Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Automatic Generation of Mechanical Models for Machine Learning Systems

Scientific Students' Association Report

Author:

Balázs Kovacsics

Advisor:

Richárd Szabó
dr. Oszkár Semeráth

2022

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

1.1 Context 1

1.2 Problem statement 1

1.3 Objectives 2

1.4 Contribution 2

1.5 Added value 2

2 Preliminaries **3**

2.1 Running example 3

2.2 Manufacturing steps 3

2.2.1 Turning and boring 4

2.2.2 Milling 4

2.2.3 Drilling 5

2.2.4 Sawing 6

2.3 Computer-aided design 6

2.3.1 Geometric representations 6

2.3.2 Boundary representation 7

2.3.3 Modeling algorithms 8

2.3.4 CAD exchange formats 8

2.4 Design space exploration 9

2.5 Related works 9

3 Overview of the approach **11**

3.1 Defining the generation method 11

3.2 Generation process 12

3.3 Architecture 12

4 The geometric model generator **14**
4.1 Representation of manufacturing steps . . . 14
4.1.1 Turning and boring . . . 14
4.1.2 Milling . . . 14
4.1.3 Drilling . . . 15
4.1.4 Sawing . . . 16
4.2 Implementation . . . 16
4.2.1 Entities . . . 17
4.2.2 Operations . . . 18
4.2.3 Random variables . . . 19
4.2.4 Execution engine . . . 19
4.3 Limitations . . . 21

5 The structural model generator **22**
5.1 Defining the metamodel of the blueprints . . . 22
5.2 Implementation . . . 23
5.2.1 Transformation rules . . . 23
5.2.2 Interfacing with the geometric model generator . . . 23
5.2.3 State coding . . . 23
5.3 Exploration strategies . . . 24
5.4 Defining the model fitness . . . 24

6 Evaluation **25**
6.1 Performance measurements . . . 25
6.1.1 RQ1-2: Distribution of execution time and scalability . . . 25
6.1.2 RQ3: Evaluation of different strategies . . . 26
6.2 Interfacing with expert systems . . . 28

7 Conclusion **29**

List of Figures **30**

Bibliography **30**

Appendix **33**
A.1 Blueprint metamodel . . . 33

Kivonat

A számítógéppel támogatott tervezést és gyártást (CAD és CAM) évtizedek óta széles körben alkalmazzák a gépészeti alkatrészek termék-életciklusa folyamán, a 3D gépészeti modellek és műszaki rajzok készítésétől, a gyártástechnológiai módszerek meghatározásán keresztül, a CNC megmunkáló gépek programkódjának generálásáig. Ez a folyamat tipikusan magas szintű mérnöki tudást igényel, bár manapság vannak törekvések arra, hogy ezt a folyamatot gépi tanulás alapú intelligens rendszerekkel támogassák.

Mivel a mérnöki munkaórák költségesek, a gyárthatósági elemzés és költségbecslés maga is egy jelentős fix költséggel jár, ezért, ha az alkatrész gyártása az aktuális tervek alapján kivitelezhetetlennek vagy gazdaságtalannak bizonyul, akkor ez a költség teljes mértékben veszteséggé válik. Következésképpen ennek a folyamatnak a részleges kiváltása automatizált eszközökkel jelentősen növelni tudja a produktivitást, és csökkenteni a költségeket. Azonban ilyen komponensek tanítása során több nehézséggel is szembekerülünk. Először is hiányoznak szabadon rendelkezésre álló gépészeti tervrajzok, mivel nagy részük ipari titok. A különböző CAD/CAM szoftverek specializált és inkompatibilis adatformátummal és modellezési stílussal rendelkeznek. Legvégül az alulreprezentált és kiegyensúlyozatlan tanító adathalmazok miatt ezen eszközök használata sok esetben few-shot/zero-shot learning feladattá válik: ekkor a MI komponensnek olyan bemenetre kell választ adnia, amelyhez hasonló nem, vagy csak kis mennyiségben szerepelt a tanítóhalmazban.

A dolgozat célja egy nagy mennyiségű, változatos és/vagy valósághű adathalmazt előállítani képes módszer kidolgozása, amely használható tanítóhalmazként a költségbecslésre és megmunkálási lépésekkel kapcsolatos javaslattételre képes intelligens rendszerek számára, valamint ezen rendszerek validálására, mérésére és összehasonlító elemzésére.

A dolgozatban bemutatom (i) a gépészeti modellek generálásához használt módszert, (ii) a fent említett intelligens rendszerek tanítására, validálására és összehasonlítására használható stratégiákat, (iii) lehetséges generálási módszerek összehasonlító elemzését, valamint (iv) a generálási módszer használatát egy létező, ipari prototípus szakértői rendszer vizsgálatára.

A dolgozat eredménye egy mesterségesen generált gépészeti modelleket előállító működő prototípus keretrendszer, mely használható a költségbecslő eszközök tanítóhalmazának gazdagítására, valamint ezen eszközök tesztelésére, validálására, mérésére és összehasonlító elemzésére, melynek eredményeképpen jobb minőségű gyártási költség becslésére képes eszközök fejleszthetők.

Abstract

Computer-aided design and manufacturing (CAD/CAM) have been extensively used in the manufacturing industry for decades throughout the lifecycle of mechanical parts, from creating 3D mechanical models and manufacturing drawings, to making decisions about manufacturing methods and even generating the programming of CNC machines. This process traditionally requires significant engineering expertise, but there have been recent efforts to support this process with the use of machine learning-based intelligent systems.

As engineering man-hours are typically expensive, manufacturability analysis and cost estimation incur a significant fixed cost, which might even turn into a complete loss if manufacturing the current design proves to be infeasible or simply uneconomical. Therefore, substituting a part of this process with an automated tool can greatly improve productivity and reduce costs. However, due to the scarcity of data (most of it being proprietary), the specialized and incompatible data formats and modeling styles used by different CAD/-CAM software, and underrepresented and imbalanced training data, the use of these tools frequently results in the problem of few-shot/zero-shot learning.

The objective of this report is to propose a method for providing a high amount of diverse and/or realistic data, to be used as learning input by such intelligent systems used for cost estimation and advising about manufacturing steps, and also to validate, benchmark and compare these systems.

In the report, I am presenting (i) a general overview of how the mechanical models are being generated, (ii) strategies for using the data to train, validate and compare such intelligent systems, (iii) an evaluation of multiple possible generation methods and (iv) using the generating method for evaluating an existing prototype industrial expert system.

The outcome of the report is a working prototype system producing artificially generated mechanical models, usable for enriching the training data of estimator tools, and for the testing, validation, benchmarking and comparison of such tools, ultimately resulting in better quality tooling used for manufacturing cost estimation.

Chapter 1

Introduction

1.1 Context

Computer-aided design and manufacturing (CAD/CAM) have been extensively used in the manufacturing industry in the last decades. Those models are used in the complete lifecycle of the design and manufacturing mechanical parts:

1. from cost estimation
2. to making decisions about manufacturing methods,
3. through the 3D modeling of the mechanical object
4. and manufacturing drawings,
5. and even generating the programming of CNC machines.

This process traditionally requires significant engineering expertise, but there have been recent efforts to support this process with the use of machine learning-based intelligent systems.

1.2 Problem statement

As engineering man-hours are typically expensive, manufacturability analysis and cost estimation incur a significant fixed cost. This can be a complete loss if manufacturing the current design proves to be infeasible or simply uneconomical. Therefore, the early detection of infeasible or costly task can be critical in such processes. Thus substituting a part of this process with an automated tool can greatly improve productivity and reduce costs.

However, most of the existing design data is proprietary, which makes the number of existing models scarce. This is especially problematic for machine learning approaches, which require a large number of models for training. Moreover, the specialized and incompatible data formats and modeling styles used by different CAD/CAM software, and underrepresented and imbalanced training data, the use of these tools frequently results in the problem of few-shot/zero-shot learning.

1.3 Objectives

The objective of this report is to propose a method for providing a high amount of diverse and/or realistic data, to be used as learning input by such intelligent systems used for cost estimation and advising about manufacturing steps, and also to validate, benchmark and compare these systems.

1.4 Contribution

In the report, I am presenting:

- An approach which can automatically derive a large number of mechanical blueprint usable for testing and training machine learning components.
- The approach is parameterized with different strategies and metrics for using the data to train, validate and compare such machine learning systems.
- I evaluate my approach of multiple possible generation methods.

1.5 Added value

The outcome of the report is a working prototype system producing artificially generated mechanical models, usable for enriching the training data of estimator tools, and for the testing, validation, benchmarking and comparison of such tools, ultimately resulting in better quality tooling used for manufacturing cost estimation. Additionally, such blueprint generation can greatly help the testing of design tools.

Chapter 2

Preliminaries

2.1 Running example

In the following example, we're going to examine the execution of an internal turning operation, performed on a cylindrical body. In the case of example A, the open curve used as the input of this operation is a straight line that goes inward from the front face of the cylinder, towards the symmetry axis. This results in a valid turning operation, where the output is still a single body. In example B, on the other hand, the line used as the input of the turning operation goes outward from the cylinder face, and even crosses the outline of the cylindrical surface. As a result, after performing the turning operation, the body falls apart into two pieces, which is not considered valid by the generator (figure 2.1).

After examining the validity of the results, let's look at the complexity of the valid one. If our complexity metric is defined by the ratio of surface area to volume, this operation clearly increases this metric, as the surface increased, but the volume decreased as a result of performing the operation.

2.2 Manufacturing steps

As the purpose of this work is to generate mechanical models similar to ones used in a conventional manufacturing process, only machining steps were considered, additive manufacturing (3D printing) is out of scope for the current implementation. The following machining operations were selected for usage in the model generator:

- Turning and boring
- Milling
- Drilling
- Sawing

The following section provides a quick overview of the properties of these machining operations [22].

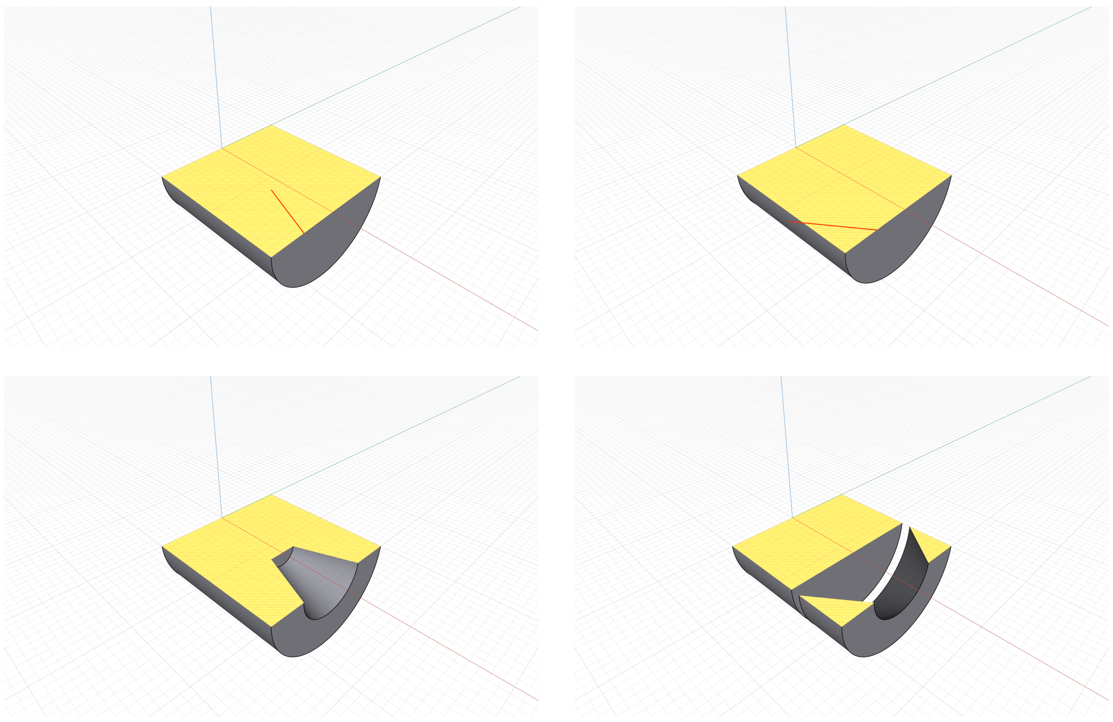


Figure 2.1: Example A (left) shows the execution of a valid turning operation, and example B (right) shows an invalid one, both displayed in section view

2.2.1 Turning and boring

Turning and boring are machining operations performed with a single-edged cutting tool. Turning usually refers to working on the external surface of the workpiece, while boring is performed on an internal surface, usually created by a preceding drilling operation. Apart from this distinction, they have the same basic setup: the primary rotating motion is performed by the workpiece, and the feed motion is executed by the tool, resulting in a machined feature with axial symmetry. As the workpiece is rotated at high speed, to ensure precision, typically the initial raw workpiece is also symmetric, or at least balanced around the axis of rotation (having e.g. a hole drilled through the center, or slots milled in a circular pattern, created in a previous operation (figure 2.2)). A special, significantly more challenging use case of this operation is the machining of thin-walled workpieces, which has widespread usage in the aerospace industry[9][13], therefore covering this case in the generation method increases both the diversity and the potential complexity of the resulting models.

2.2.2 Milling

Milling is a machining operation performed with a multi-edged cutting tool. The primary rotating motion is performed by the tool, while the feed motion can be executed by either the tool or the workpiece (or in some cases, especially in multi-axis CNC machines, by both). Based on the number of possible axes of the feed motion (irrelevant of whether it is performed by the tool or the workpiece), milling operations can be grouped into the following categories:

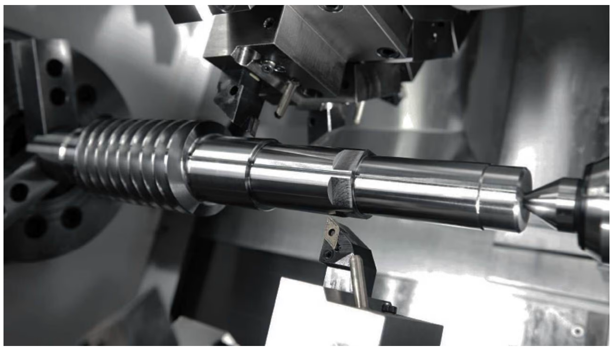


Figure 2.2: A complex workpiece machined by turning, with pre-existing milled surfaces in a hexagonal pattern

Source: `https://www.intermacheng.co.uk/cnc-machining/cnc-turning/`

- *2D milling:* the feed motion can be performed on the two axes perpendicular to the rotational axis of the tool. This kind of milling can only be used to machine planar-faced features.
- *2.5D milling:* in addition to the above, feed motion is also possible along the rotational axis of the tool, but cutting can not be performed along it simultaneously with the other two axes. This makes the creation of closed pockets and the rough machining of slanted surfaces possible (figure 2.3).
- *3D milling:* the feed motion can be performed on all translational axes. This enables the creation of any surface which is accessible from a single direction, i.e. one that doesn't have any overhanging parts (these are also called 2.D surfaces).
- *5D milling:* the feed motion can be performed on all translational axes, and the two rotational axes perpendicular to the axis of primary rotation. This enables the creation of arbitrarily complex surfaces, and also makes it possible to optimize the cutting angles.

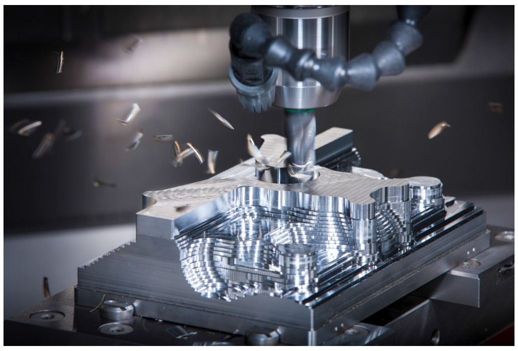


Figure 2.3: A rough milling performed on a workpiece, with visibly distinct Z-steps

Source: `https://www.solidcam.com/imachining/imachining-3d`

2.2.3 Drilling

Drilling is a machining operation where both the primary rotating motion, and the feed motion, which is parallel to the axis of rotation, are performed by a multi-edged cutting tool. This setup makes it possible to create, or enlarge existing cylindrical and conical holes in the workpiece. Drilled holes can be either blind holes, or through holes, depending on whether they exit on the other end of the workpiece. A cylindrical enlargement of the entrance of a hole is called counterboring, while a conical enlargement is called countersinking, both of which are performed frequently on holes created to accommodate screws.

2.2.4 Sawing

Sawing is performed with narrow multi-edged cutting tools, which makes it possible to cut the workpiece with only a minimal amount of material waste. In CNC sawing machines, the typically used tool is a band saw, which consists of a continuous loop of metal band moving in a single direction. In these machines, the feed motion is usually performed by the workpiece.

2.3 Computer-aided design

Mechanical CAD software are based on geometric modeling kernels, which provide the geometric data representation and low-level modeling algorithms. The modeling operations and other tools available to the user in the CAD software are an abstraction layer over the kernel, implemented as combinations of modeling algorithms[11].

2.3.1 Geometric representations

The basic principles of geometric representations used by most solid modeling kernels belong to one of the following [11]:

- *Constructive solid geometry (CSG):* solid bodies are created from geometric primitive bodies (typically cuboids, prisms, pyramids, cylinders, cones and spheres) using Boolean operations. This method guarantees that the resulting geometries will always be valid, but finding the correct combination of operations for complex geometries can be difficult, or even impossible in some cases.
- *Boundary representation (B-rep):* geometric entities are defined with explicit mathematical descriptions, and topological entities are defined with these geometric entities bounded by topological entities of a lower dimension. This method provides an easier process for defining complex geometries than CSG, and a wider range of construable geometry, but the results are not guaranteed to be valid, thus extensive checking, and in some cases repairing of geometries is needed.
- *Implicit modeling:* solid volumes are defined by a signed distance function, with negative values representing points inside the given volume. This approach, like CSG, also provides strong validity guarantees, but interacting directly with implicit models is not simple, so it's mostly used for refining models created with a different method (usually B-rep)[4].

As geometric modeling kernels are immensely complex pieces of software, implementing one from scratch for the model generator was not an option. Most modern CAD software

are based on B-rep kernels, and the STEP format also uses it for the representation of geometric data. The most stable and feature-rich open source B-rep kernel is OpenCascade, making it the ideal choice (of free alternatives) for implementing the model generator [1]. As the OpenCascade library is written in C++, the language was also used for implementing the model generator, enabling native interfacing with the library.

2.3.2 Boundary representation

The B-rep implementation of the OpenCascade library uses the following geometric and topological entity types[1].

Topological entities:

- *Vertex:* a 0D topology positioned at a geometric point
- *Edge:* a piece of a geometric curve, bounded by vertices at its ends
- *Wire:* a sequence of edges connected by their vertices
- *Face:* a part of a geometric surface, bounded by wires
- *Shell:* a collection of faces connected by edges of their wire boundaries
- *Solid:* a finite closed part of 3D space bounded by shells
- *Compound solid:* a collection of solids connected by faces of their shell boundaries

Geometric curves:

- *Line*
- *Conics:* circle, ellipse, hyperbola, parabola
- *Bounded curves:* B-spline and Bezier curves
- *Offset curve:* a curve at constant distance from a basis curve in a direction perpendicular to both its tangent and a reference direction

Geometric surfaces:

- *Elementary surfaces:* plane, cylindrical, conical, spherical, toroidal
- *Bounded surfaces:* B-spline and Bezier surfaces
- *Offset surface:* a surface at constant distance from a basis surface in the direction of its normal
- *Swept surfaces:* surfaces constructed by sweeping a curve with another curve, special cases are surfaces of a linear extrusion and those of a revolution

To preserve consistency, the library does not provide direct access to the data stored in these entities, they can only be interacted with through the use of modeling algorithms.

2.3.3 Modeling algorithms

The OpenCascade library provides several ways to create and modify geometries and topologies[1]. The lower-level APIs enable the direct creation of geometric entities (`Geom*` classes), and piecewise bottom-up building of topological ones (`BRepBuilderAPI_*` classes). High-level APIs provide the following functionalities:

Creating primitive topologies:

- Boxes
- Cylinders
- Cones
- Spheres
- Toruses

Kinematic modeling operations:

- Prisms (sweeping a curve along a straight path)
- Revolutions (sweeping a curve around an axis)
- Pipes (general swept bodies)
- Lofts (interpolation between a series of cross-sections)

Boolean operations:

- Common (intersection)
- Fuse (union)
- Cut (subtraction)

Local modifications:

- Shelling (removing a set of faces and creating a thick-walled body from the remaining ones)
- Tapering faces
- Chamfering and filleting edges

2.3.4 CAD exchange formats

Each CAD software has its own data representation formats, which are generally proprietary in case of commercial software. In addition to the geometric data itself, these may also contain information about the design history, parametric formulas, geometric dimensioning and tolerancing (GD&T), and other kinds of metadata [17]. Even if the geometric data format is known, different CAD systems may represent the same entities in incompatible ways, especially if they are not based on the same geometric modeling kernel. To make the data exchange between these systems possible, several CAD-neutral data formats exist, which are supported by most industrial CAD software [25]. Some proprietary software have their own open exchange formats, like Autodesk's DXF [2]. There are also standardized exchange formats, the older, no longer maintained and nowadays seldom used ANSI standard IGES [21], and the ISO standard STEP format [7]. STEP supports several different so-called application protocols (APs) for the exchange of different data types. The recently developed AP242 unifies several older protocol versions, providing a way to exchange both geometric data and several kinds of metadata[6].

Based on these considerations, a data exchange format had to be chosen for the model generator as well, to enable the conversion of its internal geometric representation into one which provides interoperability with other CAD systems. The OpenCascade library supports both IGES and STEP formats[1], of which STEP was selected to be used for the model generator.

2.4 Design space exploration

Design space exploration (DSE) is a technique of searching for solutions within the possible design alternatives, which meet a specific set of criteria. For the model generator, the VIATRA-DSE library was selected to be used, which has the following properties [3]:

- *Model-driven:* the problem is represented as a typed attributed graph, which is stored as an Eclipse EMF[23] model, and constraints are defined with VIATRA graph query patterns.
- *Rule-based:* the model is modified according to graph transformation rules, and solutions are defined as a sequence of rule applications (called a trajectory), that transform the model from the initial state into the desired state.
- *Multi-Objective Optimization:* multiple objectives can be defined for the exploration, which can either be hard (which must be satisfied by a goal state) or soft (which should be optimized). Objectives can be derived from the model or from the trajectory.
- *Meta-heuristic strategies:* these techniques are widely used in optimization problems. VIATRA-DSE contains multiple built-in exploration strategies, such as depth and breadth first search, fixed-priority strategy, hill climbing strategy, and evolutionary algorithms. It is also possible to integrate custom-defined strategies.

2.5 Related works

Related work is based on the [19] paper.

Hybrid approaches divide the model generation into multiple sub-tasks and use a different underlying technique for resolving each one. The PLEDGE model generation tool [20] combines metaheuristic search for model structure generation with an SMT-solver based implementation for handling attributes, providing a scalable implementation. The Evacon tool [12] generates tests for object-oriented programs by implementing search-based evolutionary testing, followed by symbolic execution. Autograph [16] is using a sequential combination of a tableau-based approach for generatiing the model structure, with an SMT-solver based approach for handling attributes. These approaches combine the techniques sequentially, which restricts mutually dependent structural and numerical constraints. Additionally, these techniques do not assure completeness of model generation. Another category of hybrid approaches involves assessing multiple components of the model generation task in parallel. This requires the implementation of a certain decision procedure such as DPLL(T) [10, 15] to iterate between underlying techniques, or combine them by sharing variables in their proofs [14]. Such decision procedures are presented alongside their associated properties (e.g. soundness and completeness) at an abstract level in [15, 8], which allows for formal reasoning about their implementations. However, those approaches handle graph-based models inefficiently [24, 18], thus the scalability of those techniques is limited.

Chapter 3

Overview of the approach

3.1 Defining the generation method

To ensure that the generated mechanical models are sufficiently complex and diverse, but at the same time realistic, a well-defined generation method had to be determined. Initially, a declarative approach of using mathematical constraints to define the geometries was considered, but ultimately rejected, as it was too difficult to find precise mathematical definitions for all but the most simple mechanical components. The chosen approach is a step-by-step procedural generation of the CAD models, where each step is a relatively accurate representation of real-life manufacturing methods.

In summary, the system is parameterized with the following:

- possible modeling steps (operations)
- geometric and structural validity conditions
- complexity requirement for the results
- model size constraints
- desired number of distinct results

Within the constraints defined by these parameters, the system executes a series of modeling steps, which results in a set of valid and sufficiently complex mechanical models (figure 3.1).

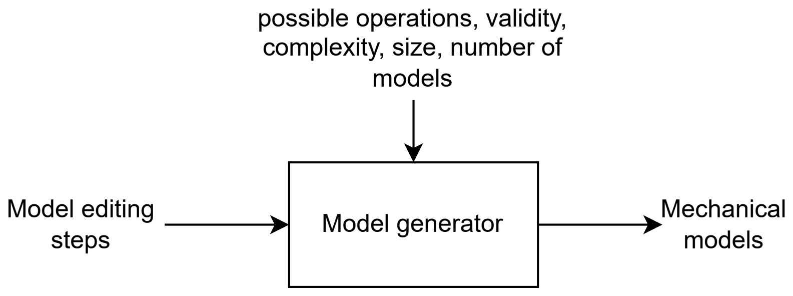

possible operations, validity, complexity, size, number of models
Model editing steps
Model generator
Mechanical models

Figure 3.1: A high-level overview of the model generator

3.2 Generation process

The model generator is split into two parts: a geometric and a structural generator. The role of the structural generator is to create the list of operations for a model (called a blueprint), following a set of rules defined for the blueprint structure. It also makes decisions about the complexity of a model, and if it fulfills the requirement, adds the current blueprint to the set of solutions. The geometric generator is responsible for performing the operations on the geometric model, checking the results for consistency and feasibility, and storing the results of already performed operations for further processing. It also manages the exporting of the geometric models to a CAD exchange format (figure 3.2).

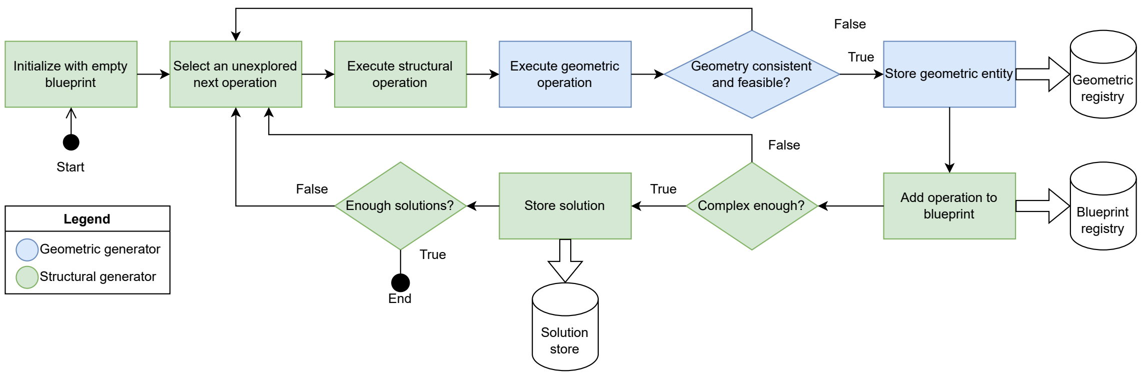


Figure 3.2: Flow of the generation process

3.3 Architecture

The structural generator uses design space exploration (DSE) techniques to search for blueprints that satisfy the defined validity and complexity requirements. This is achieved by representing the blueprint as a graph, where the nodes are the modeling operations, and directed edges designate that the output of an operation is used as the input of the other. A set of rules is created to define the allowed graph transformations on the blueprint in a certain state. These graph transformation and DSE capabilities are provided to the structural generator by the VIATRA library, while the metamodel of the blueprint itself is defined using the Eclipse Modeling Framework (EMF).

The geometric generator uses the OpenCascade geometric modeling kernel to create, modify and examine the generated models. Modeling operations are implemented using a combination of the low-level modeling algorithms provided by the kernel. Geometric validation is performed both by using the built-in model checking functionalities of the kernel (e.g. self-intersection of edges), and examining additional user-defined properties of the results (e.g. an operation resulting in multiple bodies). The generator is also capable of evaluating the same operation structure with randomized input parameters, which already provides a limited capability of generating diverse models in isolation, without a structural generator component.

Both the structural and the geometric generator have external dependencies, which also impose constraints on their implementation. The structural generator is implemented in Java, for compatibility with the EMF and VIATRA libraries, while the geometric generator is implemented in C++ for native interfacing with the OpenCascade library. As a result, the two parts are running in separate processes, which introduces the need for a way to provide interoperability, and an orchestration service. The resulting architecture is shown on figure 3.3.

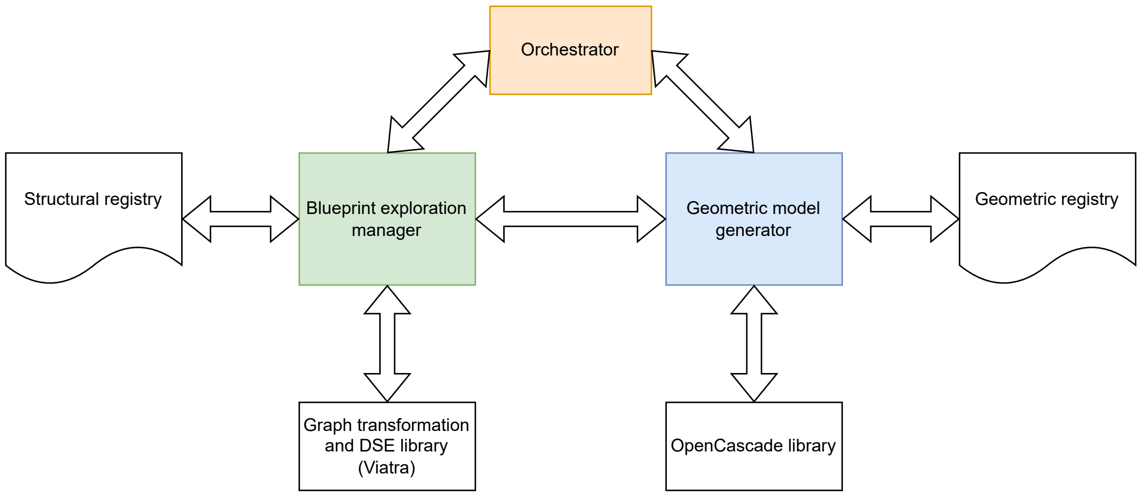

Orchestrator
Structural registry
Blueprint exploration manager
Geometric model generator
Geometric registry
Graph transformation and DSE library (Viatra)
OpenCascade library

Figure 3.3: Architecture of the model generator

Chapter 4

The geometric model generator

Based on the desired properties of the generation method, defined in the previous chapter, the actual capabilities required of the geometric model generator were determined as such:

- *Geometric modeling:* the ability to create and modify geometric entities
- *Evaluation of geometric models:* the ability to evaluate the validity of geometric entities
- *CAD data exchange:* the ability to export these geometric entities in a standard exchange format
- *Generating geometrically diverse models:* the ability to evaluate a series of operations with varying parameters in ways that result in a high amount of complex and diverse mechanical models

4.1 Representation of manufacturing steps

4.1.1 Turning and boring

This operation is implemented in the model generator as an algorithm that receives an axially symmetric solid body, which represents the workpiece, and an open curve representing the path of the cutting tool (figure 4.1). The curve is not allowed to intersect either the axis of rotation or itself, as that would result in the body falling apart into multiple pieces. It is also possible to create a new solid body using an open curve, which is a simplification being equivalent to applying an external turning operation on the entire length of a cylinder (figure 4.2).

To represent the case of thin-wall turning in the generator, a separate operation is implemented for creating symmetric bodies with a constant wall thickness, also using an open curve as input (figure 4.3).

4.1.2 Milling

This operation is implemented as an algorithm that receives a solid body, which represents the workpiece, and a closed curve representing the boundary of the cutting tool path. The removal of material from the solid body can be performed either inside or outside these

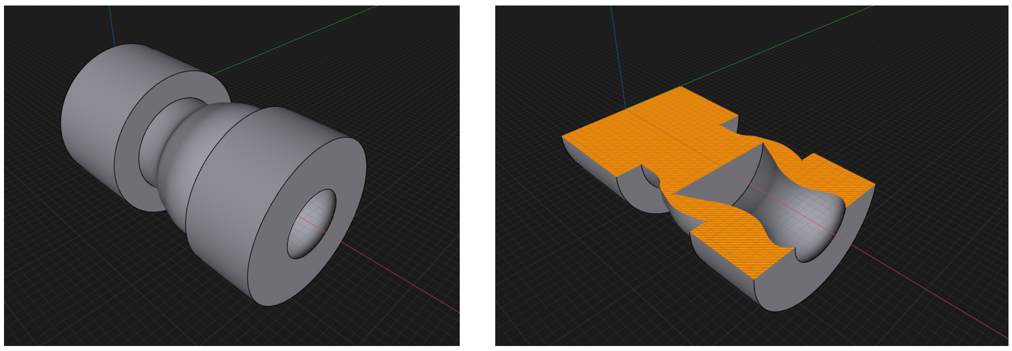


Figure 4.1: A cylinder with external and internal turning (boring) applied, normal and section views

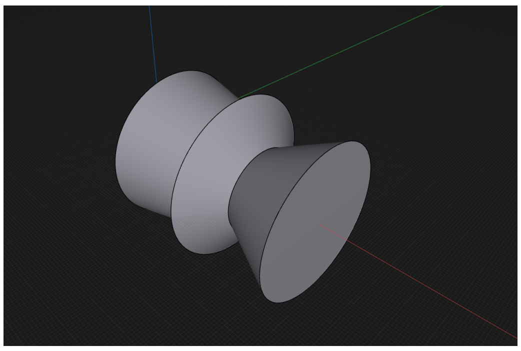


Figure 4.2: A turned solid created directly from an open curve

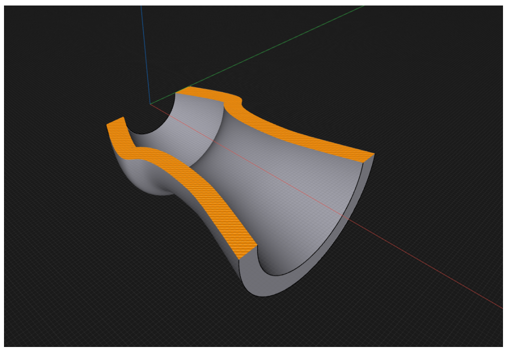


Figure 4.3: A thin-walled turned solid in section view

bounds (figure 4.4). Similarly to the turning operations, it is also possible to create a new solid body using a closed curve, which is a simplification being equivalent to applying an external milling operation on the entire height of a rectangular block (figure 4.5). Applying these operations on a body repeatedly, at different depths, allows the creation of features similar to those possible with 2.5D milling (figure 4.6).

4.1.3 Drilling

The model generator has the capability of creating cylindrical blind or through holes on planar faces. As holes regularly occur in linear or rectangular patterns on real-life mechanical parts, the modeler is also able to mirror these holes along midplanes of the bounding box of the target face (figure 4.7). Extending the capabilities with the creation of

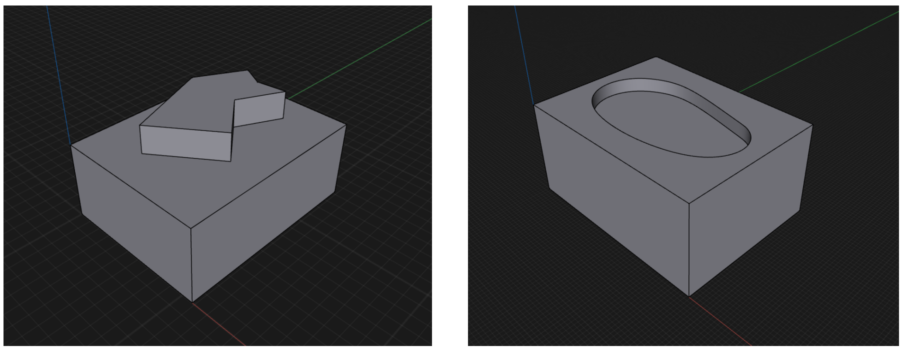


Figure 4.4: A rectangular block with external and internal milling applied

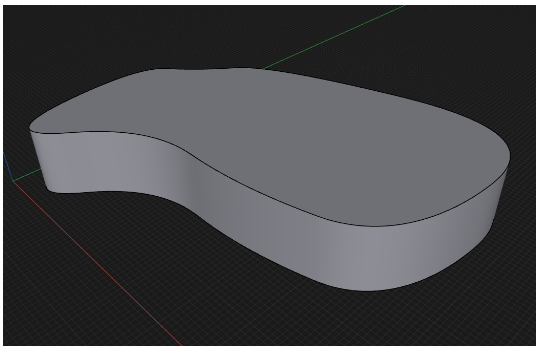


Figure 4.5: A milled solid created directly from a closed curve

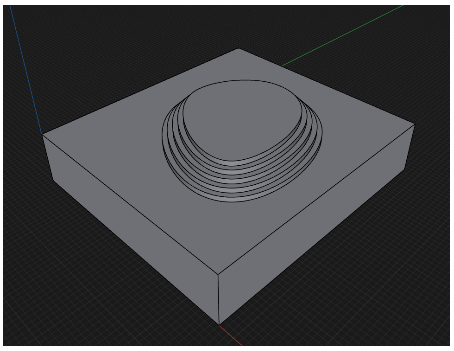


Figure 4.6: Repeated milling operations at different depths, using the same curve scaled to different sizes

counterbored and countersunk holes, and performing the drilling operation on non-planar faces have been considered, but not yet implemented.

4.1.4 Sawing

This function is implemented in the model generator as an operation that is capable of splitting the workpiece with an arbitrary plane, keeping only one of the halves (figure 4.8).

4.2 Implementation

The model generator defines a set of *operations*, which either create or modify *entities*, optionally using other entities as input. These operations can be used to create *execution*

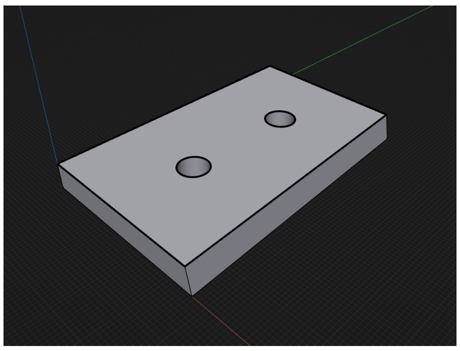


Figure 4.7: A pair of holes created on the top face of the workpiece, mirrored along the midplane parallel with the X axis

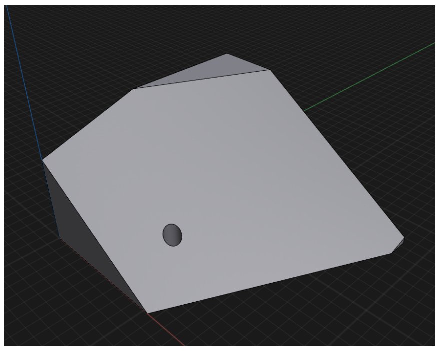


Figure 4.8: A solid block cut with an angled plane

nodes, and a set of these connected nodes defines an *execution graph*. This graph can be evaluated by the *execution engine*, with the final result being a single entity. As the parameters of operations are defined with random variables, evaluating the same graph multiple times will result in structurally similar, but distinct entities.

4.2.1 Entities

The model generator defines different entity types, which are backed by OpenCascade topological entities, extended with additional metadata to determine the operations that are allowed to be performed on them. The following entity types exist in the generator:

- *OpenCurve:* a bounded, open planar curve, represented by the `TopoDS_Wire` topological entity type. It can be used as an input for turning operations.
- *ClosedCurve:* a periodic planar curve, also represented by the `TopoDS_Wire` topological entity type. It can be used as an input for milling operations, and curves can be combined by Boolean operations.
- *SolidBody:* a solid body represented by the `TopoDS_Solid` topological entity type. Contains an optional axis of symmetry member, which can be one of the world axes (X, Y, Z).

4.2.2 Operations

The operations defined in the model generator are either creation or modification operations, and they may or may not expect a second entity as input (interface hierarchy: figure 4.9). The non-entity parameters of operations are random variables of arithmetic, enumeration or Boolean types, which are set at the time of construction, and sampled at the time of execution. The subject and input entities are received as parameters each time the operation is executed. The specific operations implemented in the generator are the following:

- ***Open curve creation:***
  - *CreateLine:* creates a straight line between two points.
  - *CreateArc:* creates an arc of a circle defined by three points.
  - *CreateOpenSpline:* creates a spline of a given degree, with a list of control points as input.
- ***Closed curve creation:***
  - *CreateRectangle:* creates a rectangle with the specified width and height.
  - *CreateCircle:* creates a circle with the specified radius.
  - *CreateClosedSpline:* creates a periodic spline of a given degree, with a list of control points as input.
- ***Closed curve modification:***
  - *BooleanClosedCurve:* creates planar faces from the input curves, applies the specified boolean operation on them, and outputs the outer bound of the resulting face.
- ***Solid body creation:***
  - *CreateBox:* creates a rectangular box with the specified length, width and height.
  - *CreateCylinder:* creates a cylinder with the specified radius, height and axis. The axis of symmetry is stored in the result.
  - *CreateRevolvedBody:* creates a body with a revolve kinematic modeling operation, using an open curve as input.
  - *CreateRevolvedThinWall:* creates a body with a revolve kinematic modeling operation, using an open curve as input, then applies a shelling operation on the result with the specified wall thickness.
  - *CreateExtrudedBody:* creates a body with an extrude kinematic modeling operation, using a closed curve as input.
- ***Solid body modification:***
  - *AddHolesToFace:* Create a cylindrical blind or through hole on a planar face of the body, with the specified radius, depth and clearance (minimum distance from the face bounds). The hole can optionally be mirrored along either of the midplanes of the face bounding box.

- *SplitWithPlane:* Split the body by a plane, specified with an origin point, and degrees of orientation and inclination. The result contains only one half of the split body.
- *ExternalTurning:* Remove material on the external surface of a symmetric body, along a path defined by an open curve.
- *InternalTurning:* Remove material from the inside of a symmetric body, along a path defined by an open curve.
- *ExternalMilling:* Remove material from a body at a specified depth, outside an area defined by a closed curve.
- *InternalMilling:* Remove material from a body at a specified depth, inside an area defined by a closed curve.

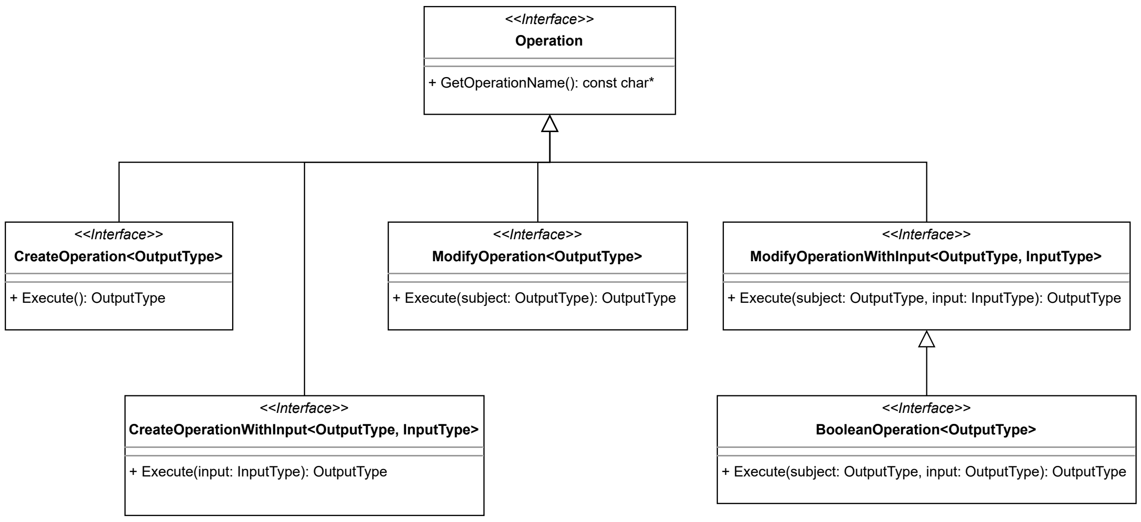

<<Interface>>
Operation
+ GetOperationName(): const char*
<<Interface>>
CreateOperation<OutputType>
+ Execute(): OutputType
<<Interface>>
ModifyOperation<OutputType>
+ Execute(subject: OutputType): OutputType
<<Interface>>
ModifyOperationWithInput<OutputType, InputType>
+ Execute(subject: OutputType, input: InputType): OutputType
<<Interface>>
CreateOperationWithInput<OutputType, InputType>
+ Execute(input: InputType): OutputType
<<Interface>>
BooleanOperation<OutputType>
+ Execute(subject: OutputType, input: OutputType): OutputType

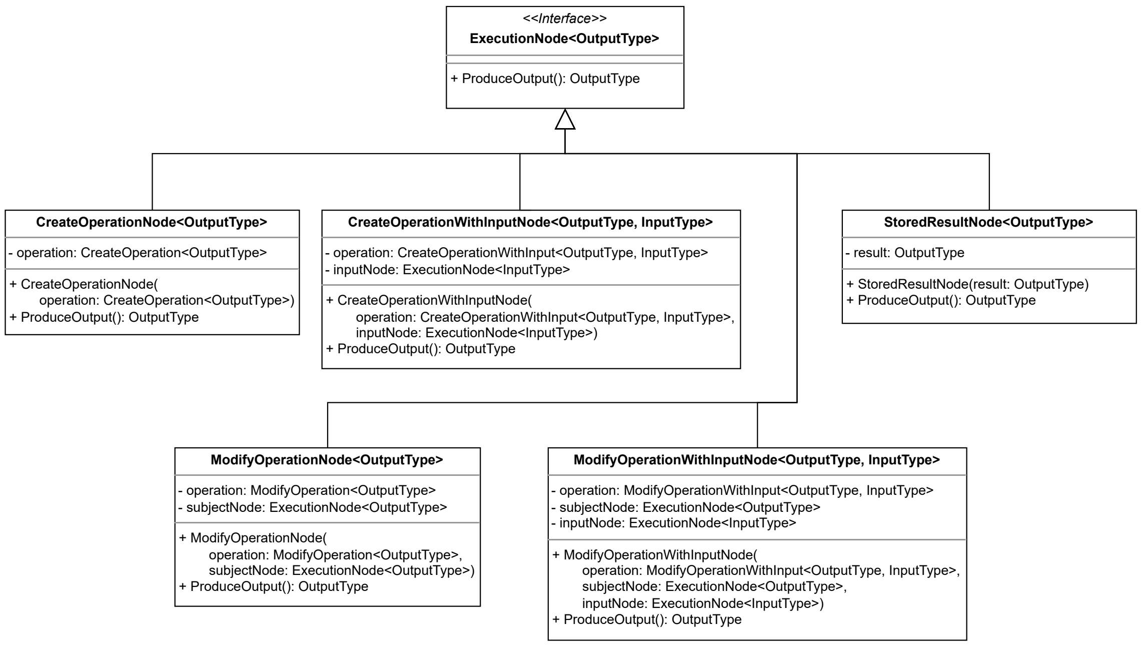
Figure 4.9: The interface hierarchy of operations

4.2.3 Random variables

All non-entity parameters of an operation are random variables, which can be evaluated to different numeric values each time the operation is executed. The data stored in a `RandomVariable` object is either an exact numeric value, returned without modification at the time of each evaluation, or a value distribution defined by the minimum and maximum values of the range, and the distribution type (in the current implementation, this is either a uniform or a Gaussian normal distribution). With an optional parameter, the rounding of floating point types to the nearest integer value can also be enabled. Random boolean and enumeration parameters are derived from the arithmetic types, returning a range of integers between 0 and 1, or the maximum value of the enumeration type, respectively.

4.2.4 Execution engine

The model generator is able to create execution graphs from a series of operations, using `ExecutionNode` objects. Each execution node contains an operation, and references to other execution nodes, which act as the sources of subject and input entities to the operation, if necessary (type hierarchy: figure 4.10). As the source nodes have to be specified at the time of construction, the resulting directed graphs are guaranteed to be acyclic, thus their evaluation can be safely implemented with a simple recursive traversal algorithm. A special type of execution node exists for storing and later returning already generated entities, which can also be used as inputs to other execution nodes. An example of a simple execution graph (figure 4.11), and the resulting entities of repeated evaluations (figure 4.12) are visible below. Note that even though the input of the external and internal turning are the same node, it is re-evaluated each time encountered during the evaluation, resulting in different curves.



<<Interface>>
ExecutionNode<OutputType>
+ ProduceOutput(): OutputType
CreateOperationNode<OutputType>
- operation: CreateOperation<OutputType>
+ CreateOperationNode(
operation: CreateOperation<OutputType>)
+ ProduceOutput(): OutputType
CreateOperationWithInputNode<OutputType, InputType>
- operation: CreateOperationWithInput<OutputType, InputType>
- inputNode: ExecutionNode<InputType>
+ CreateOperationWithInputNode(
operation: CreateOperationWithInput<OutputType, InputType>,
inputNode: ExecutionNode<InputType>)
+ ProduceOutput(): OutputType
StoredResultNode<OutputType>
- result: OutputType
+ StoredResultNode(result: OutputType)
+ ProduceOutput(): OutputType
ModifyOperationNode<OutputType>
- operation: ModifyOperation<OutputType>
- subjectNode: ExecutionNode<OutputType>
+ ModifyOperationNode(
operation: ModifyOperation<OutputType>,
subjectNode: ExecutionNode<OutputType>)
+ ProduceOutput(): OutputType
ModifyOperationWithInputNode<OutputType, InputType>
- operation: ModifyOperationWithInput<OutputType, InputType>
- subjectNode: ExecutionNode<OutputType>
- inputNode: ExecutionNode<InputType>
+ ModifyOperationWithInputNode(
operation: ModifyOperationWithInput<OutputType, InputType>,
subjectNode: ExecutionNode<OutputType>,
inputNode: ExecutionNode<InputType>)
+ ProduceOutput(): OutputType

Figure 4.10: The type hierarchy of execution nodes

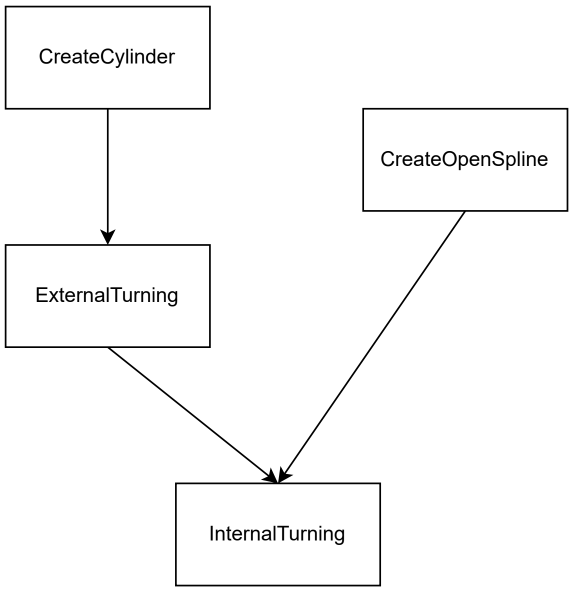


Figure 4.11: An execution graph consisting of a cylinder, and an external and internal turning operation with open splines as inputs

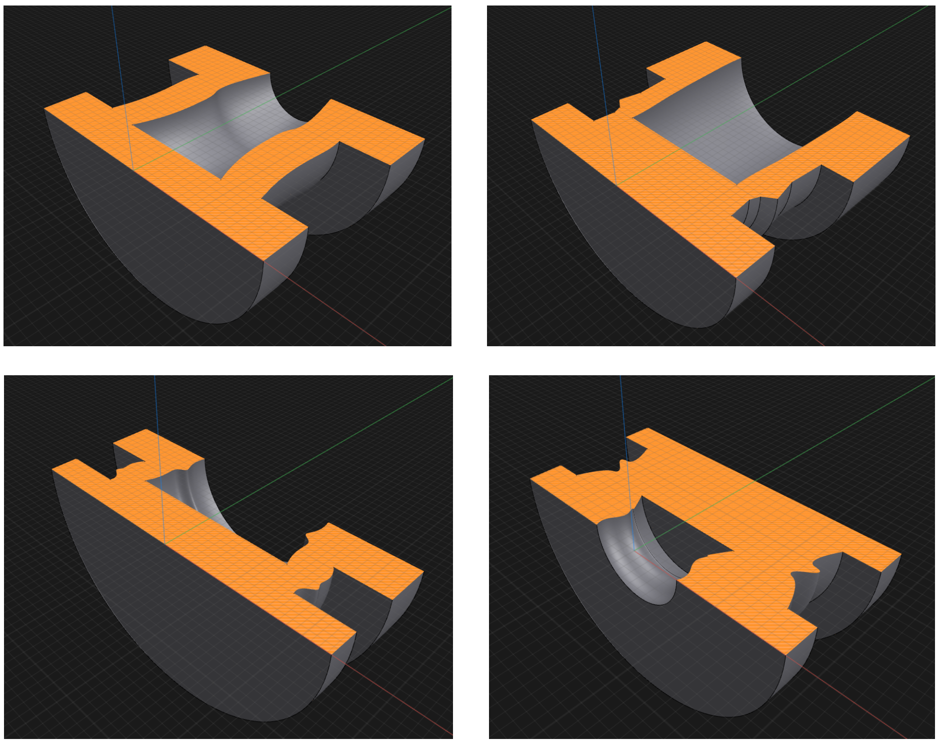


Figure 4.12: The results of repeated evaluations of a single execution graph

4.3 Limitations

As shown in the previous sections, the implemented geometric generator is capable of producing structurally similar solid bodies with varying parameters, partially fulfilling the main goal of the project. However, the automatic generation of structurally diverse models is not possible with this generator alone. To achieve this goal, a structural generator was implemented, which creates diverse blueprint structures, and drives this generator to create and evaluate the geometric models.

Chapter 5

The structural model generator

Based on the desired properties of the generation method, defined earlier, the capabilities required of the structural model generator were determined as such:

- *Structural modeling:* the ability to create and modify the blueprint structure, and realize its geometry with the help of the geometric generator
- *Evaluation of structural models:* the ability to evaluate the validity and complexity of a generated model, based on its structure (available to the structural generator directly), and its geometry (accessible through the interface of the geometric generator)
- *Generating structurally diverse models:* the ability to combine a limited set of operations in ways that result in a high amount of complex and diverse mechanical models

5.1 Defining the metamodel of the blueprints

The metamodel has to be able to represent the series of operations that make up a blueprint, in a way that can be repeatedly evaluated with identical results. In order to achieve this, the random parameters of the operations also have to be reproducible, which can be achieved by using a set of pre-generated seeds for the random generator. The root object of the model is the blueprint itself, which stores the list of possible random seeds, and the list of already performed operations. The operations store the random seed used to create them, the parameters of the specific operation generated from that random seed, and optionally subject and input geometries, represented by earlier operations which created them. Upon successfully performing an operation, the identifier of the geometric result, returned by the geometric model generator, is also stored in the operation, to be used as an input of future operations. The operation hierarchy contains abstract types for each output entity type (solid body, closed curve, open curve), which can be used to search for a suitable input of another operation. The resulting complete metamodel is visible on figure A.1.1.

5.2 Implementation

Using the previously defined metamodel, the generator is implemented with the help of the VIATRA-DSE library. The blueprint can be modified by a set of *transformation rules*, which create and evaluate operations, and append them to the list in the blueprint if the result is valid. After each transformation, the model fitness has to be calculated, for which a set of *complexity objectives* are used. The exploration is performed according to a *strategy*, which makes the decision about the next step based on the model fitness of the current state. The DSE library requires the coding of the model state by a *state coder*, to be able to store and compare different states.

5.2.1 Transformation rules

The transformation rules consist of two parts: a pattern to be matched on the model, and an action to be performed using this match. Each operation type has its own transformation rule. Patterns are defined using the graph query language of VIATRA [3], each of them matching a random seed in the blueprint, and optionally input operations (an example pattern is shown below 5.1). Actions are defined as Java classes, each of them constructing the corresponding operation, using the random seed for generating the parameters, then sending the operation to the geometric modeler, which creates and validates the resulting geometry. If the geometric operation is successful, the operation is added to the blueprint, otherwise the model state does not change.

```
pattern
internalTurningPrecondition(
  blueprint : Blueprint, seedOption : SeedOption, subject : SolidBodyOperation, input :
     OpenCurveOperation
) {
  find leafSolidBody(blueprint, subject);
  find symmetricSolid(subject);
  find openCurve(blueprint, input);
  Blueprint.seedOptions(blueprint, seedOption);
}
```

Listing 5.1: Graph query pattern for the internal turning operation

5.2.2 Interfacing with the geometric model generator

The geometric model generator is accessed by the structural generator via a Thrift [5] RPC interface. It is used both for performing geometric operations on the model, and for querying geometric information used for calculating the model fitness. The structural generator does not use the geometric modeler's capabilities of generating random values and building complex execution graphs, as these tasks are performed by the structural modeler itself in this setup. Instead, all operations are created with exact values as parameters, and executed as a graph consisting of only a single operation node, with stored result nodes used for the inputs. The result of the executed operation is used to create a new stored result node, which can later provide the input of a future operation.

5.2.3 State coding

Coding of the model state is required by the VIATRA-DSE library, as the state has to be represented in a format which is efficiently storeable and comparable, and independent of specific model structures. In the generator, state coding is implemented with a simple string serialization of the list of operations, each containing the operation name, and the random seeds of the current operation and the inputs. This representation was chosen for its simplicity, to be used as a proof-of-concept only, as it is not particularly memory-efficient, but it can be easily improved in later iterations.

5.3 Exploration strategies

The exploration strategy defines how should the model transformations be performed in a certain state. The VIATRA-DSE library contains several built-in exploration strategies, such as depth and breadth first search, best first strategy, fixed-priority strategy, hill climbing, evolutionary algorithms[3], and it is also possible to define new ones. In the generator, the built-in breadth first, depth first, and best first strategies were used during the evaluation.

5.4 Defining the model fitness

The model fitness is used by the exploration strategy to define whether a specific model fulfills the search objective, and the effect of a certain transformation on the fitness can also influence the direction of the exploration in some strategies. In our case, the objective is to create models that are sufficiently complex, thus fitness should be defined as some kind of complexity metric. The current implementation of the generator defines two objectives:

- *Solid body complexity:* this has been defined as the ratio of the surface area and the volume of the body, multiplied by the diagonal of its bounding box, to be independent of the absolute size (as surface area scales by the square, and volume scales by the cube of the dimensions). This is the primary goal, thus it has been defined as a hard objective, which has to be over a certain limit for a model to be considered a solution.
- *Curve complexity:* this has been defined as the length of the curve, divided by the diagonal of its bounding box, to be independent of the absolute size. This objective is needed because otherwise curve creation would have no effect on the fitness, which would adversely effect the diversity when using strategies that consider incremental fitness changes. As it does not directly correlate with the complexity of the solid body, which is our main interest, it is not defined as a hard objective.

Like in the case of the state coder, the complexity metrics used for measuring the model fitness are only used as a proof-of-concept, more refined metrics should be determined for future use.

Chapter 6

Evaluation

6.1 Performance measurements

Various measurements were conducted to answer the following research questions:

- *RQ1:* How is the execution time distributed between the generators, and how much is the communication overhead?
- *RQ2:* How does the generation method scale in relation to the number of desired solutions?
- *RQ3:* How does the usage of different execution strategies affect the execution time?

6.1.1 RQ1-2: Distribution of execution time and scalability

Setup: in this measurement, the generation of 10, 50, 100, 500 and 1000 models was performed, using a breadth first search strategy, with a solid body complexity objective of 50 (calculated as defined in section 5.4).

Time spent performing the following actions were measured:

- executing an operation in the structural generator
- executing an operation in the geometric generator
- calling operation execution on the geometric generator from the structural generator
- calculating model fitness in the structural generator
- examining geometric properties in the geometric generator
- calling geometric property examination on the geometric generator from the structural generator
- exporting a solution to STEP in the structural generator
- exporting a solution to STEP in the geometric generator
- calling export to STEP on the geometric generator from the structural generator

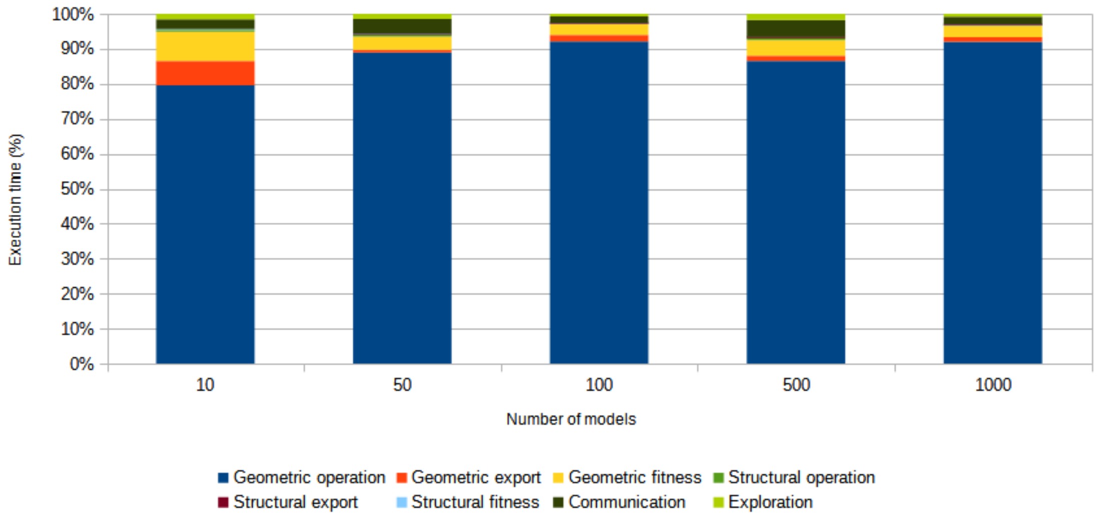


Figure 6.1: Relative distribution of execution times

The communication overhead is calculated from the difference between time spent calling an action on the geometric generator from the structural generator, and actually performing it in the geometric generator. The total time of the exploration process is also measured, and time not spent in any of the above actions is categorized as "Exploration", which is mostly time spent in the internals of the VIATRA-DSE. To account for the randomness of the exploration method, the measurements were performed 20 times, of which the first 5 were ignored to reduce the effect of the Java VM warm-up on the results, and the median value of the remaining was taken.

Analysis of the results:

- *RQ1:* figure 6.1 shows the relative distribution of time between the generators and their actions at different numbers of generated models. It is clearly visible that most of the time (80-90%) is spent with performing the geometric operations. The communication overhead between the processes is between 2-5%, which is not insignificant, but not high enough to consider giving up the advantages of modularity and native interfacing provided by this architecture.
- *RQ2:* figure 6.2 shows the absolute execution times, with distribution between the generators and their actions, at different numbers of generated models. The increase in execution time slightly below linear, which is most likely the result of reusing partial solutions as the exploration progresses.

6.1.2 RQ3: Evaluation of different strategies

Setup: in this measurement, the generation of 10, 50 and 100 models was performed, using breadth first search, depth first search and best first strategies, with a solid body complexity objective of 50 (calculated as defined in section 5.4). The timeout for the exploration was 5 minutes. To account for the randomness of the exploration method, the measurements were performed 20 times, of which the first 5 were ignored to reduce the effect of the Java VM warm-up on the results, and the median value of the remaining was taken.

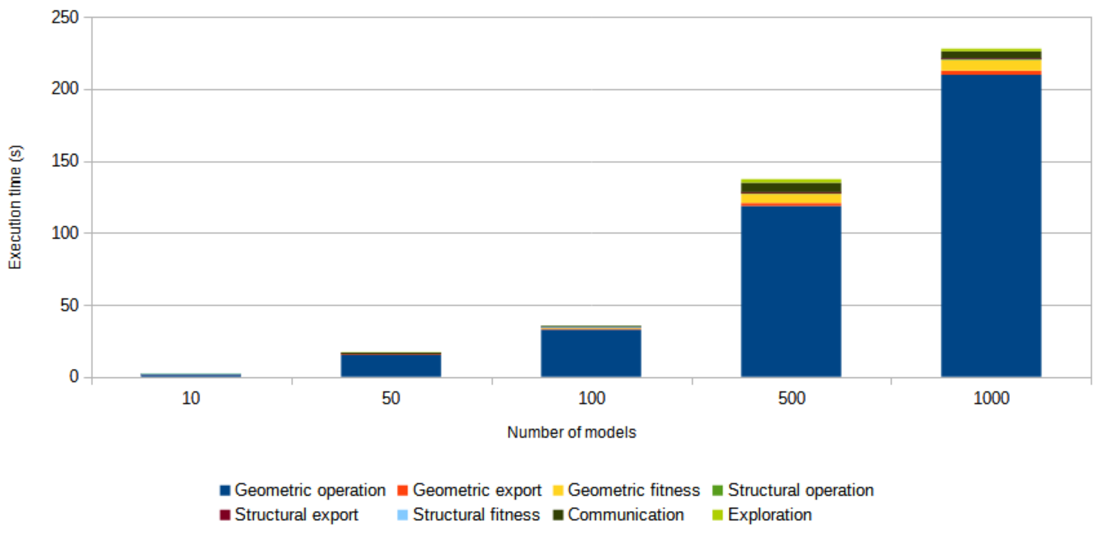


Figure 6.2: Absolute execution times

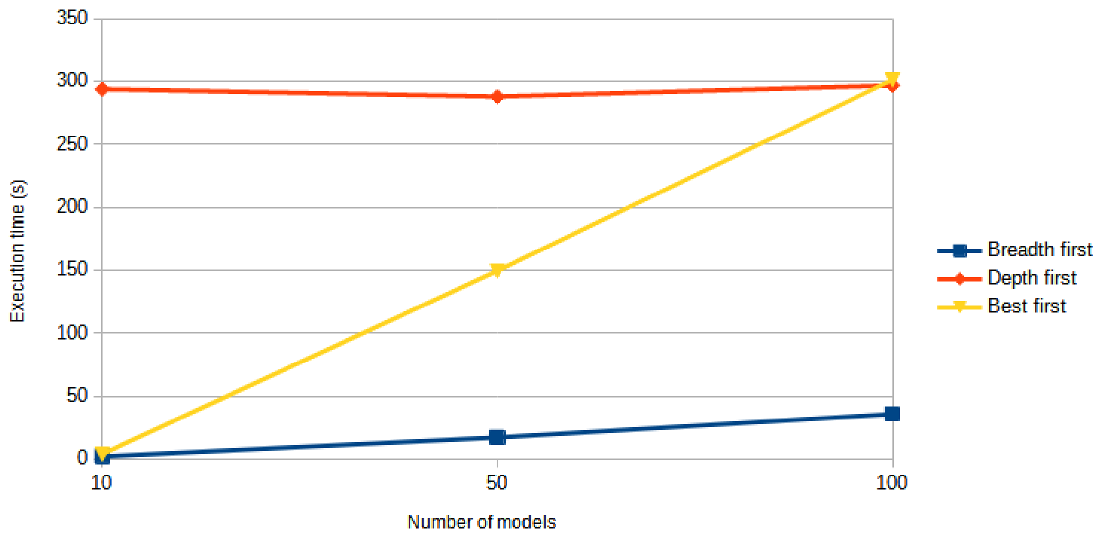


Figure 6.3: Execution times of different strategies

Analysis of the results: figure 6.3 shows the execution times of the different strategies. In case of the depth first search strategy, about 40% of the executions timed out without finding a single solution, and other executions timed out after finding only a few solutions. The best first strategy was relatively fast with finding 10 models, but became significantly slower at 50, and timed out with only a partial result at 100 models. The performance of breadth first search was already examined in the previous questions, it is only displayed here as comparison. The most likely explanation of the poor performance of the other strategies is that they are going on trajectories which try to optimize the soft objective, without increasing the solid body complexity metric. This issue should be fixed by refining the fitness metrics.

6.2 Interfacing with expert systems

As the generator outputs STEP files, the expert system used for analyzing mechanical parts and advising about their manufacturing process should be able to read these files. Considering that the same format would be most likely used to export manually created models from CAD software, this is a reasonable expectation of such systems.

Training these systems could be performed by supervised learning, feeding the created models and their calculated complexities as training data. An adversarial training setup can also be created by calculating one of the DSE objectives from the output of the analyzer, with the explorer trying to find models that are complex according to the expert system's metric, but not according to its own.

Chapter 7

Conclusion

In this report, I suggested a novel way of generating diverse mechanical models using a combined approach of generating and evaluating a geometric and a structural model in parallel. The approach was implemented with an architecture using VIATRA-DSE for performing design space exploration on the structural model, and OpenCascade for creating and analyzing the geometric model. The performance of the implementation was measured, analyzing its scalability and the efficiency of the distributed architecture. Finally, suggestions were made regarding using the generator as a source of training data for manufacturing-related expert systems.

List of Figures

2.1 Example A (left) shows the execution of a valid turning operation, and example B (right) shows an invalid one, both displayed in section view . . . 4

2.2 A complex workpiece machined by turning, with pre-existing milled surfaces in a hexagonal pattern . . . 5

2.3 A rough milling performed on a workpiece, with visibly distinct Z-steps . . . 5

3.1 A high-level overview of the model generator . . . 11

3.2 Flow of the generation process . . . 12

3.3 Architecture of the model generator . . . 13

4.1 A cylinder with external and internal turning (boring) applied, normal and section views . . . 15

4.2 A turned solid created directly from an open curve . . . 15

4.3 A thin-walled turned solid in section view . . . 15

4.4 A rectangular block with external and internal milling applied . . . 16

4.5 A milled solid created directly from a closed curve . . . 16

4.6 Repeated milling operations at different depths, using the same curve scaled to different sizes . . . 16

4.7 A pair of holes created on the top face of the workpiece, mirrored along the midplane parallel with the X axis . . . 17

4.8 A solid block cut with an angled plane . . . 17

4.9 The interface hierarchy of operations . . . 19

4.10 The type hierarchy of execution nodes . . . 20

4.11 An execution graph consisting of a cylinder, and an external and internal turning operation with open splines as inputs . . . 20

4.12 The results of repeated evaluations of a single execution graph . . . 21

6.1 Relative distribution of execution times . . . 26

6.2 Absolute execution times . . . 27

6.3 Execution times of different strategies . . . 27

A.1.1Blueprint metamodel . . . 34

Bibliography

[1] Project overview. https://dev.opencascade.org/about/project_overview, 2011. Accessed: 2022-10-25.

[2] DXF Reference. https://help.autodesk.com/view/OARX/2018/ENU/?guid=GUID-235B22E0-A567-4CF6-92D3-38A2306D73F3, 2018. Accessed: 2022-10-25.

[3] Viatra. https://wiki.eclipse.org/VIATRA, 2018. Accessed: 2022-11-01.

[4] Implicit modelling for complex geometry. https://www.3dcadworld.com/implicit-modelling-for-complex-geometry/, 2020. Accessed: 2022-10-30.

[5] Apache thrift. https://thrift.apache.org/, 2022. Accessed: 2022-11-01.

[6] ISO/TC 184/SC 4. *Industrial automation systems and integration — Product data representation and exchange — Part 242: Application protocol: Managed model-based 3D engineering.* International Organization for Standardization, ISO 10303-242:2020 edition, 2020. URL https://www.iso.org/standard/66654.html.

[7] ISO/TC 184/SC 4. *Industrial automation systems and integration — Product data representation and exchange — Part 1: Overview and fundamental principles.* International Organization for Standardization, ISO 10303-1:2021 edition, 2021. URL https://www.iso.org/standard/72237.html.

[8] Martin Brain, Vijay D'Silva, Leopold Haller, Alberto Griggio, and Daniel Kroening. An abstract interpretation of dpll(t). In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 455–475, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-35873-9.

[9] Lisa Croppi, Niccolò Grossi, Antonio Scippa, and Gianni Campatelli. Fixture optimization in turning thin-wall components. *Machines*, 7:68, 10 2019. DOI: 10.3390/machines7040068.

[10] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll(t): Fast decision procedures. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 175–188, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-27813-9.

[11] C.M. Hoffmann. *Geometric and Solid Modeling: An Introduction.* Morgan Kaufmann series in computer graphics and geometric modeling. Morgan Kaufmann, 1989. ISBN 9781558600676. URL https://books.google.hu/books?id=GYhRAAAAMAAJ.

[12] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 297–306, 2008.

[13] Hareendran Manikandan and Tufan Chandra Bera. Modelling of dimensional and geometric error prediction in turning of thin-walled components. *Precision Engineering*, 72:382–396, 2021. ISSN 0141-6359. DOI: `https://doi.org/10.1016/j.precisioneng.2021.05.013`. URL `https://www.sciencedirect.com/science/article/pii/S0141635921001562`.

[14] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[15] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, November 2006. ISSN 0004-5411.

[16] Sven Schneider, Leen Lambers, and Fernando Orejas. Automated reasoning for attributed graph properties. *STTT*, 20(6):705–737, 2018.

[17] Stephen J. Schoonmaker. *The CAD guidebook : a basic manual for understanding and improving computer-aided design*. Marcel Dekker, 2003. ISBN 9780824708719, 0824708717.

[18] Oszkár Semeráth, András Vörös, and Dániel Varró. Iterative and incremental model generation by logic solvers. In *FASE*, pages 87–103. Springer, 2016.

[19] Oszkár Semeráth, Aren A. Babikian, Anqi Li, Kristóf Marussy, and Daniel Varró. Automated generation of consistent models with structural and attribute constraints. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, page 187–199, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370196. DOI: `10.1145/3365438.3410962`. URL `https://doi.org/10.1145/3365438.3410962`.

[20] Ghanem Soltana, Mehrdad Sabetzadeh, and Lionel C. Briand. Practical constraint solving for generating system test data. *ACM Trans. Softw. Eng. Methodol.*, 29(2), April 2020. ISSN 1049-331X.

[21] IGES-PDES Organization Staff. *And U. S. Pro-IPO-100 1993: Initial Graphics Exchange Specification Version 5.2*. U. S. Product Data Association, 1993. ISBN 1885389000, 9781885389008.

[22] Attila Szmejkál and Péter Ozsváth. *Járműszerkezeti Anyagok és Technológiák II.* Typotex Kiadó, 2011.

[23] *Eclipse Modeling Framework*. The Eclipse Project, 2019. `http://www.eclipse.org/emf`.

[24] Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, volume 10800 of *LNCS*, pages 285–312. Springer, 2018.

[25] Xun Xu. *Integrating Advanced Computer-Aided Design, Manufacturing, and Numerical Control: Principles and Implementations*. Information Science Reference, 2009. ISBN 1599047144, 9781599047140. DOI: `10.4018/978-1-59904-714-0`.

Appendix

A.1 Blueprint metamodel

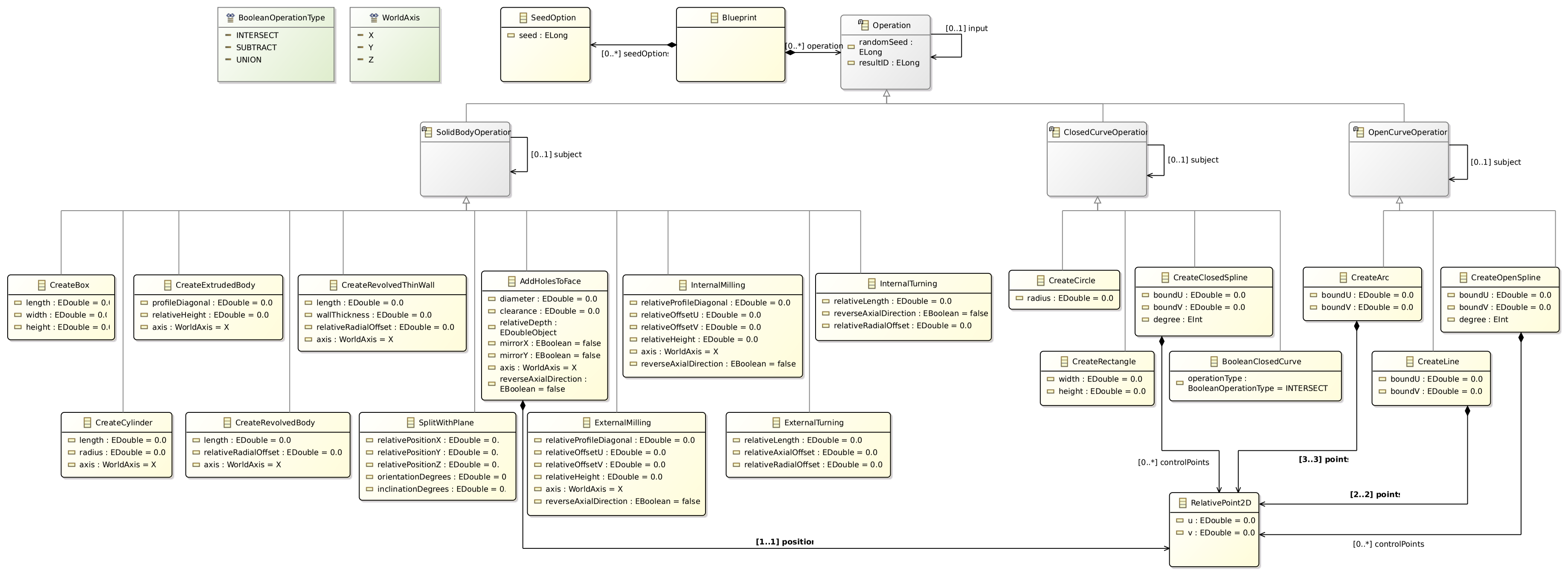


Figure A.1.1: Blueprint metamodel