DAN CASCAVAL, RASTISLAV BODIK, and ADRIANA SCHULZ, University of Washington, USA

3D Computer-Aided Design (CAD) modeling is ubiquitous in mechanical engineering and design. Modern CAD models are programs that produce geometry and can be used to implement high-level geometric changes by modifying input parameters. While there has been a surge of recent interest in program-based tooling for the CAD domain, one fundamental problem remains unsolved. CAD programs pass geometric arguments to operations using *references*, which are queries that select elements from the constructed geometry according to programmer intent. The challenge is designing reference semantics that can express programmer intent across all geometric topologies achievable with model parameters, including topologies where the desired elements are not present. In current systems, both users and automated tools may create invalid models when parameters are changed, as references to geometric elements are lost or silently and arbitrarily switched. While existing CAD systems use heuristics to attempt to infer user intent in cases of this undefined behavior, this best-effort solution is not suitable for constructing automated tools to edit and optimize CAD programs. We analyze the failure modes of existing referencing schemes and formalize a set of criteria on which to evaluate solutions to the CAD referencing problem. In turn, we propose a domain-specific language that exposes references as a first-class language construct, using user-authored queries to introspect element history and define references safely over all paths. We give a semantics for fine-grained element lineage that can subsequently be queried; and show that our language meets the desired properties. Finally, we provide an implementation of a lineage-based referencing system in a 2.5D CAD kernel, demonstrating realistic referencing scenarios and illustrating how our system safely represents models that cause reference breakage in existing CAD systems.

$\label{eq:ccs} \texttt{CCS Concepts:} \bullet \textbf{Theory of computation} \rightarrow \textbf{Program constructs}; \bullet \textbf{Computing methodologies} \rightarrow \textbf{Shape modeling}.$

Additional Key Words and Phrases: shape modeling, persistent naming, computer-aided design programs

ACM Reference Format:

Dan Cascaval, Rastislav Bodik, and Adriana Schulz. 2023. A Lineage-Based Referencing DSL for Computer-Aided Design. *Proc. ACM Program. Lang.* 7, PLDI, Article 109 (June 2023), 24 pages. https://doi.org/10.1145/3591223

1 INTRODUCTION

Modern Computer-Aided Design (CAD) systems, such as SolidWorks [Systèmes 1995], OnShape [PTC 2015], CATIA [Systèmes 1982], Siemens NX [Siemens 1973], and PTC Creo [PTC 2011] represent 3D models as domain-specific programs that construct geometry by adding or subtracting geometric elements [Foley 1996]. What distinguishes CAD programs from generic shape modeling systems used in art or animation [Brunelli 2014] is that CAD programs are parameterized, allowing designers to generate a variation of the model geometry by adjusting the parameters and reevaluating the program, rather than by low-level geometric edits such as moving individual vertices. Ideally, a

Authors' address: Dan Cascaval, cascaval@cs.washington.edu; Rastislav Bodik, rastislavb@google.com; Adriana Schulz, adriana@cs.washington.edu, University of Washington, USA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2023 Copyright held by the owner/author(s). 2475-1421/2023/6-ART109 https://doi.org/10.1145/3591223 parametric CAD program also ensures safety in that, no matter what the parameters, the resulting geometry either is a valid design (i.e., it meets desired constraints) or the program terminates with an error; the program should never fail silently, producing invalid geometry such as a metal plate missing critical holes, a structural element that is cut off – insidious failures that are difficult to spot outright.

Treating CAD models as programs also enables novel applications, such as synthesizing parametric CAD programs from existing non-parametric designs [Du et al. 2018; Nandi et al. 2018; Xu et al. 2021], optimizing design properties such as cost and physical strength [Schulz et al. 2018, 2017], increasing interactivity of modeling interfaces [Cascaval et al. 2021; Gaillard et al. 2022; Michel and Boubekeur 2021], and otherwise expanding the capability of CAD tools [Hafner et al. 2019; Jones et al. 2021b]. Unfortunately, the full potential of parametric CAD has not been realized in existing tools [Yares 2013] because CAD programs tend to fail silently, producing invalid geometry and requiring imperfect workarounds (see Section 2). This brittleness necessitates constant manual examination of program output, which in turn prevents synthesis and optimization that need to execute the program autonomously to search for desired parameter values. Domain-Specific Languages have proven to be a fruitful intersection point between Programming Languages and CAD, which we describe further in Section 6.3.

This paper analyzes the sources of failures in existing systems and traces them to semantic issues in how CAD languages define so-called *geometric references*, which select geometric elements created by the program such as the edges where a chimney intersects the roof. The referenced elements are used to position new elements, such as the flashing that seals the chimney opening in the roof. We identify the challenges in designing a referencing semantics suitable for CAD, formulate properties that the semantics should adhere to, and provide an efficient design and implementation of a domain-specific language with first-class referencing constructs.

Figure 1 elaborates the motivation for including a *reference* construct in a CAD language. The primary motivation is to replace absolute spatial positioning of geometry, which is hard to parameterize and encapsulate, with relative positioning. References can be thought of as queries that select geometric elements (points, segments, or surfaces) previously constructed by the program; once identified, these elements can be used by the program to position, resize and otherwise parameterize the rest of the geometry relative to the existing geometry. In Figure 1, the orange query returns a reference to an edge into which the program then cuts a groove with the InsetEdge operation. We have positioned the groove relative to the edge, so the program constructs a valid design even when the parameters change because the groove moves with the edge.

Relative positioning is of course possible without a query construct. However, the lack of references leads to overwhelming complexity and edge cases. Figure 1 shows how the programmer using procedural modeling tools [McNeel 2007; SideFX 1996] might *compute* the position of the desired edge, rather than *finding* it in the constructed geometry object. To achieve relative positioning with such "geometric positioning," the programmer must manually re-derive the positions of existing elements, (i) duplicating the work done inside CAD operations such as Chamfer, and (ii) potentially recomputing the positions with a different rounding error. This approach also (iii) causes silent failures when those elements do not exist for some input parameters. In contrast, our scheme allows references to all existing elements based on their computational history (lineage), addressing all three concerns.

To understand the referencing problem, it may help to relate it to the classical problem of constructing data structures. When a data structure is modified, a client-facing function returns references to the interesting elements just constructed. CAD operations also modify a data structure, but they are higher-level and can result in the creation of multiple elements per operation dependent on the state of the execution so far. The goal is to return not just a handle to the complex shape

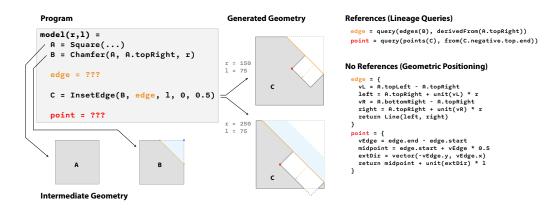


Fig. 1. **Lineage queries vs. Global Positioning** In this example, we describe a program (left), in which the user chamfers the corner of the square and then insets part of the created diagonal segment (orange) from the midpoint (0.5). The user is then concerned with referring to a point on this inset (red). We show the generated geometries on two parameter points, and on the right show how this point can be accessed using references based on lineage queries (this work) versus without referencing constructs (geometric positioning).

created, but also handles to the individual elements created within the data structure. Our approach is to denote those elements based on their lineage, and query the complex shape using this lineage name.

The key challenge in designing such a naming semantics is that the *topology* (number of points, segments, surfaces, and their connectivity) of the generated geometry may vary with program parameters, i.e. parameter changes do not merely scale the design but can cause elements to appear or disappear. Figure 1 shows two distinct topologies of geometry *C* under two parameter values. If the topology was fixed, then we could rely on indexing names such as "the 7th vertex of the 3rd polygon created." Unfortunately, indexing is not invariant under topology changes.

Additionally, topology changes introduce ambiguity: given a reference to a point in one topological configuration, which point should that reference resolve to in a different configuration, if any? The answer is model-specific, yet CAD systems do not give the programmer control over resolving references across topology changes. Instead, they employ opaque heuristics that track how referenced elements move in space as parameters change [Cheon et al. 2012; Farjana and Han 2018]. We describe the failures of such entity matching approaches in Section 2. Existing libraries for CAD programming such as CADQuery rely on programmer-written functions that select elements based on geometric state (Section 6), which introduce additional issues; others such as OpenSCAD [Kintel and Wolf 2010] disallow referencing altogether, resulting in significantly decreased usability.

This problem extends outside CAD, for example in the domain of Dynamic Geometry Systems used in mathematics education [Kösa and Karakuş 2010] where students make geometric constructions, label specific elements, and move them around; and the problem of directly editing program-generated geometry as done by the Sketch-n-Sketch [Chugh et al. 2016] language, where references are made to nearly any type of geometry. Language-based solutions have also been applied to slideshow layout [Findler and Flatt 2006], requiring references into a hierarchical functional data structure. However, none of these solutions meet the needs of CAD, because CAD requires the ability to define a program over multiple topologies, and to unambiguously define references in those topologies. Based on our study of CAD designs and CAD system failures, we propose three criteria that a referencing language must meet to express user intent (see Section 2.2). At a high level, these are: every element can be uniquely queried (*Selectability*), queries can be made in a way that does not "jump" discontinuously (*Smoothness*), and topologies can be distinguished (*Distinguishability*), allowing users to explicitly disallow topologies and bound the input domain.

In this work, we provide a language of core CAD operations and referencing semantics that achieves all of these properties, which no prior systems do. We exploit the key insight that **for each CAD operation, there exists a correspondence between the elements in its input and output**. Consider a particular piece of geometry (e.g. a point) we would like to refer to - while the *global* program slice generating this point may change as the program is edited (by adding and removing operations, or changing parameters causing conditional branch switches), the operation that generates the point can still *locally* identify it; e.g. as the intersection of a particular two lines. As a result, we can construct a *lineage relation* of such correspondences throughout a program, and use this relation to select elements via queries on *local properties* of their computational history, as opposed to a specific global program slice. We note that considering element lineage is not a novel idea: lineage information is tracked and used as input to heuristic based matching methods as well [Cheon et al. 2012; Michel and Boubekeur 2021]. However, we are the first to formalize this notion and define specific criteria for a referencing system, and to design a domain-specific language for solving the referencing problem robustly via lineage queries – prior systems that use this notion still end up with systems that can fail silently.

We show that our language provides the above properties. Our system can select any subset of vertices and achieves smoothness. In fact, we permit a single query to refer elements across topology changes, which is safe as long as the selected element(s) do not jump as the parameters move. This is done by formulating queries as boolean combinations of reachability in a program *lineage graph* composed of two different kinds of edges. As the operations assign edges, CAD users do not have to author any geometric computations (Figure 1) or rely on heuristics perform reliable references.

Using a small user study, we show that queries can be formulated concisely by programmers. We demonstrate the language's applicability on real-world models using a set of three case studies that expose reference robustness issues in existing systems; eliminating issues with lost, spurious, or ambiguous references across program parameter changes. Additionally, we show that our design imposes only a small (less than 20%) amount of overhead on the necessary geometric computations in our benchmarks. To summarize, we contribute:

- An analysis of referencing failures in existing CAD systems (Section 2) and desirable properties to avoid such failures (Section 4).
- The design and implementation of a novel DSL based on these domain insights (Section 3)
- An empirical and qualitative evaluation of the DSL with respect to specified properties and model benchmarks (Section 5)

2 OVERVIEW

2.1 Reference Failures

To get a sense of the classes of different reference failures that existing systems exhibit, we will look at several examples of CAD models constructed in OnShape [PTC 2015], a popular browserbased CAD system built using the Parasolid Kernel [Siemens 1985], which underlies many other major CAD systems. OnShape uses a referencing scheme based on entity-matching, leading to the corner cases we see in Figure 2. (A) A *spurious reference* occurs when the model moves to a new configuration and the reference resolves to elements a user does not want to select, in

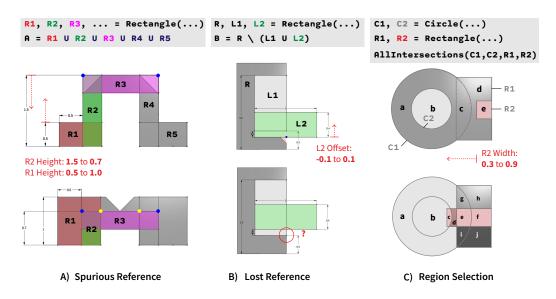


Fig. 2. **Reference failures in OnShape.** We demonstrate three classes of reference failure caused by topological changes due to parameter shifts on simple programs. The initial program is displayed on the top row, and a silent failure due to parameter changes (red) is displayed on the bottom row. (a) chamfered corner points on the union of rectangles (blue) move "inside" the union when the height of the sides changes, the user would like not to apply chamfers in this case, but heuristics spuriously select different points (yellow) and apply an exterior chamfer in the middle section, (b) a point created by subtracting a union of two polygons from a rectangle *R* disappears when one negative polygon moves. The user expects the point to be the corner in the red circle, but heuristics fail to find a match, (c) a number of 2D curves are intersected, and specific intersecting regions are individually selected to create a solid, but the system creates a garbled solid when a rectangle curve is lengthened and a different set of regions are created.

a discontinuous way. (B) A *lost reference* occurs when there is a valid notion of a continuous referenced object, but the referencing system fails to track this across configurations. (C) Both can happen simultaneously, and on any type of geometric element, not just points - in Figure 2, specific 2D areas created by curve intersections are selected (a,c,d). It is ambiguous to the system which areas those regions should map to (a,g,h,i,j) when the curves intersect differently, resulting in a spurious undesired element (d), and a missing element (a).

In addition to CAD systems, Dynamic Geometry Systems used in mathematics education such as Cabri [Kösa and Karakuş 2010] face this problem as well. In Cabri, users construct mathematical diagrams and "refer" to elements by giving them labels, which move around when the geometry is manipulated. Similar issues occur: references can be lost or discontinuous, but Cabri attempts to resolve the issue by placing ad-hoc restrictions on how elements can be moved after labels are applied, seemingly preventing certain topological changes that would cause labels to disappear. Unfortunately, these restrictions are not completely robust, since elements can still be made to disappear, and are not viable in a CAD context, where user control over model variation is critical.

2.2 Robustness Properties

However, a language that adheres to the three robustness properties mentioned in Section 1 will avoid these lost or spurious reference errors. Figure 3 shows examples of these properties in practice. Selectability of elements precludes never being able to recover a lost reference, and ensures that

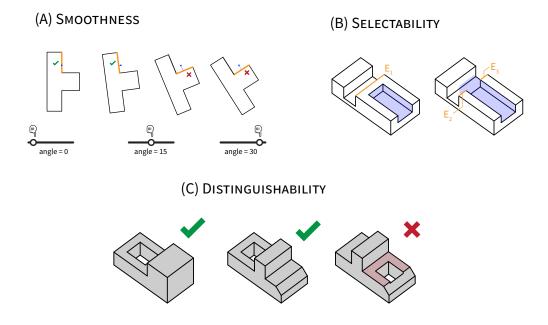


Fig. 3. **Robustness properties. (a)** The user adjusts a slider, smoothly rotating a polygon around its center. An algorithm to select an edge by checking the nearest edge to the blue point will exhibit a discontinuity at the point where the closest edge changes – even though the program geometry has no discontinuities with respect to the angle parameter. A language that exhibits smoothness does not introduce discontinuities via querying. **(b)** The user adjusts another slider, lengthening the blue slot until it intersects the other slot. This causes the edge E1 to bifurcate into edges E2 and E3. Selectability of elements guarantees that we will be able to select E2, E3, both, or none, in the case that they exist, and select E1 in the case it exists. This specifying potential is not available in previous CAD systems, which only allow specifying a selection on a single topology. **(c)** The user's program outputs different topological configurations, but not all of them should be considered valid. Distinguishability of topologies allows users to disallow classes of topologies based on the presence or absence of elements with particular lineage - in this case, the user asserts against the configuration where the hole and diagonal chamfer both affect the same face, due to manufacturing concerns.

ambiguous cases can always be handled. Distinguishability of topologies ensures that a program exists that can distinguish between any two topologies, ensuring that a non-spurious selection can be constructed on a specific, given topology. Since there are combinatorially-many topologies, manual constraints may be painful – instead we use an assertion on the presence or absence of a topological element with specific lineage to allow or disallow entire classes of topologies, rather than the user reasoning about valid parameter ranges directly.

In turn, query smoothness ensures that references *can be* constructed that vary smoothly with respect to parameter changes on that topology. It does not imply that *every* referenced point will vary smoothly, or that the program overall is continuous. It is important that this be possible, because the referenced point is often used as the basis for a subsequent operation – for example, if we select a corner in order to trim it, we would like that corner to vary smoothly with a continuous parameter change so that our cut does not jump around in space.

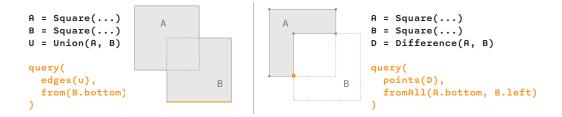


Fig. 4. An example of lineage queries identifying elements on programs whose topologies may vary as squares change size or relative position. On the left (a), an edge is queried, and on the right (b) a point is queried.

We will demonstrate in Section 5 how our DSL can be used to construct these references, avoiding both lost and spurious elements by constructing lineage queries that return the intended result on both parameter inputs. We give a motivating example of such queries in Figure 4, where the user is able to select specific sets of elements from the union or difference of two squares: while the set of segments and points in the resulting topology may change depending on how the squares intersect, queries are conceptually able to *trace* an already-referenced element (in this case, a primitive) through a given operation, using it to disambiguate between the sets of points and edges in the intersection. We say the edge in 4a is from(B.bottom) because it is the corresponding edge on the union, and the point in 4b is fromAll(A.bottom, B.left) because it results from the intersection of both of these two lines, and can be uniquely identified as the only point with this property.

2.3 Design Motivation

To motivate our query design, we describe two alternative designs. First, we examine the design of Sketch-n-Sketch [Chugh et al. 2016], which performs a similar tracing approach to determine the provenance of geometric elements in terms of parameters. This procedure finds an unambiguous function for all selectable elements that determines, for a given manipulation of the element, how the parameters should change. The advantage of this design is ease of use – unambiguous selections can be made simply by clicking an object. However, this tracing procedure has only limited support for conditionals, so cannot account for elements that result from geometric intersections (which may, or may not, exist) and therefore cannot represent programs that generate multiple topologies.

Another potential design builds up a unique global lineage trace identifying each element (e.g. by recursively defining an identifier string) and uses it as a global index. This allows parameter changes to move across topologies, as long as those changes do not affect the reference (i.e. they are not part of the lineage), as that would change the trace. However, this design is quite verbose, since users have to specify the entire trace; and inflexible, as the trace must be modified when the program changes (e.g. an intermediate operation is added) and precludes a single program that refers to an element which is continuous but whose lineage does change. In contrast, our query design both allows references through topological changes (as queries can be seen as analogous to a class of lineage traces), and minimizes verbosity and editing overhead when the program is modified. This is because queries filter on *properties* of an element's history, rather than an exact globally-unique tag or program slice. Moreover, this history is specified transitively: a query can reference lineage through an arbitrary number of operations. This property of the design results in query length being independent of program length.

Our design is motivated by (a) usability in terms of effort in authoring and reasoning about queries, (b) computational overhead, and (c) expressivity in the domain, i.e. the geometries and



Fig. 5. **Geometric types.** Segments can be either lines or arcs. Polygons are closed, simply-connected (no holes) sequences of segments. Faces represent 2D areas, by representing each *boundary* (i.e outer contour or interior hole) as a polygon. 3D solids are faces extruded along a vector, or revolved along an axis. The subset relation denotes *containment*.

references we care about. We aim for a language that is both straightforward to reason about and able to express realistic models.

3 LANGUAGE

3.1 Language Domain

Our primary geometric representation is known as a Boundary Representation (B-Rep), which is used pervasively in CAD tools because it represents solid volumes with analytical precision. A B-Rep is a graph in which the vertices are the geometric elements (physical points, segments, faces, and solids) and the edges denote connectivity between elements. We say that a *topological change* has occurred if elements are added or removed from the graph, or the connectivity otherwise changes. We define a *parametric model* as a function that takes in a set of scalar input parameters and produces a B-Rep of zero or more solids. In CAD systems, such a model is typically produced as a series of calls to library operations which take as argument a mix of geometric and scalar parameters.

For the purposes of our discussion, we will consider referencing schemes as referring to specific geometric elements created by a parametric model. We note that the representation allows a natural hierarchy of geometric elements: each segment (line, or arc) *contains* two points, each simple face contains a closed loop of segments, each complex face contains 1 or more simple polygons, and each solid contains multiple faces. We define *containment* as a natural property of this structure; where an element recursively contains all elements contained by its elements through the single-step relation, and additionally contains itself. We use the meta-notation elements(e) to refer to the set of contained elements of e.

3.2 Constructs & Operators

3.2.1 Modeling Operators. Geometrically, the language provides:

- **2D Primitives** such as Point, Line, Circle, Rectangle, and Polygon. Each element produced by a primitive can be uniquely referred to *statically* via fields (e.g. rect.top).
- Transforms (e.g. rotation, translation) which do not change topology,
- Set operations, which compose polygons via union, difference, or intersection to form new polygons or faces in the style of Constructive Solid Geometry [Foley 1996].
- 2D CAD operators (ExtrudeSegment, Chamfer, and Fillet) which can be seen as syntactic sugar for creating a new primitive, transforming it, and performing a static series of set operations to combine it with an existing polygon or face.
- 3D operations to Extrude or Revolve a 2D face along an axis to create a solid.

| geo | = | Point Segment Polygon Face | | | | | |
|-----------|-----|---|--|--|--|--|--|
| primitive | = | Point Line Arc Rectangle Circle | | | | | |
| transform | = | Translate Rotate Scale Mirror | | | | | |
| field-op | = | points segments polygons | | | | | |
| set-op | = | Union Difference Intersection | | | | | |
| cad-op | = | Chamfer Fillet ExtrudeSegment InsetSegment | | | | | |
| assert-op | = | single empty | | | | | |
| ор | ::= | primitive : $\mathbb{R}^k \to qeo$ | | | | | |
| - | | set-op : Polygon \times Polygon \rightarrow Face | | | | | |
| | Ì | cad-op: Polygon × (Point Segment) × $\mathbb{R}^k \to$ Face | | | | | |
| | i | transform : $\tau \in qeo \times \mathbb{R}^k \to \tau$ | | | | | |
| | i | field-op: $\tau \in qeo \rightarrow \{\tau\}$ | | | | | |
| | i | $query - op: \{\tau\} \times q \to \{\tau\}$ | | | | | |
| | i | assert-op: $\{\tau\} \rightarrow \tau \mid ()$ | | | | | |
| | | | | | | | |
| q | ::= | all $ $ from $(e : geo) $ derived From $(e : geo)$ | | | | | |
| | ļ | contains(q) or(q, q) and(q, q) or(q, q) not(q) | | | | | |
| | ļ | fromAll((geo {geo})*) fromAny((geo {geo})*) | | | | | |
| | | derivedFromAll(($geo \{geo\}$)*) derivedFromAny(($geo \{geo\}$)*) | | | | | |
| seq-op | ::= | Tabulate : $int \times (int \to \tau) \to [\tau]$ | | | | | |
| | | $Map: [\alpha] \times (\alpha \to \beta) \to [\beta]$ | | | | | |
| | i | Index : $[\tau] \times int \rightarrow \tau$ | | | | | |
| | • | | | | | | |

Fig. 6. **Language Operations.** In addition to the rules described the language allows for arithmetic operators on scalars as well as standard variable assignment for *expr*. In the type signatures, $\{\tau\}$ denotes a homogenous set of elements of type τ , and $[\tau]$ denotes an *indexable* (statically-sized and ordered) sequence of type τ . *int* must be a statically-known constant. Lambdas are created using an arrow-notation syntax described in the supplemental material.

Each operation is immutable, e.g. chamfering a corner of a polygon will return a new polygon with the chamfer applied. Note that we do not provide the union, intersection, or difference operations for solids, and our implementation is limited to objects that can be represented using straight lines and circular arcs. This set of operations can be extended by implementing lineage for a new operation in accordance with the semantics in Section 3.3.

While set and transform operations are guaranteed to successfully execute, CAD operations may not. These operations have natural bounds: chamfering or filleting a corner can only go so far before the other ends of the segments adjacent to that corner are reached, so attempting to perform these operations with a parameter that is too large results in an error for geometric reasons. In addition, user-defined assertions may assert the cardinality of a query result, erroring if the condition is violated. In the case of a type-error (such as accessing a non-existent field), an error can be detected statically. However, none of these failures are silent: in all cases, an error is displayed to the user, and the parameter-point is considered invalid.

Figure 6 provides a partial description of the signatures of these operations. Please see our supplemental material for full documentation of the modeling operations included in the language, as well as their types, accessible fields, and error conditions.

3.2.2 Referencing Operators. Our referencing scheme centers around one key property: while CAD operations can produce shapes with dynamic topologies, for each CAD operation there exists a correspondence between the elements in its input and output. We formalize this into a *lineage relation*, namely, a fine-grained tracking through operations relating entities in the input to entities in the output. In our system, this is represented as a relation (Figure 7) between geometric elements, with different types of *lineage edges* being drawn by operations between these elements.

Each operation is responsible for defining lineage in accordance to some formal rules (Section 3.3), and we give examples of such definitions for our provided operations (Section 3.5). On top of this relation we define *lineage queries* that filter elements of a set whose contents vary dynamically using reachability in the lineage relation. This creates a safe way of identifying those elements that is both independent of geometry and requires no additional geometric logic on the part of the user.

This mechanism of queries allows us to reference *transitively* through a series of operations. Because operations generate lineage dynamically on each execution path, queries can be resolved without storing state between executions. Because operation lineage adheres to an unambiguous semantics, references can be reasoned about across topologies ahead of time, and operations being immutable allows the intermediate states of a transitive reference query to be inspected (e.g. by drawing them), aiding reasoning. Further, assertion primitives allow us to bound the space of models, and to use the knowledge of that assertion to safely access specific elements.

3.3 Lineage semantics

We provide two main rules for defining operation lineage; using three types of lineage edges: *subset* edges, *transform* edges, and *deriving* edges; each of which are directed from input elements to output elements of a given operation, these are the lineage relations L_S , L_T , and L_D respectively. We refer to geometric elements as as lower case; and sets of such elements in the upper case.

We define an element v as *from* an element u if v is reachable from any $e \in elements(u)$ using subset or a transform edges. We say v is *derived* from u if there if is is reachable using any type of lineage edge (Section 3.4). All three edge types are created in Figure 7.

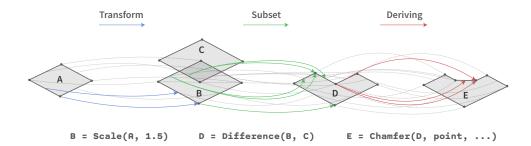


Fig. 7. **Lineage Relation.** We depict the lineage edges created by executing a program that transforms a square A, subtracts a square C from it, and chamfers a resulting corner. This incorporates all three types of lineage edges - we highlight a few edges of each type for legibility, while the remaining edges are dashed.

A subset edge (u, v) is created when an operation creates an output element v that is computed as a *geometric subset* of an input element u on a given execution path; for example, shortening a segment or intersecting two segments (Figure 7). A subset edge is also drawn when an element is copied from input to output.

An operation *T* is a *transform* if for any input *U*, it produces an output *V* with exactly the same topology as *U*; and additionally that *elements* $(\bigcup_{u \in elements(U)} T(u)) = elements(T(U))$. In other words, there is a clear bijection between input and output elements over all paths. In this case there must be a transform edge (u, v) between each input element $u \in elements(U)$ and the corresponding transformed element v = T(u). Examples include translation, rotation, scaling, and mirroring.

A deriving edge (u, v) is defined by any operation F(U) = V that creates conceptually *new geometry*: Some elements $v \in V$ are neither a subset of any element of U over all inputs, and F does not meet the definition of a transform. In this case, F is required to assign deriving edges that satisfy the following two properties:

- The operation F must be able to define an overlapping partition $F_1, ..., F_k$ over *elements*(V) such that
- (1) Each F_i contains elements of homogenous geometric type τ , and
- (2) For each $e : \tau \in elements(V)$, there exists F_i and a set of geometric inputs $G \subseteq elements(U)$ such that (a) e is reachable (via L_S , L_T , and L_D together) from all $g \in G$, and (b) Any $e' : \tau \in elements(V)$ reachable from all $g \in G$ implies e = e'.

In other words, the output of the operation F is partitioned statically, and deriving edges are assigned in such a way that there exists some set of input elements G that uniquely *distinguishes e* from every other element in the output.

• There must be no edges (u, v) such that perturbing u could result in *not* perturbing v, holding all operation input parameters held equal.

Note that subset edges are also assigned by *F* where the subset rules apply, and deriving edge assignment must account for this when meeting the first property. The second property is included primarily to narrow down the space of possible lineage edge assignments that satisfy the first.

We observe two useful properties of these edge assignment semantics: First, in any set of elements V output by an operation, an element $e \in elements(V)$ is not reachable from any element of V other than itself - i.e. there are no edges between output elements themselves. Second, the edges defined by a given operation do not change or affect the reachability between any elements already part of an edge in the lineage relation, since each operation adds only directed edges towards newly created or copied geometry.

3.4 Query Resolution

The dynamic semantics for our query operators is defined in terms of a query universe term *S*, a query *q*, and a resolution operator \neg , such that $S, L \vdash q \neg R$ implies that query(*S*, *q*) evaluates to *R* when executed with the lineage relation *L*, which we use to abbreviate the 3-tuple L_S, L_T, L_D of defined relations. $L^*(u, v)$ denotes transitive reachability between *u* and *v* in *L*.

$$\frac{r_{s} = \bigvee_{g \in elements(e)} L_{S,T}^{*}(g,s)}{S,L \vdash \text{from}(e) \twoheadrightarrow \{s \in S \mid r_{s}\}} \qquad \frac{r_{s} = \bigvee_{g \in elements(e)} L_{S,T,D}^{*}(g,s)}{S,L \vdash \text{derivedFrom}(e) \twoheadrightarrow \{s \in S \mid r_{s}\}}$$

$$\frac{\forall_{s \in S} \ elements(s), L \vdash q \twoheadrightarrow R_{s}}{S,L \vdash \text{contains}(q) \twoheadrightarrow \{s \in S \mid R_{s} \neq \emptyset\}}$$

$$\frac{S,L \vdash q_{1} \twoheadrightarrow R_{1} \quad S,L \vdash q_{2} \twoheadrightarrow R_{2}}{S,L \vdash \text{and}(q_{1},q_{2}) \twoheadrightarrow R_{1} \cap R_{2}} \qquad \frac{S,L \vdash q_{1} \twoheadrightarrow R_{1} \quad S,L \vdash q_{2} \twoheadrightarrow R_{2}}{S,L \vdash \text{or}(q_{1},q_{2}) \twoheadrightarrow R_{1} \cup R_{2}} \qquad \frac{S,L \vdash q \twoheadrightarrow R_{s}}{S,L \vdash \text{or}(q_{1},q_{2}) \twoheadrightarrow R_{1} \cup R_{2}}$$

For convenience in the language, we also include constructs to easily query from a conjunction or disjunction of elements. The primary semantic distinction between these and and(from(e_1), ..., from(e_k)) or or(from(e_1), ..., from(e_k)) is that they can be used to query from a set of elements, without requiring the set members to be extracted and passed individually to from queries. Here, we consider each argument e_i to be either a set, or if a geometric element is passed directly, to be the singleton set { e_i }.

$$\frac{E = \bigcup_{i} e_{i} \quad \forall_{e \in E} S, L \vdash \text{from}(e) \twoheadrightarrow R_{e}}{S, L \vdash \text{fromAll}(e_{1}, ..., e_{n}) \twoheadrightarrow \bigcap_{e} R_{e}} \quad \frac{E = \bigcup_{i} e_{i} \quad \forall_{e \in E} S, L \vdash \text{from}(e) \twoheadrightarrow R_{e}}{S, L \vdash \text{fromAny}(e_{1}, ..., e_{n}) \twoheadrightarrow \bigcup_{e} R_{e}}$$

We also provide the analogous derivedFromAll and derivedFromAny using derivedFrom in lieu of from respectively.

3.5 Operation Lineage

In this section we describe the lineage edges assigned by each of our operations. When defining how operations draw lineage edges, we will use $(u, v) \in L_S$ to denote a subset edge, $\in L_T$ to denote a transform edge, and $\in L_D$ to denote a deriving edge, as described in Section 3.3. As each operation F is sequentially executed, additional edges are added to one or more of these lineage relations, with the property that if an edge (u, v) is added, then there is no u' such that (u', v) is in the relation before F executes – i.e. lineage edges are never deleted, or added in such a way that would affect reachability. When describing the addition of lineage edges for an operation expr given the current lineage L, we say it steps to (\neg) the lineage L' for the remainder of the program P, writing $L \vdash expr :: P \rightarrow L' \vdash P$.

Primitives. An operation is considered a primitive if all of its elements are statically accessible – implying also that its topology must be fixed and known ahead of time. As a result, geometric primitives such as points, lines, and basic polygons do not define lineage edges. Instead their elements can be accessed directly as members of the primitive.

Transforms. All transforms are defined such that a transform T(e) applied to a composite element is equivalent to applying T individually to every member of elements(e). Thus we can rely on a single rule for any transform T that draws a transform edge between every element and its corresponding transformed version.

$$\frac{e_T = transform(e) \quad e \in elements(E)}{L_S, L_T, L_D \vdash transform(E) :: P \twoheadrightarrow L_S, L_T \cup E_T, L_D \vdash P}$$
TRANSFORM

We note that elements retain their identity through transforms - namely, if a transform maps all vertices to the same location (effectively collapsing *e* into the origin), all e_T s will have the same position, but there is a lineage element only between a given element and its single corresponding transformed version, not to all of the transformed elements sharing the same coordinate.

Set Operations. We note that performing set operations (union, difference, and intersection), each element on the output is either a direct copy of an element (point or segment) on the input, a subsegment of a segment on the input (denoted by \subseteq), or a point resulting from the intersection of two segments (denoted by intersect). We note that the set of initial intersections for all three set operations is identical, and that they differ only in the elements they choose to *filter*, for example, in a union, intersection points that end up inside either input polygon would be filtered out of the final result set of points. We call this initial, pre-filtered set the *shatter* set, and provide rules for all three cases of elements in the shatter set:

Proc. ACM Program. Lang., Vol. 7, No. PLDI, Article 109. Publication date: June 2023.

$$\frac{p \in \text{points}(a) \cup \text{points}(b) \quad p_v \in \text{points}(v) \quad p = p_v}{L_S, L_T, L_D \vdash shatter(a, b) :: P \twoheadrightarrow L_S \cup \{(p, p_v)\}, L_T, L_D \vdash P} \text{POINT-COPY}$$

$$\frac{e_a \in \text{segments}(a) \quad e_b \in \text{segments}(b) \quad p = \text{intersect}(e_a, e_b)}{L_S, L_T, L_D \vdash shatter(a, b) :: P \twoheadrightarrow L_S \cup \{(e_a, p), (e_b, p)\}, L_T, L_D \vdash P} \text{INTERSECT}$$

$$\frac{e \in \text{segments}(a) \cup \text{segments}(b) \quad e_v \in \text{segments}(v) \quad e_v \subseteq e}{L_S, L_T, L_D \vdash shatter(a, b) :: P \twoheadrightarrow L_S \cup \{(e, e_v)\}, L_T, L_D \vdash P} \text{Segment-Trim}$$

After filtering the appropriate elements according to the given set operation's semantics, set operations additionally perform a *merging* step, where adjacent colinear segments (or concentric arcs) are merged into a single line or arc segment on the resulting boundary. Additional deriving edges are added between the shattered elements and their final merged element in the result:

$$\frac{e \in \text{segments}(shatter(a, b)) \quad e_v \in \text{segments}(set \text{-} operation(a, b)) \quad e \subseteq e_v}{L_S, L_T, L_D \vdash set \text{-} operation(a, b) :: P \twoheadrightarrow L_S, L_T, L_D \cup \{(e, e_v)\} \vdash P} \text{Segment-Mergeric}$$

While we do not provide a detailed treatment here, we note that the same principles apply to set operations on 3D solids, lifting up a dimension: faces in 3D set operations are analogous to segments in 2D ones: faces on shattered B-Reps are subsets of the faces on the inputs, which may be merged into one face if they are adjacent and co-planar. Intersecting two faces creates an intersection curve segment, analogous to intersecting two segments to create a intersection point in 2D, and is accordingly treated as a subset.

Constructive Operations. We define a *constructive operation* as one which can be defined as the construction of one or more primitives and a subsequent set operation. Consider extruding a segment of a polygon, which can be seen as the placement of a rectangle along the corresponding segment with the given extrusion length, and a subsequent union operation. The operation outputs one or more statically-known output sets of dynamic size, and responsible for providing deriving edges to be able to distinguish elements within these sets. We provide a few operations that meet this criterion in the core language (Chamfer, Fillet, Extrude, Inset), and note that the language can be extended with more such operations provided the created geometry is continuous and the operation ensures that uniquely-selecting queries exist (Section 4).

4 LANGUAGE PROPERTIES

We previously defined three key properties of referencing languages: selectability, smoothness, and distinguishability. In this section, we will describe these properties precisely, and show that our language exhibits them. We define a Uniquely-Selecting Query (USQ), and leverage this object in a proof sketch that our language exhibits the three properties.

For any element that exists in a given parameter configuration we want to specify a function that returns the set of elements that correspond to it, based on a model-specific notion of element *identity* defined by the user. This function should be defined over the entire parametric domain. This selection function should always return precisely the elements that the user intends to match. Selectability implies means we can select for every element in every configuration that the user wants to match, so it is always possible to select all of those entities at once. In doing so, however, the selected set may include elements that the user considers spurious. To ensure only intended

elements are returned, we need to enable users to eliminate spurious results. We achieve this through the other two properties: distinguishability enables the user to bound the model to avoid topological configurations they do not want, and smoothness ensures that elements are tracked unambiguously within a given topological configuration, as continuity gives a notion of identity independent of a given model.

4.1 Uniquely-Selecting Queries

We will show that for any model *P* and parameter input θ , for each *e* produced by $P(\theta)$ there exists a domain of elements *S* and a query *Q* such that query(*S*, *Q*) \rightarrow {*e*}. As the language includes no conditional constructs, *S* and *Q* exist over all input parameters, though they may have differing contents. We will refer to the query *Q* and domain *S* as the *uniquely-selecting query* (USQ) of *e*. USQs, as defined here, will always return a set of size at most one when evaluated at any parameters θ . Further, USQs will always produce smoothly varying geometry over the parameter domain where they evaluate to a non-empty set.

Below, we sketch out how USQs can be constructed, showing that for any operation with input U and output V, a USQ exists for every element in V given that USQs exist for every element in U, casing on the operations:

4.1.1 Primitives and Transforms. By definition an operation is a primitive iff. it takes no geometric arguments and all its outputs can be uniquely statically selected, so there is always a set S equal to $\{e\}$ for each element. Transforms create a bijection, so each $e \in V$ can be selected as from(u) given the corresponding element u in the input, in any $S \subseteq V$ that contains e.

4.1.2 Set Operations. We will first prove the property for points, considering the lineage assignment rules given in Section 3.5. Set operations always intersect polygons. Points $p' \in V$ that are direct copies of existing points $p \in U$ can be selected by the query from(p) by rule POINT-COPY. No other elements will be selected as there are no other rules that draw lineage edges from points.

For points created by segment intersection, we must distinguish between the cases where the segments intersect in a single point and where they intersect in multiple, by placing the points into different sets. Intersection points are in I_0 when they are created from segments that intersect once, and int I_1 and I_2 if the segments intersect multiple times – segments can never intersect more than twice. Let segment e_a in polygon A intersect segment e_b in polygon B. At parameters where e_a and e_b intersect in a single point, the query from (e_a, e_b) on the set I_0 is sufficient (by rule INTERSECT). We note that this referenced element moves smoothly with changes to e_a and e_b , as it always selects the geometric intersection. We consider the cases where multiple points can be produced:

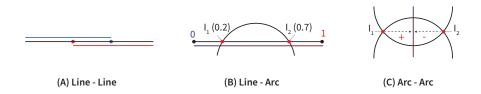


Fig. 8. Intersection cases that produce multiple points as a result of two segments

• *Line* - *Line*. If two lines overlap, the two points created must be the endpoints of e_a or e_b respectively. These are points on the input, so the output points can be queried directly from the corresponding endpoint, in either set I_1 or I_2 .

Proc. ACM Program. Lang., Vol. 7, No. PLDI, Article 109. Publication date: June 2023.

- Line Arc. We introduce geometric distinguishing information to partition into I_1 and I_2 . The operation can distinguish these points by evaluating their *parameter* along the length of the line, a normalized measure of distance between the start and end point that is 0 at the start and 1 at the end. The point with the lower parameter is placed in I_1 , and the higher in I_2 . The points can then be uniquely queried within these sets using the standard single-intersection query from (e_a, e_b) . This partition is smooth with respect to parameter changes: the elements will not ever jump between sets due to an epsilon change while the line and arc continue to intersect, and the partition is defined over all parameter points where both circles exist.
- Arc Arc. We draw a line between the arc center of e_a and the arc center of e_b , and perform a signed distance test from the line, which will always be positive for one point and negative for the other, and this sign stays constant regardless of how the arc segments are moved; the positive distance point is placed in I_1 and the negative in I_2 . In the case the two arcs share a center-point, no signed distance can be evaluated, but the points must be the endpoints of the arc segments, so they are both placed in a tertiary set I_3 , which can be queried using the arc segment endpoints as in the Line-Line case. This partition is also smooth with respect to parameter changes.

We have given USQs for all points in the result; all the segments in the result are bounded at their endpoints by points. To show segments also have USQs, we will construct a query q_1 using the contains operator to select segments $e \in S = \text{segments}(V)$ based off of its two endpoints e_{start} , e_{end} :

and(contains(from(*e*_{start})), contains(from(*e*_{end}))))

While this query will uniquely select nearly all segments, it will return two segments in the case that two segments share both endpoints, e.g. as in the Line-Arc or Arc-Arc intersection cases. In this case, the two segments must subsets of segments on different input polygons A, B to a set operation when the configuration is first created, as no primitive contains this configuration. If A and B have independent lineage, then the segments can be distinguished within the result set of q_1 above, with the final query being e.g. and $(q_1, \text{from}(a))$. If they do not, for example if B is a transform of A, then WLOG say that B is derived from A, i.e. query(*elements*(B), contains(from(A))) $\neq \emptyset$. Particularly, say that the segment e_b is from both A and B. In this case we can distinguish the edge e_b using from(B) and the edge e_a , which cannot be from B, using the query not(from(B)). We note that we must keep this final from(*polygon*) term for *all* segments, to maintain the property that the USQ will never resolve to multiple elements.

The final case we must handle is the USQ of polygons created by set operations on faces, which may themselves be comprised of several polygons describing contours and holes. Polygon USQs follow directly from point USQs, as any polygon can be uniquely selected using the query contains(from(p)) given any point p on the polygon; as we know that points are never shared between polygons. We note that faces and solids always have a trivial USQ, since a single face or solid is always created by a single operation, and each operation that takes a face or solid as input (e.g. a transform) is a 1-1 mapping, so there are never any cases where faces or solids need be dynamically queried.

4.1.3 CAD Operations. Our language additionally exposes constructive operations which can be defined by creating new geometry and performing a set operation. As set operations are already covered, we need only show USQs exist for created geometries. We show this for the Extrude and Chamfer operations (Figure 9), as the Inset and Fillet operations we include are analogous. Such USQs exist by default if they are primitives (like the rectangle created by the Extrude operation), and require the use of deriving edges if a set of geometries is created (e.g. by Chamfer, which takes subtracts a negative triangle from each argument corner to create the chamfered polygon. As there

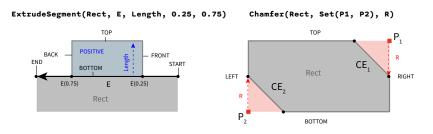


Fig. 9. Created geometry for the ExtrudeSegment and Chamfer operations. Added area is marked in blue for Extrude, removed area marked in red for Chamfer.

may be a varying number of input points passed in to Chamfer, the deriving edge from each point p is used to distinguish each created segment within the set *CE* of segments created by the chamfer, via derivedFrom(p).

4.2 Selectability and Smoothness

Selectability states that any fixed element that is produced by any program can be uniquely queried, which follows directly from the existence of USQs for every element. Smoothness asserts that each smoothly-varying element can be queried in a way that does not exhibit discontinuities where the query returns a value. An reference to a smoothly-varying element will not exhibit discontinuities in the areas where its USQ returns a result, which we can see from the fact that all elements are the result of continuous geometric operations (primitives, transforms, and intersections), and the fact that the USQs do not introduce additional discontinuities as a result of partitioning elements into the intersection sets I_1 and I_2 .

4.3 Distinguishability

A distinguishing query exists between any two different geometric topologies. We demonstrate that for every pair of distinct topologies, there exists a query that returns a non-empty set on one topology and an empty set on the other. We say that two topologies are the same if their B-Rep graphs are isomorphic, taking into account element types (such that a straight edge could not be replaced with a vertex, or an arc). Intuitively, we expect this to be true, as our USQs are analogous to abstract program slices, and if the topology has changed then a conditional branch must have executed differently, and there must be a different program slice between the two cases.

Let a particular operation result (e.g. elements(V)) produce distinct, non-isomorphic concrete topological configurations C, C' on different parameters. We break down this topological non-isomorphism into two cases:

- *C*, *C'* have a different number of geometric faces, segments, or points. We know from our proof of selectability that all USQs will return a set of at most one element over all parameters. WLOG let *C* have more elements: either there exists an element $d \in C$ such that the USQ of *d* returns a empty result on *C'* (in which case this is our distinguishing query), or there isn't in which case since *C* has more elements there must be some pair of elements $d, d' \in C$ whose USQs return the same element *e* in *C'*. In this case, the query fromAll(*d*, *d'*) is distinguishing, since it returns a non-empty result on *C'* (*e*), and returns an empty result on *C*, as since *d*, *d'* are different elements there is no element that is a subset of both.
- *C*, *C'* have the same number of faces, segments and points, but they are connected differently. Up to renaming there exists some distinguishing *topological* edge in one that is not in the other.

Topological edges correspond directly to geometric containment; i.e. there is a topological edge between a face and each constituent border segment, and from each of those segments to their own end points, which are shared between segments. Let the edge be (u, v), with v contained within u. It must thus be the case that on the configuration with this edge, the query and(from(u), contains(from(v))) returns a result, and in the other it does not, as the edge is not present and therefore no such element exists. We note that u, v are elements of the same output set V. As there are no lineage edges drawn between elements resulting from the same operation, for both u and v the only element that matches from(u) or from(v) will be u or v itself respectively.

5 EVALUATION AND DISCUSSION

We provide an empirical evaluation of our language relative to our informal design criteria stated in Section 2, as well as its performance and applicability in the broader domain of real world CAD models.

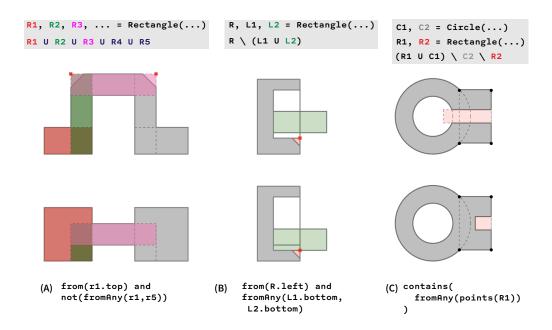


Fig. 10. **Case Studies**. These are versions of the programs from Figure 2 implemented in our system, with queries demonstrating that the desired result can be achieved using queries to avoid lost or spurious references. Both top and bottom rows display correct executions.

5.1 Hypothesis 1

Programs that exhibit reference breakage due to lost or spurious references in entity-matching systems can be safely expressed without breakage using lineage queries. In Figure 10, we show our language handles the problematic programs from Figure 2 (Section 2). Since whether a reference is lost or spurious ultimately depends on the user's modeling intent, our robustness properties guarantee this intent can be fully expressed.

First, we can select the red points in (A) without incurring a spurious reference in the bottom case by bounding this reference; e.g. saying from(r3.top) and not(fromAny(r1, r5)), which successfully selects the two points in the first case and precludes the second case, and demonstrates how references are distinguished in practice across topologies. If the user desires, a similar program can be formed asserting that the chamfer points exist and erroring on the second case instead.

Second, we select the red point in case (B) in both topologies by querying points(U) with from(R.left) and fromAny(l1.bottom, l2.bottom)). This allows the reference flexibility to jump from being the intersection of R.left and l1.bottom to intersecting l2.bottom instead; which we can locally reason is safe because the point is continuous over this topological change. This example demonstrates how references can be *unified* across paths, and allows the user to find the point in both cases.

Finally, we can select the desired 2D areas in case (C) in such a way that they match the areas we would get when subtracting R2 and C2 from R1 and C1 on these two cases. We do this with contains, selecting every region that contains any point from R1, the outer rectangle. We note that there exist many queries to select the correct 2D areas, which can be more verbose than that displayed here, and which may behave differently on different topological configurations; rather than a specific unique global name for the regions in question.

5.2 Hypothesis 2

Lineage queries incur minimal performance overhead on top of geometric computation. The risk is that a robust scheme may require storing too much metadata or require too much computation in resolving references, causing CAD models not to respond interactively. We show this is not the case and provide a sense of relative performance on example benchmarks, some of which are shown in Figure 11. We underscore that lineage tracking itself is not a novel innovation - existing entity-matching systems also use lineage information as input to heuristic matching [Farjana and Han 2018; OnShape 2016], so we are concerned mainly with the time spent in computing reachability to answer queries.

We implement our system as an interpreter over the CAD program, with lineage relations modeled as a graph scoped to a single execution of a program, and each operation assigning lineage edges during its execution. In turn, this required us to implement our own CAD operations to appropriately assign lineage in accordance with the semantics over all possible execution paths within the operation. Queries are resolved using reachability between sets of elements on this graph, using a breadth-first search starting from the context set *S* and iterating upwards towards the elements passed as arguments to from. This makes the cost of query resolution independent of when a query is executed, as only edges that contribute to the lineage of elements in *S* can be searched, and this set is fixed after *S* is constructed. Resolution is worst-case O(nm), where *n* is the maximum number of geometric elements produced by any operation, and *m* is the number of operations visited by the query, in the worst case all operations in the program. In practice, *m* is often less than 10, and *n* rarely exceeds a thousand elements on the most complex models.

The final result is that the resulting query resolutions make up a small fraction of the total computation time, which becomes even smaller when more computationally-intensive CAD operations are performed. We benchmark our system compiled to JavaScript, executing on an AMD Ryzen 2700x processor, using Chrome 107. We note that the JavaScript JIT exhibits warm-up effects and intermittent latency spikes due to garbage collection, so we display the mean of 10 running times after 5 initial executions of a given program, all with the same parameters. The benchmarks are chosen to be representative of our real-world models (Section 5.3) and cover all referencing constructs. All benchmark programs, as well as a web-based version of our implementation to run them, can be found in the supplemental material. While we implement simple algebraic and short-circuiting optimizations on top of the base lineage semantics, on these examples the number of lineage vertices visited during reachability computations is relatively small (particularly per traversal). This is worst-case O(nm), where n is the maximum number of geometric elements produced by any operation, and m is the number of operations visited by the query, in the worst case all operations in the program. In practice, m is often < 10, and n rarely exceeds a thousand elements on the most complex models. Such programs can be run interactively in response to a slider dragging in an interface or in the core loop of a numerical solver, and the responsiveness is fast enough that further optimization at this scale proves not to be necessary for maintaining performance.

| Program | Ops | Lineage Verts | Query Time | Kernel Time | Traversals | Verts Visited |
|------------|-----|---------------|------------|-------------|------------|---------------|
| Clamp | 41 | 404 | 0.76ms | 4.90ms | 88 | 469 |
| Gear | 12 | 1497 | 0.38ms | 3.74ms | 42 | 129 |
| Table | 10 | 176 | 0.34ms | 1.68ms | 49 | 247 |
| Controller | 13 | 139 | 0.72ms | 2.02ms | 41 | 217 |
| Key | 6 | 144 | 0.20ms | 1.10ms | 28 | 145 |
| Divot | 12 | 216 | 0.58ms | 1.56ms | 48 | 411 |

5.3 Hypothesis 3

Our DSL is able to express programs of comparable complexity to those found in real-world CAD scenarios. Our language does not allow the user arbitrary Turing-complete constructs, unlike the state-based schemes described in Section 6.1. For example, the smoothness property precludes the user from defining a selection that exhibits discontinuities without topological change – e.g. referring to a face as the highest face in the model, as such a face might change as parameters change even without the topology changing. Precluding such issues creates notable benefits for reasoning and query robustness. While this may reduce the space of models that can be expressed in our system, we argue that this is not representative of realistic CAD models, as entity-matching schemes (Section 6.1) do not allow such selections either, and are by far the most popular modeling paradigm.

To proxy real-world CAD complexity, we turn to an existing dataset of CAD models and their corresponding programs provided by DeepCAD [Wu et al. 2021], which is in turn derived from the ABC dataset for geometric deep learning [Koch et al. 2019], containing ~1M CAD models built using OnShape. Most models are mechanical parts with sharp edges and well defined surfaces. The DeepCAD analysis finds that most (> 95%) of CAD command sequences (programs) are no longer than 40 (operations) and use less than 8 extrusions (analogous to *set-operations* in this paper). We note that due to difference between languages used, the number of operations should be seen as a loose estimate - but as the examples in Figure 2 show, even a few operations are enough to exhibit referencing failures.



Fig. 11. Example gallery of CAD models in our DSL displayed in the web interface.

We tested our system on 23 examples ranging from simple 2D polygon intersections to full 3D models. Our test cases execute a median of 10 operations, 3 reference queries, and creates geometry with a median of 4 faces and 37 segments. Our most complex models execute 62 operations, 10 reference queries, and create 206 geometric faces respectively.

We demonstrate our system on the more complex programs which are too large to display here in full. We include in Figure 11 a snapshot of these models and summary statistics, and include the full models and their programs in the supplemental material.

5.4 Limitations

The main technical limitation in our work as presented is *geometric*, rather than reference, expressivity: our referencing scheme as described disallows the inclusion *spline curves* in their full generality, because spline curves can intersect in an arbitrary number of points that cannot be reliably partitioned into static sets using the same method we have done for arcs, which can only intersect at most twice. As a result, this would cause the selectability property not to hold, as individual intersection points cannot be referred to. In the Autodesk Fusion 360 Gallery dataset [Willis et al. 2020], fewer than 5% of CAD models contain spline curves at all, and we infer that an even smaller proportion of such models require the ability to refer to multiple points on the intersection of two splines. In practice, this means that users cannot safely model highly curved objects, for example a helmet. As a mitigating strategy, a programmer can statically assert there is only a single point on the intersection of two splines and refer to it that way. Further work is required to derive a scheme that can robustly define these points across topologies.

5.5 Future Work

There exist additional directions for application and extension of programming languages techniques in the domain of CAD programs, made possible by a fully-specified referencing semantics:

- *Graphical Query Synthesis.* In the proposed language users must provide a query in textual program form with reference to prior elements, while in CAD systems the user is able to work completely graphically and refer to elements just by clicking on them. The synthesis question is whether a given graphical selection can be transformed into an unambiguous query with a user in the loop, without the user looking at the program. Mathur et al [Mathur et al. 2020] give a synthesis algorithm for constructing references in a state-based paradigm.
- *Feasible Space Analysis.* To verify high-level design properties (such as fabrication constraints) over a space of programmatically-generated designs requires analysis of the entire feasible space of the program, including the bounds specified by assertions and reference boundedness.
- *Program Transformations*. To optimize programs towards a specific desirable objective (e.g. the strongest bridge, the lightest shelf) it is useful to perform program transformations such as automatic differentiation. Analogously, finding input parameters to a CAD model that cause geometric elements to appear in a particular configuration can be done via probabilistic programming and inference techniques.

6 RELATED WORK

6.1 Referencing Schemes

The two primary referencing schemes used in practice are state-based queries and entity-matching.

6.1.1 State-Based Queries. In systems that expose scripting or procedural interfaces such as Cad-Query [Holdings 2019], Grasshopper [McNeel 2007], and Houdini [SideFX 1996], users must provide custom geometric logic to select an element from the collection; for example selecting the highest edge parallel to the X axis, or selecting the edge nearest to a separately-computed point in space.

This is the same interface exposed by low-level CAD kernel APIs such as OpenCASCADE [Open-Cascade 1999, 2022]. Users are allowed the full power of the host language, making optimization or guarantees on referencing difficult, and forcing the user to reason about how to handle topology changes and discontinuities with each constructed reference. We abstract such concerns into a simpler DSL.

6.1.2 Entity-Matching. User-facing CAD packages used widely in Mechanical Engineering such as OnShape [PTC 2015], SolidWorks [Systèmes 1995], CATIA [Systèmes 1982] and others allow users to specify references using a GUI on a concrete instantiation of the model; without providing any logic. This makes it easy to author a model initially, but comes with a trade-off of added difficulty when editing parameters, as the problem is ambiguous: users have not defined the reference on these new program execution paths. CAD tools heuristically construct a mapping from the topology where the reference was specified to the topology where the reference is being evaluated. This mapping incorporates local geometric and topological similarity, as well as some forms of lineage information [Cheon et al. 2012; Farjana and Han 2018], and is known as the *persistent naming* problem, i.e. that of finding a "name" for a topological element that is "persistent" across topologies, and has been the subject of much debate in the CAD domain.

The work most similar to ours is done by Bidarra and Bronsvoort [Bidarra and Bronsvoort 2000; Bidarra et al. 2005], who propose a paradigm of semantic feature modeling. They class reference failures as unintended "feature interactions" between models, and attempt to bound the valid ranges of parameters to each CAD operation in such a way to preclude undesirable topological interactions such splitting, merging, disconnection, and absorption of geometric elements. This results in a limited naming scheme that allows elements to be named only if they are persistent, requiring manual user intervention when ambiguous cases are detected. Our work significantly extends the referencing capabilities of this system, provides a formalization, and allows ambiguity to be resolved a priori by specifying a program.

We refer the reader to [Farjana and Han 2018] for a comprehensive historical survey of entity-matching methods.

6.2 CAD Editing

The challenges with robustly editing parametric CAD is decades old and well discussed in literature and by practitioners [Dorribo Camba et al. 2016; Yares 2013]. For this reason, some CAD systems have proposed to directly edit the geometry, ignoring the constructive CAD program partially (e.g., Siemen's Synchronous Technology, Rhino3D, and IronCAD) or completely (e.g. SpaceClaim, KeyCreator). While direct editing addresses the robustness concerns, precise global editing becomes challenging, particularly, maintaining geometric constraints and invariants [Alba 2018; Brunelli 2014; Strater 2016]. For these reasons, programmatic parametric CAD is still the most widespread approach in engineering design, and the challenges with robust editing are typically seen as unavoidable. Indeed, there are many resources for best modeling practices that train engineers to design models that are less likely to break [Herron 0202] as well as studies of "modeling methodologies" designed to avoid breakage in practice [Bodein et al. 2014; Gebhard 2013; Landers and Khurana 2004]. This vast literature points to deeper underlying semantic issues that have been plaguing programmers, who have been forced to resort to informal methodologies and strategies in lieu of more robust underlying tools.

6.3 CAD Languages

Domain-specific languages for CAD models have seen recent interest as symbolic, interpretable representations of 3D objects that are suitable for learning tasks. Prior work has focused on

translating fuzzy, physical models into programs [Nandi et al. 2018; Schulz et al. 2018], manipulating low-level programs to form higher-level, human-editable representations [Nandi et al. 2020], learning functions for model components [Jones et al. 2021a] and assembling those components together into programs [Jones et al. 2020], a basis for novel editing interfaces using program optimization [Cascaval et al. 2021; Gaillard et al. 2022], and as a basis for reasoning about digital fabrication pipelines [Tran O'Leary et al. 2021].

Languages are also growing in popularity in practical CAD tools, with OnShape allowing users to model custom elements in an embedded DSL [OnShape 2016], and tools such as Blender's Geometry Nodes [ble 2022], and Rhino3D's Grasshopper [McNeel 2007] enabling user-automation of CAD via procedural models that can generate many potential designs. All of these tools force the user to deal directly with the issue of establishing references when writing programs, bringing the complexity of reasoning about references from the system to the user-level.

7 CONCLUSION

We have presented a novel method for CAD referencing, addressing the ability to define stable references to topological elements in CAD models generated by programs. In turns, this aids the development of future editing and design optimization applications on top of CAD models that rely on a robust semantics in the underlying tool – the lack of which has been a thorn in the side of CAD users for decades. We claim that referencing is not just limited to geometry, and applies to other complex or dynamic data structures as well. Finally, we draw attention to future avenues of work intersecting Programming Languages and CAD models – notably the synthesis of queries from graphical inputs, and lifting unstructured CAD models to a reference-based program form both form interesting questions of program synthesis, and transforming CAD programs to verify high-level properties and assertions over a wide parameter space is a valuable application of verification. We hope to see additional work intersecting Programming Languages and CAD in the future.

ACKNOWLEDGMENTS

We thank the reviewers for their helpful feedback, and Maneesh Agrawala for insightful discussions as the work developed. This work was supported by NSF CCF-2219864, and has been additionally supported in part by the NSF awards ITE-2132318, CCF-2122950, ITE-2029457, ITE-1936731, CCF-1918027, IIS-1924435, NSF BSF-2020280, the Intel and NSF joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA NSF CCF-1723352), the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741-394324 AM01, a grant DARPA FA8750-16-2-0032, as well as gifts from Adobe, Amazon, Meta, Google, Intel, and Qualcomm.

ARTEFACT AVAILABILITY

The complete source code along with development instructions is available at https://github.com/ dcascaval/lineage-based-cad-referencing. Our implementation is written in Scala, compiled to Javascript to run client-side in a browser environment. A live hosted version containing our example programs and benchmarks can be found at https://dcascaval.github.io/lineage/.

REFERENCES

2022. Blender Geometry Nodes. https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/index.html. (Accessed on 10/11/2022).

Michael Alba. 2018. What's the Difference Between Parametric and Direct Modeling? https://www.engineering.com/ DesignSoftware/DesignSoftwareArticles/ArticleID/16587/Whats-the-Difference-Between-Parametric-and-Direct-

Modeling.aspx. (Accessed on 09/06/2019).

Rafael Bidarra and Willem F Bronsvoort. 2000. Semantic feature modelling. Computer-Aided Design 32, 3 (2000), 201–225.

- Rafael Bidarra, Paulos J Nyirenda, and Willem F Bronsvoort. 2005. A feature-based solution to the persistent naming problem. *Computer-Aided Design and Applications* 2, 1-4 (2005), 517–526.
- Yannick Bodein, Bertrand Rose, and Emmanuel Caillaud. 2014. Explicit reference modeling methodology in parametric CAD system. Computers in Industry 65, 1 (2014), 136–147.
- Mark Brunelli. 2014. Parametric vs. Direct Modeling | PTC. https://www.ptc.com/en/cad-software-blog/parametric-vs-direct-modeling-which-side-are-you-on. (Accessed on 09/08/2019).
- Dan Cascaval, Mira Shalah, Phillip Quinn, Rastislav Bodík, Maneesh Agrawala, and Adriana Schulz. 2021. Differentiable 3D CAD Programs for Bidirectional Editing. *CoRR* abs/2110.01182 (2021). arXiv:2110.01182 https://arxiv.org/abs/2110.01182
- Sang-Uk Cheon, Duhwan Mun, Soonhung Han, and Byung Chul Kim. 2012. Name matching method using topology merging and splitting history for exchange of feature-based CAD models. *Journal of mechanical science and technology* 26, 10 (2012), 3201–3212.
- Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last (*PLDI '16*). Association for Computing Machinery, New York, NY, USA, 341–354. https://doi.org/10.1145/2908080. 2908103
- Jorge Dorribo Camba, Manuel Contero, et al. 2016. Parametric CAD modeling: An analysis of strategies for design reusability. (2016).
- 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 Transactions on Graphics (TOG) 37, 6 (2018), 1–16.
- Shahjadi Hisan Farjana and Soonhung Han. 2018. Mechanisms of persistent identification of topological entities in CAD systems: A review. *Alexandria engineering journal* 57, 4 (2018), 2837–2849.
- Robert Bruce Findler and Matthew Flatt. 2006. Slideshow: functional presentations. *Journal of Functional Programming* 16, 4-5 (2006), 583–619.
- James D Foley. 1996. 12.7 Constructive Solid Geometry, Computer Graphics: Principles and Practice. Addison-Wesley Professional. 557–558 pages.
- Mathieu Gaillard, Vojtech Krs, Giorgio Gori, Radomir Mech, and Bedrich Benes. 2022. Automatic Differentiable Procedural Modeling. *Computer Graphics Forum* (2022). https://doi.org/10.1111/cgf.14475
- Richard Gebhard. 2013. A resilient modeling strategy. Technical Presentation, Solid Edge University (2013).
- Christian Hafner, Christian Schumacher, Espen Knoop, Thomas Auzinger, Bernd Bickel, and Moritz Bächer. 2019. X-CAD: optimizing CAD models with extended finite elements. ACM Transactions on Graphics (TOG) 38, 6 (2019), 1–15.
- Michael Herron. 20202. THE TOP 10 SECRETS CAD AND SOLIDWORKS USERS NEED TO KNOW. https://www.simplexitypd.com/blog/the-top-10-secrets-cad-and-solidworks-users-need-to-know. (Accessed on 11/10/2022).
- Parametric Products Intellectual Holdings. 2019. CadQuery Introduction. https://cadquery.readthedocs.io/en/latest/intro. html. Accessed: 2022-10-11.
- Benjamin Jones, Dalton Hildreth, Duowen Chen, Ilya Baran, Vladimir G. Kim, and Adriana Schulz. 2021b. AutoMate: A Dataset and Learning Approach for Automatic Mating of CAD Assemblies. ACM Transactions on Graphics 40, 6, Article 227 (dec 2021), 18 pages. https://doi.org/10.1145/3478513.3480562
- R Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. 2020. Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–20.
- R Kenny Jones, David Charatan, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. 2021a. ShapeMOD: macro operation discovery for 3D shape programs. ACM Transactions on Graphics (TOG) 40, 4 (2021), 1–16.
- Marius Kintel and Claire Wolf. 2010. OpenSCAD, The Programmers Solid 3D CAD Modeller. https://openscad.org/index.html. Accessed: 2023-03-17.
- Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. 2019. ABC: A Big CAD Model Dataset For Geometric Deep Learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).*
- Temel Kösa and Fatih Karakuş. 2010. Using dynamic geometry software Cabri 3D for teaching analytic geometry. Procedia -Social and Behavioral Sciences 2 (12 2010), 1385–1389. https://doi.org/10.1016/j.sbspro.2010.03.204
- Diane M Landers and Pravin Khurana. 2004. Horizontally-structured CAD/CAM modeling for virtual concurrent product and process design. US Patent 6,775,581.
- Aman Mathur, Marcus Pirron, and Damien Zufferey. 2020. Interactive programming for parametric cad. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 408–425.
- McNeel. 2007. Grasshopper3d, Algorithmic Modeling for Rhino. https://www.grasshopper3d.com/. Accessed: 2022-10-11.

- Elie Michel and Tamy Boubekeur. 2021. DAG Amendment for Inverse Control of Parametric Shapes. ACM Transactions on Graphics 40, 4 (2021), 173:1–173:14.
- Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional Programming for Compiling and Decompiling Computer-aided Design. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 99:1–99:31. https://doi.org/10.1145/3236794
- Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (London, UK) (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/3385412.3386012

OnShape. 2016. OFeatureScript. https://cad.onshape.com/FsDoc/. Accessed: 2022-10-11.

- OpenCascade. 1999. OpenCascade Technology (OCCT) Kernel. https://www.opencascade.com/open-cascade-technology/. Accessed: 2022-10-11.
- OpenCascade. 2022. Topology Explorer Class Reference. Retrieved October 11, 2022 from https://dev.opencascade.org/doc/ refman/html/class_top_exp___explorer.html
- PTC. 2011. PTC Creo. https://www.ptc.com/en/products/creo. Accessed: 2022-10-11.
- PTC. 2015. OnShape. https://www.onshape.com/en/. Accessed: 2022-10-11.
- Adriana Schulz, Harrison Wang, Eitan Grinspun, Justin Solomon, and Wojciech Matusik. 2018. Interactive exploration of design trade-offs. ACM Transactions on Graphics (TOG) 37, 4 (2018), 1–14.
- Adriana Schulz, Jie Xu, Bo Zhu, Changxi Zheng, Eitan Grinspun, and Wojciech Matusik. July 2017. Interactive Design Space Exploration and Optimization for CAD Models. *ACM Transactions on Graphics* 36, 4 (July 2017).
- SideFX. 1996. Houdini. https://www.sidefx.com/products/houdini/. Accessed: 2022-10-11.
- Siemens. 1973. Siemens NX. https://www.plm.automation.siemens.com/global/en/products/nx/. Accessed: 2022-10-11.
- Siemens. 1985. Parasolid CAD Kernel. https://www.plm.automation.siemens.com/global/en/products/plm-components/ parasolid.html. Accessed: 2022-05-19.
- Jeff Strater. 2016. Solved: Re: Direct vs History based modelling Page 2 Autodesk Community- Fusion 360. https://forums. autodesk.com/t5/fusion-360-design-validate/direct-vs-history-based-modelling/m-p/6717750#M84483. (Accessed on 09/06/2019).
- Dassault Systèmes. 1982. CATIA. https://www.3ds.com/products-services/catia/. Accessed: 2022-10-11.
- Dassault Systèmes. 1995. SolidWorks. https://www.solidworks.com/. Accessed: 2022-10-11.
- Jasper Tran O'Leary, Chandrakana Nandi, Khang Lee, and Nadya Peek. 2021. Taxon: a Language for Formal Reasoning with Digital Fabrication Machines. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 691–709.
- Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. 2020. Fusion 360 Gallery: A Dataset and Environment for Programmatic CAD Reconstruction. arXiv:2010.02392 [cs] (Oct. 2020). http://arxiv.org/abs/2010.02392 arXiv: 2010.02392.
- Rundi Wu, Chang Xiao, and Changxi Zheng. 2021. Deepcad: A deep generative network for computer-aided design models. In Proceedings of the IEEE/CVF International Conference on Computer Vision. 6772–6782.
- Xianghao Xu, Wenzhe Peng, Chin-Yi Cheng, Karl D. D. Willis, and Daniel Ritchie. 2021. Inferring CAD Modeling Sequences Using Zone Graphs. Technical Report arXiv:2104.03900. arXiv. http://arxiv.org/abs/2104.03900 arXiv:2104.03900 [cs] type: article.
- Evan Yares. 2013. The failed promise of parametric CAD part 1: From the beginning. https://www.3dcadworld.com/the-failed-promise-of-parametric-cad/. (Accessed on 09/06/2019).

Received 2022-11-10; accepted 2023-03-31