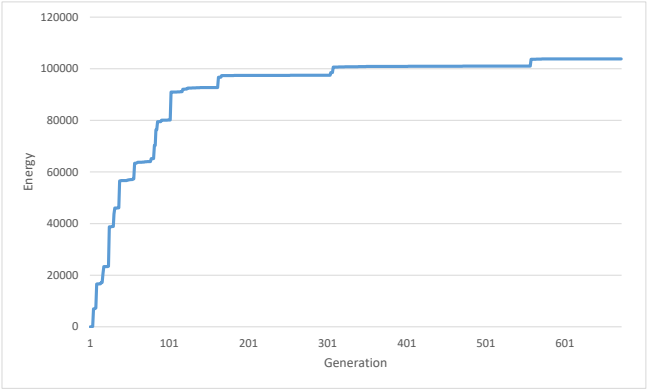

Fig. 12. The energy as defined by [Fayolle and Pasko 2016] of the elite CSG tree in the genetic algorithm's population, vs. generation. While the energy improves rapidly at the beginning, it is unable to resolve all discrepancies between the input and generated mesh after 672 iterations.

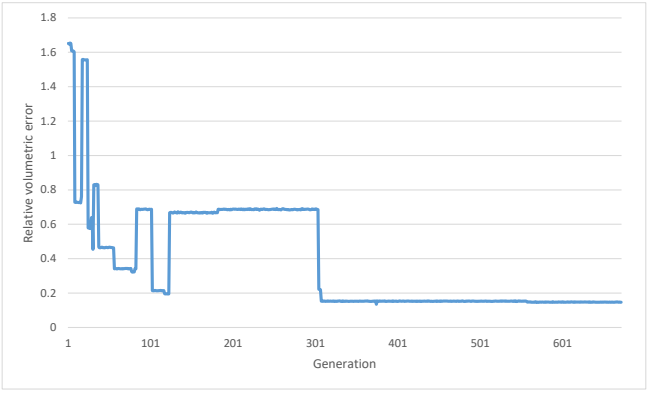

Fig. 13. The relative volumetric error of the elite CSG tree in the genetic algorithm's population, vs. generation. The energy in [Fayolle and Pasko 2016] is purely defined on the surface difference so it fails to capture the volumetric change. Thus, while energy monotonically increases, the volumetric error oscillates between generations.

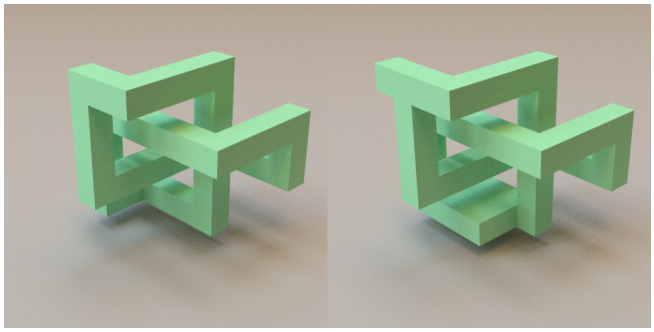

Fig. 14. The input mesh (left) and the mesh produced by the best CSG tree after termination of the genetic algorithm (right). Note that the genetic algorithm's output still has a few points of notable volumetric difference, particularly containing extra segments at the top-left-back and bottom.

could be solved in around 40 seconds. By comparison, our algorithm solves the problem single-threaded in less than 5 seconds. We note that our algorithm, when running, considers 216 solid primitives and is still around 800 times faster. If the genetic algorithm was fed 15 candidate primitives, iterations took longer than an hour apiece and was not run to completion. To give a stronger sense as to the comparative robustness between algorithms with respect to unnecessary candidate primitives, our algorithm solved example 96 having been given 216133 candidate solid primitives.

8.4 Robustness

Most examples in our benchmark were surface meshes exported from CAD software. As a result, we know in advance there exists at least one CSG program that can perfectly represent the input mesh. In this section, we demonstrate the robustness of our algorithm by pushing it to solve noisy meshes that cannot be precisely described by our CSG grammar. Moreover, to show in our pipeline the synthesizer is resilient to changes in primitive detection results, we send the synthesizer various number of surface primitives and evaluate its output. In both cases, our pipeline is able to find an approximation of the input mesh at various levels.

8.4.1 Handling General Mesh Inputs. Here we provide three examples to show our algorithm is robust to handle input meshes beyond the capability of our CSG grammar. The Fandisk example (Figure 15) contains general quadratic surfaces not yet supported by our grammar, and our pipeline found a good approximation of Fandisk using spheres, planes, and cylinders only.

In the next two examples, we alternate the inputs by adding noises (Figure 16) or remeshing (Figure 17). The results show that our pipeline can tolerate a certain amount of changes in the inputs while still generating solutions that capture the underlying structure. This is mostly because of the robustness of our primitive detector.

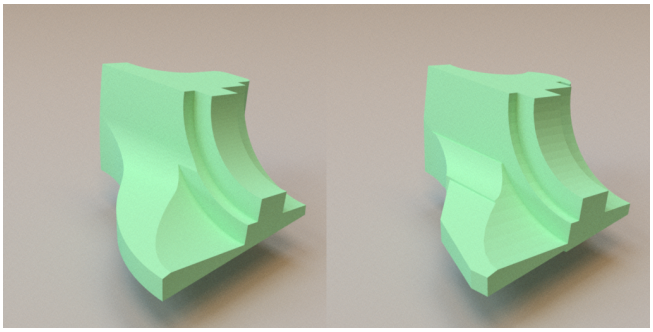

Fig. 15. Our pipeline can approximate meshes made from primitives not included in the grammar. Left: the Fandisk example that contains quadratic surfaces; Right: the output of our method that approximates the input mesh.

8.4.2 Varying Surface Primitive Number. Another question we ask is how robust our synthesizer is if the primitive detector fails to deliver the right set of surface primitives. As explained in Section 4, the synthesizer will attempt to use the given primitives to approximate the mesh as closely as possible. Figure 18 demonstrates this in example 39, which has 57 surface primitives in total. Here

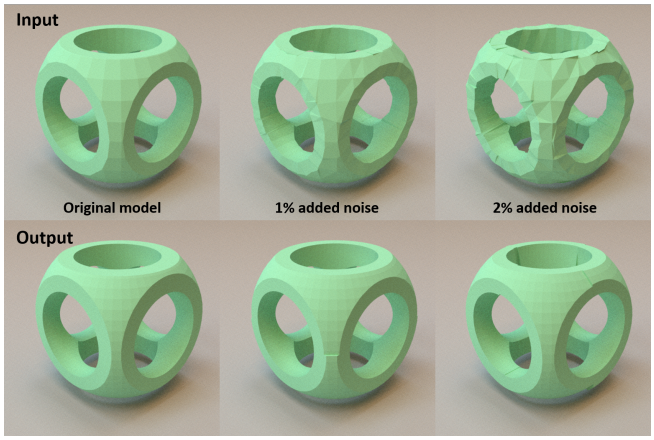

Fig. 16. Solving input meshes with different amount of noises. Top row from left to right: 0%, 1%, and 2% noises are added. Bottom row: our solutions.

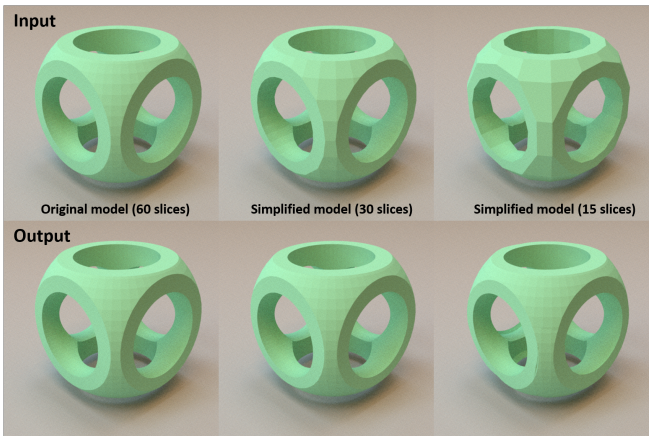

Fig. 17. Solving remeshed models. Top row: input meshes that use a different number of slices to approximate curved surfaces. Bottom row: our solutions.

we vary the threshold in our graph-cut algorithm to generate 10, 20, 40, and eventually 57 surface primitives and send them to the synthesizer. As the number of primitives increases, the volumetric difference becomes smaller and the synthesizer gradually converges to the input mesh. As shown in the figure, even with limited primitives, the synthesizer still does reasonably well in finding a good approximation of the mesh.

8.5 Post-processing

8.5.1 Program Simplification. We applied our simplification procedure to all of our output CSG solutions (Figure 19). In general, we found that our method was able to reduce the number of tree nodes by 16% on average. This reduction was in part due to flattening binary expressions of the same type and in part due to recognizing duplicated subtrees between segments. Since our simplification algorithm does not reason about geometry, it cannot, *e.g.*, merge adjacent cuboids, or remove shapes which have no impact on the

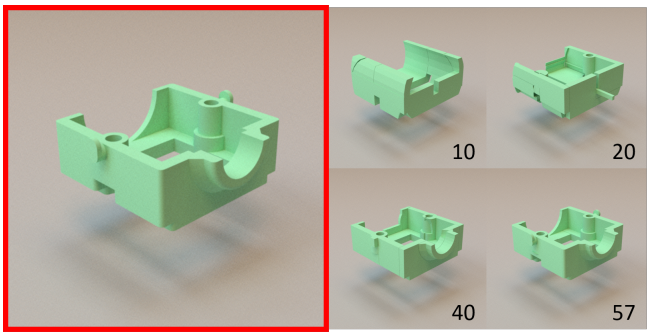

Fig. 18. Using a various number of primitives to approximate example 39. Left in the red box: the original mesh. Right: results from our pipeline using 10, 20, 40, and 57 surface primitives. As the number of primitives increases, the result converges to the input mesh.

output geometry. We leave such extensions to our method for future work.

8.5.2 Mesh Re-editing. We present an example of changing CSG parameters using our symmetry-based re-parameterization in Figure 20. We found that our method is good at detecting and meta-parameterizing most symmetries. However, symmetries may be missed in two cases. First, if two geometries are very similar but differ beyond the user-chosen threshold distance, they will not be detected as symmetric. This is an inevitable drawback of thresholding. Second, although two geometries in a design might be symmetric, the detected CSG might not be symmetric due to occlusion. For example, one primitive in a symmetric pair may be longer than the other if it protrudes into, and is completely covered by, other existing geometry in the part. This phenomenon can create both false positives and false negatives. In example 160, some symmetries of the base frame were missed for this latter reason. Frame pieces on one side are longer than the other, although visually they look the same since the extra length contributes to a corner which is already covered by an adjacent cuboid primitive. In either of these cases the geometry, though symmetric, will not be meta-parameterized as such.

9 DISCUSSION

Reverse engineering a CAD model is a classic and fundamental problem. Despite its long history, solving it in a general 3D setting is still considered to be very difficult due to its huge search space. By combining geometric processing methods and program synthesis techniques, we have presented a pipeline that pushes the problem of reverse engineering a CAD model to a larger scale that has not been seen in any previous work. This scalability improvement is mostly because of our effort to reformulating the original problem as a discrete search task that modern program synthesizers excel in, and understanding how far we can push a program synthesizer to solve this classic problem remains an exciting direction to explore.

As pointed out before, our pipeline is also robust to imperfect inputs, including meshes not describable by our grammar or missing surface primitives. This is in part due to the primitive detector being resilient to noise, and in part due to preprocessing samples in each canonical intersection term to avoid potential conflicts in the synthesizer beforehand. Alternatively, we can choose to increase the expressiveness of our grammar by including more primitive types, *e.g.*, quadratic or even B-spline surfaces, which opens up the possibility of accurately reconstructing a free-form geometry.

One limitation of our work is that our results are greatly shaped by the segmentation method used in our algorithm. Good segmentation generally leads to compact solutions and short programs, whereas bad segmentation may result in broken pieces and overcomplicated solutions. In our algorithm design, we intentionally avoid relying on a specific segmentation method in exchange for generality so that different segmentation methods can be modularly swapped. It is natural to extend our pipeline to combine mesh and point cloud-based segmentations, which we leave for future work.

Finally, the CSG grammar discussed in this paper is limited to basic CSG operations. In the future, it would be interesting to explore the possibility of integrating higher-level programming language concepts like for loops, if-else statements, or even recursive calls into the pipeline. Interestingly, it turns out we can build strong connections between these concepts and classic geometry problems. For example, synthesizing a for loop in a CSG program can be linked to detecting symmetric or repeated patterns on the mesh. As a result, we believe combining these two active research fields will open up new possibilities and inspire more exciting research work to come in the future.

10 CONCLUSION

We have presented a novel method which infers CSG programs that reconstruct an input triangle mesh. We built a dataset of 50 CAD models of varying complexity, where the most complex one has over 100 surface primitives. The dataset has more examples, and more complex examples, than previous work. By intelligently converting a mixed, over-constrained search problem into a discrete, compact form, we presented a parallelizable search algorithm that solved examples in the dataset. Further, we demonstrated the robustness of our algorithm by solving examples not describable by our grammar. Finally, since our method returns parameterized CSG programs, it provides a powerful means for end-users to edit and understand the structure of 3D meshes.

By decoupling primitive detection and search, we have created a general and flexible framework. By formulating our problem in the context of programming languages, we have been able to employ state-of-the-art program synthesis techniques, which can quickly produce high-quality, compact results. In the future, we hope to extend our method to more complex classes of geometry, such as spline surfaces. We also hope to introduce higher level programming concepts to the pipeline and further explore the connections between program synthesis, geometry processing, and computer-aided design. We look forward to seeing how the community can apply our proposed techniques to more complex, real-world problems.

A THEOREMS AND PROOFS

Proof of Theorem 5.1. To show the algorithm terminates in finite time, note that $|Q^+| + |Q^-|$ increases by at least one at each iteration.

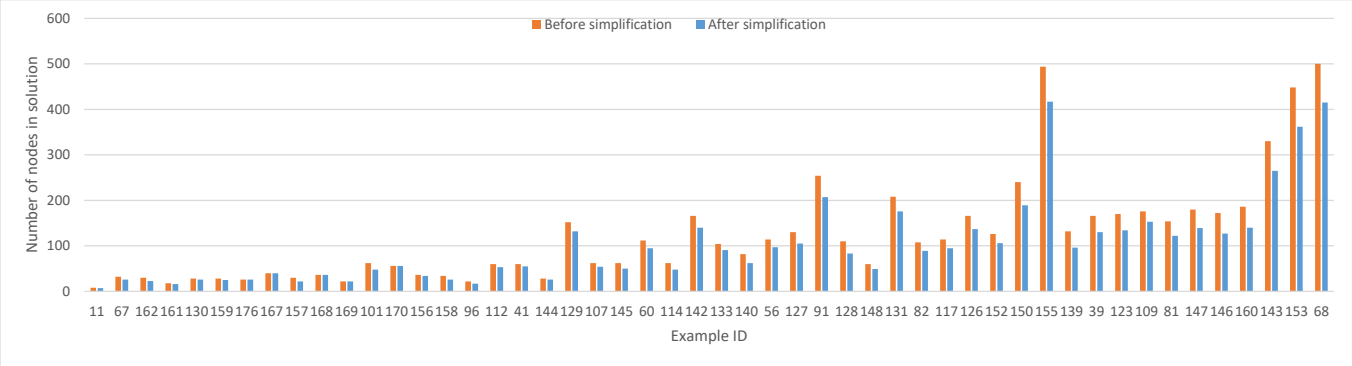

Fig. 19. The number of tree nodes in our solutions before and after applying the simplification step in Section 8.5.

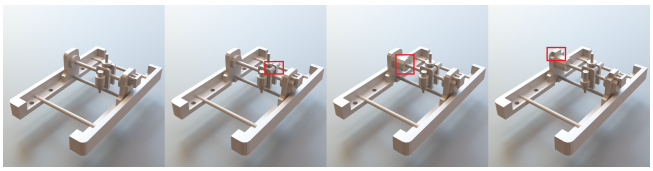

Fig. 20. Symmetry-based re-editing results in example 160. The leftmost figure shows the output shape from our algorithm, then three sequential editing operations are applied, highlighted in red boxes.

Since $|Q^+| \leq |P^+|$ and $|Q^-| \leq |P^-|$, the loop runs at most $|P^+|+|P^-|$ times. Upon termination, the Diff set is empty, meaning that CSG satisfies all points in P^+ and P^- and therefore it is a feasible solution to Problem 4.5.

Proof of Theorem 6.1. By definition, CSG uses only detected primitives. To see points in P^+ are inside CSG, pick any $p \in P^+$. Since $\{P_i^+\}$ is a partition of P^+, p must come from some P_i^+. Thus $p \in T_i \subset$ CSG. To see that all points in P^- are excluded, assume there is one counter-example $p \in P^-$ but $p \in$ CSG. Therefore, there exists i such that $p \in T_i$, which contradicts to the fact that $\forall p \in P^-, Inside(T_i, p) = False$.

Proof of Theorem 6.2. To show the recursion terminates in finite time, just note that $|P^+|$ strictly decreases in each recursive call. To show the guarantee, notice that Algorithm 1 never fails and whenever Algorithm 2 fails it is reduced to Algorithm 1 eventually. The fact that the solution is feasible comes directly from the correctness of Algorithm 1, Algorithm 2, and Theorem 6.1.

ACKNOWLEDGMENTS

We thank Kevin Ellis, Yu Wang, Justin Solomon, and Bo Zhu for insightful discussions. This work was supported by the MUSE: Mining and Understanding Software Enclaves program (Defense Advanced Research Projects Agency Grant FA8750-14-2-0242) and National Science Foundation Grants (No. 1138967, 1533753, 1644558, and 1830901).

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 1–8.

Marco Attene, Bianca Falcidieno, and Michela Spagnuolo. 2006. Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer* 22, 3 (2006), 181–193.

Martin Bokeloh, Michael Wand, Hans-Peter Seidel, and Vladlen Koltun. 2012. An Algebraic Model for Parameterized Shape Editing. *ACM Trans. Graph.* 31, 4 (July 2012), 78:1–78:10.

Suzanne Fox Buchele. 1999. *Three-dimensional binary space partitioning tree and constructive solid geometry tree construction from algebraic boundary representations.* The University of Texas at Austin.

Suzanne F Buchele and Richard H Crawford. 2004. Three-dimensional halfspace constructive solid geometry tree construction from implicit boundary representations. *Computer-Aided Design* 36, 11 (2004), 1063–1073.

Suzanne F Buchele and Angela C Roles. 2001. Binary space partitioning tree and constructive solid geometry representations for objects bounded by curved surfaces.. In *CCCG*. Citeseer, 49–52.

Siddhartha Chaudhuri, Evangelos Kalogerakis, Leonidas Guibas, and Vladlen Koltun. 2011. Probabilistic Reasoning for Assembly-based 3D Modeling. *ACM Trans. Graph.* 30, 4, Article 35 (July 2011), 10 pages. https://doi.org/10.1145/2010324.1964930

Tao Chen, Zhe Zhu, Ariel Shamir, Shi-Min Hu, and Daniel Cohen-Or. 2013. 3-Sweep: Extracting Editable Objects from a Single Photo. *ACM Trans. Graph.* 32, 6, Article 195 (Nov. 2013), 10 pages. https://doi.org/10.1145/2508363.2508378

David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. 2004. Variational shape approximation. In *ACM Transactions on Graphics (TOG)*, Vol. 23. ACM, 905–914.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 689–700. https://doi.org/10.1145/2676726.2677006

Noah Duncan, Lap-Fai Yu, and Sai-Kit Yeung. 2016. Interchangeable Components for Hands-on Assembly Based Modelling. *ACM Trans. Graph.* 35, 6, Article 234 (Nov. 2016), 14 pages. https://doi.org/10.1145/2980179.2982402

Lubin Fan and Peter Wonka. 2016. A Probabilistic Model for Exteriors of Residential Buildings. *ACM Trans. Graph.* 35, 5, Article 155 (July 2016), 13 pages. https://doi.org/10.1145/2910578

Gerald E Farin, Josef Hoschek, and Myung-Soo Kim. 2002. *Handbook of computer aided geometric design*. Elsevier.

Pierre-Alain Fayolle and Alexander Pasko. 2016. An evolutionary approach to the extraction of object construction trees from 3D point clouds. *Computer-Aided Design* 74 (2016), 1–17.

Noa Fish, Melinos Averkiou, Oliver van Kaick, Olga Sorkine-Hornung, Daniel Cohen-Or, and Niloy J. Mitra. 2014. Meta-representation of Shape Families. *ACM Trans. Graph.* 33, 4, Article 34 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601185

Chi-Wing Fu, Peng Song, Xiaoqi Yan, Lee Wei Yang, Pradeep Kumar Jayaraman, and Daniel Cohen-Or. 2015. Computational Interlocking Furniture Assembly. *ACM Trans. Graph.* 34, 4, Article 91 (July 2015), 11 pages. https://doi.org/10.1145/2766892

GrabCAD. 2018. GrabCAD: Design Community, CAD Library, 3D Printing Software. (2018). https://grabcad.com/

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330.

Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

Perttu Hämäläinen, Sebastian Eriksson, Esa Tanskanen, Ville Kyrki, and Jaakko Lehtinen. 2014. Online Motion Synthesis Using Sequential Monte Carlo. *ACM Trans. Graph.* 33, 4, Article 51 (July 2014), 12 pages. https://doi.org/10.1145/2601097.2601218

Karim Hamza and Kazuhiro Saitou. 2004. Optimization of constructive solid geometry via a tree-based multi-objective genetic algorithm. In *Genetic and Evolutionary Computation Conference*. Springer, 981–992.

Alec Jacobson, Ilya Baran, Jovan Popovic, and Olga Sorkine. 2011. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.* 30, 4 (2011), 78.

Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. 2012. A Probabilistic Model for Component-based Shape Synthesis. *ACM Trans. Graph.* 31, 4, Article 55 (July 2012), 11 pages. https://doi.org/10.1145/2185520.2185551

Pramook Khungurn, Daniel Schroeder, Shuang Zhao, Kavita Bala, and Steve Marschner. 2015. Matching Real Fabrics with Micro-Appearance Models. *ACM Trans. Graph.* 35, 1, Article 1 (Dec. 2015), 26 pages. https://doi.org/10.1145/2818648

Vladimir G. Kim, Wilmot Li, Niloy J. Mitra, Siddhartha Chaudhuri, Stephen DiVerdi, and Thomas Funkhouser. 2013. Learning Part-based Templates from Large Collections of 3D Shapes. *ACM Trans. Graph.* 32, 4, Article 70 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461933

Ali Sinan Koksal, Yewen Pu, Saurabh Srivastava, Rastislav Bodik, Jasmin Fisher, and Nir Piterman. 2013. Synthesis of biological models from mutation experiments. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 469–482.

Yuki Koyama, Shinjiro Sueda, Emma Steinhardt, Takeo Igarashi, Ariel Shamir, and Wojciech Matusik. 2015. AutoConnect: Computational Design of 3D-printable Connectors. *ACM Trans. Graph.* 34, 6 (Oct. 2015), 231:1–231:11.

Manfred Lau, Akira Ohgawara, Jun Mitani, and Takeo Igarashi. 2011. Converting 3D Furniture Models to Fabricatable Parts and Connectors. In *ACM SIGGRAPH 2011 Papers (SIGGRAPH '11)*. ACM, New York, NY, USA, Article 85, 6 pages. https://doi.org/10.1145/1964921.1964980

Truc Le and Ye Duan. 2017. A primitive-based 3D segmentation algorithm for mechanical CAD models. *Computer Aided Geometric Design* 52 (2017), 231–246.

Yangyan Li, Xiaokun Wu, Yiorgos Chrysathou, Andrei Sharf, Daniel Cohen-Or, and Niloy J Mitra. 2011. Globfit: Consistently fitting primitives by discovering global relations. In *ACM Transactions on Graphics (TOG)*, Vol. 30. ACM, 52.

Roee Litman, Alex Bronstein, Michael Bronstein, and Umberto Castellani. 2014. Supervised Learning of Bag-of-features Shape Descriptors Using Sparse Coding. *Comput. Graph. Forum* 33, 5 (Aug. 2014), 127–136. https://doi.org/10.1111/cgf.12438

Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. 2016. Interactive Sketching of Urban Procedural Models. *ACM Trans. Graph.* 35, 4, Article 130 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925951

OpenSCAD. 2018. The Programmers Solid 3D CAD Modeller. (2018). Retrieved January 18, 2018 from http://www.openscad.org/

Chi-Han Peng, Yong-Liang Yang, Fan Bao, Daniel Fink, Dong-Ming Yan, Peter Wonka, and Niloy J. Mitra. 2016. Computational Network Design from Functional Specifications. *ACM Trans. Graph.* 35, 4, Article 131 (July 2016), 12 pages. https://doi.org/10.1145/2897824.2925935

Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. 2004. Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms. *IJHPCA* 18, 1 (2004), 21–45. https://doi.org/10.1177/1094342004041291

Aristides AG Requicha and Jarek R Rossignac. 1992. Solid modeling and beyond. *IEEE computer graphics and applications* 12, 5 (1992).

Daniel Ritchie, Ben Mildenhall, Noah D. Goodman, and Pat Hanrahan. 2015. Controlling Procedural Modeling Programs with Stochastically-ordered Sequential Monte Carlo. *ACM Trans. Graph.* 34, 4, Article 105 (July 2015), 11 pages. https://doi.org/10.1145/2766895

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices* 49, 6 (2014), 53–64.

Ruwen Schnabel, Roland Wahl, and Reinhard Klein. 2007. Efficient RANSAC for point-cloud shape detection. In *Computer graphics forum*, Vol. 26. Wiley Online Library, 214–226.

Adriana Schulz, Jie Xu, Bo Zhu, Changxi Zheng, Eitan Grispun, and Wojciech Matusik. 2017. Interactive Design Space Exploration and Optimization for CAD Models. *ACM Transactions on Graphics* 36, 3 (2017).

Michael Schwarz and Peter Wonka. 2014. Procedural Design of Exterior Lighting for Buildings with Complex Constraints. *ACM Trans. Graph.* 33, 5, Article 166 (Sept. 2014), 16 pages. https://doi.org/10.1145/2629573

Tianjia Shao, Dongping Li, Yuliang Rong, Changxi Zheng, and Kun Zhou. 2016. Dynamic Furniture Modeling Through Assembly Instructions. *ACM Trans. Graph.* 35, 6, Article 172 (Nov. 2016), 15 pages. https://doi.org/10.1145/2980179.2982416

Vadim Shapiro. 2001. A convex deficiency tree algorithm for curved polygons. *International Journal of Computational Geometry & Applications* 11, 02 (2001), 215–238.

Vadim Shapiro and Donald L Vossler. 1991. Construction and optimization of CSG representations. *Computer-Aided Design* 23, 1 (1991), 4–20.

Vadim Shapiro and Donald L Vossler. 1993. Separation for boundary to CSG conversion. *ACM Transactions on Graphics (TOG)* 12, 1 (1993), 35–55.

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. 2018. CSGNet: Neural Shape Parser for Constructive Solid Geometry. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Maria Shugrina, Ariel Shamir, and Wojciech Matusik. 2015. Fab Forms: Customizable Objects for Fabrication with Validity and Geometry Caching. *ACM Transactions on Graphics* 34, 4 (July 2015), 100:1–100:12.

Rohit Singh, Venkata Vamsikrishna Meduri, Ahmed K. Elmagarmid, Samuel Madden, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Armando Solar-Lezama, and Nan Tang. 2017. Synthesizing Entity Matching Rules by Examples. *PVLDB* 11, 2 (2017), 189–202. http://www.vldb.org/pvldb/vol11/p189-singh.pdf

Armando Solar-Lezama. 2008. *Program Synthesis By Sketching*. Ph.D. Dissertation. EECS Dept., UC Berkeley.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 404–415.

Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. 2011. Metropolis Procedural Modeling. *ACM Trans. Graph.* 30, 2, Article 11 (April 2011), 14 pages. https://doi.org/10.1145/1944846.1944851

Thingiverse. 2018. Thingiverse: Digital Designs for Physical Objects. (2018). https://www.thingiverse.com/

Shubham Tulsiani, Hao Su, Leonidas J Guibas, Alexei A Efros, and Jitendra Malik. 2017. Learning shape abstractions by assembling volumetric primitives. In *Proc. CVPR*, Vol. 2.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

Julien Valentin, Vibhav Vineet, Ming-Ming Cheng, David Kim, Jamie Shotton, Pushmeet Kohli, Matthias Niessner, Antonio Criminisi, Shahram Izadi, and Philip Torr. 2015. SemanticPaint: Interactive 3D Labeling and Learning at Your Fingertips. *ACM Trans. Graph.* 34, 5, Article 154 (Nov. 2015), 17 pages. https://doi.org/10.1145/2751556

Daniel Weiss. 2009. *Geometry-based structural optimization on CAD specification trees*. Ph.D. Dissertation. ETH Zurich.

Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. 2014. Inverse Procedural Modeling of Facade Layouts. *ACM Trans. Graph.* 33, 4, Article 121 (July 2014), 10 pages. https://doi.org/10.1145/2601097.2601162

Q Wu, K Xu, and J Wang. 2018. Constructing 3D CSG Models from 3D Raw Point Clouds. In *Computer Graphics Forum*, Vol. 37. Wiley Online Library, 221–232.

Jianhua Wu Leif Kobbelt. 2005. Structure recovery via hybrid variational surface approximation. In *Computer Graphics Forum*, Vol. 24. Wiley Online Library, 277–284.

Zhige Xie, Kai Xu, Ligang Liu, and Yueshan Xiong. 2014. 3D Shape Segmentation and Labeling via Extreme Learning Machine. *Comput. Graph. Forum* 33, 5 (Aug. 2014), 85–95. https://doi.org/10.1111/cgf.12434

Mingliang Xu, Mingyuan Li, Weiwei Xu, Zhigang Deng, Yin Yang, and Kun Zhou. 2016. Interactive mechanism modeling from multi-view images. *ACM Trans. Graph* 35, 6 (2016), 236.

Dong-Ming Yan, Wenping Wang, Yang Liu, and Zhouwang Yang. 2012. Variational mesh segmentation via quadric surface fitting. *Computer-Aided Design* 44, 11 (2012), 1072–1082.

Youyi Zheng, Hongbo Fu, Daniel Cohen-Or, Oscar Kin-Chung Au, and Chiew-Lan Tai. 2011. Component-wise Controllers for Structure-Preserving Shape Manipulation. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 563–572.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *arXiv preprint arXiv:1605.04797* (2016).

Chenyang Zhu, Renjiao Yi, Wallace Lira, Ibraheem Alhashim, Kai Xu, and Hao Zhang. 2017. Deformation-driven Shape Correspondence via Shape Recognition. *ACM Trans. Graph.* 36, 4, Article 51 (July 2017), 12 pages. https://doi.org/10.1145/3072959.3073613