Understanding and Supporting Debugging Workflows in CAD

Felix Hähnlein
fhahnlei@cs.washington.edu
University of Washington
USA

Gilbert Bernstein
gilbo@cs.washington.edu
University of Washington
USA

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

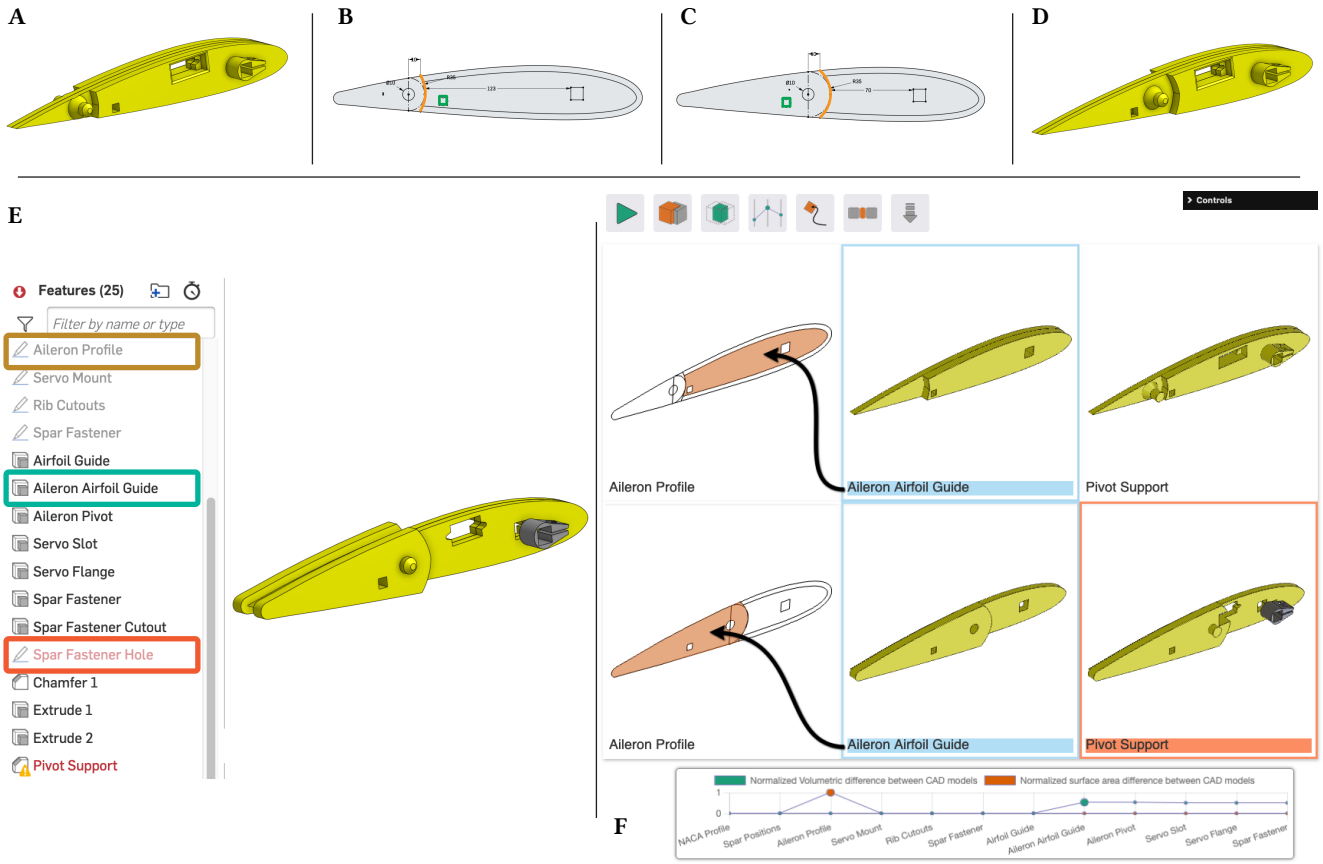

Figure 1: Parametric edits in CAD can lead to errors. The user is editing the current model (A) by modifying the sketch `Aileron Profile` (B). They intend to move the orange curve to the right side of the green square (C) to obtain the model show in (D). However, they are confronted with error messages and a broken model (E). Whereas the first error message appears in `Spar Fastener Hole`, the actual error happens in `Aileron Airfoil Guide`. (F) Our debugger, DeCAD, makes readily available the information needed to locate and understand the problem. The top and bottom rows of the visualizer show the CAD model before and after the edit, respectively. `Aileron Airfoil Guide` is extruding the wrong face after the edit.

ABSTRACT

One of the core promises of parametric Computer-Aided Design (CAD) is that users can easily edit their model at any point in time. However, due to the ambiguity of changing references to intermediate, updated geometry, parametric edits can lead to reference errors which are difficult to fix in practice. We claim that debugging reference errors remains challenging because CAD systems do not provide users with tools to understand where the error happened and how to fix it. To address these challenges, we prototype a graphical debugging tool, DeCAD, which helps comparing CAD model states both across operations and across edits. In a qualitative lab study, we use DeCAD as a probe to understand specific challenges that users face and what workflows they employ to overcome them.

UIST '24, October 13–16, 2024, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-0628-8/24/10
https://doi.org/10.1145/3654777.3676353

We conclude with design implications for future debugging tool developers.

ACM Reference Format:
Felix Hähnlein, Gilbert Bernstein, and Adriana Schulz. 2024. Understanding and Supporting Debugging Workflows in CAD. In *The 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24), October 13–16, 2024, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3654777.3676353

1 INTRODUCTION

Parametric Computer-Aided Design (CAD) software is the most widely used tool for manufacturing-oriented design. Such systems (e.g. Onshape, Solidworks, Fusion, AutoCAD) essentially represent 3D geometry as a sequence of operations that build the geometry bottom up. As shown in Figure 2, these operations take as input numerical parameters (e.g. the length of a fillet) and *references* to existing geometry (e.g., the edge that should be chamfered). While not commonly recognized as such, under this representation, a CAD model is a program.

One of the most important benefits of this modeling paradigm is that it supports further modeling iterations through parameter editing. In practice, however, editing parameters of one operation often causes subsequent operations to fail.

Most commonly, these errors are the result of a failure to resolve references. During modeling, users create references to intermediate *geometric entities* (faces, edges, and vertices) by clicking on them in the CAD GUI. CAD systems have mechanisms for recognizing these references and finding appropriate matches when the topology changes, see Fig 2. While the heuristics used to drive these algorithms are often successful, they can sometimes lead to unintended reference changes. In long, real-world modeling sequences, the effects of reference errors can propagate and cause downstream operations to fail, see Fig. 1.

It is well understood that while better reference matching algorithms can minimize the number of errors in practice, ultimately such errors cannot be completely avoided because the reference matching problem is inherently ambiguous—the act of clicking over a geometric entity does not fully determine how the reference should behave when the topology changes [12] . In this work, we therefore propose to support users in *correcting these errors when they appear.* Our key insight is that, since CAD models are programs, when the user is confronted with reference errors, they are actually confronted with a broken program, which they have to *debug*. Therefore, the main research question of this work is: What should a debugger for CAD look like?

Importantly, while CAD systems will highlight the operations that failed, reference errors are silent and may have occurred in any operation preceding the failed one. For example, in Fig. 1, the reference error happens in `Aileron Airfoil Guide` whereas the first failed CAD operation was `Spar Fastener Hole`. Therefore, the debugger should help the user answering two questions:

- Where did a reference error occur?
- How do we fix it?

Our goal is to work towards a debugger for CAD by investigating what specific challenges users face and what workflows they gravitate towards to overcome them.

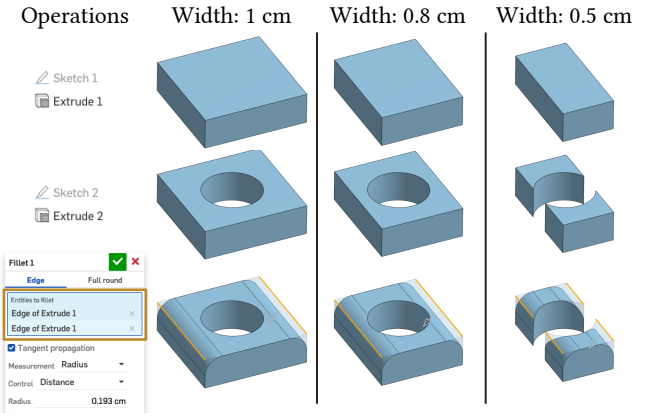

Figure 2: CAD operations (left) take as input references to intermediate geometry. `Fillet1` (bottom left) rounds two top edges of the cuboid (highlighted in orange). Editing the width of the cuboid to 0.8 cm seamlessly updates downstream operations. However, reducing the width to 0.5cm changes the input topology to `Fillet1` and the CAD system has to guess which edges to select. Here, it chooses all four edges.

Specifically, based on informal discussions with experts, real-world user interaction data and our own experience, we provide an analysis of current UI features, debugging challenges and workflows. Our analysis concludes that users have to simultaneously discover where topological changes happen and where an unintended reference change happens. This is challenging with CAD systems which are primarily designed for modeling, but not for debugging.

Informed by our analysis, we design a prototype debugger, called DeCAD. Implemented as a plugin to an industry standard CAD system, Onshape [2], we develop three key features: (1) a volume difference chart for a high-level summary of per-operation geometric change; (2) a two dimensional panel view of the CAD model to compare a model to its original design intent; (3) per-operation boolean difference, overlay visualizations and reference arrows to support per-operation change discovery.

To examine if users face our hypothesized challenges or if they face additional challenges and to understand their debugging workflows, we conduct a qualitative lab study with CAD experts. In the study, users are asked to resolve erroneous edits from real-world CAD models.

In our lab study, we use DeCAD as a probe to understand if we can tackle hypothesized debugging challenges and what new workflows we can enable. Based on our findings, participants managed to integrate DeCAD into their debugging workflow. The proposed features address our initial set of challenges but they also introduce new challenges. We synthesize our findings into a set of design guidelines for future tool-builders.

In summary, our contributions are:

- An analysis of challenges that users face when dealing with reference errors (Sec. 3).
- A publicly available prototype debugger, DeCAD (Sec. 4).

- Findings from our lab study with CAD experts and design implications for future tool builders (Sec. 5).

2 RELATED WORK

Our research lies at the intersection of 3D shape editing, references management in CAD systems, methods for visualizing programs and tools for debugging support.

2.1 Shape editing

Significant prior work focused on editing geometry directly without the constraints of parametric programs by optimizing an energy functional [29], modifying proxy models [34] or by analyzing perceptual shape properties [17]. Within parametric shape editing, prior work studied the challenges of manipulating procedural models [49]. Approaches to tackle these challenges include intermediate editing structures [10, 16] and leveraging prompt-driven models to modify parameters [18]. Bidirectional editing methods prioritize direct interaction with geometry and they have been studied for SVG [25], CSG [21] and parametric CAD [13, 44]. Instead of proposing a different method to perform an edit, our work seeks to augment the standard editing interface with tools to inspect the modeling sequence and to understand the effects of the edit.

2.2 References in CAD

Modern, GUI-based CAD systems predict how references should match to new entities after a topological change provoked by an edit. This process is called *entity matching*. Prior work has focused on improving entity matching methods, for example by using program synthesis to automatically generate reference queries [40]. Some textual CAD programming languages, such as CADQuery [1] expose reference queries as a programming primitive. While such advances can minimize the number of references errors in practice, these cannot be avoided because the problem is underconstrained.

To address this fundamental limitation, Cascaval et al. have proposed a domain-specific language (DSL) for CAD that allows users to unambiguously specify how references propagate with topological changes [12]. Although this approach offers formal guarantees that such reference errors will be avoided, it requires significantly more programming effort from the user upfront. Instead of merely clicking on a geometric entity in a user interface, users must explicitly program the reference construction.

Our work takes the position that entity matching is a good default mode since it requires minimal effort from the user *most of the time* and professional users are used to it. We acknowledge that the propagation of an edit is inherently ambiguous and additional information from the user is necessary. Our work aims at providing a graphical interaction mode for allowing users to easily understand what information is missing so that they can resolve the error.

2.3 Visualizing programs and program histories

Program visualizations have been widely studied to support students and developers to reason about their code [26, 50]. One approach consists in creating situated variable visualizations next to the original code [23, 36]. Omnicode lets the developer visualize the entire history of variable values via a matrix of scatterplots [31]. LaToza and Myers focus on visualizing the call graph to support understanding the control flow of a program [35]. Theseus also focuses on the function call behavior of programs and visualizes the call count of functions over the code [37]. BiFröst and WiFröst focus on supporting makers of IoT devices to visualize the runtime behavior of embedded code and circuits [42, 43].

Our work is aligned with the aforementioned prior work to reduce excessive cognitive load [45] by visualizing program states and the control flow, applied to 3D CAD.

We also relate to work which focuses on visualizing program histories, which has been studied in the context of manipulating editing histories in interactive data visualizations [24]. NodeGit computes the difference between two graphically created, parametric programs [47], which is a challenging problem. Instead of visualizing the entire history of edits or computing program differences automatically, we focus on providing designers with tools to locally discover changes incurred by their edit.

2.4 Debugging tools

Debugging is a challenging and time-consuming activity in software development which has motivated prior work to analyze debugging behavior, time spent on debugging activities and debugging challenges [8, 9, 20, 33, 53]. McCauley et al. provide a survey of the different causes of bugs and what kind of knowledge developers seek out during debugging [41].

To address debugging challenges, prior research has proposed methods for log-based debugging [30], breakpoint-based debugging [51], omniscient debugging [46] and interrogative debugging [32].

Debugging has also been adopted by specific domains, such as electronic circuit design [38], database queries [19], data wrangling [48], reactive programming [27] and multiverse analysis [22].

Our research aligns with prior work on domain-specific debugging tools, which we apply to 3D CAD. Similar to the goals of other tools, we want to reduce commonly observed context switches to reduce user's cognitive load [7, 32].

3 ANALYSIS OF ERRORS AND CHALLENGES

The goal of this section is to analyze how users currently deal with reference errors in CAD to inform both the design of our prototype debugger and our lab study.

First, what exactly do we mean by reference errors? We are assuming the following scenario. The user performs an edit on an operation o_{edit}. Due to this edit, some subsequent operation o_{topo} will be the first operation to now produce a different topology w.r.t. before the edit. Starting from o_{topo}, all subsequent operations are now topologically different. However, a difference in topology is not necessarily unexpected or undesirable. Then, due to the topology change, some operation o_{ref} will reassign its references in an unintended way. We say that o_{ref} has a reference error. Due to the reference error, o_{ref} now generates unintended geometry, but it does not fail, i.e., the geometry kernel reruns successfully. We say that o_{ref} exhibits a geometric error. The geometric error will change the input for downstream operations, of which o_{fail} is the first downstream operation failing to regenerate and throwing an error.

The abstracted operation sequence is the following:
$[..., o_{edit}, ..., o_{topo}, ..., o_{ref}, ..., o_{fail}, ...]$.

Importantly, a change in topology is not necessarily wrong. Caused by the reference error, it is the geometric error which derails downstream operations. The challenge is to locate o_{ref} among all the changes occurring between o_{edit} and o_{fail}.

Next, we qualitatively analyze the features provided by GUI-based CAD systems to find o_{ref}, Sec. 3.1. Then, we quantitatively analyze a dataset of reference errors and debugging traces, Sec. 3.2. Lastly, we hypothesize debugging challenges that users face in current CAD systems, 3.3.

3.1 GUI features in CAD systems

When a user encounters a reference error, the user interface will look similar to Fig. 1 (bottom left). Inspecting a failed operation o_{fail} (marked in red) will show an error message, ranging from more specific messages, e.g. two entities are not intersecting anymore, to more generic messages which simply indicate that the operation could not regenerate. These error messages will be specific to the failed operation, which is the visible symptom, but not the root cause of the problem, they are not indicating the reference error.

The geometric change in Fig. 1 (bottom left) is easily visible, the initial part has been split in two and they are assigned different colors. However, especially in a multi-part CAD model and depending on the current 3D view, geometric changes can be challenging to discover.

For debugging, users need to gather information about both the program behavior and about the geometric behavior of their model. For this discussion and throughout the rest of the paper, we will talk about the different *model states* of a CAD model. A CAD model changes its model state if the program has been edited or if the last executing operation has been changed. The main 3D modeling view of a CAD system shows one model state at a time.

Features to navigate the program. CAD systems have developed several UI features to explore the operation sequence, i.e., the program.

There are three main interactions which, starting from an operation, give the user information about the operation's geometric behavior, see Fig. 3. For example, users can inspect an operation by entering an edit mode, see Fig. 3 (middle), which changes the visualized model state to the selected operation.

Users can also inspect operation dependencies, which opens a sublist view of all operations which created or modified the operation's input entities. Note that in Solidworks[5], the CAD system closest to Onshape, users create a tree of operations, which reifies operation dependencies and exposes them by default.

Lastly, some CAD systems have version control systems (VCS), similar to VCS used in software engineering [14]. It allows users to overlay geometry from different, timestamped versions of the CAD model, which are always executed until the last operation.

Features to navigate the geometry. CAD systems propose standard visualization tools, such as hiding different parts, curves and sketches and rendering only wireframe geometry. Users can also create section views, which are invisible 3D planes to create a cut in the geometry and hide the part closest to the user, creating a visualization similar to section views found in engineering drawings [11].

Features to navigate the program and the geometry. CAD systems also provide features to interact between the two domains. Hovering over an operation highlights which geometry was created or modified, and vice-versa, clicking on a geometric entity will highlight the operation(s) which created it. Users can undo and redo actions which they performed in the GUI. This action history works linearly and mixes program actions and geometry actions. As a consequence, if a user wants to undo an edit they have to undo all intermediate actions that they have performed since the edit, including non-programmatic actions like hiding 3D parts.

Summary. We observe that CAD systems provide features for inspecting the different operations of the CAD program and for inspecting 3D geometry. These features have been designed for working with a single model state.

3.2 Creating and analyzing a CAD error dataset

In this section, we create and analyze a dataset of reference errors to gain further insight into how reference errors occur in the real world and how users overcome them. This dataset was also used to find examples for our lab study. The data originates from Onshape's public documents, which comes from users who have agreed to make their document publicly available. These documents contain a *modeling history*, which is a log of changes made to either the CAD program or to the visibility of geometry. The modeling history does not contain more detailed UI traces such as the two first temporary model state changes from Fig. 3.

Onshape provided us with 200 high-quality CAD documents (featuring a high number of the *variable* operations) which result in 1768 modeling histories. Out of all edits performed on a CAD model, 5123 of them cause a subsequent operation to fail for the first time, which is a total of 3% of all edits. In total, 75% of error messages are directly related to reference problems. Of the remaining, 8% may hide reference errors.

Debugging segments. We are interested in segments of the modeling history where a reference error occurs and the user manages to fix it. We call them *debugging segments*. Formally, we define a debugging segment by a sequence of successive modeling history entries which starts with a CAD model without any errors, followed by an edit that causes one or multiple operations to break and which ends without any errors. Additionally, we impose that (1) the CAD program should have the same operations at the start and at the end of the segment and that (2) the segment cannot contain a sufficiently high number of undo actions to remove the initial edit. In total, our dataset contains 3243 debugging segments.

We conceptualize debugging reference errors as a searching process between an edited operation o_{edit} and a failed operation o_{fail}, see Sec. 3.1. We define the number of operations between these two operations as the *error range*, i.e., the length of the operation sequence which users have to parse. The distribution of error ranges in our dataset is a long-tail distribution with many short error ranges where o_{fail} occurs only a couple of operations after o_{edit}. Similarly, if we look at how long users took for debugging an error, we can see a similar distribution where the median time is under a minute to solve an error, see Fig. 4. However, looking at segments with an error range higher than 5, the median debugging

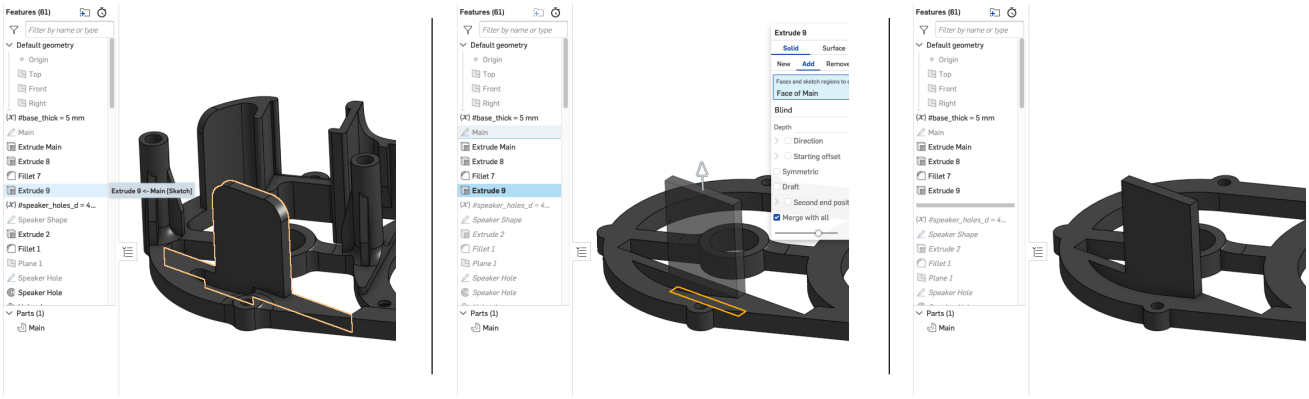

Figure 3: CAD systems provide different interactions to inspect operations. Left: Hovering over an operation highlights geometry created by the operation. Middle: Editing an operation shows the 3D model only until the selected operation and its created geometry. Right: Moving the *rollback bar* to an operation permanently stops the execution of any subsequent operation.

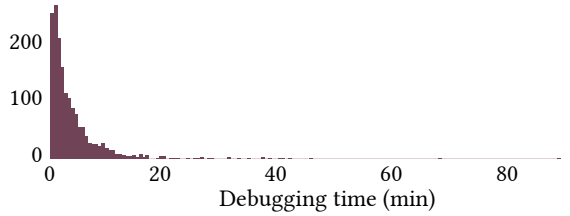

Figure 4: Distribution of debugging times in our dataset. While many errors are easy to fix, 10% of the debugging segments take longer than 4 minutes to debug.

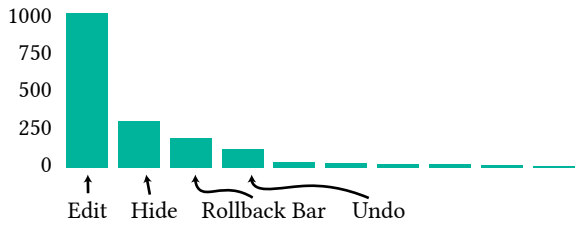

Figure 5: Distribution of actions taken during debugging. The four most common actions are: `Edit` an operation; `Hide` geometry; modify the `Rollback Bar` (see Fig. 3); `Undo` an action.

time increases to 4 minutes. This confirms that the current CAD features already work well for close-by errors but also that debugging becomes increasingly more difficult with growing program complexity. Note that our dataset contains only errors which users actually managed to solve, which means that we might not capture many challenging errors.

How do users tackled errors in our dataset? Counting the number of most used operations, see Fig. 5, we can see that a common strategy performed to tackle an erroneous edit is to perform more edits. Users also frequently toggle on and off the visibility of geometry (`Hide` action) and they change the model state by changing the last execution of the program (called `Rollback Bar` in Onshape). On fourth place, users use both the `Undo` and `Redo` action. This is aligned with the navigation features listed in Sec. 3.1.

3.3 Hypothesized challenges and conceptual model

From these observations, we hypothesize four fundamental challenges in debugging CAD reference errors:

Challenge 1: Geometric Complexity To understand reference errors, users need to identify intended topological changes and unintended geometric errors. Complex geometry can make it tedious to perform these evaluations and even lead to overlooking changes.
Challenge 2: Program Complexity Users need to analyze each operation in detail to identify where the error occurred. For longer programs, this becomes more challenging.
Challenge 3: Analyzing Multiple States Since references are broken after an edit, identifying them involves understanding how the model changed after the edit. This in turn involves understanding the model in two states (before and after the edit) and comparing them.
Challenge 4: Reference Dependencies Between Operations Since references point to geometric entities created at any point in the program, discovering reference errors and resolving them requires analyzing the operations in context and understanding the dependencies across different operations.

Inspired by these four challenges, we hypothesize a conceptual model that represents the user's investigative process as they search for an unintended reference change within the CAD program. This model, illustrated in Fig. 6, features a two-dimensional exploration. The first dimension represents movement between operations. Similar to lines in a program, the operation structure of a CAD model has to be periodically rediscovered by the user (Challenge 2). The second dimension involves transitioning between two states: before and after the edit (Challenge 3). Each point in this 2D space has geometric information that must be inspected (Challenge 1). Finally, points in this 2D space must be analyzed in context to extract reference dependencies (Challenge 4).

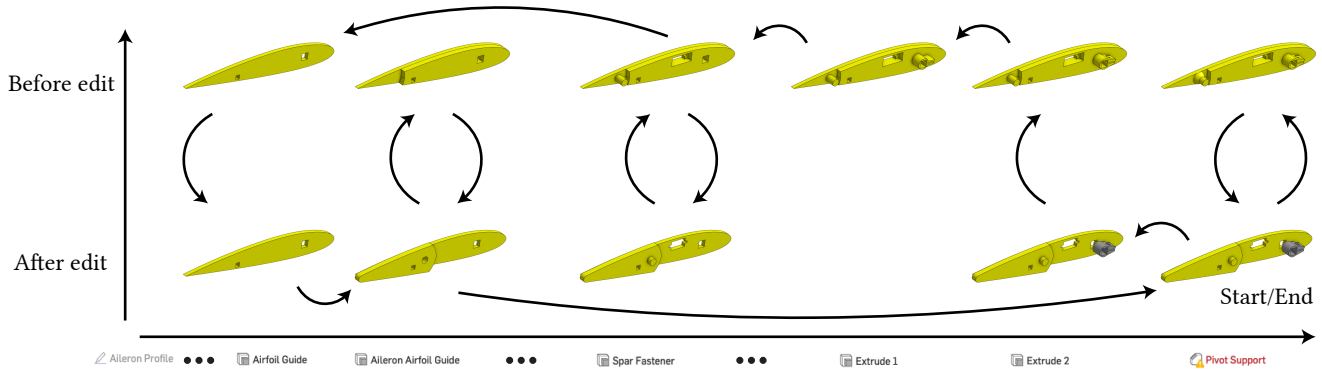

Figure 6: Our conceptual model for debugging reference errors. Users need to understand the programmatic modeling sequence (horizontal dimension) and the differences before and after the edit (vertical dimension). Since CAD systems present a single view at a time, users often create a mental map of these model states by interactively navigating this 2D space (arrows). For example, users will use undos and the rollback bar to navigate the vertical and horizontal dimensions, respectively.

4 PROTOTYPE: DECAD

Our goal is to understand challenges and workflows involved in debugging reference errors and how to support them. For this purpose, we created a DeCAD, a prototype debugger for CAD. We will use DeCAD as a probe for our lab study and to inform future tool designers.

The following design goals are directly derived from the challenges listed in Sec. 3.3:

DG1. *Facilitate the collection of per-operation information and summarize it succinctly.* After performing an edit, the user should be assisted in the search of the operation featuring an unintended reference change. The user should be able to quickly pinpoint operations worth investigating. Searching for this operation requires efficiently collecting per-operation information.

DG2. *The tool should help the user to effortlessly compare multiple operations in detail.* During debugging, users need to understand how an operation sequence works and how it changed after the edit. The tool should support this two-dimensional exploration problem.

DG3. *A debugging tool for CAD should support discovering change both in geometry and in references.* Confronted with unintended entity matching results, users need to discover arbitrary changes in references and geometry, made by the CAD system. CAD models can be complex and change blindness and out-of-view parts of interest should be taken into account.

Based on the three design goals, we have implemented three groups of features in DeCAD: (**DG1**) a volume difference chart; (**DG2**) a two-dimensional view of the CAD operations; (**DG3**) per-operation shape comparison features between edits and reference arrows. Each feature will be explained in the following sections.

4.1 Volume difference chart

When confronted with a reference error, one of the main challenges that a user is facing is to find out which operation is subject to an unintended reference change. With current CAD systems, the user can only inspect operations one at a time to gather information about changes w.r.t. the edit.

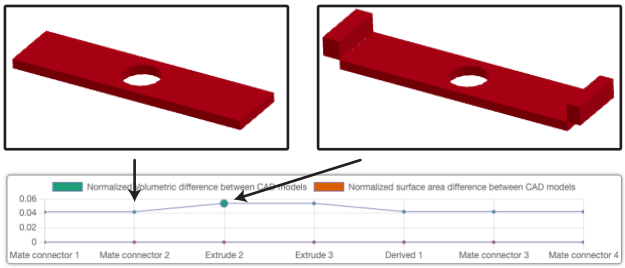

Figure 7: The volume difference chart presents scalar information about geometrically divergent operations without the need to inspect each operation at a time. The top row shows what this geometric difference corresponds to.

Instead of the user executing this repetitive task, we propose a feature to provide a succinct, scalar answer *for all operations* to a question which can be proceduralized.

More specifically, our idea is to investigate change in geometric behavior of an operation after an edit. We ask the following question: what is the boolean difference at operation o_i between the model's geometry after the edit and before the edit? Mathematically, this question corresponds to Eq. 1 :

$$Diff(o_i) = Vol(M_{\text{after}}(o_i) - M_{\text{before}}(o_i)) \quad (1)$$

where Vol is the scalar volume function, $M_{\text{after}}(o_i)$, $M_{\text{before}}(o_i)$ is the CAD model's geometry at operation o_i after the edit and before the edit, respectively. We visualize these values in an operation timeline chart, see Fig. 7. We also highlight operations with an increasing difference w.r.t. to the previous operation with a larger circle.

The intended use of this feature is to locate operations which are worth inspecting in more detail, based on the divergence in geometric behavior. For example, it is often worth inspecting the first operation which diverges geometrically after the edited operation.

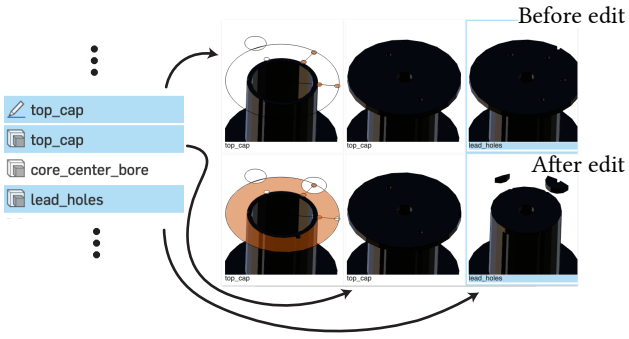

Figure 8: Using the two-dimensional view feature, the user can select operations which they want to compare.

Note that a diverging operation might be different from a failed operation, see Sec. 3.1.

4.2 Two-dimensional view

To support the comparison between multiple model states, we propose a two-dimensional panel view feature. To build the visualization, the user selects operations in the standard CAD interface and then clicks on the start button, see Fig. 8. This will unroll the model states for the selected operations in a two-dimensional grid structure, where the top row visualizes per-operation model states *before* the edit and the bottom row visualizes per-operation model states after the edit. With multiple model states unrolled in this view, the user can avoid frequent context switches to compare them.

Note that visual programs are often represented with node graphs, e.g. Adobe Substance. However, graph representations take up more screen space to account for operation dependencies. Inspired by [24], we adopt a comic-strip layout because it is the most economical layout in terms of screen real estate. This choice also stays close to the list-like representation of the host CAD system, where the operations of a model are represented as list of operations, hiding their dependency structure away from the user.

Each panel is a 3D scene, that the user can interact with. The typical 3D camera controls, i.e., zoom, pan and rotation are synchronized between all panels. Thanks to this, when the user is looking in one panel at a particular part of the 3D geometry, other panels are showing what this part of the geometry looks like at different model states. In practice, we use a single camera which is shared among all 3D scenes. Similarly, the user can select parts to show and hide, using a similar interaction as in the host interface. A selected part will be hidden among all operations in the same row. However, part identifiers before and after the edit are not necessarily the same and we therefore cannot synchronize this feature between the two rows.

Hovering over an operation panel will highlight input entities in the previous panel in which they appeared last. For example, a selected sketch face will be highlighted in the sketch operation which created the face. This highlight can be made permanent by clicking on an operation panel. The highlighting feature is synchronized among columns to facilitate comparison before and after the edit.

Figure 9: Instead of comparing side-by-side geometry (left), the user can overlay geometry and change their opacity (middle) or they can visualize the boolean difference (right).

4.3 Comparison features

We want to support the discovery of change, both in geometry and in references. The two-dimensional panel view already provides a means of comparing geometry visually by synchronizing 3D model states before and after the edit. The model before the edit will be shown in the top panel and the model after the edit will be shown in the bottom panel, see Fig. 9. The user can use this stacked view to compare geometry and highlighted input entities.

However, for more complex models, we offer additional features. First, the user can overlay the geometry from the top row onto the bottom row. The transparency of the overlayed geometry can be modified if needed, see Fig. 9 (middle). This feature is also useful to recontextualize geometry from both rows after a major change.

Second, the user can directly visualize the boolean difference in geometry between the two rows. More specifically, we subtract the geometry from before the edit from the geometry after the edit. The resulting geometry will be shown in red. We also perform the boolean difference in the opposite direction to obtain the geometry which has been *added* after the edit. This geometry will be shown in green.

Third, we provide reference arrows to support the discovery of changing references. Reference arrows point from an operation to its input entities, see Fig. 10. They are implemented as leader lines, which point from an operation panel to an entity from a previous operation, they point from right to left. We fix the starting point of each leader line on the left side of the operation label. The anchor point of each leader line, i.e. its end point [52], points to the projected area of a 3D entity. We optimize for the best anchor point, which is a challenging problem [15]. We simplify the problem by considering each leader line separately and by adopting a simplified notion of saliency to choose an anchor point. Inspired by [15], we want to select the 2D point which is furthest away from the boundary of its region and which is closest the right side of the panel. While the first criteria favors points which are local centroids and far away from thin features, the second criteria accounts for the fact that leader lines come in from latter operations, i.e., from the right side of the panel. More specifically, we maximize Eq.2.

$$saliency((x,y)) = dist_{silhouette}((x,y)) + \frac{y}{w_{panel}} \quad (2)$$

To improve runtime behavior, we solve the optimization problem only if the view has been modified by the user and only every second (and not every frame), which we found to work well in practice. While recomputing the most salient point, we lift the most salient 2D point back to its 3D entity and define this salient 3D

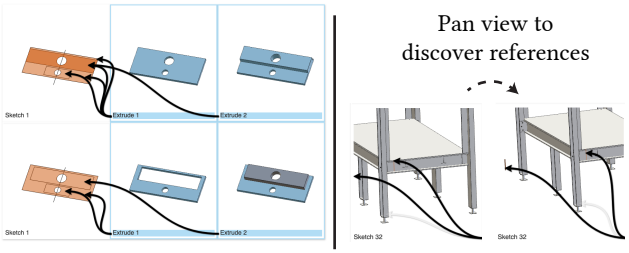

Figure 10: The user can activate reference arrows to show which entities have been used by which operation (left). Reference arrows are also useful to discover out-of-view entities (right).

point as the anchor point of the leader line, whose position is being updated every frame. Lastly, we only compute salient anchor points for currently highlighted entities.

Leader lines for non-highlighted entities are rendered transparently to not obstruct the view unnecessarily, yet they are not invisible to be discoverable. Leader lines can be toggled on or off.

Reference arrows help the user to distinguish which entities have been used by which operation and to emphasize change in use of entities, see Fig. 10 (left). They are also helpful to discover referenced entities which might be out of view from a particular 3D viewpoint. They provide the user with an incentive to discover different parts of the 3D scene.

4.4 Implementation

We implemented DeCAD as an extension to Onshape. Extensions are websites exposed through an *iframe* in the CAD system. Our website is implemented in html/css/javascript, using three.js [6] for 3D rendering. The website exchanges messages with a local python server which performs REST API calls to download program and geometry data from Onshape. The python server also performs geometric computation with a local Parasolid [3] version, which is the geometry kernel used by Onshape.

For the volume difference chart and for the two-dimensional view feature, we require intermediate geometry for two different versions of the CAD program. While CAD systems often cache intermediate geometry for the current program, for example to allow users to quickly inspect intermediate geometry, they do not keep geometry from previous versions. Similarly, the boolean computation for the volume difference chart takes additional time. We emphasize that these are not conceptual limitations for the proposed features. For the lab study, we download necessary geometry in advance and pre-compute boolean differences.

5 LAB STUDY

Using debugging as a metaphor for how users fix reference errors in CAD, we conduct a qualitative lab study. The goal of the lab study is to observe challenges that users face in practice, and what debugging workflows they gravitate towards to overcome them. Additionally, we want to understand the effects of a debugging-specific intervention in an already existing CAD system via a tangible tool, DeCAD, and to advise design implications for future tool builders.

Participant ID	P1	P2	P3	P4	P5	P6
CAD systems	O, S	O, S, I	O	O, S	S	O, S, I, F
Years of experience	> 5	> 5	< 1	1-3	> 5	> 5

Table 1: CAD systems: O: Onshape, S: Solidworks, I: Inventor, F: Fusion360.

The examples used in the study are challenging cases from the dataset created in Sec. 3.2.

5.1 Research Questions and Methods

The lab study was designed to investigate the following research questions:

RQ1 - Challenges What challenges do CAD users face when debugging reference errors? Do these match our hypothesized challenges? What additional challenges do they face?
RQ2 - Workflows What debugging workflows do users adopt? In what ways are they different from our hypothesized workflow?
RQ3 - Tool How does our tool address challenges faced by CAD users? What new workflows does DeCAD enable?

Study protocol. The study was conducted in a lab using a MacBook Pro on a 23-inch monitor, a keyboard and a mouse. The study itself was structured in 3 phases: a tutorial phase, five debugging tasks and two follow-up questions.

The tutorial phase began with a video explanation of DeCAD in the context of an erroneous edit. To familiarize participants with our tool, they were asked to use DeCAD themselves on the same model to debug it.

For the debugging task phase, participants could ask questions about Onshape, the tool or the task itself, but they were not given any hints about the source of the reference error. Each debugging task started with the modeling context in the form of an image illustrating either the real-world inspiration for the model or the use of the model in a larger assembly. Then followed a video with an edit on the model, leading to a reference error. Next, participants were asked to open an Onshape document containing the working model, and to replicate the edit themselves. Next, participants were asked to fix the model within 10 minutes. More specifically, fixing the model was framed as to *"correct all errors without removing or adding features and without changing the edit"*. Framing the task in this way prevented participants from any major remodeling, but it was still open ended enough to allow for multiple solutions.

Participants could always use all the tools available within Onshape, and for some tasks they were asked to also use DeCAD. We randomly assigned which participants could use DeCAD, so that we would observe participants with and without the tool on the same task. After the time-limit or when they finished debugging the model, we asked them what went wrong in the model and how they fix it.

In the last phase of the study, we asked participants what aspects of DeCAD they found the most useful and how they would improve the tool. In this part, participants could give feedback and engage in an open discussion around the errors and the tool.

The study lasted approximately one hour and participants were compensated with a 30$ Amazon gift card. With the consent of the participants, we recorded their audio and screen and later transcribed the sessions.

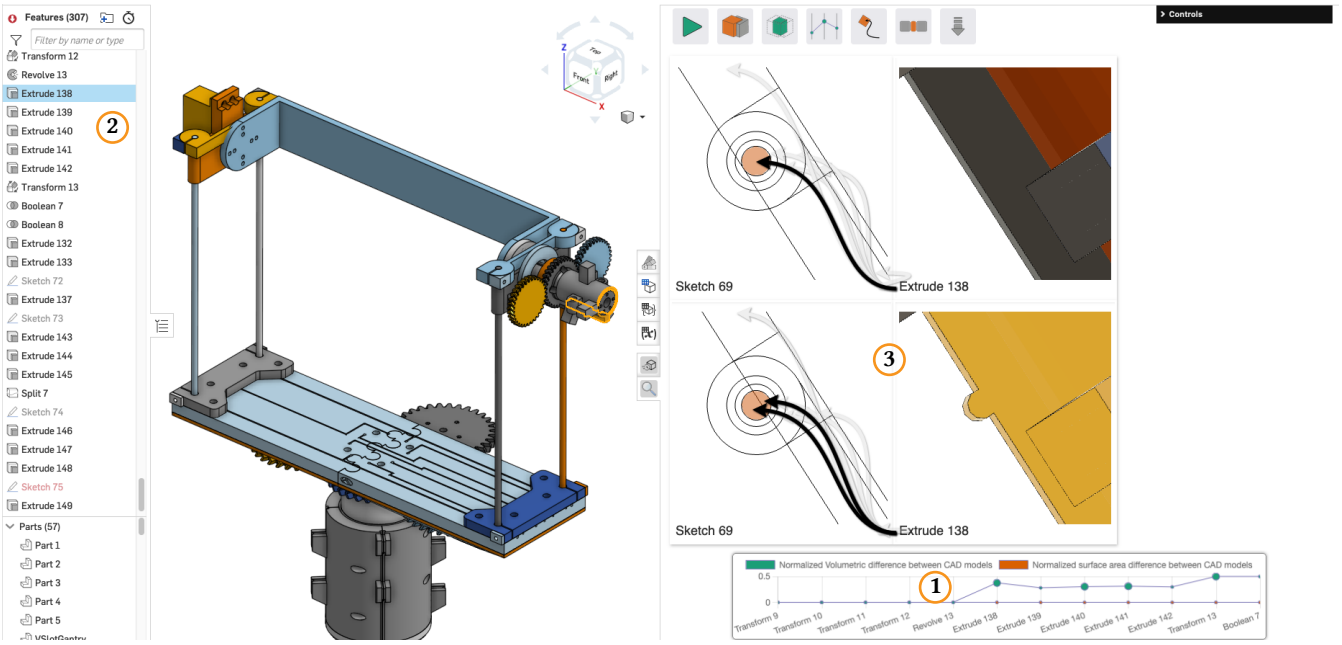

Figure 11: Interaction with DeCAD. (1) Using the volume difference chart, the user identifies the first geometrically divergent operation, `Extrude138`. After selecting `Extrude138` (2) and the edited operation `Sketch69` (here out of view), the user discovers the reference error in the two-dimensional view (3). A semi-circle has been extruded, adding an unintended bump to the previously straight edge. The first failed operation, `Sketch75` appears 47 operations after the edit, making this error difficult to locate without DeCAD.

Participants. The debugging tasks in our study involve three components which can be challenging for participants. First, CAD systems are known for having a steep learning curve. Second, participants will need to learn how to use our tool within the time allotted during the study. Third, participants will need to understand the workings of an unknown CAD model.

Anticipating these challenges, we were looking for participants with extensive experience in a parametric CAD system to lower at least the impact of the first challenge. We contacted CAD users through relevant mailing lists and messaging channels within our institution. In our sign-up form, we asked potential participants in what context they use CAD, which CAD software they are most familiar with and for how many years they have been practicing CAD. We ran the entire study with 13 participants. After observing challenges related to learning parametric CAD concepts or adapting to substantial software differences, e.g. users who were only familiar with Solidworks, we discarded 7 participants. Finally, we retained data from six expert participants who either have been using Onshape or Solidworks for a long time (over 5 years) or who have been primarily using Onshape for a shorter time, but also for side projects, see Table 1.

5.2 Results

Based on participants' interactions and answers during the lab study, we identify 5 challenges that were commonly encountered and four workflow phases. Using DeCAD, participants managed to overcome our hypothesized challenges and to employ a new workflow. However, using our tool, they also encounter two new challenges.

These results are best understood by looking at the interactions. Please refer to the accompanying video and the supplemental materials.

5.2.1 What challenges do users face when debugging reference errors? We examine the sessions to confirm that the four challenges identified in Section 3.3 reflect reality. Our analysis reveals evidence to support each of the four initially hypothesized challenges, along with the identification of an unforeseen fifth challenge.

Challenge 1: Geometric complexity. We observe that participants invest time interacting with 3D geometry to improve their understanding, e.g. by hiding 3D parts, rotating around the model, and zooming into various parts.

P4 described that geometric information can be obstructed by different parts of the scene: "*That was hard to identify, because there's other parts in the middle. This gear gets in the way this part gets in the way, and it was difficult to tell that without just going through feature by feature.*" (P4)

Whereas another participant strategically decided to get a holistic view of the geometric change before inspecting operations: "*I guess it might be good to just compare back and forth first, to see what are we expecting to change with the whole. And what are we not expecting to change with it.*" (P2)

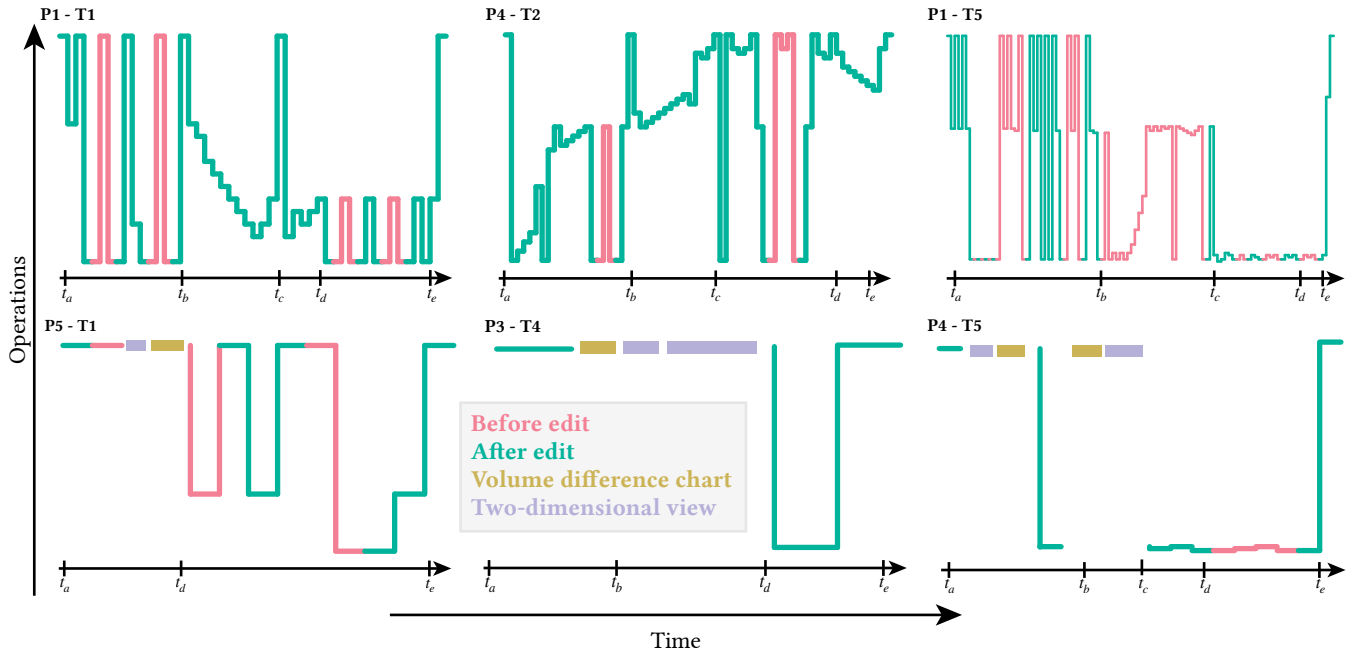

Figure 12: CAD model states of three debugging sessions without DeCAD (top row) and with DeCAD (bottom row). Each step along the x-axis represents one model state change. The y-axis indicates the index of the current operation in the CAD program. The label `Pi-Tj` stands for participant i on task j. We identify four common debugging phases: (1) $[t_a, t_b]$: Searching for the most relevant geometric error, often by undoing the edit. (2) $[t_b, t_c]$: Searching for the first operation featuring a reference error, often by inspecting many operations. (3) $[t_c, t_d]$: Gathering contextual information, often by inspecting operations leading up to the faulty operation. (4) $[t_d, t_e]$: Acquiring more detailed information to correct the reference error, often by undoing the edit. In the bottom row, we observe that frequent model state changes are being replaced by fewer but longer interactions with DeCAD's features.

Challenge 2: Program complexity. Users were challenged by the need to explore a long list of operations. When faced with an error, participants have little guidance about what operations to inspect apart from error messages from failed operations. These were generally not perceived as helpful: "*Okay, we're missing a reference to geometry. But what geometry? Good luck!*" (P2)

One common pattern was to start inspecting one operation and to discover changes leading up to that operation. Participants did not describe this process constructively: "*So I kind of spent a while digging around through some other things to see if the error happened earlier. One of the main points I wasn't sure about was at what point did it happen?*" (P6) Other participants described this as *"just going through feature by feature"* and as *"some more poking"*. Additionally, participants found it useful to undo the edit multiple times, see Fig. 12 (Phases (1) and (2)).

Challenge 3: Analyzing Multiple States. The need for simultaneous understanding of the before and after states was evident by users undoing and redoing the edit. As explained in Sec. 3.1, the undo function in CAD systems includes not only program changes, but also other UI interactions which are being used for program exploration. This means that often an edit cannot be undone via a single click and participants had to enter the edited operation and reverse the edit manually. This explains the strong *oscillations* in the first three session diagrams in Fig. 12. This also makes geometry comparison between different program states more tedious.

Challenge 4: Understanding reference dependencies between operations. To find out how to fix an error, most users will try to understand what caused an unintended change. One common strategy is to inspect operations leading up to the suspected reference error and to undo the edit several times to gain visual confirmation, Phase (3) and Phase (4) in Fig. 12, respectively.

However, we observe that the underlying mechanism for changes is not always obvious to participants: "*That is what I believe went wrong. … I don't know that I'm 100% confident in knowing why changing something in Sketch 69 caused the other part to have an error.*" (P6)

Not understanding why a change occurred might the reason for participants undoing the edit multiple times just before correcting a reference error. This general lack of explainability from CAD systems was well expressed by P1: "*I don't understand how this works, but I built enough intuition.*" (P1)

Additional Challenge: Verification. While not previously hypothesized, we observe in our study that participants not only struggled to explain why something changed but also often lacked

confidence in their fix. "*I don't know if I fixed it correctly. But it's not angry anymore.*" (P3)

Interestingly, there is no guarantee for the absence of other silent errors which have not caused any other operation to fail. "*I guess we don't have errors. I'm not entirely convinced the model will actually work.*" (P2)

5.2.2 What debugging processes do users adopt? Even though participants showcased idiosyncratic interactions with the CAD system, we observed four commonly encountered debugging phases: (1) Locate topological changes; (2) Searching for reference errors; (3) Understanding the context; (4) Understanding how to correct an error.

(1) Locate topological changes. Confronted with an erroneous edit, participants first searched for what particular part of the 3D model changed w.r.t. before the edit. This is achieved by undoing the edit multiple times and navigating the 3D geometry to understand what parts have changed.

(2) Searching for the reference error. After locating the general 3D region, many operations are being traversed to search for relevant operations.

(3) Understanding context. Once an operation is being suspected as likely to have a reference error, local context information is gathered to rule out a false positive.

(4) How to correct the error. Since no straightforward explanation about a change is provided to participants from the CAD system, participants try to semantically copy the references from before the edit, applied to the new topology of the model. This can require multiple undo actions of the edit.

5.2.3 How does DeCAD address challenges faced by CAD users? We found that participants managed to use DeCAD and to integrate it in their debugging process. In participants' workflows, DeCAD replaces frequent model state changes by a few interactions with the volume difference chart and the two-dimensional view features, see Fig. 12.

We observed that DeCAD was helpful tackling the aforementioned debugging phases. While DeCAD helps to minimize the number of undo actions to understand different model states, it does not support undoing exploratory edits to fix the error. DeCAD also introduces two new challenges: additional CAD program states can lead to confusion and sometimes too much visual information is introduced with reference arrows.

Finding unintended topological changes and reference errors. A common pattern for finding relevant geometric changes was to use the volume difference chart as a useful starting point. "*It helped me pinpoint where to go look. [...] in one case, the sketches like way over here to the left, and then you only saw that errors come up later. [...] I probably would spend a lot of time focusing on the errors right around that sketch.*" (P1)

Participants were aware of the gap between the information provided by the host CAD system and the hidden operation that they had to find: "*Having this ability to see when something is changing the model, not necessarily when an error is introduced, is really helpful. Because usually, I think in many of these cases the problem with the error is not with the feature that has the error, but it's some sort of feature leading up to it.* " (P2)

The interaction logs show that the first two to three debugging phases are often replaced with a few usages of the volume difference chart and the two-dimensional view, see Fig. 12.

Discovering changes. Once a problematic region of operations was identified via the volume difference chart, participants used the two-dimensional view feature to visually inspect these changes. In this mode, for several participants, the "*change in geometry aspect of it was really handy*" (P3), i.e., the boolean difference visualization. The boolean difference complements the standard, constructive view of the host CAD system by a comparative view which visually explains what an operation might be missing: "*We made a change in the extrude based off the sketch that was changed. It didn't include all the pieces that it should have included. So I determined that was the case by looking at the volume difference*" (P6)

Several participants made use of the reference arrows and appreciated this new visual vocabulary: "*Because I think that's one of those things that's a little bit hard to compare. Otherwise is you're looking at, okay, so we have this area of the sketches red, you know, or it's changed, or something like that.* " (P2). DeCAD's comparison features helped users to tackle the last two debugging phases.

Have I fixed it correctly? After fixing an erroneous edit, some participants lost trust in the correctness of the model because other operations could still be dysfunctional, even if it did not cause an error message. Unexpectedly, we found that the volume difference chart helped raising confidence in an edit, since participants could see which operations would introduce *additional change*: "*I was pretty confident that that would be the fix, because everything after that was just built off of Extrude 4. Because, like once you have this missing piece, everything else kind of looks like it's also missing.*" (P6)

New challenges. During one debugging task, P4 was confused by the three program states in front of them: one from the host CAD system and two from the two-dimensional view. This led to the participant trying to fix an error, even though they had temporarily undone the edit.

The program states of the two-dimensional view are detached from the host CAD system by design to provide users with a permanent view of the entire operation sequence. However, as a side effect, this introduces additional cognitive load.

Additionally, P6 commented on the lack of visibility when they used the reference arrows on task 3. When activated, too many reference arrows would be visualized by default. The participant mentioned that they wanted a more intelligent selection of relevant references, see Fig. 13.

6 DISCUSSION

In this section, we summarize our study findings and the lessons that we have learned into a set of design guidelines for future debugging tool builders. Then, we mention the limitations of our work and discuss future work in this field.

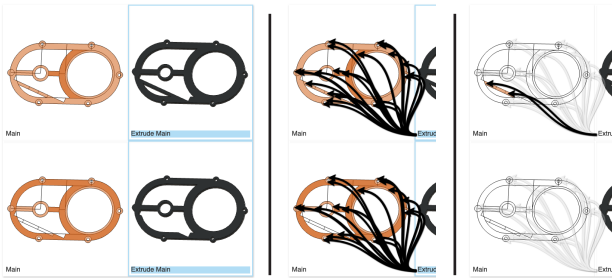

Figure 13: (Left) Initially, the user is presented a visual highlight comparison of reference inputs. (Middle) Activating the reference arrow feature can be overwhelming. (Right) Ideally, we would want to select only relevant references.

6.1 Design guidelines

Comparative features. The features in DeCAD are largely driven by the need for visibility and juxtaposition in debugging [32]. However, currently CAD systems do not provide comparative features in the two-dimensional mode state space from Fig. 6. In the study, participants extensively used the comparison features and found it useful. Future tools should consider providing even more per-operation comparative features.

Tools should be as integrated as possible. While participants appreciated the implementation of DeCAD as an extension to their already known CAD system, we observed that the tool could be even more integrated. In particular, the volume difference chart lets the user interact with a list of operations. However, in Onshape there is already such a list.

Additionally, we observed that while it was helpful that the camera was synchronized between the panels of the two-dimensional view, it would have helped some participants if the main view of Onshape would also be synchronized. We advocate for sharing a single 3D scene between editing and debugging views and a tight integration into the already existing UI instruments.

Similarly to debugging tools in software IDEs, DeCAD does not need to be open all the time, we envision that it can be opened and closed on demand.

Customize the comparative function. The volume difference chart feature is a useful comparative feature in cases where the edit does not intend to change the geometry early on. However, depending on the edit, a pure boolean difference operation as a comparison function might give a noisy and useless signal. We recommend that users can customize the comparison function that they wish to summarize in a timeline chart.

As already mentioned, the reference arrows were sometimes perceived as introducing too much noise. We advocate for more intelligent, customizable selection tools for reference arrows.

Customizable layout. Participants pointed out that they would have liked to open the two-dimensional view and the volume difference chart as external windows or as draggable panels. Indeed, introducing new debugging features should not significantly lower the screen real estate of existing UI elements. Besides from layout customization, we also recommend thinking of debugging information as a kind of reference material used for 3D modeling and to explore the idea of providing temporal features which have been proposed in the context of digital drawing tools [28].

Improve discoverability. Particularly for references, we observed that discoverability is an issue, even in a single model state context. Depending on the current 3D view, referenced entities might be overlooked or it might be hard to discover how many entities are currently being referenced. Our proposed reference arrows showed promising interaction behavior by introducing a new visual vocabulary for references. For future tool-builders, we recommend improving the discoverability of the inner workings of CAD programs.

6.2 Limitations

Observations and data are Onshape-centric. For this work, we made the assumption that Onshape represents an industry-standard parametric CAD modeling tool. While the underlying principles of parametric CAD are shared among different tools, we think that it is important to acknowledge that there are also differences. For example, Solidworks uses a different operation sequence representation, a tree, not a list, which reifies the dependencies between operations. Also in Solidworks, a sketch cannot be shared between multiple operations and the software does not allow multi-part modeling. These restrictions lower the impact of reference errors. Onshape gives users more freedom which comes at a price.

Unknown models in lab study. In our lab study, users were confronted with CAD models which they did not author themselves. Based on our discussions with experts, engineers work mostly on their own models or on models created by team members who can explain them. We argue that developing tools to explore unknown models is still useful. First, in software engineering as in CAD, a program which has not been revisited for a while has to be partially rediscovered even by the author themselves. Second, sharing CAD programs with others has been introduced by Onshape's public documents repository. Models from this repository are commonly copied by other users who need to or want to discover how they work. Third, recent industry effort has been made to translate Solidworks programs into Onshape programs [4]. This is potentially foreshadowing a future where CAD programs will be shared more widely among different CAD systems, as opposed to only geometry, as it is the case today, increasing the need for CAD program exploration tools. Lastly, with the future development of AI tools capable of working with code, images and 3D data, it is likely that CAD programs will be at least partially written by the tool itself [39]. Users will need tools to understand automatically generated CAD programs.

6.3 Future work

Broader applicability. DeCAD has been designed for debugging reference errors through visualization. However, DeCAD's visualization features can also be used to explore the functioning of a modeling sequence without the context of an edit by unrolling parts of the sequence and visualizing references. This could be especially

useful in an educational setting. Additionally, an edit does not need to provoke an explicit error message. DeCAD can be used to detect silent errors and to verify the absence of change in geometry and in reference assignment.

Make suggestions. The features implemented in DeCAD support users in discovering relevant differences in a CAD program before and after an edit. Future work could consider going one step further into working on algorithms which suggest to users which references should be verified. Since matching references is an ill-posed problem, it would be interesting to develop a sound, customizable matching tool to empower users.

Other kinds of errors. In this work, we focused on reference errors. However, another major class of CAD errors, geometric errors would also benefit from future visualization work. For example, after an edit, guide and path curves for Loft operations might no longer be intersecting even though this seems to be the case to the naked eye. Knowing which curves are not intersecting, where they intersected before and why they no longer intersect each other are interesting questions at the intersection of programming and geometry.

Trust in CAD systems. We observed in the lab study that users found it challenging to verify that the CAD model still worked after an edit, even without throwing any errors. This points to a larger question: why should a user trust a CAD system to not modify their program without their consent in unexpected places? And how do we recover the trust of the user in the model? Future work should investigate verification mechanisms for users and think about asking the user's consent before making changes.

7 CONCLUSION

In this work, we tackle errors in CAD as a debugging problem. First, we analyze the domain-specific debugging challenges which arise when reasoning about both geometry and program structure. Informed by this analysis, we prototype DeCAD, a debugger for CAD which supports users in comparing different model states. We use DeCAD in a qualitative lab study as a probe to better understand user's challenges and workflows. We hope that our findings will be informative for future tool-builders to support CAD designers and educators.

ACKNOWLEDGMENTS

We thank the reviewers for their helpful feedback, user study participants for their time and Ilya Baran and Matthew Mueller from Onshape for the insightful discussions and the provided data. This work was supported by NSF CCF-2219864 and NSF CCF-2319181 as well as gifts from Amazon and Meta.

REFERENCES

[1] 2023. CADQuery. https://github.com/CadQuery/cadquery. Accessed: 2023-12-19.
[2] 2024. Onshape. https://www.onshape.com/en/. Accessed: 2024-03-25.
[3] 2024. *Parasolid.* https://plm.sw.siemens.com/en-US/plm-components/parasolid/
[4] 2024. *SolidTranslate.* https://www.cadsharp.com/solidtranslate/
[5] 2024. Solidworks. https://www.solidworks.com/. Accessed: 2024-03-25.
[6] 2024. *three.js.* https://threejs.org/
[7] Ekansh Agrawal, Omair Alam, Chetan Goenka, Medha Iyer, Isabela Moise, Ashish Pandian, and Bren Paul. 2024. Code Compass: A Study on the Challenges of Navigating Unfamiliar Codebases. *arXiv preprint arXiv:2405.06271* (2024).
[8] Abdulaziz Alaboudi and Thomas D LaToza. 2021. An exploratory study of debugging episodes. *arXiv preprint arXiv:2105.02162* (2021).
[9] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*. 572–583.
[10] Gilbert Louis Bernstein and Wilmot Li. 2015. Lillicon: Using transient widgets to create scale variations of icons. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–11.
[11] Theodore J Branoff and James H Earle. 2016. *Interpreting engineering drawings*. Cengage Learning.
[12] Dan Cascaval, Rastislav Bodik, and Adriana Schulz. 2023. A Lineage-Based Referencing DSL for Computer-Aided Design. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 76–99.
[13] Dan Cascaval, Mira Shalah, Phillip Quinn, Rastislav Bodik, Maneesh Agrawala, and Adriana Schulz. 2022. Differentiable 3D CAD Programs for Bidirectional Editing. In *Computer Graphics Forum*, Vol. 41. Wiley Online Library, 309–323.
[14] Kathy Cheng, Phil Cuvin, Alison Olechowski, and Shurui Zhou. 2023. User Perspectives on Branching in Computer-Aided Design. *Proceedings of the ACM on Human-Computer Interaction* 7, CSCW2 (2023), 1–30.
[15] Ladislav Čmolík and Jiří Bittner. 2010. Layout-aware optimization for interactive labeling of 3D models. *Computers & Graphics* 34, 4 (2010), 378–387.
[16] Xingyi Du, Qingnan Zhou, Nathan Carr, and Tao Ju. 2021. Boundary-sampled halfspaces: a new representation for constructive solid modeling. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–15.
[17] Ran Gal, Olga Sorkine, Niloy J Mitra, and Daniel Cohen-Or. 2009. iWIRES: An analyze-and-edit approach to shape manipulation. In *ACM SIGGRAPH 2009 papers*. 1–10.
[18] Aditya Ganeshan, Ryan Y Huang, Xianghao Xu, R Kenny Jones, and Daniel Ritchie. 2024. ParSEL: Parameterized Shape Editing with Language. *arXiv preprint arXiv:2405.20319* (2024).
[19] Sneha Gathani, Peter Lim, and Leilani Battle. 2020. Debugging database queries: A survey of tools, techniques, and users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–16.
[20] David J Gilmore. 1991. Models of debugging. *Acta psychologica* 78, 1-3 (1991), 151–172.
[21] Johann Felipe Gonzalez, Danny Kieken, Thomas Pietrzak, Audrey Girouard, and Géry Casiez. 2023. Introducing Bidirectional Programming in Constructive Solid Geometry-Based CAD. In *Proceedings of the 2023 ACM Symposium on Spatial User Interaction*. 1–12.
[22] Ken Gu, Eunice Jun, and Tim Althoff. 2023. Understanding and supporting debugging workflows in multiverse analysis. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–19.
[23] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 579–584.
[24] Jeffrey Heer, Jock Mackinlay, Chris Stolte, and Maneesh Agrawala. 2008. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE transactions on visualization and computer graphics* 14, 6 (2008), 1189–1196.
[25] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-sketch: Output-directed programming for svg. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 281–292.
[26] Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, and Vladimir Lara-Villagrán. 2016. Learning principles in program visualizations: A systematic literature review. In *2016 IEEE frontiers in education conference (FIE)*. IEEE, 1–9.
[27] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2016. Visual debugging techniques for reactive data visualization. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 271–280.
[28] Josh Holinaty, Alec Jacobson, and Fanny Chevalier. 2021. Supporting reference imagery for digital drawing. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2434–2442.
[29] Takeo Igarashi, Tomer Moscovich, and John F Hughes. 2005. As-rigid-as-possible shape manipulation. *ACM transactions on Graphics (TOG)* 24, 3 (2005), 1134–1141.
[30] Peiling Jiang, Fuling Sun, and Haijun Xia. 2023. Log-it: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–16.
[31] Hyeonsu Kang and Philip J Guo. 2017. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 737–745.
[32] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.
[33] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006), 971–987.

[34] Uday Kusupati, Mathieu Gaillard, Jean-Marc Thiery, and Adrien Kaiser. 2024. Semantic Shape Editing with Parametric Implicit Templates. In *ACM SIGGRAPH 2024 Conference Papers*. 1–11.

[35] Thomas D LaToza and Brad A Myers. 2011. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 117–124.

[36] Sorin Lerner. 2020. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–7.

[37] Tom Lieber, Joel R Brandt, and Rob C Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2481–2490.

[38] Richard Lin, Rohit Ramesh, Antonio Iannopollo, Alberto Sangiovanni Vincentelli, Prabal Dutta, Elad Alon, and Björn Hartmann. 2019. Beyond schematic capture: Meaningful abstractions for better electronics design tools. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–13.

[39] Liane Makatura, Michael Foshey, Bohan Wang, Felix HähnLein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Elaine Owens, Peter Yichen Chen, et al. 2023. How Can Large Language Models Help Humans in Design and Manufacturing? *arXiv preprint arXiv:2307.14377* (2023).

[40] Aman Mathur, Marcus Pirron, and Damien Zufferey. 2020. Interactive programming for parametric cad. In *Computer graphics forum*, Vol. 39. Wiley Online Library, 408–425.

[41] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[42] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. Bifröst: Visualizing and checking behavior of embedded systems across hardware and software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 299–310.

[43] William McGrath, Jeremy Warner, Mitchell Karchemsky, Andrew Head, Daniel Drew, and Bjoern Hartmann. 2018. Wifröst: Bridging the information gap for debugging of networked embedded systems. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 447–455.

[44] Elie Michel and Tamy Boubekeur. 2021. DAG amendment for inverse control of parametric shapes. *ACM Transactions on Graphics (TOG)* 40, 4 (2021), 1–14.

[45] George A Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review* 63, 2 (1956), 81.

[46] Guillaume Pothier and Éric Tanter. 2009. Back to the future: Omniscient debugging. *IEEE software* 26, 6 (2009), 78–85.

[47] Eduardo Rinaldi, Davide Sforza, and Fabio Pellacini. 2023. NodeGit: Diffing and Merging Node Graphs. *ACM Transactions on Graphics (TOG)* 42, 6 (2023), 1–12.

[48] Nischal Shrestha, Titus Barik, and Chris Parnin. 2021. Unravel: A fluent code explorer for data wrangling. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 198–207.

[49] Ruben M Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. 2014. A survey on procedural modelling for virtual worlds. In *Computer graphics forum*, Vol. 33. Wiley Online Library, 31–50.

[50] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–64.

[51] Richard Stallman, Roland Pesch, Stan Shebs, et al. 1988. Debugging with GDB. *Free Software Foundation* 675 (1988).

[52] Ian Vollick, Daniel Vogel, Maneesh Agrawala, and Aaron Hertzmann. 2007. Specifying label layout style by example. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. 221–230.

[53] Anneliese Von Mayrhauser and A Marie Vans. 1997. Program understanding behavior during debugging of large scale software. In *Papers presented at the seventh workshop on Empirical studies of programmers*. 157–179.