Programmatic Interfaces for Design & Simulation

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Aman Shankar Mathur

Date of Defense:	February 9, 2023
Dean:	Prof. Dr. Christoph Garth
Reviewer:	Prof. Dr. Eva Bruggisser (née Darulova)
Reviewer:	Prof. Dr. Rupak Majumdar
Reviewer:	Dr. Damien Zufferey

DE-386

Abstract

Though Computer Aided Design (CAD) and Simulation software are mature, well-established, and in wide professional use, modern design and prototyping pipelines are challenging the limits of these tools. Advances in 3D printing have brought manufacturing capability to the general public. Moreover, advancements in Machine Learning and sensor technology are enabling enthusiasts and small companies to develop their own autonomous vehicles and machines. This means that many more users are designing (or customizing) 3D objects in CAD, and many are testing machine autonomy in Simulation. Though Graphical User Interfaces (GUIs) are the de-facto standard for these tools, we find that these interfaces are not robust and flexible. For example, designs made using GUI often break when customized, and setting up large simulations can be quite tedious in GUI. Though programmatic interfaces do not suffer from these limitations, they are generally quite difficult to use, and often do not provide appropriate abstractions and language constructs.

In this Thesis, we present our work on bridging the ease of use of GUI with the robustness and flexibility of programming. For CAD, we propose an interactive framework that automatically synthesizes robust programs from GUI-based design operations. Additionally, we apply program analysis to ensure customizations do not lead to invalid objects. Finally, for simulation, we propose a novel programmatic framework that simplifies building of complex test environments, and a test generation mechanism that guarantees good coverage over test parameters.

Our contributions help bring some of the advantages of programming to traditionally GUI-dominant workflows. Through novel programmatic interfaces, and without sacrificing ease of use, we show that the design and customization of 3D objects can be made more robust, and that the creation of parameterized simulations can be simplified.

Contents

List of Figures

List of Tables

Acknowledgments

This Thesis would have not been possible without the help and support of my advisors, Damien Zufferey and Rupak Majumdar. I feel incredibly lucky to have got an opportunity to work and learn from them. I started working with Rupak when I was still a Master student, and his dedication, motivation, approachability, and kindness have been incredibly influential in my development. Likewise, Damien went above and beyond in his support for me. I will always cherish our long, deep, and insightful conversations. His knack for seeing the big picture, while still being able to provide concrete and practical advice has been invaluable. All in all, I could not have asked for better mentors.

I am also grateful to Laura Stegner and Marcus Pirron, my collaborators on some of the work included in the Thesis. Working with them was a pleasure, and it felt great knowing that I could count on them.

Our work benefited from many wonderful open-source projects and communities. I would especially like to thank the CadQuery community, and Jeremy Wright in particular for our conversations about CAD, and for supporting our work, as well as many others.

I thank the Thesis reviewers, especially Eva Bruggisser (née Darulova) for the helpful suggestions for improvement, and Anthony Lin for being a great Chair to the Thesis Committee.

Last, but not the least, I'd like to thank everyone at the Max Planck Institute for Software Systems (MPI-SWS). The support I received at MPI-SWS was instrumental in whatever I achieved during my stay here. I not only experienced great personal and professional growth, but also had tons of fun! I would like to thank all my wonderful colleagues at MPI-SWS for the lively conversations, unconditional help, and the wonderful (and hopefully lasting) friendships.

Aman Shankar Mathur
Heilbronn, May 2, 2023

A display connected to a digital computer gives us a chance to gain familiarity with concepts not realizable in the physical world. It is a looking glass into a mathematical wonderland.

— Ivan Sutherland [143]

1 Introduction

Many years ago, in 1963, Ivan Sutherland developed SKETCHPAD [142], an invention that has proven to be a major milestone in the way we use and interact with computers. SKETCHPAD (among other things) introduced two powerful ideas: (a) everyday objects/geometry can be modelled and simulated using computers, and (b) a display and a pointer enable an intuitive human-computer interaction.

SKETCHPAD was the first project to demonstrate the utility of designs made on a computer, and was a precursor to the many subsequent breakthroughs in Computer Graphics. SKETCHPAD also inspired modern Graphical User Interfaces (GUIs). These interfaces are ubiquitous today, and are not just popular for modelling and simulation, but in almost every popular software we use today: from calculators to social networks. From humble beginnings in the form of the SKETCHPAD proof-of-concept, Computer Aided Design (CAD) and Simulation tools have moved on to form a mature and well-established industry today. They are virtually indispensable in many professional domains—architecture, engineering, and entertainment, to name a few. Even still, recent advances in 3D printing and machine autonomy are expanding their user base.

Traditionally, CAD software has catered to a professional audience. However, with the recent advent of cheap and good quality 3D printing, this is quickly changing. Whereas before professional designers worked on a single final object for large-scale production, 3D printing has brought manufacturing capability closer to everyday users. Anyone with access to a sufficiently capable 3D printer can manufacture objects or components that would have previously required specialized expertise and tooling. As a result, many people today are using CAD software to design, customize, and 3D print their own objects. Moreover, platforms such as THINGIVERSE [96] are enabling users all over the world to share their designs, customize these, and download or 3D print the results.

A similar transformation is also happening with Simulation. Whereas traditionally simulation tools were used to model and test machines in fixed environments, simulations are now being used to test autonomous systems operating in complex and ever-changing

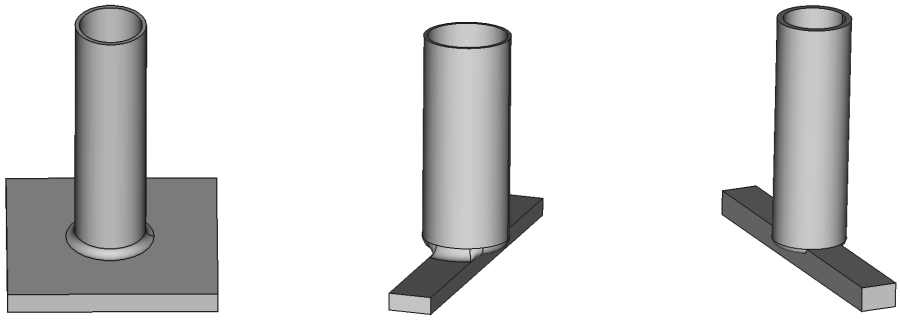
real-world scenarios [134, 40, 149, 42, 152]. Autonomous agents such as self-driving cars, drones, and robots are expected to soon become ubiquitous in many critical domains such as transportation, search and rescue, medicine, etc. As these agents interact more and more with humans, there are obvious risks of accidents. Autonomous agents have already been at the centre of several serious incidents involving damage to life and property [156, 14]. Therefore, there are many well-founded concerns regarding the reliability and safe operation of these machines, and ensuring this is an important problem. Consider the example of self-driving cars. Some estimates suggest that guaranteeing their safety may take hundreds of billions of kilometres of driving data [77]. Real-world testing is costly and time consuming. Moreover, if problematic cases are found, it is difficult to recreate them. Simulations have therefore emerged as a low-cost alternative for quick, early, and frequent testing. They allow precise control over all aspects of the environment, such as the light and weather conditions, as well as test components like road segments, traffic lights, other vehicles, pedestrians etc. There is no need for finished vehicles, expensive measurement devices, and lengthy a test approval process. This is enabling even small companies and enthusiasts to prototype and test their autonomous machine implementations.

1.1. Summary of Challenges and Contributions

Almost all major CAD and 3D Simulation tools are GUI-based. In contrast to text-based programming, GUI enables users to directly modify what they want, and provides immediate visual feedback. This is especially helpful for Design and Simulation due to the visual and 3-dimensional (3D) nature of these workflows. GUI-based interfaces, although incredibly easy to learn and use, also have some serious limitations. In our research, we find that these limitations are especially compounded by the newer use-cases of these tools, i.e., customization-heavy designs, and simulation of complex worlds. The focus of this Thesis is on novel program interfaces that help alleviate these limitations, while still retaining simplicity and ease of use. We now briefly discuss some of the limitations of GUI, and introduce our contributions.

1.1.1. Design

Most popular CAD tools, such as Autodesk Fusion 360, SolidWorks, PTC Creo, OnShape, FreeCAD, etc., share similar (GUI-based) interfaces, and follow the same modelling paradigm (*parametric* CAD). Objects are represented by the sequence of operations in their design, rather than their final form. This enables users to change design parameters, which in turn re-executes the sequence of operations, and produce a (slightly) different final object. In principle, this allows great flexibility: a single design can represent a wide-variety of final objects. Instead of designing custom-fitting objects from scratch, end-users can simply adjust parameters of existing designs. In practice, however, this flexibility does not pan out.



(a) Initial design. (b) Expected (after modification). (c) After modification (GUI-based tool).

Figure 1.1.: A simple design consisting of a hollow tube mounted on top of a rectangular base with the mounting edge filleted (smoothed). When this design is modified (base is made narrower), we expect all edges connecting the tube and the base to continue to be filleted. However, we get a different (unexpected) result.

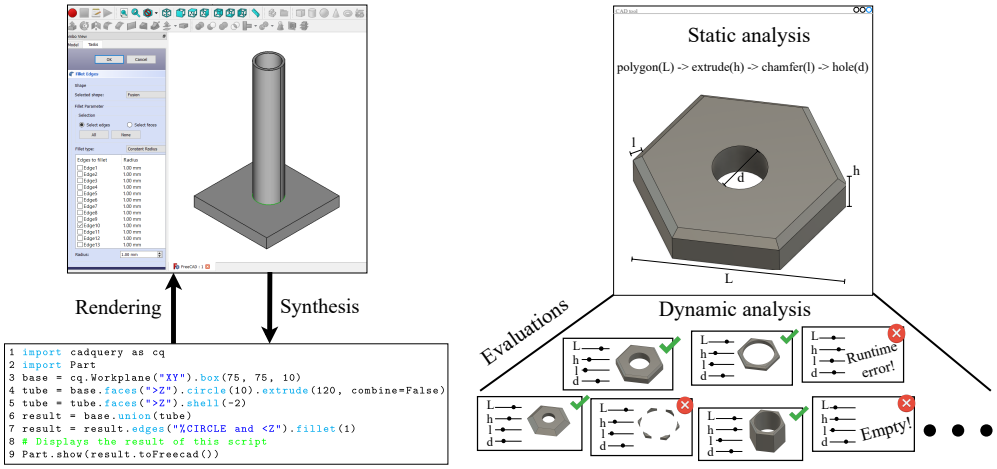
Bridging GUI and Programming for CAD

GUI-based CAD interfaces allow users to directly select elements they want to modify, and then apply operations to these. The specification of which elements users select, however, is *latent*. Users directly make selections in the GUI, rather than encoding the logic behind their selection.

Consider the example presented in Figure 1.1. Here, as depicted in Figure 1.1b, initially one circular edge is selected and filleted (rounded). As the design is parametric, its parameters can be changed. When this happens, (ideally) the CAD tool needs to recompute the design and present a new final object reflecting the *design intent*. However, as there are more (and different) edges in the design now, it is difficult for the GUI-based tool to know which edge(s) to apply the fillet operation to. There is no semantic information for this. Therefore, as depicted in Figure 1.1c, GUI-based CAD interfaces often end-up with modifications that do not match the design intent.

Program-based designs, on the other hand, are more precise. Users explicitly state, for all valid parameter values, which element(s) they want to select and modify. Therefore, in comparison to their GUI-based counterparts, program-based designs are robust to parameter changes. Many programmatic CAD interfaces exist, for example OpenSCAD, Open CASCADE and CadQuery. However, due to the complexity of writing code and the missing visual feedback, these programmatic interfaces are much less popular than GUI.

We propose bringing together the ease of use of GUI with the robustness of programming. As depicted in Figure 1.2a, we present a system that uses GUI interactions to automatically synthesize code. We use a novel Decision Tree algorithm for this, which prefers short programs that *generalize*. Moreover, we use techniques from program analysis to synthesize sub-programs at the relevant line number, and using the relevant intermediary object and



(a) Interactive design (b) Design exploration

Figure 1.2.: We apply program language techniques to the design of objects. As depicted in (a), we automatically synthesize robust program segments from GUI-based design operations. This enables interactive design with a programmatic back-end. As depicted in (b), we also apply static and dynamic program analysis techniques to parametric designs, and help in the design parameter space exploration by automatically eliminating invalid configurations.

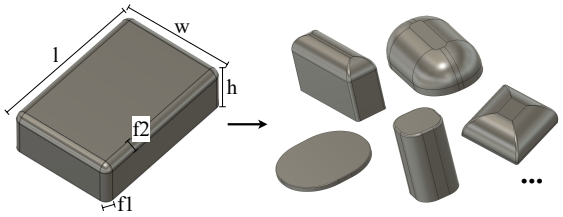


Figure 1.3.: A simple parametric design with 5 parameters, and some valid variations.

scope. Based on experimental evidence, we find that our technique synthesizes robust code, and that we do so almost instantaneously, enabling users to continue designing on a GUI, while still benefiting from a robust programmatic back-end.

Synthesis of CAD Parameter Constraints

The power of parametric design enables designers, with just a few CAD operations, to represent a large family of objects. However, even when the design is robust, there may be parameter values that lead to invalid results, for example run-time errors, or empty/broken objects. As the number of parameters and corresponding CAD operations increase, finding out which parameter values lead to valid results, and which do not, becomes difficult. For example, consider the design in Figure 1.3. The design consists of a box with its top-most edges, and edges orthogonal to these filleted. We have 3

parameters for the dimensions of the box, and 1 each for the two fillets. By varying these 5 parameters, as depicted in Figure 1.3, a large variety of final objects can be obtained. However, for this design, only 3.1% of randomly chosen parameter configurations lead to valid results. If the fillet radii are too large, the operation cannot be accommodated in the base object, and fails with a runtime error. In this design, if we constrain the two fillet radii to `f1 < 0.5*min(w,l)`, and `f2 < min(0.5*w,0.5*l,h)`, all parameter configurations shall be valid. Such constraints are quite useful as they ensure end-users always choose valid parameter values. They also provide a high-level perspective on designs, and the diversity of final objects they support. However, coming up with these constraints is difficult. Therefore, though public design repositories such as THINGIVERSE encourage designers to also provide relevant parameter constraints, very few uploaded designs have this information.

As depicted in Figure 1.2b, we propose an approach for synthesizing CAD parameter constraints automatically. We do this by using a mix of *static* and *dynamic* analysis. As each design consists of a sequence of CAD operations, we first collect constraints that must hold due to the kind of operations used in the design. This is the static analysis step. Once we have an initial set of constraints, we move on to dynamic analysis. Here, we sample the design using many different parameter values and observe the results. Correspondingly, we propose hypotheses and check if these hold. Once we arrive at a hypothesis that explains the observed results, this becomes our parameter constraint. We evaluate our technique on designs from an open-source dataset, and find that we can accurately and quickly synthesize parameter constraints for a wide variety of designs.

1.1.2. Simulation

Modern simulation tools such as AIRSIM [134] and CARLA [40] enable engineers to test their autonomous vehicle implementations in realistic urban scenarios. Typically via a GUI-based interface (see Figure 1.4), users can add, tweak, or remove components in the simulation, and create a large variety of test cases. Though such an interface is incredibly easy to use, it becomes tedious for large and complex simulations with many test parameters. Building test scenarios requires each component, such as road segments, vehicles, trees, light sources, etc., to be placed by hand. Each scenario, therefore, is a bespoke configuration requiring significant human effort for its design. As autonomous agents require testing in large varieties of different settings and environments, comprehensive tests can require many different test environments to be designed. Moreover, each test environment can have several parameters. These parameters can control environment variables (e.g. weather conditions), visual features (e.g. colors of vehicles on the road), as well as behaviors of other components (e.g. speeds of other vehicles). Though such parameters can be changed quite easily in simulation, as the number of test parameters increase, navigating the test parameter space systematically can become quite challenging.

To solve these issues, we propose PARACOSM, a high-level language for programmatically constructing parameterized environments and test cases. A PARACOSM program represents a family of tests, where each instantiation of the program's parameters is a concrete test

Figure 1.4.: Overview of the AirSim interface. Assets can be dragged and dropped on to the Scene. Their properties may be changed via the Inspector. Building a complex simulation in this way involves the arrangement of many Assets such as roads, other vehicles, pedestrians, buildings etc., and the setting of their properties.

case. A Paracosm configuration consists of a composition of several components. Using a set of system-defined components (road segments, cars, pedestrians, etc.) combined using expressive operations from the underlying reactive programming model, users can set up complex and temporally varying driving scenarios. To navigate possibly large test parameter spaces systematically, Paracosm uses a novel test generation strategy that guarantees good coverage. Our strategy essentially ensures that there are no large unexplored regions in the parameter space, and that all sub-regions are equally well explored. As depicted in Figure 1.5, using a single Paracosm program and a few test parameters, several test cases can be easily produced. Using the Paracosm interface, we have been able to test autonomous vehicle implementations in many different environments, and detect incorrect behaviors and degraded performance.

1.1.3. Why *programmatic* Interfaces?

In this Thesis, we propose novel program interfaces and techniques for the design and customization of 3D objects (see Figure 1.2), as well as the systematic testing of autonomous agents (see Figure 1.5). GUI-based interfaces have dominated these domains. We find that though GUI-based interfaces are easy to get started with, they have two critical limitations: (a) they cannot encode semantics (for example representing a family of designs or simulations via a few exposed parameters), and (b) they are tedious for large projects with repetition and structure.

Clearly, programming shines in both these concerns. Programs are essentially explicit semantics. Via parameters, they can encode a wide variety of output configurations, be

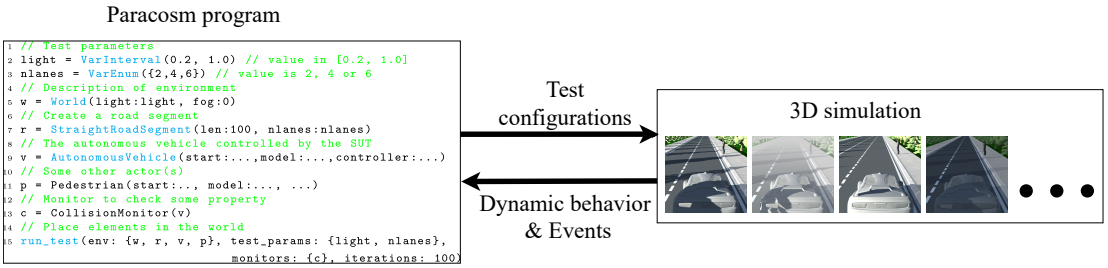


Figure 1.5.: We propose Paracosm, a programmatic interface that enables the systematic testing of autonomous agents.

it different 3D objects, or simulation environments. Moreover, via loops and functions, programs support repetition and structure, enabling the creation of complex components with just a few lines of code. This is the intuition behind our contributions.

As GUI-based CAD suffers from robustness issues, we propose automatically synthesizing robust programs to represent designs. To uncover the space of valid parameter configuration of designs, we treat designs as programs, with invalid configurations being 'bad' states that need to be characterized. Finally, to simplify the testing of autonomous vehicles, we propose a high-level language interface that enables the construction of parameterized environments and behaviors. We then use ideas from program testing to navigate this parameter space efficiently.

1.2. Organization of Content

The Thesis is divided into three parts.

Part I, entitled 'Design' contains our work on parametric CAD. Here, Chapter 2 provides details on our proposed interface for bridging GUI and programming for CAD. We also present various case studies and examples of our technique in action. Chapter 3 describes our analysis and constraint synthesis approach for parametric CAD. In Chapter 4, we summarize some related work.

Part II, entitled 'Simulation' contains our work on simulation design and testing. In Chapter 5, we provide an overview of the Paracosm language interface, and explain our test generation strategy. Then, in Chapter 6, we evaluate the Paracosm interface on different test scenarios and autonomous agent implementations. Chapter 7 summarizes the related work.

Finally, in Part III (Chapter 8), we discuss some interesting directions for future work, and conclude.

1.3. Publications

The content described in this Thesis has appeared in the following peer-reviewed publications:

(i) A. Mathur, M. Pirron, and D. Zufferey. Interactive Programming for Parametric CAD. In *Computer Graphics Forum*. © 2020 Eurographics - The European Association for Computer Graphics and John Wiley & Sons Ltd, 2020. doi: 10.1111/cgf.14046

(ii) A. Mathur and D. Zufferey. Constraint Synthesis for Parametric CAD. In S.-H. Lee, S. Zollmann, M. Okabe, and B. Wünsche, editors, *Pacific Graphics Short Papers, Posters, and Work-in-Progress Papers*. The Eurographics Association, 2021. ISBN 978-3-03868-162-5. doi: 10.2312/pg.20211396

(iii) R. Majumdar, A. Mathur, M. Pirron, L. Stegner, and D. Zufferey. Paracosm: A Test Framework for Autonomous Driving Simulations. In E. Guerra and M. Stoelinga, editors, *Fundamental Approaches to Software Engineering*, pages 172–195, Cham, 2021. Springer International Publishing. ISBN 978-3-030-71500-7

Part I.

Design

2

Bridging GUI and Programming

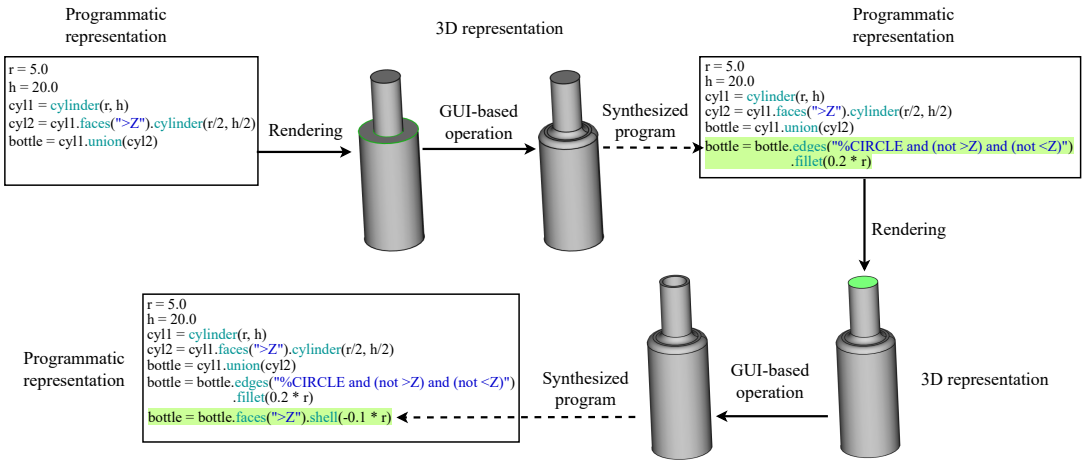
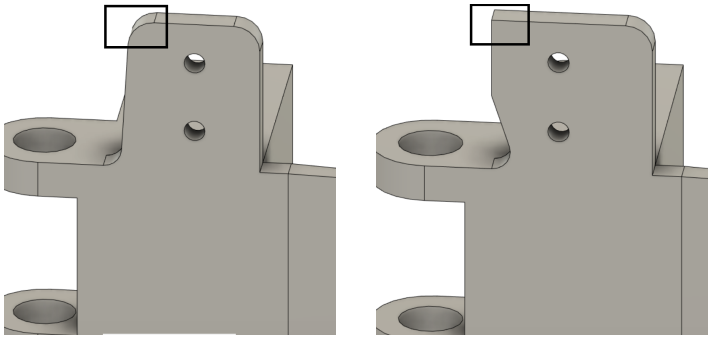


Figure 2.1.: We present a technique to bridge GUI and programmatic interfaces for CAD. With the insight that GUI interfaces are intuitive but brittle, and programming is robust but difficult, we synthesize robust program segments from intuitive GUI-based selections and operations.

2.1. Introduction

As already introduced in Chapter 1 (Section 1.1.1), *parametric* CAD is a modelling paradigm where objects are represented by the sequence of operations in their design, rather than their final forms. This enables users to change design parameters, which re-executes the sequence of operations, and results in a new final object fitting a different use case. Since its introduction in PTC Pro/ENGINEER in 1988 [160], parametric CAD has become the industry standard. Though most popular CAD tools employ parametric design methodology, in practice, changing design parameters often leads to broken results.

There are two interfaces to parametric design: programming, and GUI. Though most popular CAD tools (such as Autodesk Fusion 360, OnShape, SolidWorks, PTC Creo, and FreeCAD) are GUI-based, these interfaces are far from perfect. GUI interfaces are interactive: users can directly manipulate vertices, edges, and faces in the



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

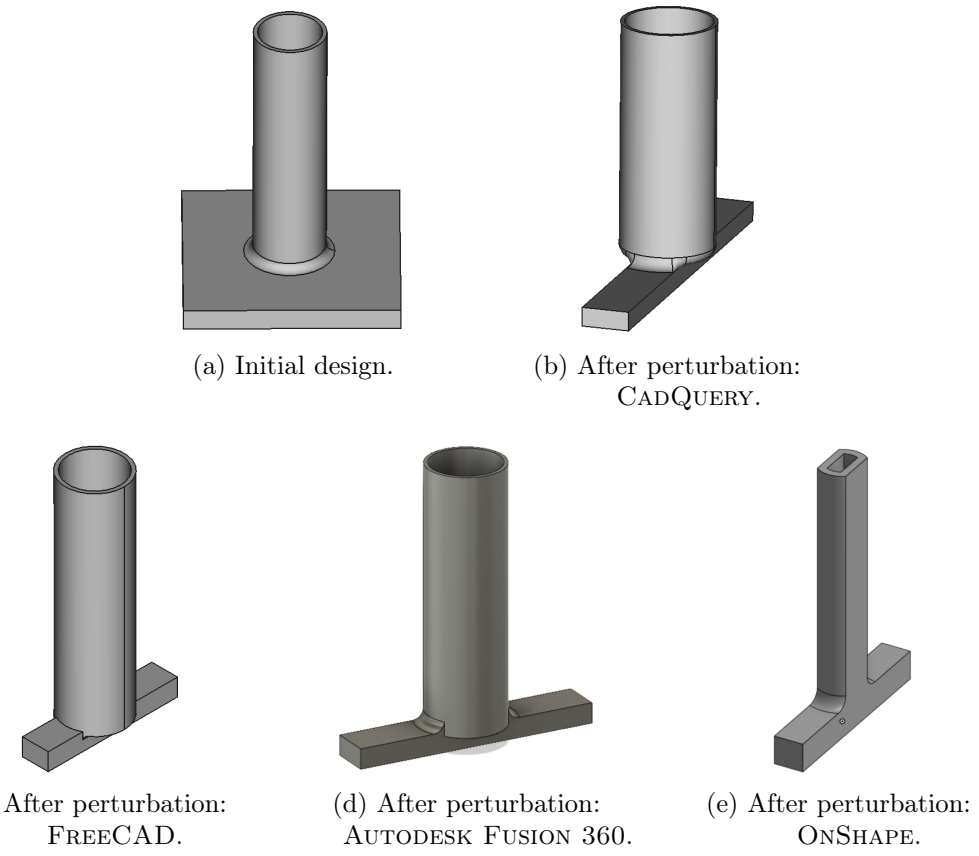
Figure 2.2.: The side bracket design fails to fillet the intended edges (see black rectangle) after the length of the top-most edges is increased.

design. This interactivity, however, is a double-edged sword. The exact specification of which elements users manipulate is not explicit. For many computer applications, this is an acceptable compromise. For parametric CAD, however, this is a severe limitation. For instance, what happens when a designer modifies 1 of 18 edges in a design, and then due to a parameter change, the design needs to be re-evaluated with 36 edges? The new result needs to reflect the 'design intent'. However, due to the underspecification of GUI, this is an impossible task. Therefore, GUI-based CAD tools often end-up with modifications that do not match the design intent. This problem is well-known and many heuristics have been proposed in prior work [28, 29, 26, 81, 3]. In our observations, we find that robustness issues are still quite ubiquitous.

Programs, on the other hand, do not have this issue. Programs clearly identify which elements need to be modified, for all valid combinations of parameter values. Depending on the API, this is done by semantically selecting elements that possess explicit features, using loops (OpenSCAD [80] and Open CASCADE [112]), or declarative queries (CadQuery [115], FeatureScript [49] and Scadla [161]). However, writing complicated semantics representative of simple GUI interactions has proven to be an unfortunate barrier-to-entry. As a result, programmatic interfaces are much less popular than their GUI-based peers. In this Chapter, we propose bringing the ease of use of GUI with the robustness of programming, and present a system that uses GUI interactions to automatically synthesize code.

2.1.1. Motivation

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



(a) Initial design.

(b) After perturbation: CadQuery.

(c) After perturbation: FreeCAD.

(d) After perturbation: Autodesk Fusion 360.

(e) After perturbation: OnShape.

Figure 2.3.: A simple dowel end-cap design and results of changing dimensions of the base in various popular CAD tools.

Design intent

GUI-based CAD tools often fail in capturing design intent. This is true even for curated designs on professional-grade CAD software. We present an example from Project Egress [13], a large collaborative project for building a replica of the Apollo 11 space hatch, designed in Autodesk Fusion 360. Figure 2.2 shows a simple perturbation on the capsule side bracket module. The perturbation causes more edges to appear in the resulting object as before, which in turn causes the fillet (rounding) operation to fail. Clearly, the CAD tool fails to capture design intent (the top-left edge is not filleted any more).

Different and unknown heuristics

Different tools use different heuristics to try and resolve ambiguity. Unfortunately, these heuristics are usually not known a priori, and designers often observe different unexpected results on different tools. Consider the simple design in Figure 2.3 (inspired by a dowel-

end cap design [23] on Thingiverse, an online repository of several CAD projects). We re-designed this example on CadQuery, a programmatic interface for a baseline specification of what we want. Using the same steps, we also designed this on FreeCAD, Autodesk Fusion 360 and OnShape, all of which are GUI-based. On the initial design, we change the dimensions of the rectangular base. All tools under test yield different results. FreeCAD fillets one edge on the right of the connection between the hollow rod and the base. Autodesk Fusion 360 fillets two opposite sides of the connection. OnShape changes the shape of the hollow rod. Reducing the width of the rectangular base leads to there being more edges than before while performing the fillet operation. FreeCAD does not use complex heuristics. Internally, it uses names for all elements in the design, and operations are applied to elements by name. So, when the design changes, the names are re-evaluated and the fillet is applied to the element(s) with the same name. As is evident, this often does not work well. Unfortunately, due to the closed-source nature of Autodesk Fusion 360 and OnShape, we cannot be certain of their ambiguity resolution heuristics.

Programming and GUI

In the context of the two examples provided above, writing a programmatic specification for each step in the design process has two advantages: (a) it makes the design intent explicit, and (b) it removes the need for ambiguity resolution heuristics. Current GUI-based CAD tools already recognize some other advantages of programming. For example, most tools support Macros, which encapsulate GUI operations into executable chunks. Additionally, the popularity of procedural design has introduced dataflow programming [74] to CAD (examples are Grasshopper3D and Dynamo by Autodesk), where GUI-based designs are connected to dataflow components. Though these extensions offer some benefits of programming, they ultimately link to the GUI and therefore suffer from the same robustness issues. Invariably, increasing intuitiveness leads to lower robustness. However, most major CAD tools today offer an exclusively GUI-based design interface, backed by opaque heuristics for capturing design intent and for ambiguity resolution.

Inspired by recent work on integration of programming and direct manipulation for vector graphics [30], we propose bridging programmatic and GUI-based CAD. We do this by synthesizing programmatic queries representative of designers' GUI selections and operations. We use a modified Decision Tree algorithm for this, which prefers short queries that *generalize*. Moreover, we use techniques from program analysis to synthesize queries at the relevant line number, using the relevant intermediary object, and when applicable, using program variables and scope. Based on experimental evidence, we find that our technique synthesizes queries that are robust, and it does so fairly quickly (taking at most a tenth of a second) for various samples. A user study reveals that our interface is faster, more accurate and preferable to a programming-alone interface.

2.1.2. Contributions

The main contributions of this Chapter are:

(i) We identify bridging programming and GUI (direct manipulation) interfaces as a possible solution to brittleness of GUI-based interfaces for parametric CAD.

(ii) We propose an algorithm for automatic synthesis of relevant selection queries from GUI-based interactions.

(iii) We validate our approach using various designs, application scenarios, and a user study.

2.2. Preliminaries and Overview

In Section 2.1.1, we discussed robustness issues in GUI-based CAD and proposed bridging GUI with programming to remedy this brittleness. We now provide a quick overview of modern CAD interfaces and representations, and how our proposed system addresses the limitations of both, GUI-based, and programmatic CAD.

2.2.1. CAD Representation and Operations

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

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

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

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

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

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

⟨FaceOp⟩ ⊨ (`Shell`) ⟨Solid⟩ ⟨Face⟩`*`

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

CadQuery programmatic interface

CadQuery is a flexible and high-level domain specific language based on Open CASCADE. CadQuery is implemented as a shallow embedding in Python, and therefore inherits its control structure and module system. There are two types of domain specific operations in CAD: (a) algebraic, and (b) *query* operations.

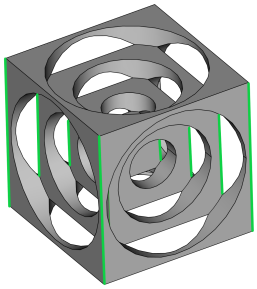
Algebraic operations have an algebraic structure, for example, affine transformations and boolean operations. These operations map directly to operators or methods in the underlying language. A distinct feature of algebraic operations is that these are often total, i.e., they are well-defined for all possible inputs, and are therefore robust. *Query* operations, on the other hand, modify specific features of objects. For example, a chamfer is applied to a specific edge. Therefore, to apply such operations, there is a need of identifying features on which the operation applies (recall that GUI-based tools use name). Programmatic interfaces such as CadQuery (as well as FeatureScript and Scadla) provide a small query language to perform such selections. A query on an object first specifies a type, and then a property. The query returns elements of the specific type which satisfy the property. Coming back to the dowel end-cap example presented in Section 2.1.1, the following code segment (in CadQuery) generates a design that is robust to parameter changes:

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

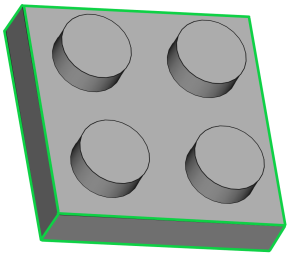
Line 7 performs an algebraic operation, which is robust, even when done via GUI. Lines 4, 5 and 9 perform *query* operations, which cannot be robustly specified via GUI. Line 4 creates a solid cylinder on the maximal face in the Z-axis of the base. Line 5 transforms the solid cylinder into a shell of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

The selection API in CadQuery includes several selection predicates (see Figure 2.4 for some examples). These predicates can be categorized on the basis of whether they depend on intrinsic properties or relative properties over multiple elements, and whether the predicates take parameters. For instance, an intrinsic predicate can select all the edges parallel to the Z-axis (Figure 2.4a), or, non-circular edges (see Figure 2.4b). A relational predicate can select the face(s) with the maximal Z-coordinate (see Figure 2.4c). Note that in addition to the standard axes, the selection predicates can be defined over any arbitrary vector. Another parametric predicate is a bounding box, which selects all the elements within it. Predicates can also be chained (using .), and combined as boolean formulae (using `and`, `or`, and `not`).

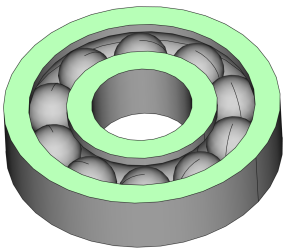
In addition to the predicates we have already seen, there are also predicates for paral-



(a) Edges parallel to the Z-axis (some selections hidden due to occlusion).



(b) Non-circular edges (some selections hidden due to occlusion).



(c) Faces with the maximum Z-coordinate.

Figure 2.4.: Some selection predicates supported by CadQuery.

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

Semantic queries versus direct manipulation

As an alternative mode of selection, writing queries requires thinking semantically. While it is usually harder to write a query than select elements in the GUI, a query carries more meaning. For instance, consider line 9 of the dowel end-cap code presented before. To do the same operation in GUI, users need to select the relevant edge, and choose the fillet operation, as demonstrated in Figure 2.5. We show FreeCAD here, but other popular CAD applications also have similar interfaces. The selection mechanism in GUI is intuitive and quick. However, notice that the GUI in Figure 2.5 uses a name identifier for the selected edge (`Edge10`). This is the source of brittleness. If this identifier changes because the shape is modified, the fillet operation would either fail, or worse, modify the object in some other way (due to unknown ambiguity resolution heuristics).

2.2.2. Towards Interactive Programming for CAD

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

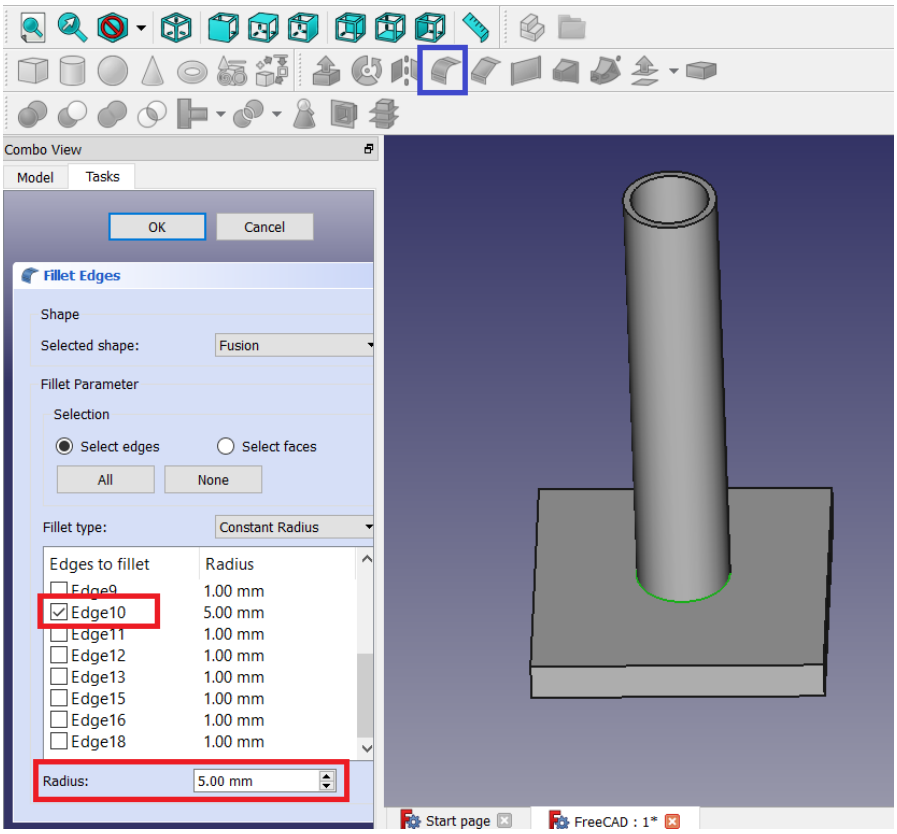
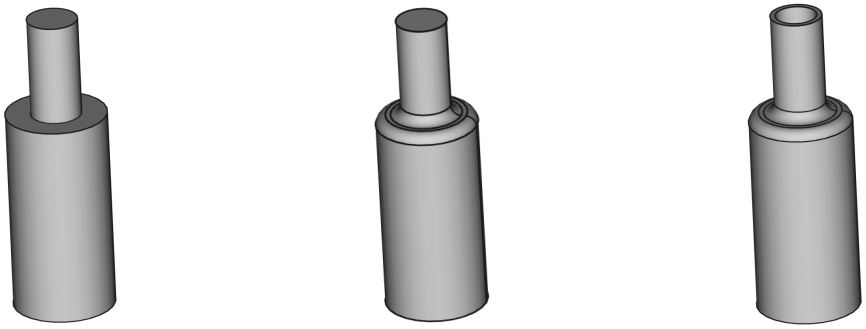


Figure 2.5.: Filleting the edge connecting the rectangular base and the cylindrical tube in FreeCAD. The edge selected in green is to be filleted. This involves selecting the edge and then choosing the 'Fillet' option in the toolbar (marked in blue). The red boxes show the corresponding edge reference and the fillet parameter.



(a) Union of two offsetted cylinders.

(b) Sharp edges at the neck smoothed.

(c) Shell created (final design).

Figure 2.6.: Some intermediary steps for designing a bottle in CAD.

We now present a simple design example to demonstrate the capabilities of our approach. Suppose we want to design a bottle as in Figure 2.6. Designers start with a blank sketch and program. They fill in some environment variables, such as the radius and height of the bottle:

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

They then create a cylinder, which serves as the body of the bottle. This operation can be directly translated to code as a one-to-one mapping from the GUI (first create a circle, and then extrude it):

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

As no user selections are made, this translation is robust. Now that the body is complete, the designers move on to the neck of the bottle, which is another cylinder. However, this new cylinder needs to be created on top of the existing one. In the GUI, users select the face on top of which they want to create the new cylinder. Our system can automatically synthesize the query that selects this face, and performs the relevant operation:

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

The next step is to union the two cylinders together. Designers can perform the operation in the GUI, and the corresponding code can be directly obtained:

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

So far, we have a design as in Figure 2.6a. Next, the designers want to smooth the edges at the neck of the bottle (as in Figure 2.6b). They perform this operation in GUI, and the relevant selection query is automatically synthesized:

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

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

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

This completes the design process and we have the final design as in Figure 2.6c.

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

Using program structure

Consider the example of making a plate of Braille text (Figure 2.7). The following code segment generates this with cylindrical bumps:

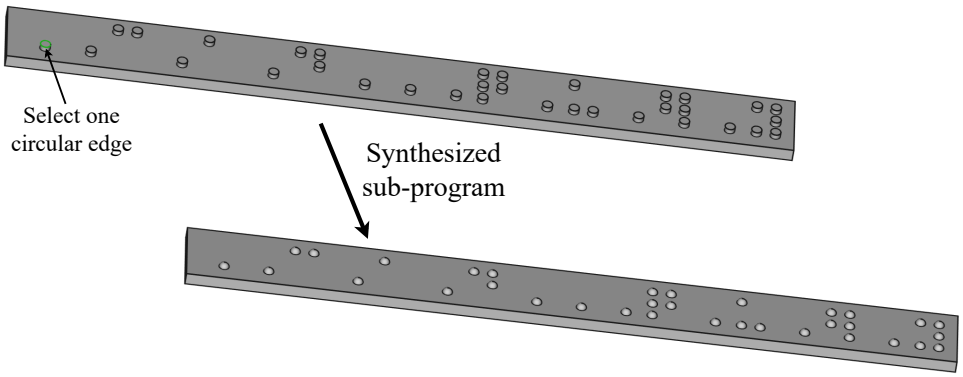


Figure 2.7.: Using the program's structure to synthesize more relevant queries. A program generates a Braille plate with several extruded circles that are created together in a collection. Selecting any one (or more) of these circles would generate a formula for all the circles in this collection. Therefore, the rounding is applied to all the extruded circles to get the final design.

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

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

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

Local modifications and debugging

The way designers typically debug and perform local modifications on their design is by selecting the specific feature of the design in GUI, and analyzing/modifying its attributes. However, this can be considerably more difficult in a programmatic interface. For example, consider a storage box design (Figure 2.8). The programmatic representation (48 lines of code) is parametric, and uses a loop and several if-else blocks. It is difficult to follow the program logic due to interdependencies between several operations. Such use cases can be drastically simplified by our system. As we track how the design changes in each successive line of code, given a specific feature in the design, using reflection, we can identify the line of code responsible for it. Users can then analyze attributes or make

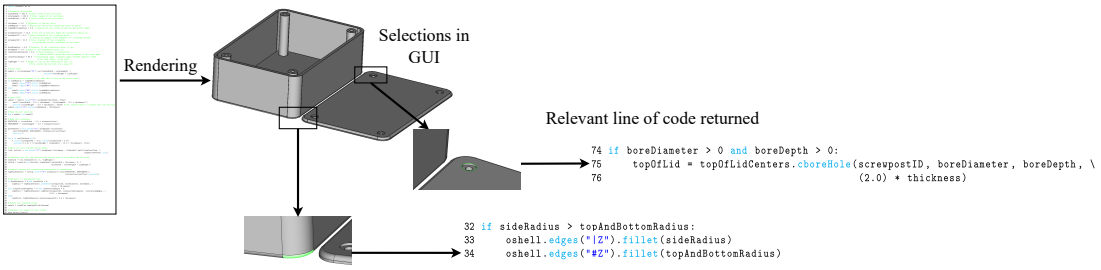


Figure 2.8.: Modification/debugging of a complicated program. As we track intermediary states of the design, selecting an element or feature in the direct manipulation interface takes us to the relevant line of code responsible for it. This aids in local modifications and debugging.

modifications there. Figure 2.8 shows two examples of user selections in the GUI and the relevant line of code returned by our system.

In the context of CAD, two important parts we do not cover in this work are 2D sketching, and algebraic operations and affine transformations. Both of these have been covered in the sketch-n-sketch framework for vector graphics [67, 30], and these techniques can be directly applied to CAD. Our focus is on the synthesis of *queries* from user selections in direct manipulation interfaces. Our implementation is based on the FreeCAD GUI and CadQuery programmatic back-end. We synthesize selection queries using a modified Decision Tree algorithm. Decision Trees are quite efficient at building such formulae. Moreover, due to their white-box nature, it is easy to understand why the synthesis procedure comes up with a certain formula rather than another one (in cases where more than one formula is possible). Our Decision Tree procedure essentially chooses a predicate to add to the formula in a greedy fashion. Doing so incrementally gives us the complete and correct-by-construction formula. That is, applying the resulting formula to our design is sure to lead us to our desired selection. Though the Decision Tree procedure cannot guarantee synthesis of the shortest formula, we find that in practice, these formulae are quite small, readable and quickly computed.

2.3. The Synthesis Framework

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

our method needs to already work on a single example.

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

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

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

2.3.1. Syntax of Synthesized Programs

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

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

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

2.3.2. Synthesis Algorithm

Decision trees are popular in Machine Learning for solving classification and regression tasks. Unlike other techniques like neural networks, Decision Tree learning is a white-box approach. It is possible to understand the logic behind decision procedures of these trees. Moreover, due to their simple design, this logic is also human readable. We use this

ALGORITHM 1: Modified Decision Tree algorithm for synthesizing a query representative of a GUI selection.

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

if $C \subseteq T$ **then**
  return True
else if $C \cap T = \emptyset$ **then**
  return False
else
  Evaluate the relational selection predicates in S on C and store the result in S_{new}.
  Calculate information gain for each selection predicate in S and S_{new}.
  Let $s \in S_{new} \cup S_{curr}$ be the selection predicate with the highest information gain, and $\emptyset \subset s \cap C \subset C$.
  if s *information gain* $< t$ **then**
    Use C and program context to generate a new selection predicate (Section 2.3.2).
    return SynthQuery(C, T, S_{curr}, $S \cup \{s\}$)
  else
    if $s \in S_{new}$ **then**
      $S_{curr} := \left\{ x \mid \begin{matrix} (x \text{ intrinsic } \wedge x \in S_{curr}) \vee \\ (x \text{ relational } \wedge x \in S_{new}) \end{matrix} \right\}$
    $L_T, L_C :=$ SynthQuery($C \cap s$, T, $S_{curr} \setminus \{s\}$, S)
    $R_T, R_C :=$ SynthQuery($C \setminus s$, T, $S_{curr} \setminus \{s\}$, S)
    $L := L_C \,?\, (s.L_T) \;:\; (s \wedge L_T)$
    $R := R_C \,?\, (\neg s.R_T) \;:\; (\neg s \wedge R_T)$
    return $L \vee R, s \in S_{new}$

feature of Decision Trees to synthesize code snippets for selection queries. For the sake of simplicity, we use a selection predicate and the set of elements it selects interchangeably.

Description of the algorithm

Our synthesis algorithm is based on the popular ID3 Decision Tree learning algorithm [120]. Given the relevant top-level object O, we start with all the elements that can possibly be selected, $O = \{o_1, o_2, o_3, \ldots\}$. We also have the set of selected elements we want to derive a selection query for, $T = \{t_1, t_2, t_3, \ldots\} \subseteq O$. We maintain the notion of a current set, C, which is the set we are currently working with in the Decision Tree procedure. In the beginning of the procedure, $C = O$. We then follow the algorithm as in Algorithm 1. First, we check if the current set C is a *base case*. Set C is a base case if $C \cap T = \emptyset$ or if $C \cap T = C$. In either case, we do not need further decision steps, as C

contains only positive or negative examples. If C is not a base case, we perform further decision steps until we reach a base case. At each decision step, we choose the selection predicate with the highest information gain (and that does not lead to selecting C or $\emptyset$).

When the rate of progress decreases, we add a new candidate selection predicate by looking at the program context. The selection predicate can either be from a pre-calculated selector set S_{curr}, or from a newly calculated predicate set S_{new}. Re-calculating (S_{new}) means we combine the predicate with '.' and using a pre-calculated predicate (S_{curr}) means we combine it with '$\wedge$'. The recursive call of the algorithm returns a flag to indicate whether to use '.' or '$\wedge$' when connecting the sub-tree to its parent.

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

Correctness and Completeness

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

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

Practical adjustments for efficiency

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

(i) To re-evaluate selection predicates that depend on the set of features (like maximum and minimum), we need to create a temporary object O_{new} and evaluate the predicates on this to generate S_{new}. This is an expensive process, especially when the set of selection predicates and elements in the object is large. Therefore, we

propose only calculating this if the maximum information gain from predicates in S_{curr} is less than a certain threshold.

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

Generating Selection Predicates

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

Non-parametric predicates

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

Parametric predicates

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

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

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

2.3.3. Querying objects in a loop or collection

Very often, programs operate on collections of objects. An obvious example is the map function, which takes as input a collection of objects and returns a collection with some transformation applied to each element. In CAD, this is fairly common as well (an example was presented in Section 2.2.2). As we maintain a snapshot of program state at each line number, given a set of elements selected using direct manipulation, we check if the selected element(s) are part of a collection. If this is the case, we generate a query fitting the whole collection. Algorithm 2 shows how this is done.

ALGORITHM 2: Synthesizing queries on collections

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

2.4. Evaluation

We now present implementation details, and provide experimental evidence to demonstrate the applicability of our approach to modern parametric CAD workflows. Section 2.4.1 provides implementation details. Section 2.4.2 provides evidence of our approach working well in practice. In Section 2.4.3, we show that even when dealing with complex designs that do not have a programmatic representation, our technique can synthesize selection queries fairly quickly. Section 2.4.4 presents experiments demonstrating the utility of range queries and support for design modifications. Finally, in Section 2.4.5, we present a user study that evaluates the ease of use of our proposed interface.

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

2.4.1. Implementation

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

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

Programming interface The programming interface is unaware of the GUI interface except for its use as the output device (as is usually the case). However, we instrument the program so as to track intermediate states of the design with the help of Python's reflection API (`inspect` module). Tracking intermediate states helps understanding the control structure of the program, as well as identifying which elements were created together, for example in a loop or in a collection (for example, see Figure 2.7 in Section 2.2.2). Moreover, it helps us determine which line of code led to a particular selected element being modified. This enables interactive local modifications and debug features (for example, see Figure 2.8 in Section 2.2.2). In addition to tracking the state of the design and control flow, we also maintain a list of variables in scope. This is done so as to include these as parameters in the synthesized queries, especially range-based queries.

Initial predicates in the synthesis procedure The choice of initial set of selection predicates is important for generating quick and human-readable queries. The following are the set of initial selection predicates we use in our experiments (and that we use as default):

(i) Intrinsic: parallelism and orthogonality to the three coordinate axes (X-, Y-, and Z-axis), centre in the positive or negative direction for each coordinate axis, and the type of edge (straight line, circle or arc), or face (flat plane, cylindrical or spherical).

(ii) Relative: maximal and minimal elements in each coordinate axis.

All of these predicates are available in CadQuery, and our synthesized queries can therefore directly be used in a CadQuery program. This initial set of selection predicates is kept the same throughout the various case studies.

The threshold on the information gain is set to 0 for our case studies. This guarantees the synthesis of a selection query for any possible user selection. However, this also has the side-effect of sometimes leading to very long selection queries. An example of this is presented later, in Section 2.4.4.

2.4.2. Synthesis robustness and runtime

We now evaluate our technique on several important metrics like robustness of the synthesized queries, and run-time of the algorithm. The evaluation is based on a wide variety of designs, both simple and complex, and encompasses a large variety of application areas. There is a need for ground truth to appropriately assess the quality of our synthesized queries. Our examples are therefore based on designs whose source code is also available. We discuss two sets of examples: (a) experiments on CadQuery samples, where we use example programs from the CadQuery public repository [116],

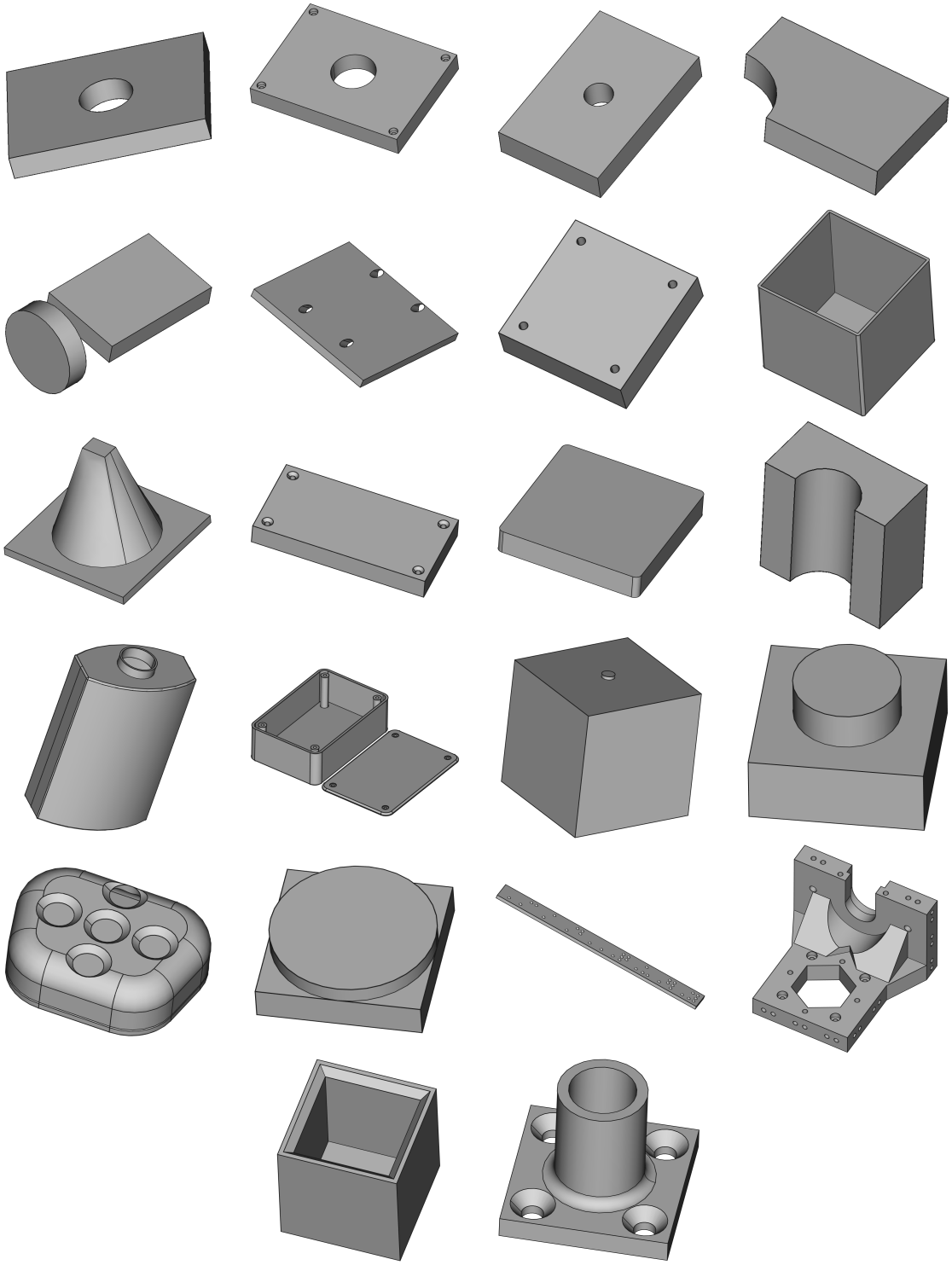


Figure 2.9.: CadQuery designs (default parameters).

and (b) experiments on Thingiverse samples, where we use some parametric designs available on Thingiverse's customizable section [96].

CadQuery examples

There are 22 designs in CadQuery's repository that use selection queries. We include all of them (see Figure 2.9 for a snapshot of the designs) for this experiment. The aim is to evaluate the run-time of our system's synthesis procedure, as well as examine whether the synthesized queries are correct (or, as intended by the original authors of the design). The experimental procedure is as follows:

(i) For each CadQuery example, we start with a blank program, and copy the example until a selection query occurs.

(ii) We use the FreeCAD GUI to display the output until this line.

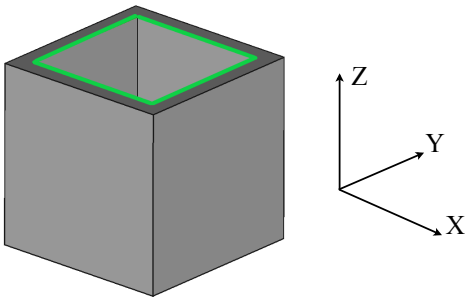
(iii) In the FreeCAD GUI, we manually select the elements selected by the ground truth query.

(iv) We append the query returned by our automatic synthesis procedure to the program, and carry on until the next selection query, or the end of the design.

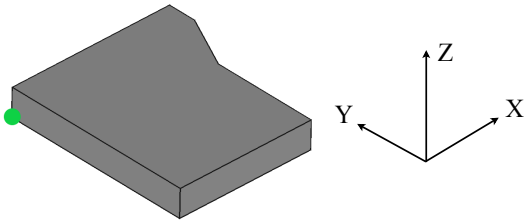
In Table 2.1, we report run-time and query size (number of predicates in the query) for these examples. The number of vertices, edges and faces in the examples, as well as lines of code (LOC) are also reported to get a better sense of the complexity of the underlying designs. To evaluate the correctness of the synthesized queries, we compare the synthesized queries to the ground truth queries. Two corresponding queries can either be equal, logically equivalent or different. Equality and logical equivalence ensure correctness of the synthesized query. However, if the synthesized query is different from the ground truth query, it does not necessarily mean it is incorrect (existence of multiple semantically equivalent queries). For synthesized queries that are different from the ground truth query, we randomly sample over the programs' parameter space and compare the resulting meshes. We sample parameters randomly as automatically finding 'intended' parameter values is known to be difficult [71]. We compare meshes by calculating the Hausdorff distance [124] between them using MeshLab [31]. If the resulting meshes are the same over 50 random samples, we mark these queries as *experimentally equivalent*. Figure 2.10 provides a visual example of the difference between logically and experimentally equivalent queries. We report the results on our synthesized queries in Table 2.1. Table 2.3 gives specific details on each example in the experiment.

Thingiverse examples

The set of ground truth examples we use here are obtained from Thingiverse's section of customizable designs. We chose 12 customizable designs representative of different application areas and of varying complexity. Figure 2.11 provides a snapshot of the chosen designs. The examples in Thingiverse's customizable section are constructed using



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



(b) Experimentally equivalent: we synthesized ‘‘(>Y and (not >X and (not >Z)))’’ vs. ground truth ‘‘(<Z) . (>Y) . (<X)’’.

Figure 2.10.: Examples of logically and experimentally equivalent queries (edges/vertex in green selected).

Table 2.1.: Analysis of query size, synthesis run-time and robustness for the CadQuery examples.

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

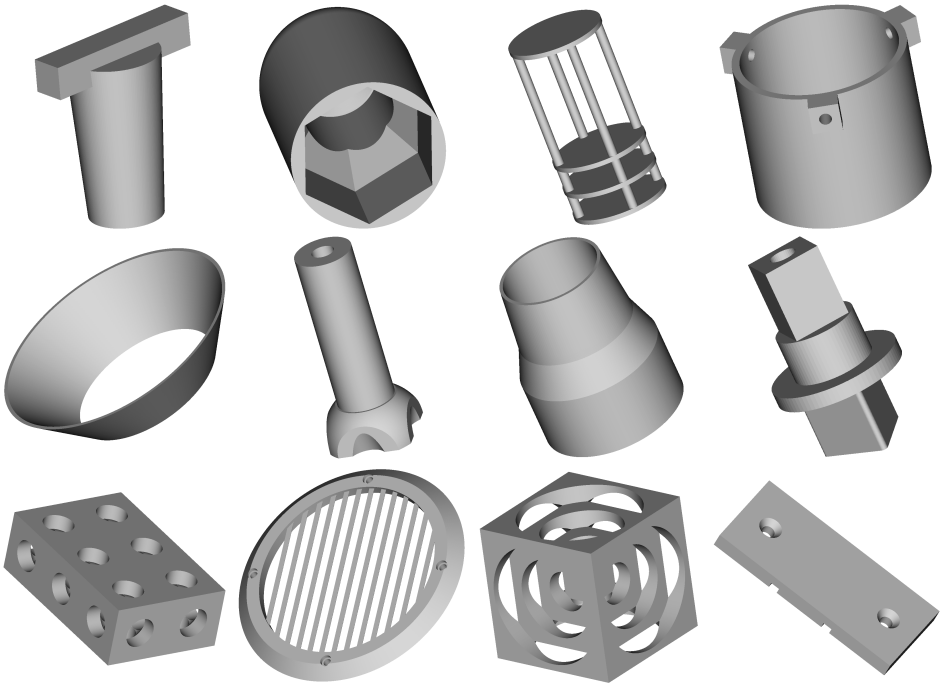


Figure 2.11.: THINGIVERSE designs (default parameters).

Table 2.2.: Analysis of query size and synthesis run-time for the THINGIVERSE examples.

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

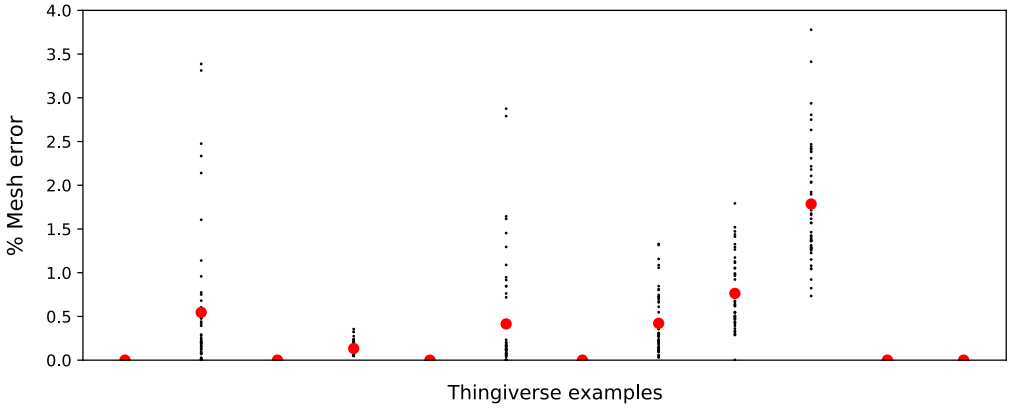
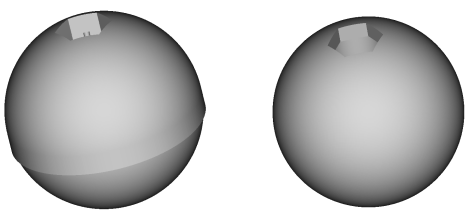


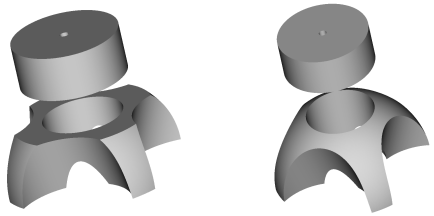
Figure 2.12.: % Mesh error on the THINGIVERSE examples (total 51 synthesized queries), enumerated on the X-axis (ordered as in Figure 2.11). Black dots represent mesh errors of each random sample. Red dots represent the average mesh error of each example.

OPENSCAD [80] and have a programmatic representation in CSG. For each example, we start with a blank program, and re-construct the design in our system. Note that a direct translation of the ground truth program is not possible due to a difference in the underlying CAD representation and available operations. We then use the same procedure as in Section 2.4.2, and synthesize a selection query wherever possible.

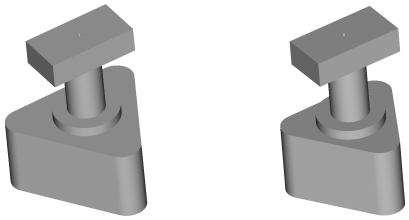
We report run-times of the synthesis process along with the complexity of the designs in Table 2.2. As OPENSCAD does not have a query language, an analysis of correctness of the synthesized queries is done using random sampling of the parameter space of the ground truth design and comparing the resulting mesh to the one generated using synthesized selectors (50 random samples). Like before, we compare meshes by calculating the Hausdorff distance between them. As parameter values often change the size of the overall design, we divide the Hausdorff distance by the length of the diagonal of the bounding box of the ground truth model. This is our mesh error metric. We plot errors for each example in Figure 2.12. The small error values here indicate that the synthesized selectors are robust, and there are no unexpected side-effects. Indeed, most of the errors here are caused due to the difference in the underlying CAD libraries. This can be confirmed by visually inspecting the meshes with the most error ($> 1\%$) in Figure 2.13.



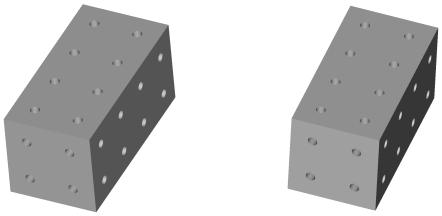
(a) Bolt cap. Mesh error 3.39 %



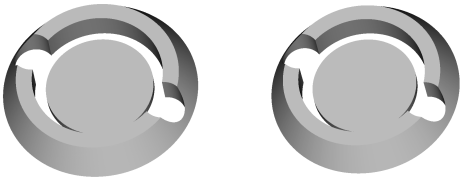
(b) GoPro Screw. Mesh error 2.88 %



(c) Lock Shaft. Mesh error 1.33 %



(d) Setup block. Mesh error 1.79 %



(e) Speaker Grill. Mesh error 3.78 %

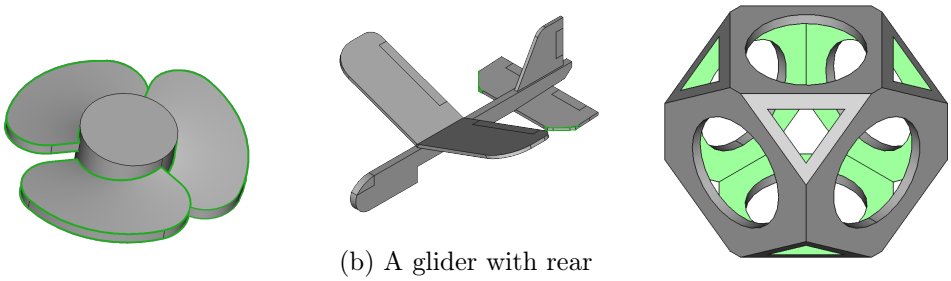
Figure 2.13.: THINGIVERSE examples with the highest mesh errors (our mesh vs. ground truth mesh on the right). Clearly, the differences are either negligible, or very small.

Table 2.3.: A summary of various CadQuery examples run on our system. We report the complexity of the model, the number of queries in the example, the number which were equal/experimentally equal, the average size of each synthesized query, and the time it took to synthesize it.

Model	Vertices	Edges	Faces	# Queries	# Equal	# Exp. Eq.	Avg. size	Avg. time (s.)
Block with Bored Center Hole	10	15	7	1	1	0	1	0.001
Pillow Block With Counterbored Holes	26	39	19	2	2	0	1	0.001
Creating Workplanes on Faces	10	15	7	1	1	0	1	0.001
Locating a Workplane on a Vertex	10	15	7	1	0	1	3	0.001
Offset Workplanes	10	15	9	1	1	0	1	0.001
Rotated Workplanes	20	30	10	1	1	0	1	0.0001
Using Construction Geometry	16	24	10	1	1	0	1	0.001
Shelling to Create Thin Features	8	12	6	1	0	1	1	0.001
Lofts	18	27	12	1	1	0	6	0.049
Counter Sunk Holes	20	32	14	1	1	0	2	0.001
Rounding Corners with Fillets	8	12	6	1	1	0	3	0.001
Splitting an Object	12	18	8	2	2	0	1	0.001
Classic OCC Bottle	14	21	10	2	2	0	3	0.001
Parametric Enclosure Filleting	124	192	90	7	6	1	1	0.089
FreeCAD Solids as CQ Objects	10	15	8	1	1	0	1	0.001
Lego Brick	16	24	11	3	3	0	1	0.001
Remote Enclosure	64	112	51	5	4	1	1	0.001
Numpy	13	22	11	1	1	0	1	0.001
Braille	110	182	108	2	2	0	1	0.001
3D Printer Extruder Support	172	252	113	13	7	5	1	0.014
Shelled Cube Inside Chamfer	16	24	11	2	2	0	2	0.001
Reinforce Junction Using Fillet	25	40	18	3	3	0	1	0.015

Table 2.4.: A summary of various THINGIVERSE examples run on our system. We report the complexity of the model, the number of queries required to be synthesized, the average size of each synthesized query, the time it took to synthesize it, and error metrics on the meshes generated using random sampling.

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



(a) A propeller with edges of the blades selected.

(b) A glider with rear wings' front edges selected.

(c) A whiffle ball with inner faces selected.

Figure 2.14.: Some complex designs without a programmatic representation. We synthesized queries for the elements colored in green.

Results

Results from these two sets of examples show that our technique is useful in a practical interactive programming setting. We find that the synthesis procedure is quick to synthesize queries. In fact, the longest running time for these examples is less than a tenth of a second, which is quite acceptable in an interactive setting. We are also able to synthesize a wide-variety of queries, going from a size of 1, up to a size of 7 in these examples. The synthesized queries are also robust. We find that our technique often synthesizes queries that users themselves would come up with, or something equivalent.

2.4.3. Synthesis Scalability

The queries generated in the previous experiments are usually small as we utilize the underlying programmatic representation of the design. We now show that our algorithm can also scale to more complicated and artistic designs that do not have an underlying programmatic representation. A typical use case for this is when designers only have access to the final object, and want to make robust modifications on these. The models for this case study are taken from the public repository of FreeCAD's official tutorials [22]. The models we used and the selections for which we synthesize queries are depicted in Figure 2.14. The results of the corresponding synthesis procedure are summarized in Table 2.5. Though it is not surprising that the query sizes are quite large for these examples, we find that our algorithm can still cope with this in a reasonable amount of time.

2.4.4. Feature specific experiments

We now present two small experiments, each demonstrating the utility of a specific feature of our technique that could not be covered in earlier experiments.

Table 2.5.: Analysis of query size and synthesis run-time on models without a programmatic representation.

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

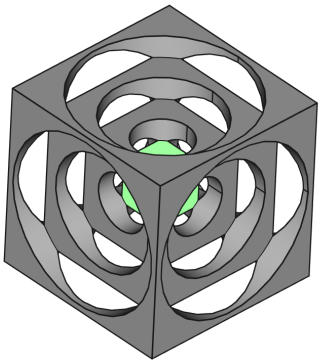


Figure 2.15.: A Turner's Cube with the inner-most faces selected.

Range queries

The following example demonstrates the utility of range queries, specifically the parametric bounding box selector. Consider the model of a Turner's cube as in Figure 2.15. For the sake of this example, let us assume that this model does not have a programmatic representation. If we start with the usual selection predicates and 0 threshold on the information gain, we get an extremely large selection query with 56 selection predicates. Due to the layered structure of the design, our algorithm has a hard time coming up with a selector for the inner faces. It tries to select the inner faces by removing faces from the top-level object layer-by-layer. A remedy to this can be a concise parametric selector.

For example, if the inner-most cube's edge length, `s` is made available to the algorithm, the synthesized query is: `faces(BoxSelector((0,0,0), (s,s,s)))`. This is an example of a range query in which all faces inside the bounding box defined by the range are selected. Not only is this query shorter, but it is also synthesized quicker: it takes only 0.05 seconds to synthesize, as compared to 23.29 seconds for the larger query.

Local design modifications

In Section 2.2.2, we discussed how our system can be used to discover which line of code is responsible for a particular feature in the design. We now demonstrate a slightly different experiment, based on the same example, wherein instead of just debugging or changing of some parameters, we change the design itself. We use the same example as before, i.e., a storage box. Figure 2.8 shows our storage box. Let us remove the fillet from the bottom

Table 2.6.: User performance on Programmatic only vs. Programmatic + GUI (our) interface. We report the percentage of queries attempted and correctness over all participants.

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

part of this box. To do this, we can select any of the rounded faces (or edges) that we wish to modify (see Figure 2.8), and ask our system to return the line of code responsible for this. Our system returns the following line of code:

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

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

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

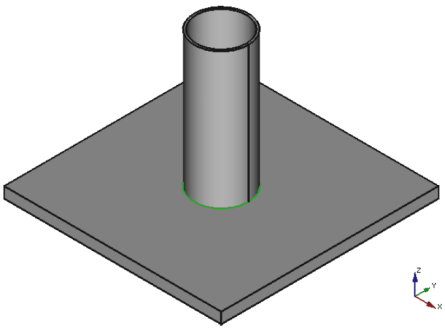
Replacing this with the previous line of code gives us the requisite modification.

2.4.5. User Study

To evaluate the usability of our proposed interface, we conducted a user study with 6 participants (all male, 20 - 35 years of age). They had varying degrees of experience with CAD, with some being beginners with less than 1 year of experience, some with more than 3 years of experience, and some who fell in-between. Each study was approximately 60 minutes long: 20 minutes for a short tutorial and feedback, 20 minutes using a Programmatic system, and 20 minutes using Programming + GUI (our system). Users were asked to write queries for 14 selected designs (total 49 queries), which contained CadQuery examples after removing redundant designs (designs with only one query, which occurs in a similar way in other designs), and adding the dowel-end cap and bottle examples presented earlier in this Chapter. The interface they were presented with is depicted in Figure 2.16. Participants were counter-balanced between doing the Programmatic interface and the Programatic + GUI interface first. All the designs that they had to complete were also counter-balanced and shuffled randomly. Participants were free to do the designs in any order or skip some if they did not want to finish them. Participants were not warned when their selection in the GUI or the query they wrote was incorrect, but were generally aided through questions about the interface and any technical queries they had. In the end, they filled-out a post study questionnaire, where they were asked questions about their experience on the two interfaces. The aim of the study was to collect quantifiable user data on efficiency and accuracy of our interface in comparison to baseline (programming alone).

T_14_c

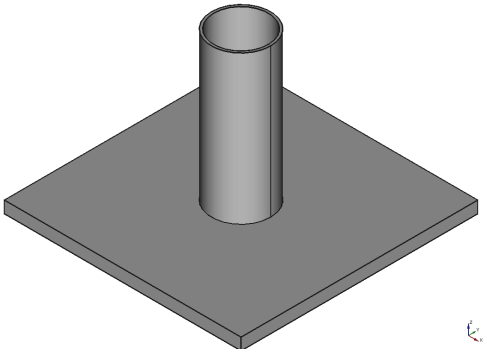
Solve the selection query at line 30.



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

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

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

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

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

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

result = base.union(tube)

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

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

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

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



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

Figure 2.16.: Participants were presented 2 or 3 tabs depending on whether they were using the Programmatic or the Programmatic + GUI interface.

Table 2.7.: A summary of qualitative opinions from the user study, reported on the Likert scale (1- Strongly agree, 5- Strongly disagree).

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

In Table 2.6 we report how fast and how accurate the participants were in the two interfaces. Using the GUI to synthesize programmatic queries significantly improved user speed, going from only attempting 77.5% of design tasks on average, to 94% with our technique. They were also more accurate, going up from 84.5% accuracy to 98.3% accuracy. In fact, there were just two instances in which interface did not give the correct result: one, where the user gave up because they needed to select several edges (whereas the query was relatively straightforward), and second, where the user selected the wrong element in a symmetric design.

Participants also filled a post study questionnaire. Table 2.7 aggregates their answers to questions based on a Likert scale . The questionnaire also asked which interface the participants preferred. Everyone preferred the GUI + Programmatic interface. When asked why they preferred this interface, the most recurring opinion was that using the GUI was faster. The participants especially preferred GUI when the selections were non-standard (more than one selection predicate, or when the object was not parallelepiped).

2.5. Conclusion

We identified bridging GUI and programming as a possible solution to getting the best of both worlds for parametric CAD, i.e., intuitiveness and ease of use of GUI, and, robustness, generalizability and modularity of programming. To this end, we presented a Decision Tree based approach that synthesizes semantics of selections made in a GUI (or direct manipulation) interface. We demonstrated how the queries thus generated can be used for interactive programming of CAD and that our technique works on many different examples. Our proposed technique is quick, and the queries we synthesize are robust. A user study confirms that our interface is faster, less error-prone, and generally preferable over plain programming.

3

Synthesis of Parameter Constraints

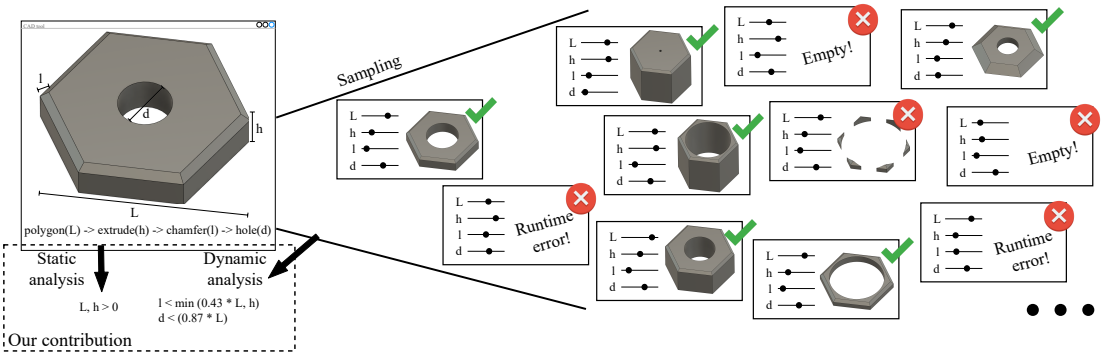


Figure 3.1.: We propose a technique for the automatic synthesis of constraints to CAD parameters. Using a mix of static and dynamic program analysis, we restrict the parameter space of designs to only those configurations that produce valid final objects.

3.1. Introduction

Parametric design is a popular design methodology in which designers encode their *design intent* by specifying the sequence of operations in the design. This enables end-users to change parameters of the design, which after a re-evaluation of the sequence of operations, results in different variations of final objects. Parametric design tools such as Autodesk Fusion 360, Solidworks, PTC Creo, OpenSCAD, and FreeCAD are ubiquitous in the industry and 'maker' community.

Moreover, websites such as Thingiverse, where users share, remix, and customize parametric designs, are extremely popular. A serious limitation of current customization pipelines is that the relationship between CAD parameters is often unknown. In practice, only a small subset of parameter values lead to valid final objects. Unfortunately, this information is usually not conveyed in shared designs. Inferring this information automatically is difficult because designs can have many parameters, each of which

influences the validity of the final object. This complexity grows as designs involve more CAD operations and parameters. At the same time, constraints on design parameters are extremely valuable to end-users. It provides them a high-level perspective on how the parameters interact, and guides them towards valid final objects.

THINGIVERSE, a popular online repository of CAD projects encourages designers to provide information on the range of supported parameter values. To illustrate how valuable this information is, we did a small experiment. On May 7^{th}, 2021, we examined the front page of THINGIVERSE's parametric designs. Sorting by the newest designs, we observed that only 7 of the top 20 designs had (partial or full) information on constraints to the design parameters. Then, sorting by popularity, we observed that significantly more, 13 of the top 20 designs had this information. Though our sample set is small, it paints a believable picture: coming up with constraints on parameters of designs is difficult, and most designers therefore do not provide them. At the same time, end-users find these constraints extremely useful, as it provides them with high-level context on the design, as well as the variability of final objects it supports.

In many research and industrial use cases, parameters of designs are sampled to find optimal (according to metrics like weight, strength, stability, etc.) or otherwise unique final objects [133, 95]. Depending on the design under test, a significant proportion of these samples lead to invalid final objects, and are therefore wasted. This can be drastically improved if information on the constraints to design parameters is available.

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

After a static analysis pass, if there are still parameters without known constraints, we move to *dynamic analysis*, i.e., we try to infer constraints based on evidence from evaluating the design on many concrete parameter values. Such a two-pronged approach (static & dynamic analysis) has been successful at finding bugs and program invariants in traditional computer programs (for example, see [45]). We adapt these ideas for parametric CAD, first by defining what it means for a CAD operation to *fail*, or for an object to be *invalid*. Then, we describe a few inference rules that help synthesize some parameter constraints statically. Finally, we present a novel guess-and-check algorithm for dynamically synthesizing constraints.

Let us present a concrete example of our technique in action. Consider the design in Figure 3.1. The design consists of first making a hexagon of diameter `L`. Then, this is extruded to a height `h`. Next, the 6 top-most edges are chamfered using a length of `l`. Finally, a hole of diameter `d` is drilled on the top-most face.

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

Our dynamic analysis algorithm samples the design over many parameter values, and tries to construct and fit hypotheses based on the observed runs. Our hypothesis generator proposes hypotheses of increasing complexity. To check whether the hypothesis fits, and to synthesize a precise constraint, we use mixed integer linear programming. For our current example, this technique synthesizes correct constraints for both, `l` and `d`: `l < min (0.43*L, h)`, and `d < 0.87*L`. Our automated technique finds these constraints in just 48 seconds. Clearly, coming up with these constraints by hand would be challenging.

3.2. Contributions

The main contributions of this Chapter are:

(i) We identify uncovering implicit assumptions in code statically, followed by learning constraints via concrete evaluations of the design as a useful strategy for synthesizing constraints.

(ii) We propose static inference rules, as well as a dynamic hypothesis generation and checking strategy for learning CAD parameter constraints.

(iii) We evaluate our system on a variety of publicly available designs and demonstrate that our technique synthesizes constraints efficiently and accurately.

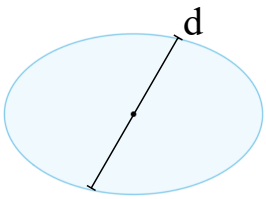
3.3. Preliminaries and Overview

We now provide an introduction to CAD representations (specifically B-rep), and a short background on program analysis. Then, we present an overview of our proposed technique.

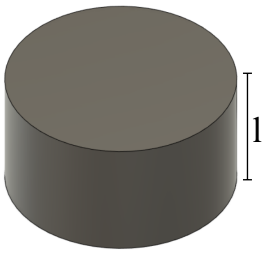
3.3.1. B-rep, CAD Operations, and Constraints

Simple representations based on Constructive Solid Geometry (CSG) [83] are still common in some CAD libraries (e.g. OpenSCAD) and web portals (e.g. Thingiverse). Most modern tools, however, use Boundary Representation (B-rep) [140]. B-rep offers a rich collection of high-level operations to create and modify 3D shapes, and we already introduced it in Section 2.2.1. Figure 3.2 provides a snapshot of some common operations available in B-rep, and the parameters they take.

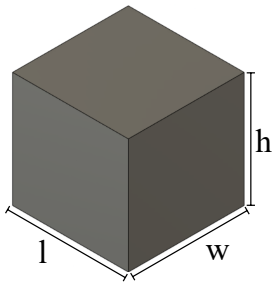
The most popular open-source implementation of B-rep is Open CASCADE [113]. We use the CadQuery [115] interface, based on top of the Python embedding of Open



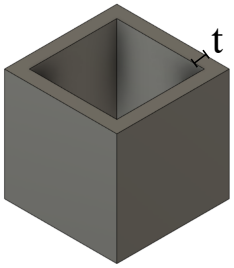
(a) Operation `circle` with param [d: diameter of the circle].



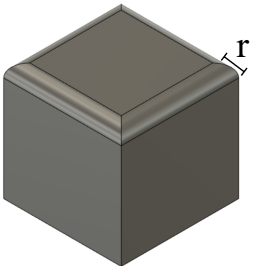
(b) Operation `extrude` with param [l: length of the extrude].



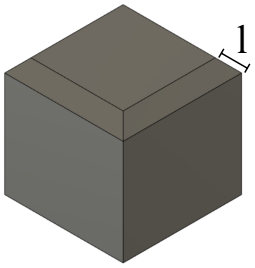
(c) Opreation `box` with params [l: length of box, w: width, h: height].



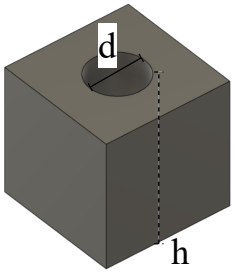
(d) Operation `shell` with param [t: thickness of the shell].



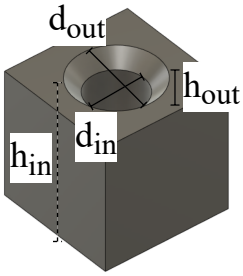
(e) Operation `fillet` with param [r: fillet radius].



(f) Operation `chamfer` with param [l: length of the chamfer]



(g) Operation `hole` with params [d: diameter of the hole, h: depth (optional)].



(h) Operation `countersinkHole` with params [d_{in}: diameter of the hole, d_{out}: diameter of the countersunk hole, h_{out}: depth of the countersink, h_{in}: depth of the hole (optional)].

Figure 3.2.: Some common CAD operations, and the parameters they take.

CASCADE. Though the CAD interface we use here is text-based, this is not a limiting assumption. Indeed, any CAD project can be translated to code, and most CAD tools support text-based macros as an alternative to GUI.

Parametric CAD requires designers to think somewhat abstractly about the design. The challenge comes from the fact that designers do not know what concrete values their design parameters will take. An important tool in improving the productivity of parametric design has been geometric constraints [144, 24]. Most major CAD tools support constrained sketching. Designers can, for example, specify if certain edges need to be of equal length, or parallel or orthogonal. The ability to do this, however, does not extend to high-level domain specific CAD operations such as fillet, shell, counterbore hole, etc. These operations often fail, or result in an invalid design when passed an unexpected parameter value. Naturally, as designs get larger, with more and more operations involved, it becomes increasingly difficult for designers (and consequently, for end-users) to understand how the design parameters should be constrained, i.e., how to *minimally restrict* the parameter space, while still avoiding invalid configurations. The focus of our work is on the automatic synthesis of such constraints.

3.3.2. Validity of Operations

Unique parameter values to a design result in unique final objects. As the number of operations (and correspondingly, the number of parameters) grows, the choice of values grows exponentially. However, many combinations of parameter values lead to *invalid* designs. We now elaborate on what this means. Abstractly, invalidity can be defined as not useful, or bad. Concretely, we define final objects, and parameter configurations that evaluate them as *invalid*, if either:

(i) an error during evaluation occurs,

(ii) the final (or an intermediary) object is empty,

(iii) the final (or an intermediary) object has unconnected components, or

(iv) an operation specific assumption is not satisfied.

Let us look at these criteria one by one. The first one is relatively straightforward. Parameter configurations leading to runtime errors are obviously invalid. In addition to runtime errors, operations can also be classified as failing in some other situations. Operations fail when they return a non-manifold object, or when the topology of the returned object is broken. Such errors can manifest in many CAD tools, especially those based on Open CASCADE. We therefore check for such errors at the end of each operation.

The next two criteria are not *errors* in the traditional sense of the word. They capture implicit assumptions of designers and end-users. It is fair to assume that users do not want an empty object. Moreover, the object must be a connected entity. Objects become empty or unconnected when, due to a CAD operation, an intermediary object cuts the main object more than expected. Unconnected objects are unwanted, as they have lost

the topology designers intended for them. They are segmented into multiple pieces, and the object that was intended to be one entity, is broken-up. Our criteria are not just restricted to the final object, but extend to intermediate objects in the design as well. This is because these are implicit assumptions for each CAD operation. For example, if in an intermediary step the designer wants object A to cut object B and return object C, it is fair to assume that neither of these objects is expected to be empty. Moreover, if these criteria are not fulfilled at the end of one operation, they are unlikely to be fulfilled later on. In the fields of program analysis and debugging, this is called *root cause analysis*, i.e., finding and blaming the operation that leads to an error.

The final criterion ensures that certain implicit assumptions of designers when performing operations are satisfied. For example, if a designer chooses a vertex to drill a hole on an intermediary object, the implicit assumption is that this vertex must lie on one of the faces of the object.

3.3.3. Program Analysis for CAD

Program analysis is an active area of research in the fields of Program Languages and Software Engineering. It involves examining programs so as to discover interesting properties of the code [109]. This has enabled automated techniques for detecting bugs and vulnerabilities in code, and even verification of computer programs. We propose applying ideas from the field of program analysis to CAD. In standard programming, it is common to have a set of preconditions, say pre, and a set of postconditions, say $post$, such that for a given operation op, if the pre-conditions hold, then after the operation the post-conditions also hold. This is usually written in the form of a Hoare triple [70]: $\{pre\}op\{post\}$.

In our setting, the post-conditions are the validity properties presented in the previous section (Section 3.3.2). Our aim is to synthesize the pre-conditions. In fact, we want to synthesize the *weakest* pre-condition pre, which means, $\forall pre'.\, \{pre'\}op\{post\} \Rightarrow (pre' \Rightarrow pre)$. All parameter configurations that satisfy our constraints lead to valid designs, and those that do not, result in invalid designs. We focus our explanation on specific operations and consider the operation's input to be parameters. However, in general, these can be complex expressions computed from the design's input parameters. To handle such cases, we can propagate the constraints to the input [38].

Our setting is a little different from the traditional program analysis set-up. Analytical solutions are not possible due to the complexity of CAD operations and topological structures. We therefore use a mix of custom *white-box* and *black-box* strategies. White-box strategies involve looking at code and making inferences based on it. These already provide some context on the parameters involved. Black-box strategies (sampling in our case) do not care about the underlying code, and can help come up with useful inferences, even when the underlying code is large and complex. This is certainly the case for most CAD operations. Program analysis techniques are typically divided into static analysis and dynamic analysis. Our method uses a mix of both, and we now provide an overview of both these parts.

Table 3.1.: Static constraints that *must* hold for valid designs. `geom` represents any operation that makes new geometry, like `circle`, `polygon`, `sphere`, `box`, etc. `cskHole` stands for `countersinkHole`, and `cboreHole` stands for `counterboreHole`.

Operation	Constraints
`geom(p,...)`	`p,... > 0`
`extrude(h)`	`h > 0`
`offset(h)...loft()`	`h > 0`
`union(A, B)`	`Params of geom of A and B > 0`
`cskHole(d_in, d_out, h_out, h_in)`	`d_in < d_out and h_out < h_in`
`cboreHole(d_in, d_out, h_out, h_in)`	`d_in < d_out and h_out < h_in`

Static Analysis

The idea of static analysis is to capture the designer's intent by looking at the way certain operations and parameters are used, or meant to be used. This is done without executing the design on any concrete parameter value (hence the term *static* analysis). Just looking at a CAD sequence of operations, we can make several inferences about the parameters involved. We summarize these inferences in Table 3.1.

Dynamic Analysis

Dynamic analysis requires evaluating the design on many concrete parameter values. This is effective at revealing dependencies between the parameter values and their validity. CAD operations already provide some hints to what their constraints may look like. For example, consider the operation of boring a hole. It is clear that the hole diameter cannot be arbitrarily large. After some limit, the hole will start fragmenting the base object, or give an empty result. As the base object can be arbitrarily complex, and depend on some other parameter values, we do not know what this limit is. Our aim, through many concrete executions, is to learn this expression. Moreover, we need not sample blindly. We know that no matter what the exact constraint to the hole diameter is, the hole diameter *must* be less than the length of the diagonal of the bounding box of the base object.

In Table 3.2, we present some *rough* constraints. The exact expressions (`expr`) of these rough constraints need to be learnt. Not only are the CAD operations here complex, but they often also operate on complex topologies. Therefore, static analysis cannot evaluate these. We need concrete runs of the design on many different parameter values to learn them.

On top of these rules, we use some practical strategies to synthesize constraints more efficiently. We now describe some of these. For operations such as `hole`, if constraints for the optional parameter (depth of the hole) also need to be synthesized, we set the depth to a value `> BoundingBox().Diagonal`. The constraint synthesized for the depth is `> 0`, and by setting this to a high value, we ensure that samples that fail, fail because of the diameter of the hole. This is also in line with how the `hole` operation executes in the

Table 3.2.: Rough constraints that need to be discovered via dynamic analysis.

Operation	Constraints
`fillet(r)`	`r < expr < BoundingBox().Diagonal`
`chamfer(l)`	`l < expr < BoundingBox().Diagonal`
`shell(t)`	`t > 0 or t > -expr; expr < BoundingBox().Diagonal`
`hole(d)`	`d < expr < BoundingBox().Diagonal`
`vert(o)`	`Vertex lies on a face`
`cskHole(d_in,_,_,_)`	`d_in < expr < BoundingBox().Diagonal`
`cboreHole(d_in,_,_,_)`	`d_in < expr < BoundingBox().Diagonal`
`cut(A,B)`	`Params of geom of B < expr < A.BoundingBox().Diagonal`

absence of a specified depth. Next, because we do not support complex curves, the upper limit for parameters to `fillet` and `chamfer` can be reduced to `BoundingBox().Diagonal / 2`. This does not affect the quality of our synthesized constraints. However we can save a few sampling cycles, as for the operations we support, this is a tighter upper bound.

3.3.4. Learning Constraints

The static analysis already gives us some constraints on parameter values. The dynamic analysis involves many `expr`, which need to be learnt. Our choice of the synthesis technique is driven by the following requirements:

(i) We want the synthesized constraints to be human-readable. This is important for the constraints to make sense to designers, and to end-users who customize and 3D-print results.

(ii) We want to synthesize accurate constraints. Over-constraining the parameter space can be detrimental to the performance of optimization and generative techniques. Under-constraining can be annoying for human end-users.

(iii) We want the technique to make inferences from just a few samples, and have a small runtime overhead.

We employ an enumerative approach for the synthesis of the missing (or rough) constraints. This involves starting with short hypotheses, and then moving to more complex hypotheses if these do not fit. The hypotheses come from domain-specific knowledge of how constraints in CAD typically look like. Each hypothesis is formulated as a linear programming problem. If a solution to the problem is found, then the hypothesis is a possible constraint. If no feasible solution is found, then the hypothesis is rejected. Incrementally, over many iterations and with many runs of the CAD representation, we can settle-in on a hypothesis with a feasible solution, and fine-tune it.

3.4. Framework

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

3.4.1. Validity of CAD Operations

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

In standard programming, it is common to have a set of preconditions, say *pre*, and a set of postconditions, say *post*, such that for a given operation *op*, if the pre-conditions hold, then after the operation the post-conditions also hold. This is usually written in the form of a Hoare triple [70]: $\{pre\}op\{post\}$. In our setting, the post-conditions are the validity conditions as presented before. Our aim is to synthesize the *weakest* pre-condition *pre*, which means, $\forall pre'.\{pre'\}op\{post\} \Rightarrow (pre' \Rightarrow pre)$. All parameter configurations that satisfy our constraints lead to valid designs, and those that do not, result in invalid designs.

3.4.2. Static Rules

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

$$\frac{p_i, p_{ii}, p_{iii}, ... > 0}{CreateGeometry(p_i, p_{ii}, p_{iii}, ...)} \qquad \frac{d_{in} \leq d_{out}, \quad h_{out} \leq h_{in}}{CounterSink/BoreHole(d_{in}, d_{out}, h_{out}, h_{in})}$$

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

The first rule captures CAD operations that are responsible for creating new geometry (excluding geometry that is later used for a cut or difference operation). Operations such as creating a `circle`, `box`, `extrude`, etc. are constrained using this rule, so are intermediary operations such as `offset`s, which later create geometry using `loft`s, for example. The second rule captures easily encoded constraints that must hold for these operations to succeed. The last two rules capture rough constraints. These rough constraints can be used later by the dynamic analysis to more effectively find precise constraints. $BB().Diagonal$ stands for the bounding box diagonal of the intermediate object on which these operations

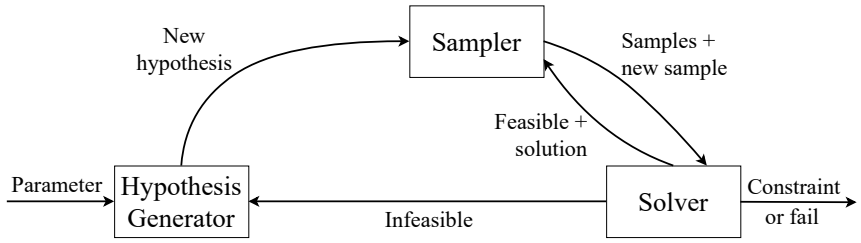


Figure 3.3.: Overview of our dynamic synthesis algorithm. We take the parameters for which constraints need to be synthesized. The Hypothesis Generator generates hypotheses of increasing complexity for these. The Sampler samples the parameter space, and evaluates the design/operations. The Solver uses samples from the Sampler to formulate each hypothesis into a linear program. If a solution to this is feasible, then more samples are collected to check and refine the constraint derived from the hypothesis (until a fixed threshold of samples is reached). Else, the hypothesis is rejected, and the Hypothesis Generator generates a new one.

are applied. Though the exact value of this expression would require evaluating the design on concrete parameter values, it is an over-approximated bound that would surely hold for any valid evaluation.

3.4.3. Constraint Synthesis Algorithm

The main algorithm of our framework is depicted in Algorithm 3. The algorithm takes as input a design for analysis, and a threshold for the amount of sampling to do. Internally, the algorithm maintains the constraints that have been found, sets of valid and invalid samples, and the parameters. First, the static analysis is run to get an initial set of constraints. This is shown in Algorithm 4. Basically, the static analysis traverses the design, and generates constraints according to the rules in Table 4. The static analysis is cheap, but only works for a few types of constraints. Still, the constraints found in this phase can be useful for restricting the sampling space of the dynamic analysis.

The dynamic analysis, at a high level, uses linear programming to fit the parameters of a hypothesis until enough supporting samples are found. An overview of our dynamic analysis approach is provided in Figure 3.3. When generating new samples, we use the current hypothesis, and target our sampling toward its boundary. The idea is that such samples help refine the hypothesis. The sampling threshold is given by the user.

If the space of valid configurations is very sparse, then the threshold needs to be high enough for the initial sampling to find a mix of valid and invalid parameter values. For each new operation, the previous invalid samples are discarded, as they are already rejected by the current set of constraints. The valid samples are reused on the new operation. Reusing samples is done to increase the efficiency of the analysis. As the discovered constraints get more restrictive, finding new samples gets more difficult.

ALGORITHM 3: Synthesizing CAD parameter constraints

```
Input: D: CAD program being analyzed,
t: sampling threshold.
Result: constr: constraints.
begin
    /* Initialization                                                    */
    constr = ∅;
    S+ = ∅ ;                                          /* Valid samples */
    S− = ∅ ;                                        /* Invalid samples */
    P = ∅ ;                                         /* Seen parameters */
    /* Get some constraints statically                                  */
    constr += StaticAnalysis(D);
    foreach op in D do
        S− = ∅ ;                        /* discard already rejected samples */
        foreach s in S+ do                          /* retest valid samples */
            if op(s) is not valid then
                S+ -= s;
                S− += s;
            end
        end
        P = P ∪ parameters(op);
        h = GenerateHypothesis(P);
        while h ≠ null do
            if LP(H,S+,S−) is feasible then                /* Section 3.4.3 */
                if |S+| + |S−| ≤ t then
                    (s+, s−) = DirectedSampling(op, constr, H);
                    S+ = S+ ∪ s+;
                    S− = S− ∪ s−;
                else
                    constr += h;
                    h = null;
                end
            else
                /* exception if no more hypothesis                      */
                h = GenerateHypothesis(P);
            end
        end
    end
    return constr;
end
```

ALGORITHM 4: StaticAnalysis

Input: D: CAD program being analyzed.
Result: *constr*: Synthesized static constraints.
begin
  constr = $\emptyset$;
  foreach *op in D* **do**
    constr += constraints on *op* from rule(s) ; `/* Table 3.1 */`
  end
  return *constr*;
end

ALGORITHM 5: DirectedSampling

Input: *op*: Operation,
constr: Constraints that must hold,
h: Current hypothesis.
Result: (s_+, s_-): valid/invalid samples.
begin
  while *True* **do**
    s = randomly sampled list of all the parameters;
    if *constr*(s) **then**
      if *op*(s) *is valid* **then**
        $s_+ = \{s\}$;
        $s_- = \emptyset$;
      else
        $s_+ = \emptyset$;
        $s_- = \{s\}$;
      end
      `/* sample the boundary of` h `*/`
      P = parameters(h);
      assume h is $\sum_{q \in P} c_q \cdot q \geq d$;
      foreach p *in* P **do**
        $b = \frac{d - \sum_{q \in P \setminus \{p\}} c_q \cdot q}{c_p}$;
        $s' = s[p \leftarrow random(b - \varepsilon, b + \varepsilon)]$;
        if *op*(s') *is valid* **then**
          s_+ += s';
        else
          s_- += s';
        end
      end
      return (s_+, s_-)
    end
  end
end

Hypothesis Generator

Algorithm 3 depends on the generation of hypotheses. Here, we follow the approach of enumerative program synthesis [7], and generate expressions of increasing complexity according to the grammar presented earlier (Section 3.4.3). The following is the grammar of a hypothesis h in our system:

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

where c denotes a constant in $\mathbb{R}$, and p denotes a parameter of the design. Our hypotheses are essentially linear expressions augmented with min and max. We start with simpler, or likelier hypotheses, and then move to more complicated ones if these do not fit.

In addition, we can also use the context, and generate different hypotheses depending on the operation. The context in which the parameters are used can guide us towards certain kinds of hypotheses. For example, fillet and chamfer act in at least 2 dimensions concurrently. This can therefore eliminate hypotheses where the parameter for fillet or chamfer is constrained by only one other parameter.

Sampler

Our hypothesis directed sampling method is described in Algorithm 5. The algorithm takes as input the current operation, the already synthesized constraints, and the current hypothesis. The operation is used to categorize the new samples into valid or invalid samples. The new samples must respect the existing constraints. We randomly sample until we find such a sample. Then we use that sample as a seed to find similar samples, but close to the boundary of the current hypothesis. Algorithm 5 shows this for linear constraints. In the case of more complex hypotheses, we reduce them to linear problems, as explained next.

Solver

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

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

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

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

However, this introduces disjunctions, which are not natively supported in linear programming. To get rid of these, we use a variation of the Big-M method [34] to generate the following inequalities:

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

$$\bigwedge_{i=1}^{n} m_i \in \{0,1\}, \quad 0 \leq \sum_{i=1}^{n} m_i < n$$

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

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

$e \leq \min(e_1, e_2) \quad \Leftrightarrow \quad (e \leq e_1 \wedge e \leq e_2)$

$e \leq \max(e_1, e_2) \quad \Leftrightarrow \quad (e \leq e_1 \vee e \leq e_2)$

and removing the disjunction as explained above.

3.5. Evaluation

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

3.5.1. Implementation

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

3.5.2. Experiments

We now evaluate our technique on some real-world designs from the recently released, and publicly accessible Fusion 360 Segmentation Dataset [84]. The dataset consists of

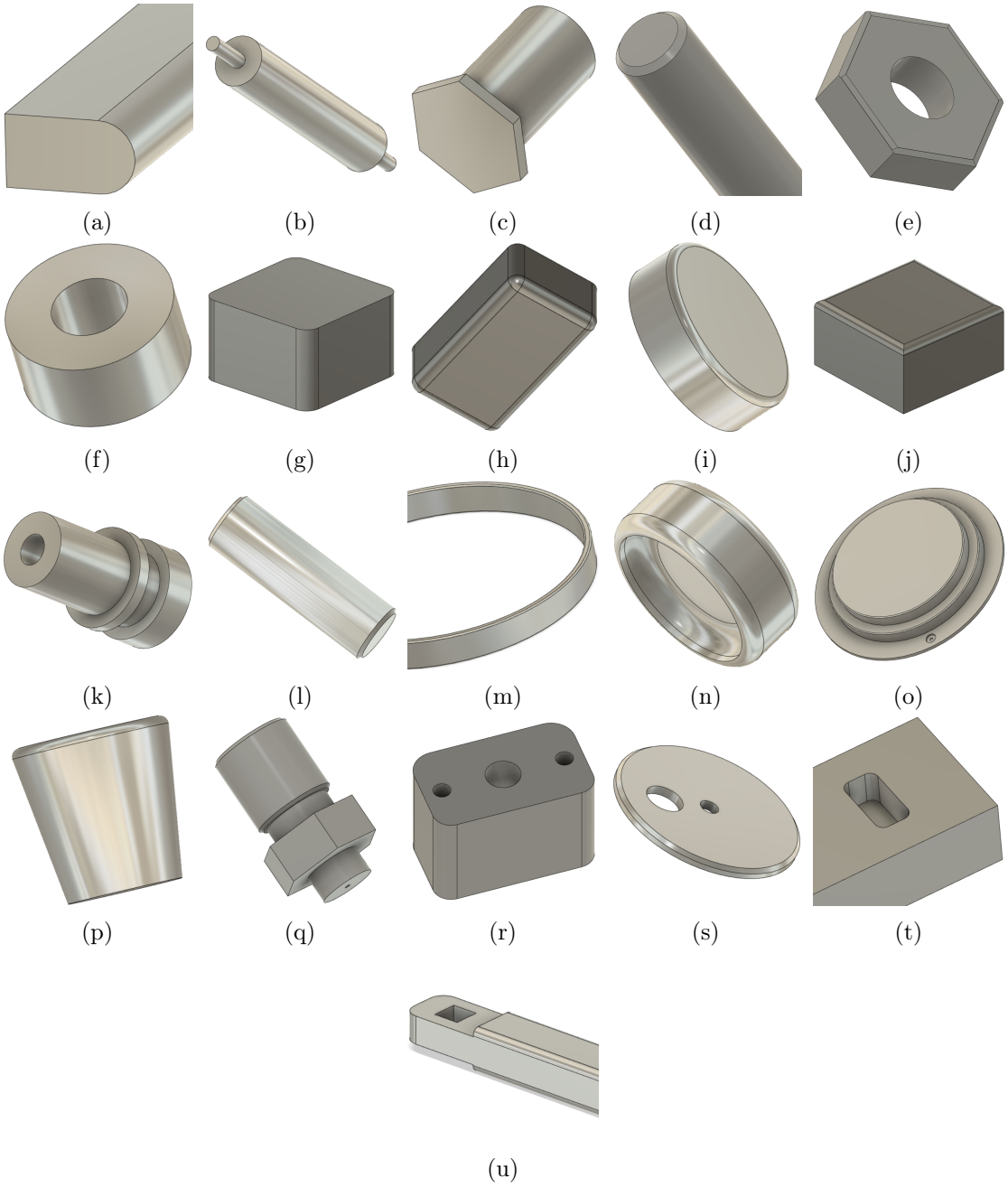


Figure 3.4.: Objects from the Fusion 360 Segmentation dataset used in the experiment (some views clipped to show prominent features).

35,858 final objects. Due to practical considerations, we can only consider a small portion of it. We first sort the dataset folder alphabetically (the object names are nondescript hashes). Then, we open and check the first 82 objects. We eliminate:

(i) Trivial objects. These are either primitives, or very close to primitives in the CAD library. This eliminates 3 objects.

(ii) Objects that require the use of unsupported, or non-straightforward CAD operations (including complex sketching) in CadQuery. This eliminates 35 objects.

(iii) Objects using spheres. We found that CadQuery was not stable when working with spheres, and produced unexpected errors. This eliminates 5 objects.

(iv) Duplicate objects (the same object can be obtained by changing parameter values of an already included design). This eliminates 18 objects.

After this filtering, we are left with 21 objects. The objects are re-designed in CadQuery to get an equivalent B-rep with sequence of operations in the CadQuery syntax. These designs are connected to our system, and used for the experiments.

Experimental Procedure

The designs we use for our evaluation are presented in Figure 3.4, labelled (a) - (u). The number of parameters in these designs range from 3 to 12, with an average of 5.5. For our dynamic analysis, we set the threshold of maximum number of samples for each constraint to 320 samples; the Hypothesis Generator generates hypotheses with at most 3 atomic predicates. The experiments are performed on a computer with an Intel Core i7-7820HK processor, 32 GB of RAM, and running on Windows 10.

Results

The results of our evaluation are summarized in Table 3.3. In Table 3.4, we summarize how complex these constraints are, in terms of the number of atomic predicates they contain. As the complexity of constraints is only an issue with dynamic analysis, we only address these. We find that we are able to synthesize constraints for almost all the designs under test. For a significant majority of the designs, we synthesize correct constraints for all the parameters involved, thereby segmenting the space of valid designs with a high degree of accuracy. There are also some designs for which we cannot synthesize constraints for all design parameters. In such cases, we synthesize constraints for some parameters, and are still able to segment the space of valid designs fairly well. There is just one design for which we cannot synthesize any constraints. For this design, we still do better than naive sampling of the parameter space because we can identify samples that have no chance of passing (restricting fillet radius to $<$ bounding box diagonal). Our technique is reasonably fast (median: 21.2 s, average: 36.1 s, max: 176.4 s). We are also able to reasonably restrict the space of invalid parameter configurations. The sampling success rate (percentage of randomly generated samples that lead to valid results) increases dramatically from

Table 3.3.: Synthesis of constraints, and runtimes for the various designs under test. For each design, we report the total number of parameters, and for how many we can synthesize constraints through static or dynamic analysis. The Success corresponds to how many random samples (out of 1000) lead to valid designs without constraints (Before) or with the synthesized constraints (After).

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

Table 3.4.: A summary of the complexity of synthesized constraints (captured by Size) in dynamic analysis. The complexity is the number of atomic predicates involved in the constraint.

	Constraints	
Id.	Size	Num.
(d)	2	1
(e)	1	1
	2	1
(f)	1	1
(g)	2	1
(h)	2	1
	3	1
(i)	2	1
(j)	3	1
(k)	1	2
(l)	2	1
(m)	1	1
	3	1

	Constraints	
Id.	Size	Num.
(n)	2	1
(o)	1	1
	2	2
(q)	2	1
(r)	2	2
(s)	1	1
	2	1
(t)	1	1
	2	1
(u)	1	1
	2	2
	3	1

26 % to 83.6 % on average. We organize further discussion by how the constraints are generated.

Fully solved statically There are 3 designs where the static analysis finds *all* the constraints. Our method has a negligible runtime for these cases. Let us consider the design (a) as in Figure 3.4. It consists of a union of a cylinder and a box, and the parameters for the dimensions of the box fully specifies the whole design. The parameter used for the length, for example gives the offset at which the cylinder is created, the width gives the diameter of the base, and the height is the height of the cylinder. All three parameters take values from the open interval $(0, \infty)$.

Fully solved statically and dynamically There are 11 designs that can be solved fully via a combination of static and dynamic analysis. Though the solved constraints fall shy of 100% accuracy, they are in-fact correct (verified manually). The small errors noticed here show up when working with curved shapes, and are due to numerical instabilities in the underlying CAD kernel. As depicted in Table 3.3, the designs here have a uniformly low success rate when sampling naively. With just a small runtime overhead, we are able to synthesize accurate constraints for *all* parameters of these designs.

We already discussed (e) from this segment in Section 3.1. Let us now look at our solution for the design (h), as depicted in Figure 3.4. A detailed view of its parameters and variations of final objects was presented earlier, in Figure 1.3. The design consists of the following sequence of operations: (i) a box with dimensions `l`, `w`, and `h` is created, (ii) the 4 edges parallel to the Y-axis are filleted with radius `f1`, (iii) the 4 edges on

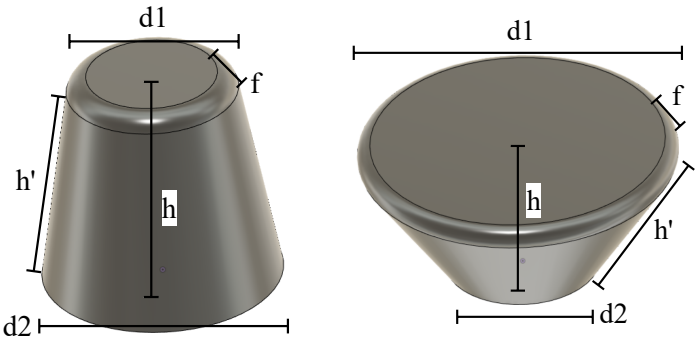


Figure 3.5.: Two variations of the design (p). The parameter `f` depends on `d1` and `d2`, as well as `h'`. However, `h'` is not an exposed parameter, but `h` (the offset between the two circles) is. We know that `f` must be less than the bounding box diagonal. However, the precise constraint cannot be synthesized, as doing so would require non-linear hypotheses with geometric functions.

top in the Y-axis are filleted with radius `f2`. Our static analyzer quickly comes up with constraints for `l`, `w`, and `h`. Then, our dynamic analyzer is asked to find constraints for `f1` and `f2`. They cannot take any arbitrary values, and *must* take values less than the bounding box diagonal of their respective intermediate objects. We enumeratively build hypotheses for both, `f1`, and `f2`. The simplest inequalities do not work out. We first find a solution for `f1` of the form: `f1 < c * min(l, h)`, with `c = 0.478` initially. This is later refined to `c = 0.499`. Similarly, we find a constraint for `f2` as: `f2 < 0.5 * min(l, w, h)`.

Partially solved For 7 designs in the experiment, we are only able to synthesize constraints for some of the parameters involved. For the rest, our hypothesis generator cannot construct correctly fitting hypotheses. As shown in the final segment of Table 3.3, our technique still significantly improves the sampling success rate vis-á-vis random sampling. This is because in addition to finding correct constraints for at least some of the parameters involved, we can often eliminate configurations that fail for sure (for example, see Figure 3.5), or find an approximate constraint (see Figure 3.6).

3.6. Conclusion

The power of parameterization is foundational to many modern applications of CAD. The flexibility it offers, however, comes at the cost of simplicity. Parameters in CAD depend on each other, and many combinations of parameter values lead to invalid final objects. Our proposed technique uses a mix of novel *static* and *dynamic* analysis to synthesize parameter constraints for a wide variety of designs. We evaluated this technique on designs from an open-source dataset, and found that our technique was able to significantly reduce the space of invalid parameter values via synthesized parameter constraints. Moreover, our technique was quick, taking an order of seconds for most designs. Our current approach

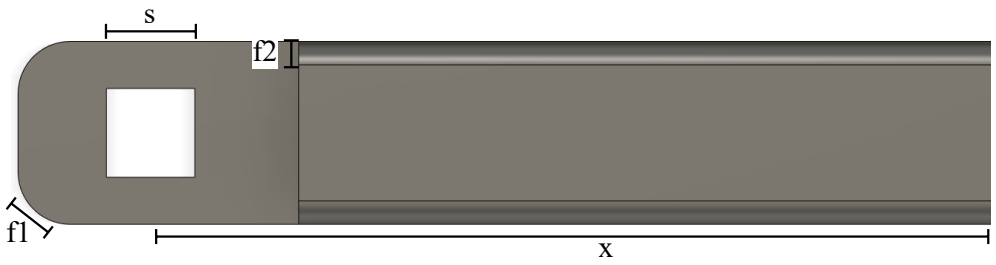


Figure 3.6.: Incomplete top-view of the design (u). We synthesize correct constraints for `f1`, `f2`, and `x`. For `s`, we obtain `s < 3.31 * max(l, w, h)`, which is an approximation.

does not support non-linear (or geometric) constraints. Supporting these would be the obvious next step, and could be achieved by generating richer hypotheses in the dynamic analysis step. This, however, is left for future work.

4
Related Work

4.1. Robustness of CAD

GUI-based CAD tools initially suffered from a severe robustness challenge of not being able to persistently and uniquely identify geometric entities of a design. This led to research on heuristics to resolve this, both, during design time [28, 29, 26], and during re-evaluation (under changed parameter values) [81, 3]. Robustness in design has been explored using geometric constraints [17], and explicit user input [62, 118]. These works successfully provide persistent and unique names to geometric entities. However, due to underspecification of GUI-based operations, these techniques cannot resolve ambiguity. Ambiguity resolution is done differently on different CAD tools, and because this information is usually opaque to users, interchange of designs between different CAD tools is also difficult [103]. No prior work has tried to bridge programming and GUI as a solution to the robustness problem.

4.2. Parametricity in CAD

Due to the modern design and fabrication landscape, the importance of parametricity in CAD has greatly increased. There has been work on systematic exploration of large parametric design spaces [88, 132, 135], using parameters for quality control [154], and interactive modification of designs [111]. There are extensions to the original idea of modelling by example [56] to ensure that the resulting design is fabricable [131], and on generation of fabrication constraints at design time [85]. Interestingly, research focussed on exploring parametricity of designs have either chosen programmatic back-ends, or highly curated designs. This is also true for the industry. Thingiverse [96], for example, only supports designs with a programmatic back-end for its customizable designs section. Our work enables designs made using a GUI to also benefit from the robust parametricity offered by programmatic back-ends.

4.3. Synthesis of CAD Programs

For modelling based on CSG, which involves creation of shapes using binary operators on solids, there has been recent work on synthesizing the underlying CSG-tree or programmatic representation [43, 104, 105]. Though these techniques can synthesize CSG-trees

(or programs) of various 3D models, CSG representation is less expressive than Boundary Representation (B-Rep) [140], the representation we and several modern CAD tools use. We view these works as complementary to the approach we present in Chapter 2 in that these try to uncover the underlying semantic representation of a tessellated 3D model, whereas we try to uncover the semantic representation of user activity *during* the design process.

Though several recent works have been looking into bridging GUI and programmatic representations for design and animation, these works have basically focussed on synthesizing small program modifications [30, 99, 100]. Our focus, on the other hand, is on synthesizing new code for the design of 3D objects. A unique problem we tackle in our synthesis technique is, unlike some other programming-by-example [65] based approaches, we only have one *example* (i.e., a user's direct manipulation action). Our underlying synthesis approach needs to be fast enough to be unobtrusive to the GUI-based interaction. In this context, we borrow ideas from Syntax-Guided Synthesis (SyGuS) [138, 6], as our abstract grammar restricts the syntax of the programs we synthesize. In SyGuS terminology, we use a compositional technique on top of an enumerative approach, i.e., we enumerate small programs and combine them to create more complex programs. However, unlike the SyGuS setting, we do not have a complete specification of what the synthesized program should do.

Our synthesis approach is based on a modified Decision Tree algorithm. Decision Trees have already been used in the context of program analysis and synthesis. For example, these have been applied to the learning of program invariants [60], and for tying-together small programs in a divide-and-conquer synthesis approach [106, 8].

4.4. Constraining CAD Parameters

We are not the first to propose the idea of synthesizing CAD parameter constraints. Fab Forms [136] employs user-specified validity conditions to synthesize constraints for design parameters. These constraints are then embedded in an interactive design explorer that enables a quick preview of the various (valid) final objects. Fab Forms, however, takes between several hours to several weeks for pre-computation. Part of this is because they, like other similar techniques [159, 146], work (albeit indirectly) with meshes. Moreover, many of Fab Forms validity checks are costly (e.g. finite element method). Our work uses the higher-level, and widely popular Boundary Representation (B-rep). We perform validity checks within the representation, which is much more efficient than similar checks on meshes. The use of B-rep and its operations also enables us to support more designs, and more design workflows than specialized constrained editing tools [21, 130]. Our validity conditions come from implicit assumptions of CAD operations and design methodology. We can therefore quickly eliminate many designs that are generally accepted to be invalid. Then, if more complicated checks are required, these can be performed on a much smaller subset of final objects.

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

some abstract metric, such as user choice [158, 51], or physical properties by sampling parametric designs [133]. Neural networks have also be used to generate programmatic representations of designs [75, 76]. Our work can be viewed as complimentary to these prior works. We constrain the parameter (or latent) space of designs, so as to eliminate invalid configurations.

Our research is inspired by other works on synthesizing easy to understand program invariants [46, 52, 59]. We basically analyze designs as conventional programs [108], and use implicit assumptions [45] from CAD-specific operations to come up parameter constraints.

Part II.

Simulation

5 Paracosm Interface

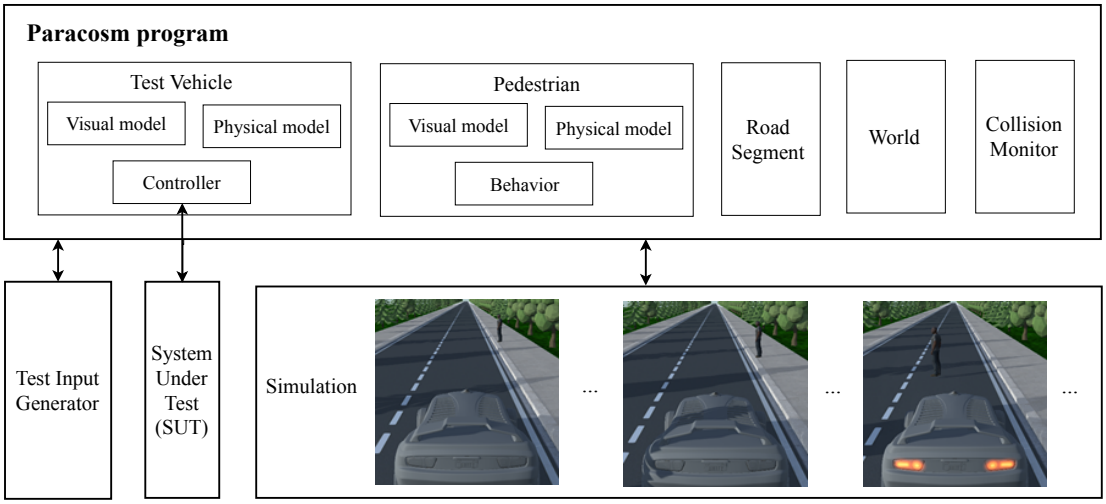


Figure 5.1.: A Paracosm program consists of parameterized reactive components such as the test vehicle, the environment, road networks, other actors and their behaviors, and monitors. The test input generation scheme guarantees good coverage over the parameter space. The test scenario depicted here shows a test vehicle stopping for a jaywalking pedestrian.

5.1. Introduction

Building autonomous driving systems requires complex and intricate engineering effort. At the same time, ensuring their reliability and safety is an extremely difficult task. There are serious public safety and trust concerns, aggravated by the many recent accidents involving autonomous cars [139, 25]. Software in such vehicles combine well-defined tasks such as trajectory planning, steering, acceleration and braking, with underspecified tasks such as building a semantic model of the environment from raw sensor data and making decisions using this model. Unfortunately, these underspecified tasks are critical to the safe operation of autonomous vehicles. Therefore, testing in large varieties of realistic scenarios is the only way to build confidence in the correctness of the overall system.

Running real tests is a necessary, but slow and costly process. It is difficult to reproduce corner cases due to infrastructure and safety issues; one can neither run over pedestrians to demonstrate a failing test case, nor wait for specific weather and road conditions. Therefore, engineers test autonomous systems in virtual simulation environments [53, 134, 40]. Simulation reduces the cost per test, and more importantly, gives precise control over all aspects of the environment, so as to test corner cases.

A major limitation of current tools is the lack of customizability: they either provide a GUI-based interface to design an environment piece-by-piece, or focus on bespoke pre-made environments. This makes the setup of varied scenarios difficult and time consuming. Though exploiting parametricity in simulation is useful and effective [16, 42, 149, 63], the cost of environment setup, and navigating large parameter spaces, is quite high [63]. Prior works have used bespoke environments with limited parametricity. More recently, programmatic interfaces have been proposed [54] to make such test procedures more systematic. However, the simulated environments are largely still fixed, with no dynamic behavior.

In this work, we present Paracosm, a programmatic interface that enables the design of *parameterized environments* and *test cases*. Test parameters control the environment and the behaviors of the actors involved. Paracosm supports various test input generation strategies, and we provide a notion of coverage for these. Rather than computing coverage over intrinsic properties of the system under test (which is not yet understood for neural networks [86]), our coverage criteria is over the space of test parameters. Figure 5.1 depicts the various parts of a Paracosm test. A Paracosm program represents a family of tests, where each instantiation of the program's parameters is a concrete test case.

Paracosm is based on a synchronous reactive programming model [153, 73, 27, 87]. Components, such as road segments or cars, receive streams of inputs and produce streams of outputs over time. In addition, components have graphical assets to describe their appearance for an underlying visual rendering engine and physical properties for an underlying physics simulator. For example, a vehicle in Paracosm not only has code that reads in sensor feeds and outputs steering angle or braking, but also has a textured mesh representing its shape, position and orientation in 3D space, and a physics model for its dynamical behavior. A Paracosm configuration consists of a composition of several components. Using a set of system-defined components (road segments, cars, pedestrians, etc.) combined using expressive operations from the underlying reactive programming model, users can set up complex temporally varying driving scenarios. For example, one can build an urban road network with intersections, pedestrians and vehicular traffic, and parameterize both, environment conditions (lighting, fog), and behaviors (when a pedestrian crosses a street).

Streams in the world description can be left "open" and, during testing, Paracosm automatically generates sequences of values for these streams. We use a coverage strategy based on *k-wise combinatorial coverage* [32, 82] for discrete variables and *dispersion* for continuous variables. Intuitively, k-wise coverage ensures that, for a programmer-specified parameter k, all combinations of values of any k discrete parameters are covered by tests (instead of covering all combinations of all discrete parameters). Low dispersion [125]

ensures that there are no "large empty holes" left in the continuous parameter space. PARACOSM uses an automatic test generation strategy that offers high coverage based on random sampling over discrete parameters and *deterministic* quasi-Monte Carlo methods for continuous parameters [125, 107].

Like many of the projects referenced before, our implementation performs simulations inside a game engine. However, PARACOSM configurations can also be output to the OPENDRIVE format [11] for use with other simulators, which is more in-line with the current industry standard. Implementation details, as well as case studies demonstrating PARACOSM are provided later, in Chapter 6.

5.1.1. Contributions

The main contributions of this Chapter are the following:

(i) We present an expressive framework for programmatically modeling complex and parameterized scenarios to test autonomous driving systems. Using PARACOSM users can specify the environment's layout, behaviors of actors, and expose parameters to a systematic testing infrastructure.

(ii) We define a notion of test coverage based on combinatorial k-wise coverage in discrete space and low dispersion in continuous space. We show a test generation strategy based on fuzzing that theoretically guarantees good coverage.

5.2. PARACOSM Language Interface

We now provide a walkthrough of PARACOSM through a testing example.

Suppose we have an autonomous vehicle to test. Its implementation is wrapped into a parameterized class:

```
AutonomousVehicle(start, model, controller) {
  void run(...) { ... } }
```

where the `model` ranges over possible car models (appearance, physics), and the `controller` implements an autonomous controller. The goal is to test this class in many different driving scenarios, including different road networks, weather and light conditions, and other car and pedestrian traffic. We show how PARACOSM enables writing such tests as well as generate test inputs automatically.

A *test configuration* consists of a composition of *reactive objects*. The following is an outline of a test configuration in PARACOSM, in which the autonomous vehicle drives on a road with a pedestrian wanting to cross. We have simplified the API syntax for the sake of clarity and omit the enclosing `Test` class. In the code segments, we use ':' for named arguments.

```
1 // Test parameters
2 light = VarInterval(0.2, 1.0) // value in [0.2, 1.0]
3 nlanes = VarEnum({2,4,6}) // value is 2, 4 or 6
4 // Description of environment
```

```
w = World(light:light, fog:0)
// Create a road segment
r = StraightRoadSegment(len:100, nlanes:nlanes)
// The autonomous vehicle controlled by the SUT
v = AutonomousVehicle(start:...,model:...,controller:...)
// Some other actor(s)
p = Pedestrian(start:.., model:..., ...)
// Monitor to check some property
c = CollisionMonitor(v)
// Place elements in the world
run_test(env: {w, r, v, p}, test_params: {light, nlanes},
    monitors: {c}, iterations: 100)
```

An instantiation of the reactive objects in the test configuration gives a *scene*—all the visual elements present in the simulated world. A *test case* provides concrete inputs to each "open" input stream in a scene. A test case determines how the scene evolves over time: how the cars and pedestrians move and how environment conditions change. We go through each part of the test configuration in detail below.

Reactive objects

PARACOSM is built around the synchronous reactive programming model [73, 27]. The core abstraction of PARACOSM is a *reactive object*. Reactive objects capture geometric and graphical features of a physical object, as well as their behavior over time. The behavioral interface for each reactive object has a set of *input* streams and a set of *output* streams. The evolution of the world is computed in steps of fixed duration which corresponds to events in a predefined `tick` stream. For streams that correspond to physical quantities updated by the physics simulator, such as position and speeds of cars, etc., appropriate events are generated by the underlying physics simulator.

Input streams provide input values from the environment over time; output streams represent output values computed by the object. The object's constructor sets up the internal state of the object. An object is updated by event triggered computations. PARACOSM provides a set of assets as base classes. Autonomous driving systems naturally fit reactive programming models. They consume sensor input streams and produce actuator streams for the vehicle model. We differentiate between static *environment* reactive objects (subclassing `Geometric`) and dynamic *actor* reactive objects (subclassing `Physical`). Environment reactive objects represent "static" components of the world, such as road segments, intersections, buildings or trees, and a special component called the *world*. Actor reactive objects represent components with "dynamic" behavior: vehicles or pedestrians. The world object is used to model features of the world such as lighting or weather conditions. Reactive objects can be *composed* to generate complex assemblies from simple objects. The composition process can be used to connect static components structurally–such as two road segments connecting at an intersection. Composition also connects the behavior of an object to another by binding output streams to input streams. At run time, the values on that input stream of the second object are obtained from the output values of the first. Composition must respect geometric properties—the

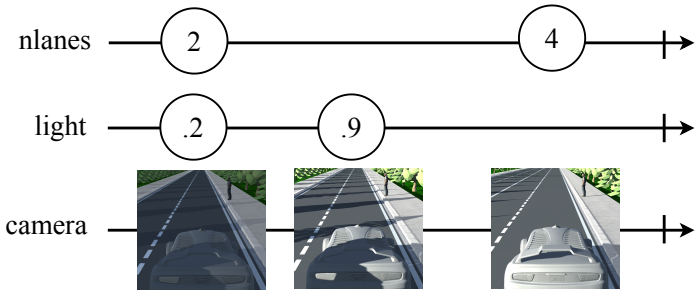


Figure 5.2.: Reactive streams represented by a marble diagram. A change in the value of test parameters `nlanes` or `light` changes the environment, and triggers a change in the corresponding sensor (output) stream `camera`.

runtime system ensures that a composition maintains invariants such as no intersection of geometric components. We now describe the main features in PARACOSM, centered around the test configuration above.

Test parameters

Using test variables, we can have general, but constrained streams of values passed into objects [128]. Our automatic test generator can then pick values for these variables, thereby leading to different test cases (see Figure 5.2). There are two types of parameters: continuous (`VarInterval`) and discrete (`VarEnum`). In the example presented, `light` (light intensity) is a continuous test parameter and `nlanes` (number of lanes) is discrete.

World

The `World` is a pre-defined reactive object in PARACOSM, with a visual representation responsible for atmospheric conditions like the light intensity, direction and color, fog density, etc. The code segment

```
w = World(light:light, fog:0)
```

parameterizes the world using a test variable for light and sets the fog density to a constant (0).

Road Segments

In our example, `StraightRoadSegment` was parameterized with the number of lanes. In general, PARACOSM provides the ability to build complex road networks by connecting primitives of individual road segments and intersections (an example is presented later in this Section). It may seem surprising that we model static scene components such as roads as reactive objects. This serves two purposes. First, we can treat the number of lanes in a road segment as a constant input stream that is set by the test case, allowing parameterized test cases. Second, certain features of static objects can also change over time. For example, the coefficient of friction on a road segment may depend on the weather condition, which can be a function of time.

Autonomous Vehicles & Systems Under Test (SUTs)

`AutonomousVehicle`, as well as other actors, extends the `Physical` class (which in turn subclasses `Geometric`). This means that these objects have a visual as well as a physical model. The visual model is essentially a textured 3D mesh. The physical model contains properties such as mass, moments of inertia of separate bodies in the vehicle, joints, etc. This is used by the physics simulator to compute the vehicle's motion in response to external forces and control input. In the following code segment, we instantiate and place our test vehicle on the road:

```
v = AutonomousVehicle(start:r.onLane(1, 0.1), model:CarAsset
    (...), controller:MyController(...))
```

The `start` parameter "places" the vehicle in the world (in relative coordinates). The `model` parameter provides the implementation of the geometric and physical model of the vehicle. The `controller` parameter implements the autonomous controller under test. The internals of the controller implementation are not important; what is important is its interface (sensor inputs and the actuator outputs). These determine the input and output streams that are passed to the controller during simulation. For example, a typical controller can take sensor streams such as image streams from a camera as input and produce throttle and steering angles as outputs. The PARACOSM framework "wires" these streams appropriately. For example, the rendering engine determines the camera images based on the geometry of the scene and the position of the camera and the controller outputs are fed to the physics engine to determine the updated scene. Though simpler systems like OPENPILOT [33] use only a dashboard-mounted camera, autonomous vehicles can, in general, mix cameras at various mount points, LiDARs, radars, and GPS. PARACOSM can emulate many common types of sensors which produce streams of data. It is also possible to integrate new sensors, which are not supported out-of-the-box, by implementing them using the game engine's API.

Other actors

A test often involves many actors such as pedestrians, and other (non-test) vehicles. Apart from the standard geometric (optionally physical) properties, these can also have some pre-programmed behavior. Behaviors can either be only dependent on the starting position (say, a car driving straight on the same lane), or be dynamic and reactive, depending on test parameters and behaviors of other actors. In general, the reactive nature of objects enables complex scenarios to be built. For example, here, we specify a simple behavior of a pedestrian crossing a road.The pedestrian starts crossing the road when a car is a certain distance away. In the code segments below, we use '`_`' as shorthand for a lamdba expression, i.e., "`f(_)`" is the same as "`x => f(x)`".

```
Pedestrian(value start, value target, carPos, value dist, value
    speed) extends Geometric {
  ... // Initialization
  // Generate an event when the car gets close
  trigger = carPos.Filter( abs(_ - start) < dist )
```

```
  // target location reached
  done = pos.Filter( _ == target )
  // Walk to the target after trigger fires
  tick.SkipUntil(trigger).TakeUntil(done).foreach( ... /* walk
     with given speed */ )
}
```

Monitors and test oracles

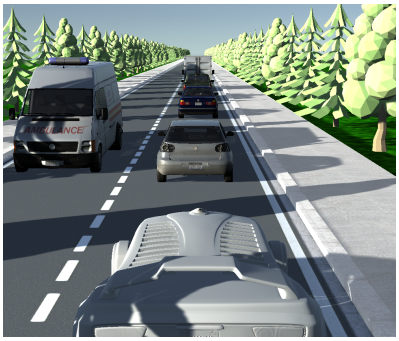
PARACOSM provides an API to provide qualitative and quantitative temporal specifications. For instance, in the following example, we check that there is no collision and ensure that the collision was not trivially avoided because our vehicle did not move at all.

```
// no collision
CollisionMonitor(AutonomousVehicle v) extends Monitor {
  assert(v.collider.IsEmpty()) }
// cannot trivially pass the test by staying put
DistanceMonitor(AutonomousVehicle v, value minD) extends
    Monitor {
  pOld = v.pos.Take(1).Concat(v.pos)
  D = v.pos.Zip(pOld).Map( abs(_ - _) ).Sum()
  assert(D >= minD)
}
```

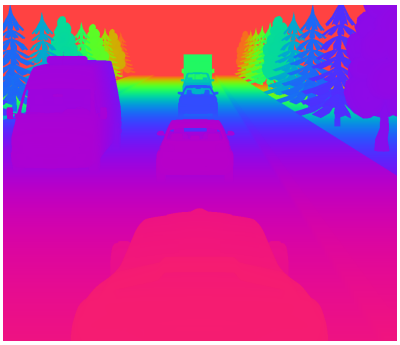
The ability to write monitors which read streams of system-generated events provides an expressive framework to write temporal properties, something that has been identified as a major limitation of prior tools [63]. Monitors for metric and signal temporal logic specifications can be encoded in the usual way [69, 36].

Sensors

Simple autonomous systems like openpilot [33] use only a dash-cam, but more complex ones mix cameras, LiDARs, radars, and GPS. PARACOSM can emulate common types of sensors which produce streams of data. In Figure 5.3, we show data coming from a few sensors. Normal cameras can be mounted at specific points on the vehicle and it is possible to vary parameters like the focal length. The camera can be ideal or include imperfections like blur or noise. During the rendering process for RGB images we can also extract depth maps (Figure 5.3b) to cheaply emulate LiDAR. [1] In general, it is possible to provide new sensors which are not supported out-of-the-box by PARACOSM. However, new types of sensors need to be implemented directly on top of the rendering engine's API.



(a) Standard RGB camera view.

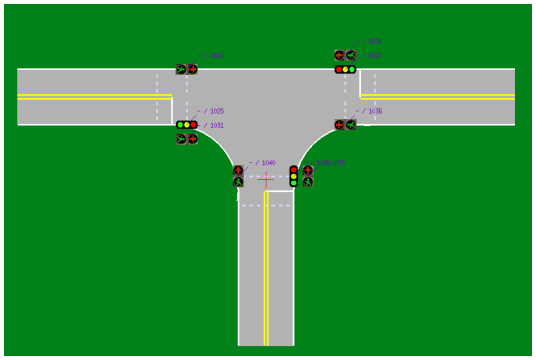


(b) Depth map (distance mapped to hue in the HSV color space).

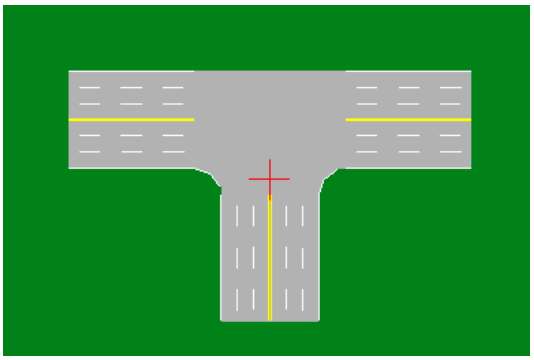
(c) Car mounted camera with long focal length.

(d) Car mounted camera with short focal length.

Figure 5.3.: Simulating different sensors in PARACOSM.



(a) T-Intersection with 2 lanes, long road segments, and traffic and pedestrian lights.



(b) T-Intersection with 4 lanes, short road segments, and no traffic or pedestrian lights.

Figure 5.4.: Parameterized road segments outputted to OPENDRIVE. Options to create vehicular and pedestrian traffic lights can also be arguments to the `TIntersection` interface.

Road networks

Road elements can be composed using the `connect` operation. PARACOSM supports complex road elements such as cross-intersections, T-intersections, and roundabouts. Connections can be established using the `connect` method, that takes physical connection identifiers and road elements as arguments. The connections are directed in order to compute the positions of the elements. One road element becomes the parent and its children are positioned relative to its position and the specified connection points. After an object is connected, a new *composite* road element which encapsulates all road elements along with requisite transformations (rotations and translations) is returned. The following example shows how road segments can be connected into a road network.

```
1 len = VarInterval(5, 100)
2 nlanes = VarInterval({2, 4})
3 // Create a parameterized T-intersection and three straight
      road elements (east, south, west)
4 t = TIntersection(nlanes:nlanes)
5 e = StraightRoadSegment(len:len, nlanes:nlanes)
6 s = StraightRoadSegment(len:len, nlanes:nlanes)
7 w = StraightRoadSegment(len:len, nlanes:nlanes)
8 // connect and get new composite object
9 net = t.connect((t.ONE, e, e.TWO),
10                 (t.TWO, s, s.ONE),
11                 (t.THREE, w, w.ONE))
```

In this example, the T-intersection is not given a specific position or orientation. It is therefore instantiated at the origin. Road elements connecting to it are then positioned with respect to it. After connection, the composite road element `net` can be used for tests in simulation or to a standardized format (OPENDRIVE). Figure 5.4 shows some samples in the OPENDRIVE viewer.

Connecting elements has two purposes. First, it allows PARACOSM to perform sanity checks like proper positioning of road elements. Second, it creates an overlay graph of the road networks which can easily be followed by environment controlled vehicles. When a road network is created, the runtime system of PARACOSM checks that compositions of road elements and intersections are topologically and geometrically valid. All road elements must be connected to a matching road correctly (for example, a 2-lane road segment cannot be connected to a 6-lane road segment directly), there can be no spatial overlaps between road segments, and the positions of the connection points must match.

In general, PARACOSM inherits all programming features of the underlying imperative programming model as well as reactive programming with streams. Thus, one can build complex urban settings through composition and iteration. For instance, the grid world shown in Figure 5.5 was created by iterating a simple road network.

[1] It is also possible to use ray casting techniques to more accurately simulate a LiDAR but the computational cost is significantly higher.

Figure 5.5.: A large grid world with several connected road elements viewed in our default 3D simulator.

5.3. Test Inputs and Coverage

Worlds in PARACOSM directly describe a parameterized family of tests. The testing framework allows users to specify various strategies to generate input streams for both, static, and dynamic reactive objects in the world.

5.3.1. Test Cases

A *test* of *duration* T executes a configuration of reactive objects by providing inputs to every open input stream in the configuration for T ticks. The inputs for each stream must satisfy `const` parameters and respect the range constraints from `VarInterval` and `VarEnum`. The runtime system manages the scheduling of inputs and pushing input streams to the reactive objects. Let In denote the set of all input streams, and $\mathsf{In} = \mathsf{In}_D \cup \mathsf{In}_C$ denote the partition of In into *discrete* streams and *continuous* streams respectively. Discrete streams take their value over a finite, discrete range; for example, the color of a car, the number of lanes on a road segment, or the position of the next pedestrian (left/right) are discrete streams. Continuous streams take their values in a continuous (bounded) interval. For example, the fog density or the speed of a vehicle are examples of continuous streams.

5.3.2. Coverage

In the setting of autonomous vehicle testing, one often wants to explore the state space of a parameterized world to check "how well" an autonomous vehicle works under various situations, both qualitatively and quantitatively. Thus, we now introduce a notion of coverage. Instead of structural coverage criteria such as line or branch coverage, our goal is to cover the parameter space. In the following, for simplicity of notation, we assume that all discrete streams take values from $\{0, 1\}$, and all continuous streams take values in the real interval $[0, 1]$. Any input stream over bounded intervals—discrete or continuous—can be encoded into such streams. For discrete streams, there are finitely many tests, since each co-ordinate is Boolean and there is a fixed number of co-ordinates. One can define the coverage as the fraction of the number of vectors tested to the total number of vectors.

Unfortunately, the total number of vectors is very high: if each stream is constant, then there are already 2^n tests for n streams. Instead, we consider the notion of *k-wise testing* from combinatorial testing [82]. In k-wise testing, we fix a parameter k, and ask that every interaction between every k elements is tested. Let us be more precise. Suppose that a test vector has N co-ordinates, where each co-ordinate can get the value 0 or 1. A set of tests A is a *k-wise covering family* if for every subset $\{i_1, i_2, \ldots, i_k\} \subseteq \{1, \ldots, N\}$ of co-ordinates and every vector $v \in \{0,1\}^k$, there is a test $t \in A$ whose restriction to the $i_1, \ldots, i_k$ is precisely v.

For continuous streams, the situation is more complex: since any continuous interval has infinitely many points, each corresponding to a different test case, we cannot directly define coverage as a ratio (the denominator will be infinite). Instead, we define coverage using the notion of *dispersion* [125, 107]. Intuitively, dispersion measures the largest empty space left by a set of tests. We assume a (continuous) test is a vector in $[0,1]^N$: each entry is picked from the interval $[0,1]$ and there are N co-ordinates. Dispersion over $[0,1]^N$ can be defined relative to sets of neighborhoods, such as N-dimensional balls or axis-parallel rectangles. Let us define $\mathcal{B}$ to be the family of N-dimensional axis-parallel rectangles in $[0,1]^N$, our results also hold for other notions of neighborhoods such as balls or ellipsoids. For a neighborhood $B \in \mathcal{B}$, let $vol(B)$ denote the volume of B. Given a set $A \subseteq [0,1]^N$ of tests, we define the *dispersion* as the largest volume neighborhood in $\mathcal{B}$ without any test:

$$\mathsf{dispersion}(A) = \sup\{\mathrm{vol}(B) \mid B \in \mathcal{B} \text{ and } A \cap B = \emptyset\}$$

A lower dispersion means better coverage.

Let us summarize. Suppose that a test vector consists of N_D discrete co-ordinates and N_C continuous co-ordinates; that is, a test is a vector (t_D, t_C) in $\{0,1\}^{N_D} \times [0,1]^{N_C}$. We say a set of tests A is (k, ε)-*covering* if

(i) for each set of k co-ordinates $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, N_D\}$ and each vector $v \in \{0,1\}^k$, there is a test $(t_D, t_C) \in \{0,1\}^{N_D} \times [0,1]^{N_C}$ such that the restriction of t_D to the co-ordinates $i_1, \ldots, i_k$ is v; and

(ii) for each $(t_D, t_C) \in A$, the set $\{t_C \mid (t_D, t_C) \in A\}$ has dispersion at most ϵ.

5.3.3. Test Generation

The goal of our default test generator is to maximize (k, ϵ) for programmer-specified number of test iterations or `ticks`.

k-wise covering family

One can use explicit construction results from combinatorial testing to generate k-wise covering families [32]. However, a simple way to generate such families with high probability is random testing. The proof is by the probabilistic method [4] (see also [93]). Let A be a set of $2^k(k \log N - \log \delta)$ uniformly randomly generated $\{0,1\}^N$ vectors. Then A is a k-wise covering family with probability at least $1 - \delta$.

Low dispersion sequences

It is tempting to think that uniformly generating vectors from $[0,1]^N$ would similarly give low dispersion sequences. Indeed, as the number of tests goes to infinity, the set of randomly generated tests has dispersion 0 almost surely. However, when we fix the number of tests, it is well known that uniform random sampling can lead to high dispersion [107, 125]; in fact, one can show that the dispersion of n uniformly randomly generated tests grows asymptotically as $O((\log\log n/n)^{\frac{1}{2}})$ almost surely. Our test generation strategy is based on *deterministic quasi-Monte Carlo sequences*, which have much better dispersion properties, asymptotically of the order of $O(1/n)$, than the dispersion behavior of uniformly random tests. There are many different algorithms for generating quasi-Monte Carlo sequences deterministically (see, e.g., [107, 125]). We use *Halton sequences*. For a given ϵ, we need to generate $O(\frac{1}{\epsilon})$ inputs via Halton sampling. In Chapter 6 (Sections 6.3.1 and 6.3.3), we provide an experimental comparison of uniform random and Halton sampling.

Cost functions and local search

In many situations, testers want to optimize parameter values for a specific function. A simple example of this is finding higher-speed collisions, which intuitively, can be found in the vicinity of test parameters that already result in high-speed collisions. Another, slightly different case is (greybox) fuzzing [9, 121], for example, finding new collisions using small mutations on parameter values that result in the vehicle narrowly avoiding a collision. Our test generator supports such *quantitative* objectives and *local search*. A quantitative monitor evaluates a cost function on a run of a test case. Our test generation tool generates an initial, randomly chosen, set of test inputs. Then, it considers the scores returned by the Monitor on these samples, and performs a local search on samples with the highest/lowest scores to find local optima of the cost function.

5.4. Conclusion

Deploying autonomous systems like self-driving cars in urban environments raises several safety challenges. The complex software stack processes sensor data, builds a semantic model of the surrounding world, makes decisions, plans trajectories, and controls the car. The end-to-end testing of such systems requires the creation and simulation of whole worlds, with different tests representing different world and parameter configurations. Paracosm tackles these problems by (i) enabling procedural construction of diverse scenarios, with precise control over elements like road layout, physical and visual properties of objects, and behaviors of actors in the system, and (ii) using quasi-random testing to obtain good coverage over large parameter spaces. In the following Chapter, we evaluate Paracosm, and provide experimental evidence of the utility of our language interface and test generation strategies.

6 Paracosm: Evaluation

6.1. Introduction

In the previous Chapter, we described the PARACOSM language interface and proposed test generation strategies for systematic exploration of large parameter spaces. Most real-world autonomous driving agents are implemented using deep neural networks, which are known to be notoriously difficult to debug. The PARACOSM language interface simplifies the process of designing complex real-world environments and behaviors. Additionally, the PARACOSM test generator ensures good coverage of the test parameter space, enabling testers to understand how different test parameters affect the performance of the autonomous agent.

PARACOSM is a complex project, involving a language interface, test generator, and a simulation interface. In this Chapter, we discuss how the PARACOSM language, test, and simulation interface are implemented. We also evaluate PARACOSM, and demonstrate its utility in the design of test environments, exploration of large parameter spaces, and detection of faulty autonomous agent behavior. Through various case studies, we show that PARACOSM can be an effective testing framework for both qualitative properties (crash) and quantitative properties (distance maintained while following a car, or image misclassification).

6.1.1. Contributions

The main contributions of this Chapter are the following:

(i) We provide implementation details of PARACOSM, discussing the various components, and the technology behind them.

(ii) We present several experiments and case studies demonstrating the utility of PARACOSM's language interface for designing parameterized simulations, and for finding faults in autonomous driving systems.

6.2. Runtime System and Implementation

PARACOSM uses the Unity game engine [151] to render visuals, do runtime checks and simulate physics (via PhysX [110]). Reactive objects are built on top of UniRx [78], an

implementation of the popular Reactive Extensions framework [122]. The game engine manages geometric transformations of 3D objects and offers easy to use abstractions for generating realistic simulations. Encoding behaviors and monitors, management of 3D geometry and dynamic checks are implemented using the game engine interface. The project code is available at: `https://gitlab.mpi-sws.org/mathur/paracosm`.

A simulation in PARACOSM proceeds as follows. A test configuration is specified as a subclass of the `EnvironmentProgramBaseClass`.Tests are run by invoking the `run_test` method, which receives as input the reactive objects that should be instantiated in the world as well as additional parameters relating to the test. The `run_test` method runs the tests by first initializing and placing the reactive objects in the scene using their 3D mesh (if they have one) and then invoking a reactive engine to start the simulation. The system under test is run in a separate process and connects to the simulation. The simulation then proceeds until the simulation completion criteria is met (a time-out or some monitor event).

Physics For practical purposes, game engines typically treat physics and rendering pipelines separately. The physics engine is informed about the physical components (subclasses of `Physical`). Physics simulations in PARACOSM work on rigid bodies, i.e., solid bodies with an assumption of no deformation. This enables accurate detection of collision, but not deformations caused by the collision. This is an acceptable limitation as a collision typically marks the end of a test case.

Output to standardized formats There have been recent efforts to create standardized descriptions of tests in the automotive industry. The most relevant formats are OPENDRIVE [11] and OPENSCENARIO [12]. OPENDRIVE describes road structures, and OPENSCENARIO describes actors and their behavior. PARACOSM currently supports outputs to OPENDRIVE. Producing these files requires the same machinery as a simulation in the game engine to initialize the world. However, instead of running the simulation in the game engine, the internal representation is serialized into the OpenDRIVE format. Due to the static nature of the specification format, a different file is generated for each test iteration/configuration.

6.3. Experiments & Case Studies

We now present several experiments and case studies performed using PARACOSM. In Section 6.3.1, via some common testing tasks, we evaluate PARACOSM on criteria such as ease of use, efficacy of test generation strategies, and the ability of finding problematic cases. In Section 6.3.2, we test some autonomous driving components trained on real-world data. Finally, in Section 6.3.3, we provide even more experiments that test autonomous driving behavior.

Table 6.1.: An overview of experiments on some common testing tasks, and the corresponding coverage (as in Section 5.3.3) we achieved. Note that even though the Adaptive Cruise Control study has 2 discrete parameters, we calculate k-wise coverage for 3 as the 2 parameters require 3 bits for representation.

	Road segmentation	Jaywalking pedestrian	Adaptive Cruise Control
SUT	VGGNet CNN [137]	NVIDIA CNN [20]	NVIDIA CNN [20]
Training	191 images	403 image & car control samples	1034 image & car control samples
Test params	3 discrete	2 continuous	3 continuous & 2 discrete
Test iters	100	100, 15s timeout	100, 15s timeout
Monitor	Ground truth	Scored Collision	Collision & Distance
Coverage	$k = 3$ with probability ~ 1	$\epsilon = 0.041$	$\epsilon = 0.043$, $k = 3$ with probability ~ 1

6.3.1. Evaluation on Common Testing Tasks

We now perform some common testing tasks in PARACOSM so as to evaluate our tool. We test various deep neural networks trained for road segmentation, safety against jaywalking pedestrians, and for Adaptive Cruise Control. Our evaluation aims to answer the following questions:

(i) **Ease of use**: does PARACOSM's programmatic interface enable the easy design of test environments and worlds?

(ii) **Efficacy of test generation**: do the test input generation strategies discussed in Section 6.1 effectively explore the parameter space?

(iii) **Ability to find problems**: can PARACOSM help uncover poor performance or bad behavior of the SUT in common autonomous driving tasks?

To answer (i), we develop three independent environments rich with visual features and other actors, and use the variety generated with just a few lines of code as a proxy for ease of design. To answer (ii), we use coverage maximizing strategies for test inputs to all the three environments/case studies. We also use and evaluate cost functions and local search based methods. To answer (iii), we test various neural network based systems and demonstrate how PARACOSM can help uncover problematic scenarios. A summary of these experiments is available in Table 6.1.

Testing task 1: Road segmentation

Using PARACOSM's programmatic interface, we design a long road segment with several vehicles. The vehicular behavior is to drive on their respective lanes with a fixed maximum velocity. The test parameters are the number of lanes ($\{2, 4\}$), number of cars in the environment ($\{0, 5\}$) and light conditions ($\{Noon, Evening\}$). Noon lighting is much brighter than the evening. The direction of lighting is also the opposite. We test a deep

(a) A good test with all parameter values same as the training set (true positive: 89%, false positive: 0%).

(b) A bad test with all parameter values different from the training set (true positive: 9%, false positive: 1%).

Figure 6.1.: Example results from the road segmentation case study. Pixels with a green mask are segmented by the SUT as a road.

Table 6.2.: Summary of results of the road segmentation case study. Each combination of parameter values is presented separately, with the parameter values used for training in bold. We report the SUT's average true positive rate (% of pixels corresponding to the road that are correctly classified) and false positive rate (% of pixels that are not road, but incorrectly classified as road).

# lanes	# cars	Lighting	# test iters	True positive (%)	False positive (%)
2	**5**	**Noon**	12	70%	5.1%
2	5	Evening	14	53.4%	22.4%
2	0	Evening	12	51.4%	18.9%
2	0	Noon	12	71.3%	6%
4	5	Evening	10	60.4%	7.1%
4	5	Noon	16	68.5%	20.2%
4	0	Evening	13	51.5%	7.1%
4	0	Noon	11	83.3%	21%

CNN called VGGNet [137], that is known to perform well on several image segmentation benchmarks. The task is road segmentation, i.e., given a camera image, identifying which pixels correspond to the road. The network is trained on 191 dashcam images captured in the test environment with fixed parameters (2 lanes, 5 cars, and *Noon* lighting), recorded at the rate of one image every $1/10^{th}$ second, while manually driving the vehicle around (using a keyboard). We test on 100 images generated using Paracosm's default test generation strategy (uniform random sampling for discrete parameters). Table 6.2 summarizes the test results. Tests with parameter values far away from the training set are observed to not perform so well. As depicted in Figure 6.1, this happens because varying test parameters can drastically change the scene.

Testing task 2: Jaywalking pedestrian

We now test over the environment presented in Section 5.2. The environment consists of a straight road segment and a pedestrian. The pedestrian's behavior is to cross the

Table 6.3.: Results of the jaywalking pedestrian test, comparing dispersion (as in Section 5.3.3), failure rates (due to a collision), and the maximum speed of collision for various sampling strategies.

Testing strategy	Dispersion (ϵ)	% fail	Max. collision
Random	0.092	7%	10.5 m/s
Halton	0.041	10%	11.3 m/s
Random+opt/collision	0.109	13%	11.1 m/s
Halton+opt/collision	0.043	20%	11.9 m/s
Random+opt/almost failing	0.126	13%	10.5 m/s
Halton+opt/almost failing	0.043	13%	11.4 m/s

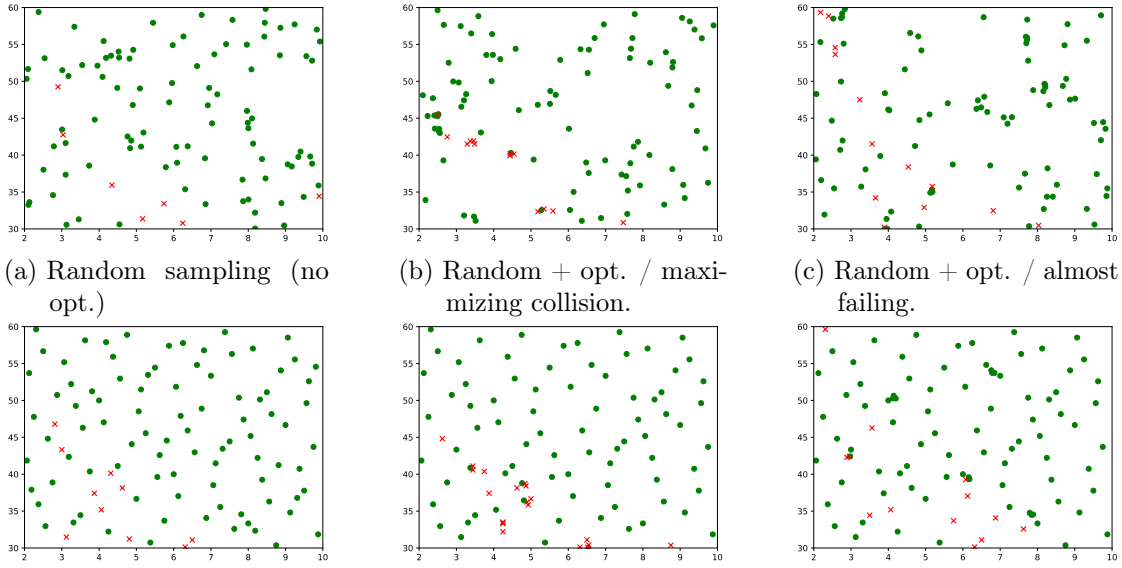
road at a specific walking speed when the autonomous vehicle is a specific distance away. The walking speed of the pedestrian and the distance of the autonomous vehicle when the pedestrian starts crossing the road are test parameters. The SUT is a CNN based on NVIDIA's behavioral cloning framework [20]. It takes camera images as input, and produces the relevant steering angle or throttle control as output. The SUT is trained on 403 samples obtained by driving the vehicle manually and recording the camera and corresponding control data. The training environment has pedestrians crossing the road at various time delays, but always at a fixed walking speed (1 m/s). In order to evaluate the efficacy of our proposed test generation strategies completely, we test the default coverage maximizing sampling approach, as well as explore two quantitative objectives: first, maximizing the collision speed, and second, finding new failing cases around samples that *almost* fail. For the default approach, the `CollisionMonitor` as presented in Section 5.2 is used. For the first quantitative objective, this `CollisionMonitor`'s code is prepended with the following calculation:

```
// Score is speed of car at time of collision
coll_speed = v.speed.CombineLatest(v.collider, (s,c) => s).
    First()
```

The score `coll_speed` is used by the test generator for optimization. For the second quantitative objective, the `CollisionMonitor` is modified to give high scores to tests where the distance between the autonomous vehicle and pedestrian is very small:

```
CollisionMonitor(AutonomousVehicle v, Pedestrian p) extends
    Monitor {
  minDist = v.pos.Zip(p.pos).Map(1/abs(_-_)).Min()
  coll_score = v.collider.Map(0)
  // Score is either 0 (collision) or 1/minDist
  score = coll_score.DefaultIfEmpty(minDist)
  assert(v.collider.IsEmpty())
}
```

We evaluate the following test input generation strategies: (a) Random sampling, (b) Halton sampling, and (c) Random or Halton sampling with local search for the two



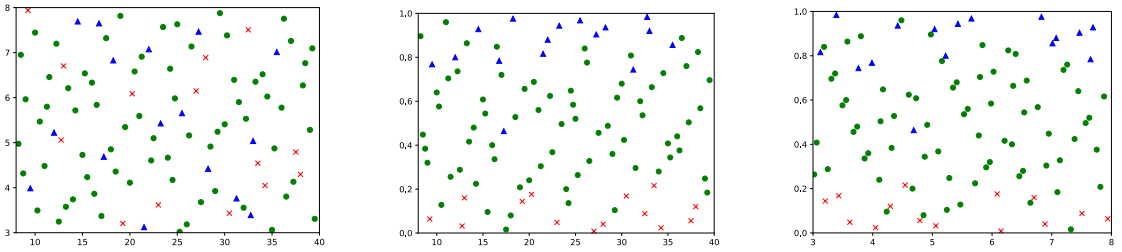
(a) Random sampling (no opt.)
(b) Random + opt. / maximizing collision.
(c) Random + opt. / almost failing.
(d) Halton sampling (no opt.)
(e) Halton + opt. / maximizing collision.
(f) Halton + opt. / almost failing.

Figure 6.2.: A comparison of the various test generation strategies for the jaywalking pedestrian case study. The X-axis is the walking speed of the pedestrian (2 to 10 m/s). The Y-axis is the distance from the car when the pedestrian starts crossing (30 to 60 m). Passing tests are labelled with a green dot. Failing tests (tests with a collision) are marked with a red cross.

quantitative objectives. We run 100 iterations of each strategy with a 15 second timeout. For random or Halton sampling, we sample 100 times. For the quantitative objectives, we first generate 85 random or Halton samples, then choose the top 5 scores, and finally run 3 simulated annealing iterations on each of these 5 configurations. Table 6.3 presents results from the various test input generation strategies. Clearly, Halton sampling offers the lowest dispersion (highest coverage) over the parameter space. This can also be visually confirmed from the plot of test parameters (Figure 6.2). There are no big gaps in the parameter space. Moreover, we find that test strategies optimizing for the first objective are successful in finding more collisions with higher speeds. As these techniques perform simulated annealing repetitions on top of already failing tests, they also find more failing tests overall. Finally, test strategies using the second objective are also successful in finding more (newer) failure cases than simple Random or Halton sampling.

Testing task 3: Adaptive Cruise Control

We now create and test in an environment with our test vehicle following a car (lead car) on the same lane. The lead car's behavior is programmed to drive on the same lane as the test vehicle, with a certain maximum speed. This is a very typical driving scenario that engineers test their implementations on. We use 5 test parameters: the initial lead of the



(a) Initial offset (X-axis) vs. max. speed (Y-axis).

(b) Initial offset (X-axis) vs. fog density (Y-axis).

(c) Max. speed (X-axis) vs. fog density (Y-axis).

Figure 6.3.: Continuous test parameters of the Adaptive Cruise Control study plotted against each other: the initial offset of the lead car (8 to 40 m), the lead car's maximum speed (3 to 8 m/s) and the fog density (0 to 1). Green dots, red crosses, and blue triangles denote passing tests, collisions, and inactivity respectively.

lead car to the test vehicle ($[8\,m,\ 40\,m]$), the lead car's maximum speed ($[3\,m/s,\ 8\,m/s]$), density of fog[1] in the environment ($[0, 1]$), number of lanes on the road ($\{2,\ 4\}$), and color of the lead car ($\{Black,\ Red,\ Yello,\ Blue\}$). We use both, `CollisionMonitor`[2] and `DistanceMonitor`, as presented in Section 5.2. A test *passes* if there is no collision and the autonomous vehicle moves atleast 5 m during the simulation duration (15 s).

We use PARACOSM's default test generation strategy, i.e., Halton sampling for continuous parameters and Random sampling for discrete parameters (no optimization or fuzzing). The SUT is the same CNN as in the previous case study. It is trained on 1034 training samples, which are obtained by manually driving behind a red lead car on the same lane of a 2-lane road with the same maximum velocity (5.5 m/s) and no fog.

The results of this case study are presented in Table 6.4. Looking at the discrete parameters, the number of lanes does not seem to contribute towards a risk of collision. Surprisingly, though the training only involves a Red lead car, the results appear to be the best for a Blue lead car. Moving on to the continuous parameters, the fog density appears to have the most significant impact on test failures (collision or vehicle inactivity). In the presence of dense fog, the SUT behaves pessimistically and does not accelerate much (thereby causing a failure due to inactivity). These are all interesting and useful metrics about the performance of our SUT. Plots of the results projected on to continuous parameters are presented in Figure 6.3.

[1] 0 denotes no fog and 1 denotes very dense fog (exponential squared scale).

[2] the monitor additionally calculates the mean distance of the test vehicle to the lead car during the test, which is used for later analysis.

Table 6.4.: Parameterized test on Adaptive Cruise Control, separated for each value of discrete parameters, and low and high values of continuous parameters. A test *passes* if there are no collisions and no inactivity (the overall distance moved by the test vehicle is more than 5 m. The average offset (in m) maintained by the test vehicle to the lead car (for passing tests) is also presented.

	Discrete parameters						Continuous parameters					
	Num. lanes		Lead car color				Initial offset (m)		Speed (m/s)		Fog density	
	2	4	Black	Red	Yellow	Blue	< 24	≥ 24	< 5.5	≥ 5.5	< 0.5	≥ 0.5
Test iters	54	46	24	22	27	27	51	49	52	48	51	49
Collisions	7	7	3	3	6	2	6	8	8	6	12	0
Inactivity	12	4	4	4	6	2	9	7	9	7	1	15
Offset (m)	42.4	43.4	46.5	48.1	39.6	39.1	33.7	52.7	38.4	47.4	36.5	49.8

Results and analysis

We now summarize the results of our experiments with respect to our original evaluation criteria:

Ease of use: All the three case studies involve varied, rich and dynamic environments. They are representative of tests engineers would typically want to do [63], and we parameterize many different aspects of the world and the dynamic behavior of its components. These designs are at most 70 lines of code. This provides confidence in PARACOSM's ability of providing an easy interface for the design of realistic test environments.

Efficacy of test generation: Our default test generation strategies are found to be quite effective at exploring the parameter space systematically, eliminating large unexplored gaps, and at the same time, successfully identifying problematic cases in all the three case studies. The jaywalking pedestrian study demonstrates that optimization and local search are possible on top of these strategies, and are quite effective in finding the relevant scenarios. The adaptive cruise control study tests over 5 parameters, and we observe good coverage, even with such a large parameter space. Therefore, it is amply clear that PARACOSM's test input generation methods are useful and effective.

Ability to find problems: The road segmentation case study uses a well-performing neural network for object segmentation, and we are able to detect degraded performance for automatically generated test inputs. Whereas this study focuses on static image classification, the next two, i.e., the jaywalking pedestrian and the adaptive cruise control study uncover poor performance on simulated driving, using a popular neural network architecture for self driving cars. Therefore, we can safely conclude that PARACOSM can find bugs in various different kinds of systems related to autonomous driving.

6.3.2. Testing Systems Trained on Standard Datasets

The experiments in the previous Section involved systems trained inside a simulation environment. However, many image segmentation and classification systems are trained on real-world data. We now test some such systems.

We design a highly parameterized environment using PARACOSM's programmatic interface. The environment consists of 4 `StraightRoadSegment`s connected by a `CrossIntersection`. The test has three discrete parameters and three continuous parameters:
Discrete: (a) the number of lanes is either 2 or 4, (b) the light condition corresponds to a morning, noon, or evening, (c) number of other cars on the road range from 2 to 9,
Continuous: (d) the camera focal length takes any value between 18 mm and 22 mm, (e) the height of the mounting point of the camera takes values between 1.9m and 2.2m, and finally, (f) the camera is angled slightly down with a pitch angle between $-12°$ and $-10°$.

Many of these parameters correspond to the vehicle's camera. These were chosen because in preliminary tests, small perturbations to the camera's properties led to drastically different results (see Figure 6.4). We perform 100 test iterations using PARACOSM's default test generation scheme.

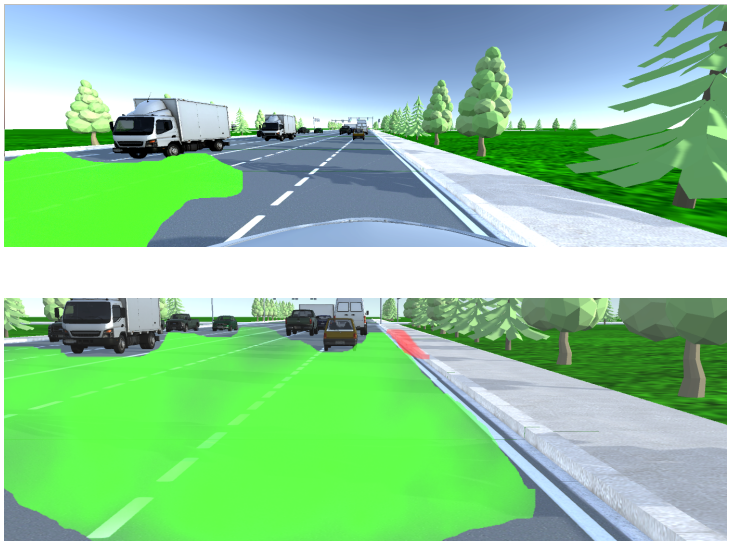


Figure 6.4.: Sample output from a road detection system. Green pixels represent correctly identified road, red pixels represent pixels incorrectly identified as road. A longer focal length (34mm) results in better road detection in comparison to a shorter focal length (10mm).

Road segmentation

The SUTs here take RGB images as input and return those pixels that are estimated to be a part of the road. We tested: (a) the convolutional neural network from Simonyan and Zisserman (popular as VGGNet) [137], (b) Multinet from Teichmann, Weber et al. [147], a top performer on the KITTI Road Estimation Benchmark, and (c) the fully convolutional network by Long, Shelhamer and Darrell [90]. All three are trained on the KITTI road segmentation dataset [55] (289 images). The first and third networks do not have a name, so we use initials of the authors' names, SZ and LSD, respectively. Figure 6.5 shows the results for the 100 test iterations (x-axis). We plot results in the order in which the tests were performed. The y-axis shows the percentage of the "ground truth" road identified as road by the method. A cursory look did not reveal any correlation between road segmentation performance and parameter choice. We observe that SZ is the best performer overall. What is quite striking is the results of LSD: in these tests, it either performs well, or not at all. Except for the poorly performing examples of LSD, false positives are not an issue, generally. Our hypothesis is that the networks do not generalize sufficiently from the limited training data and images that are too different from training lead to poor results.

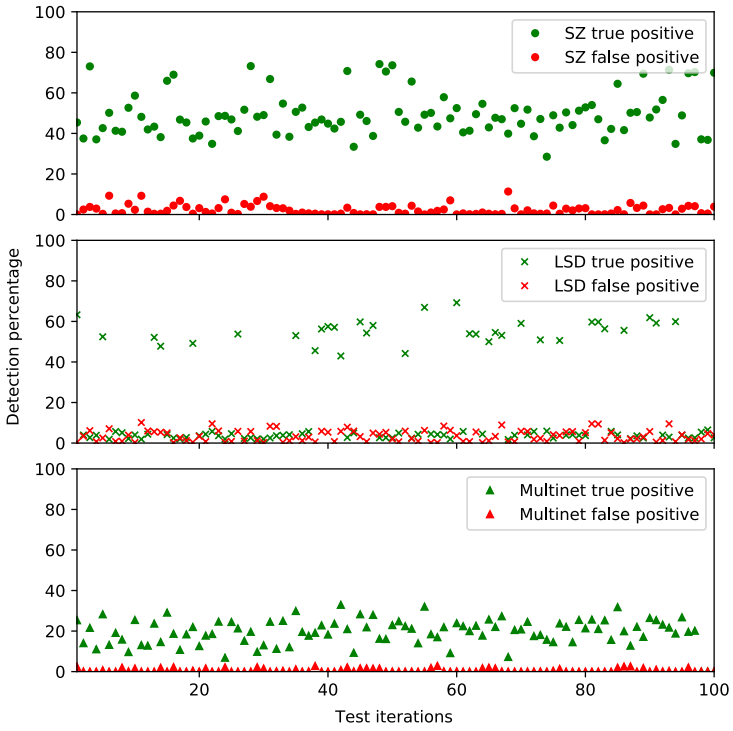


Figure 6.5.: Road segmentation rates (% of the ground truth) for three SUTs (SZ, LSD, and Multinet).

Figure 6.6.: Sample outputs of a vehicle detection system. Small changes to the environmental condition lead to missed cars.

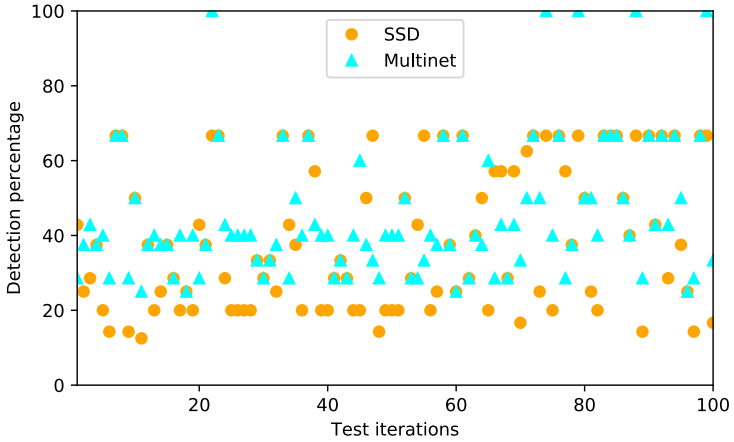


Figure 6.7.: Vehicle detection rates for the two SUTs (SSD and Multinet).

Vehicle detection

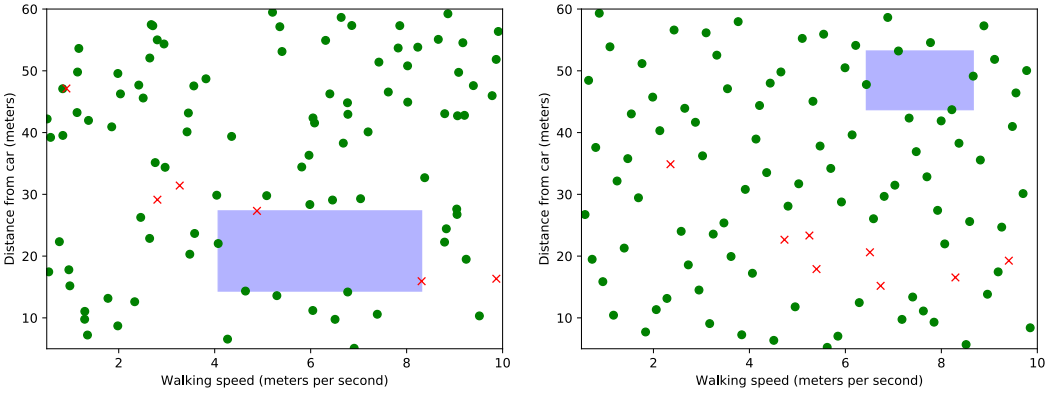
The SUTs here take RGB images as input and return bounding boxes around pixels that correspond to vehicles. Figure 6.6 shows an example of a vehicle detection system's output. To detect other vehicles in the vicinity of the autonomous car, we used: (a) the single shot multibox detector (SSD), a deep neural network [89], trained with the Pascal Object Recognition Database Collection [47], (b) Multinet [147] like in the previous experiment. Figure 6.7 summarizes the results. The results are again in the order of the tests. In this experiment, we did not observe any false positives. Overall, Multinet performs better than SSD but these systems are much closer than in the previous experiment. While the detection rates may look disappointing, factors like occlusion as seen in Figure 6.6 make it difficult to detect all the cars. The two experiments presented here highlight the fact that even with quite narrow parameter ranges (especially for the camera), the quality of results can vary widely.

6.3.3. Experiments Demonstrating Specific Paracosm Features

In the experiments that follow, we highlight specific testing features of Paracosm to demonstrate their utility in practice. The SUT in these experiments is the NVIDIA behavioral cloning framework [20] as we saw before, trained inside Paracosm's simulation environment.

Low-dispersion sequences vs. random sampling

This experiment is similar to the jaywalking pedestrian experiment presented earlier, in Section 6.3.1. We now present experimental evidence underlining the importance of low-dispersion sequences, and how coverage improves with more test iterations. Note that the parameter ranges here are different (wider) than the study presented before. However, the SUT is the same. Figure 6.8 demonstrates the advantage of low-dispersion



(a) Random (dispersion is 0.105) (b) Halton (dispersion is 0.041)

Figure 6.8.: Random vs. Halton sampling for 100 test iterations. The purple area is the largest axis-parallel rectangle without a test. X axis is the walking speed of the pedestrian ([0.5m/s, 10m/s]). Y axis is the distance from the car when the pedestrian starts crossing ([5m, 60m]). Failing tests (collisions) are marked with a red cross, passing tests are labelled with a green dot.

Table 6.5.: Random vs. Halton sampling for the pedestrian crossing experiment over various test iterations. The test parameters are the walking speed of the pedestrian ([0.5m/s, 10m/s]) and distance from the car when the pedestrian starts crossing ([5m, 60m]).

	Dispersion values		Failing tests	
# tests	Random	Halton	Random	Halton
50	0.200	0.083	6%	8%
100	0.105	0.041	6%	8%
200	0.051	0.029	7%	8%
400	0.025	0.011	8.75%	8.25%

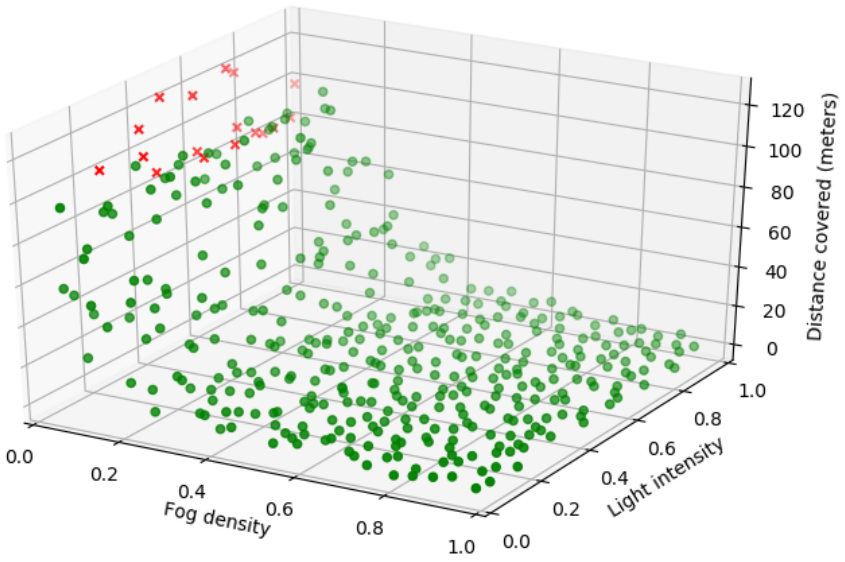
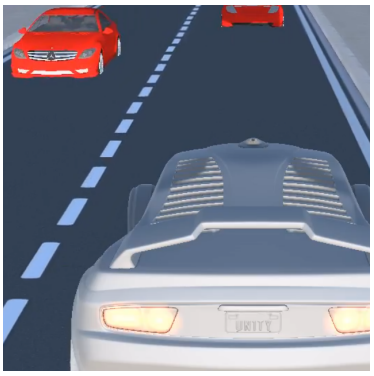


Figure 6.9.: Distance covered (Z-axis, $[0m, 120m]$) in changing fog (X-axis, $[0, 1]$) and light (Y-axis, $[0, 1]$) conditions tested with 400 iterations of the Halton sequence. Green dots and red crosses denote the absence or presence of a collision. The car is trained with a fog density of 0 and light intensity of 0.5.

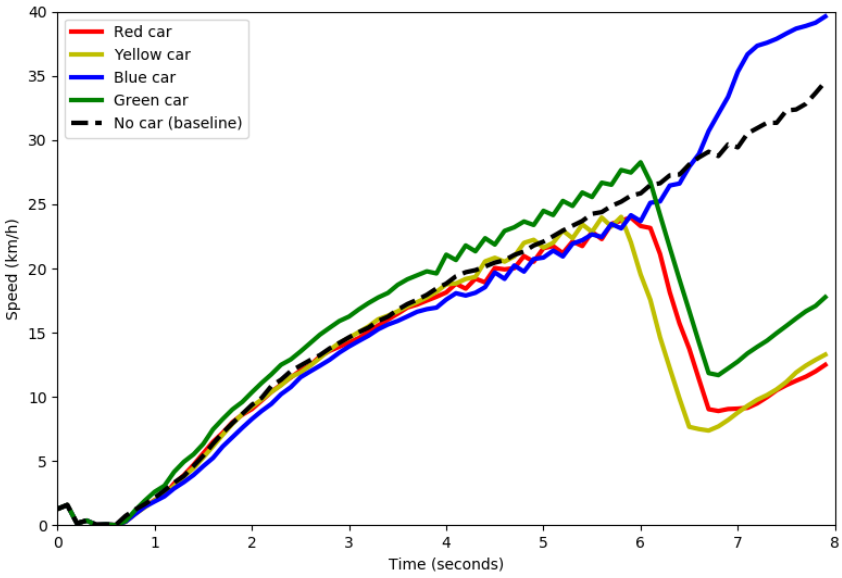
sampling over random sampling. Samples are more spread out for the Halton sequence (low-dispersion). In Table 6.5, we report the difference between random and Halton sampling for various numbers of test iterations. Halton sampling gives much better dispersion and even leads to more failure cases being revealed (especially for fewer test iterations).

Changing environmental settings

Reactive variables in Paracosm can be used to parameterize environment settings so as to describe a large class of environmental settings. To demonstrate the utility of parameterized environmental settings, we train a model at fixed light intensity and no fog. The experiment set-up is similar to the Adaptive Cruise Control case study presented earlier in Section 6.3.1. We now analyse the autonomous vehicle's performance when the light intensity and fog density are varied. We report the overall distance covered, and whether a collision happened. Each test lasts 15 seconds. Parameter values for light and fog are generated using the Halton sequence. Our results are depicted in Figure 6.9. Perhaps unsurprisingly, the car performs best around the parameter values it was trained on. The car acts extremely conservatively in the presence of fog, not covering much distance. On the other hand, perturbations to light intensity often lead to scenarios with collisions.



(a) Test vehicle braking on seeing a red car coming from the opposite direction, even though there is a large distance to the lead car (car on the same lane).



(b) Speed (X-axis, $[0km/h, 40km/h]$) over time (Y-axis, $[0s, 8s]$) of car trained to follow a red car in the presence of another car coming from the opposite direction. Depending on the color of the incoming car, the speed of the car changes vis-à-vis the baseline driving with no other car.

Figure 6.10.: Effect of features of geometric components (other cars).

Table 6.6.: Comparison of failures (due to collision) and distance covered relative to the training data (no collision) for various training strategies (100 test iterations).

Training setup	Failing tests	Distance covered
Default	68%	62.7%
Randomized	5%	20.6%
Depth map	36%	89.0%

Geometric components and their features

In the previous case study, the SUT was trained to follow a red lead car driving in front of it on a two-lane road. Under ideal conditions (conditions under which the SUT is trained), it is observed that the autonomous vehicle indeed follows the red lead car while maintaining a safe distance and not colliding with it. We now test how the SUT reacts to cars coming from the other direction. The test vehicle's throttle should not be affected by cars coming from the other direction. However, our hypothesis is that perhaps the SUT simply learns to slow the car down when there are several red pixels in the camera image. Our experiment confirms that this indeed is the case. When we test with a red car coming from the other direction, our autonomous vehicle slows down in response to this car being close (see Figure 6.10a). Speed is picked up again once this car passes. Perhaps more surprisingly, we find that the vehicle also slows down when the car coming from the other direction is yellow or green, but has no affect when the car is blue. Figure 6.10b has plots of speed vs. time for these various cases (with the baseline being no car coming from the other direction).

Different sensor models

We now demonstrate the utility of Paracosm for testing in complex environments, and under various sensor models. We program a scenario with five cross-intersections, traffic in both directions, and buildings and trees on both sides of the roads. Three training strategies are used: (a) *Default*, where the traffic only consisted of red cars, (b) *Randomized*, where cars are given random colors, and (c) *Depth map*, where depth information is used instead of an RGB camera (a depth sensor can be easily simulated in Paracosm).

During testing, cars can take any color. More specifically, there are 8 cars in total, each of which can take one of 3 colours. This gives 3^8 different vehicle color configurations. For 100 test iterations, the probability of getting k-wise coverage for $k = 2$ is almost 1, and for $k = 3$, it is atleast 0.29. Table 6.6 summarizes the results of this experiment. We find that the use of a *Depth map* is quite useful: it leads to fewer failing tests in comparison to the *Default* case, and it covers more distance than both, the *Default* and *Randomized* cases on average.

6.4. Conclusion

In this Chapter, we provided practical implementation details, and evaluated Paracosm's language and test interface. The various case studies presented here demonstrate the wide variety of realistic, complex, and useful simulation environments that can be constructed using Paracosm's language interface. These case studies also used Paracosm's novel test generation strategies to systematically explore large test parameter spaces. As demonstrated by the results of these experiments, Paracosm was able uncover poor performance and incorrect behavior in all the different systems we tested.

7 Related Work

7.1. Reactive Programming Models

PARACOSM's programming model follows a synchronous reactive style for dataflows, similar to functional reactive programming (FRP) [153] or to synchronous dataflow languages like LUSTRE [27]. FRP has been applied to control robotic and embedded systems [73], including cars [50]. In PARACOSM, rather than using reactive programming to control the SUT, we use reactive elements to build and control the world around the system under test. Our programming model shares elements found in automata-based models like reactive modules [5]. In contrast to these languages, PARACOSM natively supports geometric and physical properties of components and integrates with a game engine. Reactive programming has gained popularity in recent years as a way to build asynchronous distributed applications [87]. Our model is synchronous because we wish to test physical systems with a global time.

Traditionally, test-driven software development paradigms [15] have advocated testing and mocking frameworks to test software early and often. Mocking frameworks and mock objects [92, 102] allow programmers to test a piece of code against an API specification, even when the implementation of the API is not available. Typically, mock objects are stubs providing outputs to explicitly provided lists of inputs of simple types, with little functionality of the actual code. Thus, they fall short of providing a rich environment for autonomous driving. PARACOSM can be seen as a mocking framework for reactive, physical systems embedded in the 3D world. Our notion of constraining streams is inspired by work on declarative mocking [128].

7.2. Testing Cyber-Physical Systems

There is a large body of work on automated test generation tools for cyber-physical systems through heuristic search of a high-dimensional continuous state space. While much of this work has focused on low-level controller interfaces [10, 48, 39, 35, 129, 37] rather than the system level, specification and test generation techniques arising from this work—for example, the use of metric and signal temporal logics or search heuristics—can be adapted to our setting. Though many test generation frameworks have been proposed, very few among these employ a notion of coverage of the test parameter space [141]. In fact, we are the first to propose dispersion as a coverage criteria.

More recently, test generation tools have started targeting autonomous systems with machine learning components, under a simulation-based semantic testing framework similar to ours [41, 16, 2, 54, 149, 150, 1]. In most of these works, an underlying visual scenario is fixed by hand, and test generation explores plausible perturbations to the nominal scenario to detect bugs. Such analyses are shown to be preferable to the application of random noise on the input vector. Moreover, a simulation-based approach filters benign misclassifications from misclassifications that actually lead to bad or dangerous behavior [66]. Our work extends this line of work and provides an expressive language to design parameterized environments and tests. AsFault [58] uses random search and mutation for procedural generation of road networks for testing. Scenic [54] is a probabilistic language to define geometric configurations of objects in fixed environments (such as the GTA V environment [127]). In comparison, our reactive-style language enables specification of parameterized environments and actor behaviors. AC3R [57] follows a directed approach and reconstructs test cases from accident reports.

To address problems of high time and infrastructure cost of testing autonomous systems, several simulators have been developed. The most popular is Gazebo [53] for the ROS [119] robotics framework. It offers a modular and extensible architecture, however falls behind on visual realism and complexity of environments that can be generated with it. To counter this, game engines are used. Popular examples are TORCS [157], CARLA [40], and AirSim [134] Modern game engines and support creation of realistic urban environments. Though they enable visually realistic simulations and enable detection of infractions such as collisions, the environments themselves are difficult to design. Designing a custom environment involves manual placement of road segments, buildings, and actors (as well as their properties). Performing many systematic tests is therefore time-consuming and difficult. While these systems and Paracosm share the same aims and much of the same infrastructure, Paracosm focuses on procedural design and systematic testing, backed by relevant coverage criteria.

7.3. Test Strategies

Adversarial examples for neural networks [64, 145] introduce perturbations to inputs that cause a classifier to classify "perceptually identical" inputs differently. There has been a lot of work focused on finding adversarial examples in the context of autonomous driving, as well as on training a network to be robust to perturbations [114, 18, 155, 101, 61]. Tools such as DeepXplore [117], DeepTest [148], DeepGauge [91], and SADL [79] define a notion of coverage for neural networks based on the number of neurons activated during tests compared against the total number of neurons in the network and activation during training. However, these techniques focus mostly on individual classification tasks and apply 2D transformations on images. In comparison, we can also test the closed-loop behavior of the system, and our parameters directly change the world rather than apply transformations post facto. We can observe, over time, that certain vehicles are not detected, which is more useful to testers than a single misclassification [63]. Furthermore, it is already known that structural coverage criteria may not be an effective strategy for

finding errors in classification [86]. We use coverage metrics on the test space, rather than the structure of the neural network. Alternately, there are recent techniques to verify controllers implemented as neural networks through constraint solving or abstract interpretation [61, 72, 126, 155, 44]. While these tools do not focus on the problem of autonomous driving, their underlying techniques can be combined in the test generation phase for PARACOSM.

Part III.

Discussion

8 Discussion

GUI-based interfaces, owing primarily to their ease of use, have dominated the domains of CAD and Simulation. In our Thesis, we highlight some of the advantages programming offers over GUI. We propose techniques that bridge the gap between the ease of use offered by GUI, and robustness, ease of analysis/testing, and flexibility offered by programming.

8.1. Future Work

The idea of bringing together the best of both, interactive GUI and programming, is rather grand. A lot more work can be done in this direction, even when focussing exclusively on Design and Simulation. This Thesis lays the groundwork for many such avenues for future work. We now summarize some of these ideas.

Support for modifications

The synthesis technique we introduce in Chapter 2 only generates new code snippets, and does not modify existing code. For an even tighter integration of GUI and programming for CAD, the ability to modify underlying code can be useful. As we demonstrate in Section 2.2.2, it is already possible to track geometric features through code and pinpoint which line(s) of code each feature comes from. This infrastructure can be merged with prior work such as value trace equation solving [30] and lenses [99] to enable modifications to code.

Arithmetic expressions in synthesis

The synthesis procedure we present in Chapter 3 can use program variables in scope for the synthesis procedure. However, our technique currently only uses these variables directly. Sometimes, arithmetic expressions over program variables are needed to synthesize better queries. For example, to select the coordinate of the side of a cube, we can use the sum of one corner's position and the width (if these are available). Existing work on program synthesis [8] can be used to generate such arithmetic expressions over program variables, and then be used in our synthesis procedure.

Synthesizing complex CAD constraints

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

Accommodating user preferences in synthesis

The synthesis techniques we present in this Thesis do not take individual user preferences into account. For example, in Chapter 2, we discuss how we synthesize short code segments from GUI-based CAD interactions. By default, this algorithm uses information gain to decide which selector to use to expand its decision tree. However, we could use other heuristics, as long as they avoid non-trivial selectors that do not split the feature set. In particular, we can define an extra term which weighs selectors differently and use a modified information gain:

$$IG'(C, s) = w(s) \cdot IG(C, s)$$

where w is a function that gives a weight to the selector s. As selectors are not unique, the w function can be tailored to a user's preferences. Our algorithm can generate multiple trees by expanding multiple alternatives with roughly similar information gain, and let users select which is best. Selections can be recorded, and over time, we could learn a w function.

Similarly, for the synthesis of parameter constraints as presented in Chapter 3, we can weigh terms in the Hypothesis Generator (Section 3.4) differently so as to prefer hypotheses that users find easier to understand or better for a particular domain.

Richer training data through simulation

Paracosm's test infrastructure can be extended to aid in the training of deep neural networks that require large amounts of high quality training data. Generating data is typically time consuming and expensive. Paracosm can easily generate labelled data for static images. For driving scenarios, we can record a user manually driving in a parameterized Paracosm environment and augment this data by varying parameters that should not impact the vehicle's behavior. For instance, we can vary the color of other cars, positions of pedestrians who are not crossing, or even the light conditions and sensor properties (within reasonable limits). The new data could be generated completely automatically to augment the learning dataset.

Reducing Sim-to-Real gap

Though modern game engines offer very realistic rendering capability, there are still many perceivable differences between simulation and the real-world. These differences between rendered images vs. real-world data are collectively termed the sim-to-real gap. If the sim-to-real gap is large, autonomous machines may behave differently in simulation and in the real world. Because of this, some techniques have been recently proposed to make simulated images more realistic [123, 68].

Though Paracosm can greatly benefit from these techniques, they may not be suited for every use case. These enhancement techniques are learning-based, and their outputs depend heavily on the training data. For example, when trained on real-world images from a green region in Germany, and then tested on a simulation depicting a dry region in California, they transform barren hills into green spaces [123]. This may be counter-productive when users want to test in specific environments and don't want the underlying scene to be changed.

Version control for CAD and Simulation

A significant advantage programming has over its GUI-based counterparts is the availability of good version control. Tools such as Git enable users to track various versions of a project, and to collaboratively make and compare changes. These tools, however, are text-based. All changes are tracked, compared and integrated in text format. This modality is not suitable for CAD and Simulation, both of which are highly visual domains. An interesting direction for future work would be to develop version control tools for these domains that enable tracking changes in a visual format.

Effect of interface choice on productivity and quality

The question of productivity of users in various design interfaces is quite old [19], and still very relevant. CAD and Simulation tools are being used in new ways, and by many new and non-traditional users. We would like to study the impact of interface choice on user productivity, and the quality of designs/simulations created, considering many different kinds of users and use cases. Robustness and flexibility would be important metrics, and so would ease of use, learning and time required to accomplish certain tasks. In Chapter 2 (Section 2.4.5), we present a user study comparing traditional programming, and our proposed integration of GUI and programming for parametric CAD. A larger and longer user study would give us insight into what different interfaces are especially good at, and how user productivity and the quality of designs and simulations can be improved.

8.2. Concluding Remarks

This Thesis presents unique programmatic interfaces that improve modern CAD and Simulation workflows. For parametric CAD, we propose an interactive, yet robust design interface wherein users use the GUI to perform CAD operations, and the corresponding

code is automatically synthesized for them. We show that this interface works well for a wide variety of designs, and that designers prefer our interface over plain programming. Also for parametric CAD, a useful piece of information often missing from shared designs is the range of valid parameter values. We propose an algorithm, inspired by work on program analysis, that synthesizes constraints on design parameters such that the resulting final object is valid. We evaluate our technique on designs from a public dataset, and synthesize accurate parameter constraints for most of them in an order of seconds. Finally, for Simulation, we propose a high-level programmatic interface called Paracosm, which simplifies the process of creating parameterized autonomous vehicle tests. We also propose test generation strategies that ensure good coverage of the test parameter space. Through various case studies, we demonstrate that Paracosm enables the specification of various complex driving scenarios, and the detection of problematic cases.

Though a lot of work still remains to achieve the dream of a fully-integrated GUI and programmatic interface, we believe the work presented in our Thesis already demonstrates the power and utility of such an interface. As our work supports standard CAD representations, as well as a wide variety of common simulation tasks, we believe our proposed techniques can already have an impact on how users interact with these all-important tools.

Bibliography

[1] H. Abbas, M. O'Kelly, A. Rodionova, and R. Mangharam. Safe at any speed: A simulation-based test harness for autonomous vehicles. In *7th Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy17)*, October 2017.

[2] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1016–1026, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. URL `https://doi.org/10.1145/3180155.3180160`.

[3] D. Agbodan, D. Marcheix, and G. Pierra. Persistent naming for parametric models. 2000.

[4] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 2004. ISBN 9780471653981.

[5] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. URL `https://doi.org/10.1023/A:1008739929481`.

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

[7] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In M. Irlbeck, D. A. Peled, and A. Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015. URL `https://doi.org/10.3233/978-1-61499-495-4-1`.

[8] R. Alur, A. Radhakrishna, and A. Udupa. Scaling Enumerative Program Synthesis via Divide and Conquer. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54577-5.

[9] American Fuzzy Loop. Technical "whitepaper" for afl-fuzz. URL `http://lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed: 2019-08-23.

[10] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In *TACAS 11*, volume 6605 of *Lecture Notes in Computer Science*, pages 254–257. Springer, 2011.

[11] Association for Advancement of international Standardization of Automation and Measuring Systems (ASAM). Opendrive, 2018. URL `http://www.opendrive.org/index.html`. Accessed: 2019-08-21.

[12] Association for Advancement of international Standardization of Automation and Measuring Systems (ASAM). Openscenario, 2018. URL `http://www.opendrive.org/index.html`. Accessed: 2019-08-21.

[13] A. Barth. Learnable Programming, 2019. URL `https://dpo.si.edu/blog/project-egress`.

[14] BBC. Tesla Autopilot: US opens official investigation into self-driving tech, 2021. URL `https://www.bbc.com/news/technology-58232137`.

[15] K. L. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002. ISBN 978-0321146533.

[16] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 63–74, 2016.

[17] B. Bettig and C. M. Hoffmann. Geometric Constraint Solving in Parametric Computer-Aided Design. *Journal of Computing and Information Science in Engineering*, 11(2), 2011. doi: 10.1115/1.3593408. URL `https://dx.doi.org/10.1115/1.3593408`.

[18] A. N. Bhagoji, W. He, B. Li, and D. Song. Exploring the space of black-box attacks on deep neural networks. *CoRR*, abs/1712.09491, 2017. URL `http://arxiv.org/abs/1712.09491`.

[19] S. K. Bhavnani and B. E. John. Exploring the Unrealized Potential of Computer-aided Drafting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '96, pages 332–339, New York, NY, USA, 1996. ACM. ISBN 0-89791-777-4. URL `http://doi.acm.org/10.1145/238386.238538`.

[20] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[21] M. Bokeloh, M. Wand, H.-P. Seidel, and V. Koltun. An algebraic model for parameterized shape editing. *ACM Trans. Graph.*, 31(4), July 2012. ISSN 0730-0301. URL `https://doi.org/10.1145/2185520.2185574`.

[22] BPLRFE. Youtube-Tutorial-Models, 2016. URL `https://github.com/BPLRFE/Youtube-Tutorial-Models`.

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

[24] B. Brüderlin and D. Roller. *Geometric constraint solving and applications*. Springer Science & Business Media, 2012.

[25] B. Canis. Issues in autonomous vehicle testing and deployment. Technical report, Congressional Research Service, 2021. URL `https://crsreports.congress.gov/product/pdf/R/R45985`.

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

[27] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188, 1987.

[28] X. Chen. *Representation, evaluation and editing of feature-based and constraint-based design*. PhD thesis, Purdue University, 1995.

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

[30] R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 341–354, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. URL `http://doi.acm.org/10.1145/2908080.2908103`.

[31] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In V. Scarano, R. D. Chiara, and U. Erra, editors, *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008. ISBN 978-3-905673-68-5. doi: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[32] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche*, 59 (1,2):125–172, 2004. ISSN 2037-5298. URL `https://lematematiche.dmi.unict.it/index.php/lematematiche/article/view/166`.

[33] comma.ai. openpilot: open source driving agent, 2016. URL `https://github.com/commaai/openpilot`. Accessed: 2018-11-13.

[34] G. B. Dantzig and M. N. Thapa. *The Simplex Method*, pages 63–111. Springer New York, New York, NY, 1997. ISBN 978-0-387-22633-0. URL `https://doi.org/10.1007/0-387-22633-8_3`.

[35] J. Deshmukh, X. Jin, J. Kapinski, and O. Maler. Stochastic local search for falsification of hybrid systems. In *ATVA*, pages 500–517. Springer, 2015.

[36] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51 (1):5–30, 2017. doi: 10.1007/s10703-017-0286-7. URL `https://doi.org/10.1007/s10703-017-0286-7`.

[37] J. V. Deshmukh, M. Horvat, X. Jin, R. Majumdar, and V. S. Prabhu. Testing cyber-physical systems through bayesian optimization. *ACM Trans. Embedded Comput. Syst.*, 16(5):170:1–170:18, 2017. URL `https://doi.org/10.1145/3126521`.

[38] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. URL `https://doi.org/10.1145/360933.360975`.

[39] A. Donzé. *Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems*, pages 167–170. Springer, 2010.

[40] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.

[41] T. Dreossi, A. Donzé, and S. A. Seshia. Compositional falsification of cyber-physical systems with machine learning components. In *NASA Formal Methods - 9th International Symposium, NFM 2017*, volume 10227 of *Lecture Notes in Computer Science*, pages 357–372. Springer, 2017.

[42] T. Dreossi, S. Jha, and S. A. Seshia. Semantic adversarial deep learning. 10981: 3–26, 2018. URL `https://doi.org/10.1007/978-3-319-96145-3_1`.

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

[44] S. Dutta, X. Chen, S. Jha, S. Sankaranarayanan, and A. Tiwari. Sherlock - A tool for verification of neural network feedback systems: demo abstract. In N. Ozay and P. Prabhakar, editors, *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019.*, pages 262–263. ACM, 2019. URL `https://doi.org/10.1145/3302504.3313351`.

[45] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 57–72, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133898. URL `https://doi.org/10.1145/502034.502041`.

[46] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*, pages 449–458. ACM, 2000. URL `https://doi.org/10.1145/337180.337240`.

[47] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2011 (VOC2011) Results. http://host.robots.ox.ac.uk/pascal/VOC/voc2011/index.html.

[48] G. Fainekos. Automotive control design bug-finding with the S-TaLiRo tool. In *ACC 2015*, page 4096, 2015.

[49] FeatureScript. Welcome to FeatureScript, 2020. URL `https://cad.onshape.com/FsDoc/`.

[50] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Vehicle platooning simulations with functional reactive programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, April 21, 2017*, pages 43–47. ACM, 2017. URL `https://doi.org/10.1145/3055378.3055385`.

[51] M. Fisher, D. Ritchie, M. Savva, T. Funkhouser, and P. Hanrahan. Example-based synthesis of 3d object arrangements. *ACM Trans. Graph.*, 31(6), Nov. 2012. ISSN 0730-0301. URL `https://doi.org/10.1145/2366145.2366154`.

[52] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001. URL `https://doi.org/10.1007/3-540-45251-6_29`.

[53] O. S. R. Foundation. Vehicle simulation in gazebo. URL `http://gazebosim.org/blog/vehicle%20simulation`. Accessed: 2019-08-23.

[54] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia. Scenic: A language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 63–78, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. URL `http://doi.acm.org/10.1145/3314221.3314633`.

[55] J. Fritsch, T. Kuehnl, and A. Geiger. A new performance measure and evaluation benchmark for road detection algorithms. In *International Conference on Intelligent Transportation Systems (ITSC)*, 2013.

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

[57] A. Gambi, T. Huynh, and G. Fraser. Generating effective test cases for self-driving cars from police reports. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.*, pages 257–267. ACM, 2019. URL `https://doi.org/10.1145/3338906.3338942`.

[58] A. Gambi, M. Müller, and G. Fraser. Automatically testing self-driving cars with search-based procedural content generation. In D. Zhang and A. Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019.*, pages 318–328. ACM, 2019. URL `https://doi.org/10.1145/3293882.3330566`.

[59] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.

[60] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 499–512, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. URL `http://doi.acm.org/10.1145/2837614.2837664`.

[61] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy, S&P 2018*, pages 3–18. IEEE, 2018.

[62] P. Girard. Your Wish is My Command. chapter Bringing Programming by Demonstration to CAD Users, pages 135–162. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-688-2. URL `http://dl.acm.org/citation.cfm?id=369505.369514`.

[63] C. Gladisch, T. Heinz, C. Heinzemann, J. Oehlerking, A. von Vietinghoff, and T. Pfitzer. Experience paper: Search-based testing in automated driving control applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 26–37, 2019.

[64] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *CoRR*, abs/1412.6572, 2014. URL `http://arxiv.org/abs/1412.6572`.

[65] S. Gulwani. "Programming by Examples: Applications, Algorithms, and Ambiguity Resolution". In N. Olivetti and A. Tiwari, editors, *Automated Reasoning*, pages 9–14, Cham, 2016. Springer International Publishing. ISBN 978-3-319-40229-1.

[66] F. U. Haq, D. Shin, S. Nejati, and L. C. Briand. Comparing offline and online testing of deep neural networks: An autonomous car case study. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 85–95. IEEE, 2020.

[67] B. Hempel and R. Chugh. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 379–390, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4189-9. URL `http://doi.acm.org/10.1145/2984511.2984575`.

[68] D. Ho, K. Rao, Z. Xu, E. Jang, M. Khansari, and Y. Bai. Retinagan: An object-aware approach to sim-to-real transfer. *CoRR*, abs/2011.03148, 2020. URL `https://arxiv.org/abs/2011.03148`.

[69] H. Ho, J. Ouaknine, and J. Worrell. Online monitoring of metric temporal logic. In *Runtime Verification RV 2014*, volume 8734 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2014.

[70] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. URL `https://doi.org/10.1145/363235.363259`.

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

[72] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety verification of deep neural networks. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 3–29. Springer, 2017. doi: 10.1007/978-3-319-63387-9_1. URL `https://doi.org/10.1007/978-3-319-63387-9_1`.

[73] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, 2002. URL `https://doi.org/10.1007/978-3-540-44833-4_6`.

[74] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, mar 2004. ISSN 0360-0300. doi: 10.1145/1013208.1013209. URL `https://doi.org/10.1145/1013208.1013209`.

[75] R. K. Jones, T. Barton, X. Xu, K. Wang, E. Jiang, P. Guerrero, N. J. Mitra, and D. Ritchie. ShapeAssembly: Learning to generate programs for 3D shape structure synthesis. *ACM Transactions on Graphics (TOG)*, 39(6):1–20, 2020.

[76] R. K. Jones, D. Charatan, P. Guerrero, N. J. Mitra, and D. Ritchie. Shapemod: Macro operation discovery for 3d shape programs. *arXiv preprint arXiv:2104.06392*, 2021.

[77] N. Kalra and S. M. Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 2016.

[78] Y. Kawai. Unirx: Reactive extensions for unity, 2014. URL `https://github.com/neuecc/UniRx`. Accessed: 2018-11-13.

[79] J. Kim, R. Feldt, and S. Yoo. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1039–1049, Piscataway, NJ, USA, 2019. IEEE Press. URL `https://doi.org/10.1109/ICSE.2019.00108`.

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

[81] J. Kripac. A mechanism for persistently naming topological entities in history-based parametric solid models. In *Proceedings of the Third ACM Symposium on Solid Modeling and Applications*, SMA '95, page 21–30, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897916727. URL `https://doi.org/10.1145/218013.218024`.

[82] D. R. Kuhn, R. N. Kacker, and Y. Lei. Combinatorial testing. In P. A. Laplante, editor, *Encyclopedia of Software Engineering*, pages 1–12. CRC Press, Nov 2010. ISBN 978-1-4200-5977-9.

[83] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes. Constructive solid geometry for polyhedral objects. *SIGGRAPH Comput. Graph.*, 20(4):161–170, Aug. 1986. ISSN 0097-8930. URL `https://doi.org/10.1145/15886.15904`.

[84] J. G. Lambourne, K. D. Willis, P. K. Jayaraman, A. Sanghi, P. Meltzer, and H. Shayani. Brepnet: A topological message passing system for solid models. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[85] D. Leen, T. Veuskens, K. Luyten, and R. Ramakers. JigFab: Computational Fabrication of Constraints to Facilitate Woodworking with Power Tools. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 156:1–156:12, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5970-2. URL `http://doi.acm.org/10.1145/3290605.3300386`.

[86] Z. Li, X. Ma, C. Xu, and C. Cao. Structural coverage criteria for neural networks could be misleading. In A. Sarma and L. Murta, editors, *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2019, Montreal, QC, Canada, May 29-31, 2019*, pages 89–92. IEEE / ACM, 2019. URL `https://dl.acm.org/citation.cfm?id=3339171`.

[87] J. Liberty and P. Betts. *Programming Reactive Extensions and LINQ*. Apress, 2011.

[88] M. Lipp, M. Specht, C. Lau, P. Wonka, and P. Müller. Local Editing of Procedural Models. *Computer Graphics Forum*, 38(2):13–25, 2019. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13615`.

[89] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015. URL `http://arxiv.org/abs/1512.02325`.

[90] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014. URL `http://arxiv.org/abs/1411.4038`.

[91] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 120–131, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5937-5. URL `http://doi.acm.org/10.1145/3238147.3238202`.

[92] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In *eXtreme Programming and Flexible Processes in Software Engineering - XP2000*, 2000.

[93] R. Majumdar and F. Niksic. Why is random testing effective for partition tolerance bugs? *PACMPL*, 2(POPL):46:1–46:24, 2018.

[94] R. Majumdar, A. Mathur, M. Pirron, L. Stegner, and D. Zufferey. Paracosm: A Test Framework for Autonomous Driving Simulations. In E. Guerra and M. Stoelinga, editors, *Fundamental Approaches to Software Engineering*, pages 172–195, Cham, 2021. Springer International Publishing. ISBN 978-3-030-71500-7.

[95] L. Makatura, M. Guo, A. Schulz, J. Solomon, and W. Matusik. Pareto gamuts: exploring optimal designs across varying contexts. *ACM Trans. Graph.*, 40(4): 171:1–171:17, 2021. URL `https://doi.org/10.1145/3450626.3459750`.

[96] MakerBot Industries, LLC. Thingiverse: digital designs for physical objects, 2021. URL `https://www.thingiverse.com/`.

[97] A. Mathur and D. Zufferey. Constraint Synthesis for Parametric CAD. In S.-H. Lee, S. Zollmann, M. Okabe, and B. Wünsche, editors, *Pacific Graphics Short Papers, Posters, and Work-in-Progress Papers*. The Eurographics Association, 2021. ISBN 978-3-03868-162-5. doi: 10.2312/pg.20211396.

[98] A. Mathur, M. Pirron, and D. Zufferey. Interactive Programming for Parametric CAD. In *Computer Graphics Forum*. © 2020 Eurographics - The European

Association for Computer Graphics and John Wiley & Sons Ltd, 2020. doi: 10.1111/cgf.14046.

[99] M. Mayer, V. Kuncak, and R. Chugh. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.*, 2(OOPSLA):127:1–127:28, Oct. 2018. ISSN 2475-1421. URL `http://doi.acm.org/10.1145/3276497`.

[100] E. Michel and T. Boubekeur. Dag amendment for inverse control of parametric shapes. *ACM Transactions on Graphics (TOG)*, 40(4):1–14, 2021.

[101] M. Mirman, T. Gehr, and M. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning (ICML)*, 2018. URL `https://www.icml.cc/Conferences/2018/Schedule?showEvent=2477`.

[102] Mockito. Tasty mocking framework for unit tests in java. URL `http://site.mockito.org`. Accessed: 2019-08-23.

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

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

[105] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–44, 2020.

[106] D. Neider, S. Saha, and P. Madhusudan. Compositional Synthesis of Piece-Wise Functions by Learning Classifiers. *ACM Trans. Comput. Logic*, 19(2):10:1–10:23, May 2018. ISSN 1529-3785. URL `http://doi.acm.org/10.1145/3173545`.

[107] H. Niederreiter. *Random number generation and quasi-Monte Carlo methods*. SIAM, 1992.

[108] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer Science & Business Media, 2004.

[109] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 2015.

[110] NVIDIA Coporation. Physx, 2008. URL `https://developer.nvidia.com/gameworks-physx-overview`. Accessed: 2018-11-13.

[111] Y. Okuya, N. Ladeveze, C. Fleury, and P. Bourdot. ShapeGuide: Shape-Based 3D Interaction for Parameter Modification of Native CAD Data. *Frontiers in Robotics and AI*, 5:118, 2018. ISSN 2296-9144. URL `https://www.frontiersin.org/article/10.3389/frobt.2018.00118`.

[112] OPEN CASCADE SAS. Open cascade technology, 2019. URL `https://dev.opencascade.org`.

[113] Open CASCADE Technology. Open cascade technology, May 2021. URL `https://dev.opencascade.org/`.

[114] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS 17*. ACM, 2017. URL `https://doi.org/10.1145/3052973.3053009`.

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

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

[117] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 1–18. ACM, 2017. doi: 10.1145/3132747.3132785. URL `https://doi.org/10.1145/3132747.3132785`.

[118] G. Pierra, J.-C. Potier, and P. Girard. "The EBP System: Example Based Programming System for Parametric Design". In J. C. Teixeira and J. Rix, editors, *Modelling and Graphics in Science and Technology*, pages 124–140, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-642-61020-2.

[119] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, 2009.

[120] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar. 1986. ISSN 0885-6125. URL `http://dx.doi.org/10.1023/A:1022643204877`.

[121] S. Rawat, V. Jain, A. J. S. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[122] ReactiveX. Reactivex. URL `http://reactivex.io/`. Accessed: 2019-08-23.

[123] S. R. Richter, H. A. Alhaija, and V. Koltun. Enhancing photorealism enhancement. *CoRR*, abs/2105.04619, 2021. URL `https://arxiv.org/abs/2105.04619`.

[124] R. T. Rockafellar and R. J.-B. Wets. *Variational Analysis*, volume 317. Springer Science & Business Media, 2009.

[125] G. Rote and R. Tichy. Quasi-Monte-Carlo methods and the dispersion of point sequences. *Mathematical and Computer Modelling*, 23:9–23, 1996.

[126] W. Ruan, X. Huang, and M. Kwiatkowska. Reachability analysis of deep neural networks with provable guarantees. In J. Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, pages 2651–2659. ijcai.org, 2018. doi: 10.24963/ijcai.2018/368. URL `https://doi.org/10.24963/ijcai.2018/368`.

[127] A. Ruano. DeepGTAV: A plugin for gtav that transforms it into a vision-based self-driving car research environment. `https://github.com/aitorzip/DeepGTAV`, 2017.

[128] H. Samimi, R. Hicks, A. Fogel, and T. Millstein. Declarative mocking. In *ISSTA 2013*, pages 246–256. ACM, 2013.

[129] S. Sankaranarayanan and G. Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *HSCC 12*, pages 125–134. ACM, 2012.

[130] A. Schulz, A. Shamir, D. I. W. Levin, P. Sitthi-amorn, and W. Matusik. Design and fabrication by example. *ACM Trans. Graph.*, 33(4), July 2014. ISSN 0730-0301. URL `https://doi.org/10.1145/2601097.2601127`.

[131] A. Schulz, A. Shamir, D. I. W. Levin, P. Sitthi-Amorn, and W. Matusik. Design and Fabrication by Example. *ACM Transactions on Graphics (Proceedings SIGGRAPH 2014)*, 33(4), 2014.

[132] A. Schulz, H. Wang, E. Grinspun, J. Solomon, and W. Matusik. Interactive Exploration of Design Trade-offs. *ACM Trans. Graph.*, 37(4):131:1–131:14, July 2018. ISSN 0730-0301. URL `http://doi.acm.org/10.1145/3197517.3201385`.

[133] A. Schulz, H. Wang, E. Grinspun, J. Solomon, and W. Matusik. Interactive exploration of design trade-offs. *ACM Transactions on Graphics (TOG)*, 37(4):1–14, 2018.

[134] S. Shah, D. Dey, C. Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017. URL `https://arxiv.org/abs/1705.05065`.

[135] M. Shugrina, A. Shamir, and W. Matusik. Fab forms: customizable objects for fabrication with validity and geometry caching. *ACM Trans. Graph.*, 34(4):100:1–100:12, 2015. doi: 10.1145/2766994. URL `https://doi.org/10.1145/2766994`.

[136] M. Shugrina, A. Shamir, and W. Matusik. Fab forms: Customizable objects for fabrication with validity and geometry caching. *ACM Trans. Graph.*, 34(4), July 2015. ISSN 0730-0301. URL `https://doi.org/10.1145/2766994`.

[137] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

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

[139] L. Stewart, M. Musa, and N. Croce. Look no hands: self-driving vehicles' public trust problem, 2019. URL `https://www.weforum.org/agenda/2019/08/self-driving-vehicles-public-trust/`. Accessed: 2021-01-18.

[140] I. Stroud. *Boundary representation modelling techniques*. Springer Science & Business Media, 2006.

[141] J. Sun, H. Zhang, H. Zhou, R. Yu, and Y. Tian. Scenario-based test automation for highly automated vehicles: A review and paving the way for systematic safety assurance. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[142] I. E. Sutherland. *Sketchpad, a man-machine graphical communication system*. PhD thesis, 1963.

[143] I. E. Sutherland. The ultimate display. In *Proceedings of the IFIP Congress*, pages 506–508, 1965.

[144] H. Suzuki, H. Ando, and F. Kimura. Geometric constraints and reasoning for geometrical cad systems. *Computers & Graphics*, 14(2):211–224, 1990.

[145] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *CoRR*, abs/1312.6199, 2013.

[146] C. Tang, X. Sun, A. Gomes, J. Wallner, and H. Pottmann. Form-finding with polyhedral meshes made simple. *ACM Trans. Graph.*, 33(4), July 2014. ISSN 0730-0301. URL `https://doi.org/10.1145/2601097.2601213`.

[147] M. Teichmann, M. Weber, J. M. Zöllner, R. Cipolla, and R. Urtasun. Multinet: Real-time joint semantic reasoning for autonomous driving. *CoRR*, abs/1612.07695, 2016. URL `http://arxiv.org/abs/1612.07695`.

[148] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, pages 303–314. ACM, 2018.

[149] C. E. Tuncali, G. E. Fainekos, H. Ito, and J. Kapinski. Sim-atav: Simulation-based adversarial testing framework for autonomous vehicles. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC 2018, Porto, Portugal, April 11-13, 2018*, pages 283–284. ACM, 2018. URL `http://doi.acm.org/10.1145/3178126.3187004`.

[150] C. E. Tuncali, G. Fainekos, D. Prokhorov, H. Ito, and J. Kapinski. Requirements-driven test generation for autonomous vehicles with machine learning components. *arXiv preprint arXiv:1908.01094*, 2019.

[151] Unity3D. Unity game engine. URL `https://unity3d.com/`. Accessed: 2019-08-23.

[152] K. Viswanadha, F. Indaheng, J. Wong, E. Kim, E. Kalvan, Y. Pant, D. J. Fremont, and S. A. Seshia. Addressing the IEEE AV test challenge with Scenic and VerifAI. In *IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 136–142. IEEE, 2021.

[153] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 242–252. ACM, 2000. URL `https://doi.org/10.1145/349299.349331`.

[154] W. I. Wan Din, T. T. Robinson, C. G. Armstrong, and R. Jackson. "Using CAD parameter sensitivities for stack-up tolerance allocation". *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 10(2):139–151, May 2016. ISSN 1955-2505. doi: 10.1007/s12008-014-0235-2. URL `https://doi.org/10.1007/s12008-014-0235-2`.

[155] M. Wicker, X. Huang, and M. Kwiatkowska. Feature-guided black-box safety testing of deep neural networks. In D. Beyer and M. Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 408–426. Springer, 2018. URL `https://doi.org/10.1007/978-3-319-89960-2_22`.

[156] A. F. T. Winfield, K. Winkle, H. Webb, U. Lyngs, M. Jirotka, and C. Macrae. Robot accident investigation: a case study in responsible robotics. *CoRR*, abs/2005.07474, 2020. URL `https://arxiv.org/abs/2005.07474`.

[157] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner. TORCS, The Open Racing Car Simulator. http://www.torcs.org, 2014.

[158] K. Xu, H. Zhang, D. Cohen-Or, and B. Chen. Fit and diverse: Set evolution for inspiring 3d shape galleries. *ACM Trans. Graph.*, 31(4), July 2012. ISSN 0730-0301. URL `https://doi.org/10.1145/2185520.2185553`.

[159] Y.-L. Yang, Y.-J. Yang, H. Pottmann, and N. J. Mitra. Shape space exploration of constrained meshes. *ACM Trans. Graph.*, 30(6):124, 2011.

[160] E. Yares. The failed promise of parametric CAD part 1: From the beginning, 2013. URL `https://www.3dcadworld.com/the-failed-promise-of-parametric-cad/`.

[161] D. Zufferey. Scadla rendering backend based on open cascade, 2019. URL `https://github.com/dzufferey/scadla-oce-backend`.

Curriculum Vitae

Research Interests

Computer Aided Design (CAD), 3D Simulation, Interactive 3D (Virtual/Augmented Reality), and Program Analysis/Synthesis.

Education

[2017–]	Doctoral Researcher, Max Planck Institute for Software Systems, Germany. Advisors: Damien Zufferey and Rupak Majumdar.
[2015–2018]	M.Sc. in Computer Science, University of Kaiserslautern, Germany.
[2010–2014]	B.Tech in Computer Science & Engineering, Rajasthan Technical University, India.