Design Space Exploration in Engineering Automation

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r I n g e n i e u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Constantin Chaumet

Dortmund

2025

Tag der mündlichen Prüfung:
22. September 2025

Dekan:
Prof. Dr. Jens Teubner

Gutachter:
Prof. Dr. Jakob Rehof
Prof. Dr.-Ing. Anne Meyer

technische universität
dortmund

TU Dortmund University
Department of Computer Science

DESIGN SPACE EXPLORATION
in Engineering Automation

Supervisor / TU Dortmund University
Prof. Jakob **Rehof**

Supervisor / Karlsruhe Institute of Technology
Prof. Anne **Meyer**

Committee / TU Dortmund University
Prof. Falk **Howar**
Prof. Peter **Buchholz**

Dissertation of
Constantin **Chaumet**

Affiliation
Research Group SEAL
Chair XIV for Software Engineering

To *Vina.*

Abstract

Two approaches that enable design space exploration and improve the automation of engineering tasks are developed. The first approach leverages combinatory logic synthesis with intersection types and predicates as an instrument to persist domain-specific knowledge held by CAD designers, i.e., how individual parts connect and can be composed, to automatically generate and assemble designs in CAD software. This enables an iterative exploration and reduction of the design space to relevant results, based on a set of performance metrics and structural constraints. The approach speeds up the creation of CAD assemblies while simultaneously improving quality by ensuring that assembly trees are uniformly organized and no joints are omitted. A set of parts suitable for synthesizing robotic arms is presented and used to perform a case study evaluating the practical applicability of the design space exploration and the overall performance compared to a human user.

The second approach enables the automatic generation of form-fitting mold jaws for parts with complex geometry. This approach allows these parts, intended to be additively manufactured, to be gripped by a vise for post-processing. This provides a rigid clamping solution that is easy to operate and reduces the necessary clamping force. The generated mold jaws may accommodate multiple different parts and/or the same part in different alignments. The approach can also ensure that geometry remains accessible to the machining tool and can account for manufacturing tolerances. The automatic generation of mold jaws is utilized to implement an optimization loop that determines Pareto-optimal rotations of each part or alignment around the cutting tool's axis with respect to clamping-induced deformation. Experimental validation is performed. The software aspects of the approach are validated by utilizing a topology-optimized part, whose optimal rotations are known a priori. The technological aspects are validated by manufacturing the part through selective laser sintering and inspecting the surface-to-surface contact area during clamping with a pressure-sensitive film.

Acknowledgements

My first and foremost thanks go to Jakob Rehof for bestowing upon me the chance to create this dissertation, diligently finding time every week to discuss his researchers' ideas, and giving me the freedom to work at the intersection of practical applications and theory as part of a fantastic research training group. I would also like to thank Anne Meyer for our many interesting talks and her numerous great tips and pieces of advice.

A special thank you goes to Lars Hildebrand, who has gone above and beyond as a mentor, providing insight and advice several times a week over lunch.

I greatly appreciate my cooperation partners Jan Liß and Simon Kammerer; working on interdisciplinary research together has been fantastic.

I thank Thomas Schuster for his consistently good work as my student assistant, and Jan Liß and Florian Wöste for conducting the experiments detailed in Section 10.2.2. I thank my research group for their immense patience in explaining theory to me. I would be remiss not to offer huge thanks to Jan Bessai and Tristan Schäfer, who helped get me started, the impact of which is hard to quantify but indubitably priceless.

I would like to thank Sevda Tarkun and Ute Joschko for their truly excellent work as secretaries, both of them saving me countless hours. They are pillars upon which our chair stands.

I am grateful to the German Research Foundation for funding Research Training Group 2193 and, consequently, me and my colleagues. Speaking of which, a big thank you to all of them for truly bringing the interdisciplinary research spirit to life. There is no doubt that the memories we made improved this thesis.

To my family and friends,
thank you for all the moral support,

And to Vina,
your love got this thesis over the line.

Contents

List of Figures

List of Tables

List of Listings

Part I

Overview

CHAPTER 1

Introduction

One of the fundamental skills of a good engineer is considering the available design space within which solutions to an engineering problem can be found and then selecting particularly suitable solutions from that space. While two engineers with differing levels of experience might be similarly adept at using the design and software tools that are common in their field of engineering, the more experienced engineer is better at recognizing the available design space and picking good solutions from it. This experience is an advantage, as, for instance, in mechanical engineering, iteratively making, testing, and re-engineering new solutions is time- and resource-intensive. Even designs implemented and refined solely in software take many hours of labor to create. This makes experienced engineers valuable, as they are able to pick out solutions from the design space without having to explore it through trial and error. However, reaching an intuitive understanding of what constitutes a good design takes many years, and for novel projects, there are no workers with such expertise.

As such, there are cases where considering the entire design space cannot be avoided. However, this can be taken further: why restrict exploration of the design space? The main reasons for restricting exploration are economic and practical limitations. These can be relaxed by tackling the problem at its root, developing ways to make exploring the design space efficient. This is desirable, as the shift to Industry 4.0 results in consumers expecting products to be customizable. Meeting this expectation provides a competitive advantage; however, only if products' costs can remain unchanged [1, 2]. Reducing development and labor costs through automating exploration of the design space allows prices to stay fixed while flexibility is simultaneously afforded. In many cases, it is not particularly time-intensive to decide if a design is good

or not; however, progressing to a state that permits judging a design's efficacy and making decisions becomes a bottleneck. This is especially true when relying on software tools, as operations within design tools are repetitive but poorly automated. The creation and adaptation of designs should allow pre-existing knowledge and experience to be applied seamlessly, without friction, and be time- and resource-efficient. While time and resource efficiency are ubiquitously considered when designing products, processes, and workflows, at the meta-level of exploring their design spaces, they are rarely considered.

The aim of this dissertation is to tackle the problem of automating design space exploration within an engineering and factory context, leveraging this to increase efficiency, accelerate the execution of design-related engineering tasks, and enhance the adaptability of factories. External factors often affect production. Reacting to these efficiently necessitates changes to different strata of the processes at play. For instance, changes in the supply chain may necessitate altering a product's design; this may then affect the production process, which in turn might incentivize changes to factory layouts, potentially leading to lower energy costs and improved cycle times. Factories that are able to adapt faster to external factors have a competitive edge. This provides a financial incentive for achieving flexibility, but capitalizing on it necessitates the technological means to quickly design new products, manufacturing processes, and factory layouts.

It is in this aspect that this dissertation provides solutions toward improving the efficiency of creating these new designs. Their creation is made faster, less human-resource intensive, and more robust, eliminating human error and superfluous interaction where possible. While an all-encompassing solution for any combination of product, process, and factory is desirable, it is infeasible to develop. Each factory is different, and the requirements of industry are broad and use-case specific; a completely generalist approach is far-fetched. Instead, factory-relevant use cases from different strata of the product design and manufacturing domains are selected, and methods that increase automation of their design-related aspects are developed. The selected use cases are product design from modular components with CAD software and clamping and post-processing of additively manufactured metal parts[1].

While these use cases are not all-encompassing, they are chosen to be broadly applicable and relevant to a wide swath of industries and factories. By providing methodologies and the corresponding software to improve the automation of these design problems, this dissertation highlights how to increase the adaptability of factories through automated design space exploration.

[1] And only briefly summarized in Section 1.1, the dimensioning of key infrastructure elements relevant to factory layouts.

1.1 Overview of Publications

In the following, the author of this dissertation's publications and the specific contributions made by the author within them are briefly summarized. They are listed in the order in which research on them commenced, the goal being to provide a timeline. As such, the summary of each publication touches upon how the contributions relate to each other.

Design Space Exploration for Sampling-Based Motion Planning Programs with Combinatory Logic Synthesis [112]

Motion planning is one of the fundamental requirements for accomplishing a wide range of tasks, especially in the field of robotics. It is one of the cornerstones of all practical applications since robots need to be controlled[1]. As an important part of robotics research, a plethora of different sampling algorithms and sample-based motion planning algorithms have emerged. Samplers and planners can be mixed and matched, leading to overall motion planning programs with varying performance. The prediction of an optimal combination for a use case is difficult, as a breadth of domain-specific factors can influence performance. Additionally, use-case-specific trade-offs need to be considered, often involving computation time versus the optimality of the computed motion plan.

To address this issue, the paper utilizes combinatory logic synthesis to generate all valid[2] combinations of motion planning and sampling algorithms and interprets the resulting inhabitants as motion planning programs. Since this removes all points of manual interaction in creating motion planning programs, the design space can be explored in an automated fashion. To accelerate the design space exploration, as grid searches of the design space are time-intensive, a black-box optimizer is employed to quickly find Pareto-optimal combinations of planning and sampling algorithms. The model learned by the optimizer is investigated regarding transferability.

The second use case within the paper highlights how this methodology is applied to find motion planning solutions for synthesized robotic arms in arbitrary environments. This is an application of the master's thesis of the author of this dissertation, in which functional robotic arms are automatically generated from static sets of typed components [31]. A key takeaway is that

[1] Be it mobile robotics or industrial robotic arms, any application necessitates knowledge of how to actuate the system to achieve a given task.

[2] Not every sampler can be combined with every planner.

there are no one-size-fits-all solutions; as such, an automated way of determining the best motion planning program for an environment is beneficial. This provides an example of how automatically generated robotic arms can provide value for real use cases. In combination with automatically determining good motion planning programs, a holistic solution can be achieved. The possibility of exploring all combinations of robotic arms, samplers, and motion planners can be used to find a Pareto-optimal solution for a given use case in an automated fashion.

Contribution The author of this dissertation has contributed:

- The automatically generated robotic arms for the second use case
- The environments for the second use case
- The visualization and execution scripts for the generated motion plans of the robotic arms
- The supplementary video material for the paper

CLS-CAD: Synthesizing CAD Assemblies in Fusion 360 [33]

A key issue that occurs during the CAD design process, compounded when engineering product families, is that many repetitive steps need to be performed manually. This is due to assemblies containing generic parts. Inserting and positioning these repeatedly is time-consuming and menial work, but also necessary. While most CAD software has some form of scripting or Application Programming Interface (API) support, the time needed to automate repetitive steps for a specific product family outweighs the benefits.

To address this issue, the paper proposes a methodology and provides an implementation for a user-friendly and use-case-agnostic add-in that automates insertion and alignment operations. To this end, connectivity between parts is expressed through intersection types, ordered within taxonomies. Combinatory Logic Synthesis is utilized to generate abstract representations of the resulting designs, which the add-in can assemble in CAD software, specifically in FUSION 360. Key benefits include increased efficiency in creating designs, leaving more time for creative work; a guarantee of correctness regarding the results; the elimination of mistakes that occur when humans perform repetitive tasks over elongated periods; and a thorough exploration of the design space, as the synthesis yields all possible valid designs.

The paper tackles some of the issues present in the methodology detailed in the previous section. Firstly, robotic arms could only be automatically generated from static sets of parts. This restriction is not due to the underlying synthesis mechanism, but rather the time-intensive task of adding specific code[1] to the robot-generating software [31]. Additionally, this means that only programmers with a solid grasp of mechanical engineering could add this code, creating a high barrier to entry. Secondly, the previous methodology is limited to creating robotic arm designs as polyhedral meshes. While this is sufficient for most robotics-related tasks, polyhedral meshes cannot be edited in CAD software. This makes the results unable to be reused or customized. As such, a human designer cannot apply final touches to the output. This motivates the creation of CLS-CAD, which eliminates these issues. The paper represents the first fully functional version of CLS-CAD, albeit with some restrictions regarding its applicability, further elaborated upon in the next sections.

Contribution The author of this dissertation has contributed:

- The methodology for CLS-CAD detailed in the paper.
- The implementation of CLS-CAD shown in the paper.
- The taxonomies, parts, and experiments presented in the paper.
- The narrated video demonstrating the implementation of CLS-CAD included in the paper.

Finite Combinatory Logic with Predicates [44]

A key issue with the previously detailed methodology is that constraints are difficult to encode using regular finite combinatory logic. This hinders the practical applicability of the methodology, as unconstrained design spaces are often large and thus difficult to explore. Disregarding the fact that automatically evaluating individual designs is a challenging problem[2], for large design spaces, the assembly time in the CAD software becomes prohibitive, despite still being orders of magnitude faster than manual creation. As such, being able to constrain results is of high importance; precise specification of subspaces within results yields more manageable amounts of designs.

[1] Code that defines intersection types, physical properties, and so on.
[2] Given that each design is a mechanical assembly with several hundred individual parts.

The research detailed in the paper is motivated by this issue of constraints, addressing it by extending finite combinatory logic with predicates, which allow encoding constraints in an elegant and computationally efficient manner. Two different types of constraints are introduced: literal predicates and type predicates. Literal predicates enable numerical constraints; for instance, the total weight or number of motors in an assembly can be constrained. Type predicates allow constraints of equality; for instance, this can be used to ensure that a robotic system uses the same standardized gripper throughout. While predicates do not allow combinatory logic to handle continuous real-valued metrics, in practice, discretization is usually sufficient.

Since the contents of the paper are primarily related to type theory, they will not be covered here, as only the practical application of these advances is of importance to this dissertation.

Contribution The author of this dissertation has contributed:

- The robotic arm use case shown in the paper.
- The repository structures comparing different modeling approaches.

A knowledge-driven framework for synthesizing designs from modular components [34]

This paper expands on the previously detailed work regarding Cls-Cad and represents the publication of its matured state. The previously given motivation remains the same: creating Cad designs from sets of modular components is inefficient due to repetitive and menial tasks. By encoding and persisting knowledge of how parts connect and their purposes, this encoded knowledge can be leveraged to synthesize all possible design alternatives. This shifts the role of designers away from performing menial tasks and toward applying their understanding of the design space to identify and refine the best auto-generated designs.

As detailed in the previous two sections, an issue with prior versions of Cls-Cad is their inability to handle constraints in an efficient manner, caused by the underlying synthesis mechanism: finite combinatory logic synthesis. The key contribution of this paper is that the presented version of Cls-Cad uses finite combinatory logic with predicates, changing the way that individual components are encoded and allowing discrete numerical constraints to be applied. In practice, this means that when generating design alternatives, it is now possible to specify which components and how many of them should be

contained. This allows many use cases from the field of robotics to be tackled, as the design space can be restricted to only those solutions with desired manipulability, effectors, sensors, redundancy, and so on. While the reduction of the design space is heavily dependent on the use case, in practice, for finite sets of results, two orders of magnitude are observed. Infinite sets of results usually become finite.

The paper also introduces a new way to interact with the design process through Cls-Cad, enabled by predicates. Since results can now be steered into specific sub-spaces of the design space by specifying constraints/predicates, designers can iteratively discover and refine these spaces. This makes it easier to achieve the best possible solution and provides a structured way of getting there. By first posing a less specific request and sampling some of the resulting designs, a coarse overview of the design space is quickly achieved. From there, the designer takes note of the features and metrics that are desirable or undesirable and fixes or avoids them in the next request accordingly. Oftentimes, just by looking at previews and bills of material of the generated designs, this first coarse overview phase can be completed. From there, this process is repeated, and with every additional restriction, the designer examines results in more depth, possible since their number is reduced. The designer uses analysis tools that come with the Cad software to verify and inspect the designs, then further refines the request. After a few iterations, the request yields only a handful of design alternatives, of which several can be picked and manually customized to yield a quality design, and as a result, a quality product.

On the technical side, another key contribution of this version of Cls-Cad is its ease of use, setup, and reproducibility of results it enables. Cls-Cad is now a streamlined add-in for a popular Cad software, Fusion 360, compatible with its cloud capabilities. It provides installers that enable setting up the add-in and back-end. A set of illustrated instructions for setting up and getting started with Cls-Cad is provided to accompany the paper. All necessary additional data needed for synthesis is encoded directly into the Cad files, which makes datasets easily shareable; for instance, the accompanying modular components for synthesizing robotic arms released alongside the paper [29]. Finally, the version of Cls-Cad in this paper features a plethora of performance enhancements. In experiments, it showed an average speedup of approximately two orders of magnitude in comparison to a human designer, both when creating individual or multiple designs.

In summary, the version of Cls-Cad contributed in this paper is suitable for real-world use cases, integration into typical design workflows, iterative design space exploration, and allows a full set of analysis tools to be run on the created designs.

Contribution The author of this dissertation has contributed:

- The repository structure and corresponding implementation to enable constraints for synthesis.
- The methodology for iteratively exploring a design space with Cls-Cad.
- The functionalities for sharing Cad files enriched with the data needed for synthesis.
- The open-source release of Cls-Cad with a full set of illustrated and animated instructions for setting up and reproducing results [28].
- The stand-alone dataset for synthesizing robotic arms [29].
- The real-world assemblies of the robotic arms and the manufacturing of the necessary parts.

Automatic generation of Pareto-optimal clamping solutions for post-processing additively manufactured parts [32]

One noteworthy aspect of the first paper detailed in this chapter is that it identifies an interesting use case for black-box optimization. Research into black-box optimization enjoys ongoing interest from the scientific community. Novel optimizers that offer particularly good optimization efficiency in specific use cases are still being released, e.g., TuRBO [46]. Especially for real-world use cases, black-box optimization methods like Bayesian optimization exhibit good performance. The goal of the paper covered in this section is to explore this avenue of research and provide an additional use case in which Bayesian optimization yields good results. This helps drive further development of Bayesian optimization methods by providing an additional real-world metric to benchmark against. It also serves as an initial step toward tackling a task that is currently not well automated but is of increasing importance.

This task is the clamping of additively manufactured parts. With the emergence of rapid prototyping techniques and the industry's ongoing interest in them, it is not uncommon for small-batch runs of parts[1] to be manufactured by means of Selective Laser Melting (Slm). The resulting parts have a coarse surface finish. For many applications of the parts, some of their surfaces need to be brought within tolerance. This is done by conventional milling; however, a major challenge in conducting the necessary milling operations is fixing

[1] More than one, less than 100.

the part. This stems from the often complex geometry of these parts, which may exhibit thin organic features that cannot take much clamping force. This necessitates manually designing a clamping solution; i.e., a technician will use several hand clamps and experiment with different setups.

This approach has some major shortcomings: it is dependent on the skill of the technician, the clamping solution is not very repeatable[1], and it does not guarantee an optimal clamping solution regarding deformation of the part. These shortcomings may endanger thin features, yield poor surface finish, and result in poorer tolerances.

The paper addresses these shortcomings by introducing a novel way to automatically generate form-fitting mold jaws through utilizing the Minkowski sum of polyhedral meshes. The mold jaws are computed so that the geometry to be milled is unobstructed. Form-fitting mold jaws can be used in combination with a conventional vise to reduce the skill required by technicians, as inserting and tightening a vise is easier than applying hand clamps. Vises with form-fitting jaws also result in high repeatability. Additionally, this reduces the necessary clamping force; an ideal mold jaw necessitates no clamping force at all. To achieve mold jaws that lead to close to optimal clamping performance, the paper connects the automatic generation of clamping jaws to Finite Element Analysis (FEA) software, utilizing a CAD kernel (OpenCascade) to convert from and to necessary file formats. By embedding this setup in a Bayesian optimization, the necessary total runtime within the FEA software needed while searching for the best way to embed the part into form-fitting mold jaws is greatly reduced as opposed to grid search[2]. The proposed methodology uses multi-objective optimization to account for multiple different surfaces that need to be milled. This is necessary due to the orientation of surfaces to be post-processed being fixed in all axes except w.r.t. that of the milling tool when performing three-axis machining. In turn, this necessitates multiple operations if surfaces to be machined do not share the same non-fixed axis.

The paper verifies the efficacy of the proposed methodology and corresponding implementation by means of an example part. The example part is topologically optimized to withstand forces from two specific directions. The multi-objective optimization automatically identifies precisely these two directions.

[1] This wastes time due to needing to probe each part when post-processing multiple in a row.
[2] Binary search cannot be used due to the non-linear relationship of orientation and maximum deformation of the part when clamped with a fixed load.

Contribution The author of this dissertation has contributed:

- The methodology and implementation for generating form-fitting mold jaws for polyhedral or CAD geometry.
- The implementation for automatically generating FEA simulations for a given part, mold jaws, and orientation.
- The methodology and implementation for performing Bayesian optimization based on FEA simulations.
- The experiments detailed in the paper, including the topologically optimized part used.

Optimizing Industrial Energy Systems: A Multi-Objective Approach to Decarbonization and Cost Efficiency [73]

The goal of this paper is to further explore applications of black-box optimization techniques for industrial scenarios. However, the focus is on tackling more macroscopic problems potentially relevant to a broad range of industries. One such problem is the increasing volatility of the energy market, driven by the increased focus on and availability of renewable energy. Nearly all large industrial facilities have a considerable energy demand, which significantly contributes to expenditure. As such, companies are intrinsically encouraged to optimize the way they obtain, store, and use energy. This challenges companies with questions of how to restructure their production and infrastructure to take advantage of off-peak times, during which energy prices are low. Another important factor that drives expenditure is the cost of grid capacity, which is the maximum concurrent power draw that a company pays for to be available. By optimizing assets and infrastructure, required grid capacity can be reduced, saving money in the process.

While the fiscal incentive to do this is apparent, it is unclear how to carry out such infrastructure and operations optimization. This is precisely the problem that this paper tackles. It introduces a methodology based on a combination of Mixed Integer Linear Programming (MILP) and Bayesian optimization. By modeling the different material and energy flows as a MILP problem, inputting energy market forecasts, and then optimizing to minimize costs and emissions, actionable recommendations of when to run which processes and how to dimension which variables in production can be obtained. However, this alone is not enough for real-world use cases, as with increased

model complexity and forecasting interval, the computation time for solving a MILP optimization problem scales poorly. Additionally, optimizing for long forecasting intervals leads to side effects, i.e., solutions that over-rely on specifics of the forecast. For real use cases, solutions must be able to respond and adapt to an updated forecast and be resilient. To achieve this, the forecasting intervals can be broken down into shorter, independent intervals. However, this independence also means that the optimization can pick different variables for each interval. In practice, companies cannot establish or discard assets at a moment's notice for each such interval, so the picked variables need to be consistent throughout all intervals, allowing a feasible course of action to be established.

To achieve this, the methodology proposed in the paper embeds the MILP optimization in a Bayesian optimization loop, and the variables, referred to as hyperparameters, are optimized by the loop. Each iteration of the outer optimization runs a series of MILP optimizations for different intervals and optimizes the hyperparameters based on the combined output of these. This allows finding hyperparameters that lead to more economical operation of industrial plants over extended time periods, without over-relying on specific quirks of individual months within the forecast, i.e., those caused by the different seasons of the year. Due to the MILP optimization only needing to optimize short intervals, the runtime is improved in comparison with a single long interval. Bayesian optimization further improves this by removing parameters from the MILP problem.

The contributed implementation consists of several major parts. Firstly, a use-case agnostic modeling tool is presented, which allows graphical and interactive translation of a production site or process into a MILP problem. Secondly, the modeling tool is connected to a tool capable of solving MILP problems (GUROBI) and parameterized so that input and output variables can be dynamically set. Finally, the setup and solving of the MILP problems are embedded into a state-of-the-art Bayesian optimization loop using Sparse Axis-Aligned Bayesian Optimization (SAASBO). Custom plotting scripts are implemented to visualize results and derive actionable recommendations.

Contribution The author of this dissertation has contributed:

- The methodology and implementation with regard to Bayesian optimization.
- The implementation of the plotting scripts.

1.2 Document Organization

This document is organized into a total of four parts, this part being the first. It introduces, motivates, and contextualizes the contents of the other parts. Necessary preliminaries for the remainder of the document are covered. The second and third parts each cover one of the two sets of contributions toward automating the exploration of the design space of common engineering tasks, with the second part pertaining to the CAD assembly design space, and the third showing how clamping solutions for additively manufactured parts can be explored. The fourth part summarizes the contents of the thesis, gives an outlook, and provides closing remarks.

Part I The first part is organized as follows:

- Chapter 1 introduces the thesis, summarizes the author's contributed papers, and explains the document organization.
- Chapter 2 motivates the two main subjects of the thesis, states the individual contributions made by the presented content, and covers the related work in the respective fields.
- Chapter 3 explains the necessary preliminaries to understand the thesis from an engineering perspective.

Part II The second part is organized as follows:

- Chapter 4 discusses the methodology to automatically explore the design space of CAD assemblies and automate their creation.
- Chapter 5 dives into the implementation of the methodology for a specific CAD software with a focus on performance and reusability.
- Chapter 6 covers developed datasets and experiments carried out with them to benchmark the presented implementation.
- Chapter 7 summarizes the contents of the second part and provides an outlook for further research.

Part III The third part is organized as follows:

- Chapter 8 discusses the methodology to automatically explore and identify Pareto-optimal solutions within the design space of clamping solutions for additively manufactured parts.

- Chapter 9 covers the implementation of this methodology, discussing software and framework choices, necessary formats and conversions, runtime and stability optimizations, and calibration for tolerances.
- Chapter 10 describes experiments carried out to validate the implemented software, its usefulness, practical applicability, and achievable clamping performance.
- Chapter 11 summarizes the contents of the third part and provides an outlook for further research.

Part IV The final part is organized as follows:

- Chapter 12 summarizes the contents of this document and provides an outlook regarding the overarching vision of more flexible factories.
- Chapter 13 gives closing remarks regarding this thesis.

CHAPTER 2

Contributions

This chapter serves as an introduction to the two main parts of this thesis. For each of the two methodologies and tools presented in this thesis, the current status quo and its shortcomings are examined in an informal manner. The goal is to motivate the necessity of solutions to the highlighted issues. Contributions each part provides to alleviate or eliminate these problems are stated. This is followed by a discussion of related work in each respective field.

2.1 CLS-CAD

Within this section, the need for a tool that enhances the automation of CAD software and enables thorough exploration of the CAD design space is motivated. It is stated what contributions the methodology and implementation of CLS-CAD provide to this end. Following this, related work regarding shape, geometry, and assembly generation is discussed.

2.1.1 Motivation and Contribution

From a computer science perspective, it is difficult to overlook the comparatively low degree of automation that common CAD software offers. Whilst the majority of physical products are designed in CAD software, there are a plethora of repetitive tasks that users have to slog through. A particular annoyance, which sparked the research that led to the development of CLS-CAD, is the tedium associated with inserting and aligning large quantities of basic parts with their respective mounting points, for instance, nuts, screws,

washers, and so on. Determining where to place the screws in an assembly is a task that requires little cognitive effort; however, it is time-consuming and repetitive. As such, it is a draining form of labor, accounting for roughly a third of designers' total labor [72]. The task is also prone to errors, its repetitive nature leading to mistakes such as designers overlooking screw holes, inserting screws of wrong lengths, or similar minor mistakes. Furthermore, these mistakes are difficult to spot. Noticing the presence of a screw that is of identical width as another but erroneously differs in length, whilst obscured by a screw hole, requires thorough checking of each and every fastener in the assembly. Such a check runs into much of the same problems. Usually, these mistakes are not catastrophic, getting caught when prototypes are first assembled; however, the costs add up. Mistakes can lead to additional manufacturing during prototyping, inventory mismanagement, and other downstream issues that are easy to rationalize as being part of the prototyping phase, while in reality being avoidable. The issue here is not that CAD software does not have tools to detect these issues, e.g., by checking for interference or examining cross-section views, but that it is better to avoid these issues in the first place. This saves time, resources, and allows workers to focus on meaningful labor.

Consider Figure 2.1, which depicts two robotic arms made from modular components side by side. Between them, these assemblies require 91 screws for the left robotic arm and another 209 for the right, totaling 300 screws to be inserted and aligned with their respective screw holes. In most CAD software, this results in at least 1200 mouse clicks[1].

While a skilled CAD designer will be more efficient, i.e., by using patterning tools[2], the number of clicks alone does not account for moving the camera so that selections can be made, which is also time-consuming, especially if target locations are occluded. After spending hours matching screws to screw holes and completing the left robotic arm, a designer will not look forward to completing the right arm, given the similarity of the tasks. As a direct consequence, when a new design is being made, it is common to use previous designs as a starting point and then retool them. This is undesirable, as it risks proliferating the shortcomings of previous designs.

The reason the described task is frustrating boils down to the fact that knowledge of how the individual components need to be connected is readily available but not efficiently utilized. Designers are aware that a screw needs to be mated with a screw hole of matching size and pitch, or that a

[1] A tool that handles alignment must be clicked; both points of reference must be selected, and then the operation must be confirmed, resulting in four clicks per screw.

[2] As a consequence, this usually results in not properly mating the screws to their corresponding holes, leading to a kinematically unsound model [134].

Figure 2.1: Example of two modular robotic arms

mounting bracket for a specific motor attaches to that corresponding type of motor. As such, it is difficult to overlook that the task of making the necessary mouse clicks is busy work. The aforementioned information is usually persisted in technical drawings of individual parts or in data sheets detailing exact purposes and connectivity options. These require a time investment to understand and extract the information necessary for the task at hand. It would be preferable to extract the necessary information once from any available source and then persist this information in a structured way that directly links it to the geometry of parts. This helps achieve replicability of results, not just in an academic context. Encoded information should be easily shareable, without the receiving party needing to re-encode or re-link all information to the parts. This way, designers can receive sets of parts, look at interfaces and mating geometry, and immediately ascertain which parts can and should be connected. The specification of such a structured form of storage should be precise and expressive, allow abstraction similar to how

engineering norms define standards, and be based on well-understood formal concepts. This enables mechanical engineers to encode traits of parts and their mating interfaces with precision, designers to easily identify how parts connect and accelerate their design work, and computer scientists to give guarantees of correctness and identify limitations.

While the benefits of a methodology that enables persisting knowledge and deriving designs from it are apparent, i.e., increasing the number of design iterations possible [4], there is little to no point if the "friction" of interacting with it is high. The aim is to eliminate repetitive tasks and ensure that engineers spend their time on creative tasks instead, not to substitute a prior tedium with a new one. The process of encoding knowledge regarding parts' purpose and their connectivity, and suitably structuring the result, needs to be user-friendly. It should not exclusively be approachable for computer scientists that deal with theory but instead be intuitive for mechanical engineers and designers to interact with and integrate into their typical workflows. The implementation of automatic design assembly should not be built on CAD software that does not see widespread industry use (i.e., FREECAD [121]) or user-interface-less libraries (CADQUERY [98]). While this would make implementation easier, it would prohibit industry-relevant use-case studies from being performed, hampering thorough verification of the methodology's efficacy. Thus, a good implementation option is as an add-in for a professional CAD software that sees industry usage, providing high-quality user interfaces, so that CLS-CAD can be practically tested. An added benefit of this is that the resulting add-in is interoperable with the wide breadth of mature analysis tools in CAD software that designers and mechanical engineers are used to.

It should be pointed out that there is another way of reducing "friction" between the methodology and designers. While oftentimes Artificial Intelligence (AI) can require expert knowledge to set up and interact with, it is possible to design such systems in a way that makes them easy to use, for instance, by adding a Large Language Model (LLM) layer like CHATGPT. However, the use of AI for such a methodology brings significant potential disadvantages with it. AI-based systems have non-perfect accuracy, generating solutions that contain small errors every now and then. Also, AI-based approaches are not yet capable of generating assemblies that consist of hundreds upon hundreds of parts [126, 132]. Since the goal of increasing automation within CAD software is to eliminate the need to manually check for misplaced parts in an assembly, it can be argued that an AI-based approach may currently do more harm than good. How to augment and improve methods that utilize AI to be robust and reliably correct is a field of research in and of itself, i.e., trustworthy AI. However, the focus of this thesis is on leveraging formal methods to achieve robustness and guaranteed correctness (with respect to the input data).

Contribution CLS-CAD's contribution is:

- Automating the insertion and alignment of parts, thus eliminating large portions of repetitive labor and vastly reducing the time needed to create a design.
- Enabling a thorough exploration of the entire design space as a consequence of the increase in automation and time efficiency.
- Enhancing reusability of parts and encouraging modular design by identifying a suitable formal model for persisting knowledge of how parts connect and making them easily shareable.
- Implementing the methodology for professional CAD software in a user-friendly manner, providing the means to easily set up the add-in and replicate results through installers and deployable containers, so that designers and academics can use and explore its applications with a low barrier to entry.
- Providing a set of modular components for synthesizing robotic arms and performing experiments to gauge the potential gains in efficiency.

2.1.2 Related Work

There is a significant body of work relating to shape, geometry, and assembly generation. Mo et al., Chang et al., Koch et al., Zhou et al., and Willis et al. provide large-scale datasets of parts that are useful for this purpose [27, 74, 91, 133, 139]. Of these datasets, those by Koch et al. and Willis et al. are the only ones to contain Boundary Representation (BREP) geometry, while exclusively Willis et al. provide construction sequences [74, 133]. Those by Chang et al. and Zhou et al. are labeled [27, 139]. There is also a dataset of only sketches without 3D geometry, but with construction sequences, which may be useful to determine connectivity between parts [114]. A large number of proposed methods operate on formats other than BREP geometry/CAD data, for instance, meshes, volumes, and point clouds. Out of these, only those approaches that operate in a structure-aware fashion are related to CLS-CAD, as for assembly generation parts must remain discernible. The work by Wang et al., Li et al., Mo et al., Wu et al., Huang et al., and Harish et al. falls into this category [61, 63, 81, 90, 130, 136]. CLS-CAD differs from these methods, being able to produce assemblies in CAD software, making the results reusable and customizable, and importantly, actuatable. There are also approaches that operate on meshes but can articulate the resulting

assemblies similar to proper CAD assemblies [56, 79, 84, 120, 136]. The methods that operate on point clouds and need to reconstruct meshes as a final step are not useful in a mechanical assembly context as geometry needs to be precise. In general, a downside of all these methods is that the results cannot guarantee correctness. CLS-CAD requires an initial modeling phase but can provide a limited form of such a guarantee. Considering the world of BREP geometry, there is an approach by Lambourne et al. toward segmenting geometry [77]. This can possibly be used to automate the type annotation that CLS-CAD requires. Most approaches that progressively build assemblies operate in a top-down fashion, learning placement, not connectivity [134]. Joints for real-world use cases are created bottom-up, with an assembly tree progressively built, e.g., starting with a robotic arm base and building toward the effector [134]. To the best of the author's knowledge, CLS-CAD is the only tool currently capable of automatically generating bottom-up style assemblies. There is some research into automating the creation of joints between BREP geometry parts and determining potential locations for JOINTORIGINS[1] [72, 134]. These methods improve over CLS-CAD, not needing any prior information that needs to be manually specified via the parts. However, they are presently only able to handle toy examples, e.g., connecting two to four parts, while CLS-CAD can automatically create assemblies consisting of many hundreds of parts. Finally, Heidari and Iosifidis give an extensive overview of further datasets and methods to facilitate CAD design processes [62].

2.2 OPTI-CLAMP

THIS section discusses the current state of clamping additively manufactured parts, highlighting aspects that are inefficient and thus uneconomical. Solutions to these aspects are discussed, and it is stated how OPTI-CLAMP contributes toward these solutions becoming feasible. Related work regarding post-processing and specifically clamping of additively manufactured parts is discussed.

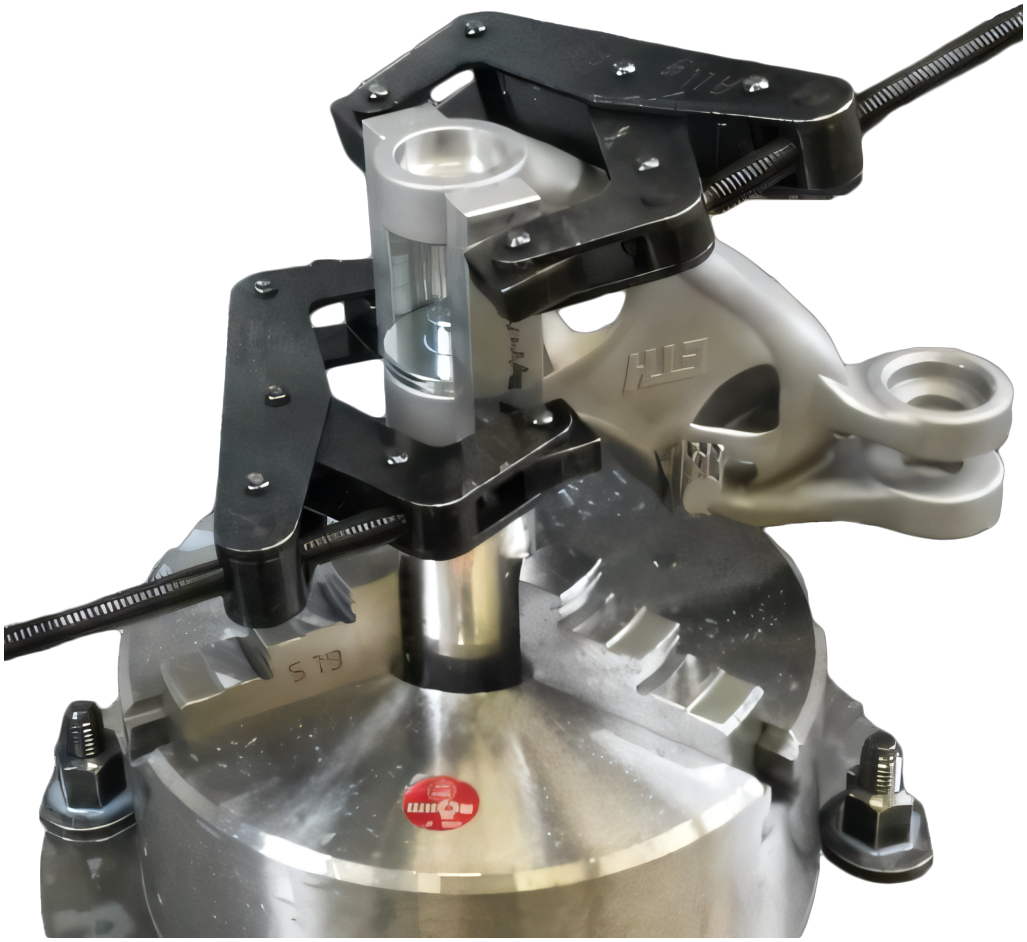
[1] Referred to as mating coordinate frames (MCFs) in the paper by Jones et al. [72].

2.2.1 Motivation and Contribution

With the increasing availability of additive manufacturing technologies able to process metals, it has become possible to manufacture very complex part geometries. Prior to these technologies, complicated geometries necessitated five-axis machining. This requires a skilled machinist with specific training to handle manufacturing such parts. Very complex geometries could not be manufactured at all, as features of the geometry might occlude each other so that a cutting tool cannot gain access. Laser powder bed fusion lifts these restrictions, allowing free-form surfaces and internal features to be manufactured from metal. This allows techniques like topology optimization, which generates part geometries based on the part's function while minimizing its mass, to be of practical relevance, as these optimized parts can now be manufactured. However, this comes with two downsides: manufacturing by melting or sintering together powdered material leads to a rough surface finish, and the dimensional accuracy is worse compared to conventional milling, as surface finish and thermodynamic effects lead to slight deformations. These downsides need to be accepted if the only way to manufacture a specific part is by powder bed fusion. For most of a part's features, the surface quality and dimensional accuracy may not matter, but for those features where technical drawings call out specific tolerances or surface finishes, the part must be post-processed. This is done by conventional milling, where a Computer Numerical Control (CNC) machine removes material, creating smooth surfaces and precise dimensions through computer-controlled movements with a cutting tool. For a CNC machine to post-process an additively manufactured part, it needs to be rigidly constrained relative to the machine, achieved by clamping the part. This is necessary to withstand cutting forces during machining. The more rigidly a part is constrained, the better the achievable surface quality, as the part vibrates less during the milling process. However, an issue arises when attempting to clamp such parts. Their complex geometry often encompasses thin or elaborate features, necessitating additive manufacturing. Precisely these features make the part difficult to clamp. Oftentimes, there are no planar surfaces, meaning clamping forces are not evenly distributed, instead being applied as point forces. Unevenly distributed forces and poor surface-to-surface contact then necessitate increased clamping forces to avoid the part slipping during the machining process. Increased forces can deform or break thin features of the part. Deformation also limits the achievable dimensional accuracy, as it is difficult for the machining process to account for it. Clamping additively manufactured parts is seen as the main hindrance preventing mass customization in the additive manufacturing of metal [75].

Figure 2.2 illustrates another issue that arises when clamping additively

Figure 2.2: Example of complex clamping solution [47]



manufactured parts. As there are no parallel planar surfaces or other geometric primitives, the part cannot be clamped in a conventional vise without risking deformation or breakage, as the vise invariably makes contact with features that cannot handle the applied clamping force. This effect is particularly pronounced in topology-optimized parts, as they are strong w.r.t. the directions of the loads they are optimized to bear, but the reduced mass makes them fragile when force is applied from other directions. Figure 2.2 depicts a clamping solution where two monkey clamps hold a part in place. Additionally, a central pin is inserted into the part, affixing it to a chuck. The clamping solution is customized to the part, avoiding the application of force to its thinner features. Creating such a customized clamping solution requires time from a skilled technician to set up. Additionally, if the part is produced in small to medium-sized batches, setting up each post-processing operation

costs more time than conventional clamping methods, as extra steps, like inserting the center pin, have to be carried out. Furthermore, the clamping solution is not very repeatable. This can be due to factors like deviations caused by the rough surface finish, play on the center pin, or mismatched forces applied by the clamps. As there are many potential causes for deviation, this necessitates probing each part before post-processing to ensure dimensional accuracy. The time for all these steps quickly adds up, and the post-processing quality is not as good as with more rigid clamping solutions, e.g., a vise.

A vise is the antithesis of a custom clamping solution: it is easy to use, even for laypeople, and rigid. From a technological point of view, a vise offers the best clamping solution. The use of vises for additively manufactured parts is not possible due to the lack of planar surfaces to interface with the vise's jaws. However, form-fitting mold jaws can enable the use of a vise. These jaws provide a negative of the part, their contour adhering to its features, creating a force- and form-fit [92]. Designing such mold jaws requires time and labor as well, typically discouraging this type of solution.

If the design of the mold jaws is automated, they become an attractive clamping solution. Using mold jaws, the required clamping force can be reduced. As the part is fully constrained by the jaws, there is no risk of it moving. However, there are two discouraging factors even for automatically designed mold jaws. The first is that they need to be manufactured, which costs money and engineering effort. There are approaches to additively manufacture the mold jaws to mitigate this issue [52, 92, 93]. The second is that the part may need to be post-processed in different orientations. Manufacturing mold jaws for each alignment of the part and needing to switch out jaws in between machining steps compounds the first issue and also costs time. Automatically generated mold jaws make designing a clamping solution accessible from a computer science perspective. This allows optimization methods to be employed, resulting in not only automatically designed mold jaws; but in jaws that hold the part optimally, i.e., rotated so that the clamping force causes minimal deformation. Designing jaws that accommodate a part in multiple alignments and consider the geometries' interdependence can also be tackled through optimization methods. The manufacturing cost of a single set of mold jaws still needs to be taken into account, but if producing a medium-sized batch, this cost is amortized by the higher throughput, reduced manual labor, and improved post-processing quality. To contextualize this, post-processing accounts for 20 to 40 percent of overall manufacturing costs [49]. OPTI-CLAMP automatically designs mold jaws in the described fashion.

Contribution OPTI-CLAMP's contribution is:

- Generating form-fitting mold jaws as BREP geometry suitable for FEA.
- Utilizing the generation of form-fitting mold jaws in conjunction with FEA software to optimize the clamping of additively manufactured parts.
- Implementing the methodology in a manner that allows technicians to use OPTI-CLAMP and evaluate the optimization progress.
- Performing sets of software-based and technological experiments that illustrate the advantages for production.

2.2.2 Related Work

The topic of post-processing additively manufactured parts is an important one, as the costs of post-processing account for 20 to 40 percent of total manufacturing costs [18]. Fixturing in isolation can account for ten to 20 percent [19]. The thesis by Ferchow provides extensive information on the topic [47]. The work by Bi and Zhang, as well as Bakker et al., gives an overview of a variety of fixture systems for additive manufacturing [8, 19]. These works cover automatic CAD design of fixture layouts, as well as modular and adaptive fixture systems. Many of these clamping systems appear complex to use. The work by Boyle et al. and Wang et al. specifically covers those solutions relating to CAD design [22, 127]. Neither of these comment on the usage of automatically generated form-fitting mold jaws. The jaws generated by OPTI-CLAMP are easier to use and cause less work for technicians than the more involved solutions covered. Additive manufacturing of fixture components has been previously considered [52, 92, 93]. Mumović et al. show that in some cases it may be sufficient to manufacture clamping jaws from polymers instead of metals [93]. There are also several approaches that suggest adding parallel surfaces [80, 113]. While this enables regular vises to clamp parts rigidly, Ferchow et al. argue that this restricts the design freedom afforded by additive manufacturing and requires topology-optimized parts to accommodate the clamping forces, increasing the necessary material [49]. This negates most of the advantages that additive manufacturing, in conjunction with topology optimization, offers. OPTI-CLAMP does not need to restrict the topology of the parts while also enabling vises to be used. There are several approaches to clamping additively manufactured parts that introduce sacrificial geometry [20, 119]. Downsides of these approaches are that sacrificial geometry needs to be manually placed and accounts for additional material. These approaches allow the usage of vises but restrict tool accessibility, which is

an important consideration [131]. OPTI-CLAMP solves this issue by (re)using obstacle geometry from topological optimization/generative design, enabling vises to be used while ensuring accessibility. There is a sophisticated semi-automated approach that introduces additional geometry (bolts) to the part for clamping [18, 49]. This bolt-clamping method allows affixing the parts to a common chuck, leading to good accessibility and ease of use for technicians. The achievable surface quality is good, and the part is not deformed by clamping. The clamping solution that OPTI-CLAMP offers is similarly easy to use but trades off more accessibility for rigidity. Similarly, there is an approach that connects parts to a sacrificial sheet metal layer for post-processing [48]. The trade-off between accessibility and rigidity is the key difference from OPTI-CLAMP with this method as well.

CHAPTER 3

Preliminaries

Several key concepts need to be covered ahead of the remainder of this document to provide necessary context and introduce techniques and tools relevant to the developed methodologies and software. This section aims to introduce the necessary preliminaries succinctly, making their purpose and usage intuitive, while not delving into details. For theoretical concepts, referenced literature within each section is chosen to be comprehensive.

3.1 CAD Software

CAD software, in general, is any kind of software that aids a user with the process of creating, modifying, or analyzing designs. Within the context of this thesis, CAD software specifically refers to the type of software utilized to create 3D models of products in mechanical engineering. Prominent examples are SolidWorks by Dassault Systèmes SE, as well as Inventor and Fusion 360 by Autodesk, Inc. [5, 6, 37], SolidWorks is the most used CAD software in industry, Inventor is the direct competitor to SolidWorks by Autodesk, and Fusion 360 is intended to phase out Inventor in the future. Fusion 360 is gaining popularity in industry[1], and is one of the most popular programs among hobbyists due to the existence of a free version with a reduced feature set.

Most CAD software operates in a sketch-based manner. This means that the user begins designing by creating a 2D drawing of a projection or cross-

[1] Usage and popularity estimated by querying how many jobs are being offered that list proficiency with the respective tool on job searching sites.

section of the object to be designed on a given plane. This sketch can then be given a third dimension by various means, e.g., by extruding it linearly along a given vector[1] or creating a solid of revolution by rotating the sketch around a given axis. This process can be repeated; however, the next sketch can be created on any planar surface present on the object that resulted from previous operations. Each operation that creates a volume from a sketch can be configured to either create a new solid, join the volume with existing solids (boolean addition), or subtract the volume from existing solids (boolean cut). The majority of CAD software additionally supports geometric intersection. Most CAD software also provides a plethora of patterning operations, which permit created geometry to be copied and repeated along paths or around an axis.

A typical workflow in most CAD software consists of first modeling atomic components, meaning models of individual parts as they will be manufactured. These parts are then arranged into ASSEMBLIES, which consist of multiple appropriately connected parts, e.g., a part with screws aligned to their respective screw holes. To do this, the user creates JOINTS. These constrain the motion of parts relative to each other. The simplest joint is a rigid joint, which rigidly connects parts, which then move as one unit. Typical examples of joints in robotics are revolute joints (pivoting around an axis) or prismatic joints (sliding along an axis). Assemblies can be nested and hierarchically ordered. The design of an object or product consists of one top-level assembly. Top-level assemblies may consist of individual parts and sub-assemblies. Sub-assemblies may, in turn, consist of individual parts or further sub-assemblies. This way, complicated objects are broken down into manageable smaller units, which can be assigned individual specifications and performance indicators and engineered to meet them.

A further important aspect of utilizing CAD software properly is parametrically designing parts, if supported by the software used[2]. Parametric models are not just saved as the resulting final geometry; instead, the entire design history is persisted, allowing for retroactive modifications of previous modeling operations. The model recomputes geometry when changes are made to previous modeling operations, succeeding as long as instances of geometrical features referenced for other operations still exist, i.e., only location, length, and other numerical values have changed. If computation fails, the user can redefine referenced geometry for problematic operations. Parametric design exploits this aspect of CAD software by defining numerical inputs to operations as variables instead of fixed values. This allows generic parts to be designed.

[1] The vector needs to be non-perpendicular to the sketch's plane's normal vector.
[2] Only a few and mostly obsolete software packages do not support parametric designs.

The same design can output parts compatible with a range of different sizes and patterns. A common example of this is bolt holes arranged circularly on a mounting plate. These can be spaced differently, and their number varied in this fashion.

3.2 Combinatory Logic Synthesizer

The Combinatory Logic Synthesizer (Cls) framework is capable of carrying out domain-agnostic synthesis from sets of modular components. It allows automating the creation of objects/artifacts by combining individual components according to predetermined rules guided by types, which encode the components' compatibilities. It is based on finite combinatory logic with intersection types, a logic that utilizes these types as a specification formalism, encoding compatibility and connectability between domain-agnostic components [106]. The inhabitation problem is solvable in Exptime for this logic [106]. The modular components are referred to as combinators in the following. Combinators form building blocks that can be assembled into compound structures. These structures are referred to as terms in the theoretical context.

Several implementations of the Cls framework have been created over time, ranging from implementations in programming languages and proof systems, like C#, F#, Java, Scala, and Python, or Coq [12, 15, 44, 105]. It is important to note that the latter three implementations are largely operating-system agnostic. As a result, they can run on a majority of systems[1]. Combinators wrap an arbitrary payload and are thus domain-agnostic. For instance, the Python implementation of the Cls framework can still wrap and output Java code. Due to the domain-agnostic nature of Cls, any use case abstractly expressible with intersection types is a potential target for synthesis.

The Cls framework has been applied to a plethora of different use cases, for instance, automatic code synthesis to reduce boilerplate coding work, cyber-physical systems, and for factory planning [13, 14, 16, 17, 88, 135].

Intersection Types The simplest form of a type is solely an identifier; for instance, Screw is a valid intersection type. Such simple types can be used to describe arbitrary constructs, serving to assign a telling name.

By introducing the intersection operator $\cap$, constructs can be given types of the form $\gamma \cap \tau$. These types encode that the construct is both of type γ and τ, meaning that in programming terms, it can be assigned to variables

[1] With the push towards increased usage of Python in the embedded system space this includes most microprocessors.

of either type. This form of specification intuitively matches well with the engineering sciences, where it is common to describe by combinations of attributes. An example of this is a metric screw. The screw is a screw, but it also follows metric standards, has a length, has a certain type of screw head, is made of a given material, and so on. This leads to a type like SCREW ∩ METRIC ∩ 3MM_DIAMETER ∩ 8MM_LENGTH ∩ SOCKET_CAP_HEAD ∩ STEEL. Such a type precisely encodes compatibility between different constructs, while also enabling a great deal of control over how much variance is permissible. For instance, assuming the existence of a catalog of screws described by such types, requesting synthesis of objects of type SCREW ∩ STEEL would lead to a large number of results[1], while SCREW∩3MM_DIAMETER∩8MM_LENGTH would narrow down the results to fewer screws but of varying materials[2].

Intersection types also allow using the arrow operator, intersection types of the form $\gamma \to \tau$. Interpreted in programming terms, this means that the type specifies a function that takes an argument of type γ and returns a value of type τ. Applied to the engineering sciences or specifically, mechanical construction, this can be read as: "This part/sub-assembly can be considered ready for use/assembled if a part compatible with γ is connected. Once that has happened, this part/sub-assembly can be used to complete an overarching assembly, which, in turn, requires a part/sub-assembly compatible with τ."

This system appears simple but is highly expressive. This is due to the fact that in all of these examples, γ and τ do not need to be simple types, but can be intersection types themselves. This means that these specifications can be combined and nested. The interpretation of intersection types in engineering terms matches the workflow described in Section 3.1, the arrow operator breaking a complex object to be engineered into layers of sub-assemblies, and the intersection operator encoding the specifications and performance indicators of each of these sub-assemblies.

$$\tau ::= \alpha \mid \tau \to \tau \mid \tau \cap \tau \tag{3.1}$$

For completeness, Equation 3.1 summarizes the rules for constructing intersection types, where α is an atomic type constant, and τ is a type variable/intersection type [42].

Subtyping The expressiveness of the previously described intersection types is further enhanced by introducing subtyping. Given a set of types, it can be ordered by taxonomies. A taxonomy is a directed graph, where each

[1] This is due to all steel screws of all combinations of lengths and diameters being valid answers to this query.

[2] This would be of importance when engineering for weight constraints.

node is an atomic type, and edges of the graph indicate that the type of an edge's target node can be used in place of any occurrence of its source node's type. This works analogously to how subtyping in object-oriented programming languages works, and is an expression of compatibility. Object-oriented paradigms with subtyping are well-suited to model engineering problems. This is evidenced by the following common textbook example of their usage from the engineering domain: Consider a class VEHICLE, of which a more specific class CAR is then a subtype, of which in turn specific brands or makes of cars are subtypes. This seamlessly translates to other aspects of engineering, as most engineering standards are conceptualized in this fashion. For instance, the International Organization for Standardization (ISO) publishes a standard for metric screw threads. This standard first defines the concept as abstract as possible, allowing for any combination of size and pitch. The standard then proceeds to specify specific profiles, and then further specifies common sizes, e.g., a typical M3 screw [66, 67, 68]. The taxonomies used by CLS allow for ALIASING; as such, taxonomies do not have to be acyclic. Cycles indicate that their constituent nodes are freely interchangeable names for the same concept. This can be useful from a management standpoint, e.g., if multiple teams work on a shared project, their differing terminologies can still be encoded in one shared taxonomy.

CLS performs subtyping on not just atomic types, but also *intersection types*. The specific subtyping used is dependent on the version of CLS [12, 44]. The complete set of subtyping rules is not necessary to understand the methodology behind CLS-CAD, as subtyping is fairly intuitive in a mechanical context. As long as each individual type of an intersection type has a matching counterpart reachable by traversing the taxonomy, intersection types can be viewed as compatible. The complete set of subtyping rules for intersection types can be found as definition 1.3 in the work of Dezani-Ciancaglini and Hindley, or an explanation can be read in section 2.3 of the dissertation by Bessai [12, 39].

Summarizing, subtyping on intersection types allows for a granular yet expressive specification formalism, permitting information about compatibility to be encoded by means of the taxonomy, and enabling separation of different concerns and attributes by means of intersection types.

Inhabitation Combinators, which within the context of this chapter can be understood as some form of data that has been assigned an intersection type, are collected in so-called repositories. A repository is a finite set of typed combinators without duplicates.

$$\Gamma \vdash ? : \theta \tag{3.2}$$

For such a repository Γ, synthesis can be defined in terms of Equation 3.2, which formalizes synthesis as a TYPE INHABITATION PROBLEM. Equation 3.2 can be understood as posing the question: "Given a repository Γ, can a term of type θ be derived by combinations of the combinators contained within Γ?" If this question is answerable with "Yes," then the set of all terms that satisfy θ is called the SET OF INHABITANTS, and each individual term is an INHABITANT.

The following three equations state the rules by which terms can be derived[1]. By chaining/combining these rules, all derivable terms can be found.

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \tag{VAR}$$

Rule VAR is straightforward, stating that the type of any combinator x is precisely the type that it has been assigned within the type context.

$$\frac{\Gamma \vdash M : \gamma \to \theta \quad \Gamma \vdash N : \gamma}{\Gamma \vdash MN : \theta} \tag{$\to$E}$$

Rule $\to$E states how two terms can be combined to make a new term. If there is a derivable term M which takes an input of γ and produces a θ, and if furthermore there is a derivable term N of type γ, then a new term MN can be derived, which is of type θ.

It should be noted that the terms M and N can also be combinators within the repository. Intuitively, this rule could be viewed as persisting information about how to combine terms, as once MN is derived, it does not need to be derived again.

$$\frac{\Gamma \vdash M : \gamma \quad \gamma \leq \theta}{\Gamma \vdash M : \theta} \tag{$\leq$}$$

Rule $\leq$ introduces the notion of subtyping to type derivation, as described in the previous paragraph on subtyping. If it is known that type γ is a subtype ($\leq$) of θ, then any term of type γ can be used in place of occurrences of θ in previous rules.

While it is simple to derive terms of new types using these rules, the type inhabitation problem, whether or not a term of a specific type exists, is, for the general case (Combinatory Logic with Intersection Types), undecidable [124]. Extensive research has been done to identify restrictions/fragments for which the inhabitation problem becomes decidable, for instance, Finite Combinatory

[1] Technically, there is a fourth intersection introduction rule, which is derivable from the other three [12].

Logic (FCL), FCL with boolean queries, FCL with predicates, and Bounded Combinatory Logic (BCL). The runtime depends on the specific fragment [42, 43, 44, 106]. The set of inhabitants that solves the type inhabitation problem is output in the form of a regular tree grammar. This is necessary to handle the oftentimes infinite set of inhabitants. Restrictions imposed by decidable fragments have little to no impact on the usefulness of synthesis by type inhabitation. Real-world problems are mostly finite or bounded. The plethora of applications already tackled by the different versions of CLS mentioned at the start of this section serve to underline that plenty of industrially significant use cases can enjoy automation by means of synthesis in practice.

Example To highlight the expressiveness of intersection types and give an intuition on how type inhabitation (hereinafter referred to as synthesis requests) can be used to explore design spaces in mechanical engineering, consider the following example.

$$\begin{aligned}&(\textsc{screw} \cap \textsc{metric} \cap \textsc{3mm\_diameter} \cap \textsc{lightweight}) \rightarrow \\ &(\textsc{propeller} \cap \textsc{40mm\_diameter} \cap \textsc{nylon})\end{aligned} \tag{3.3}$$

Type 3.3 shows how one might model a sub-assembly for the propellers of a small drone. The right-hand side of the arrow type is an intersection type specifying the resulting sub-assembly: a propeller of given wingspan, manufactured from nylon. For the sub-assembly to be completed and used in an overarching assembly, the propeller requires a screw. This screw needs to be of metric dimensions, three millimeters in diameter, and should be made from lightweight material[1]. The example assumes a taxonomy that orders and categorizes materials, like nylon, aluminum, and steel, into whether they are lightweight or not, for the purpose of subtyping. During synthesis, all different screws made from materials that are lightweight will be automatically considered.

This example highlights a benefit of modeling like this: separating the concerns of inventory and design. If there are no lightweight screws available, the set of inhabitants is empty. This indicates that the requirements cannot be met and discourages using an alternative screw for geometric modeling purposes as a workaround[2]. An engineer's job is simplified; calling out the specifications needed for the design as precisely as possible is sufficient. Omitting parts of the specification can cause undesired variance. For instance, in

[1] This additional requirement might occur for an aerospace-related application.

[2] Discouraging workarounds during the design stage leads to better quality products. Documentation that the model uses a placeholder screw that needs to be reconsidered once the correct parts are back in stock is easily forgotten and overlooked.

this example, all different lengths of screws will also be considered, as there is no type in the intersection restricting the length. The trade-off between creating correct type specifications and manual design creation is discussed in Section 6.3. It should be noted that applying subtyping to numerical identifiers is not always possible. This is the case if their values cannot be categorized into fixed discrete value ranges. However, often when numbers are involved in mechanical engineering, they are discrete, i.e., 3MM_DIAMETER, and desired variance can be achieved by grouping into fixed categories, i.e., 3TO5MM_DIAMETER. During early design stages, it is beneficial to consider dynamic value ranges for dimensions of parts, especially when designing parts parametrically. The next section details advances made in combinatory logic synthesis that permit such requirements to be expressed.

3.3 Combinatory Logic Synthesizer with Predicates

THERE are two key issues with CLS: individual combinators are monomorphic, which sometimes necessitates repositories containing combinators that are near duplicates, differing only in types that encode numerical values as literals; and the set of inhabitants is a regular tree grammar, which bars the specification of irregular properties, like the equality of subterms [44]. The Combinatory Logic Synthesizer with Predicates (CLSP) by and large abolishes these restrictions by extending the type system of FCL with quantifiers and predicates. While these additions result in the undecidability of the inhabitation problem in general, CLSP identifies and implements a decidable fragment of Finite Combinatory Logic with Predicates (FCLP) [44]. In practice, this permits the specification of equality and dis-equality on subterms, as well as a limited form of polymorphic typing utilizing literals.

$$\sigma \mid \langle \alpha : t \rangle \Rightarrow \varphi \mid \langle\!\langle x : \sigma \rangle\!\rangle \Rightarrow \varphi \mid P \Rightarrow \varphi \tag{3.4}$$

Equation 3.4 shows how this affects assigning types to combinators [44]. Types can now contain literal variables of the form $\langle \alpha : t \rangle \Rightarrow \varphi$, meaning that the variable has a fixed literal type t (usually INT, FLOAT, or similar), and will be bound (assigned a concrete value) in φ during inhabitation. These literal variables interact with the literal context Δ, which contains literals $\{l_1 : t_1, \ldots, l_n : t_n\}$. A literal variable's assigned value must be contained within the context Δ. This literal context enables a form of polymorphism. As the literal variables are assigned a concrete value during inhabitation, combinators that contain such variables can be considered parametric, differing precisely in the values of the literals.

Types can also contain term variables, for instance x in $\langle\!\langle x : \sigma \rangle\!\rangle \Rightarrow \varphi$. These work similarly to literal variables, except this time the fixed type of the variable can be an intersection type. Again, the variable is bound in φ during inhabitation, but instead of being assigned a value from the literal context, an entire subterm of compatible type is assigned to the variable. The advantage of CLSP stems from the final addition, $P \Rightarrow \varphi$. Variables previously introduced in the types can be utilized within predicates P. These may contain any statements that compute a boolean value and may utilize the variables[1].

This results in a set of three additional rules for type inhabitation, supplementing Rule VAR, Rule $\rightarrow$E, and Rule $\leq$. These rules are straightforward and are thus only briefly covered here. For more information, the paper on FCLP and the notion of the pure type system application rule can be consulted [11, 44].

$$\frac{\Gamma; \Delta \vdash M : P \Rightarrow \varphi \quad P \textit{ holds}}{\Gamma; \Delta \vdash M : \varphi} \tag{PE}$$

Rule PE states that to use a given combinator that contains a predicate P, the predicate must hold (evaluate to true). If it does, the term M can be derived, with the type φ, and the predicate removed.

$$\frac{\Gamma; \Delta \vdash M : \langle \alpha : t \rangle \Rightarrow \varphi \quad (l : t) \in \Delta}{\Gamma; \Delta \vdash M\,l : \varphi[\alpha := l]} \tag{$\langle\rangle$E}$$

Rule $\langle\rangle$E expresses the notion that if for a literal variable α there exists a literal value l of matching type within the literal context Δ, then the application $M\,l$ can be added to the type context. This application yields the type of M, but all instances of α are replaced by the concrete value l. Intuitively, this rule means that a literal variable can be substituted with literal values of compatible type from the literal context.

$$\frac{\Gamma; \Delta \vdash M : \langle\!\langle x : \sigma \rangle\!\rangle \Rightarrow \varphi \quad \Gamma; \Delta \vdash N : \sigma}{\Gamma; \Delta \vdash MN : \varphi[x := N]} \tag{$\langle\!\langle\rangle\!\rangle$E}$$

Rule $\langle\!\langle\rangle\!\rangle$E is analogous to rule $\langle\rangle$E, but instead of the literal context, it uses the type context. If there exist terms within the type context, either constructed during synthesis or initially part of the repository, of correct type to substitute σ, then all such valid applications MN are added to the type context, with the variable substituted by the corresponding subterm.

Combining these three rules leads to a system where the results of an inhabitation request contain only those terms for which all variables can be substituted in a way that satisfies all predicates. Since predicates can

[1] In practice, this means that arbitrary code can be run during the synthesis process.

contain arbitrary code, they can be use-case specific. This system can be used to concisely specify engineering requirements by using literal variables, restricting the set of inhabitants more than is possible with regular Cls. It allows encoding requirements of equality and dis-equality, which is also not possible without utilizing ClsP. The extent to which this can reduce the amount of necessary "boilerplate" combinators, and thus improve runtime, is covered in Section 4.4 and Section 4.5.

3.4 Kinematics of Mechanisms

Individual parts form the basis of Cad designs and can be assembled into complicated assemblies by means of joints, as described in Section 3.1. What parts, joints, and assemblies are to Cad design, links, joints, and kinematic chains are to general mechanical engineering. Objects initially have six Degrees of Freedom (Dof); they can be translated and rotated along the X, Y, and Z-axis of a given coordinate system, meaning they can assume any position and orientation.

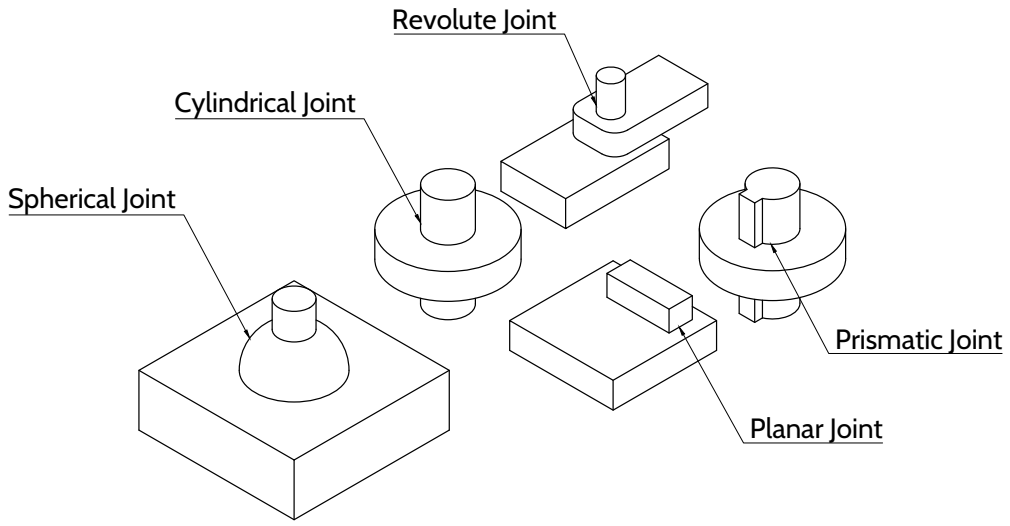
Links A link is considered to be a set of two or more parts that are fully constrained to each other. The parts have zero Dof in relation to each other; they move as one rigidly connected group [107]. For instance, after tightening a screw, it becomes part of the link to which it has been tightened. In terms of robotics, a link can be thought of as the coarsest unit into which a robotic system can be split, with its motion still fully described.

Joints A joint is considered any form of connection between two links that constrains their motion relative to each other but permits at least one Dof between the links. Joints can be categorized into lower pair and higher pair joints [107]. Higher pair joints are those where the contact between links is point or line shaped. Lower pair joints are those with a contact area. Lower pair joints are those of practical importance in robotics[1]; as such, the following overview only covers these.

Figure 3.1 provides an overview of five out of the six possible lower pair joints [36]. Screw joints are omitted, as they can be considered a combination of revolute and prismatic joints. Joints that allow exactly one Dof are prismatic joints (movement along one axis, no rotation allowed) and revolute joints (rotation around one axis). Examples of joints that allow for two Dof are cylindrical joints (movement along an axis, rotation around the same axis)

[1] This is largely related to torque transmission.

Figure 3.1: Overview of lower pair joints



and planar joints (movement along two axes). Planar joints for which rotation around the plane's normal vector is not restricted provide three DoF. Finally, a spherical joint provides three DoF, allowing rotation around all three axes of a coordinate system.

Kinematic Chains When two or more links are connected to each other by means of joints, they form a kinematic chain, and through pinning (reducing the DoF to zero) of one link within the chain, a functioning mechanical system is obtained [107]. Kinematic chains are categorized into open and closed chains. If we consider joints that connect links as nodes in a graph and links as the edges between nodes, open kinematic chains are those graphs that are acyclic, and closed kinematic chains are those that are cyclic [116]. Since Cls and ClsP both produce results in the form of tree grammars, trees being inherently acyclic, representing closed kinematic chains in a use-case agnostic manner is an open problem. As such, closed kinematic chain systems are not presently considered for synthesis by Cls-Cad. However, the majority of robotic systems are covered by open kinematic chains, evidenced by the Unified Robotics Description Format (Urdf), a popular specification language for describing robots [96, 109], which does not support closed kinematic chains either.

Kinematic chains can additionally be categorized into branched and unbranched chains. Again considering kinematic chains as graphs, any chain with a node that connects to more than one other node is considered a

branched kinematic chain. Chains where each node has between one and zero children are considered unbranched [116]. Unless further specified, the term "kinematic chain" refers to an open and unbranched kinematic chain [116]. Typical examples of this are robotic arms. Robotic systems that locomote are usually open but branched kinematic chains.

3.5 Robot Operating System

The Robot Operating System (Ros) was developed to tackle the increasing complexity of creating firmware for robotic systems [103]. This increase in complexity stems from the growing amount of heterogeneous hardware that needs to function coherently. The Ros and more modern Ros2 tackle this issue by providing the de facto standard software frameworks for robotics research, thus promoting standardization [87, 103]. The Ros supports many different programming languages, for instance, C++, Python, Octave, and LISP. This makes the Ros a good target for potential synthesis of matching control software if Cls-Cad is used in a well-defined use case, e.g., if only generating members within a single product line. Part of the Ros ecosystem is a plethora of preexisting packages, which reduce the need to create custom implementations for common tasks.

3.6 Unified Robot Description Format

In the same vein as the Ros, the Urdf seeks to encode kinematic structure, collision properties, and visual models of robots, which can be expressed as open kinematic chains [109]. The file format is an XML schema, defining tags for joints, links, colors, and so on. These tags can have nested tags, providing information on moments of inertia, visual geometry, collision geometry, and other important numerical properties.

Once a robot has been encoded in Urdf, there are many commercial and open-source projects that can load and utilize it, offloading large portions of the effort to create control software or other artifacts. A popular one of these is MoveIt! [35], which allows collision-aware motion planning for the robot, enabling synthesized systems to be used within (industrial) use cases.

3.7 Additive Manufacturing

Additive manufacturing, often also referred to as rapid prototyping, describes the manufacture of three-dimensional objects by bonding to-

gether material, commonly in layers [57]. There are several different additive manufacturing technologies. They can be broadly sorted into three categories: solids or pastes, which are melted or glued together by extrusion; liquids that are polymerized; or powder that is sintered, melted, or bound [57]. These come with several different advantages and disadvantages, differing in achievable surface finish, production speed, and expense [123].

Throughout this thesis, Laser Powder Bed Fusion (LPBF) processes like SLM and Selective Laser Sintering (SLS) are utilized as manufacturing methods. Both of these fall under the powder category and are categorized as comparatively inexpensive [123]. They operate by stacking thin layers of powder. Thermal energy is then applied to regions of the powder bed intended to be fused [123]. In this way, solid layers are built up, and arbitrary geometry can be manufactured. These processes often have no need for support structures, due to the parts being held in place during production by the powder that is not fused. However, some SLM machines do require supports. The surface quality of the manufactured parts depends on the particle size of the precursor powder [123].

Additive manufacturing techniques generally allow parts to be designed with more freedom than subtractive techniques like CNC machining [85]. This makes them suitable for complex parts, especially since SLS has close to no restrictions regarding geometries. This enables the manufacturing of topology-optimized parts. Topology optimization is a computational design method that distributes material in a given design domain. The optimization minimizes given objectives, for instance, mass, while satisfying a set of constraints. These constraints commonly define regions where no material may be present, regions where material must be present, and features that are considered fixed. The design domain consists of defined loads, for instance, gravity, expected torque the part must withstand, and so on. Topology optimization often yields parts consisting of complex free-form surfaces with innovative topology [85].

3.8 Finite Element Analysis

At its core, FEA is a method to solve systems of partial differential equations, first used in 1941 to analyze a membrane and plate model [86]. Most physical processes can be expressed as a system of partial differential equations, making FEA a powerful tool to analyze the behavior of all sorts of systems, for instance, material and structural behavior, fluid dynamics, thermodynamics, circuit board design, and many more [86]. By setting up partial differential equations describing a system and performing FEA, different scenarios can be simulated and their outcomes tested without the need to perform experiments. Especially for complex engineering projects, which fall out of the realm of analytical methods, FEA is indispensable to realize them without astronomical cost. In modern times, engineering a complex solution without making use of the predictive analysis that FEA offers is unthinkable from a cost-management point of view.

To set up the necessary partial differential equations, FEA/Finite Element Method (FEM) software is used. Such software offers a user interface that allows loading files, applying boundary conditions, initial conditions, friction, forces, and so on. In a mechanical engineering context, a common use case is examining deformation and stresses within geometry when it is exposed to a given force. By solving the partial differential equations, the behavior of the part can be examined. This is used as a feedback loop; for instance, the results can be used to determine whether a part needs to be further reinforced or if its mass can be reduced while still meeting technological requirements. As the analysis uses finite elements, geometry needs to be discretized into polyhedral meshes. The resolution of these meshes has a large impact on the accuracy and runtime of the FEA, with a higher resolution improving accuracy but increasing runtime. Technical details of the mathematics behind FEA are not directly relevant to the contents of the dissertation, but the work of Liu et al. provides a comprehensive overview and set of references [86].

3.9 Bayesian Optimization

Bayesian optimization allows for optimizing an objective function by sampling it. Commonly, this is a derivative-free function, meaning no first- or second-order derivative can be computed[1] [55]. Also, the function is assumed to be a black-box function, meaning that no domain-specific information is known that could be used to optimize in a simpler fashion, for instance, linearity [55]. Compared to other methods of optimization, Bayesian optimization differs by being computationally more expensive but requiring fewer samples. This makes Bayesian optimization a good choice for use cases where the objective function is expensive to evaluate. The cost of Bayesian optimization should be outweighed by the time savings of needing to compute the objective function fewer times.

Bayesian optimization requires two major components: the computation of a surrogate model with quantifiable uncertainty of its predictions[2], and an acquisition function, which selects new values to sample the objective function at[3]. The technological and theoretical details of how Bayesian optimization operates are not required for this thesis and are not covered here. A good starting point for the interested reader is the works of Frazier and Wang et al. [55, 128]. Relevant frameworks that can be examined for technological and implementation details of these techniques are Optuna, Ax, BoTorch, OpenBox, SMAC3, and Auto-Sklearn[4] [3, 9, 10, 50, 51, 71, 82, 83].

3.10 Minkowski Sum

The Minkowski sum operator is defined on two sets of position vectors in Euclidean space. It is computed by adding each vector within one of the sets to each vector in the other set.

$$A + B = \{a + b \mid a \in A, b \in B\} \tag{3.5}$$

In two-dimensional space, assuming that A and B are both areas, this can be thought of as the shape described by sliding one area around the contour of the other. Applications of the Minkowski sum are numerous; it mathematically describes practical problems such as collision-free path planning

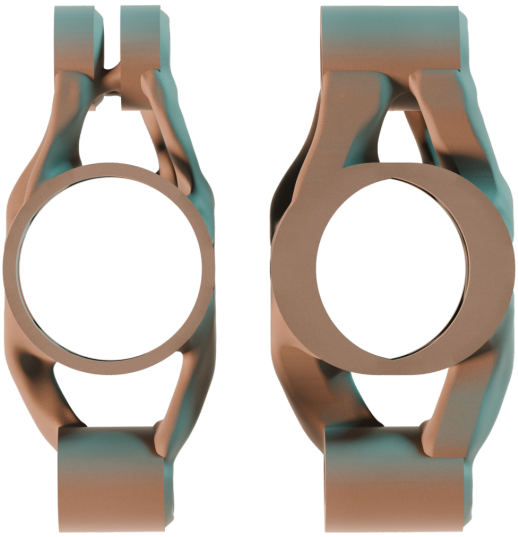
[1] If the derivative is available, there are better optimization methods, e.g., gradient descent [55].

[2] Commonly a Gaussian process.

[3] Commonly expected improvement, EI.

[4] The list is non-exhaustive and should not be considered as a particular endorsement or ranking of these frameworks. Neither should any framework not on the list be discounted.

Figure 3.2: Example of Minkowski sum of topology-optimized part with a vector

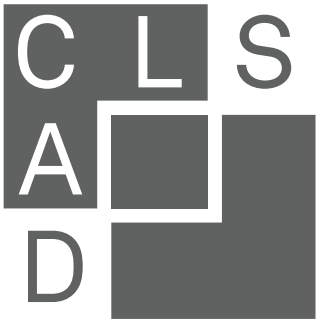


in robotics[1], CNC milling operations[2], as well as offsetting of geometry [125]. Figure 3.2 illustrates the result of performing Minkowski addition between three-dimensional geometry and a linear vector, horizontally oriented. This addition can be interpreted as the tool path created by utilizing the part geometry as a cutting tool. This interpretation highlights that a Boolean cut between any geometry and the result of a Minkowski sum, between a part and a path, allows the part to be freely and unobstructedly moved along that path. The technical implications of this are that the part is guaranteed to be insertable into the geometry.

[1] The Minkowski sum of the path with the model of the robot.
[2] The Minkowski sum of the cutting tool with the tool path.

Part II

CLS-CAD



CHAPTER 4

Methodology

In this chapter, the conceptualization, design space exploration workflow, and repository structures of Cls-Cad are explained. The methodology entails creating use-case tailored taxonomies, annotating the geometric features of physical parts that interface and connect with other parts, and utilizing this information to dynamically generate repositories that take user constraints into account. From these repositories, the Cls framework generates the set of all valid solutions as abstract terms, which can then be interpreted and assembled into real designs in Cad software. A key consideration is designing an architecture and tool that minimizes dependence on the Cad software used. While each Cad software exposes a different set of functionality to programmers and their interfaces are structured in different ways, by choosing a suitable architecture, functionality can be reused in other Cad software. The methodology is laid out so that it translates typical good engineering practices into steps to follow when working with a Cls-based system, which reduce the design space iteratively before time-intensive processes begin. The ClsP is utilized to further augment and streamline the workflow.

Chapter Organization This chapter is broken down into a general overview of the chosen architecture, a walkthrough and explanation of the intended workflow, and a comparison between the resulting repository structure with and without predicates.

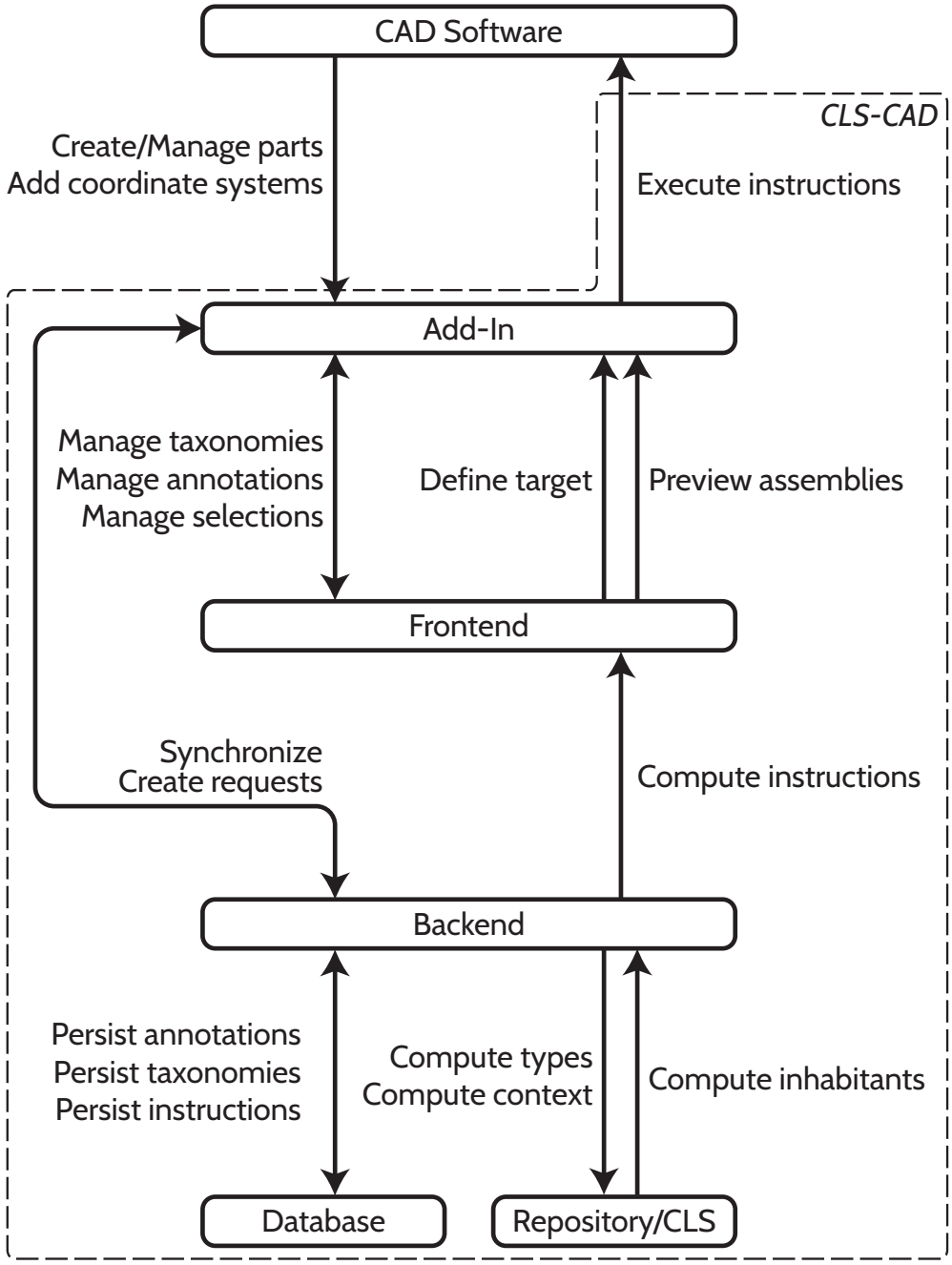
4.1 Architecture

As mentioned in Section 2.1.1, one of the core issues with other approaches toward increasing automation of the CAD design process is that they usually exhibit a high barrier to entry. Commonly, at least some form of programming expertise is required; oftentimes, a deep understanding or at least familiarity with underlying scientific concepts is also desirable[1]. The issue is that these assumptions cannot be reasonably made about designers using CAD software. As such, the architecture of CLS-CAD is designed to meet users close to their area of expertise.

As can be seen in Figure 4.1, the top level of the architecture is the CAD software itself. Arrows can be read as meaning that their source computes or creates data, while their target displays or further processes received data. Designers are typically intimately familiar with their CAD software, using it not just to design parts and assemblies but also to manage all sorts of meta-data, like materials, notes, etc., and construction geometry, like virtual axes, points, and coordinate systems. The tools to create and manage these in CAD software are developed and refined by major companies over several decades, with extensive consideration given to their design, making operations efficient while being simple and effective to use. As such, any architecture that does not attempt to make use of and instead recreates these tools is tantamount to reinventing the wheel. However, since designers use different CAD software, and reinventing the wheel is out of the question, it follows that at least one layer of the architecture must be specific to the concrete CAD software being used. In Figure 4.1 this is the "Add-In" layer. It has two major purposes: the first being to collect and set up data necessary for the automatic generation of assemblies, and the second being to instrument the CAD software such that it can carry out instructions that lead to creating assemblies within the software. To achieve the first purpose, the add-in interfaces with the CAD software, collecting user input through pre-existing features wherever possible. In practice, this means that instead of implementing a user interface that allows the user to manually define a connection point or coordinate system, the add-in prefers using an equivalent construct or tool within the CAD software; for instance, in the case of FUSION 360, the joint tool and construction geometry. This approach is possible for any CAD software capable of understanding the STEP file format, since entities displayed by the user interface must be translatable to STEP files. In the absence of suitable tools within the API of the CAD software, annotations to entities can be realized by means of comments

[1] Consider AI driven approaches, where to make good use of them one needs to have experience with what good prompts are and how to phrase them.

Figure 4.1: Overview of key elements of methodology and their interactions



in the STEP file [65]. This is possible due to STEP files assigning each entity within them a unique numerical identifier, which comments can reference. The implementation detailed in Section 5.1 does not need to resort to this,

as Fusion 360's native file format has an inbuilt annotation system, which is part of the reason for its selection.

Since it is necessary to have one layer that is implemented on a per Cad software basis, it is good practice to attempt to reuse as much of the implementation as possible. Especially complex tools managing taxonomies or allowing the definition of synthesis requests with literal constraints are difficult and time-consuming to port to the Api of another Cad software, as exposed graphical elements might be quite different. This would necessitate creating a custom solution that suitably utilizes and combines available graphical elements into a coherent Graphical User Interface (Gui) for each Cad software. To avoid this, the "Frontend" and "Backend" layers are introduced. These two layers do not depend on the Cad software being used, only requiring the "Add-In" layer to interface with these via some well-defined specification[1]. Due to their independence, technologies and programming languages for these layers can be freely chosen, making it easier to build complex Guis and persist data than otherwise possible with the limited functionality of Cad software.

The "Frontend" layer is best implemented as a web interface, using a common web framework, like React or Angular. This recommendation stems from the consideration that, regardless of the programming language of the Cad software's Api, it is virtually always possible to communicate with a website. In the worst case, this can be done by opening a browser window and polling its state, but oftentimes the Chromium Embedded Framework can be loaded as a Dynamic-Link Library (Dll) and used to directly display and interface with the frontend in Cad software. This is particularly convenient when interfacing with Fusion 360, as its Api provides the functionality to display and communicate with websites out of the box. This is another reason why Fusion 360 is chosen as the target for the example implementation and release of Cls-Cad. The "Frontend" layer only interfaces with the "Add-In" layer, relying on the add-in to pass the user input through to the backend. This avoids synchronization issues between the frontend, backend, and the add-in's current view of the data. Such issues could occur depending on the exact Api specification of each respective Cad software. For instance, if the user cancels an operation in the add-in and the frontend were to handle communication with the backend, it becomes difficult to revert already made changes in the backend, depending on whether or not the Api signals that the operation was canceled.

The "Backend" layer should be implemented with any Rest Api framework. The choice of programming language and framework does not particularly matter. Any widely used framework like Flask, Express.Js, FastApi Play,

[1] In practice, by means of a Representational State Transfer (Rest) Api

etc. is feasible. The same applies to the database layer; any typical database system used for full-stack applications is suitable. As with designing any full-stack system, developer familiarity and the anticipated final scale of the system should be the main factors in choosing frameworks. The backend itself handles all storage-related tasks, persisting data needed to utilize CLS. It also handles the particulars of dynamically computing contents of repositories and translating inhabitants into sets of instructions that permit the creation of assemblies. These sets of instructions are the final output of the backend. They can be visualized by the frontend as a list of BOMS, which are displayed in the add-in so that they can be compared against already existing assemblies in the CAD software. The add-in is capable of executing these assembly instructions, producing the assemblies that constitute the design space.

4.2 Components

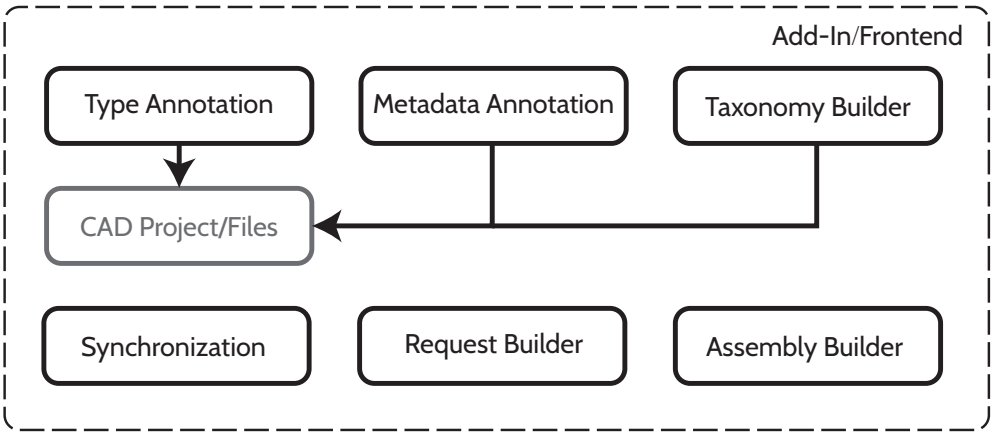
As a preface to Section 5.1, which covers the implementation of the architecture created as part of this dissertation, this section briefly touches upon the individual software components that each layer of the architecture must implement. The "Add-In" and "Frontend" layers are treated as one in this context. While it is preferable to move most of the software components from these two layers into the frontend, it is not strictly necessary. Depending on the specifics of the API of the targeted CAD software, for example, if the API offers a feature that can replace a component, it can be less effort not to do this. The goal of this section is to give an overview of the different ways that components can be implemented, detail the functionality they must provide, and the data they must be capable of processing. This is intended to aid in implementing the methodology for different CAD software.

4.2.1 Frontend and Add-In Layer

In the following, components of the frontend and add-in layers are covered, primarily touching upon GUI-related recommendations regarding their realization, defining their minimum functionality, as well as explaining CAD software-agnostic options to implement them.

Figure 4.2 depicts components that the add-in and frontend need to implement to allow the required data for synthesis to be collected. Out of these, the *type annotation* and *taxonomy builder* components are essential. If these cannot be implemented in a CAD software for technical reasons, then that

Figure 4.2: Overview of necessary components of add-in and frontend [33]



CAD software is not suitable for the proposed methodology. For the majority of CAD software, this is not a concern, for the reasons previously outlined[1].

Taxonomy Builder The taxonomy builder component provides a GUI for creating and ordering types into taxonomies. This component can be implemented in many different ways, with a simple but discouraged solution being editing and saving plain JavaScript Object Notation (JSON) files alongside the CAD files. More sophisticated implementations should provide some form of interactive and searchable nested list view, which allows displaying the often tree-like nature of the underlying taxonomy. Regarding types that are a subtype of multiple different types: as long as the underlying taxonomy graph is acyclic, they can be handled by simply "unrolling" the graph. A well-rounded but high-effort way to implement this component is by providing a fully reactive, interactive, searchable, and collapsible graph view of the actual taxonomy. This can be done by utilizing web-based visualization tools like CYTOSCAPE or D3 [21, 115]. Designing such a GUI so that it is intuitive to navigate and provides all relevant information and tools to the user without excessive panning of the view is challenging, but of high importance, as it is an essential part of reducing the friction between designers and the more theoretically grounded CLS framework.

Type Annotation The type annotation component provides a GUI for selecting reference geometry and, optionally, a reference coordinate system if the reference geometry does not provide one. It is not unusual for CAD software to

[1] Chromium Embedded Framework (CEF) and REST are ubiquitously applicable.

provide tools for defining potential connection points between parts, as this makes it easier for designers to create complex assemblies without needing to figure out suitable reference geometry "on-the-fly." The Standard for the Exchange of Product Data (STEP) file format has several suitable entity types for this purpose; for instance, the `axis2_placement_3d` entity type can be used to reference a point in three-dimensional space and define an axis from it. As such, CAD software that is capable of handling STEP files is usually capable of defining suitable reference geometry. The author of this dissertation is not aware of any CAD software/CAD kernel that does not provide any means of facilitating assemblies through mates, joints, or other forms of reference geometry. Once reference geometry has been selected, the type annotation component should provide a GUI that allows defining up to two intersection types. This GUI can be a modified version of the interface used for the taxonomy builder, restricting the user from making major changes to the taxonomy[1] but streamlined to allow for fast and efficient selection of suitable types. The two intersection types that can be selected are the *"Requires"* and the *"Provides"* types. Reference geometry being annotated has to receive at least one of these two, but may receive both. The *"Requires"* type defines what kind of part can be connected at this reference geometry, while the *"Provides"* type denotes how the part can be connected to another, as long as all of its requirements are fulfilled. Parts that only have reference geometry with *"Provides"* types are leaves of the assembly tree, parts that are complete "out-of-the-box," a typical example being screws. Parts with multiple *"Provides"* types can serve different purposes in different assemblies. Considerations regarding semantic meaning and usage, as well as how these parts get translated into combinators, are given in Section 4.4. Upon completing the selection of types for a given entity for the first time, the type annotation component assigns a persistent Unique ID (UID). It remains unchanged for the lifespan of the entity. The UID avoids confusion when making connections that have identical types.

The typing itself follows the example given in Section 3.2. The designer or engineer considers technical drawings or other specifications of the part at hand, translating these into an intersection type. The types formalize the contents of technical drawings. This process is largely abstracted away through the GUI-based input method. The designer describes parts that should connect to the selected reference geometry as a combination of traits they possess, attempting to stay as "far up the tree" as the technical specification permits to allow variance.

[1] Minor changes like renaming types and such can be allowed.

Metadata Annotation The optional metadata annotation component provides a GUI that allows adding any form of metadata, which may be interesting to aggregate during the synthesis process. The computation time for calculating abstract representations (terms) of the designs is far lower than that of assembling these in the CAD software. Many potential designs can be eliminated early based on aggregated metadata alone, for instance, if cost, weight, or availability of necessary parts is outside of permissible ranges. While such metrics can be moved into the synthesis as constraints, for numerical metrics that do not structurally impact the design and cannot be discretized well, this is not advantageous. As the design space can already be significantly cut down by means of the predicate system of CLSP, it is easier to compute these metrics for each term and filter the results. An example of this is the projected availability of off-the-shelf parts; usually, this is a rough estimate prone to being thrown off by unforeseen events, making the metric unsuitable for usage in predicates. Nevertheless, it is of interest to prioritize designs where the projected availability of required parts is high.

There are several ways this component can be implemented. CAD software typically features some form of annotation or note-taking system. If the API permits access, the add-in can write to and read from this, sanitizing manually entered data. On an even simpler level, the add-in could also read from manually entered annotations, but this makes it difficult to enforce any conventions. If such a feature does not exist as part of the API, the add-in can either save the metadata as comments within STEP files or the CAD software's native file format, or pass the information on to the backend and associate it with some unique identifier or hash computed for the part. This hash can be computed on the metadata within the header section of the STEP file format or comparable data in other formats.

Request Builder The request builder component provides a GUI that allows the user to interactively create/select an intersection type as a synthesis target. Additionally, it provides the option of specifying predicates for the synthesis. Like the type annotation component, the actual GUI can be a simplified or customized version of the taxonomy builder interface, streamlined for selecting instead of editing. A simplistic implementation of this component allows the selection of a target intersection type as well as several "counted" intersection types that form the constraints. Such an implementation accounts for nearly all structural constraints that a designer might wish to express; counted types can be intersection types that ensure all synthesized designs contain a certain number of specialized motors.

More advanced implementations can expose the entire feature set of CLSP

in a sandboxed fashion, permitting free specification of predicates. Since allowing designers to input sanitized code fragments directly goes against the core ideas of Cls-Cad, an implementation should provide a full graphical model-based editor to input these predicates and only show contextually relevant building blocks for them. The user-friendly design and implementation of such an advanced version of the model builder is difficult, as determining the look and exact features is not straightforward.

Assembly Builder The assembly builder component serves two main purposes. Firstly, it provides the option of previewing synthesized results. Secondly, it interfaces with the Api of the Cad software to execute operations that create the respective designs. There are several ways that the previewing capabilities can be implemented. A straightforward and computationally efficient method that works well with the workflow proposed in Section 4.3 is computing aggregated metadata and the Bill of Materials (Bom) for each synthesized result, displaying them in an interactive list. This can be enhanced by lazy loading, filtering capabilities, and other typical "goodies."

A more complex approach toward the implementation is submitting Stereolithography (Stl) files[1] or Step files to the backend during type annotation, and, if necessary, information about coordinate systems and connection points. This comes with an increased overhead regarding synchronization of the add-in and backend and is prone to desyncing if the Api of the Cad software does not offer some kind of callback mechanism for changes, but allows for richer previews. By using a geometry processing library to assemble and translate these files into a combined file, the previewing component can display the actual design. If results are additionally directly interpreted as Urdf in the backend, previews can also be displayed as actuated interactive models [31]. If Step files are sent to the backend, libraries such as CadQuery can be used to create an assembly out of them. While this is faster than doing the same in most Cad software due to less overhead, this is still computationally demanding, leading to prolonged waiting times for results. CadQuery does not support actuated joints. Of these two options, the Stl-file-based approach should be preferred. However, given the intended workflow, neither is necessary and may lead to designers over-complicating the decision-making process between designs, thus introducing inefficiency back into the design process.

The implementation of the second aspect is always specific to the Api of the Cad software. It depends solely on which features are available, as different pieces of Cad software have different ways of creating assemblies, managing

[1] A common file format for storing geometry as triangle meshes.

joints, i.e., allowing the setting of different joint types versus progressively restricting the DoF between geometry, and managing multiple instances of the same part. Closely examining and understanding APIs can yield large performance gains and thus an efficient implementation. Examples of this specific to FUSION 360 will be given in Section 5.1.3.

Synchronization The synchronization component's purpose is to keep data and annotations applied by the frontend in sync with the backend, as well as to keep changes made directly in the backend by management tools in sync with the frontend. Synchronization depends heavily on the implementation of all aforementioned components and the features of the API of the targeted CAD software. It is easier to implement good synchronization if the API supports callbacks for file operations like saving, renaming, moving, and so on. The implementation is also easier to create if the CAD software is not cloud-based, as then the CAD files are directly stored on the hard drive. This enables the use of file-watching capabilities of the operating system, as well as direct write access to these files without having to go through the CAD software. The synchronization of changes that occur in the backend[1] can be very challenging if targeting cloud-based CAD software[2].

4.2.2 Backend Layer

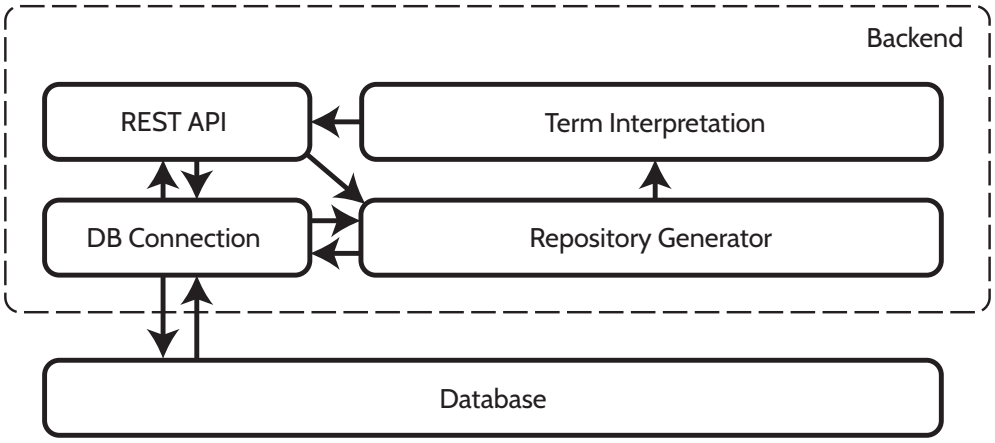
In the following, the different components of the backend layer are briefly covered, and the functionality that they need to provide is defined. Recommendations regarding technologies that can be used to implement them are given.

Figure 4.3 depicts the general structure and components that are required to implement a backend. The database and database connection components do not merit much explanation. A database with which the programmer is familiar should be selected, along with a library that provides a database connection in the chosen programming language. The CLS-CAD approach is unlikely to require handling millions upon millions of database entries. This is due to single companies or designer teams simply not having access to that many different parts. Additionally, assemblies that would necessitate that many parts and/or design alternatives quickly run into logistical issues, often being enormous projects, i.e., designing a power plant, for which automated design approaches would not be permissible from a regulatory standpoint.

[1] The proposed architecture does not intend for this to happen, but in practice, the need can always arise "in production."

[2] For example, FUSION 360 and ONSHAPE

Figure 4.3: Overview of necessary components of backend [33]



CLS-CAD is not intended to tackle engineering and design problems of this kind of complexity. Given these considerations, a NOSQL database like MONGODB should be preferred due to providing added flexibility without needing to worry about scaling, while performance is sufficient [89]. The database connection component handles all persisting and querying of stored parts and their corresponding information.

REST API The REST API component allows the backend implementation to communicate with the add-in regardless of the specific CAD software. This means that while there may be any number of different implementations of CLS-CAD, the backend does not need to be adapted. While web-based components that feature rich GUIS are considered to be part of the frontend, they can be hosted as part of the REST API as this entails starting a web server.

The REST API component's implementation fully offers Create, Read, Update, Delete (CRUD). The minimum endpoints required are one for submitting the type annotations of a part alongside corresponding metadata and potential geometry, another for receiving inhabitation requests and returning an ID, and finally an endpoint for retrieving results given a specific ID. The usage of an ID allows the inhabitation requests to be carried out asynchronously, so that multiple can be posed at the same time without the user interface of the CAD software freezing up while waiting for synthesis to complete[1].

An opinionated way to implement the backend is by enforcing hierarchical ordering of the submitted data into projects and subfolders. This mimics typical setups in CAD software. To do this, the project and its folders need to

[1] Pieces of CAD software usually do not permit threading via their APIS.

be assigned unique identifiers. For cloud-based CAD software, the API can often provide these, as they are required for the cloud functionality internally. For CAD software that runs locally, folders can be hashed, optionally together with their file metadata, to achieve consistent unique identifiers. Alternatively, hidden files that assign unique identifiers can be stored in each folder, which is least prone to causing issues when sharing datasets. This structure enables additional query parameters that allow selecting only parts from given projects and folders. Together with the database, a high degree of flexibility in selecting the parts to use for synthesis is afforded.

Repository Generator The repository generator component uses information from the database and receives inhabitation requests from the REST API. Based on the exact contents of the request, it filters the contents of the database and dynamically calculates combinators for the repository. The types of the combinators are dependent on the constraints transmitted as part of the request. How the repository is computed depends heavily on the version of CLS or CLSP that is used. For instance, the SCALA and PYTHON implementations of CLS differ by the order in which arguments to a combinator are processed.

Details of how to compute the repository with CLS and CLSP respectively can be found in Section 4.4 and Section 4.5. Choosing an implementation of CLSP is recommended[1], as the required number of combinators and resulting computational efficiency during synthesis is orders of magnitude above that achievable with a "pure" CLS approach.

Term Interpretation The term interpretation component processes results of inhabitation requests, which are enumerations of terms that satisfy requested types, and translates these terms (interprets them) into different artifact(s). The exact artifacts computed depend on the implementations of the add-in and frontend, as they need to contain all information displayed in previews.

At a bare minimum, the term interpretation must yield a set of instructions that detail how to assemble the synthesized design in CAD software.

$$\text{PLATE}(\text{SCREW}, \text{SCREW}(\text{NUT})) \tag{4.1}$$

These instructions are implicitly contained within synthesized terms due to their tree-like nature, as can be seen in Equation 4.1. It is easy to see that two screws need to be attached to the plate, and that one of the screws is then

[1] This necessitates PYTHON as the language for the backend at the time of writing.

fitted with a nut. The term interpretation component translates this into a non-nested list of instructions, "unrolling" the synthesized assembly tree into sets of instructions of the form *"Attach a screw to the first open connection point of a plate."*, or more accurately *"Attach a screw with reference geometry of given UID to any reference geometry of another given UID present on a plate."* This makes it easier for the add-in to create the design, as these assembly instructions directly translate into operations within the CAD software. Assuming multiple implementations of the add-in for different CAD software, this reduces the effort of implementing the assembly algorithm. Additionally, this can be leveraged to yield human-readable assembly instructions.

$$\text{BASE}(\text{PLATE}(\text{SCREW}, \text{SCREW}(\text{NUT})), \text{PLATE}(\text{ROD}, \text{ROD})) \tag{4.2}$$

Terms of the form given in Equation 4.2 may initially appear problematic, as once both plates are inserted into the design, screws and threaded rods that are inserted later cannot distinguish between the identical UIDs of the two different plates. This is not an issue due to the order of querying the CAD software's API being deterministic, meaning that if the left-hand plate is inserted first, inserting a screw connects it to the first element of the given UID, and inserting a rod connects it to the second element of the given UID. To improve robustness, the term interpretation can postprocess terms such that duplicate parts receive unique names, for instance, by means of an incrementing counter. The unique names can then be annotated to each part at insertion time in the CAD software.

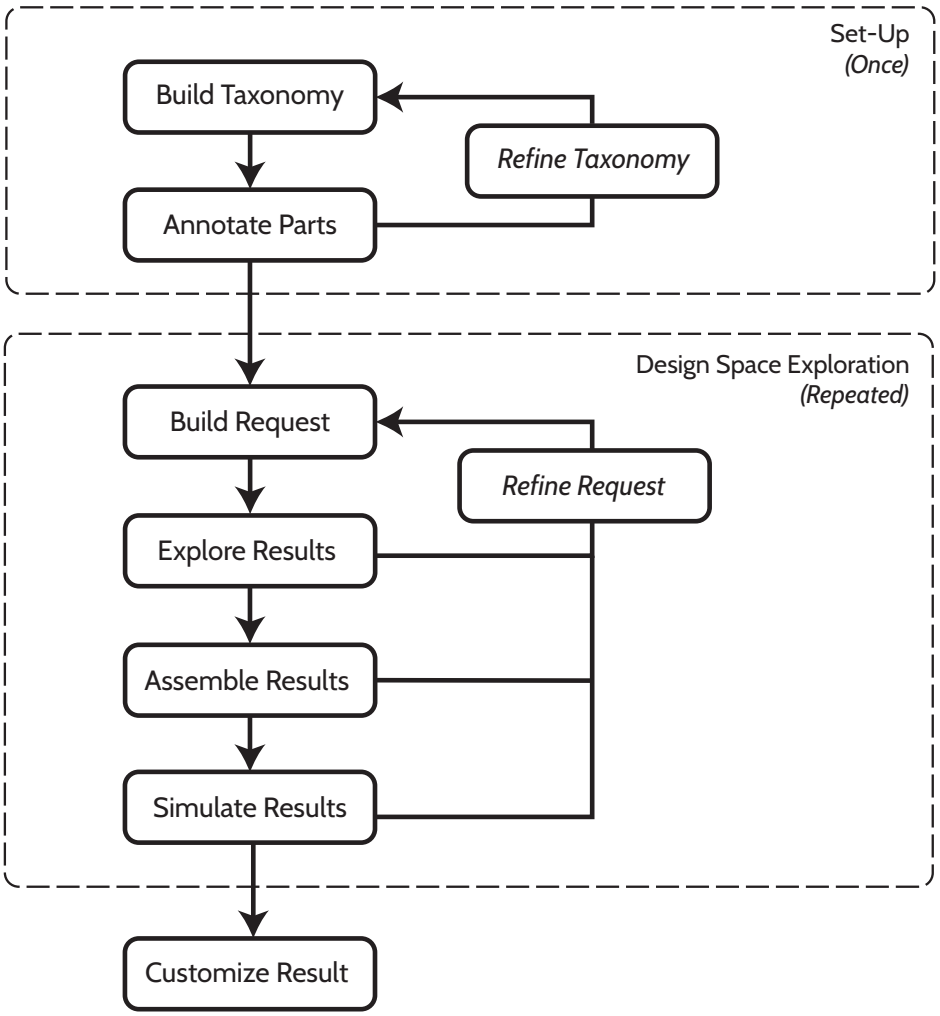
4.3 Workflow

Within this section, the workflow enabled by an implementation of CLS-CAD for a given CAD software is discussed. A high-level overview of the general workflow is provided, followed by a detailed explanation of the individual steps, outlining benefits and potential pitfalls.

As can be seen in Figure 4.4, the workflow proposed as part of the methodology is comprised of two main phases: the *set-up phase* and the *design space exploration phase*. The set-up phase can be considered the cost of avoiding the tedium described in Section 2.1.1 and is a prerequisite to reaping the efficiency gains enabled by CLS-CAD. While the set-up phase itself is also tedious, its advantage is that it only needs to be done once per part. Even if new parts are introduced into the system, this holds true[1]. The quality of the taxonomy

[1] An exception to this is if new parts expose major flaws within the taxonomy; some parts then need to be retyped.

Figure 4.4: Overview of intended workflow for creating final assemblies [34]



and the thoroughness of its application to the individual parts during annotation are vital to the following design space exploration phase. Since CLS-CAD is based on a deterministic formal method, Combinatory Logic Synthesis, synthesized results are guaranteed to fulfill the given specification. However, this is only with respect to the generated repository. If the specification of the connectivity properties of available parts is erroneous, the resulting designs may also contain errors. Approaches that lead to specifications which work well in practice are given in Section 5.1; however, assisting with this is not (yet) a focus of CLS-CAD. In an industrial setting, existing approaches for quality control, already being carried out, could be adapted to ensure the quality of

the set-up phase.

The design space exploration phase details the workflow enabled by the ability to automatically synthesize designs that meet given specifications. The steps that make up the phase each filter and narrow down the design space, ordered from coarsest to finest. The time needed for evaluating a single design in each step of the phase increases further along the workflow; however, due to there being fewer designs needing examination in each step, the overall time spent in each phase is approximately linear.

In the following paragraphs, the individual steps of each phase are explained in detail, touching upon common pitfalls observed as part of Chapter 6 and giving examples of good practices.

Set-up Phase The workflow begins by building a suitable taxonomy for typing all available parts, e.g., parts previously designed by the design team and potentially utilized off-the-shelf parts. To achieve this, the taxonomy is initialized with several root nodes. These should each be representative of individual concerns or aspects of the parts to be annotated. For instance, there could be a root node that contains all types that describe geometrical compatibility, another root node that describes electrical compatibility, a root node describing the functions of the part, and so on. Determining these root nodes is analogous to the initial rough draft phase of product or software development. As such, many of these root nodes will have already been planned and considered, in which case those considerations can be directly translated into root nodes. From there on out, designers and engineers consider criteria for compatibility between parts regarding each aspect embodied by the root nodes. Each criterion is assigned a telling identifier, for instance, `smaller_than_3mm`, `12_volt`, `din912`, and so on. These correspond to well-known concepts or norms with which designers and engineers are familiar. Criteria are then coherently grouped into categories; these are assigned telling identifiers and become a supertype of their respective group. For the previous examples, these might be `tolerance`, `voltage`, `iso_norm`. Types can be assigned to multiple categories as appropriate. Categories can be picked as coarse or fine as needed during this process, depending on how intricate the designs to be synthesized will be. It is advisable to err on the side of finer modeling, as there is no disadvantage in doing so. The identified categories are then examined again; potential groupings become new categories, and with this, supertypes of the existing categories. This is repeated until no more categories and types are identified. Concerns that arose during previous projects can be introduced as types into the taxonomy ahead of time; there is no downside to adding additional types.

This process is a bottom-up process, but it could be carried out top-down. For the use cases detailed in Chapter 6, the bottom-up approach is preferable[1]. Intuitively, it seems advisable to tailor the taxonomy to design spaces and existing parts instead of retrofitting it, which may inadvertently restrict the design space. With regard to typing, as detailed in Section 3.2, the world of engineering revolves around categorization. This makes it possible for designers and engineers to translate their understanding of compatibility and requirements into a taxonomy, while CLS-CAD provides them with an interface that abstracts away type-theoretic and formal aspects.

After a taxonomy is derived from the existing parts, the next step in the workflow is to annotate the parts so they become suitable for generating repositories needed to carry out combinatory logic synthesis. First, a part to be annotated is examined, and the interfaces it provides for creating connections are identified. These are the same interfaces from which the taxonomy is derived. For these interfaces, suitable reference geometry is either selected or added, depending on the CAD software and already present geometry. As a rule of thumb, for planar connections, selecting the center point of an interface's geometry is a good option. For example, for a screw whose head will be planar to a surface that provides a screw hole, the center point of the bottom surface of the screw's head and the center point of the top edge of the hole should be selected. For connections that are multi-planar, such as a mounting bracket that slides onto the horn and idler of a motor, virtual reference geometry should be created. The geometric midpoint of the individual center points of the geometry of these planar surfaces should be created and selected as reference geometry. For the common case of a connection being mounted to exactly two planar surfaces, this can often be done by intersecting the line between the center points of these surfaces with a mid-plane.

Once the reference geometry is selected, the GUI for selecting types from the taxonomy is used to interactively build up to two intersection types for that connection. One of these types describes the requirements for connecting other geometry, while the other describes the properties that the selected reference geometry itself has when connecting to other geometry. These are the *Requires* and *Provides* types previously covered in Section 4.2. If the taxonomy used was modeled granularly, the structural variance of the results can be reduced by picking intersection types composed of many identifiers and attributes of the mating interface, increasing its specificity. Accordingly, variance can be increased by selecting a particularly succinct intersection type. In both cases, the intersection type must remain technologically accurate. This

[1] For other smaller use cases not detailed in this dissertation, the bottom-up approach also worked better.

is why the recommendation to err on the side of more granular taxonomies is given, as this allows such adjustments to be made during annotation. If the implementation of the add-in and backend permits it, parts can also be given multiple different sets of annotations, with some being more permissive and some less.

Ideally, once the initial taxonomy is created, during the annotation of parts, all connection points can be assigned intersection types to the designers' or engineers' satisfaction. However, in practice, there are oversights, and parts with new connections may be designed and added to existing ones. Annotating the parts creates a feedback loop between the taxonomy and the annotation process. When a connection point is encountered that cannot be assigned a type that encodes all relevant design information present in the corresponding technical drawings or that the designer is aware of, the taxonomy is augmented with new types and appropriate supertypes, as described regarding the initial creation of the taxonomy. After large changes to the taxonomy, the set of parts may need to be processed again, and the connection point's annotations improved. As good engineering practices encourage reusability and compatibility with prior standards, this should rarely be the case.

Design Space Exploration The design space exploration phase is an interactive and iterative process. Designers begin by requesting a type that produces designs of interest, for instance, the base of some robotic system. Without any further restrictions and specifications, this leads to a large number of designs, spanning the entire design space and potentially being somewhat heterogeneous. Using the previewing tools provided by the implementation of CLS-CAD, these can be sampled. The designer takes note of different desirable or undesirable aspects in the synthesized designs based on the preview. Examples could be the usage of particularly cost-effective parts, revealed by the synthesis being compatible with the design goal, or, on the flip side, the usage of expensive parts that make no sense for a product intended for a budget-friendly market segment. This allows the designer to quickly get an overview of the design space, gaining an approximate sense of what trade-offs and structures are possible. Based on this, the request is refined, setting constraints that ensure the presence and absence of parts, thus reducing structural variance. This initial rough exploration of results and consequent refinement is repeated several times until the majority of previews examined are of interest.

This marks the beginning of the next step, the "Assemble Results Phase." After arriving at a more specific synthesis request that narrows down the re-

sults, little information can still be gleaned from the computationally efficient previews. At this point, one of the major features of CLS-CAD comes into play: the automated creation of assemblies in CAD software. The designer continues to sample results and take notes of likes and dislikes; however, for this step, chosen samples are assembled in the CAD software. This increases fidelity: designs can be seen in three dimensions, rendered in different environments, and, in general, a far more accurate overview is obtained. Based on this, designers identify additional constraints, further refining the synthesis request.

Once the results are narrowed down sufficiently, i.e., there are only tens of results available instead of thousands, CLS-CAD is utilized to assemble all remaining designs and matching artifacts that aid in evaluation. URDF files describing the designs, corresponding ROS packages which simulate the designs in different environments to evaluate motion planning performance and suitability for given use cases, or even completely interactive simulations powered by game engines such as UNITY can be automatically generated and provided to the designer, allowing for an in-depth impression of the designs [31, 53]. Based on the rich simulation and analysis tools that are included in CAD software, designers can evaluate mechanical properties like stresses or deformation, kinematic behavior, obtain technical drawings, moments of inertia, and so on. Using the additional artifacts, all sorts of different properties can be evaluated. As a CAD assembly of a product is the de facto industrial standard, downstream support and possibilities are quintessentially endless. CLS-CAD automatically creates assemblies that are well-organized and ready for further modification.

Utilizing all this additional information, the designer can choose suitable design solutions generated by the methodology. From there, since the results are full-fledged CAD assemblies, they can be customized to meet specific branding requirements or to improve upon any flaws identified during the simulation phase. Generating CAD assemblies affords the technological means to do so. This allows efficiency gains without losing flexibility, as the results enable any step that would have been previously carried out manually to still be done.

4.4 Synthesis without Predicates

In the following, the rules by which the typed combinators in the repository are computed are detailed. The enhancements afforded by CLSP are not used. The intuition behind the type signatures is given, and shortcomings later tackled by CLSP are pointed out.

$$\begin{aligned} C_p : & \; req(r_1) \rightarrow req(r_2) \rightarrow ... \rightarrow req(r_n) \rightarrow prov(p) \\ & \text{with } G, \text{ the set of all annotated geometry,} \\ & P = \{p \mid p \in G \wedge prov(p) \neq \emptyset\}, \\ & R = \{r \mid r \in G \wedge req(r) \neq \emptyset\}, \\ & p \in P, \\ & \text{and } \{r_1, r_2, ..., r_n\} \subseteq R \setminus \{p\} \end{aligned} \tag{4.3}$$

Equation 4.3 defines how a part that possesses annotated geometry is translated into typed combinators for synthesis. Given any part with annotations created by CLS-CAD, the set of all geometry that has received annotations is denoted by G. The sets P and R can be understood as the subsets of G wherein the contained geometries possess an assigned "provides" or "requires" type, respectively. This is resolved by the functions *prov* and *req*, which return the provided or required intersection type of the given geometry if one has been assigned, or else $\emptyset$. Equation 4.3 then states that for any given part, one typed combinator is created per "provides" type present. A total amount of $|P|$ typed combinators is created per part. The combinator's type is a nested arrow type, where all geometry except the one being considered as "providing" is translated to its respective required types.

In practice, $\{r_1, r_2, ..., r_n\} \subseteq R \setminus \{p\}$ is often $\{r_1, r_2, ..., r_n\} = R \setminus \{p\}$. This depends on the implementation details of CLS-CAD. If marking connections as optional is not permissible in the implementation, then the latter holds. The arrow type can be understood to, colloquially speaking, mean: "If a part that provides the type $req(r_1)$ can be found, this combinator can now be considered as $req(r_2) \rightarrow ... \rightarrow req(r_n) \rightarrow prov(p)$." As long as there exist parts that can provide all required types, the combinator eventually reduces to just providing the intersection type $prov(p)$, albeit now with all of the parts fulfilling the specifications of the $req(r)$'s now connected. These, in turn, also have types generated as stated in Equation 4.3, meaning that at each connection point of C_p an entire sub-assembly may be connected, which in turn may have its own connecting sub-assemblies, and so on. The interplay between how intersection types and the taxonomy are set and created, combined with the repository generation, allows combinatory logic synthesis

to tackle complicated assemblies, as long as they can be expressed as open kinematic chains.

There are two shortcomings in this repository structure. The first of these is inherent to Cls. Since every term is a tree-like structure, directly encoding closed kinematic chains is not possible. While for specific use cases, domain-specific quirks can be used to build modeling approaches able to circumvent these limitations through suitable post-processing, since Cls-Cad targets generic Cad assemblies, with no prior knowledge of how many parts are available or of their specific functions, the methodology cannot make use of such an approach. This problem is the focus of ongoing research. The second shortcoming has to do with how Cls computes inhabitants. While the description of how types are derived during synthesis holds, additionally, synthesis also considers all options for splitting the arrow type. This means that in addition to the colloquial expression previously given, the expressions "If a part that provides the type $req(r_1) \rightarrow req(r_2)$ can be found, then...", "If a part that provides the type $req(r_2) \rightarrow req(r_3)$ can be found, then...", and so on, also apply. During synthesis, no expressions that satisfy these are found, due to the way the repository is constructed; however, the implementation of Cls checks for the presence of these regardless, negatively affecting runtime.

Constraints that allow iteratively reducing and focusing the design space play an important part in the workflow for using Cls-Cad, detailed in Section 4.3. Without these, depending on the use case, finding a suitable design within the design space is akin to finding a needle in a haystack.

$$\begin{aligned} C_{i_1,i_2,i_3} : \; & req(r_1)(i_1(\omega)) \rightarrow req(r_2)(i_2(\omega)) \rightarrow prov(p)(i_3(\omega)) \\ & \text{such that } 0, ..., n \text{ are unary type constructors,} \\ & i_1, i_2, i_3 \in \{0, ..., n\}, \\ & \text{and } i_3 = i_1 + i_2 + k \end{aligned} \tag{4.4}$$

Equation 4.4 describes how the repository generation can be augmented to enable integer constraints when using Cls instead of ClsP. Given some performance indicator that the designer wants to fix to a specific value, the type of each individual connecting sub-assembly is further parameterized with a unary type constructor representing the value of the performance indicator computed for it. The resulting provided type of the combinator is then parameterized with a unary type constructor which is precisely the sum of the metrics of the sub-assemblies plus a constant. The constant k itself depends on the properties of the part; for instance, if counting the number of occurrences of motors, if the part is a motor, the constant would be one, else zero. By recursion, this allows the metric to be counted and aggregated bottom-up during synthesis. This allows requesting only designs that have a specific value for a metric, and by extension, also lesser-equal values of the

metric by concatenating multiple synthesis requests. While this approach enables restricting the design space based on any discrete or quantized metrics, the scalability to larger use cases and multiple simultaneously enforced constraints is very poor. This is due to this method adding an additional combinator for every possible combination of values of the metric for every part. This results in an exponential amount of combinators. Adding an exponential amount of combinators to the repository heavily affects the performance of the synthesis. Even synthesis of solutions for small repositories of fewer than 20 parts is usually out of reach for powerful desktop computers once three or more constraints are added with this approach. Additionally, this way of modeling the types is unintuitive. The large amount of additional combinators with obscure numerical values makes this approach difficult to debug. If there is an issue with the type annotations of a part or the implementation of the repository generation, thousands upon thousands of combinators need to be examined.

4.5 Synthesis with Predicates

The development of CLSP is partially motivated by the previously detailed issues. It tackles the poor scalability of the previous approach, leading to types that are easier to understand and match the actual part more intuitively. With this approach, there is exactly one combinator per part. This allows for larger use cases, the usage of several constraints at once, as well as easier debugging, due to the size of repositories being linear in the amount of annotated parts available for synthesis.

$$\begin{aligned} C: \quad & \langle \alpha_1 : \text{int} \rangle \Rightarrow \langle \alpha_2 : \text{int} \rangle \Rightarrow \langle \alpha_3 : \text{int} \rangle \Rightarrow (\alpha_3 = \alpha_1 + \alpha_2 + k) \Rightarrow \\ & \langle\!\langle x_1 : \mathit{req}(r_1)(\alpha_1) \rangle\!\rangle \Rightarrow \langle\!\langle x_2 : \mathit{req}(r_2)(\alpha_2) \rangle\!\rangle \Rightarrow \mathit{prov}(p)(\alpha_3) \\ & \text{with } \Delta \supseteq \{0 : \text{int}, \ldots, n : \text{int}\} \end{aligned} \tag{4.5}$$

The types of the combinators present in the repository still need to be dynamically computed. The way these types are computed is given by Equation 4.5. A number of literal variables are introduced equal to the arity of the original type as given in Equation 4.3, these being given as α_1 to $\alpha_{\mathit{arity}(C)}$. These literal values are assigned by a predicate, which contains some rule of computation for the performance indicator of interest. In the case of counting the number of occurrences of a specific type of import (i.e., motors), or the total aggregated value of some performance indicator (i.e., metadata like weight), the predicate assigns the sum of literal values representing the subassemblies plus a constant, which represents the current part's value of the

performance indicator, to the literal value representing the current assembly. These predicates are evaluated during synthesis as needed, modifying types and thus the sub-assemblies permitted by $req(r)$.

The only major restriction still in place is that constraints can only operate on discrete or quantized metrics. Reasons why this approach is faster, and not just a way of wrapping the prior approach in a nicer syntax, are given in the paper by Dudenhefner et al. [44]. As such, they are not covered here.

CHAPTER 5

Design and Implementation

This chapter moves away from the methodology and guidelines for creating an implementation, covering the specifics of the implementation of Cls-Cad for Fusion 360, documenting its structure, features, and interface. Lessons learned that do not affect the methodology directly but are crucial to achieving good performance are also discussed.

Chapter Organization The remainder of this chapter begins with a short discussion of the choice of Fusion 360 as the target Cad software for the implementation of Cls-Cad. This is followed by an overview of the software components that the methodology translates to. Then, a brief overview of the Fusion 360 Api and its features relevant to Cls-Cad is given, along with the developed user interfaces and a discussion thereof. Next, an in-depth exploration of features and quirks of Cad kernels is provided, highlighting techniques to significantly improve performance. Finally, implementation details of the backend are covered regarding the generation of repositories for combinatory logic synthesis and post-processing inhabitants into assembly instructions. The focus is on which features of ClsP can be utilized to improve performance and robustness.

5.1 Fusion 360 Add-In and Web-Interfaces

Implementing the add-in portion of the developed methodology is, by and large, the most time-intensive aspect of creating an implementation. This is due to the Apis of Cad software being restrictive. Manufacturers of Cad software want to avoid add-ins crashing the software or compromising or corrupting Cad files. As such, most Apis enforce strict rules and prohibit any kind of "programming gymnastics." This makes the Apis safe; however, it also hampers efficient code reuse and straightforward solutions, often resulting in cumbersome code. Additionally, the internal data models of Cad software are oftentimes unintuitive. It is not productive to guess the usage of functions; instead, documentation for the Api must constantly be checked. Oftentimes, experiments are necessary to determine how Api calls interact with the Cad software and which results they produce. Several such quirks are touched upon within this section, highlighting the importance of optimizing code for the Api in question.

Due to the time-intensive nature of creating an implementation, the add-in layer is implemented for a single Cad software, Fusion 360. The goal is to trade off targeting more than one Cad software for high performance and practical usability. The reasons for picking Fusion 360 are the following:

- Fusion 360 is freely available for everyone. Users affiliated with academic institutions can freely obtain a full version with the entire feature set. Unlike other Cad software, where educational versions cost upward of 100 Euros, e.g., SolidWorks, this makes results easier to reproduce, without cost becoming a factor.
- The Api allows using Python for programming. This keeps the codebase of the implementation homogeneous regarding the programming language used, excluding web interfaces. Additionally, the Api allows directly loading and executing code without the need to compile it to Dlls[1]. There exists open-source Cad software that supports using Python for programming, but regarding feature set, ease of use, and industry usage, it cannot compete with commercial software like Fusion 360. Thus, Fusion 360 is deemed more relevant to real applications.
- The native file format of Fusion 360, .f3d, provides inherent support for annotations in the form of attributes. Any entity of the Cad design can be assigned an attribute that holds up to two megabytes of data.

[1] Most Cad software developed by Dassault Systèmes requires compiling add-ins to Dlls or executables.

These attributes persist through exporting the files to a hard drive, but also integrate with the cloud-based approach of Fusion 360, making sharing datasets easy. This avoids the need to synchronize with Step files.

- The cloud functionality of Fusion 360, in combination with the support for annotations, simplifies collaborative work.
- Other software by Autodesk is shown to be suitable for knowledge-based engineering[1] [58].

An interesting contender is OnShape by PTC. It is an in-browser Cad software, and the Api is designed around interacting by means of web technologies, making it programming language agnostic. At the time of beginning work on the implementation of Cls-Cad for Fusion 360, OnShape did not yet see much widespread use, but it seems to have gained in popularity. A slight detractor from OnShape is that since the Api is purely a Rest Api, if a required feature does not exist, there is no way to work around this locally, instead necessitating extensive synchronization with the Cls-Cad backend to spoof missing features.

In the following, first, an overview of the overall structure of the implemented add-in, as well as a short introduction to the Fusion 360 Api, is given. Following that, the implemented Guis and their variants are shown, and their functionality and usage are explained. Finally, examples of Api-specific quirks that can be leveraged to significantly increase the implementation's performance are discussed.

5.1.1 Add-In Structure and Implementation Details

The implementation of the add-in for Fusion 360 is an opinionated take on the methodology and structure described in the previous sections. As such, it diverges from the proposed methodology in some aspects, being tailored toward modular robotics. One notable difference is present in the handling of the taxonomy, or, in this case, taxonomies. Instead of allowing users complete freedom over how to design the taxonomy, a split into three overarching categories for types is enforced by the implementation of the add-in: `Parts`, `Formats`, and `Attributes`. Internally, with regard to the synthesis, these are merged into a single taxonomy; but for all other intents and purposes, the add-in treats these as functionally independent taxonomies. Specifically, it

[1] The methodology of Cls-Cad can be classified as such, with the types and the repository being the knowledge.

is not possible to create a new type that has a supertype outside of its own overarching category. This way of modeling the taxonomy is efficient and easy to use and understand in practice. Connections can be typed by intersecting types from the `Parts` and `Formats` taxonomies, with the former describing what kind of structure or function the connecting component should have, and the latter describing what kind of physical connection between parts is possible. This can (optionally) be augmented by types from the `Attributes` taxonomy, which contains properties not related to the function or geometry of parts. This enables a checklist-like approach to assigning types; for instance, if no type from the `Formats` taxonomy is used, the designer scrutinizes the annotated connection point, as the concrete geometry not mattering is impossible.

Splitting the taxonomy into three independent ones is leveraged by the add-in to reduce the work needed to annotate connection points. The `Parts` taxonomy is independent of the actual connection point, with the function and structure of a part being inherent to its design. As such, instead of having to re-select intersections of many constituent types through the interface each time a connection point receives a "Provides" type, the CAD file itself is assigned an intersection type from the `Parts` and `Attributes` taxonomies. Setting this type is mandatory, with the add-in operating under the assumption that annotated parts have a purpose. Additionally, if assigning a type to a part is difficult, CLS-CAD indirectly incentivizes reevaluating the necessity or efficacy of such parts. Accordingly, "Provides" types can only utilize types from the `Formats` and `Attributes` taxonomies. When generating combinators from type information annotated to a part, "Provides" types are additionally intersected with the inherent type set on the CAD file. This reduces modeling errors and expresses that, axiomatically, the part itself does not change no matter how it is oriented.

Another change the implementation makes, which deviates from what is described in Section 4.5, pertains to handling multiple identical sub-assemblies. It is a common occurrence in robotics that the same sub-assembly occurs at multiple locations throughout a design; for instance, when designing a hexapod, all legs are identical. Such a constraint could be enforced by utilizing the features afforded by CLSP; however, the implementation of the add-in takes a different approach. Connection points intended to receive identical sub-assemblies can be selected and given a shared type, which turns them into a single virtual connection point. As far as CLS-CAD and the synthesis are concerned, only a single connection point exists. The add-in and CLS-CAD keep track of the multiplicities of individual components that occur in the

sub-assemblies internally[1] and resolve all of these during the post-processing of inhabitants. This leads to simpler combinators and improved run-time over using equivalent predicates but complicates the implementation of the post-processing step.

Table 5.1: Overview of commands and their purpose

Command	Usage
`assembleresult`	Displays an overview of synthesis requests, designs they resulted in, and allows assembling them individually or in batches.
`(var)taxonomyediting`	Allows graphical editing of taxonomy given by `var`. The variable can be `Parts`, `Formats`, or `Attributes`.
`checkandsubmit`	Checks if a part has annotations and at least one "Provides" type, then sends it to the backend.
`downloadtaxonomy`	Allows saving a local copy of the taxonomies as a JSON.
`exportproject`	Creates a local ZIP file, containing all CAD files and the taxonomies, for sharing data.
`jointtyping`	Allows selecting `JointOrigins` and assigning them "Requires" and "Provides" types.
`partmanagement`	Allows adding additional metadata to the parts.
`parttyping`	Allows assigning types from the "Parts" and "Attributes" taxonomies to the part.

Continued on next page

[1] As sub-assemblies may also contain such virtual connection points.

Table 5.1: Overview of commands and their purpose (Continued)

`requestsynthesis`	Allows selecting target type for synthesis as well as setting integer constraints for contained types.
`togglecustomgraphics`	Toggles the display of annotated types in Fusion 360.
`typecrawlingproject`	Synchronizes the current project state in Fusion 360 with the backend.
`uploadtaxonomy`	Allows selecting a local taxonomy to override the one saved for a project in the backend.
`uuidupdatecrawlingproject`	Fixes files that contain nested parts to be single-level as well as migrate annotations to newer versions of Cls-Cad.

Table 5.1 gives an overview of the set of commands that the add-in registers in Fusion 360 to enable the Cls-Cad methodology to be used. Due to the additional rules that the implementation enforces, the `checkandsubmit` command is added, which allows catching mal-typed parts and prevents them from being submitted to the backend. There are also several commands that add convenience features implemented by the add-in, these being the `(download/upload)taxonomy`, `exportproject`, and `uuidupdatecrawlingproject` commands. The former two enable sharing of persistent artifacts that allow results to be reproduced without making use of any of the cloud features of Fusion 360. The latter enables migrations between different versions of Fusion 360 and the Cls-Cad add-in, as well as automatically fixing a common modeling issue specific to Fusion 360. This common modeling issue is utilizing nested components. At the time of writing, attributes are lost on nested components of a part when they are inserted into assemblies.

The Fusion 360 Api does not enforce any particular structure of the code for newly added commands. However, the implementation of the Cls-Cad add-in follows the structure present in the template for new add-ins in Fusion 360. This template suggests a pseudo object-oriented implementation style, where each new command is implemented in a separate package. Within this package, graphical assets required by the command are stored, such as button icons, as well as an `entry.py`.

Listing 5.1: Minmal structure of added commands

```
browser_input = adsk.core.BrowserCommandInput.cast(None)

def start():
    """
    Creates a promoted command in the CLS-CAD tab.

    :return:
    """
    cmd_def = ui.commandDefinitions.addButtonDefinition(
        CMD_ID, CMD_NAME, CMD_DESCRIPTION, ICON_FOLDER
    )
    futil.add_handler(cmd_def.commandCreated, command_created)

    workspace          = ui.workspaces.itemById(WORKSPACE_ID)
    panel              = workspace.toolbarPanels.itemById(PANEL_ID)
    control            = panel.controls.addCommand(cmd_def, COMMAND_BESIDE_ID, False)
    control.isPromoted = IS_PROMOTED

def stop():
    """
    Removes this command from the CLS-CAD tab along with all others it shares a panel with.

    :return:
    """
    workspace = ui.workspaces.itemById(WORKSPACE_ID)
    panel     = workspace.toolbarPanels.itemById(PANEL_ID)

    for i in range(panel.controls.count):
        if panel.controls.item(0):
            panel.controls.item(0).deleteMe()
```

An example of the bare minimum contents of such an `entry.py` can be seen in Listing 5.1. This file is used like a class, with global variables being used instead of class attributes, for instance, `browser_input`. By convention, each of these `entry.py` files defines a `start` and `stop` method. The `start` method creates the custom command and adds it to the Fusion 360 user interface at a specified location. A custom command is created by first creating a `ButtonDefinition` and then assigning it to either an existing or add-in-created panel. The Cls-Cad add-in elects to create a dedicated tab to house all new commands, as well as several different panels within it to logically order commands. Figure 5.2 shows the Cls-Cad tab next to the "Manage" tab, with panels being "Typing Tools", "Management Tools", and so on. The `ButtonDefinition` can either be `Promoted`, displaying it as a large graphical button, or if not `Promoted`, hidden away in the sub-menu of the panel. Additionally, a custom command needs to at least register a `command_created` handler. This handler is called whenever the corresponding button is clicked by the user and contains the necessary code to then create and display an interface. The

handler registers additional handlers to interact with the selections the user makes within the CAD software.

An example of a `command_created` handler that contains the minimal functionality which most commands following the design paradigms laid out in Section 4.2 require is given in Listing 5.2. Upon the user clicking the custom command, several additional handlers are allocated. The `command_execute` and `command_destroy` handlers are central to any custom command and represent the code to be run when the user selects the "Ok" or "Cancel" button. The handler `command_execute` parses and processes all information gathered during the lifetime of the custom command window and persists changes by synchronizing with the backend, writing changes into the CAD file, or starting upload or download operations within FUSION 360. The handler `command_destroy` cleans up registered handlers to avoid negatively impacting the performance of FUSION 360[1] and resets global variables used for the command. For some variables, it is desirable to keep their values between executions of the same command, but in general, FUSION 360 operates on the principle that executions of a command are independent.

The remaining handlers allow interaction with actions that the user makes in FUSION 360. When the handled action occurs within FUSION 360, information about the event is forwarded to the handler, thus allowing the custom command to process it. This is preferable to polling the entire design's state at regular intervals to check for changes. However, polling is possible using the FUSION 360 API. Performance is generally sufficient to handle events for which the API does not provide a handler. In general, while undocumented, the FUSION 360 API allows access to all features or operations that the user could carry out by clicking menu items. By using the API to read out the user interface's state, the internal numeric IDs of undocumented elements can be obtained, and mouse clicks to them can be spoofed. The handlers shown in Listing 5.2 are representative of the methodology. The `command_select` and `command_unselect` handlers fire whenever the user selects any entity in FUSION 360. This allows the add-in to obtain target geometry for annotation.

To allow selecting geometry or displaying graphical elements in a custom command's user interface, inputs to the command must be defined. There is a range of inputs to choose from; typical elements of GUI frameworks from other languages are available [7]. However, one input is of particular note: the `BrowserCommandInput`. This input provides a version of the Chromium Embedded Framework integrated into FUSION 360. This integration eliminates the need to set up a custom solution for communicating with the web frontend

[1] Some handlers may affect performance heavily, i.e., those that fire on mouse movements and perform heavier computation.

Listing 5.2: Example of handlers and CEF

```
def command_created(args: adsk.core.CommandCreatedEventArgs):
    """
    Called when the user clicks the command in CLS-CAD tab. Registers all important
    handlers for the command.

    :param args: adsk.core.CommandCreatedEventArgs:
    :return:
    """
    global placeholder_input_a, placeholder_input_b, browser_input

    futil.add_handler(
        args.command.execute, command_execute, local_handlers=local_handlers
    )
    futil.add_handler(
        args.command.destroy, command_destroy, local_handlers=local_handlers
    )
    futil.add_handler(
        args.command.incomingFromHTML, palette_incoming, local_handlers=local_handlers
    )
    futil.add_handler(
        args.command.select, command_select, local_handlers=local_handlers
    )
    futil.add_handler(
        args.command.unselect, command_unselect, local_handlers=local_handlers
    )

    inputs          = args.command.commandInputs
    browser_input   = inputs.addBrowserCommandInput(
        id=BROWSER_ID,
        name="CEF",
        htmlFileURL=PALETTE_URL,
    )
    selection_input = selection_tab_inputs.addSelectionInput(
        "Internal Name", "Text to Display", "Description on Hover"
    )
    selection_input.setSelectionLimits(0)
    selection_input.addSelectionFilter("JointOrigins")
```

layer, saving work and relying on a robust implementation by AUTODESK. The `palette_incoming` handler allows the add-in to parse data that the frontend, loaded into such a browser, elects to send, while the `command_select` handler can forward information about selections to the frontend. This allows the frontend to remain synchronized with the add-in and FUSION 360.

Finally, Listing 5.2 shows that inputs can filter their permissible allocated values. In this case, the selection input only accepts entities of a specific type, `JointOrigin`. The implementation of the CLS-CAD add-in for FUSION 360 exclusively permits `JointOrigins` as reference geometry for connection points. This is done because JointOrigins exactly fit the requirements laid out in Chapter 4. As seen in Figure 5.1, a JointOrigin in FUSION 360 is a coordinate system with a given origin point, displayed by a multi-colored triad, where

Figure 5.1: Example of a JointOrigin

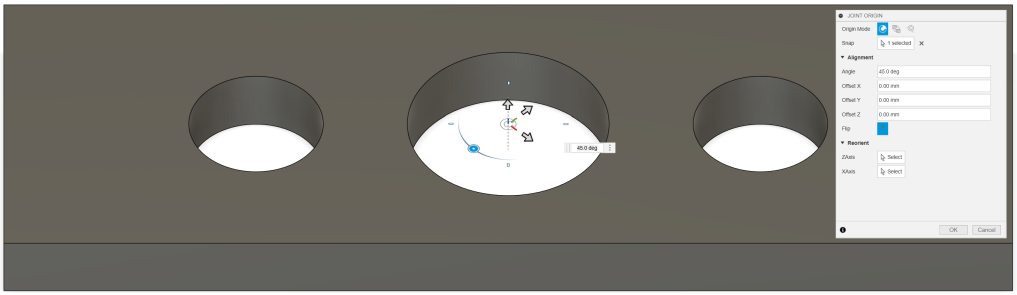
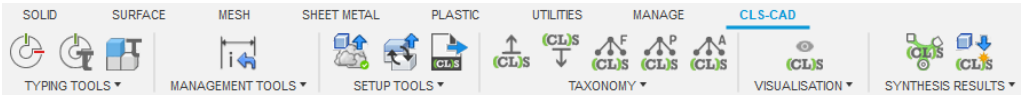


Figure 5.2: Overview of ribbon integrated in Fusion 360 interface



the x-axis is red, the y-axis is green, and the z-axis is blue. Furthermore, the tools available in Fusion 360 for defining JointOrigins are extensive and easy to use, able to attach origins to snap points (centers of gravity, midpoints, quarters), define them between two faces or at edge intersections, rotate and offset them, and reorient them toward any plane or point. Additionally, their explicit purpose is to mark connection points for creating assemblies, making them a natural fit. While it would have been possible to permit the selection of other point type entities and then select reference edges to define coordinate systems, this standardization helps with the visual recognition of connection points. This way, if users of Cls-Cad see JointOrigins that display custom text, they are immediately recognizable as entities that bear type annotations.

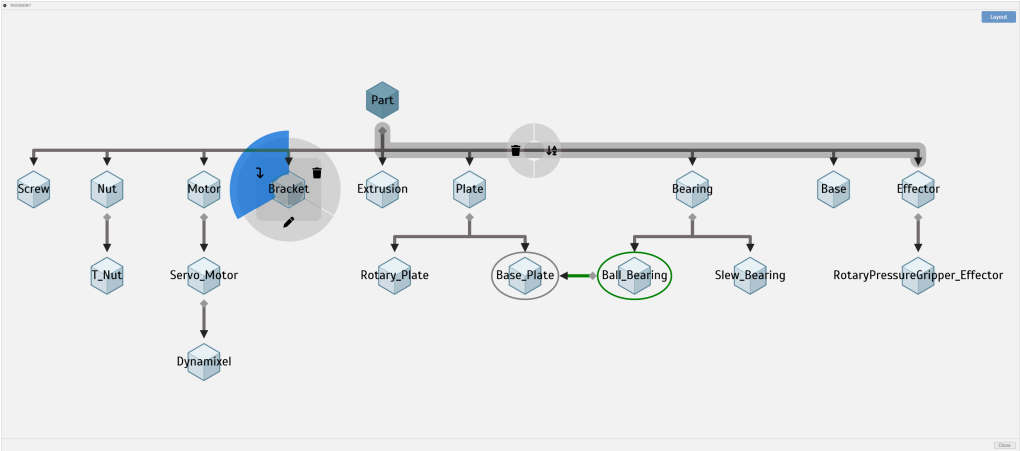
5.1.2 User Interface

In the following, a brief overview of the Guis implemented for integration into Fusion 360 is given. The images are taken "as-is" in Fusion 360. In some cases, several images are overlaid to concisely show different Gui states; this is mentioned where applicable.

Figure 5.2 shows the tab that the Cls-Cad add-in registers inside the 3d modeling workspace of Fusion 360. The tab contains six different panels. The first panel allows placing JointOrigins[1], selecting JointOrigins and

[1] This functionality assigns the pre-existing tool for this in Fusion 360 to the panel and acts as a shortcut.

Figure 5.3: Taxonomy editor as displayed in Fusion 360



annotating them with a type, as well as setting the inherent type for parts[1]. The second panel houses tools that allow writing additional metadata into the CAD file, used to set the cost and availability of parts. The third panel contains tools that interact with the CAD files and backend, allowing for sending individual annotated parts to the backend for storage, crawling the entire project and uploading crawled parts to the backend[2], exporting the project as a ZIP file containing all CAD files and taxonomies, as well as tools to migrate all parts to newer versions of CLS-CAD and fix common modeling issues related to parts containing assemblies. The fourth panel allows uploading and downloading taxonomies to and from the backend, as well as interactive editing of the three taxonomies that the implementation of CLS-CAD for FUSION 360 mandates. The fifth tab allows toggling the display of type annotations in the 3D view of parts (see Figure 5.6). The final tab allows composing synthesis requests and interactively viewing their results.

Figure 5.3 shows the taxonomy editor, which is part of the frontend layer. It is implemented with CYTOSCAPE and loaded into a web view provided by CEF. It is used to freely manipulate the taxonomy, supporting all CRUD operations [115]. This interface is intended to be used for taxonomy creation and making major changes. Right-clicking on edges, nodes, or empty space brings up radial menus. These are used to create nodes, rename, delete, or create an edge from a selected node, or relocate/delete a selected edge. Edges snap to other nodes during creation or relocation. If a left click would create an edge, the target node is circled. If an edge would create an alias within a taxonomy

[1] As described in Section 5.1.1, this diverges from the proposed methodology, as it simplifies the typing process for most use cases.

[2] Primarily used to set up imported datasets for synthesis with a single command.

Figure 5.4: Display of interface for building intersection types

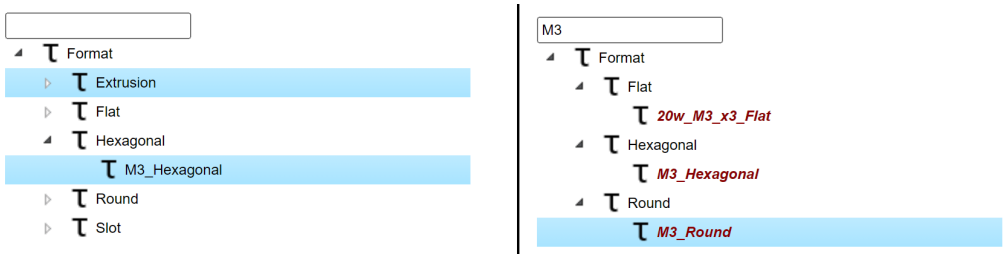
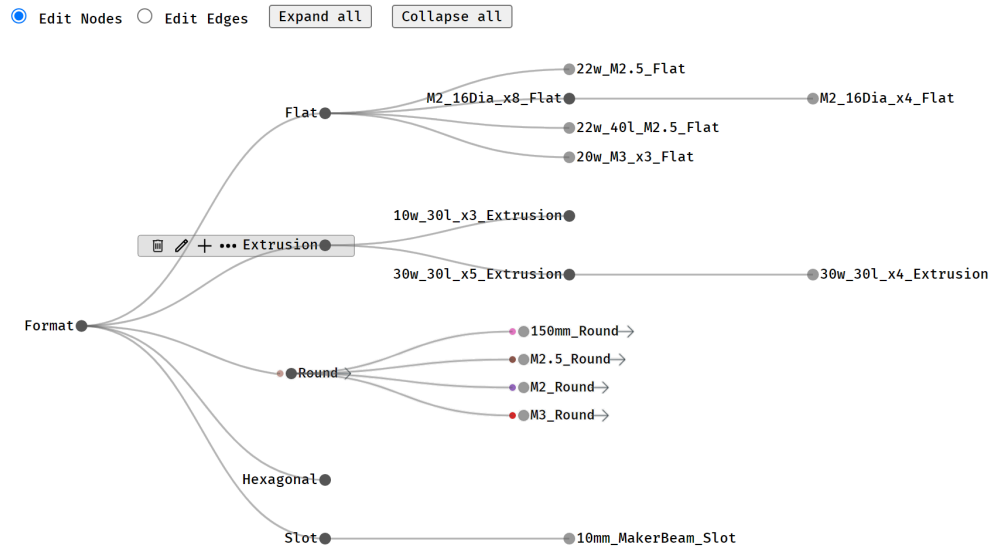


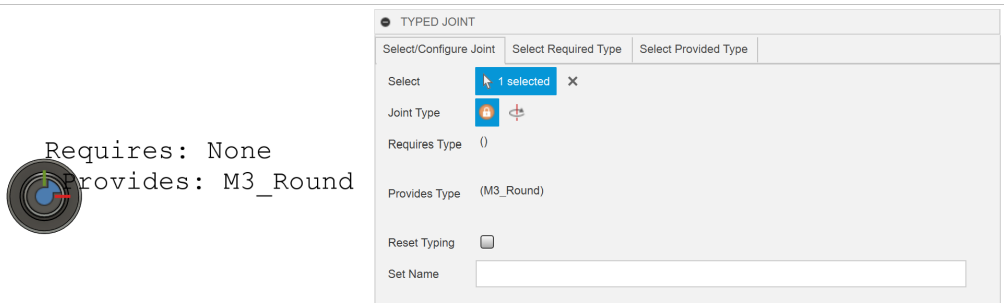
Figure 5.5: Alternative display of interface for building intersection types



and the interface shown in Figure 5.4 is enabled, the edge is shown in red instead of green. Pressing the layout button reorders, centers, and scales the taxonomy in the web view. These functions are shown in Figure 5.3 by overlaying several screenshots. Operations that create or rename types can also be carried out in the interface for building intersection types "on the fly." All types created by CLS-CAD use unique IDS in the taxonomy. There is a separate mapping that assigns these IDS telling names. This allows rename operations without needing to update the CAD files.

Figure 5.4 shows the interface used to interactively build intersection types. The underlying graph of the taxonomy is unrolled into a tree view. For this unrolling to be possible, the taxonomy must be acyclic. If this interface is enabled instead of Figure 5.5, aliasing in the taxonomy is disabled. The interface supports renaming and creating types, as well as searching for types by name, as shown on the right in Figure 5.4. Intersection types are built by selecting

Figure 5.6: Annotation of JointOrigins



multiple types to intersect by means of Ctrl-clicking.

Figure 5.5 presents an alternative interface for building intersection types. In contrast to Figure 5.4, the taxonomy is not unrolled; the underlying graph is displayed "as is." In its initial state after loading, only selecting types is permitted. Types are selected and deselected by clicking their names. Selected types are marked in green, and the resulting intersection type is displayed next to the "Expand All" and "Collapse All" buttons. The interface supports two additional modes that allow editing of nodes or edges. If the "Edit Nodes" mode is enabled, hovering a type produces the menu seen around the Extrusion type in Figure 5.5. This menu allows deleting the type, editing its name, or adding a subtype to it. The "Edit Edges" mode displays colored circles around types. Dragging and dropping these allows changing or deleting edges. Dragging out from a colored circle allows creating a new edge. The edges snap to other nodes while being dragged; clicking finalizes the operation. The colored circles are pictured in Figure 5.5 only for the Round type and its children[1]. Dark grey circles to the right of a type indicate that it has subtypes.

This user interface is less intuitive to use than the one pictured in Figure 5.4 due to being more visually busy and less familiar to most users than list views. However, it better represents the underlying taxonomy and does not impose restrictions regarding aliasing. It also supports all Crud operations, allowing experienced users to make sweeping changes to the taxonomy while annotating types. For all user interfaces that allow building intersection types, there are two available variants: one that loads the interface from Figure 5.4, and another that loads the interface from Figure 5.5.

Figure 5.6 shows how a JointOrigin is annotated in Fusion 360 by means of the Cls-Cad add-in. The pictured user interface allows selecting indi-

[1] This was done by overlaying multiple images.

vidual or multiple JointOrigins. In the latter case, annotating these treats them as a single JointOrigin with multiple locations, leading to identical sub-assemblies. The type that will be assigned to the JointOrigin upon confirmation is displayed. The JointOrigins can also be assigned a motion behavior, the default being a rigid connection[1]. If the alternative motion type, "Revolute," is selected, any joints created between the JointOrigin and another will be revolute[2]. The other two tabs of this interface allow selecting the required and provided types for JointOrigins using the previously shown interfaces. The interface for annotating parts with an intersection type works analogously, reusing the interfaces shown in Figure 5.4 and Figure 5.5. The main difference of these interfaces is which taxonomies are available to select types from, as described in Section 5.1.1.

The interface for posing a synthesis request works in a similar fashion. The first tab allows the user to utilize one of the previously shown interfaces for building intersection types to set the synthesis request target to an intersection type composed of types from the Parts and Attributes taxonomies. The second tab allows for building and assigning types to predicates, which can be added to or removed from a table that displays them. This allows numeric properties to be specified. Synthesis requests can be assigned a meaningful name; otherwise, they are allocated a human-readable unique Id by default.

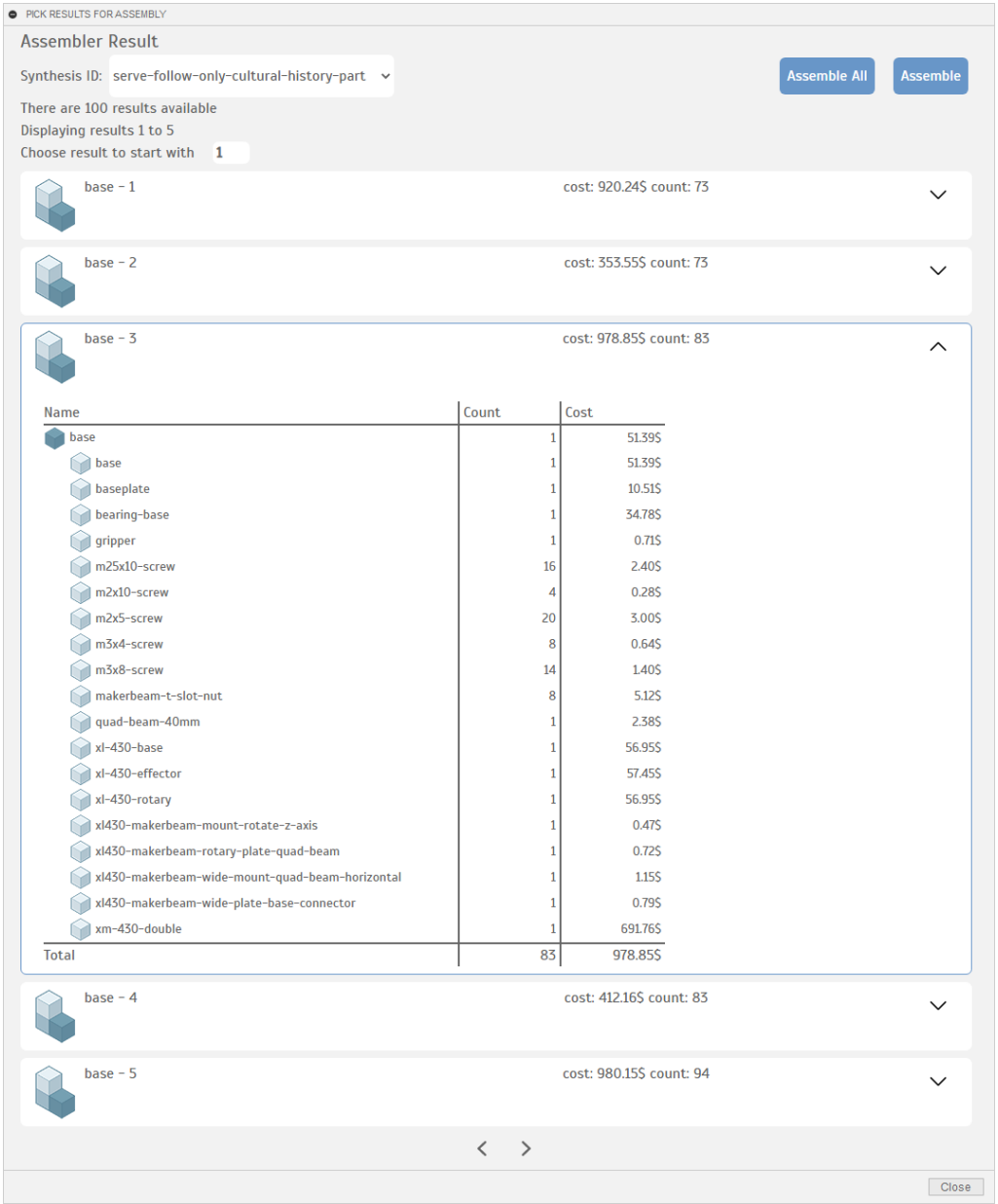
Figure 5.7 shows the interface for previewing the set of synthesis results. In the top dropdown, results containing sets of designs can be selected by assigned name. The interface defaults to displaying the most recent synthesis result. Each synthesized design is displayed as a tile, showing the material costs and the total number of parts in the assembly. Clicking a tile expands it, showing the complete Bom of that design. The designs are paginated. Due to the way that Cls works, the designs are loosely[3] ordered by ascending total number of parts.

[1] This leads to the part becoming part of a link.

[2] Rigid JointOrigins provide the identity function, always resulting in the motion type of the other JointOrigin. Identical JointOrigins result in their shared motion type. Revolute and prismatic JointOrigins compose into a pin-slot mechanism, and so on.

[3] The loose ordering is due to multiple joints being treated as one for the synthesis procedure. The total number of parts is calculated after post-processing the synthesis results. Regarding the unprocessed terms resulting from the synthesis, the number of "logical" parts is strictly ascending.

Figure 5.7: Display of synthesis results with costs and BOM



5.1.3 Optimization of Assembly Implementation

A naive implementation of creating assemblies from instructions output by the CLS-CAD backend is to simply carry them out. This means each part is inserted when the instructions call for it, and then the specified connection is made. In initial implementations of the add-in layer, this approach is used. The resulting code is easy to understand and not prone to errors; however, experiments reveal that by applying knowledge of which operations are fast in FUSION 360, substantial performance gains are achievable. Furthermore, adding quality-of-life features that make the final assemblies of the designs professional and akin to what a designer produces manually is possible. This section aims to chronologically outline observations and considerations that led to improved performance, as they may be useful when optimizing performance in other CAD software.

The first observation regarding performance is that inserting parts in and of itself is a costly operation. The entire CAD file needs to be parsed and translated into BREP. This operation is slower the more complicated the parts being inserted are. As such, the number of insertion operations should be minimal. The minimum amount is exactly one of each unique part occurring within an assembly. Achieving this seems simple: insert each part once, then create copies of it as needed. Creating copies of the part is faster than having to parse the CAD file repeatedly. In most CAD software, practically achieving this is not as simple as it seems. In FUSION 360, inserting a part into an assembly creates the part as a referenced part, meaning the part is a link to its corresponding CAD file. Most other CAD software also takes this approach. This has many advantages; for instance, if a part is changed, all assemblies containing the part update to reflect these changes, and the connections within assemblies update their locations recursively. Additionally, the referenced part can be edited within the assembly, and the changes are mirrored back to the part.

In FUSION 360, attributes added to a part are not available until it is converted from referenced to local geometry. In the following, this conversion is referred to as “breaking the link.” The absence of attributes is sensible, as they are not strictly part of the file currently being worked on. It prevents accidental unintended changes to attributes of linked files. Since attributes are required to identify annotated connection points and create necessary joints for assembly, the link between parts and original files needs to be broken. This is undesirable, as the convenience of updates to parts propagating to the automatically generated designs is lost, resulting in the assembly diverging from how a human designer would create it in this aspect. At face value, this trade-off may appear acceptable in exchange for eliminating the repetitive nature of assembly creation. However, there are also performance implications

of breaking the link. Breaking the link takes a significant amount of time, as BREP geometry that previously only needed to be displayed must be added to the design history of the current assembly and made editable. Internally, this necessitates adding the different types of geometrical entities (boundary loops, surfaces, etc.) to the correct data structures of the current CAD file. The file size of assemblies becomes the combined file size of all individual parts. Additionally, the performance of analysis tools and of motion studies degrades, as the design is then essentially an extremely complicated individual part, which CAD software is not optimized for. The necessary time to break the link depends on the complexity of the part. A further observation that can be made is that after breaking the link of a part to its original CAD file, creating copies of it becomes slower, with the time needed per copy being roughly equivalent to breaking the link of the original. However, creating copies of the part after breaking the link is observable to be marginally faster than creating copies before breaking the link of the part and then breaking the link of all copies.

To circumvent this worst-case scenario, an understanding of how most CAD kernels operate is helpful. A widespread concept in CAD software is that the actual part or component is not displayed but is an abstract entity. What is displayed to the user is an occurrence of the underlying component. In the following, to match the way the FUSION 360 API internally refers to these concepts, the actual part is referred to as "component," and the displayed versions of that component are referred to as "occurrences." Changes to any individual occurrence back-propagate to the component, and from there propagate to all other occurrences, updating them to match the new state of the component. This helps patterning tools to function efficiently; for instance, a common task when designing parts is that a feature or geometry needs to be repeated as a circular pattern along an axis. If in the design history a patterned object is changed, this abstraction allows the expected behavior (e.g., all copies being identical), to be realized. The implementation of the add-in can take advantage of this knowledge and pattern inserted parts after breaking their links instead of copying them. This preserves the attributes on each copy and runs an order of magnitude faster than the aforementioned approach, also inflating the file size to a lesser degree. However, a downside is that if attributes are changed on an individual occurrence, all occurrences reflect this. This downside is significant, as during assembly individual connection points need to be marked as "used up," so that parts do not doubly use the same connection. This can be alleviated by the add-in keeping track of an index for each type of part and attaching it to the n-th corresponding connection point, but this is a cumbersome and not particularly robust solution. As detailed in Section 5.2, generating the assembly instructions by Breadth-First Search (BFS)

instead of Depth-First Search (DFS) is specifically done to avoid the necessity of differentiating between instances of the same connection point in the CAD software. Another way to work around this is to disable the design history after all copies of parts have been made. This converts all occurrences into regular BREP geometry and preserves attributes. It is also fast, due to the geometry already being fully parsed. Losing the option of editing the design history is not particularly significant in this case, as it is automatically generated. A designer can pick a different design instead of heavily modifying a generated one.

Using the aforementioned improvements, an efficient implementation for assembling synthesized designs in FUSION 360 is achieved. However, inserted parts not being linked to their original CAD files is close to a deal-breaker for using the implementation of CLS-CAD for prototyping, as especially during this phase, the designs of parts are likely to change; i.e., non-structural changes like cutouts might be added to save mass. If designers do not remember the exact synthesis request and index of a generated result, they are forced to go through the steps of the design space exploration phase again (Figure 4.4). This is antithetical to the ideas behind CLS-CAD, as repetitive work is meant to be avoided. Additionally, further experiments yield that there are still observable performance gains "left on the table." While creating an assembly from non-linked parts slows down analysis tools, there is an additional effect during the creation of the assemblies: creating joints between non-linked parts is much slower than doing the same between linked parts in FUSION 360, irrespective of whether the design history is enabled or not. The reasons for this are unclear, although it might be assumed that in the former case, the necessary coordinate system transformation is computed for individual BREP entities, while in the latter case, a transformation can be applied to the entire linked part. As such, to create an implementation that is as efficient as possible and yields high-quality assemblies, the utilization of linked parts is necessary.

As previously explained, the issue with linked parts is that their contents are not part of the current CAD file, and thus cannot be modified or directly interacted with. However, this is perplexing, as FUSION 360 interacts with them when the user manually creates a joint. In this case, their respective `JointOrigins` are selectable; created joints are editable and become part of the design history. As such, the required information must at some point become part of the current CAD file. Performing a test that uses the API to create a joint between two randomly picked `JointOrigins` shows that this is not possible, which is consistent with the previous description of how linked files function in FUSION 360, but not with the behavior observable in the user interface. Examining the documentation of the FUSION 360 API with this in mind, sense can be made of two cryptic properties/methods of the

`JointOrigin` class.

> `JointOrigin.assemblyContext`: "Returns the assembly occurrence (i.e. the occurrence) of this object in an assembly. This is only valid in the case where this is acting as a proxy in an assembly. Returns null in the case where the object is not in the context of an assembly but is already the native object."
>
> `JointOrigin.createForAssemblyContext`: "Creates or returns a proxy for the native object - i.e. a new object that represents this object but adds the assembly context defined by the input occurrence."

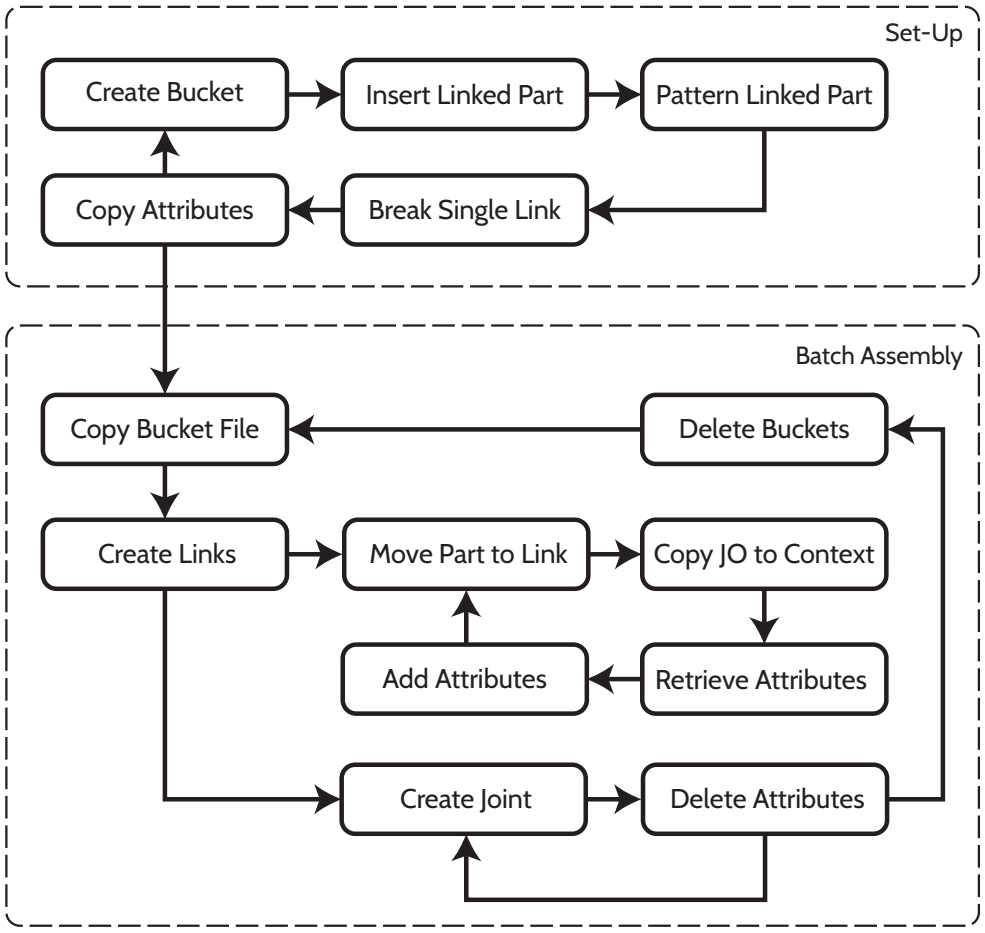
The method `createForAssemblyContext` can be used to bring solely the `JointOrigins` of linked parts into the current CAD file, thus making them part of the current design and allowing them to possess attributes. While this does not retain attributes previously set in the original linked files, as these newly created `JointOrigins` are quintessentially new entities, this can be worked around. The method `createForAssemblyContext` also allows specifying which occurrence to use as a parent for the created assembly-specific `JointOrigin`. This can be leveraged to mimic the behavior of the FUSION 360 user interface regarding which component or sub-assembly acts as the owner of a created joint, leading to a cleaner assembly structure. With the prior approach, all created joints are shown at the top level, making it very hard to discern which joints are relevant to the kinematics of synthesized designs and which are not.

Figure 5.8 depicts the steps that the final implementation of the assembly algorithm performs in FUSION 360. This version of the implementation also contains an optimization for batch assembling many designs back-to-back, increasing the speed of design creation by roughly three to four times asymptotically for designs of medium complexity[1]. The achievable speed-up increases with the complexity of assemblies.

The implementation first goes through a setup procedure, which builds a reusable file template from which all assemblies of the synthesized designs can be created. This is done because non-linked parts and initially inserting parts are the main drivers of runtime. By computing the maximum number of occurrences of each part across a batch of designs, such a template can be set up. Since there is nearly no performance difference between creating two or two hundred copies of a linked part, the only cost associated with this is that extraneous parts w.r.t. individual assemblies are present in these

[1] Designs of up to at most a few hundred parts within the assembly.

Figure 5.8: Overview of assembly algorithm



template files. However, since product families and design alternatives are generally speaking homogeneous[1], the majority of parts inserted are utilized in all designs. For the use case detailed in Chapter 6, even when assembling two designs, the template file leads to a performance gain.

The template file initializes a bucket for every type of part. This bucket is a component that serves as a parent for all occurrences of that type of part. The part is inserted as a linked component, and a patterning tool[2] is used to create copies of the part within the bucket, the number of copies being the maximum required amount. The link of the originally inserted part is then

[1] To a certain extent, reusability and sharing of parts is considered good practice in engineering.

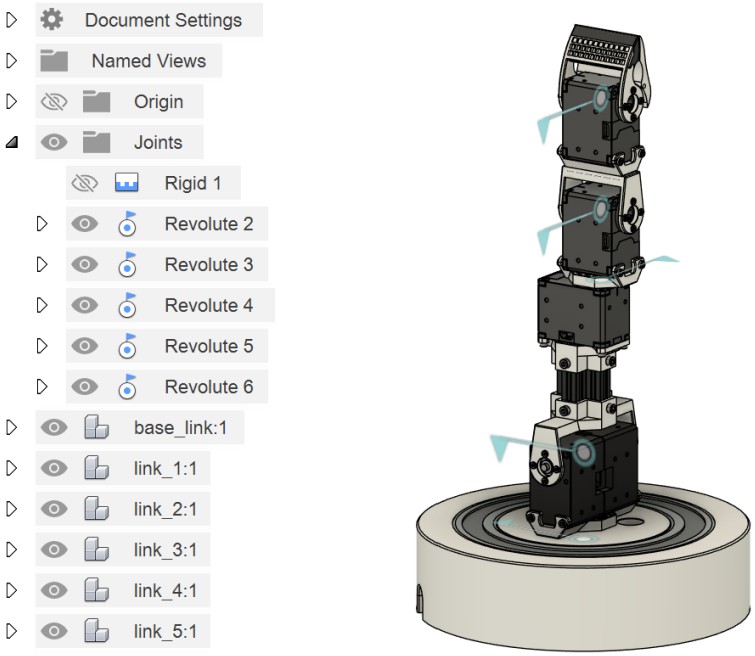
[2] The rectangular pattern tool is used, as it allows for marginally better user feedback during insertions.

broken to obtain access to its attributes. These attributes are copied over to the component serving as the bucket, and the now no longer linked part is subsequently deleted. Copying the attributes to the bucket instead of not deleting the part is done to improve file loading times; in general, non-linked parts have a greater impact on these. This procedure is repeated for each part that occurs within the batch of designs. During the entire procedure, the design history is disabled to avoid increasing file size and degrading performance, as Fusion 360 would otherwise inspect hundreds of steps that occurred during setup to be recomputed if there are later (unforeseen) changes. Since copies of linked parts, which are in turn themselves linked parts, do not have any attribute information, they are completely inert with regard to Cls-Cad.

For each design to be assembled, a copy of the template file is made. Copying the template file is a regular file operation and thus much faster than even a single part insertion or breaking of a link. Within the template file, several Link components are created (links in terms of kinematic chains). The implementation of Cls-Cad allows marking connection points as non-rigid, i.e., as revolute. The Cls-Cad backend computes the links of each design according to the definition given in Section 3.4, loosely speaking, the maximal sets of parts that act as a single rigid part together. The assembly instructions are then executed. Each assembly instruction detailing a connection needing to be made first leads to a part being retrieved from a bucket and moved to the associated link component. Next, only the `JointOrigins` present on it are brought into the assembly context and parented to the link component. Attributes present on the respective bucket are re-applied to these `JointOrigins`, essentially creating a virtual part consisting solely of `JointOrigins`, visible to the Cls-Cad add-in, with attached linked geometry. After every part is moved to the correct link, the assembly instructions are carried out in order, creating joints as specified. The resulting structure of the Cad file is akin to zipping the created `JointOrigins` to the assembly instructions. Whenever a joint is created by the Cls-Cad add-in, the attributes of the `JointOrigin` are deleted, returning that `JointOrigin` to being inert. As the order of moving and creating the `JointOrigins` matches the order of assembly instructions, there is no need to utilize any indexes or the like; the next non-inert `JointOrigins` of a given Id found are always the intended ones for construction. Finally, the buckets, including any remaining unused parts, are deleted from the file. This process is repeated for every design that is present in the batch.

Figure 5.9 shows the file structure that results after executing the assembly instructions as detailed. The pictured robotic arm is arranged into separate links. All rigid joints are correctly set; however, they are hidden alongside the now inert `JointOrigins` to avoid cluttering the design visually. Since all parts have been correctly parented to their computed links, the joints that connect

Figure 5.9: Example of resulting CAD file structure



kinematic links are parented to the root of the document. This means that out of hundreds of joints, only the joints of kinematic relevance for the actuation of the robotic arm are shown to the designer and are easily accessible in their own folder in the file structure. Whenever multiples of a single part are added to a link, the add-in wraps these in a component to give a BOM-like view to the designer, detailing how many are contained in the component's name. The entire design is made up solely of linked parts, meaning that performance is good and that the design automatically updates parts and all relative positions when changes occur. Updating relative positions is an advantage of using joints, which designers often forgo on small parts like screws due to the added time investment and manual labor it requires [134]. For FUSION 360, the good performance also means that the automatically generated and assembled designs enjoy inverse kinematics "out-of-the-box," allowing a designer to drag each part of the robotic arm, with the remaining LINK's positions updating accordingly. This allows designers to quickly perform a rough check of the manipulability and kinematic behavior of assembled robotic systems.

5.2 Implementation of Backend

In this section, technical details of the matching implementation of the backend for the previously discussed add-in for Fusion 360 are discussed. The backend is built upon FastApi, which provides the Rest Api functionality, MongoDB, providing a non-relational remote database, and MontyDB, providing an implementation of MongoDB that runs locally with no need to install additional software [76, 104]. There are endpoints for submitting parts, uploading and downloading taxonomies, requesting synthesis, retrieving all synthesis results, or retrieving synthesis results based on filters. Endpoints that allow posting data perform validation using Pydantic models. All of these endpoints enforce grouping parts, taxonomies, and results based on a project Id, to allow managing different sets of parts for experiments. These project Ids are not generated but instead correspond to the unique identifiers that the Fusion 360 cloud functionalities assign projects internally. The unique Ids of the parts stored in the database are also those that Fusion 360 assigns internally. A non-relational database is used because of such databases' ease of use, obsoleting considerations regarding database schema and future migrations. The performance degradation that non-relational databases suffer from for very large amounts of data is not considerably worse than relational databases, and for fewer than ten thousand entries, MongoDB even outperforms a relational database like MySQL [59, 89]. For the use case of synthesizing Cad designs, there are not millions of parts to choose from[1]; as such, using a non-relational database and benefiting from performance gains and simpler code is desirable.

In the following, the generation of the repository in the backend is covered, followed by considerations on how synthesized terms output by the ClsP framework are post-processed into assembly instructions.

5.2.1 Generation of Repository

The generation of the repository proceeds by and large as detailed in Section 4.5. This subsection highlights differences that the implementation of the backend makes to account for specifics of the implementation of the Cls-Cad add-in for Fusion 360, as well as changes that improve performance.

Listing 5.3 shows excerpts from the code that generates types of parts to be added to a repository. The variable `count_name` is a unique name assigned to each constraint's matching predicate, and `count_types` is the set of types forming the constraint's intersection type. Predicates count the total

[1] No human designer, even with tool support, could handle such an amount of parts.

Listing 5.3: Overview of repository generation

```
if taxonomy.check_subtype(
                    provides_type,
                    Type.intersect(
                        [Constructor(type_name) for type_name in count_types]
                    ),
                    dict(),
                ):
                    part_type = part_type.AsRaw(
                        partial(
                            collect_and_increment_part_count,
                            count_name=count_name,
                            multiplicities=multiplicities,
                        )
                    )
                else:
                    part_type = part_type.AsRaw(
                        partial(
                            collect_part_count,
                            count_name=count_name,
                            multiplicities=multiplicities,
                        )
                    )

# Instead of a -> b -> c -> d, we now take Use(a).Use(b).Use(c).In(d)
for uuid, joint_types in list(types_with_count_by_uuid.items())[:-1]:
    part_type = part_type.Use(f"{uuid}", joint_types)
provides_type = next(reversed(types_with_count_by_uuid.values()))

part_type = part_type.In(provides_type)
```

number of occurrences of parts in assemblies that satisfy their intersection type. Parts whose provided type is not a subtype of a constraint's intersection type[1] are assigned a predicate that propagates the number of occurrences found in attached sub-assemblies. As the occurrence of multiple identical sub-assemblies is abstracted away by the add-in and subsequently handled by Fusion 360, the repository generation must keep track of how many sub-assemblies each logical one will correspond to in assembled designs so that predicates operate on the correct amounts. To do this, predicates multiply the counted number of parts of each sub-assembly by its corresponding joint's multiplicity. This multiplication is a constant factor depending solely on each logical JointOrigin present on each part; the computation of values then occurs recursively during synthesis when predicates are evaluated. If the part's provided type is a subtype of a given constraint's type, a value is added to the sum of the number of parts of interest, as detailed in Section 4.4. This value is constant, due to the fact that the applicable multiplicity is applied by the "parent" during synthesis as part of the recursive computation, not by the part

[1] A type is a subtype of itself.

itself.

Another interesting aspect of the excerpt shown in Listing 5.3 is found in line 25. The implementation of CLsP provides the option to use a combination of the `Use` and `In` keywords instead of arrow types. These act similarly to arrow types, e.g., `P: Use(a).Use(b).In(c)` and $P : A \to B \to C$ behave similarly; however, the former does not permit splitting types. Disallowing splitting types means that, for instance, $P' : A \to B$ cannot be attached to JOINTORIGINS of type $A \to B$ and B for part P, but only to JOINTORIGINS of type B, still requiring a part of type A. Regarding the methodology, for some use cases, inserting a part as a result of splitting types might make logical sense; however, for modular robotics, it does not. A JOINTORIGIN is intended to only provide the type explicitly modeled by the user; generated combinators should not allow for the synthesis to deviate from that. As such, this implementation detail specifically prevents this, ensuring that multi-arrow types only provide their right-most intersection type[1]. This also leads to a performance gain during synthesis. The synthesis algorithm does not need to compute all ways that types can be split and consider how split types could be fulfilled. As this effect is recursive, the performance gain can be significant, depending on utilized parts[2].

5.2.2 Post-processing of Inhabitants

The CLS framework does not directly synthesize sets of assembly instructions; instead, it returns a tree grammar. Enumerating this grammar yields terms. Each term is a rose tree of combinators representing different parts. These terms need to be interpreted and post-processed to create assembly instructions. Each combinator within a term can be translated to a JSON description of the part it represents. This JSON has an object called `connections`, which maps unique identifiers of JOINTORIGINS to corresponding JSON representations of the children of that combinator. The JSON representations of children additionally receive the multiplicity their parent part specifies for the corresponding physical JOINTORIGINS. The representations also receive the resulting motion types of the joints to be made, computed as mentioned in Section 5.1.2 in the footnotes pertaining to Figure 5.6. This yields a JSON description of the assembly tree; however, the multiplicities of JOINTORIGINS still need to be recursively resolved.

[1] This could in theory be an arrow type. In contrast to the previous example, this would be permissible, as the user explicitly modeled the provided type. The issue with splitting is that it leads to side effects that the user did not model.

[2] The effect is more pronounced the more JOINTORIGINS are present on parts on average.

The first step in post-processing resolves the multiplicities. This is done by performing a pre-order DFS traversal of the JSON. During this traversal, each `connections` object is updated by creating as many copies of each child as the multiplicity states. The copied children and the original child's multiplicities are set to one. After this step, the resulting "expanded" JSON corresponds to the later assembly tree in FUSION 360. From this JSON, the individual links can be found; new links begin wherever a non-rigid joint occurs. Furthermore, the total number of parts, the total cost of the design, and other metrics not relevant to the predicates can be computed by using any traversal method. The post-processing step computes and saves these metrics and the resulting BOM.

It is desirable to obtain a series of instructions for the add-in to execute in order, instead of assembly trees. Executing these is simpler to implement for add-ins across different CAD software, and they offer the added benefit of serving as human-readable instructions. To compute such instructions, the post-processing performs a BFS traversal of the expanded JSON. Using BFS is recommendable; it simplifies carrying out the computed instructions. This is due to the JOINTORIGINS of multiple identical parts being indistinguishable. They have the same attributes and entity type. To be able to distinguish them, an add-in's implementation needs to keep track of each different ID of these JOINTORIGINS as well as to which of multiple identical parts in the assembly instructions they belong. If this is not done, a part intended to be connected at the tips of an assembly tree may be mated to an unused identical JOINTORIGIN closer to the root. If computing the assembly instructions using BFS traversal, such confusion cannot arise. As the instructions fill out each "layer" of the design before proceeding to the next, there can be no unused JOINTORIGINS on previous layers. As such, an implementation can simply create joints between each inserted part's JOINTORIGIN and the first occurring instance of the target JOINTORIGIN in the assembly that has yet to be used. From the perspective of physically assembling synthesized designs, these instructions are less likely to result in occluded parts, e.g., screws that are obstructed by some other part and thus cannot be inserted and tightened, as potentially physically occluding layers are only inserted later on.

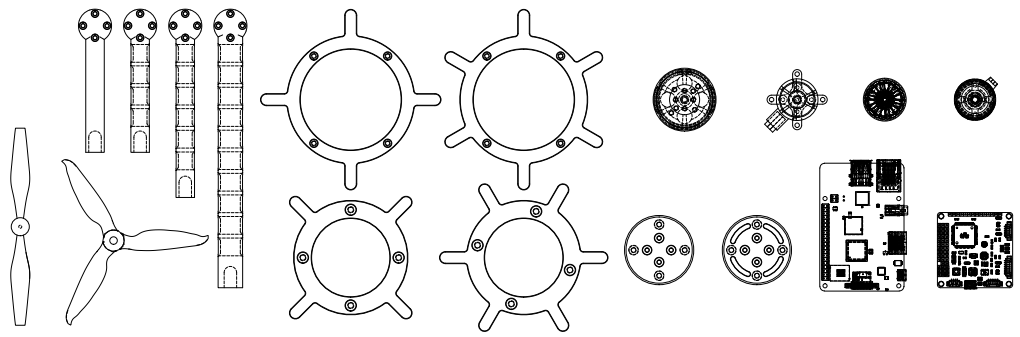
CHAPTER 6

Validation

The proposed methodology and its corresponding implementation provide the means to synthesize designs and assemble them in Fusion 360. While the implementation encompasses a full set of tests providing complete line coverage, the methodology needs to be experimentally verified. It was not possible to obtain access to a large Cad dataset from industry to this end. Companies strictly guard Cad designs of their products and the parts they are composed of. To work around this, two self-developed Cad datasets are utilized.

Chapter Organization In the following, first, these self-developed Cad datasets and the considerations taken during their design are described. Then, the carried-out experiments are summarized, and a performance benchmark between a human designer and Cls-Cad is described. Finally, the results of the experiments are evaluated.

Figure 6.1: Overview of dataset for synthesizing drones



6.1 Sets of Modular Components

The developed datasets allow the synthesis of designs within two highly relevant categories of robotics: drones, such as quadcopters and hexacopters, as well as robotic arms. The datasets were created for different versions of CLS-CAD. The dataset for synthesizing drones predates CLSP. Accordingly, this dataset is more focused on highlighting variance and flexibility; not on iterative exploration of results.

6.1.1 Dataset for Synthesizing Drones

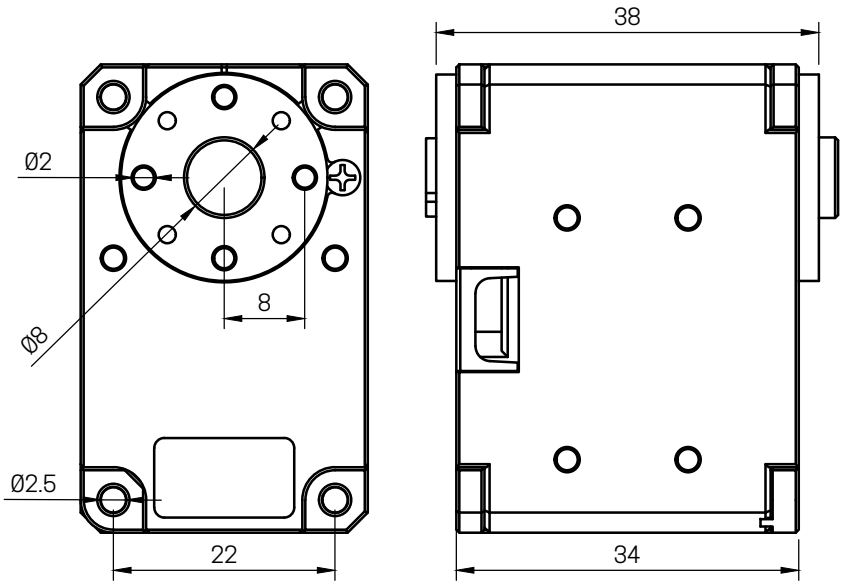
This dataset permits the creation of different drone architectures. It provides various options for single-board computers as controllers, wingspans, propellers, and so on. These parts have different compatibilities with each other; for instance, a propeller can only be attached to a matching shaft of corresponding diameter.

Figure 6.1 displays some of the parts that make up the dataset. Off-the-shelf parts like screws and nuts have been omitted, as well as batteries, lower parts of the main frame of the drones, electronic speed controllers, additional propellers, and some motors. From left to right, Figure 6.1 shows two different propellers, arms with different wingspans, some optimized to be lightweight, bases that allow mounting different single-board computers, two adapters, one of them lightweight, that allow mounting different motors to the arms, four different motors, and finally two different single-board computers[1].

This dataset illustrates one of the advantages CLS-CAD brings to larger teams. Some of the motors within this dataset cannot be mounted to the provided arms as is. If additional adapters that allow the mounting of these motors are designed and added to the dataset, then designers in charge of creating assemblies automatically receive designs that include these motors and, as a result, potentially other propellers that previously did not have a compatible motor shaft for mounting. This affords flexibility, as updating designs to account for new parts as they become available[2] takes less effort and also happens automatically without one team needing to inform the other of newly designed parts. Apart from this, the dataset is intended to illustrate the usage of the `Attributes` taxonomy, thus featuring many lightweight parts. For instance, during the creation of the dataset, the convention that lightweight parts only allow attaching other lightweight parts is applied, showing how

[1] The two single-board computers are a Raspberry Pi 3 and a MultiWii.
[2] Or unavailable.

Figure 6.2: Dynamixel XL430-W250 servomotor for modular robotics [31, 108]



some of the functionality of predicates can also be obtained by using a well-thought-out or use-case-specific model.

6.1.2 Dataset for Synthesizing Robotic Arms

This dataset stems from previous work on synthesizing robotic arms for use with ROS [31]. It is designed to utilize popular small-format servomotors (DYNAMIXELS) produced by ROBOTIS, which are especially suited for modular robotics as they can be daisy-chained, simplifying wiring. This dataset is suitable for versions of CLS-CAD based on CLSP, highlighting functionality enabled by predicates. Unlike the previous dataset, synthesis can yield infinite sets of results, specifically if the DOF of the desired robotic arms is not specified[1].

Figure 6.2 shows a technical drawing of an X-series Dynamixel servomotor, specifically an XL430-W250. This highlights how third-party engineering is compatible with the notion of taxonomies and intersection types. All motors within the X-series share their physical dimensions and have identical options for being mounted to other parts. Their horns[2] are also compatible with each other. The mounting formats of their horns form a taxonomy; the higher-end

[1] This follows by induction over the number of links; another link can always be added.
[2] The output of a servomotor that allows attaching other parts and transmits torque.

X-series servomotors provide eight tapped holes circularly arranged, while the lower-end servomotors only provide four such holes, arranged on a matching diameter. ROBOTIS themselves advertise this taxonomy in their kits; output brackets can be attached to either[1].

Figures 6.3 and 6.4 show how suitable parts for interacting with the servomotors can be designed from technical drawings. By using the key measurements of the interfaces that the servomotor offers, compatible mounting plates and output brackets are designed. While designing, the taxonomy becomes apparent. For instance, the horn of the servomotor consists of a circular pattern with a number of threaded holes. These two aspects immediately translate into a taxonomy; the circular pattern becomes a type, the diameter of the circle of the pattern becomes a type, the occurrence of threaded holes becomes a type, and the number of threaded holes becomes a type[2]. The way that designed parts can physically be connected depends on the types of these interfaces. From here, several different variants of the parts are designed, changing aspects of the part that are not important to its physical connectivity, but to its function and purpose. For instance, lightweight parts that use latticing to reduce total mass, wider parts that allow two motors to be connected next to each other, angled parts that allow for different kinematics, and so on, are introduced to create structural variance. These share the same types for how they connect or utilize subtypes, but their different purposes and attributes form the entries of the `Parts` and `Attributes` taxonomies.

Figure 6.5 gives an example of two additional interfaces that are introduced to connect parts that are not off-the-shelf within links. The technical drawing on the left shows that flat surfaces are connected by three M3 screws going through holes, centered on the surface and spaced in 10MM intervals. The technical drawing on the right shows that when extending larger distances, five extrusions of ten by ten profile are arranged in a cross pattern and affixed by M3 screws. These design decisions are also encoded in the taxonomies.

The complete set of these parts and the corresponding taxonomies are released under the APACHE2 license and can be obtained from Zenodo[29]. They provide an example of how the described methodology is applied to a set of pre-existing parts to enable usage with CLS-CAD. The use of several well-standardized interfaces, off-the-shelf parts illustrating applicability to external parts, and a plethora of attributes for the predicates to operate on makes this dataset well-suited for validating and evaluating CLS-CAD's efficacy.

[1] The arrangement of four holes is the subtype of eight holes, as long as the diameters and threads match.

[2] The numerical types can either be simple types and intersected, or be specifically subtypes of the "circle" or "threaded hole" types.

Figure 6.3: Mounting format for holding Dynamixel X-series servomotors [31]

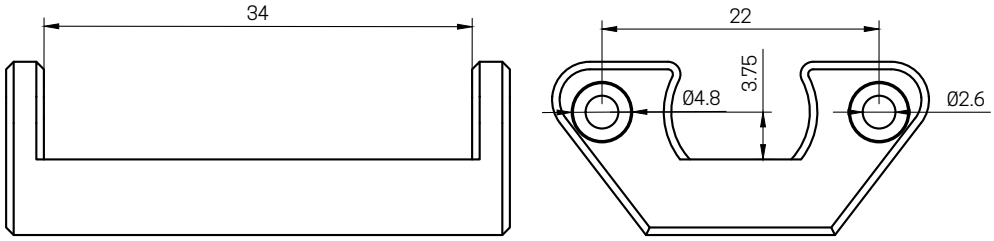


Figure 6.4: Horn format for connecting to Dynamixel X-series servomotors [31]

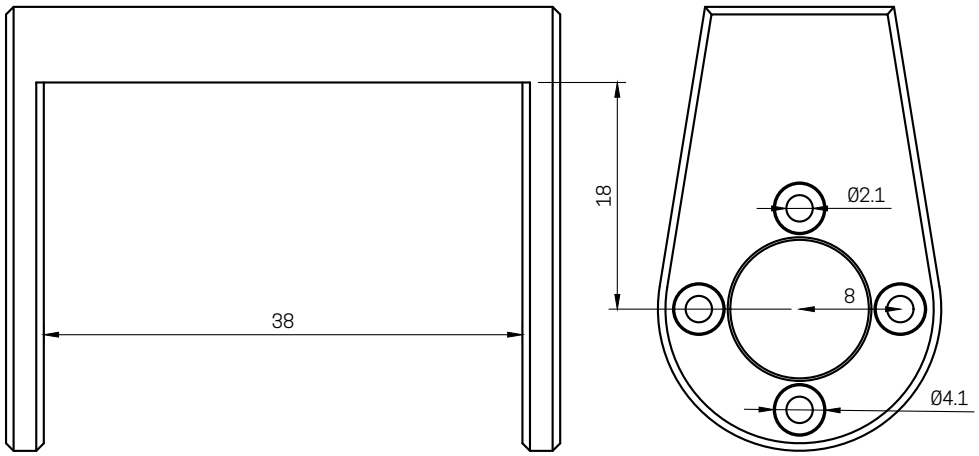
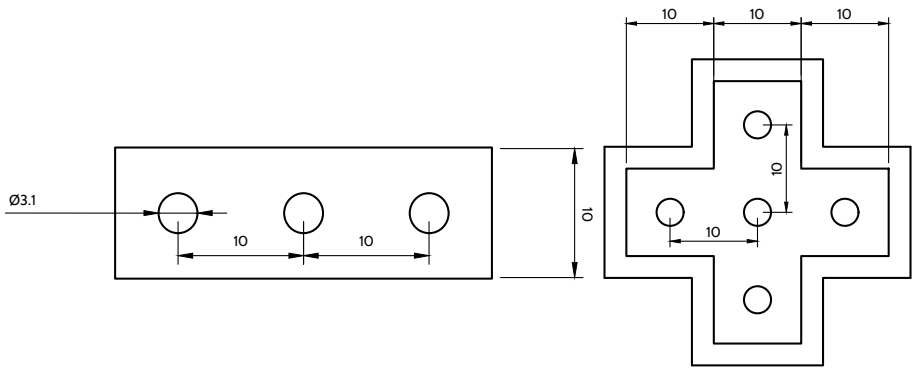


Figure 6.5: Example of flat mounting formats within links [31]



6.2 Experiments

Within this section, the experiments carried out to evaluate correct functionality, efficacy of the methodology, as well as the performance of CLS-CAD are described. Evaluating the first of these is straightforward, albeit not possible beyond providing rigorous unit tests and manually checking the designs of hundreds of automatically generated assemblies. Verifying that correct assemblies are produced in any possible use case cannot be done, due to the aforementioned lack of openly available industrially relevant datasets. However, unit tests guarantee that the assembly algorithm and the synthesis work correctly. Issues like the modeled taxonomy being faulty, annotations being incorrect, or geometric interdependencies leading to overlapping parts are not issues that CLS-CAD can fix. Section 7.2 gives some ideas on how the first two issues can be avoided, while the CAD software itself can be invoked to detect the third. For the drone dataset, all synthesized designs are assembled and manually inspected. For the robotic arm dataset, all synthesized designs up to six DOF are assembled and manually inspected. In both cases, correct function is observed.

In the following, an experiment in which the iterative design process, as proposed in Section 4.3, is carried out, followed by an experiment to determine the approximate performance gain of CLS-CAD over a human designer.

6.2.1 Iterative Design of Robotic Arm

To verify that robotic arms exhibiting specific properties useful to designers are found, this subsection gives a brief overview of how concrete steps using the methodology look for the robotic arm dataset. An initial synthesis request targets the BASE of a robotic arm, with no additional predicates. This leads to more than one hundred robotic arm designs being output[1]. Inspecting these results reveals that an issue with the request is that no DOF was specified for the robotic arms; hence, there are infinite results.

To remedy this issue, the next synthesis request introduces a predicate that operates on the `Parts` taxonomy. By specifying that designs need to contain exactly six occurrences of the type MOTOR, the DOF are constrained. Requesting designs that have six motors within them leads to 82 total robotic arms. Briefly skimming each result reveals that the robotic arms vary significantly in price. At this point, the aggregated metadata is no longer sufficient to refine

[1] The GUI truncates infinite sets of results to a maximum of one hundred, operating under the assumption that at this number a designer will only briefly glance at some of them before performing a more specific query.

the request. By utilizing the preview function for the Bom of each design, we observe that the main driver of the cost difference is the type of X-Series motors used for the shoulder joint[1] of the robotic arm. To make the results more relevant, the request is refined by specifying the desired torque category of the shoulder joint[2]. There are multiple ways to do this, but in general, it is best to be as specific as possible with the intersection type to be counted. In this case, requesting either 68mm_Mounting ∩ Larger3Nm_Torque or the same type but with a different torque identifier yields the desired results. For this dataset, 68mm_Mounting uniquely describes motor combinations suitable for the shoulder joint[3]. Alternatively, during type annotation, they could have also received an `Attribute` type that explicitly marks them as shoulder joints[4]. Requesting a robotic arm with a high-torque shoulder joint leads to just 16 results, while a low-torque shoulder joint leads to 33 results.

In the following, the low-torque variant is pursued, i.e., a budget-friendly robotic arm for educational purposes is being designed. Examining the 33 different designs, it becomes apparent that some of them are still significantly more expensive based on the aggregated metrics. Reviewing the Boms once more, it can be seen that some still contain more powerful, and thus more expensive, motors for the other joints. Additionally, it is observed that some of them use the more expensive 68mm_Mounting motors for joints other than the shoulder. The request is refined to fix these issues by disallowing the occurrence of more expensive motors by setting their count to zero.

Posing a request that eliminates all designs but those that have exactly one budget-friendly shoulder joint motor yields five results in total. These are few enough that all of them can be assembled and inspected, but to better illustrate the methodology, it is assumed that one or two of these are randomly picked and assembled. Inspecting one such assembly, two issues are immediately identified that are easy to overlook based on the Bom alone. Firstly, there is one Dof missing. Looking at the robotic arm and the displayed joints in Fusion 360 reveals the reason to be that the synthesis request did not account for the degree of freedom that the effector[5] of the robotic arm provides. Secondly, there are robotic arms that have redundant degrees of freedom, with joints adjacent in the kinematic chain sharing the same axis of rotation. In response, the request is refined, the requested number of Motors

[1] The second rotating joint of a robotic arm is usually considered the shoulder joint.

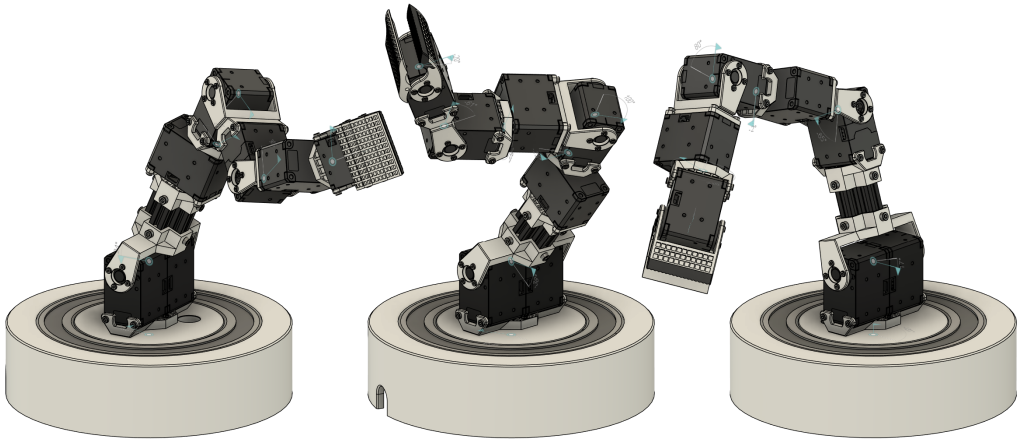
[2] In a real use case, based on the desired payload to be manipulated by a robotic arm, the torque requirement is calculated; the decision then stems from that.

[3] The shoulder joint is formed by a compound motor, which combines the output torque of two individual motors, making it physically wider.

[4] The taxonomies contain $UprightDoubleMotor\_Tag$ for this purpose.

[5] The dataset uses a gripper which contains a motor for all designs.

Figure 6.6: Robotic arms identified through iterative refinement



is increased by one, and a predicate that enforces the presence of at least two motors with a vertical instead of a horizontal axis of rotation is added[1]. This updated request returns a total of three robotic arms. With these, the last step of the workflow described in Section 4.3 is entered. The robotic arms are all assembled in Fusion 360, their kinematics interactively explored using motion studies and Fusion 360's inbuilt inverse kinematics, and final customization, like logos or other branding, is applied.

Figure 6.6 shows the three final resulting robotic arms. Due to being made from the same modular components and fulfilling the same restrictive set of predicates, they are similar; however, they differ in the placement of their axes of vertical[2] rotation. This can make a significant difference depending on the use case. For example, for pick-and-place operations where the rotation of the placed object is relevant, being able to rotate the effector in place is desirable.

[1] The dataset does not contain parts that allow vertical axes of rotation to occur consecutively. In robotics, designing a part that permits this would be fundamentally nonsensical, as out of two joints whose axes of rotation are not just parallel but identical, one of them is non-existent in a kinematic sense.

[2] Vertical w.r.t. the original orientation of each motor in its own frame of reference.

6.2.2 Performance Benchmark

As the technical correctness of the implementation of Cls-Cad and the applicability of the proposed workflow are validated, it remains to compare the efficiency of Cls-Cad with that of a human designer. The efficiency gained is important, as using Cls-Cad requires an upfront investment of time to develop taxonomies and annotate parts. While this investment amortizes with every set of synthesized designs that replaces manually created design iterations, there are use cases where total variance is low or the manual design process would not require many iterations. The speedup in comparison to a human designer determines the break-even point.

In general, it is not possible to precisely measure the speedup that Cls-Cad provides over a human designer. This is for several reasons. Firstly, this is due to the modular parts and the structure of desired results having a major impact on the gained efficiency. For instance, if designs require a lot of repetition, i.e., surfaces requiring the insertion of hundreds of screws, or if parts need to be connected in occluded areas frequently[1], then there is more efficiency to be gained by automation through Cls-Cad. Secondly, some products require fewer design iterations when being manually designed than others. Products that are constructed from cheaper parts and consist primarily of off-the-shelf parts tend to require fewer design iterations than those where several different custom parts need to be weighed against each other. Fewer design iterations mean less design work, and thus, less work to automate. In the same vein, it is not possible to generally state which manual design iteration corresponds to which step of the workflow for Cls-Cad. An initial synthesis request takes only a few milliseconds to complete, and how fast its results can be evaluated depends on the human designer. With a manual design process, the creation and evaluation processes overlap, as the designer makes judgments while creating the assembly and reacts accordingly. Finally, the speed at which designers operate their Cad software varies. Awareness of specialized tools within the Cad software accelerates operation, as does adeptness with specialized input devices like 3d-mice.

A comprehensive experimental assessment should recruit a large number of designers working in the industry and benchmark their performance against the implementation of Cls-Cad across multiple different sets of modular components and designs, varying between datasets designers are familiar and unfamiliar with. This was not possible. A Cad designer is a well-paid position, and thus it is difficult to convince companies to sacrifice a cumulative total of a few thousand working hours. Instead, Cls-Cad is benchmarked by compar-

[1] This necessitates moving the camera in the Cad software and toggling the visibility of occluding parts. In general, this is time-consuming and frustrating.

Table 6.1: Performance comparison between framework and human (hh:mm:ss) [34]

DOF	Request	Single	Batch	H-Single	H-Batch
4	00:00:01	00:00:52	00:00:14	01:06:05	00:42:25
5	00:00:02	00:00:58	00:00:16	00:59:17	00:23:11
6	00:00:05	00:01:03	00:00:22	02:01:28	00:32:06

ing the time taken by a singular CAD software user of average proficiency to recreate a set of reference assemblies with the time it took to synthesize and automatically assemble them[1]. The experiment is carried out twice, with different rules imposed on the designer as well as CLS-CAD. In the first iteration of the experiment, the designer starts with a blank file and recreates the reference assembly within it. The designer may only insert parts from the modular components but is allowed to reuse any sub-assemblies created during this process. All tools available in FUSION 360 on 06.09.2023 could be used. The implementation of CLS-CAD is restricted from using speed-ups achievable by means of template files. This is intended to compare the performance of creating an initial design between human designers and CLS-CAD. The second experiment is carried out without these restrictions. The human designer may recycle any previous designs, not needing to start with an empty file. This is intended to represent typical practices for most companies. As products are often similar, new designs are frequently created by retooling existing ones. As stated in Section 2.1.1, "This risks shortcomings of previous designs becoming the status quo over time," but such effects are not being taken into account for the performance benchmark; only speed is being measured, not quality. The implementation of CLS-CAD uses template files to speed up the creation of additional designs for the second experiment.

Table 6.1 lists the outcomes of performing the benchmark procedure for reference robotic arms of four, five, and six DOF, excluding the effector. The REQUEST column lists the time taken for the previews of the robotic arms to be generated, e.g., executing the synthesis procedure and post-processing the results. The request utilizes a similar amount of predicates as in Section 6.2.1. The SINGLE and H-SINGLE columns list the times taken by CLS-CAD and the human designer for the first experiment, and the BATCH and H-BATCH columns for the second. It can be seen that if a design iteration can be realized by only the preview mechanism, then CLS-CAD provides a speed increase of about three orders of magnitude over having to learn initial lessons by drafting a

[1] Ideas on how to perform more comprehensive benchmarks in the future are given in Section 7.2.

rough first design. In the other comparisons, CLS-CAD still provides about two orders of magnitude faster assembly generation. While industry professionals may be able to create the assemblies faster than the benchmark CAD user, it seems unreasonable to assume speed-ups past a factor of two or three.

It should be noted that in the columns H-SINGLE and H-BATCH, the time taken for assembling a design of higher DOF, and accordingly more parts[1], is lower than that of the previous DOF. This highlights the difficulties with benchmarking CLS-CAD previously described. Due to there being fewer occluded parts in the higher DOF design, less camera movement is necessary, and joints are easier to create as a result. Also, while the design may have more parts in total, it may have fewer different types of parts, which simplifies the task for humans. These effects can be expected to be more or less pronounced depending on the human designer. To average them out, more data points are needed.

6.3 Evaluation

This chapter introduced two datasets for synthesizing CAD designs and detailed experimental validation carried out on the dataset for synthesizing robotic arms. Based on the unit tests present in the implementation of CLS-CAD, which are part of the Continuous Integration (CI) pipeline that runs on any changes, the technical correctness of the implementation is guaranteed[2]. Since the CI pipeline builds installers for the add-in as well as a DOCKER image for the backend, setting up CLS-CAD to carry out experiments is easy.

Section 6.2.1 shows how CLS-CAD can be used to explore the design space for a non-trivial example, the design of a robotic arm from modular components. Following the proposed methodology allows the design space to be explored efficiently. By narrowing down the results iteratively, the 256 robotic arms of the correct DOF are reduced to three candidates[3] [34]. The same workflow can be applied to find designs for robotic arms of less conventional structure; for instance, in some cases, redundant joints[4] may be desired to allow robotic arms to reach around obstacles. Carrying out the design space exploration is fast; less than ten minutes are needed to go through all iterations detailed in Section 6.2.1. This can lead to increased motivation to try out different design choices, as they only require an additional synthesis request and

[1] The time that CLS-CAD requires to complete an assembly is monotonically rising in the number of required parts.

[2] The tests provide complete line coverage.

[3] This number differs from the 82 initially stated in Section 6.2.1 due to the "accidental" requesting of robotic arms with 5 DOF.

[4] Joints with parallel but non-identical axes of rotation.

an assembly creation time of less than a minute for the given dataset[1]. Also, Section 6.2.1 shows that going through the proposed iterative workflow helps discover errors, e.g., the wrong DOF being used. These errors also happen while manually creating designs, especially if their structure is repetitive. As CLS-CAD eliminates the tedium of assembly creation, this leaves the designer with more cognitive headroom to notice errors. Summarizing, CLS-CAD allows efficient exploration of large design spaces by using the proposed workflow, quickly converging to solutions that satisfy given constraints. Once a project has been set up correctly, it can lead to a less error-prone design process.

Section 6.2.2 gives a performance benchmark of CLS-CAD, with some caveats. It shows that the efficiency gains possible by using CLS-CAD can be significant, roughly allowing for an acceleration of the design process by two orders of magnitude. Based on the author's experience and prior experiments with the dataset for synthesizing drones, this makes using CLS-CAD worthwhile as soon as more than a single design needs to be created. The time it takes to create taxonomies and apply annotations to parts is usually under an hour. Based on the measured times taken by a human designer to create these designs, there are cases where even for a single design CLS-CAD will outperform the manual design process. Thus, for any design process that is likely to go through more than one iteration, the use of CLS-CAD should be considered. There are several caveats. Firstly, presently CLS-CAD can only support open kinematic chains. This means that there are use cases where CLS-CAD simply cannot be used; for instance, engineering a Stewart platform or a delta picker[2]. Secondly, the performance benchmark is not comprehensive. Benchmarking many different use cases to examine if the gained efficiency exhibits high variance, averaging out the effects of different working speeds of CAD designers, and gathering data on the perceived usability/barrier to entry of CLS-CAD by many different designers would allow for a more precise assessment of when CLS-CAD should be utilized. Obtaining this data is difficult, as this experiment took a single designer around five and a half hours. Investing multiple designers' time on such a scale is prohibitive to companies. However, the efficiency gained is so significant that even if reducing expected efficiency gains by an order of magnitude, there is still significant merit to utilizing CLS-CAD.

[1] For designs that have less than 200 parts on a typical CAD workstation.
[2] These are common robotic systems used in factory automation.

CHAPTER 7

Summary and Outlook

Within this part of the thesis, the methodology behind Cls-Cad, an example implementation for Fusion 360 of Cls-Cad, as well as steps to validate the efficacy of Cls-Cad were presented. In the following, each of these aspects is briefly summarized, followed by an outlook detailing further improvements and their dependencies.

7.1 Summary

The methodology behind Cls-Cad consists of a proposed architecture, a recommended workflow, guidelines for creating taxonomies and annotating parts, and rules by which to generate repositories suitable for performing combinatory logic synthesis based on this information. The proposed architecture is designed to make the implementation of Cls-Cad independent of specific Cad software, separating code that interfaces with pre-existing Apis from the remainder.

Section 4.1 gives an overview of the elements required to implement different layers of the proposed architecture. Recommendations on how to reuse user interfaces are provided as well, alongside approaches to leverage industry-standard file formats for Cad designs to mark connection points. Section 4.3 presents a workflow with a waterfall-model-like structure for performing design space exploration, showing how to quickly narrow down synthesized designs to those that fulfill use-case-specific constraints. Advice and examples of how to interpret and translate typical artifacts of mechanical engineering and product design, i.e., technical drawings, into taxonomies and intersection types for annotating Cad files with information about how they connect

to each other are presented. A bottom-up approach based on repeatedly grouping parts and documenting mating interfaces is introduced. Sections 4.4 and 4.5 give two different rules by which repositories supporting discrete numerical constraints can be computed from the annotated CAD files. Use-case-specific side effects of these repositories and how to avoid them are then discussed in Section 5.2.

Section 5.1 discusses the structure of an implementation of CLS-CAD as an add-in for FUSION 360, which enables CLS-CAD to be used by designers in practice. Examples of how the proposed architecture and design for reusability translate into the add-in are given, showing how the CEF is used and providing explanations of how the add-in may be extended. The user interface of the add-in is shown, and an explanation of how core elements of the developed user interface are used and how they correspond to the elements required by the add-in layer of the architecture is provided. Section 5.1 also explains performance considerations that utilize features present in many CAD kernels. Their application to the assembly algorithm is discussed in detail to show how large speed-ups of the implementation can be achieved and translated to other CAD software. Section 5.2 shows how the generation of repositories works on a technical level, explaining how the raw output of the combinatory logic synthesis can be translated into human-readable instruction sets. It is also explained how translation into sets of instructions is leveraged to keep the implementation of add-ins for CAD software simpler.

Chapter 6 introduces two datasets for the experimental validation of CLS-CAD. Experiments assessing how well the workflow for using CLS-CAD works in a non-trivial example are carried out. A performance benchmark of CLS-CAD versus a human designer is performed, concluding that CLS-CAD is roughly one hundred times faster than a human assembling designs. In summary, CLS-CAD achieves its goal of allowing designers to focus on their knowledge and expertise regarding which designs will perform well and eliminates the tedium of operating CAD software. It allows for efficient exploration of large design spaces by iteratively homing in on the subspace containing designs of relevance. The gained efficiency allows designers to spend more of their time evaluating designs and eliminating errors, potentially leading to better quality control.

CLS-CAD is a powerful tool to increase automation in the CAD design process and make it more robust. By allowing designers to work in a performance- and metric-based fashion and enabling more design iterations in less time, the design space can be explored more thoroughly due to the lower opportunity cost of doing so. Additionally, designs become more tailored to specific use cases, easier to adapt, and more resilient to supply chain shortages, as a redesign with different parts that may have other interdependencies can be

carried out faster than previously possible. Furthermore, Cls-Cad allows for the decoupling of designing individual parts from creating Cad assemblies, providing another avenue for increasing efficiency.

7.2 Outlook

Cls-Cad is a powerful tool to improve the automation of Cad design processes. Nevertheless, there are several areas of ongoing research that can lead to further improvement. One of the key limiting factors of Cls-Cad, and more generally, Cls, is that outputting tree grammars restricts the output to trees. For many use cases, this restriction can be worked around. Using domain-specific information about junction points, solutions that post-process resulting terms into graphs are often possible. However, as Cls-Cad is intended to work for arbitrary sets of parts and the specific use case is not known a priori, such solutions are not possible. To remedy this issue, future research can be conducted into developing a form of combinatory logic synthesis that can output directed acyclic graphs, permitting closed kinematic chains to be represented.

Cls-Cad does not make use of the parametric design features of most Cad software. While it does not restrict them, meaning that changing parameters for synthesized designs leads to the geometry of parameterized designs being updated, there is untapped potential. Exposing the parameters of parametric designs to the synthesis and design space exploration processes may enable more expressive specification mechanisms while at the same time permitting variance not possible through simply connecting modular components. However, this only makes sense in settings where the parametrically designed parts can be cost-efficiently manufactured.

Cls-Cad's performance in comparison to a human designer makes a compelling case for utilizing it whenever possible, i.e., the target design can be represented by a tree. However, the need to initially set up taxonomies and annotate parts remains. While the setup phase should not be fully automated, as this could lead to over-reliance on the corresponding tools and thus result in new errors being introduced, partial automation is desirable. With Cad datasets for machine learning being available, the development of a recommendation system that accelerates the setup procedure by proposing defaults is possible [91, 126, 133]. Specifically, the approach by Jones et al. seems promising, as it scores and ranks a series of potential JointOrigins/Joints [72]. This could be used to recommend locations that need to be annotated. A similar consideration can be applied to the actual synthesis procedure. While in Section 2.1.1 it is argued that Ai may do more harm than good when gen-

erating designs, as it cannot provide a guarantee of one hundred percent accuracy, which, in conjunction with tool over-reliance, may lead to errors going unnoticed, an AI-based approach could be used to generate initial approximate designs. These could be used to help understand the design space and similarly accelerate the setup procedure.

CLS-CAD's approximate performance, as well as its applicability, has been validated. Significant effort has been invested in creating high-quality user interfaces and achieving good usability, setting it apart from many other open-source projects. However, without input from a large number of industry professionals, who provide feedback on which features need to be introduced or enhanced for them to consider adopting the tool, it is difficult to ascertain further areas of improvement. This goes hand in hand with comprehensive validation of CLS-CAD by industry professionals. Future research should attempt to secure an industry partner and carry out trials on a larger scale. Even ten designers providing data on their usage of and efficiency with CLS-CAD would help identify shortcomings, assess gainable efficiency more thoroughly, and identify which aspects of design use cases lead to them being particularly well- or ill-suited for automation.

Part III

Opti-Clamp

Introduction

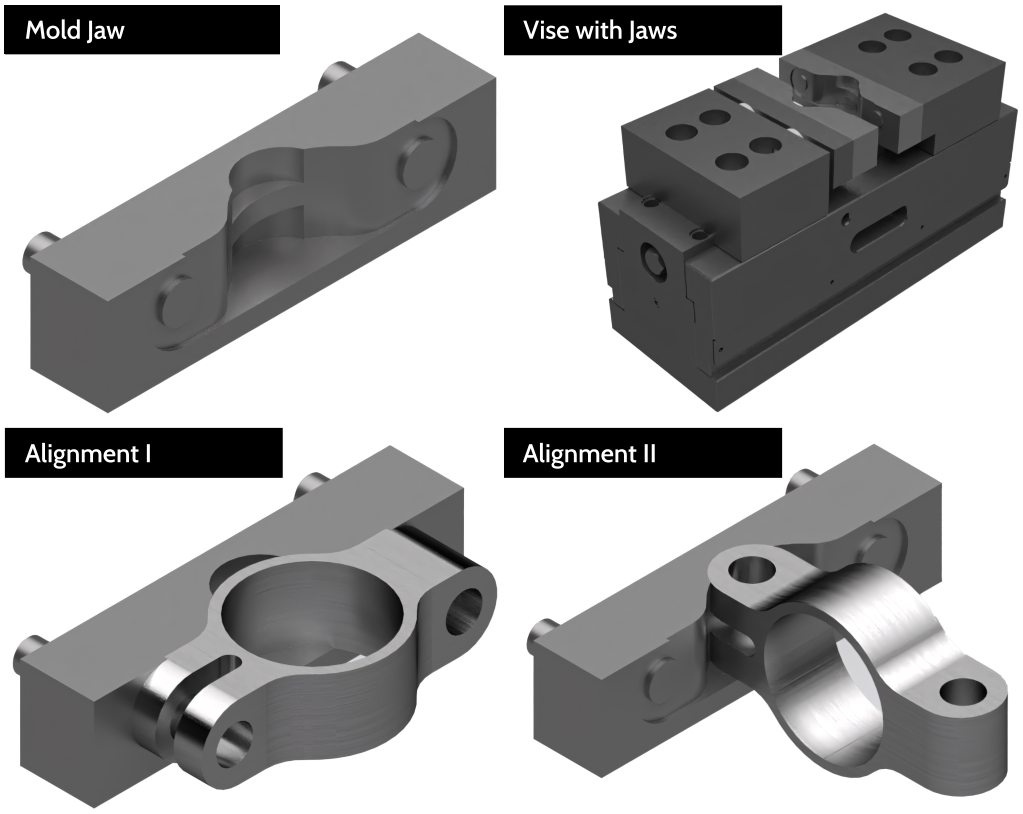
Clamping parts with complex geometries is challenging. Oftentimes, there are no parallel planar surfaces, and sometimes there are no planar surfaces at all. This rules out simple clamping solutions using vises[1]. Instead, more complex clamping solutions, such as the one shown in Figure 2.2 need to be utilized. These necessitate a skilled worker and exhibit either poor repeatability due to the lack of clear reference points for the clamps or require an additional probing step before every milling operation to compensate. Additionally, care must be taken not to deform or break thin features on the parts[2]. Furthermore, the surface quality achievable by the milling operation depends on the rigidity of the clamping solution. Complex clamping solutions, as pictured in Figure 2.2 are generally less rigid than a vise. Finally, the dimensional accuracy of important features is reduced if the clamping solution deforms the part. Complex clamping solutions can require more force to secure the part, increasing overall deformation.

In comparison, using a vise to clamp a part is preferable in all the previously mentioned aspects. Vises are easy to operate and very rigid. Their repeatability is high; the vise itself and one of its jaws do not move, meaning no repeated probing is necessary. From a technological point of view, vises offer the best overall properties. Figure 8.1 depicts a set of mold jaws, as well as a vise utilizing these jaws. Mold jaws enable vises to be used to clamp complex parts, replacing the need for a skilled machinist to individually clamp these parts with the need to design and manufacture suitable mold jaws. Mold jaws conform to the shape of a part, allowing it to be recessed into the jaw, eliminating any risk of slippage. Conforming to the part's shape evenly distributes the clamping force across the part's surface, reducing the chance that thin features will deform. Depending on the part's geometry, the

[1] The clamping force that a vise applies is along a single vector, commonly orthogonal to the jaws. This means that if there are no parallel planar surfaces, the clamping force is applied at an angle and is likely to cause the part to slip.

[2] Commonly, parts that have been topology-optimized are manufactured through LPBF. These often possess thin features, which may be optimal for the intended use of the part but easily deform if force is applied in a way that the part is not optimized around.

Figure 8.1: Example of mold jaws enabling usage of vise for complex part



necessary clamping force is also reduced or eliminated, as the part can be fully constrained by the geometry of the mold jaws[1]. Figure 8.1 also shows that these mold jaws do not need to be specific to a single alignment of the part during post-processing, if suitably designed. Each pictured mold jaw can accommodate the part in two different alignments with respect to the cutting tool. Once suitable mold jaws are in place, any worker can set up the post-processing operation for a complex LPBF-manufactured part by inserting the part into the jaws and tightening the vise. The job can then be run without individual probing. This saves time when batches of the same part need to be post-processed. However, designing mold jaws costs time. Additionally, it is not clear which rotation of the part along each respective alignment's upright axis leads to the best possible clamping performance. Determining this requires several iterations of designing and simulating the performance of such mold jaws in FEA software. This still does not guarantee that the

[1] In practice, the clamping force can never be reduced to zero due to manufacturing tolerances of the part and the corresponding mold jaw.

best-performing mold jaw, determined this way, is close to the true optimum, especially if the part behaves non-linearly. As a result, the effort of designing mold jaws is commonly viewed as unmerited.

CHAPTER 8

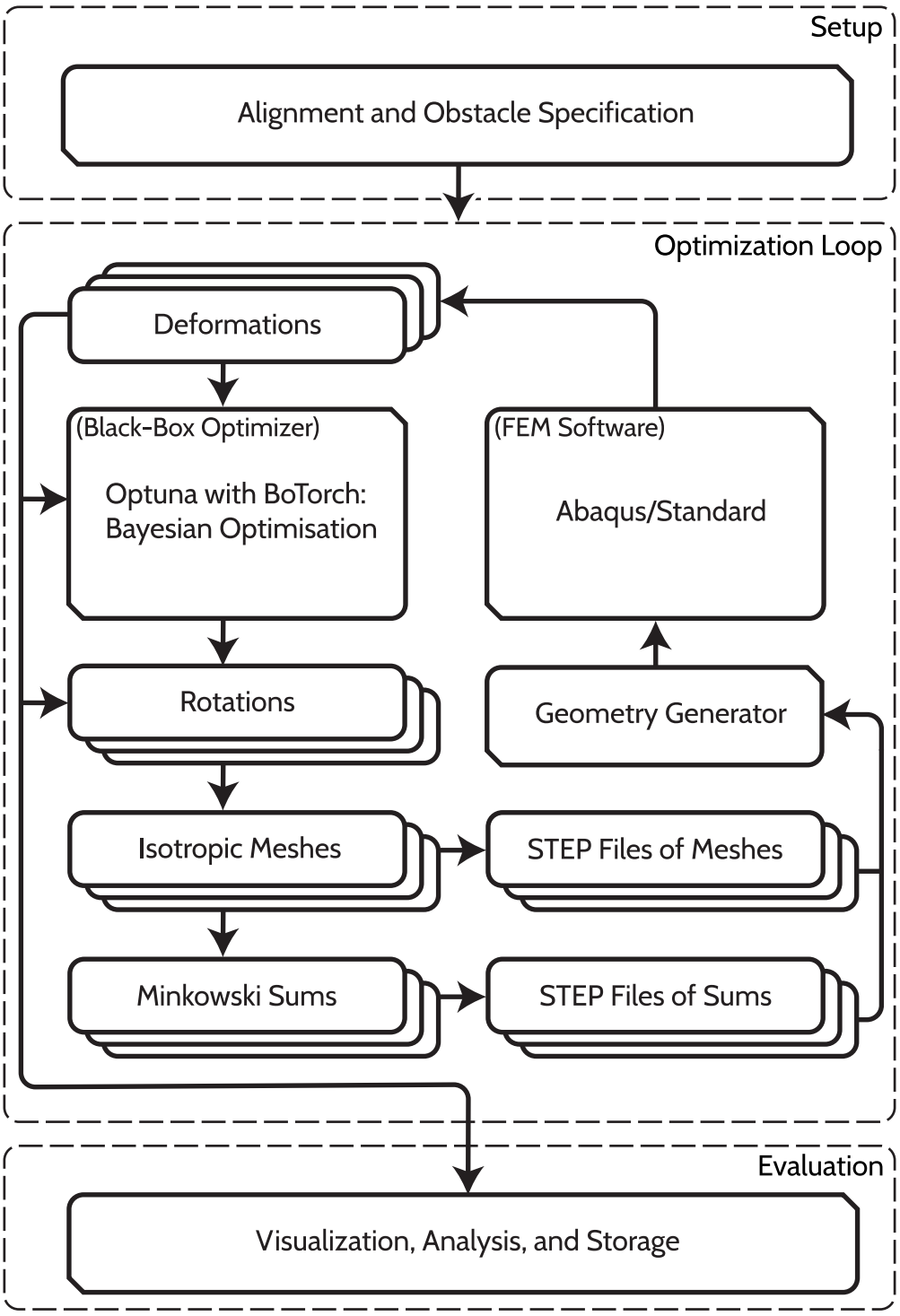
Methodology

In this chapter, a methodology that permits automatically generating and identifying Pareto-optimal form-fitting mold jaws is introduced. The resulting mold jaws allow clamping additively manufactured parts in multiple orientations so that the clamping force-induced deformation is Pareto optimal w.r.t. the orientations. The automatic generation, in conjunction with determining Pareto-optimal solutions, eliminates several downsides of mold jaws, e.g., their creation being time- and labor-intensive, by allowing them to be designed asynchronously. The proposed methodology requires significant execution time, as many FE simulations need to be performed[1]. This is non-problematic, as there is usually a time gap between when a job is received and the physical execution of it on a workshop's machines begins. The proposed methodology can be executed during this time gap.

Chapter Organization The remainder of this chapter begins by describing the overall process of the methodology. Then, the three individual blocks of the process are described in detail. First, the setup phase is covered, explaining the concept of alignments and obstacles. Following this, the individual steps of the optimization loop are covered, explaining the necessity of each step. Finally, the evaluation phase is described, and pointers are given on how to interpret results.

[1] These are very time-intensive, requiring upwards of five minutes even for coarse simulations, and do not scale well with parallelization in this use case.

Figure 8.2: Overview of methodology to generate Pareto-optimal mold jaws



8.1 Process Overview

In this section, a general overview of the steps required by the proposed methodology to automatically generate Pareto-optimal form-fitting mold jaws for arbitrary part geometries is given. The general idea is to automatically compute mold jaws that fit multiple alignments of a part, with a different rotation for each alignment. For these mold jaws, a simulation is then programmatically created and executed in FEA software to obtain the resulting deformations of each alignment. These two steps are encapsulated in a black-box optimization loop to search for Pareto-optimal solutions. Figure 8.2 gives an overview of the steps this entails. To begin the process, the optimization loop needs to be initialized with the geometries of the different alignments as well as their corresponding obstacle geometries. The reason for these being separate files is given in Section 8.2. Obstacle geometries are used in the same fashion as for topology optimization, ensuring that the resulting mold jaws account for access by the cutting tool. Isotropic remeshing is performed on all input geometries to ensure that the mesh topology will be suitable for FE simulation. Details regarding the necessity of remeshing are given in Section 8.2.

After the optimization has been initialized with the necessary meshes, the optimization loop begins. The loop consists of three main parts:

- Generating artifacts needed to perform FE analysis. This consists of rotating isotropic meshes, computing their Minkowski sums with the clamping force vector of the vise, converting meshes into faceted BREP geometry, and finally performing translations and boolean operations on the BREP geometry.
- Loading generated artifacts into ABAQUS, performing the FE analysis, and extracting the resulting deformations.
- Inputting extracted deformations into an optimizer, refitting the underlying model[1], and outputting new candidate rotations based on the fitted model.

Section 8.2 and Section 8.3 provide details that intuit why omitting any of these steps is difficult. Regarding the overall runtime of the proposed methodology, the FE simulations are the main driver. While they can be sped up to some degree, parallelization is only effective for meshes with high resolution.

[1] Bayesian optimization based on Gaussian processes.

However, meshes with high resolution significantly increase the runtime. Considerations regarding performance optimizations are outlined in Section 9.2, and ideas to majorly increase performance are discussed in Section 11.2.

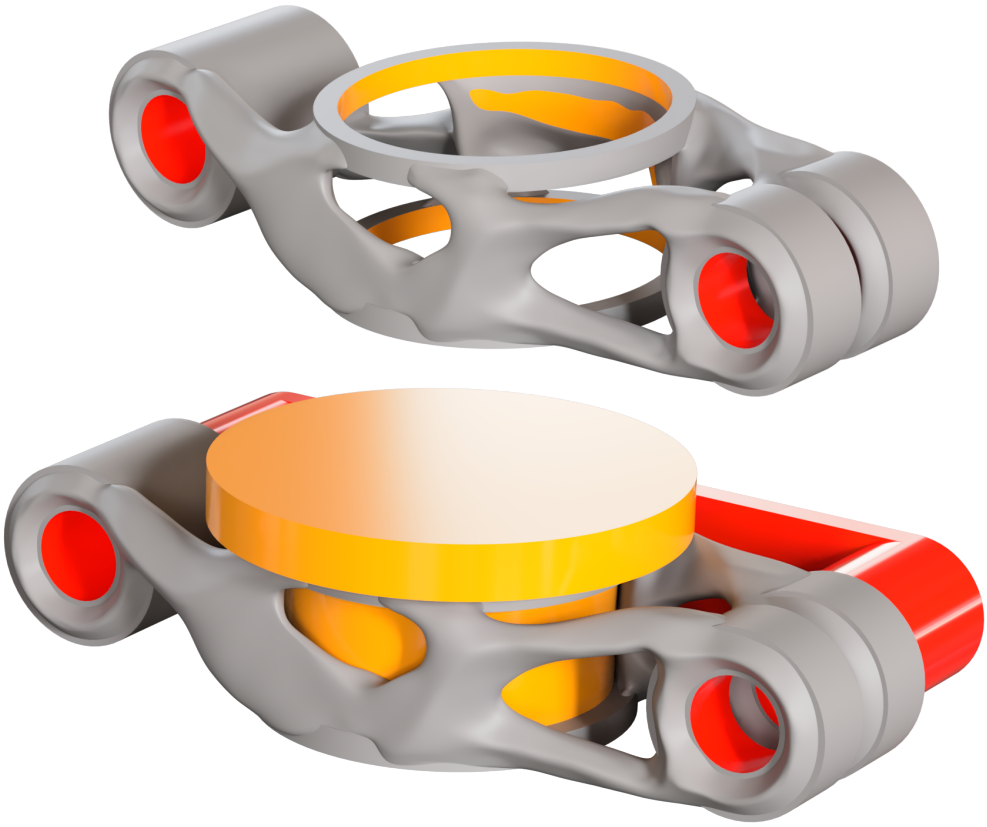
During the execution of the loop, created artifacts and data are constantly aggregated and stored. The data is visualized, and there are analysis tools available while the black-box optimization loop runs. As such, there is no set termination criteria for the optimization loop. It can be executed either for as long as possible, e.g., the maximum amount of time until the job has to physically start being prepared and machined, or until the results are deemed conclusive based on the analysis tools. This means that the evaluation phase is carried out in parallel to the optimization loop.

8.2 Setup Phase

In the setup phase, the user of Opti-Clamp needs to define the fixed parameters and geometries that serve as inputs to the optimization loop. While the development of Guis is not as much a focus of Opti-Clamp as it is for Cls-Cad, there should be interfaces to guide users through the setup. It is important to keep in mind that users are likely technicians or designers, which rules out console applications.

The first inputs to be set by the user are pairs of part and obstacle geometry. Figure 8.3 gives an example of how these pairs look. The surfaces colored orange in the top image are those that need to be machined as part of the first alignment, while those colored red belong to the second alignment. It can be seen that these require the part to be rotated by 90 degrees relative to the cutting tool's axis of rotation in between the orange and red surfaces being processed; hence, they form two different alignments. The bottom image shows the same part with the same two alignments but also includes the corresponding obstacle geometry. The obstacle geometry is used during the creation of the mold jaws to ensure that the cutting tool can access the surfaces. While computing the Minkowski sum of part geometry and the clamping force vector guarantees that the part is not occluded by other geometry with respect to the direction of the clamping force vector, it does not guarantee that there are no occlusions in the direction of the cutting tool's axis of rotation. Such occlusions may not be problematic, as pilot holes could be drilled; the cutting tool could then mill away the occluding parts of the mold jaw during post-processing. However, this requires additional machining, leading to an increased amount of material that needs to be cut. Also, depending on which material is used for mold jaws and whether or not they are hardened, this leads to a reduction in feed speeds, increased chip load, or

Figure 8.3: Example of two alignments and corresponding obstacle geometry



also affects the lifespan of the cutting tool. As such, explicitly encouraging and accommodating the usage of obstacle geometry as part of the methodology is desirable. The obstacle geometries seen in the bottom image in Figure 8.3 illustrate how to achieve this. The orange obstacle geometry ensures that the inside of the large center hole is free of mold jaw geometry, as well as ensuring that the space above the center hole is accessible, allowing it to be milled flat. The obstacle geometry may be omitted if the geometry of the part is such that there can be no occlusions with respect to the tool axis. The red obstacle geometry is an example of when this may be the case, only ensuring that the space above the holes marked in red is accessible. The holes themselves are fully enclosed; when projecting their geometry onto a plane orthogonal to the clamping force vector, they form a simple polygon[1]. This indicates that the

[1] As the geometry is input as polyhedral meshes, the projection always forms a polygon. If the polygon is simple, then the projected feature is “safe”; if it is a polygon with holes, it may need obstacle geometry.

inside of those holes does not require obstacle geometry.

The reason for each alignment consisting of separate polyhedral meshes instead of transforming one mesh accordingly is that the geometry may change between post-processing steps. This is negligible if surfaces are milled to be within tolerance, where very little material is removed; however, the LPBF printing process may require support structures. These supports can be thought of as scaffolding towers that hold up overhanging surfaces during printing, which would otherwise deform due to sagging. As the support structures are difficult to remove by hand, i.e., with pliers, they are sometimes cut from the part during post-processing. In this case, each milling operation on the respective alignments causes a large change in geometry. To account for this, specifying a mesh for each alignment is necessary.

Typically, when using FEA software, the CAD file is loaded, and the BREP geometry is isotropically meshed by the software. Polyhedral meshes generated directly from CAD software have surface topology unsuitable for FE analysis. They exhibit polyhedrons with high aspect ratios, i.e., one side being far longer than the other side. This reduces the information gained by performing FE analysis, and such ill-conditioned elements with short edges impact numerical stability[1] [117]. This is due to FE analysis only yielding data for every vertex of the resulting polyhedral mesh, so polyhedrons with high aspect ratios lead to large sections of the part where there are no nodes. Generally speaking, meshes with a high resolution and good isotropy yield results that more accurately reflect the real behavior of the part under load. For the proposed methodology, it is not possible to load the original CAD file and let the FEA software mesh it. This is due to the combination of two factors:

- Computing the Minkowski sum for general BREP geometry is an unsolved problem. The problem is already difficult for polyhedral meshes and has been the focus of research [23]. Most pieces of CAD software do not offer tools for sweeping solid bodies; the few that provide such tools (SOLIDWORKS, FUSION 360) either restrict the swept geometry to not have any concave faces or fail for more complex geometries without explanation.

- Computing only the mold jaws by using polyhedral meshes and then loading original CAD files is problematic. Since polyhedral meshes only approximate the BREP geometry with planar surfaces, vertices that the FEA software generates during meshing are offset from the original surface. This leads to a discrepancy between the mold jaws and the

[1] FEA software, e.g., ABAQUS, has options to identify and eliminate such polyhedrons for this reason.

part[1]. This can lead to issues like clipping, gaps, and overclosures. The FE analysis no longer reflects reality and is usually not numerically stable, leading to it failing to solve.

The proposed solution for these issues is to perform isotropic remeshing on the input and obstacle geometries, closely mimicking the meshing that FEA software performs. By later converting the remeshed input geometry into faceted BREP geometry, loading this into the FEA software, and meshing it so that the generated mesh matches the facets, the Minkowski sum can be computed and the FE analysis carried out. Achievable fidelity is comparable to loading the original CAD file, as the isotropic remeshing implemented by OPTI-CLAMP recreates most options provided by FEA software[2].

Finally, the user is prompted for numerical input values, such as how deep each alignment of the part is supposed to penetrate into the mold jaws or the initial rotation of each alignment around the tool axis.

8.3 Optimization Loop

After input data is collected and processed as detailed in the previous section, the main optimization loop is entered. The first iteration begins at the "Rotations" step, as no deformations have yet been computed. The initial set of rotations, one for each alignment, can be randomly picked. However, it is common to generate these by means of a Sobol sequence for the first few iterations[3] [10]. These rotations are then applied to the isotropic meshes of all input geometries, rotating them along their Z-axis[4]. The polyhedral meshes are then converted to BREP CAD geometry. The CAD geometry is faceted, meaning that it consists entirely of BREP planar surfaces, with each surface matching a polygon of the surface of the polyhedral mesh. The Minkowski sum of the rotated polyhedral meshes and a linear vector parallel to the clamping force of the vise is then computed[5]. A simple boolean cut with a normal clamping jaw is not sufficient to create a mold jaw. Computing the Minkowski sum is needed to ensure that the part can be inserted into the later mold jaw, as there may be undercut features present, i.e., dovetails. The vector's length is twice the desired penetration depth; the maximum depth of the features of

[1] Essentially, the mold jaws are no longer mold jaws.

[2] The FEA software itself can also be used to create the meshes of the input geometry. This is discussed specifically for ABAQUS in Chapter 11.

[3] Many popular black-box optimization frameworks like AX or OPTUNA recommend generating initial samples for optimization by Sobol sequence.

[4] The input geometries' Z-axis corresponds to the cutting tool's axis of rotation by convention.

[5] This vector is usually orthogonal to the face of the clamping jaws.

the mold jaw that conform to the part. The resulting polyhedral mesh of the Minkowski sum is then treated identically, being converted to faceted CAD geometry. The "Geometry Generator" then uses these faceted CAD files to create the necessary geometry for the FE simulation. This entails post-processing the CAD files to remove defects occurring as a consequence of the conversion process (Section 9.1), carrying out a boolean cut between the Minkowski sum and the normal clamping jaws to create mold jaws, and then translating the mold jaws to their correct positions. The "Geometry Generator" then outputs the final geometry for the FE simulation as well as a script for the de facto industry-standard FEA software ABAQUS. Executing the script loads the geometry, sets up the simulation in ABAQUS, executes it, and evaluates the results. This procedure is carried out for each alignment, resulting in values for the deformation of each alignment of the part given a fixed clamping force. These values are then fed into the black-box optimizer, resulting in an optimization loop.

The main factor for the runtime of the proposed methodology is the FE simulations. The feasible set of solutions is simple to evaluate, forming a hyper-cube[1], but the resulting deformations do not clearly correlate with them, i.e., linearly or monotonically. There is no clear analytical derivative of the deformations in this use case, which would need to be determined numerically, e.g., by FE simulations. Additionally, each sample is expensive to evaluate, requiring more than five minutes for a single sample. These observations fulfill the criteria given by Frazier for when Bayesian optimization is the correct approach [55]. Bayesian optimization is more sample-efficient than other optimization methods [10]. While Bayesian optimization requires more computation/fitting time than other optimization methods, this is significantly outweighed by the time saved by running fewer simulations. If parallelization is not an issue, i.e., if a compute cluster is available and there are enough cores to run many FE simulations in parallel, then instead of Bayesian optimization, evolutionary algorithms like CMA-ES can be used[2] [97].

An interesting effect that occurs with this methodology is that the interdependencies of multiple alignments with respect to generating the mold jaws do not need to be taken into account. When generating the mold jaws, the boolean cut to accommodate the second alignment can remove geometry conforming to the first alignment[3]. Since this effect is dependent on the rotations

[1] Assuming permitted values for rotations are all angles within equal intervals; otherwise, a hyper-rectangle.

[2] The paper by Ozaki and Nomura is written in Japanese; translate and consult Table 5 "Guidelines for selecting optimization methods" within the paper.

[3] This effect can be somewhat mitigated by using different penetration depths for each alignment.

output by the optimizer and, in turn, affects the results of FE simulations and thus the deformations input to the optimizer, these interdependencies are learned and accounted for by the underlying model during the optimization loop.

8.4 Evaluation Phase

The evaluation phase is carried out in parallel with the optimization phase. Different data visualizations are computed for each iteration of the optimization. These visualizations consist of:

- A scatter plot showing the Pareto front in conjunction with the other samples.
- A scatter plot showing the value of each alignment's deformation in each iteration, with a monotonically decreasing function illustrating the optimization progress.
- A line plot of the empirical density function for each alignment's deformations, showing the cumulative probability of each deformation being smaller than a given value.
- A bar plot showing the hyperparameter importance, in this case, the degree to which each rotation affects the deformation of individual alignments.
- A contour plot showing expected deformations for each alignment based on previous samples.

Out of these plots, the first three are useful for judging when to stop the optimization loop, while the other two are of less interest to users and more of academic interest. By considering the two scatter plots, a technician can determine if a Pareto-optimal solution has already been found that meets the given technological requirements and if it is likely that meaningful improvement is still possible. For instance, if some of the alignments have specified tolerances of 0.1 mm in technical drawings, there is no need to continue the optimization loop if the Pareto front already contains a point with significantly lower deformation than this for those alignments. By considering the second scatter plot, if the monotonically decreasing function fitted to the minimum values of deformations has already decreased by an order of magnitude and no progress has been made for several samples, the technician can judge that it is unlikely for further iterations to yield much improvement. The line plot of the EDF aids the technician in a similar manner, interacting with the logistics

of the workshop. If there is a lot of lead time on an order, the technician could decide that potentially identifying a point that exhibits ten percent lower deformation of some alignments is worth pursuing; if the lead time is short, the optimization loop may be stopped even if there is still a lot of headroom for optimization.

The hyperparameter importance bar plot can be used to discern if finding mold jaws for given parts and alignments is technologically feasible. Ideally, each alignment's rotation primarily affects the value of its own deformation. If the bar plot shows that there is a high interdependence of the alignments, meaning each alignment removes geometry from the mold jaw needed for other alignments, then the penetration depths need to be adjusted[1]. The contour plot can be used to explore the design space past the sampled points, based on the surrogate model that the Bayesian optimization creates. This is of less interest for this use case, as explored points would need to be FE-simulated regardless, considering that it would be negligent to manufacture mold jaws based solely on a surrogate model. This is due to the cost of manufacturing these in conjunction with the design space behaving non-linearly, resulting in a risk that chosen designs may lead to unexpectedly high deformation unless actually simulated. It is also recommended that the Pareto-optimal mold jaws picked by a technician be FE-simulated an additional time with increased resolution of the meshes for precisely those reasons. Potential improvements in this aspect are discussed in Section 11.2.

[1] Or the part is not suitable for multiple alignments, forcing the use of multiple mold jaws.

CHAPTER 9

Implementation Details

This chapter documents how the methodology translates into an implementation. While the methodology could be summarized as a typical black-box optimization loop, much of the challenge and contribution stems from the implementation details. To carry out black-box optimization, the black box needs to accurately and consistently reflect the real-world scenario. This means that all of the steps outlined in the previous chapter need to be error-free, i.e., any chance that the generation of STEP files representing mold jaws results in erroneous geometry needs to be eliminated. This is difficult, as CAD kernels and FEA software are quintessentially black boxes in and of themselves. A good understanding of the respective APIs is necessary to achieve this. Several choices of framework and software are made when implementing the methodology, which are briefly outlined in the following.

Software and Frameworks There are several options for FEA software, which is an integral part of the black-box function. A key consideration here is open-source versus closed-source. A prominent open-source example is CALCULIX [40, 41], which is free of charge and enjoys widespread community support, allowing integration with a number of other frameworks[1]. On the other hand, there are several closed-source options available, which see widespread use in industry. The most well-known of these are ABAQUS by Dassault Systèmes and the offerings from ANSYS. These options can be assumed to be more robust than their open-source counterparts due to being commercial products used by many users for long periods of time. There are no significant

[1] See `https://github.com/calculix`.

advantages or disadvantages to any of the options with respect to the methodology. Required features to simulate a clamping situation are basic; virtually all FEA software possesses them. Automatically searching for potential contact pairs[1] is available in most FEA software as well. As such, the choice should be made based on familiarity and budget constraints, erring toward open-source, as this allows the implementation to make modifications to the FEA software. This offers more flexibility for future changes to the methodology. For the presented implementation, ABAQUS is chosen due to the author's familiarity with the software and its widespread use. One noteworthy aspect of ABAQUS is that its operation can be fully automated by means of a PYTHON scripting interface. Operations carried out through the user interface can be executed by a matching function call in the script.

There are also several candidates for the choice of black-box optimization framework. The choice depends mainly on which Bayesian optimization methods are implemented, how well the framework is maintained, the visualization options, ease of use, and how much customization the framework allows. Options include AX [9], OPTUNA [3], OPENBOX [71, 82], as well as HYPERMAPPER[2] [94]. Most major frameworks are similar in terms of their feature sets. OPENBOX has the most comprehensive set of optimization algorithms implemented and includes several non-Bayesian options. OPTUNA is easy to set up, integrates with BOTORCH [10], and offers live visualization tools. AX is built upon BOTORCH and is actively maintained by META. It has many pre-implemented useful visualizations; however, many are undocumented. In general, it is feature-complete and allows access to cutting-edge optimization methods, but these methods' documentation is often lacking or difficult to find. Being actively maintained helps in this regard; questions and issues are promptly answered and resolved. For the implementation of OPTI-CLAMP, initially, OPENBOX is chosen. As the use case of finishing the optimization procedure "just-in-time" was discovered, making trade-offs between further optimization progress and the time until production needs to start, as described in Section 8.4, the framework is switched to OPTUNA, using the BOTORCH integration to allow for more customization options. The actual optimization is carried out by using a Gaussian process as a surrogate model and using QNEHVI [38] as the acquisition function. This is in line with the recommendations given by Ozaki and Nomura, as the solution space is low-dimensional and continuous [97], and it can be assumed that the available resources for parallelization are limited[3]. Switching to using AX may be an

[1] Surfaces that may come into contact during the simulation.

[2] At time of writing, the framework seems no longer maintained, and will be disregarded in the following.

[3] CNC shops typically do not have on-site high-performance computing clusters.

option in the future, as it would allow a more comprehensive set of Bayesian optimization methods to be tested, and most of the features that Optuna offers can be recreated due to the Api of Ax being flexible.

Finally, a choice of framework for manipulating Cad data to generate the mold jaws needs to be made. There are two possible approaches: Cad software can be instrumented and used by means of their Api to carry out the necessary operations, or software frameworks that allow direct programmatic usage of a Cad kernel can be used. The first option adds overhead to the actual optimization loop, requiring the Cad software to be running. Additionally, this limits the features of the Cad kernel that can be used. Most kernels are full-featured, but many of their advanced features are not exposed in Cad software. As such, the second option is chosen. This limits the choice of kernel to OpenCascade, being the only actively maintained open-source Cad kernel [95]. Bindings to use OpenCasdade in other programming languages are available; for using Python there are two maintained options. Either Ocp, which provides bindings as part of the overarching project CadQuery can be used [98, 99], or the pyOcct project [78]. As both of these provide auto-generated bindings to OpenCascade, their Apis are close to identical. For Opti-Clamp, Ocp is used by loading CadQuery, as CadQuery provides higher-level functions that wrap common tasks and calls to OpenCascade, while not losing any of the flexibility when directly interacting with the Cad kernel.

Chapter Organization The remainder of this chapter provides documentation about how selected individual steps of the methodology are implemented. The focus is on those steps that are technically challenging. The implementation underwent several iterations, and there are non-obvious pitfalls. The sections describe lessons learned during the process, aiming to enable future work. In the following, first, the generation of the Step files to be used with the Fea software is covered in detail. Then, the development of the script utilizing the Api of Abaqus is detailed. Finally, an approach for handling manufacturing tolerances is given.

9.1 Generation of Input STEP Files

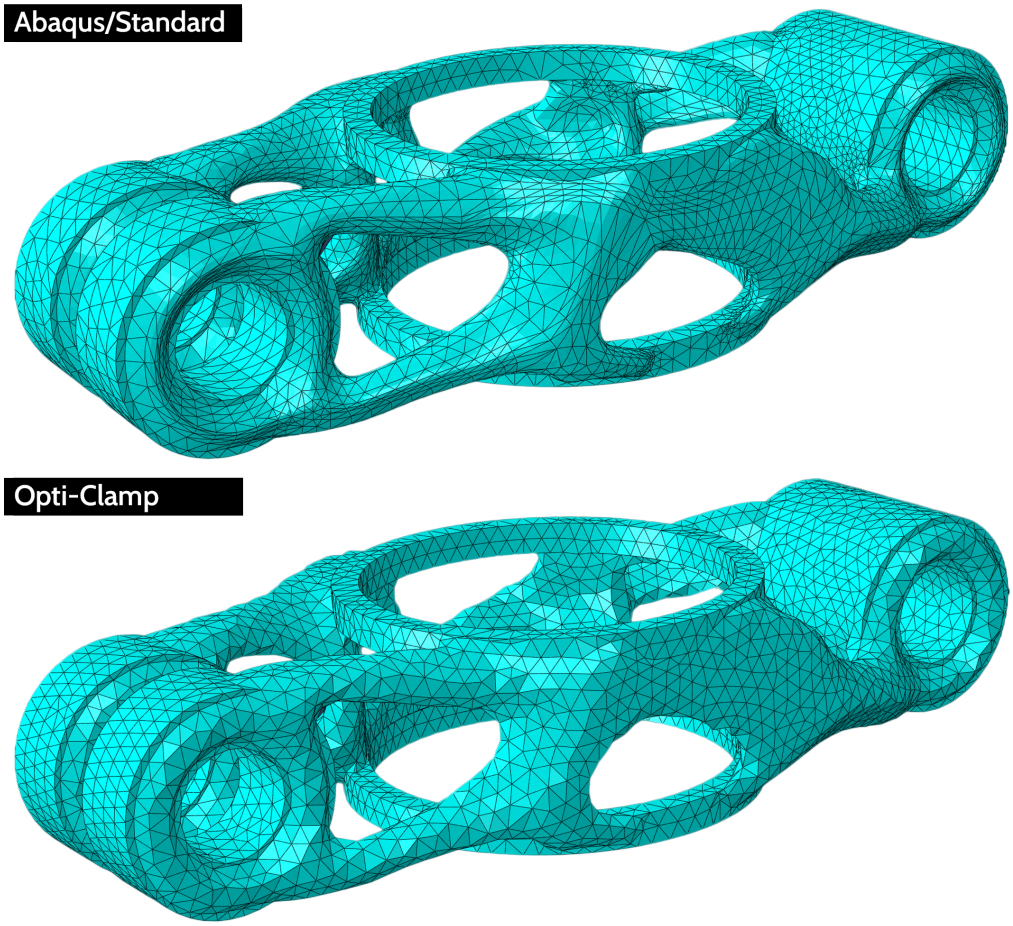
The generation of the STEP files of the faceted original part geometry, as well as the mold jaws, is a pivotal part of the methodology. Ideally, the part geometry should not be faceted, and instead, the Minkowski sum of the original CAD geometry should be used. However, computing the Minkowski sum of arbitrary BREP geometry with a vector or curve is an unsolved problem, to the best of the author's knowledge. This is indicated by the fact that CAD software rarely features tools to sweep solids, and those that do, like FUSION 360 and SOLIDWORKS, only permit this under restrictions[1]. As such, the original part geometry needs to be converted to an STL file, for which the Minkowski sum can then be computed. If the meshes created by the Minkowski sum operation closely match those that would have been produced by meshing the original geometry in ABAQUS, this workaround has negligible impact on the results of FEA, as these only depend on the FE meshes. Isotropic remeshing of triangular meshes has been extensively researched, and many mesh processing libraries include implementations of remeshing algorithms[2]. Figure 9.1 compares the mesh output by ABAQUS on the original geometry with the mesh created by isotropically remeshing the STL output by CAD software. The bottom mesh is computed using the isotropic remeshing algorithm by Pietroni et al. [102] implemented in PYMESHLAB. The mesh densities are calibrated by trial and error to determine how the mesh resolution options in ABAQUS correspond to those in PYMESHLAB. Comparing the two meshes, it can be seen that they are overall very similar[3]. The main difference is that the mesh generated by ABAQUS increases mesh resolution based on the curvature of surfaces, while the mesh used by OPTI-CLAMP is more isotropic. The differences are small enough that they can be assumed to have a negligible impact on the results of the FEA. This can be determined by using a part where the corresponding mold jaw does not require the Minkowski sum to be computed, due to being free of undercuts, and comparing the two different approaches. For parts with very fine structures and extreme changes in curvature, differences may become more noticeable. By using the adaptive isotropic remeshing algorithm by Dunyach et al. [45], the remeshing for OPTI-CLAMP could be changed to

[1] In practice, these commands fail once the geometry consists of anything other than primitives.

[2] Such as the Polygon Mesh Processing Library [118] or PYMESHLAB(`https://github.com/cnr-isti-vclab/PyMeshLab`)

[3] Some of the differences perceivable are due to the original part being smoothly shaded in ABAQUS as compared to the faceted geometry.

Figure 9.1: Comparison of resulting finite element meshes.



match ABAQUS even closer[1]. Before remeshing the input files, common errors in triangle nets that may affect the remeshing operation are repaired. Close vertices are merged together, non-manifold edges and vertices are removed, and faces that have an area of zero are removed by merging them with an adjacent face.

Once the isotropic meshes have been prepared, the Minkowski sum can be computed. This is done by utilizing the `minkowski_sum` function of the library PYMESH [138]. The algorithm used for computing the `minkowski_sum` in PYMESH is from LIBIGL [70], a popular mesh and graphics processing library. The library is unmaintained but is easy to use through a provided DOCKER

[1] Unfortunately, the Polygon Mesh Processing Library does not provide any bindings for Python. As such, a binary that performs only the remeshing and returns the remeshed files would need to be compiled.

image. Using PyMesh through Docker was compared with creating a singular-purpose executable based on Libigl directly. Both approaches load an Stl, apply the Minkowski sum operation, and then export the result. Both the binary and Libigl are compiled in `Release` mode on the Windows 11 operating system, with typical compiler optimizations for improving runtime enabled[1]. The Docker-based approach's runtime is one order of magnitude faster than the compiled Libigl-based binary. This is surprising, as the expected outcome is that a natively compiled binary would outperform virtualization. This may be due to Libigl being optimized for Linux usage. It is often difficult to identify why some libraries run slower on Windows, as it is not clear which dependency of which library contains code that uses sub-optimal operations. As the usage of Abaqus necessitates using Windows, the Docker variant utilizing PyMesh is used for Opti-Clamp. An alternative is compiling a Libigl-based executable in the Subsystem for Linux (WSL2) on Windows. This is likely slightly faster than utilizing Docker, as WSL2 is tightly integrated into Windows and usually outperforms other virtualization options. However, the potential gains are not significant enough to warrant much development effort, as the runtime for computing the Minkowski sums only accounts for a few seconds of the runtime of each execution of the black-box function, which takes upward of five minutes in total. It should be noted that the Minkowski sum operation preserves the pre-existing topology of the mesh, as each triangle is offset along the given vector. The new triangles introduced to the mesh all have normals that are orthogonal to the direction of the clamping force. As such, newly introduced triangles are irrelevant with respect to the Fea, since they only contribute static friction, which, in turn, is of no consequence to a clamping operation. This is due to the part being fully constrained by the mold jaws after being clamped; as such, static friction only plays a role during the vise being closed around the part. The Fea considers only the effects of tightening past that point. Furthermore, Section 9.3 deals with tolerances of the mold jaws. If tolerances are correctly set, faces with normals orthogonal to the clamping force can only make contact with the mold jaws through deformation, as in their undeformed state, there is a gap between them and the mold jaws. As such, no friction occurs during the insertion of a part into the mold jaws. Meshes resulting from the Minkowski sum operation are repaired as previously detailed. This prepares them for the conversion to Brep geometry, reducing unwanted artifacts that need to be handled, e.g., internal edges.

Listing 9.1 details the steps necessary to convert an Stl file into Brep solid geometry. Converting every face of the Stl into a corresponding Brep face

[1] The binary is compiled with Microsoft Visual C++. The /O2 option is utilized. This roughly corresponds to the -O2 option of the GCC compiler.

Listing 9.1: Conversion of STL to BREP solid

```
def import_stl_as_shape(filename: str):
    shape: OCP.TopoDS.TopoDS_Shape = OCP.TopoDS.TopoDS_Shape()
    StlAPI.Read_s(shape, filename)

    sew = BRepBuilderAPI_Sewing()
    sew.Add(shape)
    sew.Perform()
    shape = sew.SewedShape()

    if shape.IsNull():
        raise Exception("Critical error, STL could not be sewn into a shape.")
    if shape.ShapeType() == TopAbs_ShapeEnum.TopAbs_COMPOUND:
        print("Shape is a compound, finding largest shell.")
        iterator = TopExp_Explorer(shape, TopAbs_SHELL)
        max_children = 0
        while iterator.More():
            if iterator.Current().NbChildren() > max_children:
                max_children = iterator.Current().NbChildren()
                shape = iterator.Current()
            iterator.Next()
        if max_children == 0:
            raise Exception("Critical error, no shell had any content.")

    shape_upgrade = ShapeUpgrade_UnifySameDomain(solid_builder.Shape())
    shape_upgrade.SetAngularTolerance(4e-5)
    shape_upgrade.SetLinearTolerance(2e-4)
    shape_upgrade.Build()

    shape_fix = ShapeFix_Shape(shape_upgrade.Shape())
    shape_fix.Perform()

    wire_fix = ShapeFix_Wireframe(shape_fix.Shape())
    wire_fix.ModeDropSmallEdges = True
    wire_fix.FixSmallEdges()
    wire_fix.FixWireGaps()
    shape = wire_fix.Shape()

    solid_builder = BRepBuilderAPI_MakeSolid(OCP.TopoDS.TopoDS().Shell_s(shape))
    solid_builder.Build()

    cq_shape = Shape.cast(solid_builder.Shape())
    return cq.Workplane(f"XY").newObject([cq_shape])
```

is straightforward as the OPENCASDADE kernel contains a function for this, `StlAPI.Read_s`. However, the returned shape is not a usable solid at this point. The import of the STL returns a shape as a collection of BREP faces. Due to the repair operations carried out on the input STL, it can be assumed that the imported geometry forms a manifold solid. Specifically, there should be no major internal edges or faces to take into account. As such, utilizing the `Sewing` API is a safe operation; there exists at least one water-tight shell of the part that can be created. Executing `BRepBuilderAPI_Sewing` may yield two

different types of results. The returned shape can be a `TopAbs_SHELL`, in which case no special handling of the result is required. However, the returned shape can also be a `TopAbs_COMPOUND`, indicating that the sewing operation yielded multiple shells. This can happen due to floating-point errors, or malformed edges or faces not identified during the repair of the STL. The additional shells are internal or redundant faces that the Minkowski sum operation created[1]. In this case, all shells present in the compound need to be traversed. The shell with the most children[2] is identified as the shell of the actual part, while all other shells get discarded. Under ideal circumstances, the identified shell is suitable to be boundary-filled and thus converted to a solid. In practice, this is not the case for two reasons. Firstly, the isotropic remeshing can adversely affect the performance while setting up the FEA analysis in ABAQUS. This is due to large planar surfaces consisting of many isotropic triangles that share the same normal. When these triangles are translated into a BREP surface, they slow down detecting contact surfaces in ABAQUS, as instead of one large plane, many small redundant ones need to be considered. Secondly, there can still be undesired BREP geometry present. Straight edges may be split into several short edges as a remnant of the STL, similar to the effect that occurs with planar surfaces. Additionally, the shell is not necessarily water-tight; there may be gaps within the boundaries of faces (so-called wires, typically called edge loops in mesh processing). This is due to floating-point errors leading to some triangles not being "closed" but "open." The sewing operation connects faces into a shell based on their edges and is thus not affected by this; however, to obtain a solid, these faces need to be corrected.

To remedy the first issue, the `ShapeUpgrade_UnifySameDomain` API is used. It identifies adjacent faces that have normals with directions that are close to equal, as well as faces with identical normals, offset from each other along the normals' axes by small distances[3]. The tolerance values used in Listing 9.1 are experimentally calibrated. An example of the effect this unification has is given in Figure 9.2, depicting on the left faceted BREP geometry obtained by converting an isotropically meshed planar surface "as is," and on the right the result of executing the `ShapeUpgrade_UnifySameDomain` API.

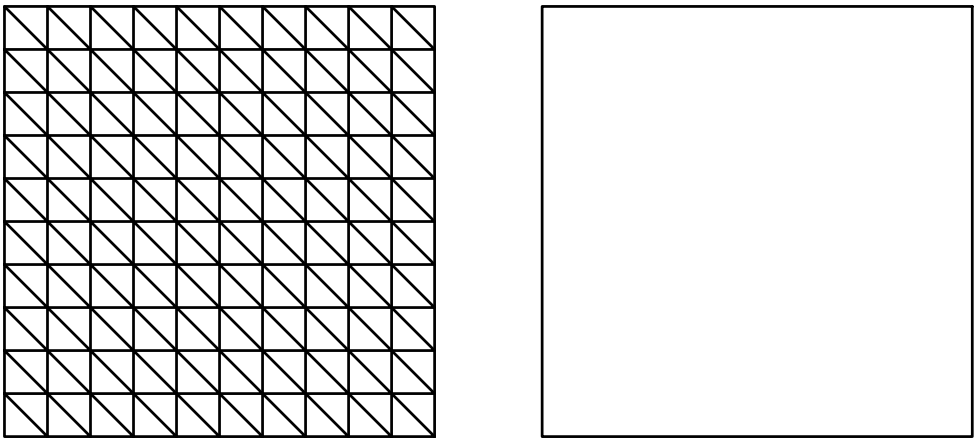
The second issue is a common one that also happens within CAD software when performing a series of extrudes and cuts toward the same plane, where floating-point errors can lead to malformed edges. The OPENCASCADE kernel

[1] Surfaces close to orthogonal to the vector of the computed Minkowski are prone to causing such faces, due to limited floating-point accuracy.

[2] Children are faces, edges, and vertices.

[3] This eliminates stair-stepping, e.g., planar surfaces at an angle that are discretely approximated by a series of faces that look like a stair, as well as surfaces that can be thought of as a series of pillars that are all close to the same height.

Figure 9.2: Effect of utilizing unification on faceted planar surface



contains functionality to specifically address this type of malformed geometry. By utilizing the `ShapeFix_Shape` API, any remaining internal edges or isolated overhanging edges are eliminated. This API also runs a series of other fixes[1]. Following this, the `ShapeFix_Wireframe` API is called[2]. It is able to fix edges that consist of many individual segments and to close all wires that serve as boundaries of faces. Overall, the `ShapeFix` tools run a comprehensive set of repairs regarding intersections, boundaries, connectivity, and so on, which would be difficult to re-implement[3].

Finally, after carrying out these repairs, the shell is processed by the `MakeSolid` API, which appears to use some form of boundary fill method in OpenCascade. This means that any parts of a shell that do not enclose a volume are discarded or trimmed[4]. At this point, the generated solid is suitable for further usage, for instance, performing a boolean cut between the Minkowski sum solid and the vise jaws to generate mold jaws, and importing the geometry into Abaqus. The solid is cast into a CadQuery object, which allows the high-level API of CadQuery to be used for translating the solid and

[1] Specifically, tools that apply fixes to the shell, edges, and wires are called; the solid fixing tool is not used since there is not yet a solid present, `https://github.com/Open-Cascade-SAS/OCCT/blob/master/src/ShapeFix/ShapeFix_Shape.cxx`.

[2] This API is not identical to the `ShapeFix_Wire` API, but does make use of it. It wraps the `ShapeFix_Wire` API, applying fixes that are commonly effective.

[3] Considering `https://github.com/Open-Cascade-SAS/OCCT/tree/master/src/ShapeFix`, the code that implements these features is over 10,000 lines and considers numerous cases not touched upon here.

[4] Due to the previous processing, this effect may not matter, as there should be no such faces or parts of faces to discard.

Listing 9.2: Export of BREP solid to STEP file with OpenCascade

```
def export_solid_to_step(shape, path):
    writer = STEPControl_Writer()
    # writer.SetTolerance(1e-6)

    Interface_Static.SetIVal_s("write.surfacecurve.mode", False)
    Interface_Static.SetCVal_s("write.step.schema", "AP203")
    writer.Transfer(shape, STEPControl_ManifoldSolidBrep)

    # writer.Transfer(shape, STEPControl_AsIs)
    writer.Write(path)
```

carrying out boolean operations. The export of the resulting solids of the mold jaws and the faceted part is not done with CADQUERY, as it does not offer enough control over the STEP files. Listing 9.2 shows code that directly uses OPENCASCADE to carry out the export. Most importantly, the STEP schema and `surfacecurve` mode can be set. As the methodology precludes the existence of non-planar surfaces in the resulting solids, the entity used in the STEP file to encode surfaces is changed to only support planes. This reduces file size and is potentially more numerically stable, being less prone to floating-point errors. The STEP schema is set to the older `AP203` format due to it being better supported in most tools that interface with CAD data than the newer `AP242` and `AP214` standards [64, 65, 69]. This is evidenced by the `AP203` standard not having been withdrawn, despite having been superseded, in contrast to the `AP214` standard [64, 69]. This method of exporting also allows for control over how solids are translated/transferred to files. They can be transferred as is, but can also be specified as faceted BREP, which guarantees that all faces are planar, and all edges are straight[1]. The exact influence these transfer modes have on the resulting STEP file has not been extensively tested, but basic experiments yield that they omit elements and do not convert them. The documentation[2] for OPENCASCADE does not make this clear, but only indicates that the STEP entity types written to the file are changed. In practice, by following the previously detailed steps to generate the BREP geometry and using `STEPControl_AsIs`, the resulting artifacts work well in ABAQUS. Finally, especially for FEA, it may be important to set the tolerance value for the STEP file. By generating the mold jaws, the faceted part geometry, and the Minkowski sum all with OPENCASCADE and using uniform tolerances for the

[1] An overview of all the options and specifically the transfer options, the `shape_representation` entities, can be found in the documentation of OPENCASCADE under `https://dev.opencascade.org/doc/overview/html/occt_user_guides__step.html`.

[2] `https://dev.opencascade.org/doc/refman/html/_s_t_e_p_control___step_model_type_8hxx.html`.

STEP files, occurrences of unintentional overclosures or clipping of geometry in ABAQUS can be avoided.

9.2 Abaqus Simulation Setup

THIS section is intended to give insight into the development of the script that sets up the FEA within ABAQUS. The focus is on covering not immediately obvious sources of errors that may occur. Due to their infrequent nature, the covered sources of errors are difficult to identify a priori. These errors are identified throughout weeks of runtime and thousands of simulations. The goal of this section is to document these lessons learned, thus accelerating the re-implementation of the methodology in different FEM software.

The overall structure of the necessary PYTHON script is obtained by recording a macro within ABAQUS. Carrying out the steps to set up a single FE simulation manually then yields a PYTHON script. This script loads the STEP files of the mold jaws and of the faceted part. The loaded geometry is prearranged during its previously detailed generation. As such, no translation operations within ABAQUS occur. The script then creates the appropriate objects for the parts within ABAQUS. Similar to CAD software, ABAQUS differentiates between parts and instances thereof. Both parts and instances can then be added to different sets and other container objects that encapsulate some of their attributes. For instance, a set could contain the entire surface of the part, while another could contain only select vertices of the mesh for a given instance of the part. The precise details of this can be ascertained by considering the full script in the open-source release of OPTI-CLAMP[1] [30], but are not the focus of this section. Each part is meshed with displacement elements of type `C3D10` [32]. This type represents quadratic tetrahedrons with ten nodes. A minimum element length of 0.40 mm is chosen [32]. The minimum element length should be set to be smaller than or equal to the minimum edge length of the isotropic meshes. This is due to FE meshes being volumes, meaning they have internal vertices. The triangle meshes used for STL files only describe the surface of the part. While a large minimum element length still outwardly preserves the facets of the part's geometries, it leads to there not being enough nodes within the volume of the part. This, in turn, would limit the accuracy of results the FE simulation can provide. If only small deformations of the part are expected, which only require sufficient internal resolution of the FE mesh close to its surface, `sizeGrowth=MAXIMUM` should be set when meshing parts. This dynamically increases the minimum element

[1] Available under `https://github.com/tudo-seal/OptiClamp/blob/63981136ac194a0b9ee117004e7e2aa23929f741/coam/templates/clamp.py`.

Table 9.1: Mechanical properties of 316L (LPBF) for FE analysis [129]

Parameter	Value	Unit
Density	7900	$kg\,m^{-3}$
Young's modulus	193	GPa
Poisson's ratio	0.3	–
Yield strength	520	MPa
Ultimate tensile strength	605	MPa

length further away from the surface of the mesh. This, in turn, positively affects the runtime of the simulation, as there are fewer nodes, while still maintaining accuracy within the regions in which the deformations occur.

Table 9.1 gives an overview of the material properties assumed for the faceted part if manufactured through means of LPBF[1]. These values are measured experimentally by Wang et al. [129]. Based on the findings of Yasa et al., the friction coefficient used for sliding contact/tangential behavior that may occur is set to 0.685[2] [137][3]. The clamping force is fixed at 1200 Newtons. Alternative approaches and metrics may allow viewing the clamping force as a hyperparameter, by linearly increasing the clamping force until a given target deformation is reached.

The mold jaws are assumed to be fully rigid regarding the FE simulation. This assumption is overall accurate, as compared to the part with thin features, the mold jaws are solid blocks of material, possibly hardened. There are two different ways this assumption can be modeled in ABAQUS. Listing 9.3 shows both of these approaches. The first approach shown imports the mold jaw geometry as a surface. During meshing, this geometry will receive no internal elements. This surface is set to be discrete, i.e., a polyhedral mesh, and rigid, meaning it is not considered for deformation. This approach speeds up the meshing operation, as no internal meshing needs to happen. The user interface for importing parts into ABAQUS suggests this is the default approach many users take. During the experiments detailed in Section 10.2, it is identified that the second method shown in Listing 9.3 is more reliable and avoids simulations occasionally failing. The second method treats the mold jaws the same way as the part to be clamped, meshing them as FE meshes with internal nodes. Then, the entire mesh is marked as a rigid body. Due to the

[1] Material properties of the mold jaws do not matter, as their deformation is not considered.
[2] Coulomb friction.
[3] See Table 12 and Point 3 in the Conclusion.

Listing 9.3: Comparison of initializing mold jaws in Abaqus

```
jaw = model.PartFromGeometryFile(
    combine        = False,
    dimensionality = THREE_D,
    geometryFile   = jaw_acis,
    name           = "jaw",
    type           = DISCRETE_RIGID_SURFACE,
)

# More stable alternative:

jaw = model.PartFromGeometryFile(
    combine        = False,
    dimensionality = THREE_D,
    geometryFile   = jaw_acis,
    name           = "jaw",
    type           = DEFORMABLE_BODY,
)

model.RigidBody(
    bodyRegion     = jaw_instance_set,
    name           = "jaw_Rigid",
    refPointAtCOM  = ON,
    refPointRegion = Region(
        referencePoints = (jaw_instance.referencePoints.values()[0],)
    ),
)
```

existence of internal nodes, Abaqus better resolves overclosures. Overclosures can occur if, during deformation, geometry would intersect itself, with one part clipping portions of its surface through another. When using a rigid body instead of a rigid surface, the simulation converged faster in edge cases and exhibited an overall higher average increment.

The increment describes how the load applied during the simulation is broken down into several smaller intervals. This enables non-linear effects, e.g., the geometry changing during deformation, which in turn leads to a different contact situation, to be examined by the FE simulation. The settings used for increments in the simulation are given in Listing 9.4. As the expected deformation during clamping with mold jaws is low, and since the mold jaws "encase" the part being clamped, it is rare for dynamic effects to occur. For most orientations of the part and corresponding mold jaws, a single increment is sufficient, making FE simulations comparatively fast. This is generally the case for orientations that lead to good solutions from a technological point of view, i.e., low deformation/high contact surface area. For clamping solutions that are technologically poor, convergence takes longer, as the part may be close to slipping out of the mold jaws. In this case, the simulation must identify the point in time during which slippage happens; thus the increment

Listing 9.4: Increment used for FE Simulation

```
analysis_step = model.StaticStep(
    initialInc = 1,
    maxInc     = 1,
    maxNumInc  = 100,
    minInc     = 1e-10,
    name       = "Apply-Force",
    nlgeom     = ON,
    previous   = "Initial",
)
```

becomes much lower. To avoid wasting time on simulating cases that are close to unstable, the minimum increment is set relatively high.

In the following, two adjustments made to the script to improve the overall reliability of the FE simulations are discussed. While the extensive processing of the meshes and BREP geometry during the generation of the STEP files avoids most issues that arise from the need to use a faceted part instead of the original CAD file, there are some edge cases that are not caught. This is due to two reasons: the values set for eliminating common defects are not set aggressively in prior steps, and no error processing is carried out on the results of boolean cuts generating mold jaws. Not setting aggressive values for any of the fix-up operations is preferable to avoid accidentally removing features of the part. Not processing the mold jaws by default is an intentional choice to ensure that the faces of the mold jaws *exactly* mate with the part in ABAQUS. If the mold jaws are post-processed, this may result in initial overclosures in ABAQUS. As such, there can still be malformed geometry present. As the desired FE meshes have internal tetrahedrons, malformed geometry propagates inwards, causing entire regions of the part to be unable to be meshed by ABAQUS.

Listing 9.5 shows how to remedy this while setting up the simulation. If, after the meshing operation, there are unmeshed regions present, `createVirtualTopology` is called. This feature carries out similar error-fixing methods as previously described in Section 9.1, but using the CAD kernel within ABAQUS. This time, more aggressive parameters are used. This relaxes the topology of the outer shell of the part, creating larger contiguous surfaces. This gives the meshing operation more options to create suitable tetrahedrons with good aspect ratios.

There are several different methods to establish contact in ABAQUS. The simplest one is for the user to graphically select pairs of surfaces anticipated to come into contact during the simulation. This method is reliable and ensures that only necessary contacts are calculated during the FE simulation.

Listing 9.5: Example of using virtual topology in Abaqus

```
if object.getUnmeshedRegions() is not None:
    object.createVirtualTopology(
        applyBlendControls            = False,
        cornerAngleTolerance          = 30.0,
        faceAspectRatioThreshold      = 10.0,
        ignoreRedundantEntities       = True,
        mergeShortEdges               = True,
        shortEdgeThreshold            = 0.1,
        mergeSliverFaces              = True,
        mergeSmallAngleFaces          = True,
        mergeSmallFaces               = False,
        mergeThinStairFaces           = True,
        smallFaceCornerAngleThreshold = 10.0,
        thinStairFaceThreshold        = 0.12,
    )
    object.generateMesh()
```

However, this is not an option for Opti-Clamp, as the optimization loop is not interactive; the user only supervises it.

There are two options within the software to determine contact automatically, both shown in Listing 9.6. One of these is `contactDetection`, which checks each surface against all other surfaces present in the simulation model. Based on their separation, contact pairs are automatically added to the Fea. As computed Step files are set up to precisely match with equal tolerances, selecting all surfaces that have a separation of exactly zero yields correct contacts. However, due to a bug in the Api of Abaqus, options to merge contact surfaces into larger ones do not work. As such, this method creates one contact object for each pair of matching facets. In practice, this creates several hundred contact objects. This leads to the Fea simulation being difficult to work with for a human once optimization finishes. That means that potentially interesting Pareto-optimal solutions cannot be double-checked or modified without much effort. An alternative to this is General Contact. Enabling this contact method allows all potential contacts to be automatically handled with finite-sliding surface-to-surface contact. There are many configurable parameters to ensure that the behavior is as desired. This makes this method more flexible and robust [60]. Especially in an automated setting, it is better to rely on a sophisticated tool specifically designed to abstract away defining contacts precisely. The resulting Fea is easier to modify in consequence. As this option in Abaqus is somewhat newer than contact pair detection, it is not as widely utilized. If similar options are present in other Fem software, usage is encouraged. As the faceted geometry of the part and mold jaws has many surfaces to check, general contact is customized in Listing 9.6 to only consider jaw-to-part interactions, as these should never occur for feasible solutions.

Listing 9.6: Comparison of methods to establish contacts in Abaqus

```
# Contact Detection by Pairs

model.contactDetection(
        interactionProperty               = "IntProp-Fric",
        separationTolerance               = 0.0,
        includeMeshShell                  = ON,
        includeMeshSolid                  = ON,
        includeMeshMembrane               = ON,
        createUnionOfMainSecondarySurfaces = ON, # Does not work
        createUnionOfMainSurfaces         = ON, # Does not work
        createUnionOfSecondarySurfaces    = ON, # Does not work
        meshedGeometrySearchTechnique     = USE_MESH
)

# Global Contact makes the simulation more maintainable for humans and is more flexible

global_contact = model.ContactStd(createStepName="Initial", name="GlobalContact")
global_contact.setValues(globalSmoothing=False)
global_contact.contactPropertyAssignments.appendInStep(
    assignments=((GLOBAL, SELF, "IntProp-Fric"),), stepName="Initial"
)
global_contact.includedPairs.setValuesInStep(
    addPairs=(
        (jaw_actuated_instance.surfaces["Surface"], part_instance.surfaces["Surface"]),
        (jaw_fixed_instance.surfaces["Surface"]   , part_instance.surfaces["Surface"]),
    ),
    stepName="Initial",
    useAllstar=OFF,
)
model.StdInitialization(name="ContactInitialization")
global_contact.initializationAssignments.appendInStep(
    assignments=(
        (
            jaw_actuated_instance.surfaces["Surface"],
            part_instance.surfaces["Surface"],
            "ContactInitialization",
        ),
        (
            jaw_fixed_instance.surfaces["Surface"],
            part_instance.surfaces["Surface"],
            "ContactInitialization",
        ),
    ),
    stepName="Initial",
)
```

9.3 Manufacturing Tolerances

TOLERANCES and how to deal with them in additive manufacturing, specifically in LPBF, are ongoing topics of research in the metal additive manufacturing community [110]. The fact that there is geometric deviation between BREP geometry and manufactured parts leads to the need for post-processing

and, consequently, clamping parts. The reasons for these deviations are primarily twofold: Firstly, the conversion from BREP geometry to triangle meshes always causes an approximation error, and secondly, the thermal properties of the material during manufacture lead to feature-specific deformation during cooling. There are several approaches toward reducing deviations and estimating them a priori, for instance, through FEM-based methods [100, 101], or a combination of FEM-based and analytical methods [140, 141]. However, these approaches are not yet mature, and at the time of writing, no open-source code to utilize them is available. Furthermore, given the simple examples presented[1], it is unlikely that their performance is sufficient to handle complex topology-optimized geometries with hundreds of thousands of faces. This rules out dynamically reducing deformations for OPTI-CLAMP, as such geometries are the intended use case.

For the FEA to match reality, the geometric deviations that occur during manufacturing need to be accounted for. Performing a boolean cut without any tolerance makes inserting the part into the generated mold jaws impossible in practice, resulting in a friction fit. As such, the mold jaws' geometry needs to be offset just enough to guarantee that inserting the part is possible, but not result in poor surface-to-surface contact. The closer the manufactured part and mold jaws match, the less clamping force is required[2]. As dynamically offsetting the faces of the STL based on the previously discussed methods is not yet possible, the faces of the mold jaw are negatively offset by a *fixed* amount. The necessary amount should be calibrated experimentally for the specific machines used to produce the mold jaws and the part.

Attempting to use offsetting tools within CAD software to achieve this is not possible; CAD software is not optimized for parts that have hundreds of thousands of faces, crashing if an attempt is made. Offsetting meshes has been researched in the mesh processing community, and a number of different methods that work for complex STL files have been developed. Cao et al. provide a comprehensive overview of existing methods [26].

Table 9.2 shows the different approaches that Cao et al. cover and their evaluation of them. The table omits information not relevant to OPTI-CLAMP, i.e., runtimes. As the proposed methodology is intended to be carried out during lead times of several days, the runtime to calculate the final offset meshes is negligible compared to the main optimization loop. The main criterion relevant to OPTI-CLAMP is robustness. The meshes generated by OPTI-CLAMP are complex, with high face and vertex counts, and can have a mixture

[1] Geometric primitives like discs, cones, etc.

[2] If the geometries perfectly matched and there are both faces with normals that exhibit positive and negative angles with respect to the clamping vector, the required clamping force would be zero.

Table 9.2: Comparison of offsetting methods [26]

Method	Sharp Feature Preserving	Robustness
Direct Offsetting	3	2
Distance Field	2	3
Minkowski Sum	3	5
Ray-based	3	5
Cao et al.	5	5

of short and long edges. This is due to the final STL files used to produce the mold jaws and the part not being isotropically remeshed but computed based on a boolean intersection of the STL files at a high resolution. This is done because isotropic remeshing is not guaranteed to precisely preserve topology. For the FEA, small deviations are of little consequence, but the final production part should be as topologically accurate as possible. As such, suitable offsetting methods must be able to handle the high face and vertex counts. Accordingly, a second important criterion is how well the offsetting methods preserve topology. This is not just for the previously stated reason, but also because surface-to-surface contact can degrade if features are lost. Sharp features serve as anchors during clamping. Methods that round these, like Minkowski sum-based offsetting[1], may introduce slippage of the part if the clamping force is not increased. Methods that reconstruct the original topology with voxels to offset it, e.g., ray-based methods, are technologically particularly unsuitable. Even with a very high resolution, the structure of the surface is altered, and the manufactured mold jaws exhibit stair-stepping on their surface. This may result in point-to-surface contact, instead of surface-to-surface contact[2].

Figure 9.3 shows an example of the offsetting method by Cao et al. versus direct offsetting, with the original part on the left, direct offsetting in the middle, and the method by Cao et al. on the right. It can be seen that the right geometry preserves all sharp features of the original geometry, while the direct offsetting method produces a plethora of artifacts at the sharp edges.

To utilize the parallel-feature preserving offsetting method by Cao et al. [26], the provided GITHUB repository implementation is forked, and a port capable

[1] The Minkowski sum of a sphere and the mesh is calculated, thus offsetting it. This inherently rounds sharp features. If the size of the sphere is larger than the spot size of the laser of the LPBF machine, then additional rounding is introduced.

[2] This may be desirable in some cases, as the voxels could act as teeth if the jaws are hardened, but this would need to be experimentally verified.

Figure 9.3: Graphical comparison of offsetting methods [26]

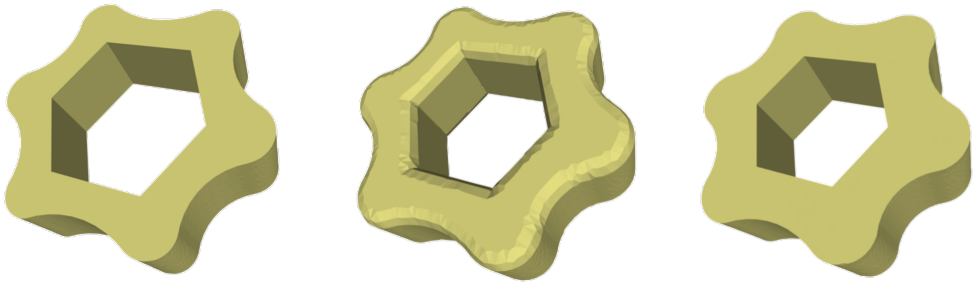


Figure 9.4: Example of part that has been PFP offset

of running on the WINDOWS operating system is created [24, 25]. Figure 9.4 shows an example of the resulting offset mesh on a complex topology optimized part. The translucent green mesh is the offset mesh, while the gray mesh is the original mesh. It can be seen that all sharp features are preserved, even though the part has around 270,000 faces. The required computation time on a high-end workstation is less than an hour. Section 10.2 gives examples of how this offsetting method performed in practice.

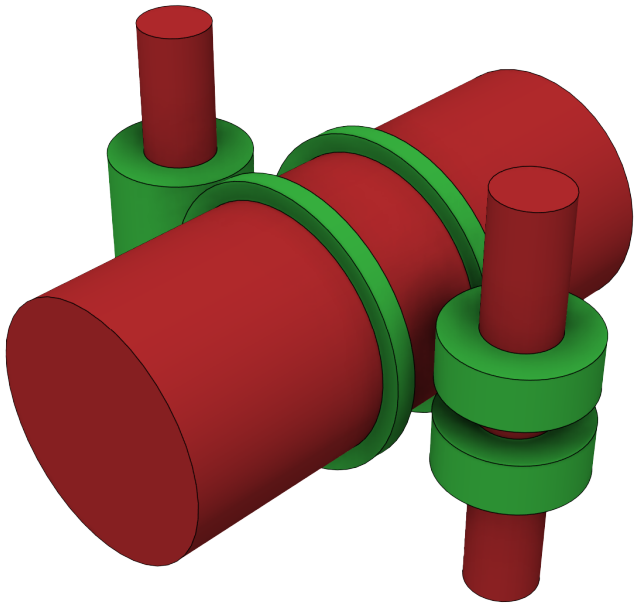
CHAPTER 10

Validation

Experimental validation is crucial to evaluate the efficacy of the methodology and implementation of Opti-Clamp. However, due to the costs associated with manufacturing parts and carrying out technical experiments, comprehensive experiments across entire sets of benchmark parts that showcase specific combinations of features of interest are not possible. The long runtimes associated with the optimization loop, in combination with license constraints for Abaqus, also prohibit comprehensive experiments across such parts regarding the software aspects of Opti-Clamp. As such, validation is carried out by means of an example part that contains many features representative of additively manufactured parts. Considerations for designing such a part are covered within Section 10.1.

Chapter Organization The remainder of this chapter first covers the process of designing an example part suitable for meaningfully testing Opti-Clamp. Considerations to ensure that the part is suitable for drawing conclusions from are covered. Following this, the experimental setups of the two main areas that need validation are presented. First, the optimization loop is verified, validating that the software components identify optimal rotations for each alignment. Secondly, technological experiments with manufactured parts are discussed, carried out to verify that the mold jaws and their offsetting work as intended. Finally, the results of the experiments are interpreted and evaluated.

Figure 10.1: Preserve and obstacle geometry for topology optimization



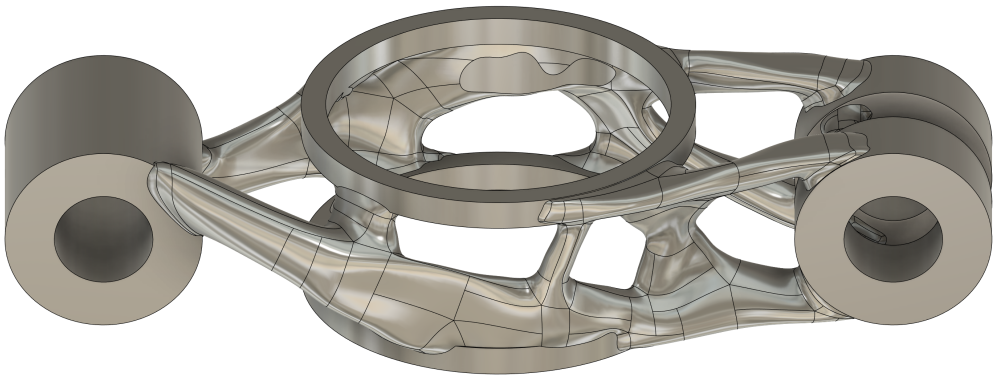
10.1 Design of the Example Part

Given the resource constraints on both the software and technological aspects, it is vital that the example part allows meaningful conclusions to be drawn while also provoking the identification of many edge cases. This means that the part's geometry needs to be complex, featuring heterogeneous[1] free-form surfaces, as well as thin and thus easily deformable and breakable features. However, the optimal rotations of the part also need to be known prior to carrying out FEA. To design such a part, topology optimization is used. Setting a high resolution for the topology optimization and minimizing the part's mass yields thin features. Furthermore, topology optimization virtually guarantees that complex free-form surfaces are generated. Additionally, manufacturing topology-optimized parts is one of the typical use cases for which SLM is utilized. As topology optimization specifically optimizes a part to withstand given loads, the clamping force acting on each alignment is applied as the load to optimize around for each given rotation.

Figure 10.1 shows the setup of the topology optimization in FUSION 360. The green geometry is the geometry to be preserved, indicating to the optimization that it needs to be present in the final optimized part. The red geometry represents obstacles; no geometry may be present in the areas marked by it in

[1] Not continuously concave or convex.

Figure 10.2: BREP geometry of the topology-optimized part



the final part. Clamping forces, i.e., the optimization loads, are applied along the center axis of the large red cylinder, as well as along the center axes of the two small red cylinders. As the part structurally imitates a flange or connector, torque is also applied along the axes. The topology optimization is configured to minimize the final mass of the resulting part, up to a safety factor larger than two. The material properties for the optimization are those of AISI 304 stainless steel. An additional condition is imposed on the optimization to disallow generated surface angles lower than 30 degrees, making the resulting part easier to additively manufacture.

These settings result in the BREP geometry shown in Figure 10.2, utilized in all previous chapters as an example part. Examining the shading of the individual faces, many free-form faces exhibit a mixture of convex and concave regions. Some of the connecting geometry of the center two rings exhibits thin cross-sections, with one pillar-shaped feature being particularly frail. This combination of features provokes instabilities in the FEA, motivating many of the implementation details regarding the generation of faceted STEP files and the configuration of ABAQUS to enhance robustness. Accurately meshing this part also yields large STL files with hundreds of thousands of faces, provoking errors in the offsetting procedure. This illustrates the need for the state-of-the-art offsetting procedure used.

Because the part has been topology-optimized using the clamping load as an input at specific rotations of the alignments, the expected outcomes of a successful optimization of the clamping solution by OPTI-CLAMP are precisely these rotations[1].

[1] Discounting dependencies between the alignments.

10.2 Experiments

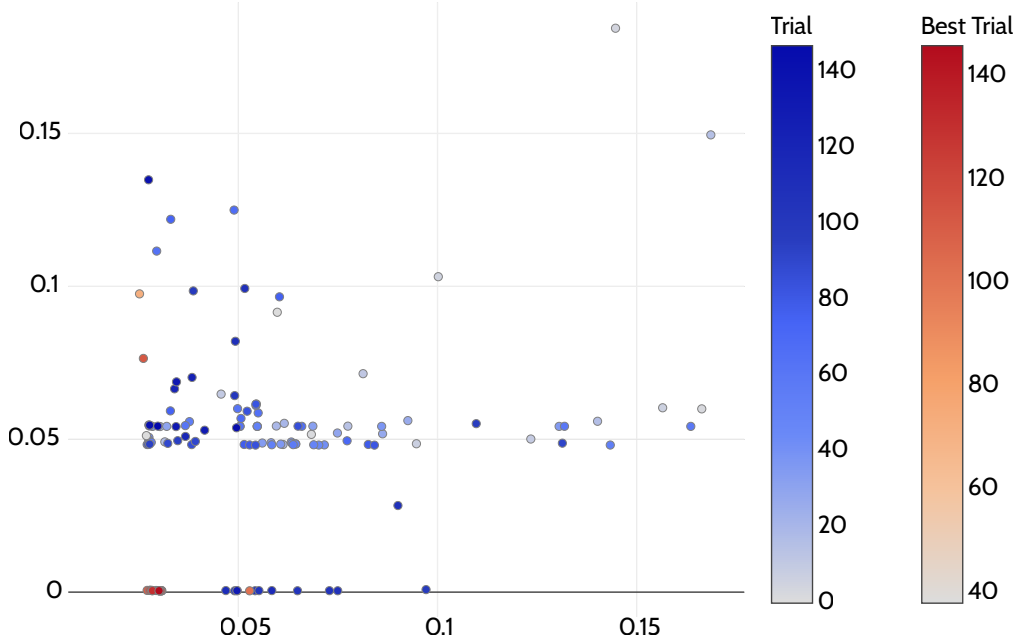
In the following, the setups of both the experiments carried out in software as well as the technological experiments are explained. The software experiment serves to validate the methodology and implementation of Opti-Clamp, attempting to automatically identify optimal rotations of the example part. The technological experiment investigates whether the generated mold jaws are suitable for clamping the parts in practice.

10.2.1 Automatic Identification of Known Optimal Solution

To verify that Opti-Clamp finds the rotation of each alignment of the part that aligns the clamping force applied during the Fea with those forces the part is optimized around, the part's alignments and the corresponding obstacle geometry are exported as Stl files from Fusion 360 at the highest resolution possible. These files are then loaded into Opti-Clamp. For the first alignment, the part is oriented as shown in Figure 10.2. For the second alignment, the part is rotated so that the axes of the two small red cylinders shown in Figure 10.1 are oriented parallel to the tool axis/Z-axis. The first alignment is set to utilize a penetration depth of three millimeters, and the second alignment, six millimeters.

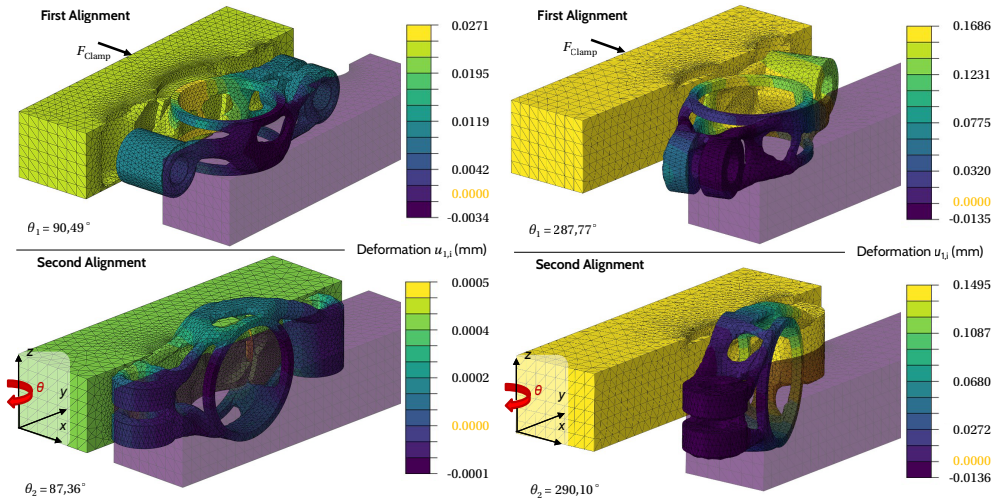
The main optimization loop of Opti-Clamp is then entered. The optimization loop is run for a total of 147 iterations, being terminated as described in Section 8.4, based on the observed results. Figure 10.3 depicts a plot of the deviations of each alignment. The x-axis shows the deformations of the first alignment, and the y-axis shows the deformations of the second alignment. The unit of both axes is millimeters. Data points not on the Pareto front are marked in shades of blue, while those on the Pareto front are marked in red. Darker shades indicate points identified during later iterations. It can be seen that especially during the first iterations, the sampled rotations of the alignments perform poorly. As the optimization progresses, the process gets stuck in a local minimum, leading to the cluster of samples at the intersection of around $x = 0.025$ mm and $y = 0.05$ mm. Toward the end of the optimization, this local minimum is exited and an improved Pareto front is discovered. The achievable value for x, the deformation of the first alignment, remains around 0.025 mm; however, the value for y is improved to a deformation of around 0.005 mm. This "jump" is explainable due to the strongly non-linear properties of the topology-optimized part. While the part is guaranteed to withstand the loads precisely along the direction it is optimized around, unlike a conventional part with more mass, the topology-optimized part may

Figure 10.3: Deviations of first and second alignments per iteration [32]



be significantly weaker against loads that come from even a slightly different angle. Figure 10.4 compares the three-dimensional visualization of the FEA of a sub-optimal iteration, picked from the random-sampling warm-up phase of the optimization loop, with a Pareto-optimal iteration. The sub-optimal solution is shown on the left, while the Pareto-optimal solution is shown on the right. The top images display the clamping solution for the first alignment, and the bottom images display the clamping solution for the second alignment. The respective rotation angles for each pictured clamping solution are given in the lower left corners. The scales for each clamping solution are colored relative to their respective maximum deformation. The Pareto-optimal solution's alignments exhibit far lower deformation than the randomly picked solution's alignments. For the first alignment, the reduction is about one order of magnitude; for the second alignment, between two and three orders of magnitude. The rotations of the Pareto-optimal clamping solution align the part so that the clamping load and the loads of the topology optimization are aligned. This explains the significantly lower deformations; the part is optimized to take loads along these axes. The second alignment, exhibiting a rotation that is some degrees off of ninety degrees, is within expectations, as the applied torque during topology optimization skews the optimal axis for this load.

Figure 10.4: Comparison of Pareto-optimal solution with random solution [32]



10.2.2 Manufacture and Testing of Mold Jaws with Offsets

To verify that the results computed by means of FEA are applicable in practice, the example part and mold jaws pictured on the left of Figure 10.4 are manufactured, and the contact pressure between them is visualized. Figure 10.5 shows the manufactured part and mold jaws in both alignments. The parts and the mold jaws are created through selective laser sintering NYLON PA12. While production from metal powder by means of SLM is preferable, for initial evaluation steps, SLS using PA12 is deemed sufficient and more economical. NYLON PA12 manufactured this way achieves about 80 percent of the strength of injection molded parts and behaves close to isotropically [54]. The mold jaws are produced with different offsets: 0.0 mm, 0.1 mm, and 0.2 mm, on a FORMLABS FUSE 1 SLS printer. Additionally, the part itself is produced with different layer orientations. In the following, horizontal layers refer to the normal of each layer being parallel to the axis of the large red cylinder in Figure 10.1, and accordingly, vertical layers refer to the normal of each layer being parallel to the axes of the small red cylinders in Figure 10.1.

To measure and visualize the distribution of the contact pressure across an entire surface, FUJIFILM PRESCALE ULTRA LOW LLLW (3LW) film is utilized. This type of film contains microcapsules filled with colored ink. If the pressure at a microcapsule exceeds a certain threshold, the capsules burst, creating a colored spot on the film [122]. The used film is suitable for visualizing contact pressures along surfaces of 0.2 MPa to 0.6 MPa [122]. Figure 10.6 illustrates which parts of the surface are measured for each alignment. Shown on the left, for the first alignment, a part of the center of the mold jaw that makes contact

Figure 10.5: Example of mold jaws and example part SLS-printed from PA12

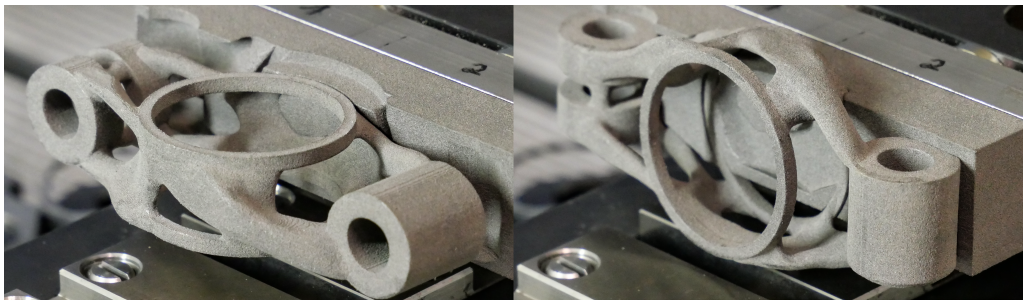
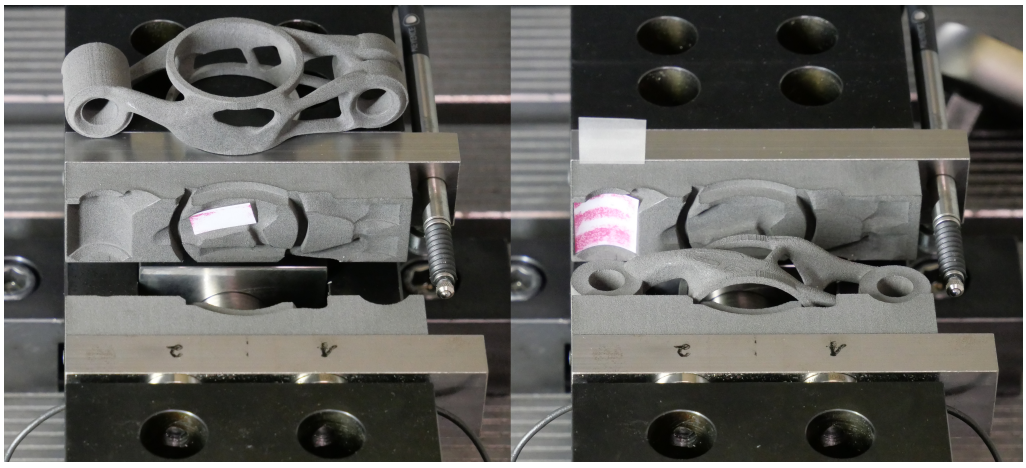


Figure 10.6: Surfaces for which pressure is measured



with thin features of the part is measured. The measured area is around 18 by 8 mm. For the second alignment, contact with preserve geometry, whose mass has not been optimized, is measured. The measured area is around 18 by 18 mm. The measurements are performed for the mold jaws that are offset by 0.1 mm and 0.2 mm, each set of mold jaws being tested with each different layer alignment of the part, leading to four measurements in total. The non-offset reference mold jaws are not tested, as the part cannot be inserted without mechanical force. Figure 10.7 shows the results of the four measurements of the cavity shown on the left of Figure 10.6. Contact is achieved, but not uniformly. Overall, offsetting by 0.1 mm leads to better contact than offsetting by 0.2 mm, but the effect is not pronounced. The alignment of the layers appears to have a significant impact on the uniformity of the contact. Horizontal layers lead to more uniform contact. Since the measured surface is in a small cavity of high curvature, the difference in resolution achievable by the layers in comparison to the laser spot size may be the cause of this. The

Figure 10.7: Pressure between mold jaw and first part alignment

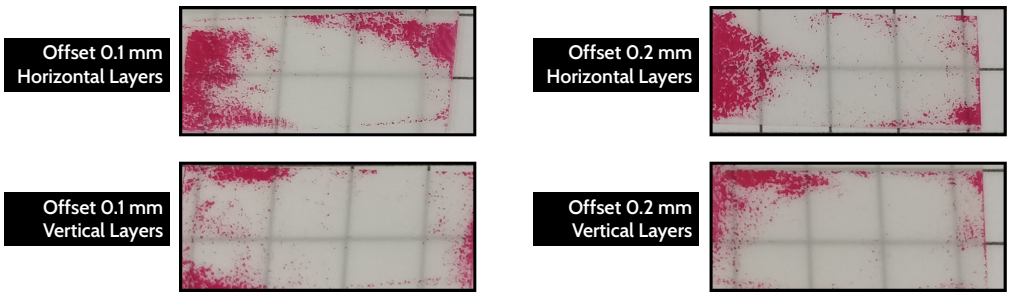
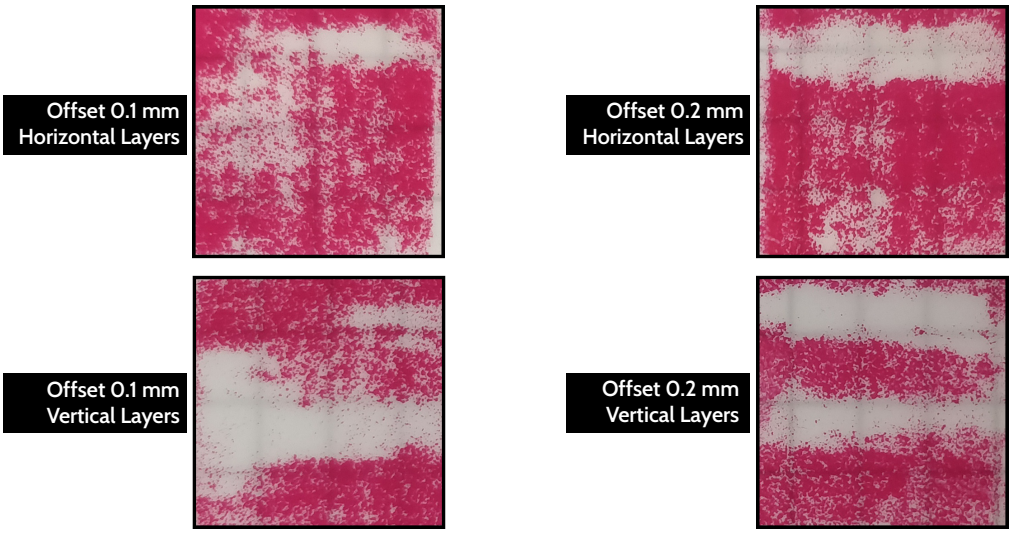


Figure 10.8: Pressure between mold jaw and second part alignment



laser spot size of the Formlabs Fuse 1 is 200 microns, while the layer height is 110 microns. However, the laser spot size moves continuously, not creating the stair-stepping surface approximations of discrete layers. As such, a smoother surface alongside the less curved horizontal direction of the surface yields better contact. Figure 10.8 shows the results of the four measurements of the cavity shown on the right of Figure 10.6. Overall, uniform contact is achieved between the measured surfaces of the mold jaws and the part. The total area of contact looks similar across both values for offsetting the mold jaws, with 0.1 mm appearing to offer slightly better contact once more. Notably, there are discrete bands of no contact on the pressure-sensitive film. This is likely an effect caused by the manufacturing machine. Toward the end of maintenance intervals of the Formlabs Fuse 1, the movement of the build chamber can cause small layer shifts. These layer shifts usually propagate for a few

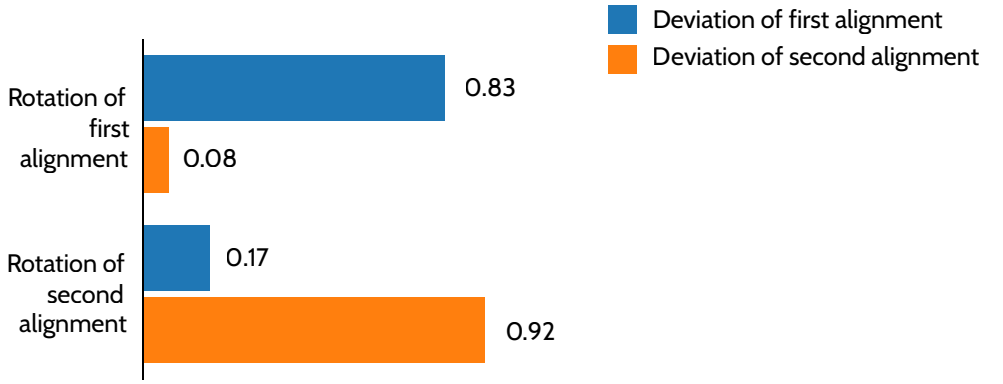
millimeters before the part becomes accurate again during the manufacturing process[1]. The bands occurring for both horizontal and vertical layers of the part can be explained by the mold jaws only having been manufactured with horizontal layer alignment; layer normals are orthogonal to the clamping direction and point in the direction of the cutting tool.

10.3 Evaluation

Considering the outcomes of the experiments detailed in Section 10.2, the methodology and implementation of Opti-Clamp can be considered to successfully enable the usage of conventional vises for the post-processing of additively manufactured parts. The software-based experiments show that the optimization loop is able to determine both optimal rotations for the example part within 147 iterations. This is significantly better than performing a grid search for two alignments, as even discretizing the possible rotations into steps of ten degrees results in a grid search needing to sample at $36 \cdot 36 = 1296$ points in total. With a runtime of at least five minutes for each sample, this procedure would take 108 hours. With finer discretization, this number quadratically increases, quickly scaling to an overall time taken that makes it no longer plausible to carry out within lead times for manufacturing parts. In comparison, using Opti-Clamp, this time can be reduced to just around twelve hours. Additionally, the optimization loop implicitly learns interdependencies between the rotations of alignments. A human may be able to visually recognize obvious interdependencies but will not be able to quantify them or derive which rotations improve overall results. Figure 10.9 shows the hyperparameter importance identified by the optimization. For each alignment's rotation, the bars indicate how much they affected their own deformation as well as their influence on the other alignment's deformation. This can help explain unexpected behavior during experimental benchmarking, such as the optimizer sometimes not outputting previously expected values, homing in on a Pareto front that is close to the expected one, but consistently slightly off. If there is significant interdependence evidenced by the hyperparameter importance, this can cause such a skew. Additionally, this plot can be utilized by a human user to gauge the probability of a satisfactory Pareto front not existing for a given part and its alignments. For instance, if the optimizer cannot provide solutions that meet the minimum technological requirements for permissible deformation after many iterations, and the interdependencies between the

[1] This is due to spindle binding in the build chamber and the nylon powder needing a few layers to be precisely level again.

Figure 10.9: Hyperparameter importance of the rotations of each alignment



alignments' rotations are high, then separate mold jaws for each alignment may need to be used.

The results of the technological experiments are mixed. The fact that the mold jaws work in practice, parts can be inserted, and the vise tightened to affix them verifies that Opti-Clamp generates practically applicable solutions. The contact between surfaces of primitive geometry is good. However, surface-to-surface contact is suboptimal for complex free-form surfaces. It is unclear if this is caused by the Sls manufacturing process of the part itself, as explained for the larger cavity. Depending on the part's geometry, this may necessitate higher clamping forces to eliminate any play during the machining process. Alternatively, some form of surface coating may be applied to the manufactured mold jaws to combat this effect, ideally being applied and then molded by inserting the part coated with a release agent. However, this may not be necessary. Whether the resulting contact is sufficient needs to be further evaluated by manufacturing the part from metal and carrying out an actual cutting operation. With the material properties of steel, the contact may be sufficient to hold the part securely with lower clamping forces than utilized for the Fea. Furthermore, the Slm manufacturing process typically does not have a moving build plate. This may mean that some of the effects that occur during Sls manufacturing do not occur, leading to overall better surface-to-surface contact. Additionally, the Fea considers the clamping force fixed, optimizing the deformation. In practice, the clamping force is not fixed and can be further reduced. This could be incorporated into the optimization loop. This means that there is leeway in the application of clamping force to compensate for dimensional inaccuracies.

The quality of surface-to-surface contact may still be improved by tighter tolerances. Further tests with different offsets can be carried out to see if the

trend of improved contact at lower offset values continues. More advanced methods to improve dimensional accuracy for SLM-manufactured parts may also allow for compensating effects, e.g., thermodynamic ones dependent on the geometry itself, for instance, those by Peng et al. and Zhu et al. [100, 101, 140, 141]. These account for the types of deformations that occur specifically during SLM, e.g., on thin features, concave and convex features, and so on. Parallel-feature preserving methods do not account for such effects. This may significantly improve surface-to-surface contact of free-form surfaces.

Overall, the experiments indicate that automatically finding good solutions from a technological point of view is possible with OPTI-CLAMP. The design space for mold jaws can be more thoroughly explored and evaluated than is possible through manual labor. The solutions are technologically sound and work in practice. Improvement in surface-to-surface contact is possible, but the necessity needs to be experimentally evaluated, especially since the parts are fully constrained via many surfaces at once. Thus, unevenly distributed contact pressure on complex surfaces may not matter. Even without further improvement, the mold jaws offer reduced clamping forces, hold the part more securely than hand clamps can, and allow the use of vises, which are easy to operate and highly repeatable.

CHAPTER 11

Summary and Outlook

Within this part of the thesis, the methodology, implementation, and experiments to evaluate the practical applicability and usefulness of Opti-Clamp were presented. In the following, a summary of these previously covered aspects is given. Then, an outlook highlighting the large body of potential further research into the proposed method is provided.

11.1 Summary

At its core, Opti-Clamp proposes a methodology that marries the automatic generation of mold jaws for given geometries with Bayesian optimization. This combination allows for identifying Pareto-optimal rotations for multiple different alignments or geometries, which can all be clamped with a conventional vise by utilizing a single set of auto-generated mold jaws. The methodology is able to take interdependencies between the multiple parts and alignments into account, as the optimizer learns these during sampling. Furthermore, guidelines on how to practically interact with Opti-Clamp are provided, such as running the optimization loop infinitely and terminating based on time constraints or if a technologically satisfactory solution is found.

Chapter 8 introduces the concept of mold jaws and alignments of a part with respect to the cutting tool axis. It is explained that alignments may have differing geometries based on the already carried out post-processing steps. The concept of obstacle geometries is introduced. In Section 8.1, the individual steps that need to be implemented for Opti-Clamp are covered, and each step's necessity is explained. Each of the three phases of the methodology

is then individually covered. Section 8.2 introduces the concept of obstacle geometries, which ensure that regions that need to be machined remain accessible in the mold jaws. Reasons for why an involved process of converting BREP to STL to BREP is necessary are given. Section 8.3 explains the main optimization loop. Primary drivers of runtime and considerations regarding the optimization are covered. Finally, Section 8.4 explains different plots computed by the visualization component for OPTI-CLAMP and how to technologically interpret these within the context of clamping solutions to determine when to exit the optimization loop.

Chapter 9 covers the implementation of OPTI-CLAMP. First, the choices of software frameworks for OPTI-CLAMP are explained. The choices of optimization framework, CAD data manipulation framework, and of FEA software are discussed. Relevant alternative choices are explored as well, highlighting that the methodology is modular and that individual framework choices can be switched out. Following this, Section 9.1 provides a detailed overview of the steps needed to generate the input files for FEA. Reasons for each individual step are explained, performance considerations are discussed, and the interdependence between the generation of these files and the quality and robustness of the FEA is highlighted. Section 9.2 explains how the FEA within ABAQUS is set up. Information regarding improving reliability and robustness, as well as enhancing the performance of the FEA is provided, aiding in implementing OPTI-CLAMP for different FEA software. Facilitating technological experiments, Section 9.3 gives an overview of different offsetting methods to account for manufacturing tolerances. Reasons for choosing a parallel-feature preserving method are explained.

In Chapter 10, an overview of the steps taken to test the practical applicability of OPTI-CLAMP is given. Section 10.1 explains how to design a part that allows verifying many different aspects simultaneously, for instance, testing the robustness of the implemented STEP file generation, while verifying that the optimization yields expected results. After this, in Section 10.2, the experiments carried out on both the software and the technological/practical side are explained, and their results are shown. Finally, their results are evaluated in Section 10.3.

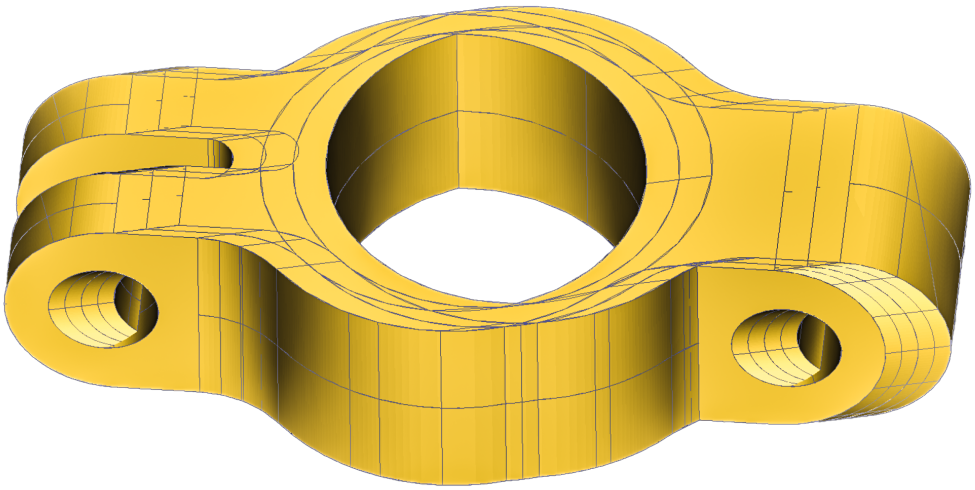
OPTI-CLAMP's introduced methodology and corresponding implementation give technicians easy-to-use tools to improve the efficiency of clamping complex additively manufactured parts for post-processing. If producing such parts in small production runs, e.g., more than ten parts but less than one hundred, there is an economic incentive to use OPTI-CLAMP, as it allows the design space to be efficiently explored in a fully automated manner. Automation allows the exploration to be asynchronous and thus be performed in lead times for orders, leading to a low opportunity cost. As such, only the produc-

tion costs of the mold jaws need to be amortized. As the Pareto-optimal mold jaws allow the usage of vises, the arming times between post-processing operations of parts are lower, there is no need for probing before each operation, and the task of fastening the part requires a less skilled worker. Additionally, the achievable surface quality and dimensional accuracy may be better due to lower clamping-induced deformation. These advantages directly translate into monetary gain, thus amortizing the production costs of the mold jaws or even generating revenue. This also makes the production site more flexible, as challenging new parts that may need to be produced on short notice do not need to be first technologically examined and a clamping solution designed; the automated process to determine the mold jaws can be started immediately when the part design is finalized, even before production is scheduled.

11.2 Outlook

The presented methodology and implementation of Opti-Clamp provide a solid set of tools to allow technicians with no prior knowledge of black-box optimization to automatically create and select Pareto-optimal mold jaws to simplify the clamping of complex parts. The general methodology is sound, with the only meaningful improvement possible being research into solving the problem of computing the Minkowski sum of arbitrary and highly complex Brep geometry. A solution to this problem would eliminate many of the discussed implementation details and the aspect of Step file generation. Creating the mold jaws would become straightforward, i.e., compute the Minkowski sum of the part, and then perform a boolean cut between the sum and regular rectangular vise jaws. The overall approach toward computing the Minkowski sum of a vector and arbitrary Brep geometry would consist of sweeping non-planar faces and planar faces with normals that are not orthogonal to the clamping force along the clamping force vector for a fixed distance, creating a set of prisms. Alternatively, the faces can be moved by a fixed distance along the clamping force vector, and only the edge boundary extruded, creating an open prism consisting of two surfaces. For planar faces with normals that are orthogonal to the clamping force, the edges would need to be extruded along the clamping force vector to create faces. However, this does not account for self-intersections of the created geometry; for instance, if a “C” shape is swept at a steep angle, the upper part of the C will intersect with itself. The resulting geometry would need to be suitably split into separate entities, and all such undesirable effects resolved. Computing the boundary fill over the compound shape formed by the set of all computed entities, for instance by ray-casting methods, would yield a Brep Minkowski sum. Preliminary

Figure 11.1: Example of BREP geometry for which Minkowski sum fails



research carried out by the author indicates that this is a difficult problem and not as straightforward as it may appear. Due to the nontransparent nature of the inner workings of CAD kernels[1], it is difficult to determine why BOPALGO_CELLSBUILDER and other algorithms for unions of compounds into solids simply fail. Figure 11.1 illustrates the results of an implementation in OPENCASCADE that outputs a BREP Minkowski sum for a part of low complexity that exhibits both convex and concave surfaces. Examining the gray lines reveals how self-intersections of surfaces are present. A boundary fill of all outer surfaces of this part would yield the correct Minkowski sum, but the CAD kernel fails to perform the operation without any error messages, making the cause difficult to debug.

There are a number of possible technical enhancements that can be evaluated. One of these consists of benchmarking whether different STEP file export modes have an effect on the robustness of the FEA. For instance, the `STEPControl_FacetedBrep` export mode may reduce the file size and eliminate all splines from the result. This needs experimental testing, as the implementation of the STEP file format standard varies between implementations and different CAD software beyond core features. Additionally, different isotropic remeshing algorithms may be tested, e.g., the adaptive remeshing algorithm by Dunyach et al. [45].

There is a lot of potential for improving the overall runtime of OPTI-CLAMP.

[1] Even though OPENCASCADE is open-source, understanding the hundreds of thousands of lines of its different algorithms is an arduous task.

Based on the conducted experiments, it becomes evident that, by and large, an increase in the contact surface area negatively correlates with the deformation values. While obtaining the values of the deformation requires carrying out FEA, an initial optimization of the contact surface area could be performed before the main optimization loop begins. Computing the surface contact area costs significantly less runtime than performing FEA and can be done on the STL files without any need to move into the domain of BREP geometry. This means that each alignment's geometries only need to be remeshed once; remaining operations per iteration are then the application of a rotation matrix, a boolean cut, and the contact surface computation. Such operations take less than a second of computation time on a typical workstation. Running an initial optimization would yield intervals for the main optimization loop, narrowing down the full search space of 360 degrees for each alignment to a significantly narrower one. This, in turn, means that the runtime-intensive main optimization loop can be expected to require fewer iterations, saving time. In the same vein, it can be considered to begin the optimization loop with meshes of coarser resolution and gradually increase the mesh resolution as the optimization progresses. This improves the overall runtime of the FEA, also saving time.

Finally, comprehensive experiments with parts produced by LPBF should be carried out. A variety of different offsetting methods, offsets, orientations during manufacture, and different parts can be tested. Contact pressure should then be measured, the post-processing step carried out, and the dimensional accuracy and surface finish measured. Due to the number of variation points, this necessitates producing a large number of example parts and mold jaws, and significant amounts of machining time. As such, the costs are prohibitive. These experiments should be carried out once the improvements detailed previously have been implemented and thoroughly tested.

Part IV

Conclusions

CHAPTER 12

Summary and Outlook

The presented approaches and tools to increase the automation of the CAD and clamping solution design processes show promising results. These two use cases cover significant parts of the entire design process of a new or modified product. In conjunction with previous work by Schäfer et al. to leverage synthesis for planning machining operations [111] and work by Kammerer and Chaumet toward optimizing industrial energy systems [73], a resilient and flexible response to many unforeseen circumstances becomes possible. Factory layouts can be replanned faster, new product designs and modifications to existing products can be performed quicker, and the production of these designs can be planned and set up in parallel. This allows factories to adapt to the challenges of an increasingly volatile and fast-moving world. This chapter briefly summarizes all previously covered material.

12.1 Summary

Within this thesis, the contributions made toward improving the automation of engineering tasks through design space exploration are covered. Necessary preliminaries cover theoretical and practical concepts. Theoretical concepts are explained from an applied technological and engineering viewpoint, making them accessible to these disciplines.

The methodology and conceptualization of a tool that automates exploring the design space of CAD assemblies, CLS-CAD, is described, with a focus on reusable, modular, and CAD software-agnostic design. The technical details of the implementation of this tool are given with a focus on runtime

optimization. The tool is evaluated through sets of experiments, which reveal that it enables a significantly more efficient CAD design workflow, eliminates repetitive manual labor, and emphasizes designing driven by metrics and constraints. This allows designers to more easily break out of the mold of prior designs and explore available structural variance.

A methodology that enables combining the generation of mold jaws with embedding FEA into a black-box optimization loop is covered. The tool that implements said methodology, OPTI-CLAMP, is presented in detail. Implementation details regarding accuracy, stability, and runtime are covered. The tool is experimentally verified on the software side as well as through practical experiments with SLS-printed parts. The result is that lead times of orders can be utilized to prepare a technologically better solution for clamping and post-processing additively manufactured parts. The resulting clamping solutions require less clamping force, allow for better rigidity, and are easier to operate, demanding less skill from technicians.

In combination, the contributions afford factories more flexibility in the face of uncertain supply chains and mass customization entailed by Industry 4.0. Faster and asynchronous design processes allow for quicker responses to external influences.

12.2 Outlook

THE research presented in this dissertation targets key individual aspects of the product cycle. The usage of CAD software is ubiquitous and a significant component of virtually all product development, and clamping and fixturing are essential parts of CNC machining. As mentioned at the beginning of this chapter, other work not covered in detail within this thesis expands further on this, covering aspects of designing and adapting factory layouts. This can be taken much further. The final goal should be an entirely smart factory that requires the minimum amount of manual labor, in which humans are enabled to focus on creative and engaging tasks, and are assisted in making meaningful choices without the tedium of performing prerequisite tasks manually.

On one hand, given the significant efficiency gains that CLS-CAD and OPTI-CLAMP can offer, further research into automating other aspects and tasks that are part of the product design and manufacturing process is advisable. As advances in software and manufacturing increase the complexity of product design and manufacture, the design space available increases, and thus tool support and automation are needed to enable understanding, managing, and exploring it. On the other hand, both of the presented methodologies

and corresponding tools individually have room for further improvements, as covered in Section 7.2 and Section 11.2 respectively. Especially the opportunities offered by machine learning research, which is actively ongoing and also driven by many companies from the CAD sector, seem promising. As more datasets consisting of CAD assemblies, meshes, and BREP geometry are released, applying machine learning to the product design and manufacturing processes seems natural, as they inherently deal with shapes and assemblies thereof.

CHAPTER 13

Closing Remarks

Once the tools of design and the means of manufacture become so potent that everything imaginable can be brought into reality, the core problem of engineering is no longer purely to find an efficient solution to a problem, but also how to efficiently select one out of a whole slew of such efficient solutions. The overarching goal of this thesis, to provide tools and approaches that enable automating design space exploration for challenges within engineering disciplines, appears to become progressively more relevant. As modern technologies provide new alternatives and different approaches, and thus, sources of variance, the design phase for engineering any process, product, layout, etc., becomes ever more involved. Tools to handle the sheer volume of different choices to be made and options available are required to avoid decision paralysis, repetition of previous designs, and rejection of new technology.

Within this thesis, two such tools are covered in detail, but hopefully, also much of the philosophy behind them shines through. As such, the author of this dissertation hopes to see the emergence of many other tools and approaches that bring further automation capabilities to engineering tasks.

Acronyms

AI . . . . . Artificial Intelligence

API . . . . . Application Programming Interface

BCL . . . . . Bounded Combinatory Logic

BFS . . . . . Breadth-First Search

BOM . . . . . Bill of Materials

BREP . . . . . Boundary Representation

CAD . . . . . Computer-aided Design

CEF . . . . . Chromium Embedded Framework

CI . . . . . Continuous Integration

CLS . . . . . Combinatory Logic Synthesizer

CLSP . . . . . Combinatory Logic Synthesizer with Predicates

CNC . . . . . Computer Numerical Control

CRUD . . . . . Create, Read, Update, Delete

DFS . . . . . Depth-First Search

DLL . . . . . Dynamic-Link Library

DOF . . . . . Degrees of Freedom

FCL . . . . . Finite Combinatory Logic

FCLP . . . . . Finite Combinatory Logic with Predicates

FEA . . . . . Finite Element Analysis

FEM . . . . . Finite Element Method

GUI . . . . . Graphical User Interface

ISO . . . . . International Organization for Standardization

JSON . . . . . JavaScript Object Notation

LLM . . . . . Large Language Model

LPBF . . . . . Laser Powder Bed Fusion

MILP . . . . . Mixed Integer Linear Programming

REST . . . . . Representational State Transfer

ROS . . . . . Robot Operating System

SAASBO . . . . . Sparse Axis-Aligned Bayesian Optimization

SLM . . . . . Selective Laser Melting

SLS . . . . . Selective Laser Sintering

STEP . . . . . Standard for the Exchange of Product Data

STL . . . . . Stereolithography

UID . . . . . Unique ID

URDF . . . . . Unified Robotics Description Format

Bibliography

[1] Shohin Aheleroff, Naser Mostashiri, Xun Xu, and Ray Y. Zhong. "Mass Personalisation as a Service in Industry 4.0: A Resilient Response Case Study". In: *Advanced Engineering Informatics* 50 (2021), p. 101438. ISSN: 1474-0346. DOI: 10.1016/j.aei.2021.101438.

[2] Shohin Aheleroff, Ross Philip, Ray Y. Zhong, and Xun Xu. "The Degree of Mass Personalisation under Industry 4.0". In: *Procedia CIRP* 81 (2019). 52nd CIRP Conference on Manufacturing Systems (CMS), Ljubljana, Slovenia, June 12-14, 2019, pp. 1394–1399. ISSN: 2212-8271. DOI: 10.1016/j.procir.2019.04.050.

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. "Optuna: A Next-generation Hyperparameter Optimization Framework". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Ed. by Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis. New York, NY, USA: ACM, 2019, pp. 2623–2631. ISBN: 9781450362016. DOI: 10.1145/3292500.3330701.

[4] Kristian Amadori, Mehdi Tarkian, Johan Ölvander, and Petter Krus. "Flexible and robust CAD models for design automation". In: *Advanced Engineering Informatics* 26.2 (2012). Knowledge based engineering to support complex product design, pp. 180–195. DOI: 10.1016/j.aei.2012.01.004.

[5] Autodesk. *Autodesk Fusion*. Nov. 20, 2024. URL: https://www.autodesk.com/products/fusion-360/overview (visited on 11/20/2024).

[6] Autodesk. *Autodesk Inventor: 3D modeling software for designers and engineers*. Nov. 20, 2024. URL: https://www.autodesk.com/products/inventor/overview (visited on 11/20/2024).

[7] Autodesk. *Command Inputs*. Nov. 20, 2024. URL: https://help.autodesk.com/view/fusion360/ENU/?guid=GUID-8B9041D5-75CC-4515-B4BB-4CF2CD5BC359 (visited on 11/20/2024).

[8] O.J. Bakker, T.N. Papastathis, A.A. Popov, and S.M. Ratchev. "Active fixturing: literature review and future research directions". In: *International Journal of Production Research* 51.11 (June 2013), pp. 3171–3190. ISSN: 1366-588X. DOI: 10.1080/00207543.2012.695893.

[9] Eytan Bakshy, Lili Dworkin, Brian Karrer, Konstantin Kashin, Benjamin Letham, Ashwin Murthy, and Shaun Singh. "AE: A domain-agnostic platform for adaptive experimentation". In: *Conference on neural information processing systems*. 2018, pp. 1–8.

[10] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. "BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization". In: *Advances in Neural Information Processing Systems 33* (Dec. 6, 2020). URL: http://arxiv.org/pdf/1910.06403v3.

[11] Henk Barendregt. "Introduction to generalized type systems". In: *Journal of Functional Programming* 1.2 (Apr. 1991), pp. 125–154. ISSN: 1469-7653. DOI: 10.1017/s0956796800020025.

[12] Jan Bessai. "A Type-Theoretic Framework for Software Component Synthesis". en. PhD thesis. Technical University Dortmund, 2019. DOI: 10.17877/DE290R-20320.

[13] Jan Bessai, Tzu-Chun Chen, Andrej Dudenhefner, Boris Düdder, Ugo de'Liguoro, and Jakob Rehof. "Mixin Composition Synthesis based on Intersection Types". In: *Logical Methods in Computer Science* 14.1 (2018). DOI: 10.23638/LMCS-14(1:18)2018.

[14] Jan Bessai, Boris Düdder, George T. Heineman, and Jakob Rehof. "Combinatory Synthesis of Classes Using Feature Grammars". In: *FACS*. Vol. 9539. Lecture Notes in Computer Science. Springer, 2015, pp. 123–140. DOI: 10.1007/978-3-319-28934-2_7.

[15] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. "Combinatory Logic Synthesizer". In: *ISoLA (1)*. Vol. 8802. Lecture Notes in Computer Science. Springer, 2014, pp. 26–40. DOI: 10.1007/978-3-662-45234-9_3.

[16] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. "Combinatory Process Synthesis". In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016*. Lecture Notes in Computer Science 9952 (2016). Ed. by Tiziana Margaria and Bernhard Steffen, pp. 266–281. DOI: 10.1007/978-3-319-47166-2_19.

[17] Jan Bessai, Moritz Roidl, and Anna Vasileva. "Experience Report: Towards Moving Things with Types - Helping Logistics Domain Experts to Control Cyber-Physical Systems with Type-Based Synthesis". In: *F-IDE@FM*. Vol. 310. EPTCS. 2019, pp. 1–6. DOI: 10.4204/EPTCS.310.1.

[18] Patrick Beutler, Julian Ferchow, Marcel Schlüssel, and Mirko Meboldt. "Semi-Automated Design Workflow for Bolt Clamping Interfaces to Post-Process Additive Manufactured Parts". In: *Procedia CIRP* 119 (2023), pp. 596–601. ISSN: 22128271. DOI: 10.1016/j.procir.2023.01.013.

[19] Z. M. Bi and W. J. Zhang. "Flexible fixture design and automation: Review, issues and future directions". In: *International Journal of Production Research* 39.13 (Jan. 2001), pp. 2867–2894. ISSN: 1366-588X. DOI: 10.1080/00207540110054579.

[20] Wutthigrai Boonsuk and Matthew C. Frank. "Automated fixture design for a rapid machining process". In: *Rapid Prototyping Journal* 15.2 (Mar. 2009), pp. 111–125. ISSN: 1355-2546. DOI: 10.1108/13552540910943414.

[21] M. Bostock, V. Ogievetsky, and J. Heer. "D^3 Data-Driven Documents". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pp. 2301–2309. ISSN: 1077-2626. DOI: 10.1109/tvcg.2011.185.

[22] Iain Boyle, Yiming Rong, and David C. Brown. "A review and analysis of current computer-aided fixture design approaches". In: *Robotics and Computer-Integrated Manufacturing* 27.1 (Feb. 2011), pp. 1–12. ISSN: 0736-5845. DOI: 10.1016/j.rcim.2010.05.008.

[23] Marcel Campen and Leif Kobbelt. "Polygonal Boundary Evaluation of Minkowski Sums and Swept Volumes". In: *Computer Graphics Forum* 29.5 (2010), pp. 1613–1622. ISSN: 0167-7055. DOI: 10.1111/j.1467-8659.2010.01770.x.

[24] Hongyi Cao, Gang Xu, Rengshu Gu, Jinlanh Xu, Xioayu Zhang, and Timon Rabczuk. *PFP-Offset*. 2024. URL: https://github.com/iGame-Lab/PFPOffset (visited on 02/16/2025).

[25] Hongyi Cao, Gang Xu, Rengshu Gu, Jinlanh Xu, Xioayu Zhang, Timon Rabczuk, and Constantin Chaumet. *Win-PFP-Offset*. 2024. URL: https://github.com/tudo-seal/WinPFPOffset (visited on 02/16/2025).

[26] Hongyi Cao, Gang Xu, Renshu Gu, Jinlan Xu, Xiaoyu Zhang, and Timon Rabczuk. *A Parallel Feature-preserving Mesh Variable Offsetting Method with Dynamic Programming*. Oct. 13, 2023. DOI: 10.48550/ARXIV.2310.08997. arXiv: 2310.08997 [cs.GR].

[27] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. "ShapeNet: An Information-Rich 3D Model Repository". In: (Dec. 9, 2015). DOI: 10.48550/ARXIV.1512.03012. arXiv: 1512.03012 [cs.GR].

[28] Constantin Chaumet. *CLS-CAD*. 2024. URL: https://github.com/tudo-seal/CLS-CAD (visited on 12/20/2024).

[29] Constantin Chaumet. *Modular Components for synthesizing Robotic Arms with CLS-CAD*. 2023. DOI: 10.5281/ZENODO.10051243.

[30] Constantin Chaumet. *OptiClamp*. 2025. URL: https://github.com/tudo-seal/OptiClamp (visited on 02/13/2025).

[31] Constantin Chaumet. "Robotic Arms as a Product Line: Synthesizing Hardware and Software from modular Components". MA thesis. TU Dortmund University, Sept. 22, 2021.

[32] Constantin Chaumet, Jan Liß, Jakob Rehof, and Petra Wiederkehr. "Automatic generation of pareto-optimal clamping solutions for post-processing additively manufactured parts". In: *Procedia CIRP* (2025). 18th CIRP Conference on Intelligent Computation in Manufacturing Engineering (CIRP ICME '24). (in press).

[33] Constantin Chaumet and Jakob Rehof. *CLS-CAD: Synthesizing CAD Assemblies in Fusion 360*. Nov. 30, 2023. DOI: 10.48550/ARXIV.2311.18492. arXiv: 2311.18492 [cs.RO].

[34] Constantin Chaumet, Jakob Rehof, and Thomas Schuster. "A knowledge-driven framework for synthesizing designs from modular components". In: *Procedia CIRP* 128 (2024), pp. 304–309. ISSN: 2212-8271. DOI: 10.1016/j.procir.2024.05.096.

[35] David Coleman, Ioan Sucan, Sachin Chitta, and Nikolaus Correll. "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study". In: *Journal of Software Engineering for Robotics* (2014). DOI: 10.6092/JOSER_2014_05_01_P3.

[36] John J. Craig. *Introduction to Robotics: Mechanics and Control. Mechanics and control*. Addison-Wesley Series in Electrical and Computer Engineering: Control Engineering. Pearson/Prentice Hall, 2004, p. 408. ISBN: 9780131236295.

[37] Dassault Systèmes - SolidWorks Corporation. *SolidWorks*. Nov. 20, 2024. URL: https://www.solidworks.com/product/solidworks-3d-cad (visited on 11/20/2024).

[38] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. "Differentiable Expected Hypervolume Improvement for Parallel Multi-Objective Bayesian Optimization". In: *Advances in Neural Information Processing Systems 33* (2020). URL: http://arxiv.org/pdf/2006.05078v3.

[39] Mariangiola Dezani-Ciancaglini and J.Roger Hindley. "Intersection types for combinatory logic". In: *Theoretical Computer Science* 100.2 (June 1992), pp. 303–324. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90306-z.

[40] Guido Dhondt. *CalculiX*. 2025. URL: https://github.com/Dhondtguido/CalculiX (visited on 02/10/2025).

[41] Guido Dhondt. *The Finite Element Method for Three-Dimensional Thermomechanical Applications*. Wiley, May 2004. ISBN: 9780470021217. DOI: 10.1002/0470021217.

[42] Boris Düdder, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. "Bounded Combinatory Logic". In: Leibniz International Proceedings in Informatics (LIPIcs) 16 (2012). Ed. by Patrick Cégielski and Arnaud Durand, pp. 243–258. DOI: 10.4230/LIPIcs.CSL.2012.243.

[43] Andrej Dudenhefner, Felix Laarmann, Jakob Rehof, and Christoph Stahl. "Finite Combinatory Logic extended by a Boolean Query Language for Composition Synthesis". In: *29th International Conference on Types for Proofs and Programs TYPES 2023–Abstracts*. 2023, p. 105.

[44] Andrej Dudenhefner, Christoph Stahl, Constantin Chaumet, Felix Laarmann, and Jakob Rehof. “Finite Combinatory Logic with Predicates”. en. In: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. DOI: 10.4230/LIPICS.TYPES.2023.2.

[45] Marion Dunyach, David Vanderhaeghe, Loïc Barthe, and Mario Botsch. *Adaptive Remeshing for Real-Time Mesh Deformation*. en. 2013. DOI: 10.2312/CONF/EG2013/SHORT/029-032.

[46] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. “Scalable Global Optimization via Local Bayesian Optimization”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/6c990b7aca7bc7058f5e98ea909e924b-Paper.pdf.

[47] Julian Ferchow. “Additive Manufacturing towards Industrial Series Production: Post-Processing Strategies and Design”. Dissertation. ETH Zurich, 2021. DOI: 10.3929/ethz-b-000514979.

[48] Julian Ferchow, Marvin Bühler, Marcel Schlüssel, Livia Zumofen, Christoph Klahn, Urs Hofmann, Andreas Kirchheim, and Mirko Meboldt. “Design and validation of a sheet metal clamping system for additive manufacturing and post-processing”. In: *The International Journal of Advanced Manufacturing Technology* 119.11-12 (2022), pp. 7947–7967. ISSN: 0268-3768. DOI: 10.1007/s00170-022-08773-5.

[49] Julian Ferchow, Dominik Kälin, Gokula Englberger, Marcel Schlüssel, Christoph Klahn, and Mirko Meboldt. “Design and validation of integrated clamping interfaces for post-processing and robotic handling in additive manufacturing”. In: *The International Journal of Advanced Manufacturing Technology* 118.11-12 (2022), pp. 3761–3787. ISSN: 0268-3768. DOI: 10.1007/s00170-021-08065-4.

[50] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. “Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning”. In: *Journal of Machine Learning Research 23(261), 2022* (July 8, 2020). DOI: 10.48550/ARXIV.2007.04074. arXiv: 2007.04074 [cs.LG].

[51] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. “Efficient and Robust Automated Machine Learning”. In: *Advances in Neural*

Information Processing Systems. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper_files/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf.

[52] Falko Fiedler, Jannik Ehrenstein, Christian Höltgen, Aileen Blondrath, Lukas Schäper, Amon Göppert, and Robert Schmitt. "Jigs and Fixtures in Production: A Systematic Literature Review". In: *Journal of Manufacturing Systems* 72 (2024), pp. 373–405. ISSN: 02786125. DOI: 10.1016/j.jmsy.2023.10.006.

[53] Florian Fischer. *fusion2urdf*. Jan. 11, 2025. URL: https://github.com/SpaceMaster85/fusion2urdf (visited on 01/11/2025).

[54] Göran Flodberg, Henrik Pettersson, and Li Yang. "Pore analysis and mechanical performance of selective laser sintered objects". In: *Additive Manufacturing* 24 (2018), pp. 307–315. DOI: 10.1016/j.addma.2018.10.001.

[55] Peter I. Frazier. *A Tutorial on Bayesian Optimization*. July 8, 2018. DOI: 10.48550/ARXIV.1807.02811. arXiv: 1807.02811 [stat.ML].

[56] Daoyi Gao, Yawar Siddiqui, Lei Li, and Angela Dai. "MeshArt: Generating Articulated Meshes with Structure-guided Transformers". In: (Dec. 16, 2024). DOI: 10.48550/ARXIV.2412.11596. arXiv: 2412.11596 [cs.CV].

[57] Tomasz Grzegorz Gawel. "Review of Additive Manufacturing Methods". In: *Solid State Phenomena* 308 (July 2020), pp. 1–20. ISSN: 1662-9779. DOI: 10.4028/www.scientific.net/ssp.308.1.

[58] Paul Christoph Gembarski, Haibing Li, and Roland Lachmayer. "KBE-Modeling Techniques in Standard CAD-Systems: Case Study—Autodesk Inventor Professional". In: *Managing Complexity*. Ed. by Jocelyn Bellemare, Serge Carrier, Kjeld Nielsen, and Frank T. Piller. Cham: Springer International Publishing, Aug. 2017, pp. 215–233. ISBN: 978-3-319-29058-4. DOI: 10.1007/978-3-319-29058-4_17.

[59] Cornelia Gyorodi, Robert Gyorodi, George Pecherle, and Andrada Olah. "A comparative study: MongoDB vs. MySQL". In: *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE, June 2015, pp. 1–6. DOI: 10.1109/emes.2015.7158433.

[60] Bilal Abdul Halim. *Understanding Abaqus General Contact*. Feb. 13, 2025. URL: https://www.goengineer.com/blog/understanding-abaqus-general-contact (visited on 02/19/2024).

[61] Abhinav Narayan Harish, Rajendra Nagar, and Shanmuganathan Raman. "RGL-NET: A Recurrent Graph Learning framework for Progressive Part Assembly". In: *2022 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*. IEEE, Jan. 2022, pp. 647–656. DOI: 10.1109/wacv51458.2022.00072.

[62] Negar Heidari and Alexandros Iosifidis. "Geometric Deep Learning for Computer-Aided Design: A Survey". In: (Feb. 27, 2024). DOI: 10.48550/ARXIV.2402.17695. arXiv: 2402.17695 [cs.CG].

[63] Jialei Huang, Guanqi Zhan, Qingnan Fan, Kaichun Mo, Lin Shao, Baoquan Chen, Leonidas Guibas, and Hao Dong. "Generative 3D Part Assembly via Dynamic Graph Learning". In: (June 14, 2020). DOI: 10.48550/ARXIV.2006.07793. arXiv: 2006.07793 [cs.CV].

[64] International Organization for Standardization. *Industrial automation systems and integration – Product data representation and exchange –Part 214: Application protocol: Core data for automotive mechanical design processes*. Standard. Geneva, Switzerland: International Organization for Standardization, 2010.

[65] International Organization for Standardization. *Industrial automation systems and integration – Product data representation and exchange –Part 242: Application protocol: Managed model-based 3D engineering*. Standard. Geneva, Switzerland: International Organization for Standardization, 2022.

[66] International Organization for Standardization. *ISO general purpose metric screw threads – General plan*. Standard. Geneva, Switzerland: International Organization for Standardization, 1998.

[67] International Organization for Standardization. *ISO general purpose metric screw threads – Selected sizes for screws, bolts and nuts*. Standard. Geneva, Switzerland: International Organization for Standardization, 1998.

[68] International Organization for Standardization. *ISO general purpose screw threads – Basic profile – Part 1: Metric screw threads*. Standard. Geneva, Switzerland: International Organization for Standardization, 1998.

[69] International Organization for Standardization. *Industrial automation systems and integration – Product data representation and exchange –Part 403: Application module: AP203 configuration controlled 3D design of mechanical parts and assemblies*. Standard. Geneva, Switzerland: International Organization for Standardization, 2010.

[70] Alec Jacobson and Daniele Panozzo. "libigl: prototyping geometry processing research in C++". In: *SIGGRAPH Asia 2017 Courses*. SA '17. ACM, Nov. 2017. DOI: 10.1145/3134472.3134497.

[71] Huaijun Jiang, Yu Shen, Yang Li, Beicheng Xu, Sixian Du, Wentao Zhang, Ce Zhang, and Bin Cui. "OpenBox: A Python Toolkit for Generalized Black-box Optimization". In: *Journal of Machine Learning Research* 25.120 (2024), pp. 1–11. URL: http://jmlr.org/papers/v25/23-0537.html.

[72] Benjamin Jones, Dalton Hildreth, Duowen Chen, Ilya Baran, Vladimir G. Kim, and Adriana Schulz. "AutoMate: a dataset and learning approach for automatic mating of CAD assemblies". In: *ACM Transactions on Graphics* 40.6 (Dec. 2021), pp. 1–18. ISSN: 1557-7368. DOI: 10.1145/3478513.3480562.

[73] Simon Kammerer, Constantin Chaumet, Christian Rehtanz, and Jakob Rehof. "Optimizing Industrial Energy Systems: A Multi-Objective Approach to Decarbonization and Cost Efficiency". In: *2024 6th International Conference on Smart Power & Internet Energy Systems (SPIES)*. 2024, pp. 19–25. DOI: 10.1109/SPIES63782.2024.10983425.

[74] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. "ABC: A Big CAD Model Dataset For Geometric Deep Learning". In: (Dec. 15, 2018). DOI: 10.48550/ARXIV.1812.06216. arXiv: 1812.06216 [cs.GR].

[75] Olga M. Kushnarenko. *Entscheidungsmethodik zur Anwendung generativer Verfahren für die Herstellung metallischer Endprodukte*. Berichte aus dem Institut für Fertigungstechnik und Qualitätssicherung, Magdeburg 14. Aachen: Shaker, 2009. 167 pp. ISBN: 9783832281212.

[76] David Lai. *MontyDB*. URL: https://github.com/davidlatwe/montydb (visited on 01/27/2025).

[77] Joseph G. Lambourne, Karl D. D. Willis, Pradeep Kumar Jayaraman, Aditya Sanghi, Peter Meltzer, and Hooman Shayani. "BRepNet: A topological message passing system for solid models". In: (Apr. 1, 2021). DOI: 10.48550/ARXIV.2104.00706. arXiv: 2104.00706 [cs.LG].

[78] Trevor Laughlin. *pyOCCT*. 2025. URL: https://github.com/trelau/pyOCCT (visited on 02/10/2025).

[79] Jiahui Lei, Congyue Deng, Bokui Shen, Leonidas Guibas, and Kostas Daniilidis. "NAP: neural 3D articulated object prior". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. NIPS '23. New Orleans, LA, USA: Curran Associates Inc., 2023.

[80] Leutenecker-Twelsiek, Bastian. "Additive Fertigung in der industriellen Serienproduktion: Bauteilidentifikation und Gestaltung". de. PhD thesis. 2019. DOI: 10.3929/ETHZ-B-000347164.

[81] Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. "GRASS". In: *ACM Transactions on Graphics* 36.4 (July 2017), pp. 1–14. DOI: 10.1145/3072959.3073637.

[82] Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, et al. "Openbox: A generalized black-box optimization service". In: *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 2021, pp. 3209–3219.

[83] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhopf, René Sass, and Frank Hutter. "SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization". In: *Journal of Machine Learning Research 23 (2022) 1-9* (Sept. 20, 2021). DOI: 10.48550/ARXIV.2109.09831. arXiv: 2109.09831 [cs.LG].

[84] Jiayi Liu, Hou In Ivan Tam, Ali Mahdavi-Amiri, and Manolis Savva. "CAGE: Controllable Articulation GEneration". In: (Dec. 15, 2023). DOI: 10.48550/ARXIV.2312.09570. arXiv: 2312.09570 [cs.CV].

[85] Shutian Liu, Quhao Li, Jingyu Hu, Wenjiong Chen, Yongcun Zhang, Yunfeng Luo, and Qi Wang. "A Survey of Topology Optimization Methods Considering Manufacturable Structural Feature Constraints for Additive Manufacturing Structures". In: *Additive Manufacturing Frontiers* 3.2 (June 2024), p. 200143. ISSN: 2950-4317. DOI: 10.1016/j.amf.2024.200143.

[86] Wing Kam Liu, Shaofan Li, and Harold S. Park. "Eighty Years of the Finite Element Method: Birth, Evolution, and Future". In: *Archives of Computational Methods in Engineering* 29.6 (June 2022), pp. 4431–4453. ISSN: 1886-1784. DOI: 10.1007/s11831-022-09740-9.

[87] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (May 2022). ISSN: 2470-9476. DOI: 10.1126/scirobotics.abm6074.

[88] Alexander Mages, Carina Mieth, Jens Hetzler, Fadil Kallat, Jakob Rehof, Christian Riest, and Tristan Schafer. "Automatic Component-Based Synthesis of User-Configured Manufacturing Simulation Models". In: *2022 Winter Simulation Conference (WSC)*. IEEE, Dec. 2022. DOI: 10.1109/wsc57314.2022.10015425.

[89] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, Dimitrios Zissis, and Dimosthenis Anagnostopoulos. "MongoDB Vs PostgreSQL: A comparative study on performance aspects". In: *GeoInformatica* 25.2 (June 2020), pp. 243–268. ISSN: 1573-7624. DOI: 10.1007/s10707-020-00407-w.

[90] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy J. Mitra, and Leonidas J. Guibas. "StructureNet: hierarchical graph networks for 3D shape generation". In: *ACM Transactions on Graphics* 38.6 (Nov. 2019), pp. 1–19. ISSN: 1557-7368. DOI: 10.1145/3355089.3356527.

[91] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. "PartNet: A Large-Scale Benchmark for Fine-Grained and Hierarchical Part-Level 3D Object Understanding". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2019. DOI: 10.1109/cvpr.2019.00100.

[92] Hans-Christian Möhring, Felix Flöter, and Berend Denkena. "Messtechnische Analyse formflexibler Spannmethoden*". In: *wt Werkstattstechnik online* 102.11-12 (2012), pp. 795–800. ISSN: 1436-4980. DOI: 10.37544/1436-4980-2012-11-12-795.

[93] M. Mumović, N. Šibalić, A. Vujović, and J. Jovanović. "Topological optimization of vice jaws model for pipe clamping". In: *Journal of Physics: Conference Series* 2540.1 (2023), p. 012027. ISSN: 1742-6588. DOI: 10.1088/1742-6596/2540/1/012027.

[94] Luigi Nardi, David Koeplinger, and Kunle Olukotun. “Practical Design Space Exploration”. In: *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Oct. 2019, pp. 347–358. DOI: 10.1109/mascots.2019.00045.

[95] OPEN CASCADE SAS. *OCCT*. 2025. URL: https://github.com/Open-Cascade-SAS/OCCT (visited on 02/10/2025).

[96] Open Robotics. *URDF*. Dec. 7, 2024. URL: https://docs.ros.org/en/jazzy/Tutorials/Intermediate/URDF/URDF-Main.html (visited on 12/07/2024).

[97] Masahiro Ozaki and Masaaki Nomura. “Hyperparameter Optimization Methods in Machine Learning: Overview and Characteristics”. In: *IEICE Transactions on Information and Communication Engineers* J103-D.9 (2020), pp. 615–631. DOI: 10.14923/transinfj.2019JDR0003. URL: https://doi.org/10.14923/transinfj.2019JDR0003.

[98] Parametric Products Intellectual Holdings, LLC. *CadQuery*. Dec. 16, 2024. URL: https://github.com/CadQuery/cadquery (visited on 12/16/2024).

[99] Parametric Products Intellectual Holdings, LLC. *OCP*. 2025. URL: https://github.com/CadQuery/OCP (visited on 02/10/2025).

[100] Hao Peng, Morteza Ghasri-Khouzani, Shan Gong, Ross Attardo, Pierre Ostiguy, Bernice Aboud Gatrell, Joseph Budzinski, Charles Tomonto, Joel Neidig, M. Ravi Shankar, Richard Billo, David B. Go, and David Hoelzle. “Fast prediction of thermal distortion in metal powder bed fusion additive manufacturing: Part 1, a thermal circuit network model”. In: *Additive Manufacturing* 22 (Aug. 2018), pp. 852–868. ISSN: 2214-8604. DOI: 10.1016/j.addma.2018.05.023.

[101] Hao Peng, Morteza Ghasri-Khouzani, Shan Gong, Ross Attardo, Pierre Ostiguy, Ronald B. Rogge, Bernice Aboud Gatrell, Joseph Budzinski, Charles Tomonto, Joel Neidig, M. Ravi Shankar, Richard Billo, David B. Go, and David Hoelzle. “Fast prediction of thermal distortion in metal powder bed fusion additive manufacturing: Part 2, a quasi-static thermo-mechanical model”. In: *Additive Manufacturing* 22 (Aug. 2018), pp. 869–882. ISSN: 2214-8604. DOI: 10.1016/j.addma.2018.05.001.

[102] Nico Pietroni, Marco Tarini, and Paolo Cignoni. "Almost Isometric Mesh Parameterization through Abstract Domains". In: *IEEE Transactions on Visualization and Computer Graphics* 16.4 (July 2010), pp. 621–635. ISSN: 1077-2626. DOI: 10.1109/tvcg.2009.96.

[103] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software* 3 (2009).

[104] Sebastián Ramírez. *FastAPI*. URL: https://github.com/fastapi/fastapi (visited on 01/27/2025).

[105] Jakob Rehof. "Towards Combinatory Logic Synthesis". In: *1st International Workshop on Behavioural Types, BEAT*. 2013.

[106] Jakob Rehof and Pawel Urzyczyn. "Finite Combinatory Logic with Intersection Types". In: *Typed Lambda Calculi and Applications*. Vol. 6690. Lecture Notes in Computer Science. Berlin, Germany: Springer, 2011, pp. 169–183. DOI: 10.1007/978-3-642-21691-6_15.

[107] F. Reuleaux and A.B.W. Kennedy. *The Kinematics of Machinery: Outlines of a Theory of Machines*. New York, United States: Dover Publications, 1876. ISBN: 9780598975379.

[108] ROBOTIS. *XL430-W250-T*. URL: https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/ (visited on 09/14/2021).

[109] ROS-Industrial. *Introduction to URDF*. Dec. 7, 2024. URL: https://industrial-training-master.readthedocs.io/en/humble/_source/session3/0-Intro-to-URDF.html (visited on 12/07/2024).

[110] Baltej Singh Rupal, Nabil Anwer, Marc Secanell, and Ahmed Jawad Qureshi. "Geometric Tolerance Characterization of Laser Powder Bed Fusion Processes Based on Skin Model Shapes". In: *Procedia CIRP* 92 (2020), pp. 169–174. ISSN: 2212-8271. DOI: 10.1016/j.procir.2020.05.185.

[111] Tristan Schäfer, Jim A. Bergmann, Rafael Garcia Carballo, Jakob Rehof, and Petra Wiederkehr. "A Synthesis-based Tool Path Planning Approach for Machining Operations". In: *Procedia CIRP* 104 (2021), pp. 918–923. ISSN: 2212-8271. DOI: 10.1016/j.procir.2021.11.154.

[112] Tristan Schäfer, Jan Bessai, Constantin Chaumet, Jakob Rehof, and Christian Riest. “Design Space Exploration for Sampling-Based Motion Planning Programs with Combinatory Logic Synthesis”. In: *Algorithmic Foundations of Robotics XV*. Springer International Publishing, Dec. 2022, pp. 36–51. DOI: 10.1007/978-3-031-21090-7_3.

[113] John Schmelzle, Eric V. Kline, Corey J. Dickman, Edward W. Reutzel, Griffin Jones, and Timothy W. Simpson. “(Re)Designing for Part Consolidation: Understanding the Challenges of Metal Additive Manufacturing”. In: *Journal of Mechanical Design* 137.11 (Oct. 2015). ISSN: 1528-9001. DOI: 10.1115/1.4031156.

[114] Ari Seff, Yaniv Ovadia, Wenda Zhou, and Ryan P. Adams. *SketchGraphs: A Large-Scale Dataset for Modeling Relational Geometry in Computer-Aided Design*. 2020. DOI: 10.48550/ARXIV.2007.08506.

[115] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. “Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks”. In: *Genome Research* 13.11 (Nov. 2003), pp. 2498–2504. ISSN: 1088-9051. DOI: 10.1101/gr.1239303.

[116] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Berlin, Germany: Springer London, Limited, 2008. ISBN: 9783540303015.

[117] Daniel Sieger, Pierre Alliez, and Mario Botsch. “Optimizing Voronoi Diagrams for Polygonal Finite Element Computations”. In: *Proceedings of the 19th International Meshing Roundtable*. Springer Berlin Heidelberg, 2010, pp. 335–350. ISBN: 9783642154140. DOI: 10.1007/978-3-642-15414-0_20.

[118] Daniel Sieger and Mario Botsch. *The Polygon Mesh Processing Library*. Version 3.0.0. Aug. 2023. DOI: 10.5281/zenodo.10866532. URL: https://github.com/pmp-library/pmp-library.

[119] Harshad Srinivasan. “Automated Model Processing and Localization of AdditivelyManufactured Parts for Finish Machining”. PhD thesis. North Carolina State University, May 3, 2016.

[120] Minhyuk Sung, Hao Su, Vladimir G. Kim, Siddhartha Chaudhuri, and Leonidas Guibas. “ComplementMe: weakly-supervised component suggestions for 3D modeling”. In: *ACM Transactions on Graphics* 36.6 (Nov. 2017), pp. 1–12. ISSN: 1557-7368. DOI: 10.1145/3130800.3130821.

[121] The FreeCAD Team. *FreeCAD*. Dec. 16, 2024. URL: https://github.com/FreeCAD/FreeCAD (visited on 12/16/2024).

[122] Tiedemann & Betz GmbH & Co. KG. *Brochure Tiedemann FUJIFILM Prescale*. Apr. 9, 2022. URL: https://tiedemann-instruments.de/fileadmin/Downloads/englische_medien/Brochure_Tiedemann_FUJIFILM_Prescale.pdf (visited on 02/19/2025).

[123] Syed A.M. Tofail, Elias P. Koumoulos, Amit Bandyopadhyay, Susmita Bose, Lisa O'Donoghue, and Costas Charitidis. "Additive manufacturing: scientific and technological challenges, market uptake and opportunities". In: *Materials Today* 21.1 (Jan. 2018), pp. 22–37. ISSN: 1369-7021. DOI: 10.1016/j.mattod.2017.07.001.

[124] Paweł Urzyczyn. "Inhabitation of Low-Rank Intersection Types". In: *Typed Lambda Calculi and Applications*. Springer Berlin Heidelberg, 2009, pp. 356–370. ISBN: 9783642022739. DOI: 10.1007/978-3-642-02273-9_26.

[125] Gokul Varadhan and Dinesh Manocha. "Accurate Minkowski sum approximation of polyhedral models". In: *Graphical Models* 68.4 (July 2006), pp. 343–355. ISSN: 1524-0703. DOI: 10.1016/j.gmod.2005.11.003.

[126] Lucas Vergez, Arnaud Polette, and Jean-Phillipe Pernot. "Automatic CAD Assemblies Generation by Linkage Graph Overlay for Machine Learning Applications". In: *CAD'21 Proceedings*. CAD Solutions LLC, May 2021. DOI: 10.14733/cadconfp.2021.46-50.

[127] Hui Wang, Yiming (Kevin) Rong, Hua Li, and Price Shaun. "Computer aided fixture design: Recent research and trends". In: *Computer-Aided Design* 42.12 (Dec. 2010), pp. 1085–1094. ISSN: 0010-4485. DOI: 10.1016/j.cad.2010.07.003.

[128] Xilu Wang, Yaochu Jin, Sebastian Schmitt, and Markus Olhofer. "Recent Advances in Bayesian Optimization". In: (June 7, 2022). DOI: 10.48550/ARXIV.2206.03301. arXiv: 2206.03301 [cs.LG].

[129] Xue Wang, Liping Zhao, Jerry Ying Hsi Fuh, and Heow Pueh Lee. "Experimental characterization and micromechanical-statistical modeling of 316L stainless steel processed by selective laser melting". In: *Computational Materials Science* 177 (2020), p. 109595. ISSN: 09270256. DOI: 10.1016/j.commatsci.2020.109595.

[130] Y. Wang, K. Xu, J. Li, H. Zhang, A. Shamir, L. Liu, Z. Cheng, and Y. Xiong. "Symmetry Hierarchy of Man-Made Objects". In: *Computer Graphics Forum* 30.2 (Apr. 2011), pp. 287–296. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2011.01885.x.

[131] Yin Wang, Yukai Chen, Chuyue Wen, Ke Huang, Zhen Chen, Bin Han, and Qi Zhang. "The process planning for additive and subtractive hybrid manufacturing of powder bed fusion (PBF) process". In: *Materials & Design* 227 (Mar. 2023), p. 111732. ISSN: 0264-1275. DOI: 10.1016/j.matdes.2023.111732.

[132] Karl D. D. Willis, Pradeep Kumar Jayaraman, Hang Chu, Yunsheng Tian, Yifei Li, Daniele Grandi, Aditya Sanghi, Linh Tran, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. *JoinABLe: Learning Bottom-up Assembly of Parametric CAD Joints*. Nov. 24, 2021. DOI: 10.48550/ARXIV.2111.12772. arXiv: 2111.12772 [cs.LG].

[133] Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. "Fusion 360 gallery: a dataset and environment for programmatic CAD construction from human design sequences". In: *ACM Transactions on Graphics* 40.4 (Aug. 2021), pp. 1–24. DOI: 10.1145/3450626.3459818.

[134] Karl D.D. Willis, Pradeep Kumar Jayaraman, Hang Chu, Yunsheng Tian, Yifei Li, Daniele Grandi, Aditya Sanghi, Linh Tran, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. "JoinABLe: Learning Bottom-up Assembly of Parametric CAD Joints". In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2022, pp. 15828–15839. DOI: 10.1109/cvpr52688.2022.01539.

[135] Jan Winkels, Julian Graefenstein, Tristan Schäfer, David Scholz, Jakob Rehof, and Michael Henke. "Automatic Composition of Rough Solution Possibilities in the Target Planning of Factory Planning Projects by Means of Combinatory Logic". In: *ISoLA (4)*. Vol. 11247. Lecture Notes in Computer Science. Springer, 2018, pp. 487–503. DOI: 10.1007/978-3-030-03427-6_36.

[136] Zhijie Wu, Xiang Wang, Di Lin, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. "SAGNet: structure-aware generative network for 3D-shape modeling". In: *ACM Transactions on Graphics* 38.4 (July 2019), pp. 1–14. ISSN: 1557-7368. DOI: 10.1145/3306346.3322956.

[137] Evren Yasa, Gökçe Mehmet Ay, and Anıl Türkseven. "Tribological and mechanical behavior of AISI 316L lattice-supported structures produced by laser powder bed fusion". In: *The International Journal of Advanced Manufacturing Technology* 118.5-6 (2022), pp. 1733–1748. ISSN: 0268-3768. DOI: 10.1007/s00170-021-08069-0.

[138] Qingnan Zhou. *PyMesh*. 2025. URL: https://github.com/PyMesh/PyMesh (visited on 02/10/2025).

[139] Qingnan Zhou and Alec Jacobson. "Thingi10K: A Dataset of 10,000 3D-Printing Models". In: (May 16, 2016). DOI: 10.48550/ARXIV.1605.04797. arXiv: 1605.04797 [cs.GR].

[140] Zuowei Zhu, Nabil Anwer, and Luc Mathieu. "Deviation Modeling and Shape Transformation in Design for Additive Manufacturing". In: *Procedia CIRP* 60 (2017), pp. 211–216. ISSN: 2212-8271. DOI: 10.1016/j.procir.2017.01.023.

[141] Zuowei Zhu, Nabil Anwer, and Luc Mathieu. "Shape Transformation Perspective for Geometric Deviation Modeling in Additive Manufacturing". In: *Procedia CIRP* 75 (2018), pp. 75–80. ISSN: 2212-8271. DOI: 10.1016/j.procir.2018.04.038.