Component-based Synthesis of Motion Planning Algorithms

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

Tristan Schäfer

Dortmund

2021

Dekan: Prof. Dr.-Ing. Gernot Fink
Gutachter:

Prof. Dr. Jakob Rehof (Technische Universität Dortmund, Deutschland)
Prof. Dr.-Ing. Petra Wiederkehr (Technische Universität Dortmund, Deutschland)

Danksagung

Ich bedanke mich vor allem bei meinem Betreuer und Vorgesetzten Jakob Rehof für die Möglichkeit, in diesem interessanten Themengebiet der Softwaresynthese zu arbeiten. Durch seine fortlaufende Unterstützung und die richtigen Impulse wurde das Dissertationsprojekt maßgeblich geprägt. Dabei hatte ich dennoch immer die nötige Freiheit, eigene Ideen zu verfolgen. Ich bedanke mich außerdem bei Petra Wiederkehr für die Arbeit an der gemeinsamen Publikation und die Bereitschaft, als Gutachterin für die Dissertation zu fungieren. Darüber hinaus danke ich Anne Meyer und Peter Buchholz herzlich für die Beteiligung an der Prüfungskommission. Ich danke außerdem Lars Hildebrand für seine Tätigkeit als Mentor im Rahmen des strukturierten Promotionsprogramms.
Ich bedanke mich bei allen Mitarbeitern des Lehrstuhl 14 für die gemeinsame Zeit und die angenehme Arbeitsumgebung. Unter meinen Kollegen gilt mein besonderer Dank Jan Bessai für die vielen interessanten und lustigen Gespräche, aber auch für die wertvollen inhaltlichen Diskussionen und seine große Hilfsbereitschaft. Jim Bergmann danke ich herzlich für die produktive Zusammenarbeit und seine Unterstützung in Fragen der spanenden Fertigung. Danke an Fadil Kallat und Christian Riest für unsere netten Dienstags- und Mittwochsrunden und die erfolgreiche Arbeit am Industrieprojekt. Ebenso bedanke ich mich bei Anna Vasileva, Doris Schmedding, Jan Winkels und Constantin Chaumet für die Zusammenarbeit. Mein großer Dank gilt Ute Joschko und Sevda Tarkun dafür, dass sie den Lehrstuhl am Laufen halten und uns vor den unbequemen Aspekten der Verwaltung schützen.
Besonderer Dank geht an meine Freunde und meine Familie für die Unterstützung.

Abstract

Combinatory Logic Synthesis generates data or runnable programs according to formal type specifications. Synthesis results are composed based on a user-specified repository of components, which brings several advantages for representing spaces of high variability. This work suggests strategies to manage the resulting variations by proposing a domain-specific brute-force search and a machine learning-based optimization procedure. The brute-force search involves the iterative generation and evaluation of machining strategies. In contrast, machine learning optimization uses statistical models to enable the exploration of the design space. The approaches involve synthesizing programs and meta-programs that manipulate, run, and evaluate programs. The methodologies are applied to the domain of motion planning algorithms, and they include the configuration of programs belonging to different algorithmic families. The study of the domain led to the identification of variability points and possible variations. Proof-of-concept repositories represent these variability points and incorporate them into their semantic structure. The selected algorithmic families involve specific computation steps or data structures, and corresponding software components represent possible variations. Experimental results demonstrate that CLS enables synthesis-driven domain-specific optimization procedures to solve complex problems by exploring spaces of high variability.

Zusammenfassung

Combinatory Logic Synthesis (CLS) generiert Daten oder lauffähige Programme anhand von formalen Typspezifikationen. Die Ergebnisse der Synthese werden auf Basis eines benutzerdefinierten Repositories von Komponenten zusammengestellt, was diverse Vorteile für die Beschreibung von Räumen mit hoher Variabilität mit sich bringt. Diese Arbeit stellt Strategien für den Umgang mit den resultierenden Variationen vor, indem eine domänen-spezifische Brute-Force Suche und ein maschinelles Lernverfahren für die Untersuchung eines Optimierungsproblems aufgezeigt werden. Die Brute-Force Suche besteht aus der iterativen Generierung und Evaluation von Frässtrategien. Im Gegensatz dazu nutzt der Optimierungsansatz statistische Modelle zur Erkundung des Entwurfsraums. Beide Ansätze synthetisieren Programme und Metaprogramme, welche Programme bearbeiten, ausführen und evaluieren. Diese Methoden werden auf die Domäne der Bewegungsplanungsalgorithmen angewendet und sie beinhalten die Konfiguration von Programmen, welche zu unterschiedlichen algorithmischen Familien gehören. Die Untersuchung der Domäne führte zur Identifizierung der Variabilitätspunkte und der möglichen Variationen. Entsprechende Proof of Concept Implementierungen in Form von Repositories repräsentieren jene Variabilitätspunkte und beziehen diese in ihre semantische Struktur ein. Die gewählten algorithmischen Familien sehen bestimmte Berechnungsschritte oder Datenstrukturen vor, und entsprechende Software Komponenten stellen mögliche Variationen dar. Versuchsergebnisse belegen, dass CLS synthese-getriebene domänenspezifische Optimierungsverfahren ermöglicht, welche komplexe Probleme durch die Exploration von Räumen hoher Variabilität lösen.

Contents

Abstract **v**

List of figures **xii**

1 Introduction **1**

1.1 Organization of this Thesis 2

2 Software Synthesis **3**

2.1 Type-directed Synthesis 5

2.2 Component-based Synthesis with Combinatory Logic 5

2.2.1 Intersection Types 6

2.2.2 Typed Terms 7

2.2.3 Inhabitation 8

2.3 CLS Framework 8

2.3.1 Kinding and Taxomomy 9

2.3.2 Repository Construction 9

2.3.3 Example 10

3 Motion Planning **13**

3.1 The general Piano Movers Problem 13

3.2 Geometric Representation 14

3.2.1 Continuous Path Representation 15

3.3 Configuration Space 16

3.4 Transformations 17

3.4.1 Rotation 18

3.4.2 3D Rigid Body Planning 20

3.5 Combinatorial Motion Planning 20

3.6 Sampling-Based Motion Planning 21

3.7 Coverage Path Planning 22

4 Composition of Combinatorial Motion Planning Algorithms 25
4.1 Data Structures and Native Types . . . 27
4.2 Cell Segmentation . . . 28
4.2.1 Grid-Based Cells . . . 28
4.2.2 Triangulation and Tetrahedralization . . . 28
4.2.3 Vertical Cell Decomposition . . . 30
4.3 Graph Construction . . . 31
4.4 Graph Traversal and Graph Search . . . 34
4.5 Variations and Results . . . 35
4.6 Discussion . . . 40

5 Component-Based Synthesis for Tool Path Planning 41
5.1 Planning Problem Instance . . . 43
5.1.1 Assessment of Tool Engagements . . . 44
5.1.2 Tool Parameterization . . . 46
5.2 Literature Review . . . 48
5.2.1 Optimization-based Tool Path Generation . . . 48
5.2.2 Formal Methods in Machining . . . 49
5.2.3 Conventional Planning Methods . . . 50
5.2.4 Constant Tool Engagement . . . 50
5.2.5 Planning with Feature Extraction . . . 50
5.3 Brute-Force Search . . . 52
5.3.1 CNC Model Representation . . . 53
5.3.2 Path Coverage Step . . . 54
5.3.3 Path Coverage Configuration . . . 55
5.3.4 Path Coverage Result . . . 56
5.4 Path Planning Primitives . . . 56
5.4.1 Zig-Zag Patterns . . . 57
5.4.2 Contour-Parallel Tool Paths . . . 58
5.4.3 Directed Trochoidal Slot Milling . . . 61
5.4.4 Contour-Parallel Trochoidal Slot Milling . . . 61
5.4.5 Spiral Roughing . . . 61
5.4.6 Dive-in Motions . . . 62
5.5 Repository Structure and Combinators . . . 62
5.5.1 Generic Composition . . . 64
5.5.2 Convex Hulls . . . 65
5.5.3 Model Rotation . . . 65
5.6 Numerical Evaluation of Tool Paths . . . 66
5.6.1 Acceleration Model for Machine Dynamics . . . 66

5.6.2 Construction of Lookup Tables . . . 68
5.6.3 Calculation of Feed Rates . . . 68
5.7 Experiments and Results . . . 71
5.7.1 Experiment 1: Open Pocket . . . 72
5.7.2 Experiment 2: Closed Pocket . . . 78
5.7.3 Experiment 3: Isles . . . 81
5.7.4 Generation of Machine Control Instructions . . . 86
5.8 Discussion . . . 87
5.8.1 Brute-Force Search Strategy . . . 87
5.8.2 Holonomic and Non-holonomic Planning . . . 88
5.8.3 Further Planning Primitives . . . 88

6 Design Space Exploration for Sampling-Based Motion Planning 91
6.1 Motion Planning as a Multi-Objective Optimization Problem . . . 93
6.2 Experimental Setup . . . 94
6.2.1 Communication Protocol . . . 95
6.2.2 Open Motion Planning Library . . . 96
6.2.3 Generation of Python Scripts . . . 97
6.2.4 Examination of Randomized Algorithms . . . 97
6.3 A Repository for Sampling Based Motion Planning Programs . . . 98
6.3.1 Top-level Combinator . . . 99
6.3.2 Dynamic Construction of the Repository . . . 100
6.4 Planners . . . 100
6.4.1 Probabilistic Roadmap Method . . . 101
6.4.2 Tree-based Planners . . . 101
6.5 Sampling Strategies . . . 102
6.5.1 Uniform Sampling . . . 102
6.5.2 Gaussian Valid State Sampling . . . 102
6.5.3 Obstacle-Based sampling . . . 103
6.5.4 Maximum Clearance Sampling . . . 104
6.6 Assignment of Planners and Samplers . . . 105
6.7 Collision Detection . . . 106
6.8 State Cost and Optimization Objectives . . . 107
6.9 Results . . . 108
6.9.1 Problem instance "Abstract" . . . 108
6.9.2 Problem instance "Alpha 1.5" . . . 110
6.9.3 Problem instance "Home" . . . 111
6.9.4 Success Rate . . . 112
6.9.5 Parameter Importance . . . 113

6.10 Related Work . . . 113
6.11 Discussion . . . 114

7 Conclusion **117**
7.1 Results . . . 118
7.2 Combinatorial Complexity . . . 118
7.3 Incorporated Methods . . . 119
7.4 Outlook . . . 119

A Inhabitant Trees for the Examples in Section 4.5 **121**
A.1 Tree for the Grid Approximation Example . . . 121
A.2 Tree for the Triangulation Example . . . 122
A.3 Tree for the VCD Example in Section 4.5 . . . 123

B Remarks on Published Work **125**

C Inhabitant Trees for the Machining Experiments in Section 5.7 **127**

D Planners and Samplers used in Chapter 6 **133**
D.1 List of Planners . . . 133
D.2 List of Samplers . . . 134

Bibliography **134**

List of Figures

3.1 Polygon with holes defined by ordered sequences of points, taken from [61] . . 15
3.2 Workspace covered by different robot configurations, taken from [22] 16
3.3 Representation for a configuration space, taken from [22] 17
3.4 Rotation in 2d about point through angle alpha, taken from [43] 18
3.5 Adding samples to the graph, taken from [61] 22

4.1 Free cells formed by a grid-based approximation of $\mathscr{C}_{free}$ 29
4.2 Decomposition of $\mathscr{C}_{free}$ using parameterized triangulation 30
4.3 Cases for line construction in vertical cell decomposition, taken from [61] . . . 31
4.4 Vertical cell decomposition as an example for a 2D segmentation strategy . . . 32
4.5 An example for a minimum spanning tree computed from the roadmap 36
4.6 A grid-based approximation of $\mathscr{C}_{free}$ with a roadmap built from centroids . . . 37
4.7 Cell decomposition by triangulation and graph construction with connecting nodes 38
4.8 Vertical cell decomposition with centroids and connecting nodes. The cells' subgraphs contain edges from cell vertices to connecting nodes and centroids. 39

5.1 Chip formation and shear zones, taken from [87] 45
5.2 Tool engagement angle for conventional and trochoidal milling, taken from [58] 45
5.3 Machining parameters, taken from [75] 46
5.4 Synthesis of machining strategies and search procedure 52
5.5 (a) Zig-zag pattern, (b) Zig pattern 58
5.6 (a) Spiral roughing, (b) Contour-parallel tool path 59
5.7 (a) Trochoidal channel, (b) Trochoidal slot 62
5.8 PathCoverageStep tree representation of candidate 1.1 73
5.9 Generated tool paths and geometric models for candidate 1.1, steps a and b . . 73
5.10 PathCoverageStep tree representation of candidate 1.2 74
5.11 Generated tool paths and geometric models for candidate 1.2, steps a and b . . 74
5.12 PathCoverageStep tree representation of candidate 1.3 75
5.13 Generated tool paths and geometric models for candidate 1.3, steps a-f 76

5.14 Feed rate visualization for candidates 1.1 and 1.2 77
5.15 Feed rate visualization for candidate 1.3 77
5.16 PathCoverageStep tree representation of candidate 2.1 78
5.17 Generated tool paths and geometric models for candidate 2.1, steps a-f 79
5.18 Feed rate visualization for candidate 2.1 80
5.19 PathCoverageStep tree representation of candidate 3.1 82
5.20 Generated tool paths and geometric models for candidate 3.1, steps a and b . . 82
5.21 Generated tool paths and geometric models for candidate 3.1, steps c-h 83
5.22 PathCoverageStep tree representation of candidate 3.2 84
5.23 Generated tool paths and geometric models for candidate 3.2, steps a and b . . 84
5.24 Generated tool paths and geometric models for candidate 3.2, steps c-f 85
5.25 Feed rate visualization for candidates 3.1 and 3.2 86

6.1 Architectural Overview for the Machine Learning Procedure 92
6.2 Halton Sampling and Gauss Sampling 103
6.3 Obstacle based roadmap example, taken from [4] 104
6.4 Example for a maximum clearance path, taken from [36] 105
6.5 Problem instance "Abstract" 108
6.6 Result for the Problem instance "Abstract" 109
6.7 Problem instance "Alpha 1.5" 110
6.8 Result for the Problem instance "Alpha" 110
6.9 Problem instance "Home" 111
6.10 Result for the Problem instance "Home" 112
6.11 Moving average over the success rates of the iterations for multiple optimization runs 112

C.1 Inhabitant tree structure for solution candidate 1.1 127
C.2 Inhabitant tree structure for solution candidate 1.2 128
C.3 Inhabitant tree structure for solution candidate 1.3 129
C.4 Inhabitant tree structure for solution candidate 2.1 130
C.5 Inhabitant tree structure for solution candidate 3.1 131
C.6 Inhabitant tree structure for solution candidate 3.2 132

Chapter 1

Introduction

Motion planning is a fundamental problem in the field of robotics. It is part of robot navigation and aims to find collision-free paths for robot motions. The trend towards application specific-planning involves the consideration of domain-specific constraints and objectives. In general, a motion planning algorithm's performance and applicability depend on the planning space, the robot's space of possible configurations, and the problem instance. An application-specific problem definition introduces further constraints that affect planning approaches. For instance, planning algorithms can incorporate differential constraints that restrict admissible velocities and accelerations of robotic systems. Moreover, constrained task planning can involve uncertainty in planning, robotic states like the charge status of the robot's battery, or further aspects of computation such as real-time requirements.

The rising demand for configurable products and manufacturing systems requires efficient methods for handling variability, and planning workflows benefit from automated planning techniques. Software systems must adapt to changing requirements, and employed motion planning algorithms must expose unique characteristics. The different planning objectives and constraints lead to a broad range of techniques to accomplish distinct tasks during planning. These aspects and the corresponding requirements of the robotic systems cause high variability for global motion planning algorithms that build search graphs to represent the robot-specific accessibility of the planning space.

This thesis deals with the component-based representation of different algorithmic families targeting global motion planning and path coverage planning. Combinatory Logic Synthesis can help to describe the space of high variability. It automatically generates variations of motion planning algorithms, which enables the application of problem-specific brute-force search or optimization procedures. Multiple proof-of-concept implementations and different experiments demonstrate the strengths of this approach.

1.1 Organization of this Thesis

Including this introduction, this thesis consists of seven chapters. The contents are as follows:

- The second chapter outlines the formal background and some practical aspects of the CLS framework.
- The third chapter explains essential concepts of motion planning and forms the basis for understanding the domain-specific explanation in the following chapters.
- The fourth chapter presents a study to synthesize variants of combinatorial motion planning algorithms. It specifies selected variability points and demonstrates the corresponding adaption to a semantic layer of type specifications. Several results illustrate variations of this algorithmic family.
- Chapter 5 deals with the synthesis of machining strategies from tool path planning components for milling. The presented work enables a CLS-based brute-force search procedure and uses a domain-specific semantic layer to reduce the search space with high variability. Selected results illustrate the generated terms, data structures, and tool paths with corresponding model transformations. The selection aims to illustrate the impact of different planning components.
- Chapter 6 describes an approach for the synthesis of sampling-based motion planning algorithms. A selection of domain-specific variability points provides the basis for the repository design, and its configuration regulates the synthesis of Scala programs that generate Python scripts based on templating schemes. These scripts are runnable Python code that solves the problem instance. A machine learning loop encapsulates the synthesis and evaluation of planning programs in a black-box function. This way, CLS enables a synthesis-based optimization procedure.
- The thesis is concluded with Chapter 7, which summarizes the work and gives a short outlook.

Chapter 2

Software Synthesis

Software synthesis is the task of discovering a program that corresponds to a user-defined specification, and it is an active field of study in computer science [40]. The automated search for programs enables non-expert users without particular software development expertise to construct programs that meet their requirements.

The distinction of different software synthesis approaches can follow the classification regarding the **dimensions of software synthesis** identified by Gulwani [38]:

- User intent
- Search space
- Search technique

There are several ways to express the **user intent**. For instance, input-output examples can impose constraints on the computation results of the required program. The synthesis discipline that adapt this particular way to express user intent is called programming-by-examples (PBE) or inductive synthesis. Moreover, statements in an appropriate logical calculus can specify the synthesis target.

The **search space** describes the set of possible programs, which often incorporates rules of program construction. For instance, syntax-guided synthesis suggests a search space that builds programs based on a context-free grammar.

The **search technique** defines how the user intent and the definition of search space are incorporated to find a matching program. Examples are enumerative search, deductive search, or constraint solving.

FlashFill [39] is one of the first synthesis techniques, which found commercial use in Microsoft Excel to fill data in worksheet cells automatically. This work by Gulwani performs string processing based on input-output examples. An enumerative search yields programs of a predefined language for string manipulation that includes control structures and operators for string transformation. Further development led to Blinkfill [88], which uses semi-supervised learning for recognizing recurring string patterns in the input examples, enabling a better search performance that works well with fewer examples.

The search techniques involve a deductive search on specifications and eventually led to the development of **PROSE** [78], a synthesis framework that works for different domain-specific languages (DSLs) and has found adaption in several commercial software systems.

PBE allows for a simple presentation of the user intent. However, it comes with the drawback of underspecification because examples only partially represent a program's behavior. For this reason, some PBE techniques make use of **data-driven learning** to interpret the user intent and yield the most promising program [8, 35, 49]. Kalyan et al. proposed a PBE approach to combine the strengths of machine learning and deduction-based search [54], which enables a real-time search that works well for a small number of examples.

In 2006, Solar-Lezama et al. proposed a program synthesis technique based on **sketching** [89], where the developer provides a partial program containing *holes*. The synthesis resolves these markers with suitable code according to user-specified assertions. The enumerative search procedure involves solving a corresponding generalized boolean satisfiability problem.

This work inspired a family of techniques called **syntax-guided synthesis** (SyGuS, [3]), where a context-free grammar defines the construction of terms. The SyGuS problem is finding a program that complies with semantic constraints formulated in the underlying background theory. The Syntax-Guided Synthesis Competition (SyGuS-Comp) is a yearly program synthesis competition that compares different program synthesis approaches in a competitive event. It comes with a language standard [79] which builds upon the SMT-LIB standard [10]. While the popularity of this competition resulted in a variety of active contributions, these methods are limited to small-scale programs.

Abstraction refinement [97, 30] is a search technique that makes use of symbolic execution to find equivalence classes of programs regarding their abstract input-output behavior. These abstract semantics lead to a reduction of the search space. The search procedure includes a counterexample-guided abstraction refinement that eventually finds a program that complies with the given input-output examples.

2.1 Type-directed Synthesis

The overall goal of synthesis is to produce some well-defined artifact based on a high-level description, automating as much of the way of production as possible. A particular class of approaches uses *types* for the high-level description of programs and conducting the synthesis process.

Zdancewich et al. [33] interpret input-output examples as refinement types and build a system that allows the deductive and enumerative search for inhabitants. The type system includes intersection types and union types, and it can express negation, which enables the consideration of counterexamples.

SYNQUID [76] performs the synthesis based on liquid type checking and enumeration of program terms. These terms are built from typed components where the liquid types encode their functionality. Several other works build upon the synthesis with liquid types [59, 77].

Guo et al. [41] use abstraction refinement in connection with type checking to perform the search. It involves abstraction over types, where abstract types denote polymorphic types with free type variables. Selected candidate terms are type-checked, and ill-typed terms trigger the abstraction refinement to identify well-typed terms.

2.2 Component-based Synthesis with Combinatory Logic

Combinatory Logic Synthesis [80, 13] is a type-based, component-oriented [82] synthesis approach that incorporates a decade of type-theoretic research. Unlike other techniques that generate programs from scratch, it synthesizes programs from a repository of user-defined software components. The components can be large programs or simple operations, which allows different levels of granularity. A finer granularity leads to a more precise representation of variability, but the adaptions in the semantic layer result in a more extensive search space.

The combinators' implementation is use-case-specific and might include heuristics or domain-specific software solutions. With this approach, it is possible to generate advanced software products that incorporate third-party libraries.

The component-based approach enables the use in different domains and the reuse of existing implementations. Semantic type expressions describe features of the components and their arguments, and the user-defined semantic layer originates from type specifications on component-level, a semantic taxonomy, and a kinding that guides type substitutions. CLS

is well-suited to describe solution spaces with high variability and captures the inherent combinatorial complexity of these spaces.

A logical type specification describes the user intent, the search space is the set of well-formed terms and the search procedure finds solutions by solving the problem of inhabitation. The remainder of this section will provide an overview of these aspects. It starts with the definition of intersection types, provides the rules of the underlying type system, and outlines the mechanics of the inhabitation algorithm. After that, an example demonstrates the rules of composition for a set of software components.

2.2.1 Intersection Types

Coppo and Dezani proposed the idea of intersection types in 1980 [24] as an extension of the λ-calculus. This work is the basis for the corresponding research field, and several scientific works have covered various aspects of the intersection type discipline since then. The BCD system introduced in [9] turned out to be the primary reference for intersection type systems. It was established in 1983 and is named after its creators Barendregt, Coppo, and Dezani.

The structure of intersection types is displayed in Equation 2.1. σ and τ are type expressions containing function types ($\rightarrow$, also called arrow types) or intersection types ($\cap$) whereas a denotes atomic type constants. The types are commutative, associative, and idempotent concerning the intersection operator. The universal type ω exposes special subtyping properties. The subtyping relation $\leq$ is the least reflexive and transitive relation that is closed under the following axioms:

$$\sigma, \tau ::= \omega \mid a \mid \sigma \rightarrow \tau \mid \sigma \cap \tau \tag{2.1}$$

$$\begin{aligned} &\tau \leq \omega \qquad \omega \leq \omega \rightarrow \omega \qquad \tau \leq \tau \cap \tau \qquad \sigma \cap \tau \leq \sigma \qquad \sigma \cap \tau \leq \tau \\ &(\sigma \rightarrow \rho) \cap (\sigma \rightarrow \tau) \leq \sigma \rightarrow (\rho \cap \tau) \\ &\sigma' \leq \sigma, \tau \leq \tau' \Rightarrow \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau' \end{aligned} \tag{2.2}$$

Equation 2.2 establishes the distributivity property in Equation 2.3, and Equation 2.4 defines the equivalence relation $\sim$ as the symmetric closure of $\leq$.

$$\sigma \leq \sigma', \tau \leq \tau' \Rightarrow \sigma \cap \tau \leq \sigma' \cap \tau' \tag{2.3}$$

$$\sigma \sim \tau \Leftrightarrow \sigma \leq \tau \leq \sigma \tag{2.4}$$

As a convention, intersections bind stronger than arrows, and arrows associate to the right, as shown in Equation 2.5.

$$\begin{aligned} A \cap B \to C \cap D &= (A \cap B) \to (C \cap D) \\ A \to B \to C &= A \to (B \to C) \end{aligned} \tag{2.5}$$

2.2.2 Typed Terms

Combinatory Logic Synthesis aims to build applicative terms from a set of typed components. A definition for these terms is shown in Equation 2.6. Equation 2.7 defines repositories, denoted as Γ, which contain components and their affiliated types. Intersection types can express feature vectors of a program or software component. Moreover, the types express the rules of composition as they declare the types of the components arguments. They consist of semantic types that describe properties of components and native types representing the component's data type in the underlying programming language.

$$M, N ::= x \mid (MN) \tag{2.6}$$

$$\Gamma = \{(x_1 : \tau_1), ..., (x_n : \tau_n)\} \text{ with } x_i \not\equiv x_j \text{ for } i \neq j \tag{2.7}$$

The typing of terms follows the rules of Finite Combinatory Logic (FCL) with intersection types [81], that are shown in Equation 2.8. The (var) rule allows a type judgment according to the information held in the repository, where $\Gamma, x : \tau$ describes a repository holding the information that x has the type τ. The $(\to_E)$ rule states that the composition of terms requires matching types. With M and N representing arbitrary terms according to Equation 2.6, the application of M to N is possible *iff* M has an arrow type $\tau \to \sigma$, and $\Gamma \vdash N : \tau$ holds. The $(\leq)$ rule incorporates the subtyping relation defined in 2.2 and shows its effect on type judgments.

The properties of a type system directly impact the theoretical complexity bounds for the corresponding inhabitation problem. For instance, the inhabitation problem for intersection types in the λ-calculus is undecidable unless further restrictions on the types are considered

[95, 96]. For this reason, *Bounded Combinatory Logic* (BCL, [28]) involves restrictions on the depth of substitutions for type variables. The relativized inhabitation problem for the k-bounded combinatory logic BCL_k is `(k+2)-EXPTIME` complete. In contrast, inhabitation in FCL with intersection types does not involve type schemas on variables, and it is EXPTIME complete.

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}(\mathsf{var}) \quad \frac{\Gamma \vdash M:\tau \to \sigma \quad \Gamma \vdash N:\tau}{\Gamma \vdash MN:\sigma}(\to_E) \quad \frac{\Gamma \vdash M:\tau \quad \tau \leq \sigma}{\Gamma \vdash M:\sigma}(\leq) \tag{2.8}$$

2.2.3 Inhabitation

The algorithm implementing CLS synthesizes terms by solving the type-theoretic problem of inhabitation:

$\Gamma \vdash ? : \tau$
Given Γ and τ, is there a term M such that $\Gamma \vdash M:\tau$?

CLS solves the inhabitation problem by performing a backward search according to the rules defined in Equation 2.8. A term M is called an inhabitant of type τ if the corresponding type judgment holds: $\Gamma \vdash M:\tau$. From a logical perspective, the terms are proof for the type in the context Γ, and therefore arrow types correspond to the implication. The $\to_E$ rule describes the modus ponens, and the components represent logical combinators.

For the problem $\Gamma \vdash M:\tau_0$, the inhabitation algorithm searches for applicative compositions by selecting a combinator $(X:\sigma)$ from the repository Γ. Provided that σ has the form $\sigma = \tau_1 \to ... \to \tau_m \to \tau_0$, the algorithm tries to find matching terms $M_1, ..., M_m$ such that $\Gamma \vdash M_i : \tau_i$ for i in $[1, ..., m]$. τ_0 is called the target of σ or the target type of the corresponding combinator, and X applied to the determined terms will yield an inhabitant of τ_0.

2.3 CLS Framework

The CLS framework [1] allows practical use of the synthesis approach, and the dissertation of Jan Bessai contains a detailed description of its theoretical foundations [11]. The framework uses algebraic specifications through type signatures to achieve language independence. The underlying algebra allows abstraction over different calculi and systems. This specification allows constructing a combinatory logic context Γ that is used for synthesis. While the un-

[1] https://github.com/combinators/cls-scala

derlying type system is built upon the definitions in Sections 2.2.1 and 2.2.2, it also includes constructor types and product types.

The framework implements an algorithm for solving the inhabitation enumeration question. It employs a tree-based enumeration strategy to return a list of inhabitants instead of just answering the decision problem for inhabitation. More precisely, it builds a grammar that helps to perform a lazy enumeration of inhabitants. This way, also infinite solution spaces can be represented.

The implementation language Scala allows the integration of a broad range of JVM libraries to support the framework and the definition of Scala-based combinators. CLS has been used in a variety of scientific works. Domains of applications cover planning workflows [100], factory planning [101], business process generation [14], health care processes [85], applications in logistics [15], and generation of simulation models [98, 52, 53].

2.3.1 Kinding and Taxomomy

With type variables, CLS allows the specification of type schemes that transform into intersection types with a user-defined substitution map. A corresponding kinding uses a substitution map to provide variable substitutions. CLS uses suitable substitutions for the type variables of every combinator and builds an intersection type that captures all possible variations.

The framework resolves the different variable assignments to form corresponding closed sorts, i.e., specifications without type variables. Therefore, after performing the substitution, the inhabitation algorithm does not need to consider type variables. Hence, the resulting inhabitation problem corresponds to the FCL inhabitation problem for component-based synthesis, which can increase performance. The user-defined restriction of valid substitutions can be considered a modeling feature.

A partial application of a substitution for a type variable to a type scheme is not admissible. That means that, given a type scheme $\alpha \to \alpha$ and a substitution map $S = \{\alpha \mapsto (\tau, \sigma)\}$, the translated intersection type will be $(\tau \to \tau) \cap (\sigma \to \sigma)$. The substituted type will not contain the types $\sigma \to \tau$ or $\tau \to \sigma$.

2.3.2 Repository Construction

The construction of a repository can involve a compile-time static reflection mechanism or the dynamic addition of combinators. The CLS framework offers a convenient way to define combinators containing Scala code. The framework employs the Scala reflection API

to investigate traits representing the repository specification and builds a corresponding reflective repository. The annotation *@combinator* signals the relevance of a combinator for the construction of the repository. The *apply* method provides the native type signature and the wrapped code, and the value *semanticType* holds the combinator's semantic type specification.

Alternatively, dynamic construction of the repository is possible. The CLS API provides the *addCombinator* method, which includes a component in the synthesis. The added Scala object must have an *apply* function and a value *semanticType*. A Scala program can use this operation to implement a flexible repository construction suited to a given use case.

2.3.3 Example

The following example illustrates how CLS synthesizes terms from a repository and how semantic types can describe the properties of components. It features a basic repository for the generation of machining operations with corresponding tool usage. Table 2.1 shows the collection of available components and their types.

Combinator Name	*Type*
DrillingToolComponent	$tool \cap drilling$
MillingToolFinishing	$tool \cap finishing$
MillingToolRoughing	$tool \cap roughing$
DrillingOperation	$tool \cap drilling \rightarrow machiningOp \cap drilling$
MillingOperation	$tool \cap milling \rightarrow machiningOp \cap milling$
MillingOperationVar	$tool \cap alpha \rightarrow machiningOp \cap alpha$
MachiningProcess	$tool \cap drilling \rightarrow tool \cap milling \rightarrow machiningProcess$

Table 2.1: Example Repository: Machining

The type $tool$ describes an arbitrary tool, and $machiningOp$ denotes a machining operation. $machiningOp \cap drilling$ annotates drilling operations, and correspondingly, the semantic type $tool \cap drilling$ denotes a tool for drilling operations. The semantic types $finishing$ and $roughing$ describe special milling operations. The semantic taxonomy introduces a subtyping relation which expresses that $finishing$ and $roughing$ are subtypes of $milling$, i.e., $finishing \leq milling$ and $roughing \leq milling$. Moreover, to demonstrate the function of type variables, the variable α can be substituted with the types $finishing$ and $roughing$, i.e., the substitution map is $S = \{\alpha \mapsto (finishing, roughing)\}$.

The combinator `MillingOperation` works with subtyping. Therefore, it accepts any kind of milling tool as an argument and yields a milling operation. Equations 2.12 and 2.13 show that

the subtyping rule allows the given terms to inhabit the more general type $machiningOp$. However, the `MillingOperation` can only yield the type $machiningOp \cap milling$ as shown in Equation 2.11 and is therefore not considered in an inhabitation request that asks for a particular kind of milling operation, such as $\Gamma \vdash ? : machiningOp \cap roughing$ (see Equation 2.10).

The type scheme of combinator `MillingOperationVar` is resolved to the intersection type shown in Equation 2.9. It can therefore construct terms that describe machining operations more precisely, as shown in Equations 2.14 and 2.15.

$$\begin{aligned}&(tool \cap roughing \rightarrow machiningOp \cap roughing) \cap \\ &(tool \cap finishing \rightarrow machiningOp \cap finishing)\end{aligned} \tag{2.9}$$

The component `MachiningProcess` requires two different tools, and its type includes two arguments. The first argument a_1 must satisfy the type judgment $\Gamma \vdash a_1 : tool \cap drilling$ while the second term a_2 must satisfy $\Gamma \vdash a_2 : tool \cap milling$. According to the $(\rightarrow_E)$ rule in Equation 2.8, this component can only yield a term that inhabits its target type $machiningProcess$ if the inhabitation algorithm finds well-typed terms for both argument positions.

$$\Gamma \nvdash \text{MillingOperation(MillingToolRoughing)} : machiningOp \cap roughing \tag{2.10}$$
$$\Gamma \vdash \text{MillingOperation(ToolFinishing)} : machiningOp \cap milling \tag{2.11}$$
$$\Gamma \vdash \text{MillingOperation(ToolFinishing)} : machiningOp \tag{2.12}$$
$$\Gamma \vdash \text{MillingOperation(ToolRoughing)} : machiningOp \tag{2.13}$$
$$\Gamma \vdash \text{MillingOperationVar(ToolRoughing)} : machiningOp \cap roughing \tag{2.14}$$
$$\Gamma \vdash \text{MillingOperationVar(ToolFinishing)} : machiningOp \cap finishing \tag{2.15}$$

Chapter 3

Motion Planning

This work includes three CLS repositories which represent algorithmic families and allow synthesizing corresponding planning algorithm variations. These algorithmic families are:

- Combinatorial Motion Planning
- Sampling-based Motion Planning
- Domain-Specific Coverage Path Planning

The Chapters 4, 5, and 6 demonstrate how components cover variability points of the particular planning domain and how component-based synthesis can configure possible algorithm variations. This automated configuration with CLS leads to new methods of finding solutions for complex planning tasks.

This chapter will define the motion planning problems and introduce the basic notions. There is a wide range of literature covering the field of motion planning. In particular, LaValle [61] and Latombe [60] provide a remarkable introduction to the relevant concepts and algorithms. The planning tasks in this thesis are limited to global planning, which means that the planning environment is static and known. Global planning algorithms can calculate results a priori, which can further support application-specific planning tasks.

3.1 The general Piano Movers Problem

- Let $\mathbb{W}$ be a **world** or workspace in which the robotic system operates
- Let $\mathbb{O}$ be a set of **obstacles**

- Let $\mathscr{C}$ be a **configuration space**, which describes possible robot configurations
- Let $\mathbb{R}$ be a representation of the the **robot model**, which is either a semi-algebraic space or a point
- Let $q_0 \in \mathscr{C}$ be a **start** configuration and $q_G \in \mathscr{C}$ a **goal** configuration of the robotic system
- Let $t_{\mathbb{R}}$: $\mathscr{C} \rightarrow \mathbb{W}$ be a robot-specific **transform function** that yields a geometric representation in $\mathbb{W}$ for every configuration c

The general *Piano Movers Problem* is finding a continuous path from the start configuration q_0 to the goal configuration q_G of the robot. The workspace $\mathbb{W}$ is usually bounded, and it corresponds to the operational area for the given navigation task. Thus, every point of the continuous result path must map to a geometric space contained in the workspace for a given robot geometry $\mathbb{R}$ and its transform function $t_{\mathbb{R}}$. Moreover, a path must be collision-free, i.e., there is no contact of the robot model with the obstacles $\mathbb{O}$.

3.2 Geometric Representation

The workspace is a continuous geometric space that is usually bounded, and it prescribes the planning domain. An essential factor for the applicability of an algorithm is the planning dimension. Moreover, the properties and the representation of contained geometries influence the applicability of a motion planning algorithm. This work assumes that a semi-algebraic model representation describes environment obstacle regions and the robot shape. From a practical perspective, the most common semi-algebraic data structures are **polygons and polyhedra**.

A standard encoding of polygons consists of vertices that form a ring in a predefined order. This order can specify either the enclosed or outside spaces as part of the polygon. That way, a formulation of polygons with holes is possible, as shown in Fig. 3.1. The arrows illustrate the order of the sequence, and in this example, a counter-clockwise order encloses a space while the clockwise order excludes a space from the geometry. The same principle applies to polyhedra, where the normal vectors of its facets distinguish semi-algebraic spaces.

Motion planning algorithms differ in their robustness regarding geometric flaws, such as overlapping geometries or self-intersections. Moreover, some algorithms only work for convex shapes as their technique does not apply to non-convex forms. Hence, the geometric properties of the planning problem directly impact the applicability of algorithms and algorithmic families.

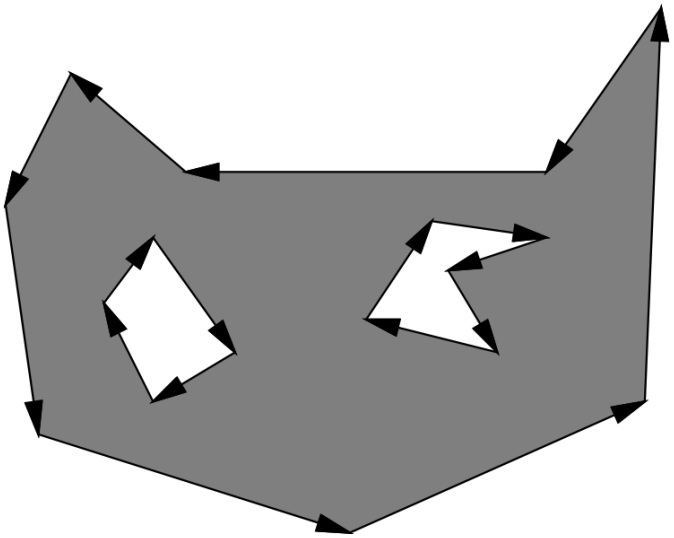


Figure 3.1: Polygon with holes defined by ordered sequences of points, taken from [61]

An affine transformation can produce concrete vertex positions upon application to **geometric primitives**. For convex primitives, the affine transformation will always yield convex objects because it can only describe translation, rotation, reflection, shearing, and scaling. Consequently, the obstacle region of convex shapes can be described by the convex hull of the given vertices.

3D volumetric **meshes** built from voxels are a common format to encode polyhedrons of complex geometric objects. Alternatively, these objects can also be described by surface meshes. Both mesh formats usually hold a list of points in $\mathbb{R}^3$ and a set of either tetrahedra or triangles defined by list indices. They often come in a data format transferable to other domains such as CAD software, 3D graphic engines, or simulations.

Another approach to describe objects is **constructive solid geometry** (CSG). The application of intersection, union, and difference operations to geometric primitive leads to complex objects. A CSG tree is a possible encoding of a series of operations, and CSG frameworks usually offer means to transfer this tree to a mesh that approximates the resulting shape.

3.2.1 Continuous Path Representation

Given a start configuration q_0 and a goal configuration q_G, the solution of the generalized piano movers problem is a continuous map $r : [0,1] \rightarrow \mathscr{C}$, with $r(0) = q_0$ and $r(1) = q_G$. For a valid path, the codomain is $\mathscr{C}_{free}$ so that every configuration in the image of r transforms to a collision-free geometry in the workspace.

Most motion planning algorithms yield a sequence of configurations that forms a series of path segments. In this case, a linear interpolation transforms these segments to a continuous solution path. Given two points $p1, p2 \in \mathbb{R}^3$, the points of the connecting path can be calculated from the parametric equation for a line segment, which is shown in Equation 3.1.

$$x = \begin{bmatrix} p1_x \\ p1_y \\ p1_z \end{bmatrix} + t \begin{bmatrix} p2_x - p1_x \\ p2_y - p1_y \\ p2_z - p1_z \end{bmatrix} \text{ for } 0 \leq t \leq 1 \tag{3.1}$$

3.3 Configuration Space

The configuration space $\mathscr{C}$ is an abstract notion for the space of all configurations for a particular robot system, which is also called C-space. The robot system's degrees of freedom (DOF) specify the dimensions of this configuration space, and any particular configuration, denoted q, is a point in the robot's state system.

Fig. 3.2 shows a simple example of such a configuration space. It represents a robot with two degrees of freedom, θ_1 and θ_2. A transformation to a 2D representation involves separating the space along the lines suggested in the left figure. This operation maps points on the surface to the new configuration space, and the positions R, P, F, and G are orientation marks. As the resulting dimensions represent angles, the dimensions span from 0 to 2π.

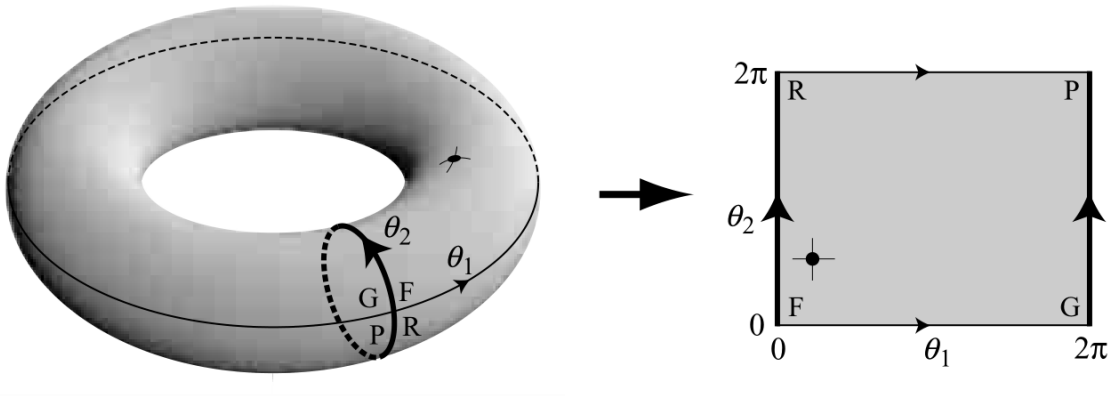


Figure 3.2: Workspace covered by different robot configurations, taken from [22]

Fig 3.3 displays the corresponding transformations and the area describing the attainable points of the effector. It can reach any point in the circle by choosing a corresponding point of the robot's configuration space that specifies suitable values for θ_1 and θ_2. The mathematical process to calculate the joint positions and angles that are needed to place a robot's end-

effector at a specific position and orientation is called **inverse kinematics**. Inverse kinematics usually involves the formulation of an equation system based on the transformations imposed by the variables of the configuration system. The equations include the desired effector position as an additional constraint. The corresponding system must consider the robot's shapes and degrees of freedom. Therefore, inverse kinematics solvers are specifically designed for a particular robot or families of robotic systems.

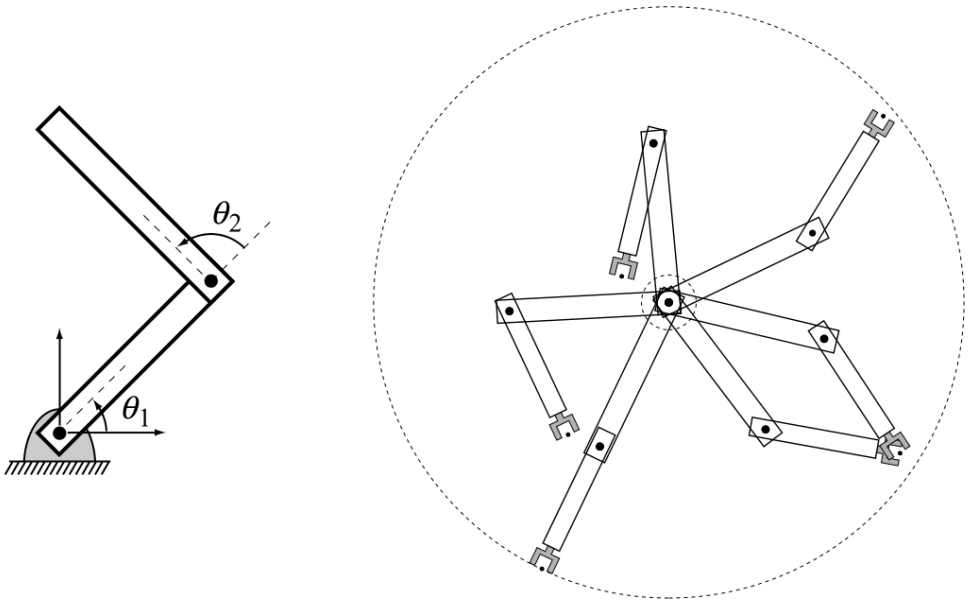


Figure 3.3: Representation for a configuration space, taken from [22]

3.4 Transformations

Transformations are complete bijective functions that map geometric objects to a set of points. They describe the position, orientation, and shape of geometric objects in the environment or encode a change of an object's pose. When applied to geometric primitives, these functions help to define semi-algebraic spaces that characterize obstacles' placement and shape. In motion planning algorithms, transformations describe a change of the robot's position and orientation. The robot's model is a geometric shape that comes with its own coordinate system, which is also called *frame* of a robot.

The robot-specific transformation $t_{\mathbb{R}}$ applied to a configuration $q \in \mathscr{C}$ yields a geometric space that can be checked for overlaps with obstacles $\mathbb{O}$. The affine transformation can always specify a rotation and a translation at the same time. As the rotation part is intended to

describe the robot's orientation, a suitable origin of the coordinate system for the given robot frame could be its center of mass.

3.4.1 Rotation

Fig. 3.4 shows an example for a rotation that transforms a two-dimensional workspace. This operation rotates the object given by the points A, P, and Q about the point A through the positive angle α. The resulting object is enclosed by the points $AP'Q'$. A rotation is an isometric operation, i.e., it preserves the distance between two points. As a consequence, the angles β and γ are equal. The 2D rotation matrix R_{2d} for rotations about the coordinate system's origin through the angle α is shown in Equation 3.2. For every point $p \in \mathbb{R}^2$, the rotation yields a new p' given by the matrix multiplication $(R_{2d}(\alpha))p^T$.

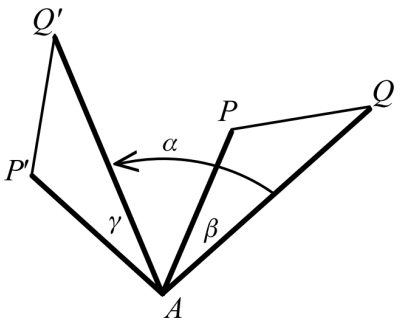


Figure 3.4: Rotation in 2d about point through angle alpha, taken from [43]

The principle is transferable to the three-dimensional space, but instead of selecting a reference point, a rotation now refers to an axis. Any possible rotation consists of a series of non-associative rotations over angles about the x, y, and z-axes. To fully describe a rotation, such a parameterization with **Euler angles** refers to either the intrinsic axis of the object or the fixed axes of the workspace. Moreover, the corresponding rotations must apply to an axis in conjunction with a predefined rotation order. The matrix representation for rotations about the x, y, and z-axes are displayed in Equations 3.3 to 3.5. While Euler angles are a comprehensible illustration of 3D rotations and allow for easy encoding in matrices, they have several drawbacks. For instance, this system can lead to the loss of a dimension, called a *gimbal lock*, which can occur when two axes align due to preceding rotations. This property limits the applicability of this principle to motion planning algorithms as it restricts the composition of rotations and interpolation capabilities.

The **axis–angle representation** is more suitable for the given domain. In this case, a complete rotation description consists of a normalized vector and an angle. The corresponding rotation follows the right-hand rule, and the rotation axis is called the Euler axis.

$$R_{2d}(\alpha) = \begin{pmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{pmatrix} \tag{3.2}$$

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{pmatrix} \tag{3.3}$$

$$R_y(\alpha) = \begin{pmatrix} cos(\alpha) & 0 & -sin(\alpha) \\ 0 & 1 & 0 \\ sin(\alpha) & 0 & cos(\alpha) \end{pmatrix} \tag{3.4}$$

$$R_z(\alpha) = \begin{pmatrix} cos(\alpha) & sin(\alpha) & 0 \\ -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.5}$$

Quaternions are 4-tuples, and their applicability to rotations originates from an extensive mathematical framework that exceeds the scope of this work. However, from a practical perspective, quaternions expose different strengths that favor their utilization in motion planning. For instance, they come with a compact data representation that allows memory-efficient storage. Moreover, quaternions can express sequences of rotations with simple mathematical operations, which is more efficient than applying different rotations in sequence to geometric objects.

The different approaches can be converted to an affine transformation matrix representation. A summary of different data structures for rotations and their conversions can be found in [26]. The corresponding matrix represents an element of the special orthogonal group $SO(3)$, an algebraic group of distance preserving transformations of a three-dimensional Euclidean space.

3.4.2 3D Rigid Body Planning

Given a 3×3 rotation matrix $R \in SO(3)$ and a translation vector defined by $p \in \mathbb{R}^3$, the matrix displayed in Equation 3.6 describes the complete affine transformation which can calculate the new points of a rigid body.

$$T = \begin{pmatrix} R_{11} & R_{12} & R_{13} & p_1 \\ R_{21} & R_{22} & R_{23} & p_2 \\ R_{31} & R_{32} & R_{33} & p_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \in SE(3) \tag{3.6}$$

The corresponding transformation is part of the special Euclidean group $SE(3)$ for the three-dimensional Euclidean space. The transformation adapts the properties of $SO(3)$ and extends the group by translation. Therefore, it does not cover reflection, shearing, or scaling and describes only the transformations applicable to rigid body planning (i.e., a robot without movable parts) as intended by the piano movers problem. This matrix applied to the semi-algebraic shape describing the robot will yield a new shape with updated position and orientation. The transformations allow mapping the configuration space to spaces of the workspace, which can be checked for validity by the planning algorithms.

Following the same principle, robot-specific transformations can apply to different movable parts of the robot. In the case of a robot described in Section 3.3, one might not only be interested in the position of the effector but also in the position and orientation of the robot's arm segments. With this information, motion planning can also avoid collisions between obstacles and the robot's arm. The resulting robot-specific transformation must adapt to the configuration space. Here, θ_1 determines the orientation of the first robot arm and, therefore, the link's position between the segments. The second arm must be translated accordingly, and its orientation depends on the orientation of the first arm and the value of θ_2.

While an extension of the planning space is generally possible, the planning problems in this thesis are limited to rigid body planning and transformations in $SE(3)$.

3.5 Combinatorial Motion Planning

Combinatorial motion planning algorithms work with explicit geometric representations of the obstacles, from which they conclude the free configuration space of the robot. In practice, the representation of these obstacles uses polygons or polyhedra. A spatial segmentation

procedure analyzes $\mathscr{C}$ and aims to find subspaces, called free cells. These non-overlapping cells usually completely cover $\mathscr{C}_{free}$. Correspondingly, for a given convex cell, the connection of two contained points is also in $\mathscr{C}_{free}$ and therefore a valid path segment. The algorithmic family uses roadmaps to represent the connectivity between free cells, and the traversal of this graph can solve the motion planning problem.

Combinatorial motion planning algorithms are often exact, i.e., the decomposed free configuration space exactly covers $\mathscr{C}_{free}$. Consequently, these algorithms are complete, meaning they will find a path if such a path exists. These properties, however, directly depend on the decomposition techniques employed.

Some techniques, such as the vertical cell decomposition, are limited to two-dimensional planning problems. While the extension to higher dimensional spaces is theoretically possible with plane sweeps, the algorithms and decomposition techniques do not scale well. Moreover, a high-dimensional configuration space can require incorporating information about the robot during the cell decomposition. Such robot-specific decomposition methods further limit the applicability of these algorithms.

3.6 Sampling-Based Motion Planning

Sampling-based motion planning is a class of heuristics that operates by relaxing the completeness property (finding a path if such path exists) to probabilistic completeness (finding a path if such path exists when executing infinite iterations). The planning discipline is capable of solving challenging problems in practical applications with high-dimensional planning domains. This family of planning algorithms performs the search by generating samples in the robot's configuration space, checking their validity for a known, static environment, and building a graph representing the free configuration space.

Sampling-based motion planning algorithms can be classified by the incorporated graph structure, which allows for the distinction between tree-based planners and roadmap planners. A tree construction starts from the root node and considers samples as leaves. Therefore, cycles can not occur in trees, but they may be part of roadmaps. As in combinatorial motion planning, the graph includes undirected, weighted edges with positive weights that describe the distance between two states.

Figure 3.5 shows the graph construction, where a sample $\alpha(i)$ helps to extend the graph structure. The tree root is the configuration state q_0, and a former iteration has connected the node q_n to q_0. The sample $\alpha(i)$ is located in the infeasible area $\mathscr{C}_{obs}$. A heuristic selects the tree node q_n and handles the invalid sample by finding q_s. The resulting roadmap allows

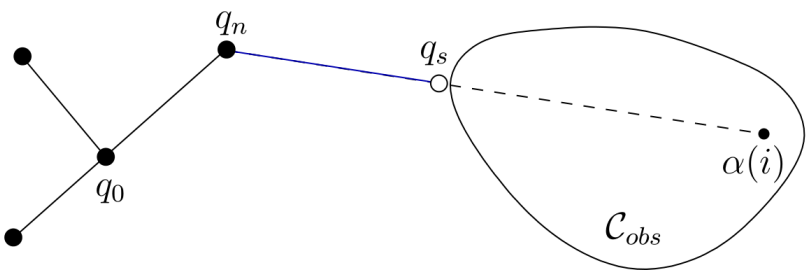


Figure 3.5: Adding samples to the graph, taken from [61]

solving the planning problem with a graph search algorithm like the A* algorithm. This outline already gives a rough idea that the algorithmic family of sampling-based motion planning algorithms is a space with high variability. Possible variability points include different data structures, the determination of connecting nodes, sampling strategy, and treatment of invalid samples.

Sampling-based methods use functions that determine if any given point or path segment collides with any obstacle. However, these algorithms are not limited to geometric representations and can work with arbitrary definitions of valid states. Thus, robot-specific validity models can be incorporated with an adaption of corresponding validity functions. The algorithmic family scales well with higher dimensions and can incorporate different configuration spaces. Sampling-based motion planning algorithms work for a broad range of robots with different degrees of freedom and adapt well to practical planning challenges.

Sampling-based motion planning algorithms are, in theory, asymptotically optimal as the solution path converges to a global optimum with a growing number of iterations. In practice, limited computation time due to real-time requirements or problem-specific limitations prevents optimal planning results. Aiming for a lower computation time, some planners involve termination upon finding a feasible path rather than optimizing the result. Local planning techniques can supplement these initial results to smooth and simplify paths [36] or account for dynamic changes in the environment [20].

3.7 Coverage Path Planning

The field of path planning originated in robotics and has a broad spectrum of applications that includes floor cleaning, harvesting or spraying in agriculture, demining, or painting. With drones and rising applications for autonomous robots, this planning discipline finds increasing interest in the scientific community. The planning techniques aim to find a continuous path

that passes a sensor with a particular field of view, a tool, or an abstract geometric shape over a specified area so that every point of the area is covered. The application to real-life planning scenarios usually comes with additional domain-specific optimality objectives or planning constraints.

A planner is complete when it is guaranteed to find a path if such a path exists. Due to the domain of interest, this property is a crucial characteristic of a coverage path planning algorithm. Another planning criterion is the admissibility of overlapping paths, which must find domain-specific consideration.

Coverage path planners often use heuristics to cover a given area. These heuristics may, however, not apply to more complex geometric forms. Hence, many coverage planning approaches involve decomposing the target space to break down the planning task to cover simple cells. Eventually, a continuous path can be determined by traversing every cell and connecting the respective paths.

Chapter 4

Composition of Combinatorial Motion Planning Algorithms

This chapter introduces a repository to synthesize combinatorial motion planning algorithms that solve planning problems for a point robot in two-dimensional and three-dimensional workspaces. It demonstrates how the semantic types specify a configuration for this algorithmic family. For this purpose, a customized variable substitution map reduces the space of inhabitants, managing the otherwise extensive runtime for inhabitation. Type variables represent the variability points, and they are part of the specified arguments and the target type of a top-level combinator. The repository contains software components that implement the corresponding variations. It consists of 68 combinators, and 10 type variables can configure more than 28.880 unique algorithm variations. Hence, a program and its different mechanics can be precisely specified.

Moreover, different data structures handle different geometric representations of environments and obstacles. As algorithms or parts of algorithms require a specific geometric data structure, the repository contains corresponding data types and transformation functions, which must comply with the dimensionality of the workspace. For instance, obstacles can be cubes equipped with an affine transformation matrix or a representation of a surface mesh consisting of vertex lists and triangles.

The repository contains components that use Python scripts, Scala Code, and Java frameworks. An interface to the Unity[1] game engine allows for an easy specification of 2D and 3D motion planning problems. In Unity, the user can create obstacles in a scene and apply transforma-

[1] https://unity.com

tions to manipulate their shape. The repository also contains components that read 3D model data files for the environment and robot model.

The MQTT protocol permits the transfer of the workspace with contained obstacles to the Scala program, where a listener waits for messages to initiate the inhabitation and starts solving the motion planning problem upon receiving matching arguments over the specified MQTT topic.

Equation 4.1 shows the semantic type of the top-level combinator. Such a combinator constructs the resulting program, and the inhabitation algorithm attempts to resolve its arguments. First, the combinator's implementation transforms the environment geometry to the corresponding data format using the supplied transform function. After that, the cell decomposition function is applied, which yields a planning environment object containing a cell segmentation. The function for the graph construction uses this intermediate result for its computation.

The resulting graph represents the roadmap that provides the basis for the graph traversal, which finds the overall result for the specified motion planning problem. The repository supports the customizable construction of the roadmap, specified by semantic types.

$$
\begin{aligned}
&(sd\_unity\_scene\_type \rightarrow sd\_polygon\_scene\_type) \cap dimensionality\_var \rightarrow \\
&cmp\_sceneSegFct\_type \cap sd\_poly\_scene\_cell\_segmentation\_var \cap \\
&\quad dimensionality\_var \rightarrow \\
&cmp\_cell\_graph\_fct\_type \cap rmc\_cellNodeAddFct\_var \cap \\
&\quad rmc\_startGoalFct\_var \cap rmc\_usingCentroids\_var \cap \\
&\quad rmc\_centroidFct\_var \cap sd\_cell\_type\_var \cap rmc\_cellGraph\_var \cap \\
&\quad rmc\_connectorNodes\_var \cap dimensionality\_var \rightarrow \\
&cmp\_graph\_algorithm\_var \rightarrow \\
&cmp\_algorithm\_type \cap cmp\_graph\_algorithm\_var \cap \\
&\quad rmc\_connectorNodes\_var \cap rmc\_centroidFct\_var \cap rmc\_cellGraph\_var \cap \\
&\quad sd\_cell\_type\_var \cap sd\_poly\_scene\_cell\_segmentation\_var \cap \\
&\quad dimensionality\_var \cap rmc\_cellNodeAddFct\_var \cap \\
&\quad rmc\_startGoalFct\_var \cap rmc\_usingCentroids\_var
\end{aligned}
\tag{4.1}
$$

4.1 Data Structures and Native Types

The top-level combinator builds a data object that contains the individual result of each planning step. The corresponding class represents a segmented environment, the roadmap, and the resulting path. For this reason, the data structure contains the following fields:

- Vertices

  A list of vertices represents the global points in the workspace. The $\mathbb{R}^2$ or $\mathbb{R}^3$ position of each vertex is stored in a list of float values.

- Obstacles

  This list describes the obstacles in the environment. Each obstacle is a list of integers, and these values can resolve to $\mathbb{R}^3$ points by referencing the corresponding element of the vertex array. For the given segmentation techniques, obstacles are convex shapes. Therefore, an unordered list is a sufficient representation for obstacle regions.

- Boundaries

  A list of three float values describes the size of the planning dimensions. The origin of the coordinate system represents the center of the planning environment.

- Free Cells

  The free cells are the result of the cell decomposition. Following the same principle as described for obstacles, this field is a nested list of integer values.

- Roadmap

  The roadmap is a *Graph for Scala*[2] object with undirected, weighted edges with positive values. The labeled nodes represent points in the workspace $\mathbb{W}$, and the edge weights denote the Euclidean distance between points.

- Path

  The path is the overall result of the combinatorial motion planning algorithm, and this list of points describes a continuous path from the start node to the end node.

The repository works with data structures containing only a subset of these fields. These native types encode the availability of partial results such as free cells or the roadmap.

[2] http://scala-graph.org/

4.2 Cell Segmentation

The repository covers the following segmentation variations:

- Vertical cell decomposition
- Grid-based cell approximation
- Triangulation and tetrahedralization

The vertical cell decomposition and the grid-based approximation have been implemented in Scala, while triangulation and tetrahedralization make use of Python scripts.

4.2.1 Grid-Based Cells

The grid-based approach to represent $\mathscr{C}_{free}$ leads to an approximation of the free configuration space, and the resulting algorithm configuration is therefore not complete.

The grid resolution is configurable and given as the number of cells per dimension. The division of the planning space by the maximum number of cells in the corresponding dimension yields the vertical and horizontal distances of neighboring tiles. Grid-based cells allow for the easy identification of neighboring cells which helps the construction of the roadmap.

A grid cell is only valid when it does not overlap with any obstacle, and the roadmap connects only adjacent valid cells. Fig. 4.1 shows free cells constructed by a grid-based approximation of $\mathscr{C}_{free}$. The red dots represent the start point and endpoint, gray quadrangles are obstacles, and green lines illustrate the determined cells.

4.2.2 Triangulation and Tetrahedralization

Delaunay triangulation and Delaunay tetrahedralization can divide a 2D space into triangles or a 3D space into tetrahedrons. The application of these operations to the free configuration space $\mathscr{C}_{free}$ yields geometric primitives that represent free cells for the construction of the roadmap.

The repository makes use of the Python library PyMesh[3] to perform these operations. For this reason, these decomposition variations have to transform obstacles into strings representing PyMesh mesh data objects and generate Python scripts accordingly. The $\mathscr{C}_{free}$ mesh

[3] https://github.com/PyMesh/PyMesh

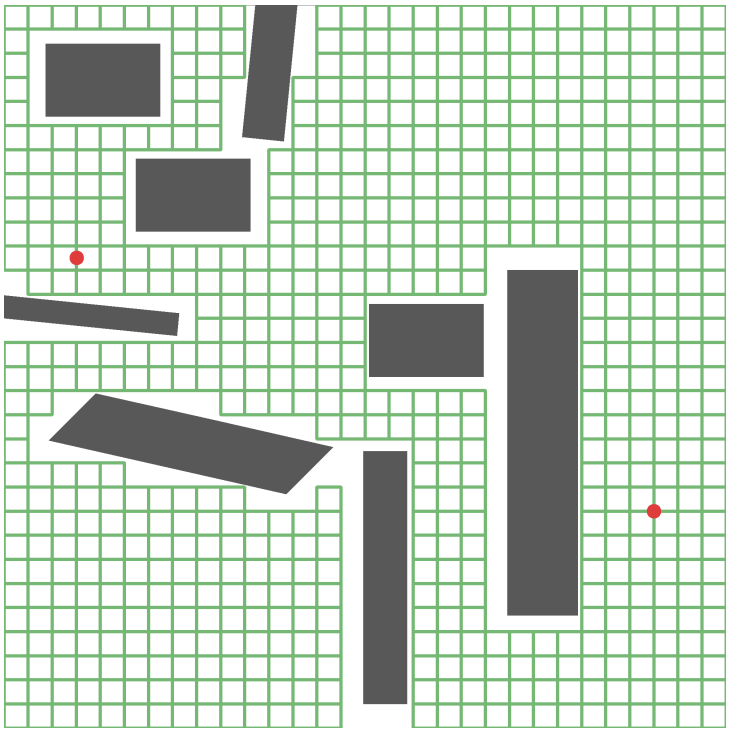


Figure 4.1: Free cells formed by a grid-based approximation of $\mathscr{C}_{free}$

data is formed by subtracting obstacles from a box mesh that complies with the dimension boundaries. The triangles of this resulting mesh resemble free cells, as they cover $\mathscr{C}_{free}$ completely.

Triangles can also be formed by explicitly calling triangulation functions of PyMesh with a given parameterization. The repository includes a parameterized triangulation to allow a maximal triangle area of 1000 arbitrary area units and minimal admissible interior angles of 20°. The execution of the generated Python scripts yields a JSON representation of the results that incorporates the vertices as float lists and the cells represented by a list of integers, respectively.

The unparameterized triangulation can cause irregular shapes, for which the link between centroids of neighboring triangles can lead to invalid paths. The semantic layer has therefore been adapted to express the requirement for additional graph nodes. Fig. 4.2 shows the cells, the resulting graph, and the path for a parameterized triangulation.

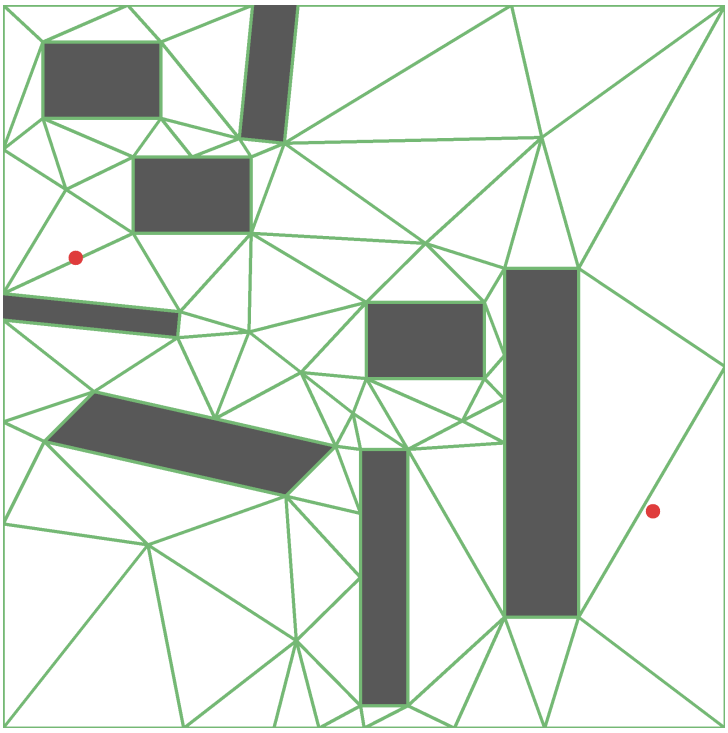


Figure 4.2: Decomposition of $\mathscr{C}_{free}$ using parameterized triangulation

4.2.3 Vertical Cell Decomposition

The vertical cell decomposition is a line-sweep algorithm that aims to build trapezoid cells by introducing vertical lines. These lines intersect with at least one obstacle and separate two cells, which can help to construct the roadmap. The algorithm moves through the environment in x-direction while opening and closing cells in the process. For this reason, the algorithm includes sorting of the vertices in ascending order concerning the x-value. Thus, the segmentation includes an obstacle-preserving transformation into an environment representation with sorted vertices.

The segmentation technique considers vertical lines for every vertex of the environment, and the construction of lines corresponds to the different cases displayed in Fig. 4.3. The procedure builds extending lines in positive or negative y-direction, according to the geometric case.

The sweep algorithm forms cells based on these lines by keeping track of opened cells, i.e., cells that require a closing line. A cell can be opened or closed by multiple horizontal lines, and it holds information about the enclosing obstacles to assign lines to cells. During sweep in the positive x-direction, there can be multiple opened cells with distinct pairs of bounding obstacles. Segmentation lines can also border on the upper and lower boundaries of the

planning space. For this reason, the vertical cell decomposition regards the corresponding boundary lines as obstacles.

Fig. 4.4 shows an example for the resulting free cells. This particular cell decomposition technique results in cells that are often irregular. As a consequence, the generated roadmap does not cover the free configuration space with consistent density.

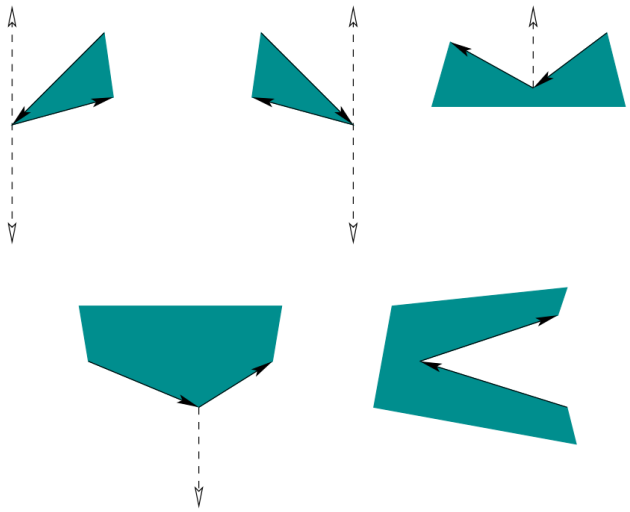


Figure 4.3: Cases for line construction in vertical cell decomposition, taken from [61]

4.3 Graph Construction

Based on the developed free cells, there are several ways to generate the roadmap which describes $\mathscr{C}_{free}$ by capturing the connectivity between free cells.

The graph construction components include type variables to precisely define this roadmap, and the corresponding roadmap combinator has the semantic type displayed in Equation 4.2. The use of type variables allows the roadmap construction to consider every aspect specified by a user-provided substitution map. The component requires the six functions as arguments supplied by the inhabitation algorithm according to the given kinding. These arguments represent variability points of roadmap construction, and they take care of the following tasks:

- Finding neighboring cells for every cell
- Computing the centroid for a cell
- Finding cell vertices

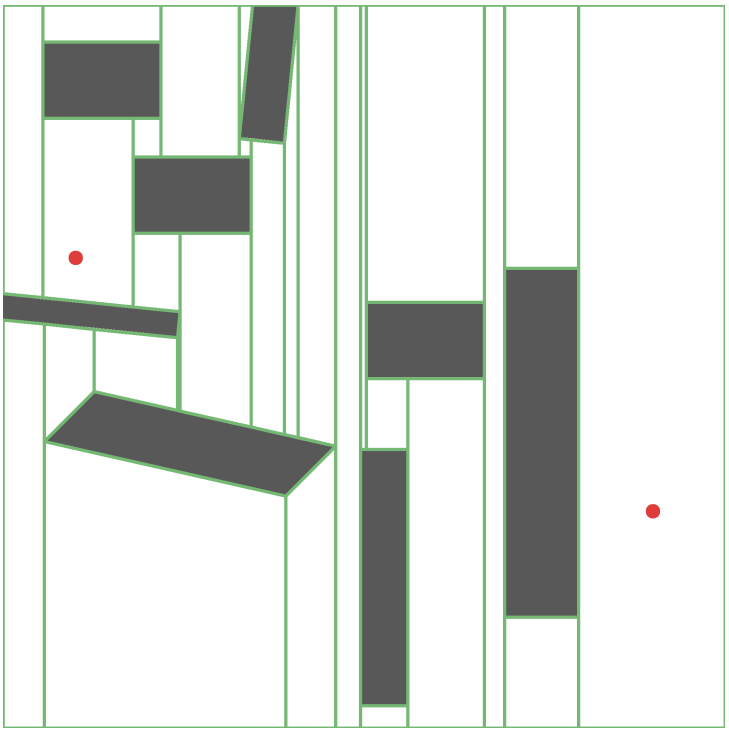


Figure 4.4: Vertical cell decomposition as an example for a 2D segmentation strategy

- Introducing additional nodes on edges that delimit at least two neighboring cells
- Building edges connecting graph nodes for single cells
- Connecting neighboring cells based on the respective cell graphs
- Adding start and goal nodes to the complete roadmap

Several components implement functions that perform these tasks and manipulate a graph object in the process. A higher-level roadmap component runs these functions for the given planning environment, determines the nodes, and builds a graph for every cell. The corresponding components collect nodes and edges for free cells or manipulate the overall graph directly.

This graph can include centroid nodes, the vertices that delimit the cell, or additional nodes on the boundaries of a cell. Corresponding components in the repository represent variations and provide the selection of these nodes. For instance, there are several ways to define centroids, depending on the geometry of the cell. Triangle cells can utilize the triangle's incenter point, while other shapes might require computing the center of mass or using the center of the axis-aligned bounding box of the cell.

Semantic types describe the absence of a feature explicitly. For instance, the use of centroid nodes is specified by the type variable $rmc\_usingCentroids\_var$ which can resolve to the semantic types $rm\_withCentroids\_type$ or $rm\_withoutCentroids\_type$. This explicit typing is necessary because the negation of a feature is not inherently expressible with intersection types. The existence of centroids requires consideration in several other arguments of the higher-level combinator, such as subgraph construction and connection of neighboring nodes.

The integration of start and goal nodes can determine an edge to the nearest node of the graph, the nearest centroid of the graph, or the nearest node of the containing cell. This function must comply with the preceded roadmap construction.

$$
\begin{aligned}
&rmc\_neighbourFct\_type \cap dimensionality\_var \rightarrow \\
&rmc\_centroidFct\_type \cap rmc\_usingCentroids\_var \cap \\
&\quad rmc\_centroidFct\_var \cap sd\_cell\_type\_var \cap dimensionality\_var \rightarrow \\
&rmc\_cellNodeAddFct\_type \cap rmc\_cellNodeAddFct\_var \rightarrow \\
&rmc\_connectorNodeFct\_type \cap rmc\_connectorNodes\_var \rightarrow \\
&rmc\_edgeAdd\_type \cap rmc\_cellGraph\_var \rightarrow \\
&rmc\_startGoalFct\_type \cap rmc\_startGoalFct\_var \cap \\
&\quad rmc\_cellGraph\_var \cap dimensionality\_var \rightarrow \\
&cmp\_cell\_graph\_fct\_type \cap rmc\_cellNodeAddFct\_var \cap \\
&\quad rmc\_startGoalFct\_var \cap rmc\_usingCentroids\_var \cap \\
&\quad rmc\_centroidFct\_var \cap sd\_cell\_type\_var \cap rmc\_cellGraph\_var \cap \\
&\quad rmc\_connectorNodes\_var \cap dimensionality\_var
\end{aligned}
\tag{4.2}
$$

4.4 Graph Traversal and Graph Search

The road map yields an undirected weighted graph, as described in Section 4.1. The graph is a *Graph for Scala* data structure, and the result of the graph traversal is a sequence of points. The native type of a shortest path graph search is

`(Graph[List[Float], WUnDiEdge], MpTaskStartGoal) => Seq[List[Float]]`,

where `MpTaskStartGoal` denotes a data type that contains start state q_0 and goal state q_G for the motion planning problem. Five combinator implementations represent different graph traversal techniques. The type variable $cmp\_graph\_algorithm\_var$ covers this particular variability point, and the repositories kinding allows the substitution with the following semantic types:

- $cmp\_graph\_dijkstra\_type$
- $cmp\_graph\_fw\_type$
- $cmp\_graph\_a\_star\_type$
- $cmp\_graph\_mst\_type$
- $cmp\_graph\_tsp\_type$

This repository contains three shortest path graph traversal algorithms, and two of them use SciPy[4] graph tools. For that reason, the repository contains transformation functions to translate graph data structures into a string representing a SciPy distance matrix. This string is inserted into a template, forming a runnable python script enriched with data. After executing the generated script, a parser analyzes the string from the console output to extract the result.

The **Dijkstra algorithm** [27] is a greedy algorithm that traverses the graph by selecting nodes from a set of unvisited nodes. These nodes hold a distance value that describes the shortest distance to the initial node. During initialization, these values are set to positive infinity while the start node holds the value 0. A node selection is made by choosing the lowest distance value, consequently removing it from the set of unvisited nodes. As this algorithm requires positive edge weights, the weight of the selected node is guaranteed to correspond to the shortest path length to the initial node. For every unvisited neighbor of the selected node, the distance value to the initial node updates if the path length of the current node added to the weight of the connecting edge is shorter than the currently held value. The algorithm terminates when the goal node is marked as visited. The combinator for this shortest path algorithm makes use of the Dijkstra implementation provided by the *Graph for Scala* library.

[4] https://www.scipy.org/

The **Floyd–Warshall algorithm** [31] was published in 1962 by Robert Floyd. The result of this algorithm contains the shortest path lengths between all pairs of nodes. The basic concept is estimating the shortest path length for vertex pairs that is incrementally improved until the estimate is optimal. For a given vertex pair, the estimate identifies irrelevant edges, and the calculations only incorporate edges that could be part of the shortest paths.

Hart, Nilsson, and Raphael first published the **A*** algorithm in 1968 [42]. A* is an informed shortest path algorithm that uses a heuristic function to choose the edges and nodes for evaluation. The algorithm is well-suited for routing problems [104], and the most prominent example for such a heuristic is the Euclidean distance function to the goal state. That way, A* incorporates geometric information to prioritize the evaluation of nodes that are closer to the goal node. Moreover, the function result can rule out that some nodes are part of the shortest path, which allows for an efficient implementation of a complete search algorithm.

The introduction of graph search algorithms that do not look for the shortest path extends the algorithmic family of combinatorial motion planning algorithms to work for further fields of application, such as routing. For instance, a **minimum spanning tree (MST)** computation finds a subgraph that allows access to every cell of the free configuration space. Hence, a depth-first traversal of the tree can produce the resulting path, which can eventually allow the visitation of every cell. A possible field of application could be supply chain routing problems. Figure 4.5 shows the resulting MST for a roadmap generated from triangle centroids.

Correspondingly, the graph structure can also be the basis to solve the relaxed **traveling salesman problem (TSP)** [6]. The resulting path allows the robot to visit every cell and eventually return to the initial starting point. In this repository, the corresponding graph search combinator transforms the graph to a distance matrix and applies the TSP solver of Google OR-Tools[5].

4.5 Variations and Results

The following results demonstrate different algorithm configurations with varying segmentation and roadmap generation techniques. Therefore, the kinding and the outcome are displayed in each case. Additionally, the Appendix A contains textual representations of the inhabitant trees.

Fig. 4.6 results from the type variable substitution in Equation 4.3 and uses a grid-based approximation to find free cells. The graph construction is based on cell centroids. Fig. 4.7 and Equation 4.4 represent a cell decomposition by triangulation and a graph construction

[5] https://github.com/google/or-tools

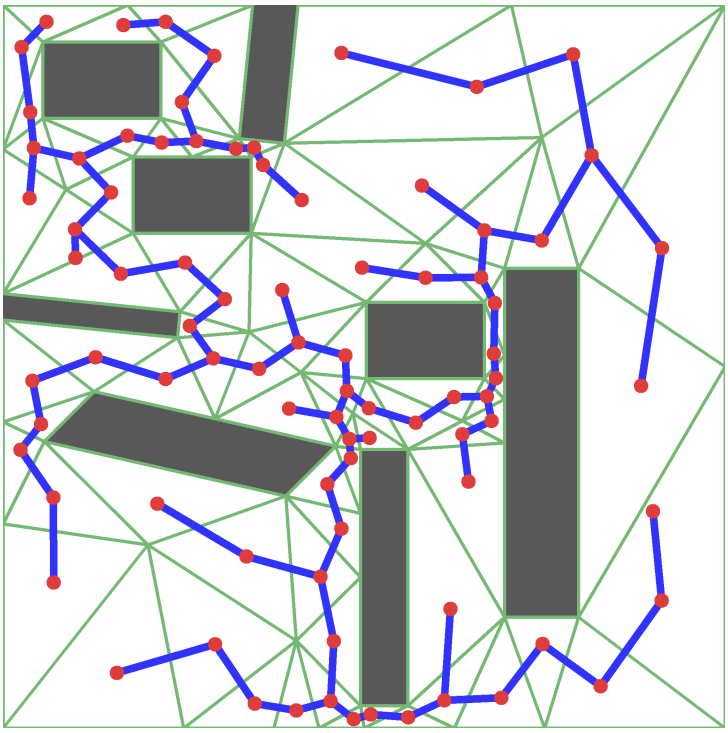


Figure 4.5: An example for a minimum spanning tree computed from the roadmap

that introduces connecting nodes between cells. The type variable assignment in Equation 4.5 leads to the result illustrated in Fig. 4.8. Vertical cell decomposition forms free cells, and the corresponding cell subgraphs contain centroids, cell vertices, and connecting nodes.

$$
\begin{aligned}
S = \{ & \\
& rmc\_connectorNodes\_var \mapsto rmc\_cn\_withoutConnectorNodes \\
& rmc\_cellNodeAddFct\_var \mapsto rmc\_cna\_withoutCellNodes\_type \\
& rmc\_cellGraph\_var \mapsto rmc\_cg\_centroidsOnly \\
& rmc\_usingCentroids\_var \mapsto rm\_withCentroids\_type \\
& rmc\_startGoalFct\_var \mapsto rmc\_startGoal\_nn\_type \\
& cmp\_graph\_algorithm\_var \mapsto cmp\_graph\_dijkstra\_type \\
& rmc\_centroidFct\_var \mapsto cFct\_centroids\_naive\_type \\
& dimensionality\_var \mapsto dimensionality\_two\_d\_t \\
& sd\_poly\_scene\_cell\_segmentation\_var \mapsto sd\_seg\_grid\_type \\
& sd\_cell\_type\_var \mapsto sd\_cell\_vertical\_type \\
\} &
\end{aligned}
\tag{4.3}
$$

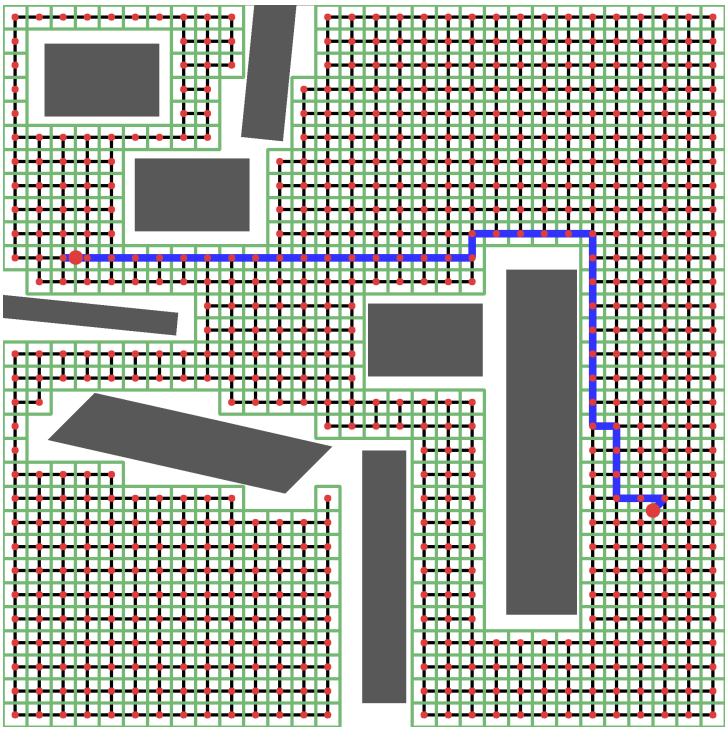


Figure 4.6: A grid-based approximation of $\mathscr{C}_{free}$ with a roadmap built from centroids

$$
\begin{aligned}
&S = \{ \\
&rmc\_connectorNodes\_var \mapsto rmc\_cn\_withConnectorNodes \\
&rmc\_cellNodeAddFct\_var \mapsto rmc\_cna\_withoutCellNodes\_type \\
&rmc\_cellGraph\_var \mapsto rmc\_cg\_centroidCellVertices \\
&rmc\_usingCentroids\_var \mapsto rm\_withCentroids\_type \\
&rmc\_startGoalFct\_var \mapsto rmc\_startGoal\_nn\_type \\
&cmp\_graph\_algorithm\_var \mapsto cmp\_graph\_dijkstra\_type \\
&rmc\_centroidFct\_var \mapsto cFct\_centroids\_naive\_type \\
&dimensionality\_var \mapsto dimensionality\_two\_d\_t \\
&sd\_poly\_scene\_cell\_segmentation\_var \mapsto sd\_seg\_triangles\_para\_type \\
&sd\_cell\_type\_var \mapsto sd\_cell\_triangle\_type \\
&\}
\end{aligned}
\tag{4.4}
$$

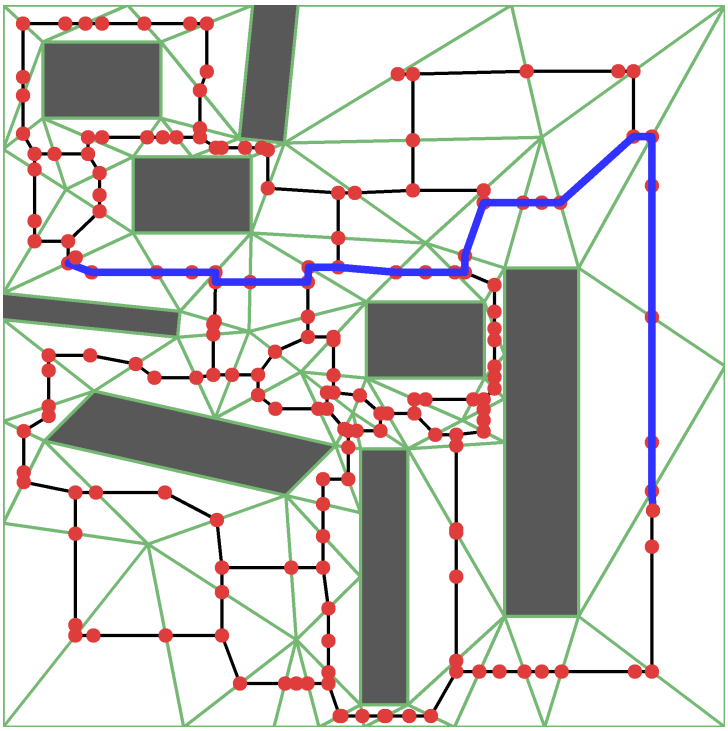


Figure 4.7: Cell decomposition by triangulation and graph construction with connecting nodes

$$
\begin{aligned}
S = \{ & \\
& rmc\_connectorNodes\_var \mapsto rmc\_cn\_withConnectorNodes, \\
& rmc\_cellNodeAddFct\_var \mapsto rmc\_cna\_withCellNodes\_type, \\
& rmc\_cellGraph\_var \mapsto rmc\_cg\_allVertices, \\
& rmc\_usingCentroids\_var \mapsto rm\_withCentroids\_type, \\
& rmc\_startGoalFct\_var \mapsto rmc\_startGoal\_nn\_type, \\
& cmp\_graph\_algorithm\_var \mapsto cmp\_graph\_dijkstra\_type, \\
& rmc\_centroidFct\_var \mapsto cFct\_centroids\_naive\_type, \\
& dimensionality\_var \mapsto dimensionality\_two\_d\_t, \\
& sd\_poly\_scene\_cell\_segmentation\_var \mapsto sd\_vertical\_cell\_decomposition\_type, \\
& sd\_cell\_type\_var \mapsto sd\_cell\_vertical\_type \\
\} &
\end{aligned}
\tag{4.5}
$$

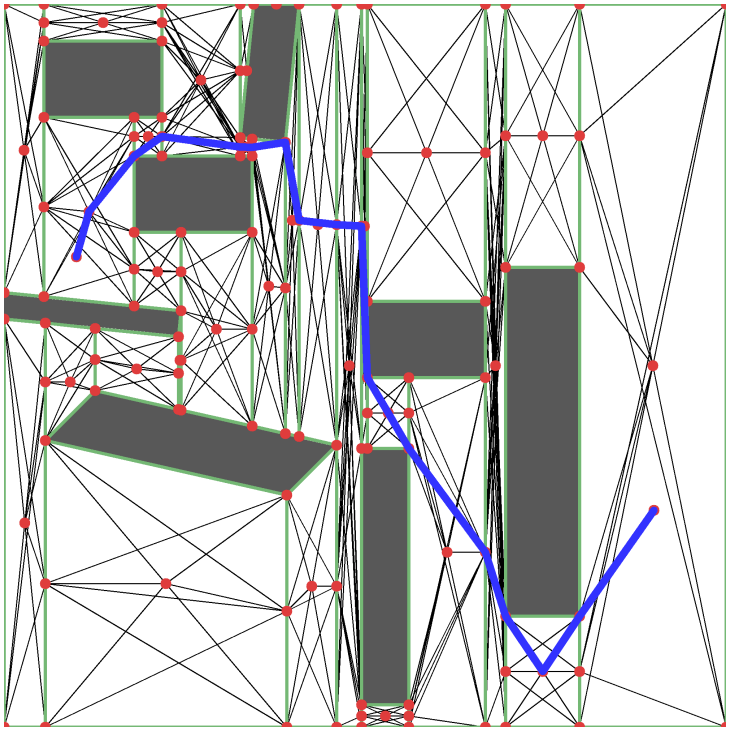


Figure 4.8: Vertical cell decomposition with centroids and connecting nodes. The cells' subgraphs contain edges from cell vertices to connecting nodes and centroids.

4.6 Discussion

The cell segmentation subroutines form free cells based on the obstacles' vertexes. Vertical cell decomposition illustrates that the cell formation depends on the planning environment geometries and can produce irregular cells resulting in imbalanced graph node density. Moreover, cell decomposition and graph construction cover regions that might not be relevant to the planning problem. However, the steps produce a complete roadmap, which allows for multiple motion planning queries.

Vertex-based cell segmentation techniques prohibit some geometric representations. For instance, a semi-algebraic definition of a circle by a center point and radius must be approximated with a polygon, undermining the completeness property. Moreover, the segmentation strategies are vulnerable to geometric flaws, such as overlapping obstacles, obstacles that exceed the planning boundaries, or non-manifold topologies. For this reason, the component for vertical cell decomposition employs a function to cut obstacles so that their vertices are within the planning boundaries.

The incorporation of robot models in combinatorial motion planning algorithms is only possible to some extend. For instance, robot geometries with a fixed orientation can be regarded by replacing obstacles with the Minkowski sum of obstacles and the robot shape. However, this algorithmic family does not support higher-dimensional configuration spaces.

While the analysis of combinatorial motion planning is interesting from an algorithmic perspective, it exposes limitations for the application in real-life planning scenarios.

Chapter 5

Component-Based Synthesis for Tool Path Planning

The contents of this chapter have been partially published in [86]. Further remarks on this publication are included in the Appendix B.

The ongoing trend for the customization of products and production systems enforces the necessity for adaptable process planning, which the scientific community acknowledged [70]. New automated manufacturing methods must deal with these requirements. The compensation of the shortage of skilled workers is an essential aspect of digitalization that requires automated workflows requiring little human interaction.

In machining, Computer-Aided Manufacturing (CAM) systems support the planning personnel by semi-automated means. With the rising variety of workpiece shapes and part requirements, this planning support is insufficient as it still involves the manual operation of trained technicians. For instance, this CAM planning includes manually selecting process parameters, such as tooling, and assigning tool path planning patterns to selected workpiece shapes.

The milling of complex parts is a domain that exhibits variability and works well with the component-based principle of CLS. This chapter presents a technique to find different machining strategies, consisting of machining operations that use different tools. The approach uses CLS to compose individual planning components to form machining strategies.

The applicability of a machining operation (i.e., planning program) depends on the targeted geometry. For instance, some planning components require an accessible area that allows an initial positioning of the tool. The result of a single planning step is a single connected tool

path. The order in which planning components are applied matters, as they lead to different machined areas, machining time, and tool wear. The tool selection also influences these aspects. For instance, narrow areas might only be accessible for tools with a small diameter.

CLS synthesizes path planning programs from a repository of domain-specific software components. These components can be classified as:

- Planning components
- Model transformations without tool interaction
- Predefined machining sequences
- Tool data

An inhabitation request yields any planning program that matches a user-provided configuration. The generated programs expose high structural variability, and they translate to a sequence of machining operations with a corresponding evaluation procedure. Each inhabitant represents one path planning program that applies machining operations to a geometric shape in a specific order. The generated programs transform the geometric model that considers the machining task, the machined area, the left-over material, and workpiece boundaries. The result of program execution is a series of tool paths and a transformed model.

This work generates tool planning programs fitted to different materials, namely aluminum and steel. The semantic layer of the repository is an encoding of this domain knowledge. Corresponding type specifications determine the suitability of a tool for a given material and the material-specific applicability of a planning component. The machining of steel exposes higher forces on the machining center and the tool, and correspondingly, the methodology avoids planning components that might include temporary demanding tool engagements. With consideration of these aspects, it is possible to reduce the cost of tooling and machine maintenance.

This technique performs process planning and tool path planning to produce precise descriptions of manufacturing steps. The elements of the result set are evaluated in multiple stages to find a set of solution candidates that can facilitate planning workflows. Further processing could incorporate simulations, the use of expert systems, or optimization techniques. Also, the generated tool paths could help to improve the efficiency of semi-automated planning with CAM software.

It is possible to encode principles of process planning in the semantic layer of the repository. The repository contains domain knowledge such as the incorporation of roughing and finishing machining operations. Roughing aims to remove large amounts of material quickly, while

finishing aims to produce high-quality surfaces. Some combinators represent predefined machining sequences that consist of matching roughing and finishing operations. This encoding efficiently narrows down the search space for the brute-force approach.

This work benefits from the enumeration capabilities of CLS, which incorporates a lazy retrieval of inhabitants from an infinite result set. The search on this result set only considers well-typed terms that fit the target specification, and correspondingly, the search space is the space of inhabitants. Due to the discrete space of inhabitants and the candidate selection procedure, the methodology resembles a well-typed brute-force technique to find suitable machining strategies.

5.1 Planning Problem Instance

Tool path planning extends the coverage planning problem described in Section 3.7 with domain-specific constraints. This work targets **three-axis end milling** tasks, which allow the positioning of the tool center in $\mathbb{R}^3$. In machining, a common technique to plan tool paths for 3D volumes is the projection to horizontal planes. The corresponding surface is called 2.5D, and the milling tool is positioned perpendicular to this XY plane. This projection causes a reduction of the planning space to two dimensions, allowing the use of two-dimensional coverage path planning patterns. The experiments conducted in this chapter consider a depth of cut of 3 millimeters and involve only single-layered operations. However, extending this methodology to more complex three-dimensional shapes is possible by *slicing* volumes in the z-direction, yielding an individual planning task for each resulting layer.

An initial rectangular workpiece shape describes the planning space, and the planning task is given by a target shape that must be machined. The intended final workpiece consists of shapes that may not be machined and correspond to obstacle regions. Thus, the planning space is a two-dimensional, bounded, Euclidean workspace that contains polygonal obstacles. The initial path coverage planning is considered holonomic, i.e., the planning components do not take machine dynamics into account. However, the evaluation procedure for machining sequences considers this aspect in a post-processing step.

The tool is the object following the resulting path, and its geometric representation is a disk that matches the tool's diameter. Correspondingly, the planning aim is to find a tool path, such that every point of the target geometry was at some time covered by the tool geometry. A tool position describes the center point of the disk geometry representing the tool. The problem-specific validity of a tool position depends on the current configuration, the tool parameterization, and the planning environment given by the workpiece.

Contrary to conventional coverage path planning techniques, this approach does not use decomposition strategies and does not aim to find a single continuous path. It aims to find machining sequences that consist of different machining operations with different tools. The proof-of-concept considers two tools per material, one for roughing and one for finishing. The integration of additional tools or parameter sets for existing tools is possible and requires only little effort.

5.1.1 Assessment of Tool Engagements

The evaluation of machining strategies considers different aspects of productivity. The most important criterion is the successful completion of the planning task, i.e., the trajectory of the tool must cover every point of the target geometry while respecting workpiece constraints.

Another primary criterion for the evaluation of machining strategies is the overall **machining time** to produce the outcome. The machining time affects the overall energy consumption and the required amount of cooling fluids or the utilization of pressurized air. As planning primitives expose different material removal rates, the selection of planning components impacts the cost-efficiency of a machining process. A summarization of the different factors for the energy demand of machining operations with special regard to tool paths can be found in [7].

While a calculation of the running time of a machining sequence is possible, the analysis of **tool wear** is not trivial and subject to ongoing research. An automated measurement or quantification of tool wear is therefore not integrated as a numeric criterion.

An excessive tool temperature reduces the wear resistance of the tool's material and facilitates damage at the cutting edges due to cut forces (see [75], p. 46). The **chip formation** and the **chip evacuation** behavior have an impact on the tool temperature because chips carry away a significant share of the heat (see [21]) that originates in the primary and secondary shear zones (see Fig. 5.1). Another notable factor for tool wear is the **tool engagement angle**, an angular measurement that describes the partition of the tool's circumference that is in contact with the material. Small tool engagement angles reduce spindle load and allow higher feed rates while increasing the overall machining time. Fig. 5.2 shows the different engagement angles for conventional milling and trochoidal milling, where the tool progression involves circular motions that are illustrated in the bottom right corner of the figure.

Fig. 5.3 shows a selection of machining parameters for a side milling process, and the set of parameters complies with the end milling operations used in this chapter. The tool's cut width a_e, spindle speed n, number of teeth z, and feed rate v_f result in a particular feed per tooth f_z and a corresponding chip thickness. For this work, the compliance of a tool engagement

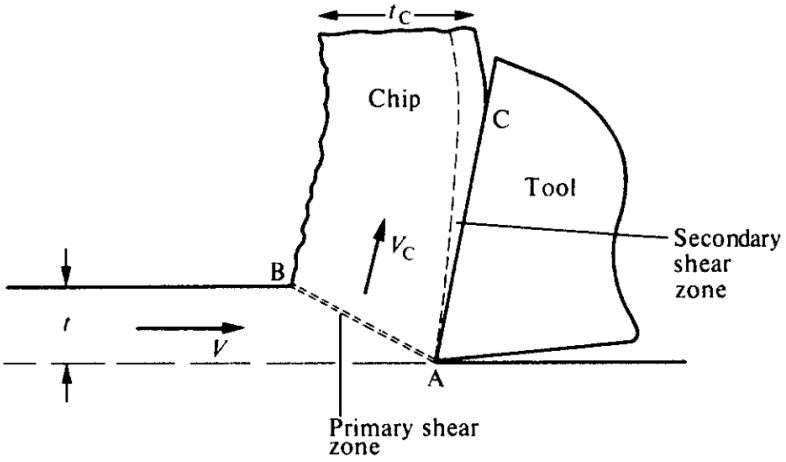


Figure 5.1: Chip formation and shear zones, taken from [87]

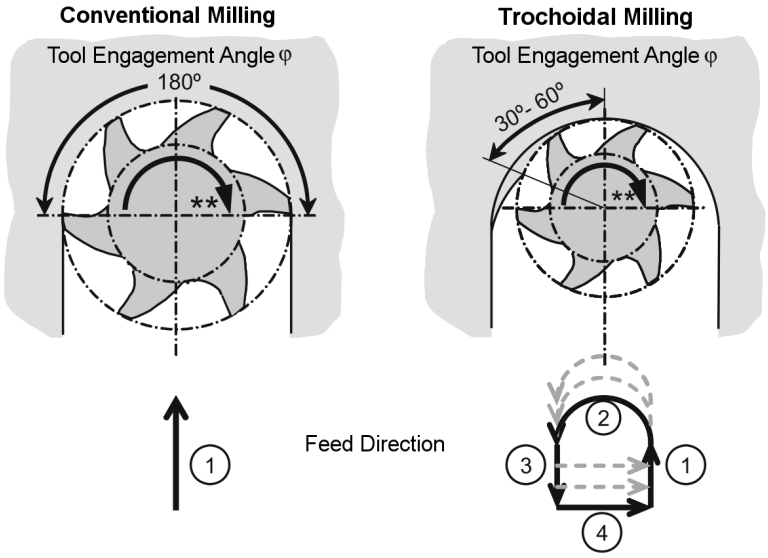


Figure 5.2: Tool engagement angle for conventional and trochoidal milling, taken from [58]

with these tool parameters is assumed to ensure stable cutting conditions. Consequently, this avoids unfavorable engagement effects such as chip thinning.

The characteristics of tool engagement situations also have an impact on the stability of the cutting process. Unplanned tool engagements might lead to oscillations which can cause higher tool wear, tool damage, and excess material removal. In worst-case scenarios, fatal tool damage can affect the machining center, workpiece, and workplace environment.

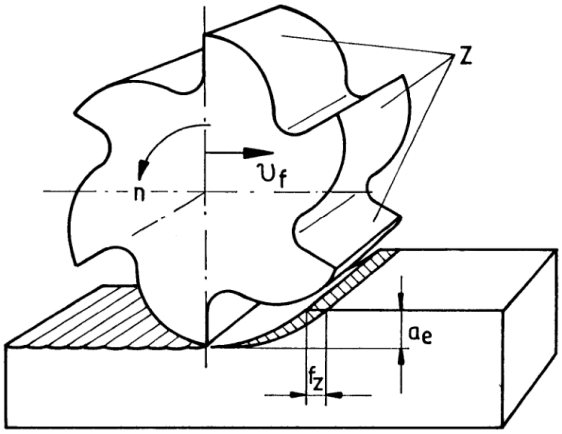


Figure 5.3: Machining parameters, taken from [75]

The tool parameterization also significantly impacts tool path planning, and it predetermines the planning parameters for the corresponding path coverage problem. The planning components, which were designed for this study, follow the parameterization. As such, the tool engagement preferably considers the cut width while moving at the desired feed rate. However, in corner cases, the path planning heuristics do not entirely conform to these parameters for the benefit of faster material removal, higher applicability to geometries, or ease of implementation.

This aspect affects the suitability of a motion plan for different materials, as the machining of hard materials imposes higher forces on the tool. Correspondingly, the planning components for these materials should emphasize compliance with the given parameterization. This study proposes different path planning components that favor either short machining time or reduction of tool wear.

5.1.2 Tool Parameterization

The tool selection must consider several aspects, such as the suitability for a material, depth of cut, the tool diameter, number of teeth, and the tool's price. The different tool properties affect its suitability for specific machining operations like finishing and roughing. The parameters

for tool engagement are well-matched, and compliance with these parameters leads to stable processes, constant chip evacuation, and controlled temperature development.

The component-based approach to synthesize machining operations involves selected tools with predefined parameters that trained professionals determined. The parameterization stems from previous work done by other researchers who integrated experimental force models in a geometric physically-based simulation system.

The corresponding tool components have semantic types that describe the tool's suitability for a milling operation (i.e., roughing or finishing processes) and the suitability for a material. This study uses the following tools and parameters.

Aluminum roughing:

Diameter ⊘:	12.0 mm
Width of cut a_e:	3.0 mm
Feed rate v_f:	3990.0 mm/min
Number of teeth z:	2
Corner radius r_c:	2.0 mm
Spindle speed n:	13300 min^{-1}

Aluminum finishing:

Diameter ⊘:	8.0 mm
Width of cut a_e:	4.0 mm
Feed rate v_f:	1280.0 mm/min
Number of teeth z:	2
Corner radius r_c:	0.0 mm
Spindle speed n:	8000 min^{-1}

Steel roughing:

Diameter ⊘:	10.0 mm
Width of cut a_e:	6.0 mm
Feed rate v_f:	1948.0 mm/min
Number of teeth z:	4
Corner radius r_c:	1.0 mm
Spindle speed n:	4775 min^{-1}

Steel finishing:

Diameter ⌀:	5.0 mm
Width of cut a_e:	0.5 mm
Feed rate v_f:	380.0 mm/min
Number of teeth z:	4
Corner radius r_c:	0.0 mm
Spindle speed n:	6365 min^{-1}

5.2 Literature Review

5.2.1 Optimization-based Tool Path Generation

Machining is often an optimization problem with several opposing objectives. As an example, the multi-objective optimization of milling parameters in [103] considers the trade-off between energy, production rate, and cutting quality. The following works employ optimization techniques for specific applications by introducing adapted parameter sets and optimization criteria.

In [57], a multi-objective evolutionary algorithm is employed to optimize the tool orientation for five-axis milling. This approach aims to smooth tool paths that expose tool positions that match the surface normals. That way, potential surface defects can be avoided. In [32], process parameters are tuned for different machining operations with fixed tool paths. The incorporated objectives are machining time, left-over material, and machining error as the deviation between the modeled and machined surface. The method yields optimized parameters that describe tool selection, tool parameterization, and geometric constraints such as admissible scallop height. These parameters have an impact on the path generation according to a predefined tool path pattern.

In [65], Lu et al. propose a differential evolution algorithm with gradient-based mutation to produce smooth tool paths for five-axis flank milling, which enables the machining of narrow objects like turbine blades. For that reason, the objective function is a mathematical notion for surface smoothness, while geometric deviations are constraint functions. The result is a smooth tool path described by a sequence of tool positions and orientations. Hsieh and Chu propose a different optimization procedure for the same domain [44]. They use variants of particle swarm optimization to minimize an estimate for the machining error. Another application of particle-swarm optimization allows the tuning of feed rate and spindle speed for existing tool paths [93]. The problem formulation consists of constraints describing the

machining conditions and a formula for the production cost as the objective function. These metrics allow the formulation of the overall production cost, which involves the machining time and a tool life estimate based on experimental findings.

In [1], a modified ant colony optimization approach is used to machine separated pieces of left-over material from a preceding contour-parallel machining operation. That way, the resulting machining sequence covers the complete machining task.

These works optimize machining conditions based on existing tool paths that entail construction with CAM software. This requirement reduces the potentially available solutions and limits the automated search for machining strategies. Methodologies that build paths from scratch target small planning tasks consisting of a single machining step, as the extensive search space does not allow generating sequences of machining operations.

In contrast, the methodology proposed in this work produces machining strategies with corresponding tool path patterns. As such, it covers the fields of process planning and tool path planning.

5.2.2 Formal Methods in Machining

In [72, 2], an approach for the modeling of manufacturing operations and geometric shapes uses formal methods, namely the Z notation. The formalism allows for representing geometries with semi-algebraic spaces and the encoding of operations by preconditions and postconditions. The Z notation primarily consists of the first-order predicate calculus with types.

In [73], this logical framework describes process planning by considering planning as an optimization problem. The corresponding formulation minimizes a problem-specific property under the constraints imposed by other formulas. The work shows how the notion can describe a process planning objective that aims to minimize energy consumption.

The application of this formalization in an actual planning program is not part of the proposal. However, a reasoner to construct formulas that describe sequences of manufacturing processes could implement a corresponding process planning methodology.

Process planning is on a higher abstraction level that does not include the planning of tool paths. While the formal description of machining operations with Z could be appropriate for process planning, the intersection type system of CLS specifies the rules of composition. Types express features of software components, effectively guiding automatic composition

on a combinator level, which allows the generation of large-scale machining sequences that provide the corresponding tool plans.

5.2.3 Conventional Planning Methods

Early tool path planning approaches make use of the general coverage path planning principles pointed out in Section 3.7. In [63], the tool path generation for free-form surfaces uses a grid representation of the target geometry. This method is essentially a projection of the 3D shape onto a two-dimensional planning space. An application-specific cost map describes the changes in the z-direction, and a corresponding weighted graph helps to find minimum cost connections. Eventually, these subgraphs enable a continuous tool path calculation using a minimum spanning tree (MST).

In [48], the medial axis, a subset of the Voronoi diagram, is used to identify *critical* regions that might expose unfavorable tool engagement situations when machined with a contour-parallel pattern. These regions are machined with a trochoidal grooving operation, segmenting the geometric space into smaller areas that are easier to machine. From a planning perspective, this operation corresponds to a cell decomposition, a common approach to path coverage planning.

5.2.4 Constant Tool Engagement

Several scientific works are motivated by the requirement for constant tool engagement. Early work by Uddin et al. identifies this problem and proposes a tool displacement strategy for machining corner regions [94]. A similar approach addresses long curved tool paths, and it performs the tool path modification with an offset curve [25]. Both approaches rely on initial tool paths for their computations. Therefore, these methods are tool path modification techniques rather than tool path planning approaches.

A planning approach that takes this aspect into account uses a contour parallel algorithm that iteratively checks tool engagement angles [29]. A path displacement is performed during computation when engagement angles exceed the predefined range of admissible values.

5.2.5 Planning with Feature Extraction

Several works propose process planning methods with feature-based methods to recognize geometric features and assign corresponding manufacturing processes. The primary purpose of process planning is to find a sequence of abstract manufacturing processes. In an early

work from 2002, Sadaiah et al. used feature extraction to implement a generative process planning system. They identify contained geometric primitives for complex geometries and provide a mapping to machining operations.

Huang et al. [47] present a multi-level structuralized model representation that holds information about the geometry and derives machining-specific features for manufacturing. The incorporated tree model is comparable to the CSG approach, and feature extraction helps to identify reusable subparts. Complete tool paths for complex geometries are built from user-defined paths for subparts, resulting in a semi-automated approach that reduces human interaction.

Xu and Hinduja [102] make use of feature recognition to assign roughing, finishing, and semi-finishing operations to surfaces. In contrast to the work presented in this thesis, they assign operations without tool path generation. Therefore, the planning domain exhibits a higher level of abstraction. The geometries consider 2.5D planning with multiple layers that approximate volumes. They use fuzzy logic to mitigate errors in feature recognition and use an iterative approach that considers intermediate states of the volume. The methodology involves a sub-division of the model to geometric primitives, similar to the subtractive operations on geometric models employed in this thesis.

5.3 Brute-Force Search

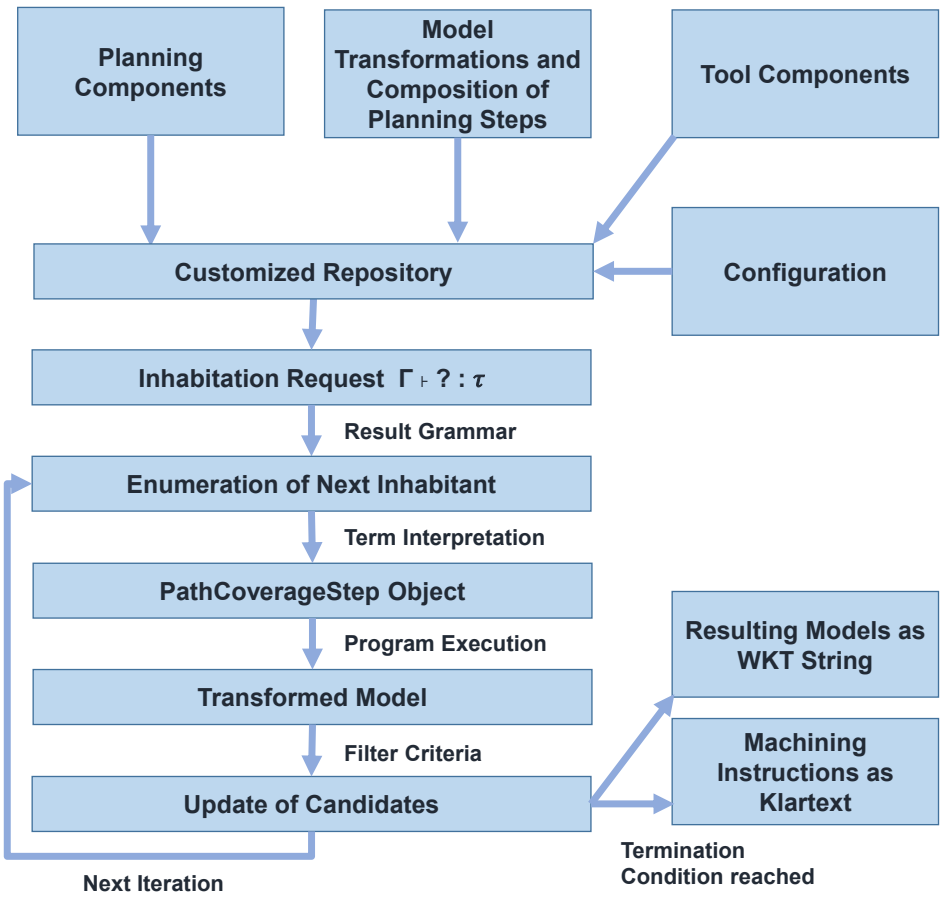


Figure 5.4: Synthesis of machining strategies and search procedure

Fig. 5.4 shows the procedure to find candidate solutions for a given machining problem instance. It involves a brute-force search that utilizes the inhabitant enumeration of the CLS framework.

The repository contains planning components, tool components, and components that perform additional model transformations or allow the composition of planning steps. The initial configuration of the repository considers the distinction between steel and aluminum experiments. The problem configuration arranges the kinding (i.e., the substitution of type variables) and the inhabitation target accordingly. Moreover, the experimental setup allows addressing open pocket or closed pocket milling tasks. Closed pockets will require machining sequences that start with a dive-in motion in the z-direction.

CLS enumerates the infinite set of synthesis results, and the found inhabitants are data structures representing the path coverage programs. The programs' execution starts with applying the planning functions to a geometric model and a planning configuration. The inhabitants are filtered by examination of the resulting updated model and the sequence of tool paths. In general, not every sequence of machining operations is guaranteed to fulfill any machining task. For this reason, the first filter criterion uses the remaining target geometry of the updated model to check if the current machining strategy completes the machining task. Further computation is performed only for programs that remove a specific percentage of the initial target area.

For selected planning programs, a post-processing step computes the tool feeds for every machining operation. This data forms the basis for estimates of the total processing time of machining strategies. A customizable filter criterion on the estimated time leads to a set of solution candidates for further investigation.

The brute-force search will continue enumerating and evaluating inhabitants until it finds a predefined number of solution candidates. Additional filtering leads to a refined set of candidate solutions that can be supplied to the planning personnel. Further evaluation of candidate solutions could include using geometric physically-based simulation systems, optimization techniques, or decision support systems.

This work uses the LocationTech JTS Topology Suite[1] for representation and manipulation of geometries. The implementation of several data types and functions supports the construction and evaluation of tool path planning. The corresponding native types are Scala classes that hold the information required to determine tool paths. The remainder of this section describes these data structures in detail.

5.3.1 CNC Model Representation

The model is a representation of the machining process at hand and is contained in the class `Cnc2DModel`. This model data type is sufficient to represent a 2D machining task and also intermediate geometric states. It contains the following fields:

- Planning Boundaries

  A list of float values represents the planning domain boundaries. It holds values for intervals given by minimal and maximal values for each geometric dimension $(x_{min}, x_{max}, y_{min}, y_{max})$.

[1] https://locationtech.github.io/jts/

- Target Area

  The target area denotes the geometry which the machining sequence must subtract to fulfill the specification. The final workpiece is the difference between the boundary rectangle and the targeted area.

- Machined Area

  A list of machined geometric objects represents the areas that were removed by preceding machining steps. This list describes the evolution of a model that can help to evaluate different steps of the machining sequence. A complete representation of the machined area can be determined by unionizing the elements of the list. The process terminates when the machined area equals the target area, and the result equals the specification.

- Remaining area

  This property is a list of geometric objects that must be subtracted from the model to complete the machining task, representing the remaining area that must be machined. In contrast to the target area, every planning step includes an update of this field. Therefore, it describes the current state of the remaining machining task.

The class `Cnc2DModel` includes several functions to manipulate its geometric models. These functions update the machined area and the remaining geometry, using several geometric operations to ensure robust computations.

5.3.2 Path Coverage Step

A successive application of path coverage functions manipulates the `Cnc2DModel`. These functions are wrapped in the class `PathCoverageStep`. It contains the following fields:

- Path Coverage Function

  A planning function has the native type `(Cnc2DModel, PathCoverageStepConfig)` → `(List[List[List[Float]]], Cnc2DModel)`. It receives a `Cnc2DModel` object and a set of configuration parameters as input, and generates a manipulated model and a sequence of paths. The updated `Cnc2DModel` includes a new geometric representation of the machined area and the remaining geometries.

- CNC Tool

  The tool parameters directly impact the resulting path as they configure the planning function, which uses the tool diameter and its cut width to find valid paths.

- Path Coverage Steps

  This list of machining operations contains child nodes of this path coverage step. The resulting tree structure resolves to a machining sequence by using an evaluation procedure.

- Description

  The description identifies the current planning component. The string can help to debug and denote machining instructions for export. The description states the name of the motion primitive and an identifier of the CNC tool.

With this data, a `PathCoverageStep` object can represent a single machining step and machining strategies consisting of multiple path coverage steps. A nested `PathCoverageStep` can contain several processing steps with different tools.

5.3.3 Path Coverage Configuration

The `PathCoverageConfiguration` object contains a set of parameters that allow tuning of the different steps that are required to execute a planning program and evaluate the result. The following fields configure the path generation behavior of planning components:

- Minimal path node clearance

  This value defines the minimal clearance between two consecutive points of a result path. Concatenation of different motion primitives or rounding errors in geometric operations can introduce close points, which can be problematic for the further evaluation of the tool path. Therefore, a violation of the minimal clearance constraint results in the collapse of two points by removing one point from the tool path. The default value for minimal clearance is 0.01 mm, corresponding to the tool deflection's order of magnitude caused by process dynamics.

- Area ignore threshold

  This technical parameter allows ignoring geometric artifacts under a given threshold. Planning components can include this value in a filter based on area size to disregard small areas. This way, geometric artifacts that result from rounding errors can be ignored.

- Path ignore threshold

  This parameter is in the sub-millimeter range and allows discarding path segments that result from rounding errors or limited applicability of planning components. Path coverage functions evaluate their resulting path length and only return planning results if the length of the determined path is greater than this threshold.

5.3.4 Path Coverage Result

The `PathCoverageResult` data structure contains the results of a planning program execution, and it is used for evaluation and debugging purposes. It can help to analyze the outcome of a sequence of machining operations by holding the geometric representation of the machined area model. The class offers utility functions to export the overall result for a machining step to XML files.

The evaluation procedure applies the planning functions of the given `PathCoverageStep` object to a model and a planning configuration. Upon application of the function to suitable arguments, the path and transformed model are built.

The evaluation of `PathCoverageStep` object corresponds to a depth-first traversal of this tree with pre-order. Each node will be supplied with the accumulated result of the preceding path coverage functions and performs its calculation based on the most current model state. For a given node, the evaluation procedure is implemented with a *fold* over the list of children, accumulating their partial results and applying their path planning functions to the updated list of models and paths. The outcome of a node corresponds to its result concatenated with the results of its children.

5.4 Path Planning Primitives

Every path planning primitive makes use of the data structures presented in Section 5.3. These primitives require a `CnCTool` object to determine the tool machining parameters such as cut width and tool diameter. The following collection of path planning primitives is not complete and can be further extended. However, this initial assembly of planning components shows that this technique can find candidate solutions for complex machining tasks.

5.4.1 Zig-Zag Patterns

The zig-zag and zig motions (Fig. 5.5) are seemingly simple machining patterns that employ straight, parallel tool paths. However, several variability points allow for different configurations of these motion primitives. These variability points may impact the suitability of a machining strategy for a given machining task. The zig pattern involves repositioning motions without tool engagement. While this behavior results in a lower overall material removal rate, the generated path can exclusively incorporate climb milling or conventional milling. By contrast, zig-zag alternates between climb milling and conventional milling.

The first stage of planning involves the construction of parallel lines according to the machinable target geometry. A geometric buffer applied to the final workpiece with a distance given by the tool radius yields the space of valid tool positions, i.e., the points that do not violate the final workpiece geometry. The geometric envelope of this shape intersected with the left-over material is the frame for the generation of vertical lines. The line displacement is based on the tool's cut width a_e to generate tool paths compliant with tool parameterization.

For these lines, the intersection with the geometry of the remaining area represents relevant tool path segments. In the most trivial case, those intersections match the original lines or contain one line per x-coordinate. However, this step can also generate multi-line patterns, i.e., multiple lines for a particular x-coordinate, as candidates for the zig-zag pattern. A loop iterates over these lines in ascending x-direction to decide if they can be considered in the tool path. In the case of a multi-line geometry, the planning component must select a line from the set of candidates for a given x-coordinate.

A function that computes a full tool path must connect the selected lines. The zig-zag pattern introduces alternating connections between the top and bottom nodes. On the other hand, the zig pattern connects either the bottom or top nodes of the lines. Therefore, these patterns are variations of the same program with different approaches for connecting consecutive lines.

The connection of nodes first conducts a validity check for a straight connection between selected nodes. This test validates if the tool path is within the space of valid tool positions, i.e., it does not violate workpiece boundaries. If a collision was detected, several horizontal connecting lines are generated over the y-interval that both lines share. They are checked for conformity with the constraints iteratively, and a valid connection segment is introduced by adding new nodes to the path.

This family of planning programs generates results that employ sudden changes of the tool's motion, which can be unfavorable for the machining center and the tool. The corresponding deceleration and acceleration phases affect chip evacuation, heat development, stress on the machining center, chatter characteristics, and machining time. It would be possible to

mitigate these effects by introducing an alternative connection routine that produces rounded corners. An additional finishing step could handle the additional leftover material.

The admissibility of higher tool engagement angles that do not comply with tool parameters is another variability point of this planning component. Such a tool engagement situation is demanding for the tool, resulting in higher tool wear. These patterns apply to a more extensive set of geometries but are not suitable for hard materials.

The compliance with tool parameterization can also be affected at the end of paths. The tool engagement angle rises at the end of the straight tool paths as it also removes leftover material ahead of the tool in the feed direction. The same principle applies to the connection of line segments.

Additional planning components can address these particular situations. While further considerations can benefit the tool engagement angle and lead to more constant forces, it limits the applicability to a target geometry. Corresponding implementations likely require supplementation by other path planning primitives, e.g., precedence of a trochoidal slot milling along the specimen contour.

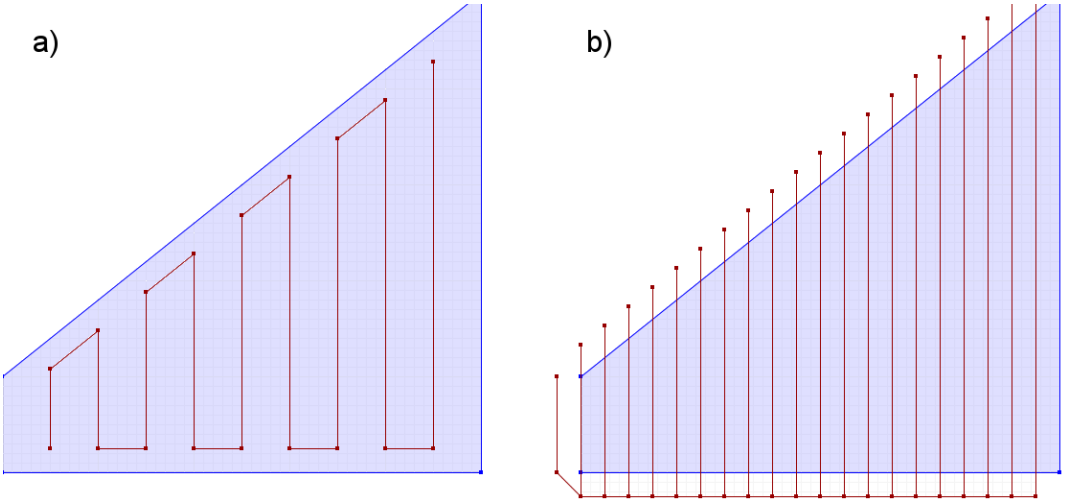


Figure 5.5: (a) Zig-zag pattern, (b) Zig pattern

5.4.2 Contour-Parallel Tool Paths

Contour-parallel tool paths (Fig. 5.6b) can orient themselves to the contour of the remaining geometry or the contour of the workpiece. The planning component iteratively determines paths to remove one material lane, and an iteration returns the updated machined area and a

new target polygon as a result. The overall result is built by connecting the iteration results with a domain-specific bug algorithm.

According to the tool's parameters, this planning primitive first finds the subtractable area for a selected target geometry. For this purpose, applying a geometric buffer operation expands the machined area by the tool's cut width. The workpiece geometry must be subtracted from this new polygon to avoid a violation of the workpiece boundaries. The tool path's computation selects a target area from the resulting geometry and uses a series of buffer and distance filtering operations to find the corresponding tool path.

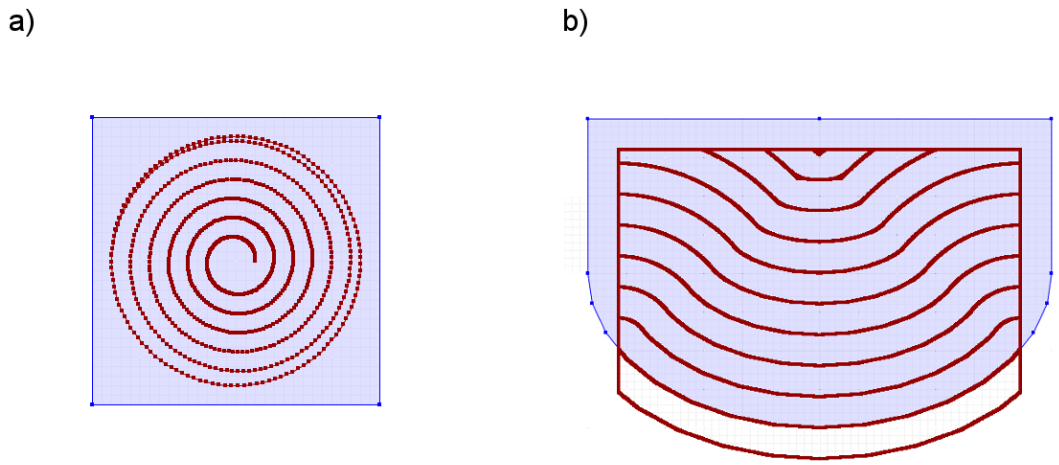


Figure 5.6: (a) Spiral roughing, (b) Contour-parallel tool path

Update of the Target Geometry

Each iteration, the latest contour path is buffered by the tool radius and removed from the remaining geometry. This operation might split the geometry into multiple parts. In that case, the planning primitive selects the remaining polygon with the largest area and continues the loop with an updated target geometry.

Every planning primitive contains a termination condition. For contour-based patterns, the calculation stops when an iteration fails to find a valid tool path or the selected remaining geometry is empty, i.e., the planning sequence completed the machining task.

Connecting Paths

Each iteration of this path planning function yields a single path, while the overall result of the planning component must be a connected path. Thus, a motion planning procedure ensures that the connecting tool path segments are collision-free concerning the remaining material. These repositioning motions must be located in the area of tool positions without tool engagement based on the current machined area.

A geometric buffer operation with a negative tool radius value applied to the machined area finds the space of admissible tool positions, which is also $\mathscr{C}_{free}$ for the given motion planning task. First, a validity check tests the straight line connecting the aggregated path's last point and the new path. If this trivial solution fulfills the planning problem, the result is the concatenation of the path segments.

If this is not possible, the tool path is computed based on the sequence of line segments that follows the contour of the leftover material. For this reason, two points in the machined area are selected by their minimal distance to the nodes of the tool paths, and the connecting path follows the exterior ring of the machined area shape. The function to connect path segments can be considered a domain-specific adaption of the bug algorithm [66, 67]. With this approach, the contour-parallel tool path planning component can produce valid paths for non-convex, irregular geometries with isles. The bug algorithm cannot find a connecting path for corner cases where the selected points are on separate polygons of the machined area. In this case, the planning component aborts the loop and returns the accumulated path.

Contour-Parallel Paths based on the Specimen

In this particular contour-parallel planning step, machining operations consist of a single step targeting the workpiece boundaries. This motion checks if it is possible to follow the workpiece contour while staying within tool parameters, i.e., the removed area must comply with the tool's cut width. With leftover material smaller than the tool's cut width, contour-parallel paths regarding workpiece and remaining target area will have the same outcome. This plan works well even if there are many separate leftover areas in the proximity of the workpiece boundaries (e.g., scallops). Instead of searching a planning primitive for each area, the contour-parallel path based on the specimen is one continuous path from a single planning step. Moreover, a finishing operation based on this planning component can account for the corner radius of the roughing tools.

5.4.3 Directed Trochoidal Slot Milling

In certain situations, milling with maximal tool engagement is not applicable. The directed trochoidal slot milling program, depicted in Fig. 5.7b, results in a straight slot in the y-direction, formed by a trochoidal base motion. First, an examination of the target geometry's vertices bordering the machined area finds the parameters of this machining operation. It determines the nodes adjacent to the machined area and uses a bounding ball to find the parameters for the circling motion, i.e., center position and radius. Based on these nodes, the planning primitive determines the first circle and selects points in the vicinity of the targeted polygon for the tool path construction. The planning component works iteratively and investigates circular tool paths displaced in the positive y-direction by the tool's cut width. It terminates when the new tool path contains infeasible points that violate the workpiece boundaries.

5.4.4 Contour-Parallel Trochoidal Slot Milling

Fig. 5.7a shows a planning primitive which generates a channel that follows the boundary of the workpiece. It can provide an initial machining step for a closed pocked machining task as it is preceded by a spiral dive-in motion that moves into the material in the z-direction.

The planning component traverses boundary lines of the selected workpiece shape, starting from the leftmost position. This way, it determines two lines that follow the upper and lower boundaries of the target geometry, which enclose a possibly non-convex polygon. The tool path generation involves the tool-specific arrangement of circles along the lines because the tool's cut width designates the distance of adjacent circles. A similar planning approach could involve a continuous progression along the boundary region with a parameterized over-loaded rotary motion. This formulation, however, would produce scallops on the workpiece boundaries.

As with other trochoidal planning primitives, this planner produces paths that expose constant tool forces while causing a longer overall machining time. The channel works as an enabler for motion primitives that require a machined area, such as zig-zag without full tool engagement.

5.4.5 Spiral Roughing

The spiral tool motion is displayed on the left side of Fig. 5.6 and is a very efficient roughing pattern that removes the material with circular motions. The benefits of this motion plan are the constant tool engagement and cut forces that result from smooth motions. It offers a high material removal rate within the constraints given by the tool parameters. However, it

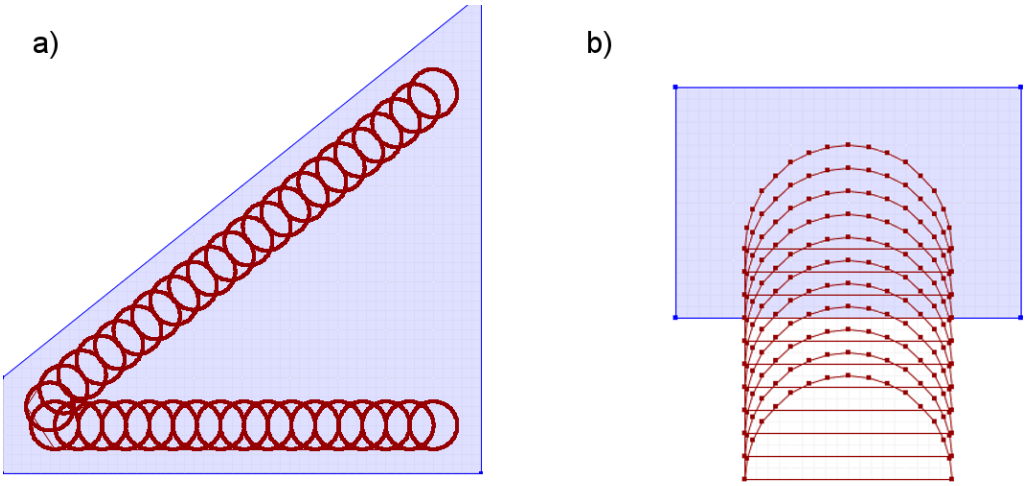


Figure 5.7: (a) Trochoidal channel, (b) Trochoidal slot

has limited applicability to irregular geometries. The curvilinear tool path method proposed by Bieterman and Sandstorm [16] addresses this drawback. It builds a spiral motion that starts from the center region of the target geometry and gradually adapts itself to the outer workpiece geometry.

The planning component to generate regular spiral motions finds the largest inscribed circle for a given geometry and selects the target area accordingly. This way, a starting point and a maximal radius are given. The tool's cut width specifies the step size corresponding to the radius increase per full circle motion. A z-directional dive-in step precedes this operation, and it concludes with a full circle motion with a constant radius.

5.4.6 Dive-in Motions

Machining tasks for closed pockets require motion primitives to enter the material in the z-direction. For this purpose, a rotary tool path primitive implements a linear decrease of the z-coordinate. The tool progresses in a radial trajectory with a fixed radius. The z-velocity corresponds to the allowed depth of cut a_p for the given tool. Similarly, a straight movement with a linear decline of the z-coordinate can accomplish a dive-in motion.

5.5 Repository Structure and Combinators

CLS is used to synthesize machining strategies by composing `PathCoverageStep` objects. The tables 5.5 and 5.6 contain the combinators and their types.

Combinator Name	*Type*
AluminumRoughing	CncTool $\cap$ $roughing \cap aluminum \cap atomicStep$
AluminumFinishing	CncTool $\cap$ $finishing \cap aluminum$
SteelRoughing	CncTool $\cap$ $roughing \cap steel$
SteelFinishing	CncTool $\cap$ $finishing \cap steel$
GenericCompositionPcStep	PathCoverageStep $\cap$ $\alpha \cap pcFct \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcFct \cap atomicStep \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcFct$
RotateModelPcs	PathCoverageStep $\cap$ $\alpha \cap pcFct \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcFct$
ConvexHullPcs	PathCoverageStep $\cap$ $\alpha \cap pcFct \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcFct$

Table 5.5: Components representing tools, model transformations and composition of planning steps

Combinator Name	*Type*
SpiralRoughing	CncTool $\cap$ $\alpha \rightarrow$ PathCoverageStep $\cap$ $\alpha \cap pcFct \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcRoot$
SpecimenContour	CncTool $\cap$ $\alpha \rightarrow$
	PathCoverageStep $\cap$ $pcFct \cap \alpha \cap atomicStep$
SpecimenContourFinishing	(CncTool $\cap$ $\alpha \cap finishing \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcFct \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcFctWithFinishing$)$\cap$
	(CncTool $\cap$ $\alpha \cap finishing \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcFctRoot \rightarrow$
	PathCoverageStep $\cap$ $\alpha \cap pcFctRootWithFinishing$)
ContourRoughing	CncTool $\cap$ $roughing \cap aluminum \rightarrow$
	PathCoverageStep $\cap$ $aluminum \cap pcFct \cap atomicStep$
ContourFinishing	CncTool $\cap$ $finishing \cap aluminum \rightarrow$
	PathCoverageStep $\cap$ $aluminum \cap pcFct \cap atomicStep$
ContourMultiTool	CncTool $\cap$ $roughing \cap aluminum \rightarrow$
	CncTool $\cap$ $finishing \cap aluminum \rightarrow$
	PathCoverageStep $\cap$ $aluminum \cap pcFct \cap atomicStep$
ZigSteel	CncTool $\cap$ $steel \cap roughing \rightarrow$
	PathCoverageStep $\cap$ $steel \cap pcFct \cap atomicStep$
ZigZagAlu	CncTool $\cap$ $aluminum \cap roughing \rightarrow$
	PathCoverageStep $\cap$ $aluminum \cap pcFct \cap atomicStep$
TrochoidalChannel	CncTool $\cap$ $steel \cap roughing \rightarrow$
	PathCoverageStep $\cap$ $steel \cap pcFct \rightarrow$
	PathCoverageStep $\cap$ $steel \cap pcFctRoot$
TrochoidalSlot	CncTool $\cap$ $steel \rightarrow$
	PathCoverageStep $\cap$ $steel \cap pcFct \cap atomicStep$

Table 5.6: Planning components for tool path planning.

The semantic types of the combinators were defined to distinguish between path planning programs that aim to reduce tool wear and programs to find fast machining operations. The type variable α distinguishes between materials and can take the values $steel$ or $aluminum$. Correspondingly, the repository contains the four tool combinators *AluminumRoughing*, *AluminumFinishing*, *SteelRoughing*, and *SteelFinishing*.

Some path planning primitives only apply to specific materials, assuming that tool wear is a higher priority for steel plans while aluminum planning components have a bias towards machining time. For this reason, steel path planning avoids machining operations with full tool engagement, whereas these are allowed in aluminum motion plans. The semantic structure ensures that only suitable tools and motion primitives will be considered to generate machining strategies.

The semantic types $pcFct$, $pcFctRoot$, and $pcFctRootWithFinishing$ allow the user to specify constraints on the root element of the inhabitant tree. For closed pocket machining tasks, the terms must include a dive-in step at the beginning of the machining sequence, described by $pcFctRoot$. $pcFctRootWithFinishing$ indicates that the combinator appends a finishing step at the end of the machining sequence by building a corresponding `PathCoverageStep` object, which holds the finishing path planning step as its last child element.

5.5.1 Generic Composition

The combinator `GenericCompositionPcStep` composes two `PathCoverageStep` objects. Its type states that it can yield an object typed `PathCoverageStep` $\cap\, \alpha$ when inhabitation can determine two arguments that comply with the specified type `PathCoverageStep` $\cap\, pcFct \cap \alpha$.

Moreover, the second argument requires terms that have the type $atomicStep$, which denotes a single planning component. It reduces redundant representations of a particular machining sequence by different inhabitant trees, significantly narrowing the search space for the brute-force procedure.

This combinator causes the inhabitation result to be infinite since a nested composition of arbitrary depth is possible. The implementation of the combinator builds a `PathCoverageStep` object with a children list holding the arguments. This path coverage step does not contain tool data or a path coverage function but expects the children to supply this information. Thus, an evaluation procedure must analyze this data and execute the embedded path coverage steps in the correct order.

5.5.2 Convex Hulls

A convex hull operation can lead to better handling of material isles, such as depicted in Fig. 5.20a. It uses model transformations to block partitions of the model that are inside the convex hull of isles. That way, the following planning components will consider these isles as convex, leading to better applicability of planning primitives.

Usually, a planning component transforms the model based on the generated tool path. The corresponding transformation can be determined by applying a geometric buffer to the tool path and adding the resulting polygon to the machined area of the new model. This convex hull component, however, manipulates the model without tool interaction. The changes are revoked later in the program to ensure that the convex hull operation does not influence the model directly.

The corresponding `PathCoverageStep` data contains three children, and the combinator interposes its argument between two model transformations. It applies a model transformation, executes the arbitrary planning component supplied by the argument, and retransforms the resulting model.

In order to prevent subsequent planning components from machining the blocked area, the geometry is considered part of the specimen and removed from the remaining area. When the argument planning strategy completes, a model retransformation releases the blocked geometry. That way, embedded planning strategy respects the convex hull, and from an outside perspective, there is no manipulation without tool interaction. Thus, after evaluation of the complete model, the resulting model contains the original non-convex isles. The `Cnc2dModel` data structure keeps track of changes, and a stack of model transformations holds the geometric information required to compute the corresponding retransformation. Based on this information, several fields of the model data are updated.

5.5.3 Model Rotation

The combinator for model rotations is another example of a model transformation without tool path generation. It further boosts the applicability of path coverage functions with heuristics based on a fixed direction, eliminating the need to implement these components with variable orientation. Instead, this combinator composes variations of planning steps that include rotated models.

More precisely, it applies a rotation by 90° to the model and then performs the embedded planning sequence provided by the synthesis algorithm. After evaluating this sequence, the

model and the accumulated results are retransformed to the initial geometries by applying a $-90°$ rotation. The nesting of this combinator can also yield 180° and 270° rotations.

This operation uses an affine transformation matrix that alternates every geometry of the model. The transformation applies to the model's boundaries, machined areas, left-over material, target geometry, target workpiece, and the accumulated paths. The type specification of this combinator is similar to the type of convex hulls.

5.6 Numerical Evaluation of Tool Paths

Coverage path problems such as the generation of tool paths are generally solved based on heuristics. The corresponding components incorporate planning patterns that might involve abrupt changes to the tool progression. As such, they often consist of a series of connected path segments. However, these linear segments are not inherently differentiable, and the machining center resolves the machining process by applying directed forces to the tool, incorporating control routines. Consequently, every directional change might require a deceleration and acceleration of the tool center, which impacts the overall machining time and the compliance with tool parameters.

In general, a tool path can prescribe a set of control instructions for a machine or robot. The planning results are sequences of three-dimensional points, which can be translated into machine instructions to guide the tool. A machining center can handle a sequence of coordinates to position the tool center with a given target feed.

For this experiment, the generation of machining instructions will consider feed rates that comply with the computed path and an acceleration model to describe machine dynamics. The incorporation of machine dynamics yields paths that are likely to match the desired outcome.

5.6.1 Acceleration Model for Machine Dynamics

Using tool feeds, an estimate for the machining time is possible, which allows a more precise evaluation of machining sequences. This proof-of-concept uses a numerical acceleration model that results from the work of Freiburg et al. [34]. They use a physically-based geometric simulation system to find differential equations for the tool velocity vector and evaluate this representation of machine dynamics with experiments.

After executing the planning sequence based on components, a post-processing step uses this particular model to compute feed rates along the resulting path. For this purpose, the target feed v_f is determined for every path segment considering the machining centers' dynamic constraints. It describes the velocity of the tool's motion towards the target node and is part of the machine instructions.

The model approximates the acceleration as a function of the feed rate. A machining center exposes a different acceleration behavior for x, y, and z-direction, and Equations 5.1 - 5.3 show the corresponding functions. Therefore, a velocity vector is set up for every path segment (i.e., pair of path nodes) to determine admissible x- and y-velocities.

$$a_x(v_f) = \begin{cases} -0,0624 \cdot v_f^2 \\ +0.2603 \cdot v_f \\ +0.0216, \\ -0.0027 \cdot v_f^2 \\ +0.0880 \cdot v_f \\ +0.1536, \end{cases} \quad \begin{array}{l} \\ \text{if } 0 < v_f < 1 \\ \\ \\ \text{otherwise} \\ \\ \end{array} \tag{5.1}$$

$$a_y(v_f) = \begin{cases} -0.1979 \cdot v_f^2 \\ +0.3970 \cdot v_f \\ +0.0184, \\ -0.0036 \cdot v_f^2 \\ +0.0967 \cdot v_f \\ +0.1427, \end{cases} \quad \begin{array}{l} \\ \text{if } 0 < v_f < 1 \\ \\ \\ \text{otherwise} \\ \\ \end{array} \tag{5.2}$$

$$a_z(v_f) = \begin{cases} -0.1429 \cdot v_f^2 \\ +0.3411 \cdot v_f \\ +0.0263, \\ -0.0041 \cdot v_f^2 \\ +0.1090 \cdot v_f \\ +0.1280, \end{cases} \quad \begin{array}{l} \\ \text{if } 0 < v_f < 1 \\ \\ \\ \text{otherwise} \\ \\ \end{array} \tag{5.3}$$

$$v_{f,max}(\omega) = \begin{cases} -14.8311 \cdot 1/\omega^2 \\ +28.4664 \cdot 1/\omega \\ -0.1496, \end{cases} \quad \begin{array}{l} \\ \omega > 1^\circ \omega < 180^\circ \\ \\ \end{array} \tag{5.4}$$

5.6.2 Construction of Lookup Tables

The computation of the feed rates involves a trade-off between accuracy and run-time. The functions in Equations 5.1 - 5.3 were transformed into different lookup tables by sampling the values with a configurable time increment (0.01 s) to avoid redundant calculations.

Let v denote the feed rate, t the time progression, and s the traveled distance for an accelerated motion. The progression of the velocities and the corresponding positions are determined by $s_{i+1} = s_i + v_{f,i} \cdot \Delta t$ and $v_{i+1} = v_i + a(v_i) \cdot \Delta t$. A table for a given starting feed rate v_0 is defined for $v_0 < v < v_{f,tool}$, and it represents an accelerating motion.

A sequence of tuples (v, t, s) is built incrementally by solving the differential equations and using a small Δt to determine the updated velocity. A lookup table for a given feed rate v_0 takes its entries from an initial table that contains acceleration motions starting at $0.0 mm/min$. For every tuple of the new table, offsets adjust the values of t and s according to v_0. That way, a series of points can be retrieved for a given starting velocity, yielding a precomputed, discrete representation of accelerating and decelerating motions.

Another lookup table represents the function in Equation 5.4 which describes the maximal admissible feed rate for a given directional change of the tool motion. The table's initialization happens at the beginning of a brute-force search, and it allows to find values with a angle resolution of 1°. The determination of feed rates includes transforming angles to be in (0, 180) and performing a query on the lookup table.

5.6.3 Calculation of Feed Rates

Listing 1 shows the main function for the calculation of feed rates for a given path and a tool. The path representation is a sequence of points, each represented by a sequence of numeric values. For the list named $path$, $path_i$ denotes the element at position i of the list and $path.size$ describes the number of contained elements.

A first step (lines 6 - 8) determines the maximal feasible feed rate for the angles between consecutive path segments and concatenates this value to the connecting node. For this purpose, the function CALCULATEFEEDFORANGLES in Listing 2 constructs vectors representing the enclosing path segments, calculates the angle, and performs a lookup for the angle using a table representing Equation 5.4.

Acceleration and deceleration phases are represented by introducing new segments for every part of the path. The given acceleration model can be used to determine the possible acceleration and deceleration behavior, and both have to be determined to find an accurate

```
1: function CalculatePathFeeds(path, tool)
2:     v_max ← tool.v_f
3:     A ← empty list
4:     D ← empty list
5:     P ← empty list
6:     for i in 1...path.size − 2 do
7:         v_α ← CalculateFeedsForAngles(path_{i−1}, path_i, path_{i+1})
8:         path_i ← path_i + v_α
9:     for i in 0...path.size − 2 do
10:        a ← BuildAccSeq(path_i, path_{i+1})
11:        path_{i+1}.v ← Min(path_{i+1}.v, a_last.v)
12:        A ← A + a
13:    reversedPath ← ReverseList(path)
14:    for i in 0...reversedPath.size − 2 do
15:        d ← BuildAccSeq(reversedPath_i, reversedPath_{i+1})
16:        reversedPath_{i+1}.v ← Min(reversedPath_{i+1}.v, d_last.v)
17:        D ← D + d
18:    D ← ReverseList(D)
19:    for i in 0...path.size − 2 do
20:        resolvedList ← ResolveAccList(p_i, p_{i+1}, A_i, D_i)
21:        p ← ComputePathSegment(p_i, p_{i+1}, resolvedList)
22:        P ← P + p
23:    return P
```

Listing 1: Calculation of Feed Rates for a Tool Path

```
1: function CalculateFeedsForAngles(p_1, p_2, p_3)
2:     s_1 ← VectorFrom(p_1, p_2)
3:     s_2 ← VectorFrom(p_2, p_3)
4:     α ← AngleBetween(s_1, s_2)
5:     v_f ← LookupFeed(α)
6:     return v_f
7: function BuildAccSequence(p_1, p_2, v_0, v_max)
8:     A_acc ← getLookupTable(v_0, v_max)
9:     s_0 ← A_{acc,0}.s
10:    distance_{p1,p2} ← Distance(p_1, p_2)
11:    for a in A_acc do
12:        a.s ← a.s − s_0
13:    A_acc ← A_acc.filter(a.s < distance_{p1,p2})
14:    return A_acc
```

Listing 2: Feed-Rate Lookup for Angles and Construction of Acceleration Sequences

```
1: function ResolveAccList(p1, p2, A, D)
2:     if distance(p1, p2) > A.last.s + D.last.s then
3:         return p1 + A + D + p2
4:     else
5:         return ResolveAccListOverlap(p1, p2, A, D)
6: function ResolveAccListOverlap(p1, p2, A, D)
7:     Au ← empty list
8:     Du ← empty list
9:     for a in A do
10:        if ∃d ∈ D : d.s < a.s ∧ d.v > a.v then
11:            Au ← Au + a
12:        else
13:            break
14:    for d in D do
15:        if d.s > Au.last.s then
16:            Du ← Du + d
17:        else
18:            break
19:    return p1 + Au + Du + p2
```

Listing 3: Resolving Acceleration Motions

tool trajectory. Thus, the traversal of the path in lines 9-18 of Listing 1 constructs acceleration and deceleration sequences for every path segment. The corresponding function BuildAccSequence in Listing 2 extracts subsets of a precomputed lookup table.

The procedure builds these acceleration sequences for the x- and y-direction. However, this is omitted in the pseudo-code for clarity. Given these sequences, comparing the dimension's time value of the last contained tuples enables the identification of the limiting dimension. The projection of the corresponding sequence onto the path segment vector yields the resulting acceleration sequence with updated distances and velocities.

The given model considers a deceleration as a negative acceleration. Therefore, the traversal of the reversed path will find the admissible feeds limited by deceleration for the path nodes (lines 13 - 17 in Listing 1).

A final step resolves possibly overlapping acceleration and deceleration sequences for each path segment. The function ResolveAccListOverlap in line 6 of Listing 3 selects elements from the acceleration sequence until a deceleration is necessary, and the selection involves the comparison of the respective feed rates along the path. The subsequence of the deceleration table D_u is built in conjunction with the last element of A_u (line 15). The resulting path segment consists of the starting point, the selected acceleration and deceleration motions, and the endpoint (line 19). Thus, this procedure also works for path segments without any

acceleration or deceleration because both sequences can be empty. The procedure thus aims to select the maximal feed rate while remaining in the feasible regions of the acceleration model. The resulting entries for the description of dynamics are mapped to the path segments, consisting of a list of points with feed rates.

5.7 Experiments and Results

This section presents several experiments and selected solution candidates produced by the proposed methodology.

The synthesized programs are terms that are composed of components, and the Appendix C contains the **inhabitant trees** for these planning programs. The trees are bipartite graphs with directed edges and have been generated by the CLS IDE [12]. One vertex set represents type specifications with yellow boxes, and the opposite one contains combinators symbolized by blue circles. A combinator with a directed edge to a type node means that this combinator requires an argument with the specified type. The labeling of these edges indicates the position of the argument. A directed edge from a type node to a combinator node means that the term represented by the appending subtree covers the corresponding argument position.

Upon evaluation, the `PathCoverageStep` **tree** is built and evaluated by using the data structure `PathCoverageResult`. The representation of `PathCoverageStep` trees allows for a more comprehensible interpretation of the resulting machining strategies. The machining operations, i.e., the steps that contain a path coverage function and perform the removal of material, are highlighted with a green background. Furthermore, a node's label states the name of the machining operation and the used tool, where (R) designates a roughing tool and (F) a finishing tool. The directed edges express the parent-child relationship between nodes.

The resulting tool paths are transformation operations on the `Cnc2dModel` object. Therefore, the representation of a machining strategy contains the target geometry, the tool paths, and the accumulated machined area of each machining operation. The target geometry corresponds to the blue polygon, the machined area is a red polygon, and the tool path is a red line within the machined area. A single process step displays only the current tool path, while the machined area also includes the removed geometries from previous steps. This way, the machining operations of a machining strategy can be recognized as modifications of the `Cnc2dModel`.

The **feed rates** are visualized with colorized paths where the colors scale linearly with the feed rate. Green path segments indicate a motion with the tool's maximal feed rate v_f, whereas red path segments show slowed tool movements due to the machining centers' acceleration constraints. For a machining sequence consisting of multiple paths, a single figure contains

all paths. The start and endpoints of each machining operation have a minimal target feed rate of 50.0 mm/min.

5.7.1 Experiment 1: Open Pocket

The first experiment is a small open pocket machining task with an irregular target geometry. The brute-force procedure can find complex machining strategies consisting of different machining steps and is suitable for complex geometries. However, this experiment shows that the procedure works well for simple machining tasks. Open pocket plans are obligated not to include planning components with a dive-in step. Instead, the tool paths must start outside the target geometry and move into the material. The corresponding `Cnc2dModel` includes an initial area that is accessible for the tool. In this case, it is depicted as the rectangular area below the target geometry.

Solution Candidate 1.1

The brute-force search strategy determined the first candidate for an aluminum setup which consists of a zig-zag step (Fig. 5.9a) with the aluminum roughing tool and a contour finishing step (Fig. 5.9b). Fig. C.1 shows the inhabitant tree and Fig. 5.8 displays the associated `PathCoverageStep` tree. The aluminum planning components assume that a machining step with a 180° tool engagement angle is not strictly prohibited. Therefore, the first path segment of the zig-zag step is valid, although it comes with a high tool engagement angle. Due to the geometric properties of the model, this zig-zag step resembles a zig pattern. The feed rates for the solution candidate 1.1 are shown in Fig. 5.14a.

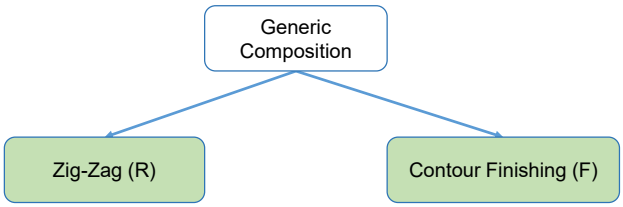
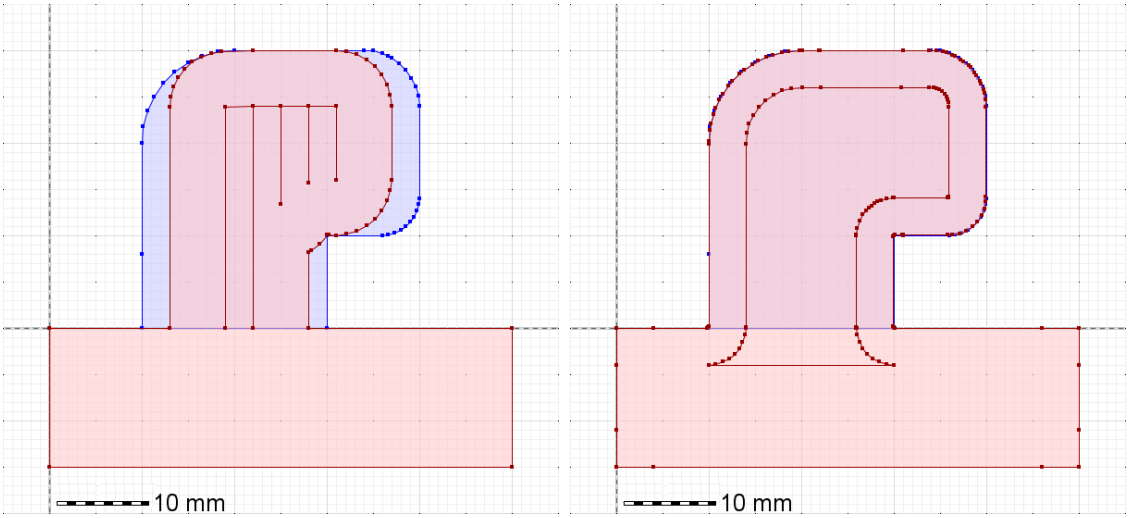


Figure 5.8: PathCoverageStep tree representation of candidate 1.1



(a) Result 1.1a (b) Result 1.1b

Figure 5.9: Generated tool paths and geometric models for candidate 1.1, steps a and b

Solution Candidate 1.2

The inhabitant for the second candidate solution is displayed in Fig C.2, and Fig. 5.10 shows the corresponding data structure. The solution uses a predefined contour planning sequence consisting of a multi-contour roughing step (Fig. 5.11a) and a finishing step (Fig. 5.11b). The roughing operation removes a large amount of material and involves several repositioning steps of the tool. The finishing step is required because of the edge radius of the roughing tool. Fig. 5.14b displays the resulting feed rates.

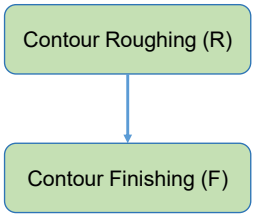
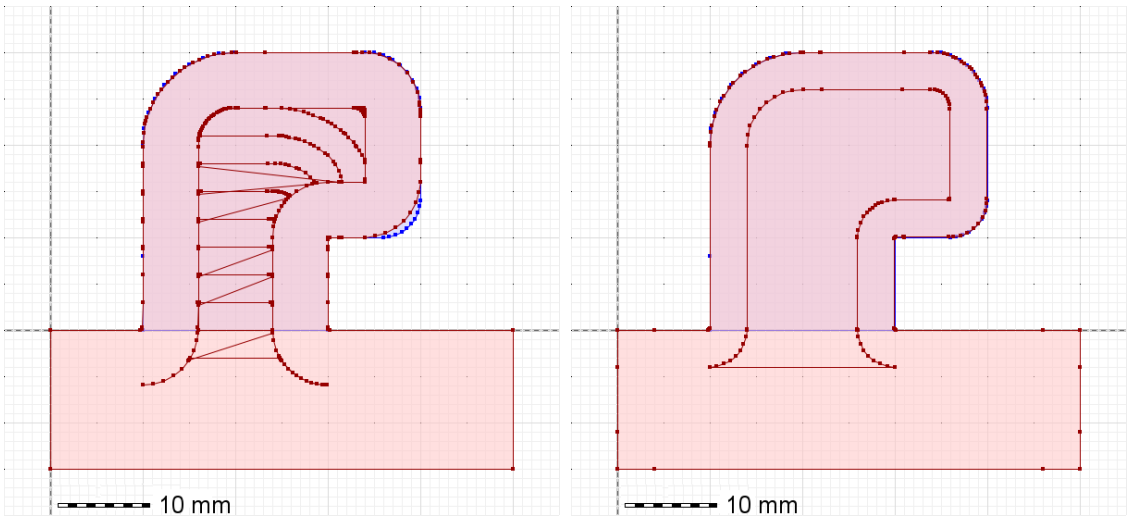


Figure 5.10: PathCoverageStep tree representation of candidate 1.2



(a) Result 1.2a (b) Result 1.2b

Figure 5.11: Generated tool paths and geometric models for candidate 1.2, steps a and b

Solution Candidate 1.3

Candidate 1.3 is a solution for a steel configuration, and it demonstrates the differences in machining strategies for aluminum and steel. The inhabitant is represented by Fig. C.3 and the resulting data object by Fig. 5.12. This planning sequence starts with directed, trochoidal slot milling (Fig. 5.13a), as steel prohibits moving into the material with a maximal tool engagement angle. After that, a 90° rotation transforms the model (Fig. 5.13b). Fig 5.13c shows that the embedded machining step, another directed slot milling step, can remove the remaining material of the rotated model.

The implementation of this machining step only progresses into the material in the positive y-direction. With the rotation combinator, this combinator can be applied to a broader range of geometries. The machining sequence concludes with two contour-parallel steps consisting of a roughing step and a finishing step (Fig. 5.13e, Fig. 5.13f).

The tool feeds are illustrated in Fig. 5.15. Due to the trochoidal tool paths, the feed rates mostly comply with the given tool feed rate. The few motions with decreased feed rate do not occur during tool engagement.

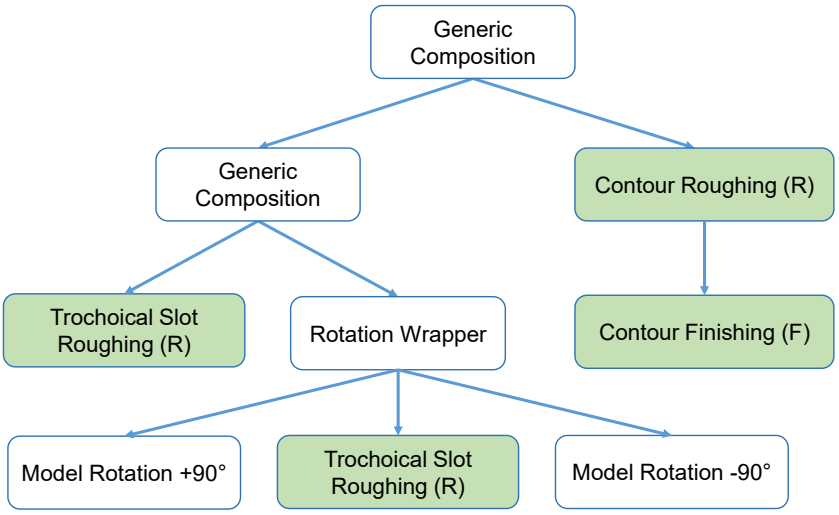
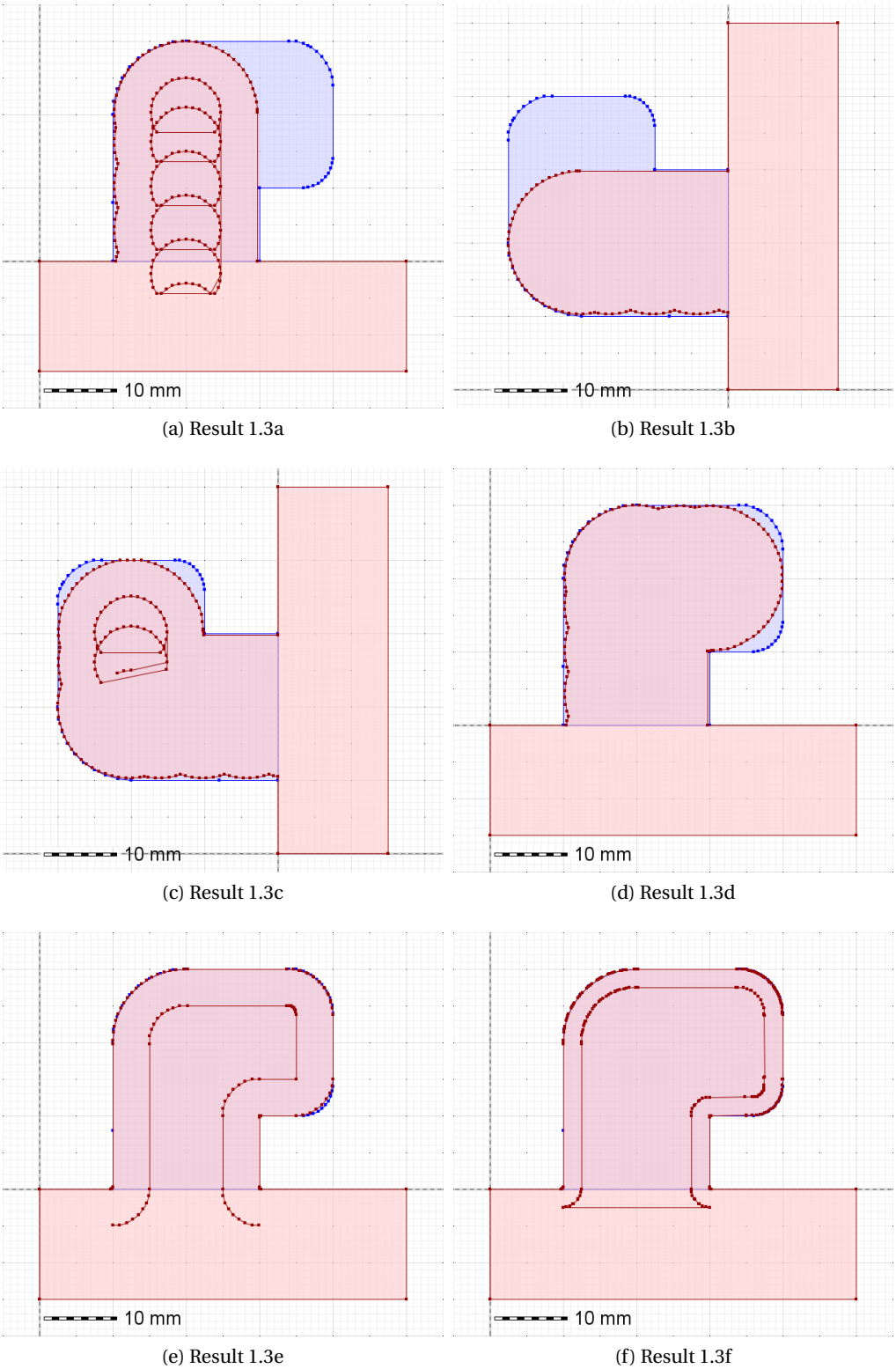


Figure 5.12: PathCoverageStep tree representation of candidate 1.3



(a) Result 1.3a

(b) Result 1.3b

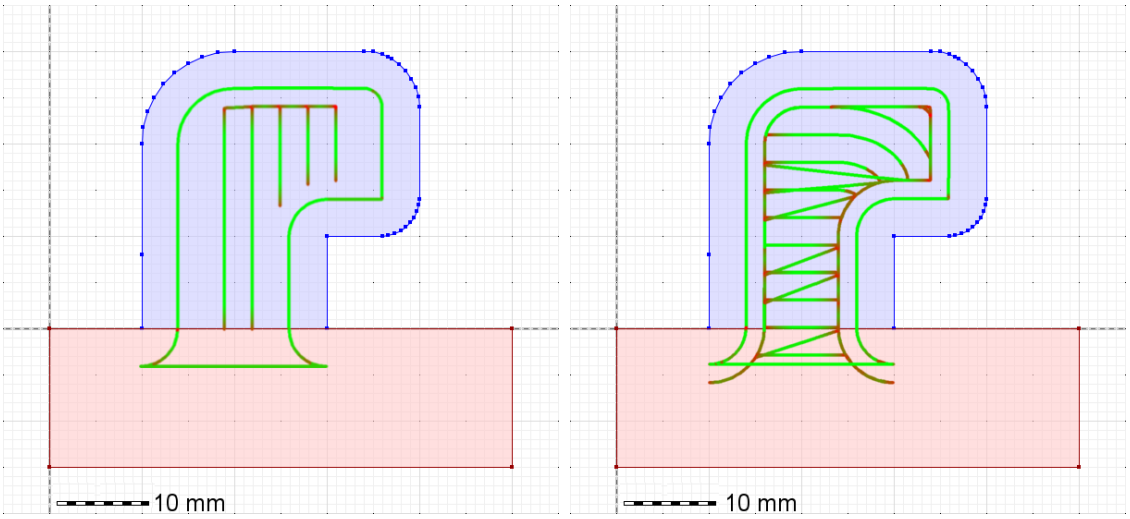
(c) Result 1.3c

(d) Result 1.3d

(e) Result 1.3e

(f) Result 1.3f

Figure 5.13: Generated tool paths and geometric models for candidate 1.3, steps a-f



(a) Feed rates, candidate 1.1 (b) Feed rates, candidate 1.2

Figure 5.14: Feed rate visualization for candidates 1.1 and 1.2

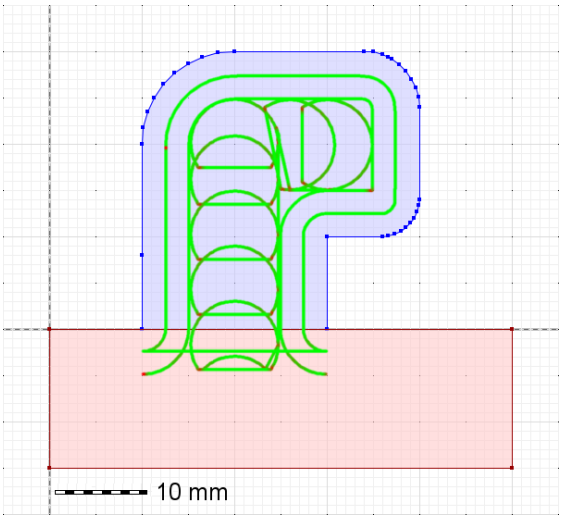


Figure 5.15: Feed rate visualization for candidate 1.3

5.7.2 Experiment 2: Closed Pocket

This aluminum experiment is a nontrivial closed pocket milling task with a large non-convex irregular center area with an adjacent irregular pocket. The target geometry and the machining steps of this candidate are displayed in Fig. 5.17. Fig. 5.18 illustrates the feed rates, Fig. C.4 shows the inhabitant tree, and the data structure is depicted in Fig. 5.16.

The machining strategy starts with a radial dive-in step and a spiral roughing operation displayed in Fig. 5.17b. It is then followed by two zig-zag patterns (Fig. 5.17c, Fig. 5.17d) and a contour-based roughing operation (Fig. 5.17e). The specimen contour finishing step in Fig. 5.17f removes material in the proximity of the specimen, where it also clears material that remained due to the edge radius of the roughing tool. This particular finishing operation follows the contour of the final specimen instead of the remaining material.

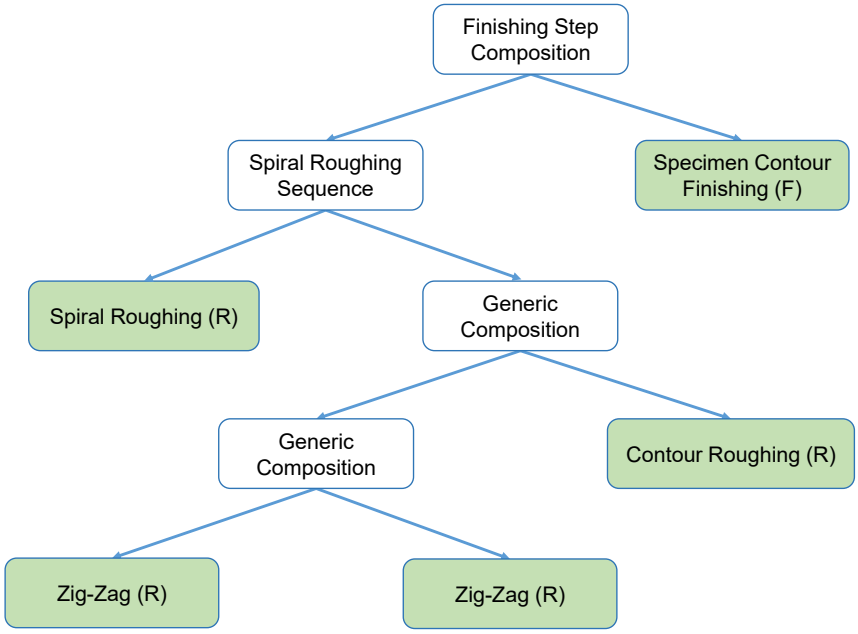
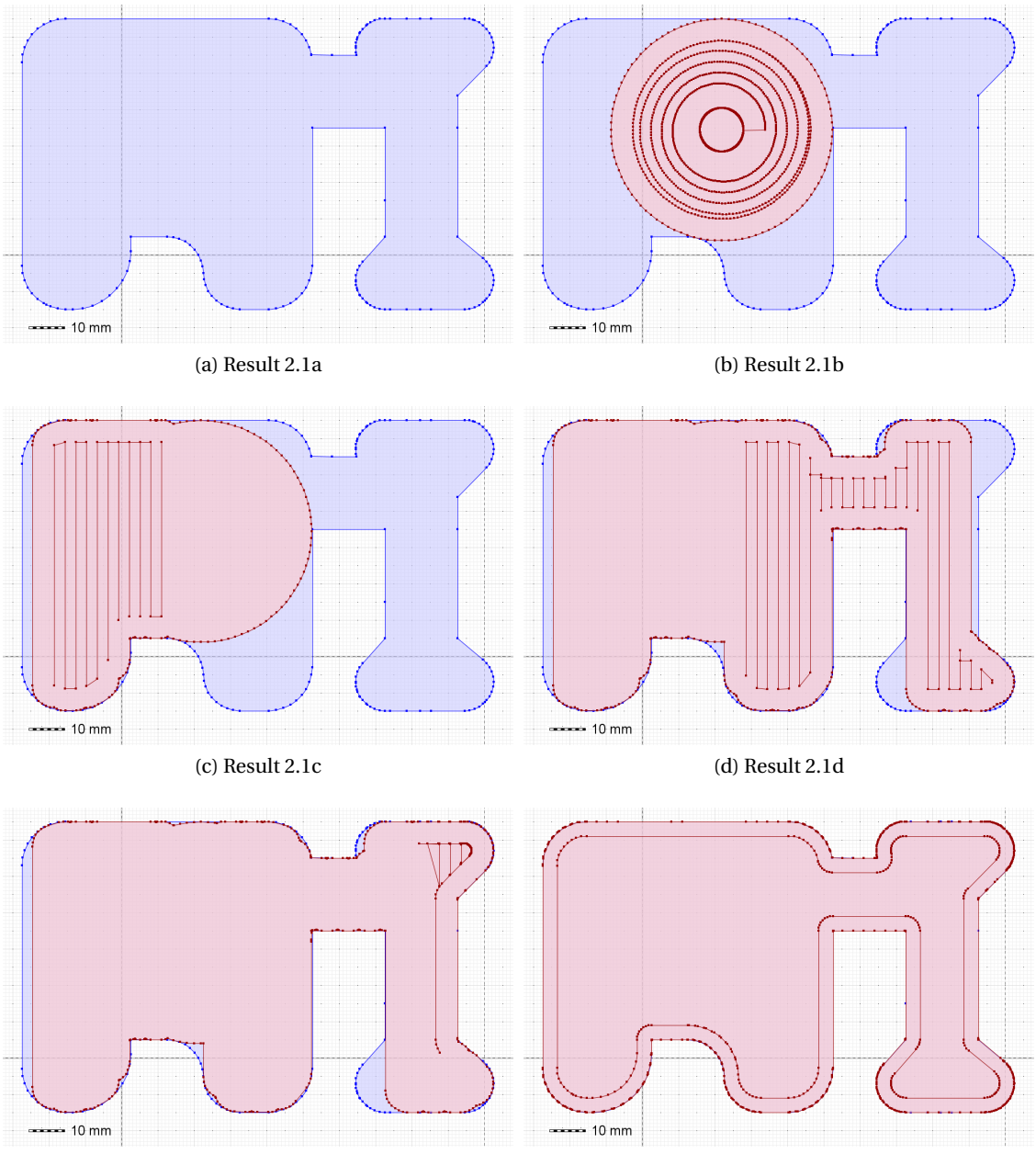


Figure 5.16: PathCoverageStep tree representation of candidate 2.1



(a) Result 2.1a

(b) Result 2.1b

(c) Result 2.1c

(d) Result 2.1d

(e) Result 2.1e

(f) Result 2.1f

Figure 5.17: Generated tool paths and geometric models for candidate 2.1, steps a-f

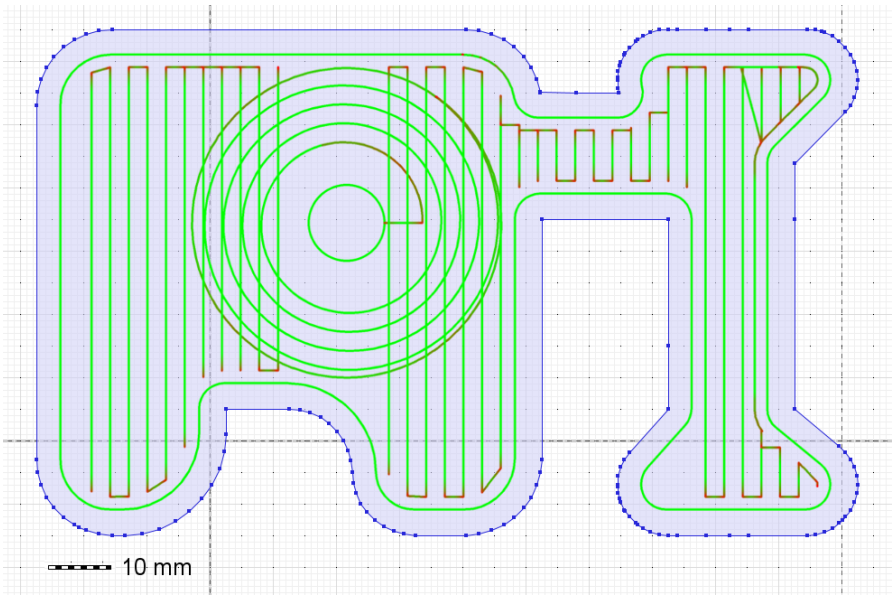


Figure 5.18: Feed rate visualization for candidate 2.1

5.7.3 Experiment 3: Isles

The third experiment incorporates a workpiece that contains two free-form isles (Fig. 5.20a). With the introduction of corresponding combinators, the generated motion planning programs can handle this new class of geometries, which helps to improve the quality of candidate solutions. Moreover, this experiment shows that the brute-force search on synthesized planning programs can manage large-scale strategies.

Solution Candidate 3.1

The machining sequence of solution candidate 3.1 is built from the inhabitant in Fig. C.5 and the corresponding `PathCoverageStep` object in Fig. 5.19. It starts with a zig-zag step (Fig. 5.20b). Due to the non-convex form of the upper isle, the connecting paths of this step resemble repositioning steps. After the first roughing step, a 90° rotation transforms the model, and another zig-zag pattern is applied (Fig. 5.21a, Fig. 5.21b).

The rotation changes the applicability of the following zig-zag planning component for the given geometry. After retransformation in Fig. 5.21c and the machining of the workpiece boundaries (Fig 5.21d), the contour-based planning step in Fig. 5.21e selects and removes the largest remaining area around the lower isle. The machining strategy concludes with a finishing step based on the contour of the specimen and its isles (Fig. 5.21f). The feed rates for this machining strategy are shown in Fig. 5.25a.

Solution Candidate 3.2

The second solution candidate for this experiment starts with a convex hull model transformation that determines and blocks the convex hulls of the two contained isles (Fig. 5.23a). The contour-parallel machining step in Fig. 5.23b targets the upper isle and causes the first tool interaction. After that, the retransformation of the convex hull operation releases the blocked areas (Fig. 5.24a). The next step in Fig. 5.24b performs a contour-parallel tool path that starts from the machined area and removes most of the remaining geometry. Fig. 5.24c and Fig. 5.24d show the conclusion of the machining sequence with contour-parallel finishing steps that guide the tool alongside the isles and the workpiece boundaries. Fig.5.25b displays the feed rates for the resulting tool paths.

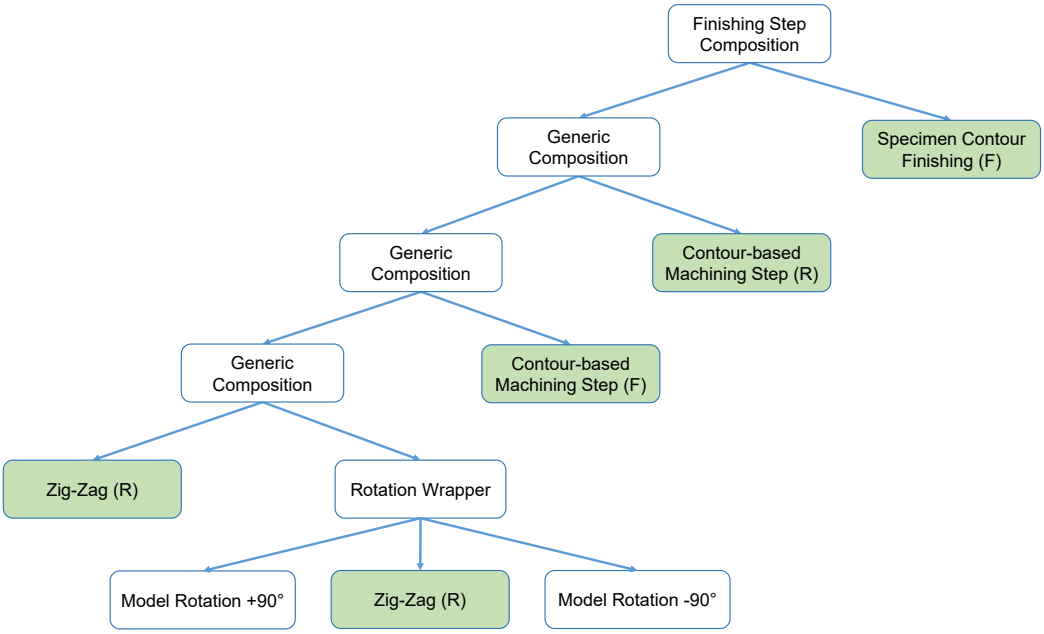
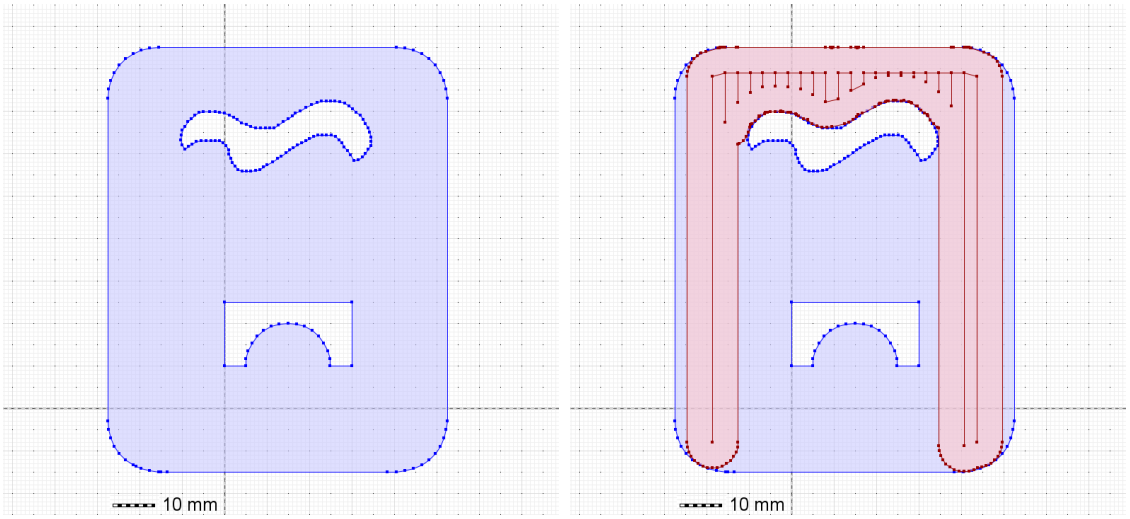
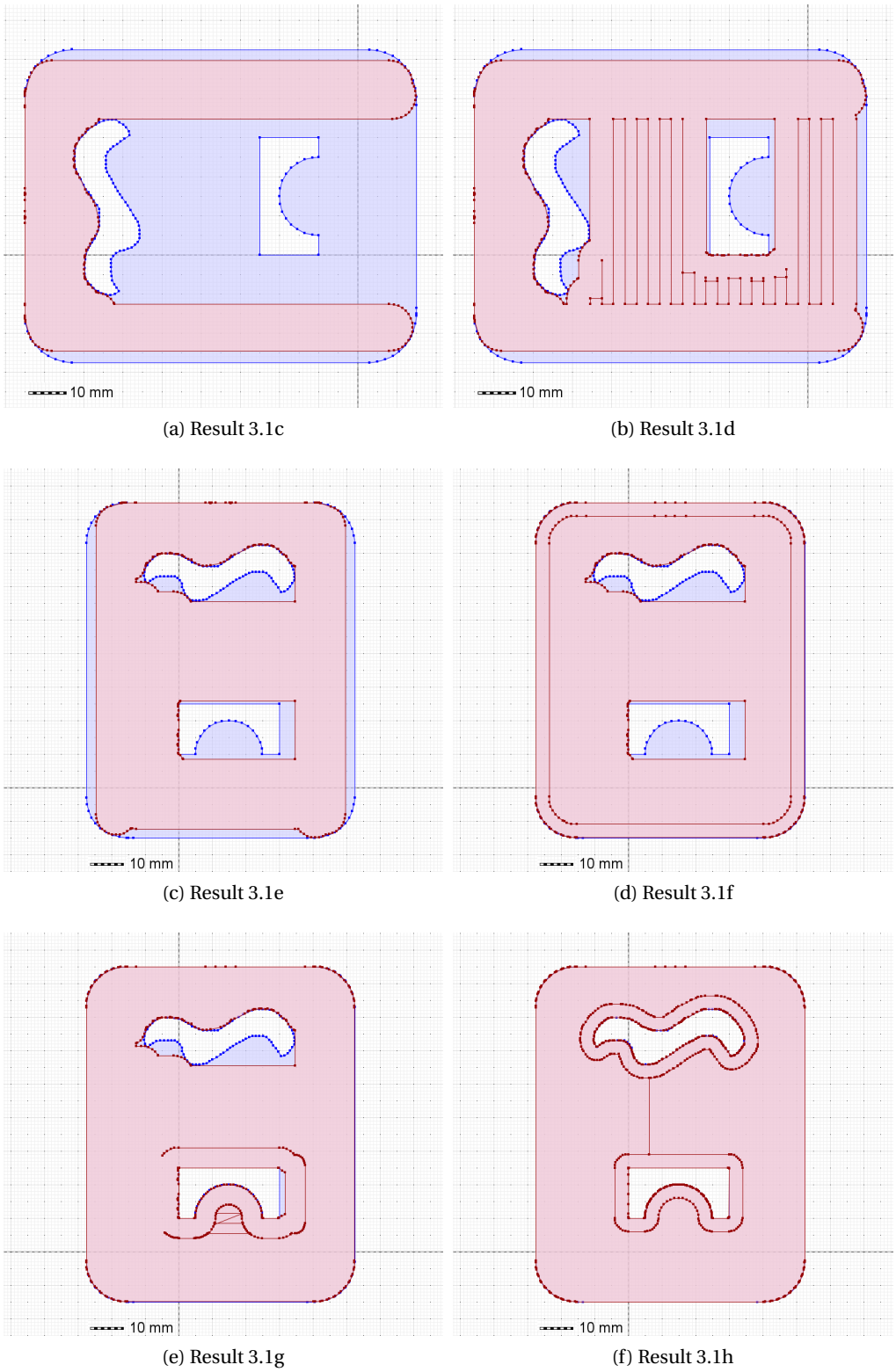


Figure 5.19: PathCoverageStep tree representation of candidate 3.1



(a) Result 3.1a (b) Result 3.1b

Figure 5.20: Generated tool paths and geometric models for candidate 3.1, steps a and b



(a) Result 3.1c

(b) Result 3.1d

(c) Result 3.1e

(d) Result 3.1f

(e) Result 3.1g

(f) Result 3.1h

Figure 5.21: Generated tool paths and geometric models for candidate 3.1, steps c-h

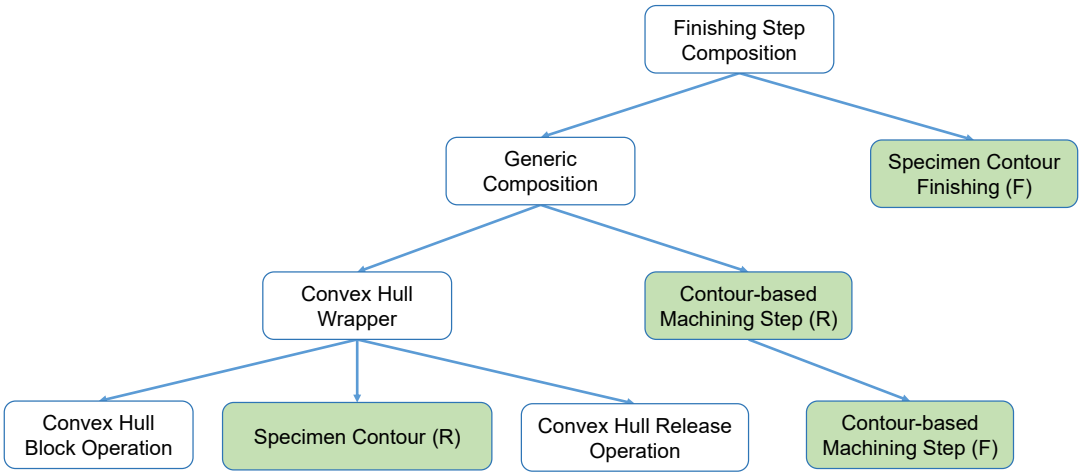
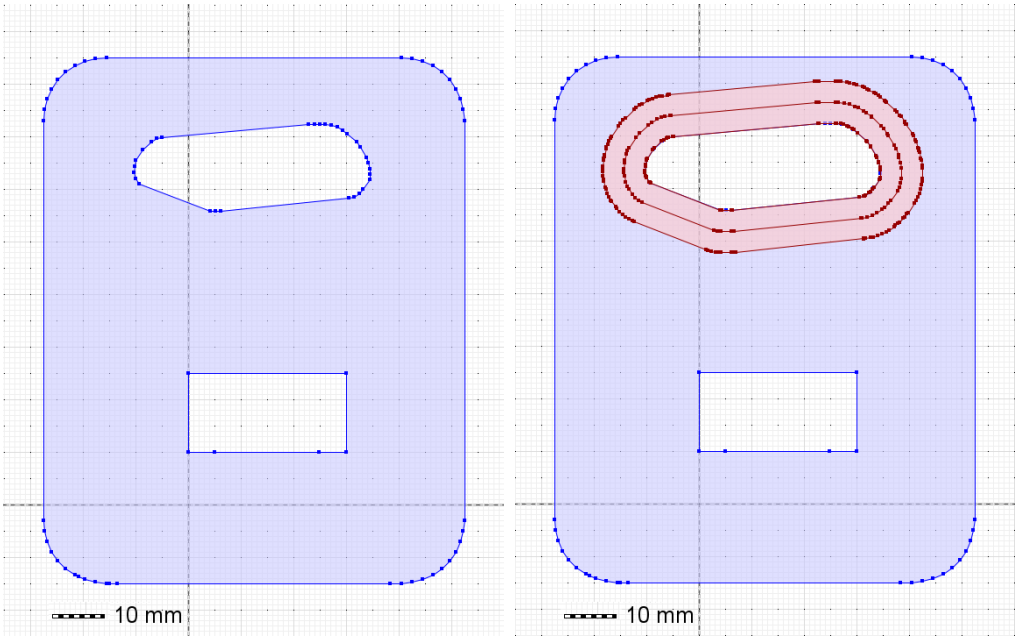
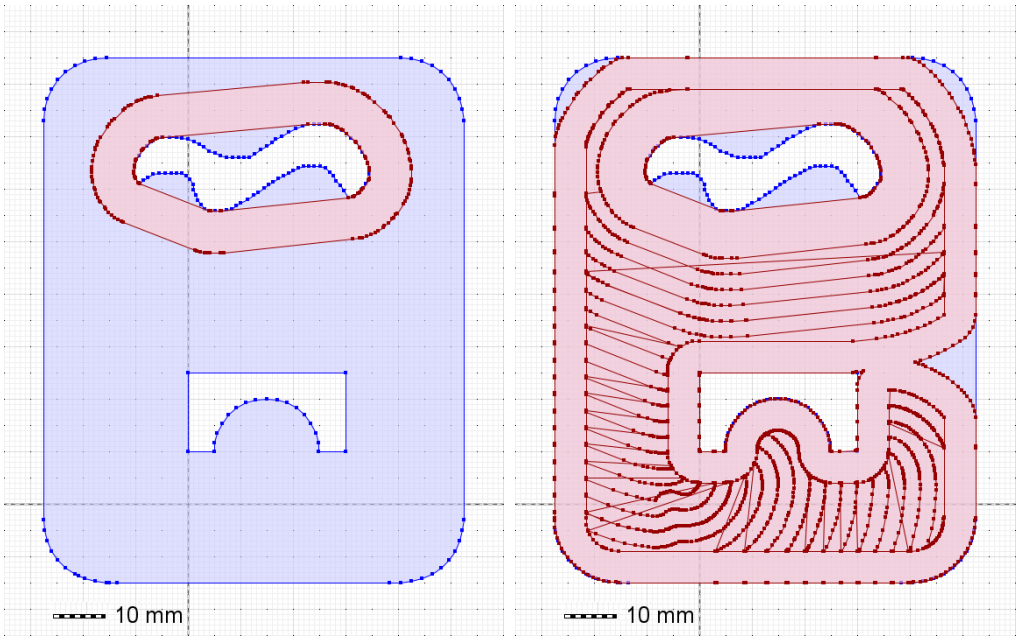


Figure 5.22: PathCoverageStep tree representation of candidate 3.2

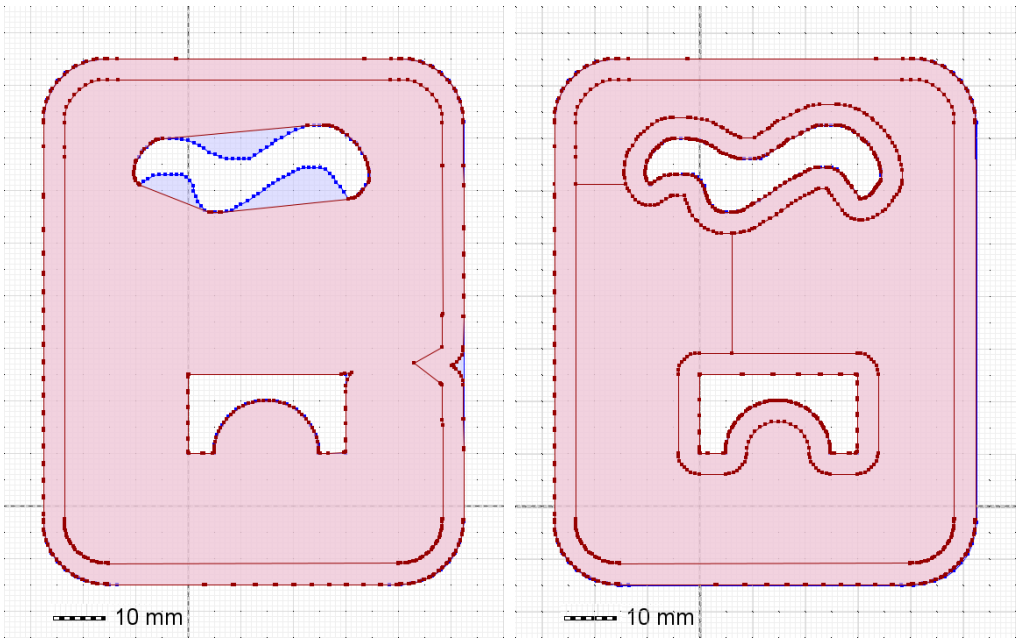


(a) Result 3.2a (b) Result 3.2b

Figure 5.23: Generated tool paths and geometric models for candidate 3.2, steps a and b

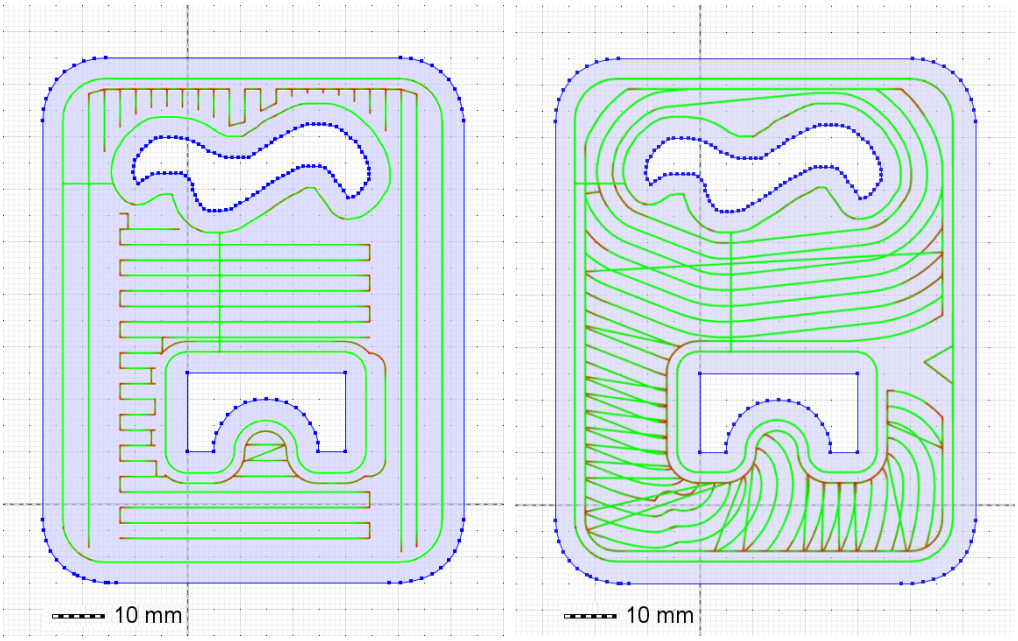


(a) Result 3.2c (b) Result 3.2d



(c) Result 3.2e (d) Result 3.2f

Figure 5.24: Generated tool paths and geometric models for candidate 3.2, steps c-f



(a) Feed rates, candidate 3.1 (b) Feed rates, candidate 3.2

Figure 5.25: Feed rate visualization for candidates 3.1 and 3.2

5.7.4 Generation of Machine Control Instructions

Machining instructions can accomplish the programming of CNC machining centers. In this work, these machining instructions bridge the gap between planning algorithms and practical machining applications. For this purpose, machining strategies consisting of multiple machining operations are translated to instruction sets, using the Heidenhain Klartext format.

The concatenation of different machining operations involves repositioning the tool in the z-direction to avoid tool engagement. Additional path segments with a corresponding predefined z-coordinate connect the different machining steps. However, this principle does not apply to machining operations that use different tools or involve dive-in machining steps. Listing 5.1 demonstrates a Heidenhain Klartext output with instructions containing x, y, z-coordinates, and the targeted feed rate. Tool positioning without material engagement is performed with maximum tool speed FMAX, and motions in z-direction use a predefined feed rate of 500 mm/min.

```
1 BEGIN PGM Single MM
2 TOOL CALL 2 ; Alu finishing, 8mm, cut width 4mm, vf 1430mm/min, n
    11935
3 L X16.0 Y6.0097966 Z20.0 FMAX
4 L X16.0 Y6.0097966 Z0.0 F500.00000074505806
5 L X16.0 Y18.009188 F164.88254070281982
6 L X23.986914 Y18.009998 F1430
7 L X24.0493 Y18.009745 F1430
8 L X24.11168 Y18.008974 F1430
9 (...)
```

Listing 5.1: Example Klartext output

5.8 Discussion

5.8.1 Brute-Force Search Strategy

The idea of this work does not intend to use types for an exact specification of sequences motion planning steps. Rather, types specify a wide range of terms with high structural variability representing different machining sequences. The inhabitation algorithm yields every composable data structure that complies with the machining task configuration. Due to a general composition of machining operations, the result set is infinite.

A corresponding evaluation technique translates these data structures into machining sequences. **CLS enables the brute-force search** because it handles the variability of the search space and limits the results to the set of well-typed terms. The procedure has shown to be viable due to the restrictions imposed by the semantic structure. Combinators encode predefined sequences of machining operations, which allows handling complex 2.5D machining tasks.

The brute-force search creates a variety of unique geometric shapes caused by distinctive updates of target areas and machined areas after specific machining operations. The planning components use geometric operations, which can cause runtime errors if the geometric model is flawed. Consequently, the procedure imposes immense **robustness requirements** on the employed planning components. The identification and extensive logging of runtime errors facilitate a sophisticated test suite.

The **caching of partial tree results** could lead to further improvement of the overall search performance. As trees evaluate to sequences of machining operations, a caching mechanism

could store intermediate results of sequences for reuse in successive iterations of the brute-force search.

A corresponding model-specific data structure could map a suitable representation of machining sequences to the adapted model and the accumulated tool paths. The evaluation procedure must incorporate a lookup for the given planning task and only performs the computation when the cache does not contain a corresponding entry. This way, redundant computations can be avoided, leading to a speed-up of the search.

5.8.2 Holonomic and Non-holonomic Planning

The planning components in this work assume holonomic tool positioning, and the search procedure incorporates the machine dynamics in a post-processing step. A different approach to the planning task is non-holonomic planning, which uses machine dynamics during the tool path planning to retrieve viable path segments. With this approach, planning components could use optimization methods and yield results that strictly comply with the tool parameterization. These methods, however, lead to a significant increase in planning cost, which limits the applicability of the multi-layered filtering techniques and therefore obstructs the scaling to more complex forms.

In the proposed methodology, the planning components encode the principles of tool motion, and a post-processing step only enriches viable solutions with feed rates. This concept enables the synthesis of complex machining strategies while ensuring a close match between planning outcome and actual machining results.

5.8.3 Further Planning Primitives

While this work focuses on the composition of suitable planning primitives, these planning components can also expose variability that CLS can address. One possible example is the incorporation of post-processing to improve tool paths further.

Milling operations in corners tend to show higher tool engagement angles which can cause higher tool wear. **Corner slicing** is one possible post-processing technique that aims to mitigate these effects. This approach slices the corner region into multiple layers and calculates the corresponding operations to subtract the material. It produces longer tool paths with lower tool engagement angles.

The application of this strategy is also a variability point. It can apply to every tool path corner that exceeds a specified angle in the tool path direction. An alternative is a limitation to proximity regions of the workpiece to ensure the final workpiece's shape accuracy.

While this work focuses on synthesizing machining sequences with valid tool assignments based on material-specific planning components, CLS could also generate variations of planning primitives. For instance, the following variability points are possible for zig-zag patterns:

- Admissibility of high tool engagement angles
  This aspect affects the construction of vertical lanes, as described in Section 5.4.1. One policy could strictly avoid tool engagement situations that do not comply with the tool parameterization. However, this method limits the applicability of this planning component. A possible relaxation of this constraint could apply to the end regions of a vertical lane or the advancement into a new lane.

- Connection of straight lines
  There are different variations for the connection of the vertical lanes. A special kind of motion planning primitives can represent these variations and generate tool paths for this particular motion. The corresponding variability point would also incorporate tool repositioning. Consequently, this variability point can express tool patterns that are restricted to conventional or climb milling, and it allows the distinction between zig-zag and zig patterns.

- Selection of partial geometries
  In some cases, the planning component must select a particular geometry among potential areas to process during the planning process. Selection policies could select the geometry closest to the latest tool position, by the highest or lowest y-value, or by the maximal machinable area.

- Post-processing
  This variability point accounts for an optional manipulation of the resulting path. Possible variations are the aforementioned corner slicing strategies or functions for path smoothing. A post-processing function can also expose variability as it may involve the sequential application of several different post-processing techniques.

With the corresponding component design, CLS could regulate the behavior of the synthesized planning component.

Chapter 6

Design Space Exploration for Sampling-Based Motion Planning

The requirements of real-life applications and the high-dimensional configuration space of robotic systems form the need for configurable motion planning solutions. There is a tendency to adapt planning algorithms with application-specific constraints or optimization objectives. While the configuration of algorithms and management of heterogeneous components increases in importance, this work also presents a systematic examination for the resulting space of high variability, demonstrated for the algorithmic family of sampling-based motion planning algorithms.

A feature vector represents points in the feature space of sampling-based motion planning programs. Combinatory Logic Synthesis resolves these points by synthesizing programs that comply with the given algorithm configuration. While CLS can represent spaces with high variability, the user intent is often not expressible as a precise algorithm specification. Similar to the planning task in Chapter 5, a user aims to solve a problem without knowing which configuration yields the best results. This inspired the development of a systematic, CLS-based **design space exploration** methodology using machine learning.

Fig. 6.1 shows the corresponding optimization procedure, which analyzes a specific problem instance with a machine-learning technique that is capable of black-box optimization and supports categorical input parameters. The Hypermapper optimization tool [71] guides the design space exploration for single query planning tasks, using CLS to synthesize runnable programs that comply with selected algorithmic configurations.

Therefore, the experimental setup involves the definition of optimization objectives and the definition of the design space. The matching problem-specific synthesis and the evaluation of programs are an essential part of the black-box function used by the optimization tool.

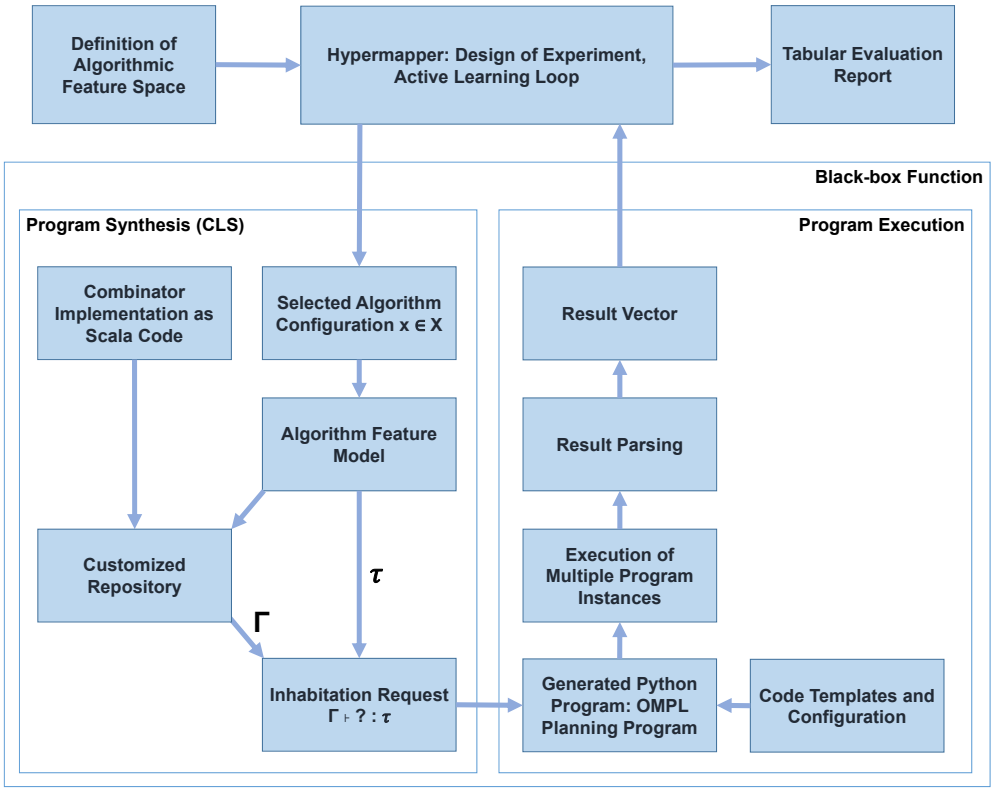


Figure 6.1: Architectural Overview for the Machine Learning Procedure

The optimization procedure aims to find the Pareto-front, and the overall results of an optimization run can also help to analyze algorithmic variations' strengths for a given problem instance.

A repository of components was established based on a selection of variability points that this particular algorithmic family exposes. This CLS repository is the foundation for exploring the space of planning programs, and its automated configuration involves a domain-specific dynamic construction of adjusted repositories to handle the complex search space. Comparable to the approach in Chapter 4, this work also includes generating programs that manipulate and execute code of Python scripts, and analyze the outcome. The sampling-based motion planning programs in Python make use of the Open Motion Planning Library (OMPL, [91]), and the repository is a set of combinators representing planners, samplers, collision detection procedures, and additional code required to set up executable planning programs.

6.1 Motion Planning as a Multi-Objective Optimization Problem

The **multi-objective optimization problem** (MOP) describes a class of optimization problems with multiple objective functions. These are often conflicting objectives in real-life applications, and solving a MOP problem involves finding solutions for these criteria simultaneously. For a multi-objective optimization problem, it is assumed that there are n criteria that can be expressed by a numeric value. Let f_i be an objective function mapping the design space $\mathbb{X}$ to one individual objective i. Following the notion in [23], the optimization function F with m objectives maps a point $x \in \mathbb{X}$ to $\mathbb{R}^m$:

$$F(x) = (f_1(x), \ldots, f_m(x)) \tag{6.1}$$

The MOP is finding $x \in \mathbb{X}$ to minimize this function. However, due to conflicting criteria, it is unlikely to find only one *ideal* x. Instead, a MOP can be considered the search for **Pareto optimal** points. A specific point is called Pareto optimal if it is not *dominated* by any other point. For minimization problems, a vector $u = F(x) = (f_1(x), \ldots, f_m(x))$ is said to **Pareto-dominate** the vector $v = F(x') = (f_1(x'), \ldots, f_m(x'))$, if and only if:

$$\begin{aligned} &\forall i \in \{1, \ldots, m\}, u_i \leq v_i \text{ and} \\ &\exists j \in \{1, \ldots, m\} : u_j < v_j. \end{aligned} \tag{6.2}$$

The set of Pareto optimal points in $\mathbb{X}$ is called **Pareto set** and F applied to these points yields the **Pareto front** in $\mathbb{R}^m$.

The examination of sampling-based motion planning assumes the following dimensions for the design space $\mathbb{X}$:

- Planner
- Sampler
- Motion validator
- Maximal allowed planning time

The optimization aims to minimize the following objectives:

- Averaged solution path length
- Averaged computation time
- Number of failed computation attempts

Categorical variables for the configuration of a planner, sampler, and motion validator represent feature dimensions. The admissible planning time, on the other hand, is a numerical variable. It finds use in the termination condition that will cause an optimizing planner to stop the execution after the given amount of time. This variable enables the comparison of optimizing planners and geometric planners that terminate instantly upon finding a solution.

This particular formulation of the optimization problem does not allow the incorporation of analytical methods to solve the problem. There is no information to compute derivatives for dimensions that directly represent the variability points. Here, the points of these dimensions are the variations of the corresponding variability point. For this reason, this particular optimization problem requires black-box optimization, also called derivative-free optimization or model-free optimization, and the algorithmic feature space of motion planning algorithms builds upon different possible features for predefined variability points. The design space definition uses a map from a set of categorical input parameters to the corresponding feature of the respective dimension.

For this study, HyperMapper guides the exploration of the search space. The framework determines the Pareto front for a given multi-objective function using multiple random forests to learn a model abstracting the MOP function. Based on this model, the active learning loop selects a point in the design space. The black-box function evaluates the point, and the optimization framework updates the internal models according to the results.

An optimization run starts with a Design of Experiment (DoE) phase. There is no significant model available during this phase, so this procedure must collect data unbiased. Possible search heuristics include random search, grid search, Standard Latin Hypercube samples, or K Latin Hypercube samples [92].

6.2 Experimental Setup

An optimization run solves the multi-optimization problem for one planning problem instance at a time. The planning problems include 3D models for the environment, the robot model,

and the motion planning task defined by start and goal state. The experiments employ the problem instance examples of the OMPL.

The motion planning instance represents holonomic rigid body planning in the special Euclidean group $SE(3)$, consisting of a $\mathbb{R}^3$ component and a $SO(3)$ part for robot orientation. Selected planners include geometric planners, planners suitable for multi-query and single-query usage, and optimizing planners.

The experiments consist of a DoE phase with 50 randomly sampled algorithm configurations and 50 active learning iterations. The input value for the maximal computation time was defined to be in the range of 2.0 to 90.0 seconds, which matches the expected time for given problem instances. A failure to find an exact solution after a given time is considered an unsuccessful program execution, and the corresponding result vector is empty. Moreover, the solution paths of the sampling-based motion planning programs are subjected to a post-processing step using the OMPL integrated simplification procedures. The simplification has a maximal allowed time of 2.0 seconds.

6.2.1 Communication Protocol

The black box function has been implemented in Python. It takes a point in the configuration space as a parameter and produces a result vector. The communication between Python and Scala uses the MQTT protocol. For this reason, a corresponding Scala MQTT endpoint designed for this experiment ensures robust communication with the Python script. A broker connects the Scala endpoint and the Python-based client. The Python function can initiate the synthesis and execution of the generated Scala program and receive the result vector. The following communication procedure ensures the robust evaluation of a given point in the design space for this study.

The evaluation of a given algorithm configuration involves several communication steps for this experiment. Every call of the black-box function involves generating a unique identifier to distinguish messages by setting up separate communication channels. First of all, the optimization script checks if the Scala inhabitation service is accessible over MQTT. The algorithm configuration is then built according to the selected point in the design space, encoded as a JSON string, and sent to the Scala listener. This call causes the initialization with the algorithm configuration and the unique ID. It uses a handshake procedure, ensuring that both actors agree on communication channels. The step also triggers the dynamic repository construction and starts the program synthesis. If CLS cannot find an inhabitant, e.g., when the given planner and the sampling strategy do not match, an error response will cause the black-box function to signal the learning loop that the supplied configuration was invalid.

```
{
    "planner" : "sbmp_planner_RRTConnect",
    "sampler" : "sbmp_max_clearance_valid_state_sampler",
    "stateValidator" : "sbmp_fcl_validator",
    "motionValidator" : "sbmp_discrete_motion_validator",
    "costs" : "not_specified",
    "optObjective" : "not_specified",
    "simplification" : "sbmp_no_simplification",
    "sceneInput" : "scene_input_data_file",
    "dimensionality" : "dimensionality_three_d_t",
    "id" : "fff0681f-442c-4402-bb0f-21db7b833389",
    "configurableAlg" : true,
    "withStates" : false
}
```

Listing 4: JSON example

Listing 4 shows a JSON string for an algorithm configuration that uses an *RRT Connect* planner, maximum clearance sampling and discrete motion validation. For the machine learning experiment, the experimental design predefines the additional fields. The protocol provides the maximum allowed computation time in a separate message as it is supplied as an argument to the synthesized program.

Upon successful inhabitation, the following communication steps are required to find the result vector:

- Supply of arguments for program execution
- Start request for program execution
- Transmission of the result vector

The communication makes use of three separate MQTT topics, which contain the UUID in their name. For all steps, timeout value and retry mechanisms ensure to handle communication problems and runtime errors.

6.2.2 Open Motion Planning Library

The Open Motion Planning Library (OMPL) is an open-source library for sampling-based motion planners. It contains several planner implementations, which cover geometric planners,

control-based planners, and optimizing planners. OMPL provides data structures to build planning programs from a predefined set of planners and sampling strategies or develop new planners. The library is written in C++, and integration in Python is possible by using the corresponding Python bindings.

OMPL is a state-of-the-art motion planning library, and it is best known to play an essential role in the MoveIt![1] motion planning framework, which is part of the Robot Operating System (ROS)[2]. MoveIt! incorporates different planning approaches and handles the practical aspects of motion planning such as manipulation, 3D perception, robot control, kinematics, and navigation.

6.2.3 Generation of Python Scripts

The synthesized planning program generates Python scripts from template files. The templating mechanism of the synthesized Scala program substitutes designated code fragments based on the substitution map held by the combinators. The current planning task entails substitutions for the definition of the planning space ($SE(3)$), the boundaries of the planning dimensions, goal state, and target states. Moreover, the Python program loads the environment and the robot models from files based on the Scala program's arguments. Predefined methods load the model data from files and provide access to the mesh data with global variables.

The combinators make sure that all occurrences in the template are substituted and produce runnable Python code. For that reason, they map predefined placeholders to substituting values and encode the associated template and output files. The main script contains the problem setup code, the planning dimensions, sampling strategy, and the definition of the motion- and state validators that use the collision detection libraries. The Scala program saves the generated files in a source directory and executes the main script over SSH commands.

The Scala planning program contains a parsing function that extracts the result data from the console output of the Python scripts. This result is a Scala value which the evaluation procedure can further analyze according to the experimental setup.

6.2.4 Examination of Randomized Algorithms

The randomness of the samples affects the quality of the planning results and the algorithms' runtime. This aspect has to be addressed in the experimental design to raise the significance

[1] https://moveit.ros.org/
[2] https://ros.org

of evaluating a configuration. For that reason, the execution of multiple planning program instances admits the computation of the average-case running time and path length, mitigating the variations of result values caused by randomized algorithms.

In OMPL, there are two general kinds of planners that expose a different termination condition that directly impacts the path length and runtime of the algorithm. While geometric planners terminate as soon as they find a feasible solution, the optimizing planners try to improve the result by refining their internal graph structure with additional iterations. For this reason, an optimization run determines the percentage of successful runs for a given time limit. Moreover, the different termination conditions lead to a different number of samples for these algorithmic families. For this reason, the experiments use path simplification as a post-processing step to reduce the impact of samples and roadmap construction on the resulting path length.

6.3 A Repository for Sampling Based Motion Planning Programs

The sampling-based motion planning programs expose variability points that exceed the configurable parameters for this study. The repository covers the following aspects, and each variability point can form at least two different variations:

- Planning space
- State cost, distance metrics, and optimization objective
- Input data type and result type
- Planner type
- Sampling strategy
- Collision detection technique
- Path simplification and its parameterization

For this study, the planning space is determined to be $SE(3)$ without state cost and Euclidean distances for the robot's position. The optimization objective refers to the default optimization objective of optimizing planners, which is the path length. A post-processing step modifies the resulting paths using OMPL's integrated simplification functions. The repository for the component-based synthesis of sampling-based motion planning programs contains software components implemented as CLS combinators. It includes 24 planner combinators, six

sampler combinators, and three state and motion validation combinators. Appendix D.1 and D.2 provide an overview of the range of planners and samplers employed in this chapter.

6.3.1 Top-level Combinator

OmplPlannerCombinator[A,B] :
PlanningSchema[B] $\cap$ $any\_sbmp\_planner\_type \rightarrow$
SubstitutionSchema $\cap$ $any\_sbmp\_state\_validator\_type \rightarrow$
SubstitutionSchema $\cap$ $any\_sbmp\_motion\_validator\_type \rightarrow$
SubstitutionSchema $\cap$ $any\_sbmp\_simplification\_type \rightarrow$
(A $\rightarrow$ SubstitutionSchema) $\cap$ $sbmp\_input\_data \cap any\_dimensionality\_type \rightarrow$
OmplPlanner[A, B] $\cap$ $sbmp\_planning\_algorithm$

(6.3)

A top-level combinator constructs the specified Scala program. For the given experiments, it will yield a function typed `ProblemDefinitionFile → List[List[Float]]` which loads an OMPL configuration file, performs the substitution, and subsequently executes the generated motion planning program. It extends a Scala trait with two type parameters. These parameters denote the native types of the argument and the result type of the evaluated program, i.e., the extension of this trait yields a combinator with corresponding native types. This approach allows for an easy setup of top level-combinators with different types. The type expression in Equation 6.3 shows the parametric type expression. For the given experiments, CLS uses a planning combinator with target type `OmplPlanner[ProblemDefinitionFile, List[List[Float]]]` to compose planning programs.

The native data type of the planning result is `List[List[Float]]`, where the inner list of values denotes a point in the n-dimensional planning space. Parametric quaternions encode the orientation of the robot. A point in $SE(3)$ starts with three float values representing the position given by the Cartesian axes, followed by four list elements for the rotation representation using the x, y, z-components of the quaternion vector, followed by its scalar value. Using linear interpolation, one can compute a continuous path based on this list of states.

The repository makes use of the subtyping relation for semantic types by using a semantic taxonomy. For instance, the semantic types $any\_planner\_type$ and $any\_sampler\_type$ in Equation 6.3 are base types that match every semantic type describing a planner or sampler, respectively.

A data substitution schema is produced based on the input type and has the signature `A → SubstitutionSchema`. For this study, this function loads the problem-specific data

based on the information held in the provided configuration file. The resulting scheme contains Python code to specify the planning space and integrates the start and goal state. Moreover, the code generation introduces the problem-specific references to the geometric models for the environment and the robot model. The corresponding geometries are loaded as bounding volume hierarchy objects using the **Flexible Collision Library** (FCL, [74]), which enables collision checks for the state and motion validator implementations. The corresponding combinators handle the allocation of correct state and motion validators for the OMPL planning instance.

The repository offers further configurations not used for the experiments. For instance, it can configure programs that read the samples from OMPL programs. This function requires different data structures and parsing functions and can help to analyze the sampling behavior for an algorithm configuration. Moreover, the adaption of the planning space involves different geometric transform functions for collision checks, search space definitions with corresponding state representation, and parsing functions.

While these variability points are not immediately relevant for the work in this chapter, they show how CLS can help to configure and analyze the performance of a given algorithm or family of algorithms.

6.3.2 Dynamic Construction of the Repository

A large number of combinators and variability points would lead to many inhabitants. Moreover, solving the inhabitation for the complete repository would entail an extensive search. Accordingly, this work does not use the reflection feature of CLS, and the dynamically constructed repository contains only combinators matching the algorithm configuration. Based on this configuration, the combinator objects are built and dynamically added to the repository. This principle leads to a semantic structure that yields fast inhabitation results even for large feature spaces. The configuration also helps to determine the native and semantic target types for the inhabitation request that yields one inhabitant for the specified type. The algorithm configuration class incorporates default values for algorithm configuration to allow partial specifications.

6.4 Planners

Continuous efforts of the scientific community resulted in a wide variety of planning approaches, and the repository includes numerous components for planning algorithms provided by the Open Motion Planning Library. Due to the large number of planners, this section

will not cover all of them in detail. However, for the analysis of the experimental results of this work, it is helpful to understand some basic principles of established planners. For this reason, this section will provide a short overview of selected sampling-based motion planning algorithms.

6.4.1 Probabilistic Roadmap Method

The **probabilistic roadmap algorithm** (**PRM**, [56]) is one of the most prominent planning algorithms. It was established in 1996 by Kavraki et al. [55] and is one of the earliest and most influential works in sampling-based motion planning. As the name suggests, the planner constructs a roadmap of the entire environment that can be used for multiple queries. A node of this roadmap is a valid sample, and an edge represents a feasible direct path between these samples. Variations of the PRM planner implement lazy collision checking (LazyPRM, [17]) or optimizing planning (PRM*, [56]).

6.4.2 Tree-based Planners

The **Rapidly-exploring Random Tree** (**RRT** [62]) algorithm is a renowned tree based planner developed by LaValle. This iterative algorithm focuses on a fast expansion of the tree into the unexplored regions of the free configuration space. It is intended for single-queries as it starts from the root, i.e., the graph is problem-specific and not suitable for multiple queries. The algorithm checks if the goal state is reachable from the tree for every iteration. For this reason, RRT allows "anytime" planning as it finds feasible paths fast, or it can yield approximate solutions otherwise. In contrast, the PRM in its initial design uses an a priori sampling phase.

The **Expansive Space Trees** algorithm (**EST**, [45]) builds trees from the start and the goal state, therefore resembling a bidirectional planning approach. It assigns a weight to the tree nodes based on the inverse of the number of samples in its neighborhood, defined by a configurable distance metric. As a result, areas with a low sample density are more likely to be explored. The algorithm returns a path when the trees originating from the goal and start states can be connected.

The **Single-Query Bi-Directional Probabilistic Roadmap Planner (SBL)** [84] also builds trees that originate from the start and end state. The idea of this planner is to use a lazy evaluation strategy for connectivity paths. It initially assumes that every node and edge of the trees are valid. Under this assumption, it tries to find a result path and then tests it for validity. Detection of a collision leads to removing the corresponding node or edge from the graph, and the next iteration will further expand and test the tree. The algorithm in its initial design

also incorporated a multiple-stage collision checking strategy with different resolutions and a corresponding priority queue for motion validation. Moreover, the sampling strategy uses a density metric for regions to avoid oversampling.

6.5 Sampling Strategies

Hsu et al. introduced a study to confirm the intuition that sampling strategies can have a clear impact on the performance of planners [46]. Using the PRM as an example, they argue that a visibility property of the environment called *expansiveness* ([45]) explains the algorithm's particular good performance for specific planning tasks. They conducted experiments and presented a formal way to classify geometric shapes. Based on experiments, they argue that PRM learns connectivity in $\mathscr{C}_{free}$, which usually exposes non-uniform visibility over the geometric space. Therefore, using non-uniform samplers can benefit the overall planner performance by identifying and focusing on these hard-to-learn areas. They explained how a sampling strategy influences the planning results by increasing the samples' chances to find connectivity to the graph.

Many strategies deal with particular challenges in planning with different ideas. For this reason, the repository includes some of the most prominent and established sampling strategies to represent this vital variability point. This section introduces these sampling strategies with their core concepts, mechanics, and strengths.

6.5.1 Uniform Sampling

An obvious strategy for the generation of samples is a random uniform sampling of the workspace. This approach was the incorporated sampling strategy of the initial publication of the PRM [56]. For this particular planner, the concept of valid state samplers was established to speed up the planner's performance. A uniform generation of state samples does not guarantee a uniform density of samples across the space. For this reason, uniform sampling is also performed with the Halton sequence [19], which leads to a quasi-random uniform distribution of Halton points in n-dimensional spaces. The left side of Fig. 6.2 shows a roadmap constructed with Halton sampling.

6.5.2 Gaussian Valid State Sampling

Boor, Overmars, and van der Stappen presented the Gaussian sampling strategy in 1999 [18]. This sampling strategy intends to cover the difficult parts of the free configuration space. It

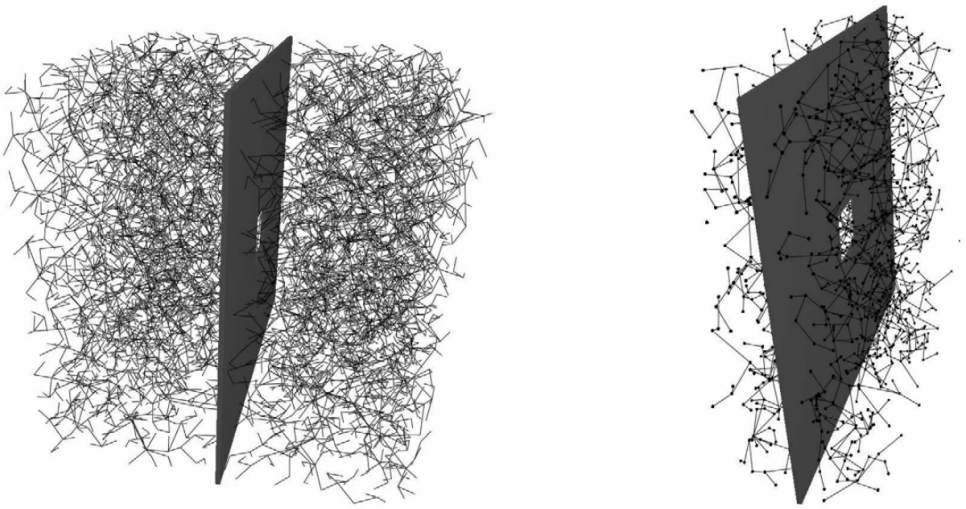


Figure 6.2: Halton Sampling and Gauss Sampling

makes use of elementary operations to find valid samples near objects. The right side of Fig. 6.2 illustrates the outcome of Gaussian valid state sampling. These samples are more likely to find paths close to objects, ignoring the uninteresting parts of large free spaces. The strategy starts sampling by randomly selecting a first sample and a random distance value based on a Gauss distribution. A second sample is generated at this particular distance to the first sample. If the constructed sample pair contains one valid and one invalid sample, the sampling strategy returns the valid sample but discards both samples otherwise. The standard deviation of the distribution is an important parameter to tune the behavior of the sampling strategy as it determines the sample density in the spaces close to the obstacles. A small value will lead to sample points near the obstacles' surfaces but will likely cause more unsuccessful sampling attempts. While the initial paper only presented a two-dimensional experiment, later work describes the extension of this sampling strategy to a three-dimensional planning problem in which the robot has six degrees of freedom [18]. The strategy has also been shown to handle industrial-scale problem instances containing hundreds of partially overlapping obstacles.

6.5.3 Obstacle-Based sampling

The obstacle-based sampling (Fig. 6.3) was introduced by Amato et al. [4, 5] and it shares the principle idea with the Gaussian valid state sampler. It aims to target critical regions of a configuration space by looking for samples in the vicinity of obstacles. However, this sampling strategy does not use a Gaussian distribution but a different approach to find these samples. Moreover, a single sampling iteration of this strategy yields multiple valid samples.

A sampling iteration starts by finding a point inside an obstacle, from which multiple rays advance in random directions of the configuration space. After that, a binary search on these rays finds collision-free states, i.e., the robot's configurations in $\mathscr{C}_{free}$. The resulting set of valid configurations then helps to find an even distribution over the obstacle's surface, using the distance between neighboring points to identify improvable regions and introducing new samples accordingly. This sampling strategy suitable for workspaces that include a lot of narrow passages and for which $\mathscr{C}_{free}$ represents only a small share of the overall planning space. These cluttered spaces expose a higher chance of finding samples inside of obstacles, which are the starting point for the sampling procedure. Experiments in [4, 5] show that this approach is suitable for 3-dimensional workspaces.

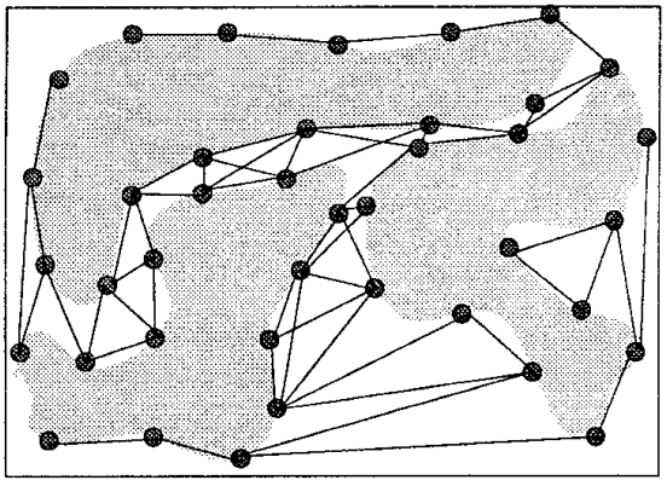


Figure 6.3: Obstacle based roadmap example, taken from [4]

6.5.4 Maximum Clearance Sampling

Wilmarth, Amato, and Stiller proposed the maximum clearance state sampling in 1999 [99]. The strategy aims to mitigate the degradation in performance in narrow spaces by producing samples with higher clearance to the bounding objects, as illustrated in Fig. 6.4. For this reason, samples are retracted to the middle-axis of the free configuration space after their initial random generation. As a result, a sample is usually equidistant to at least two geometries, and these samples are more likely to be connectable to other valid samples. This approach improves the planner's performance for problems that require the traversal of long, narrow passages.

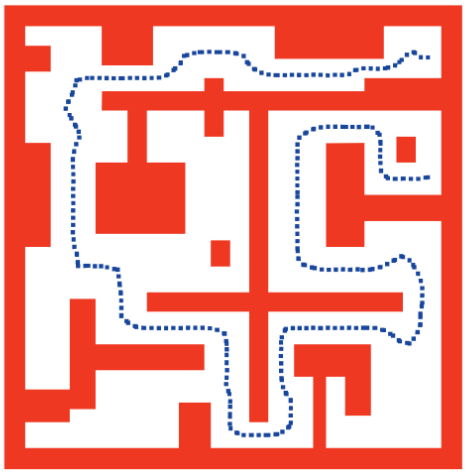


Figure 6.4: Example for a maximum clearance path, taken from [36]

6.6 Assignment of Planners and Samplers

CLS semantic types encode the rules of composition and can thus capture domain knowledge. The repository's semantic layer expresses which specific planners can use samplers and optimization objectives. The semantic types displayed in Equation 6.4 contain this information for the OMPL implementations of the two planners Probabilistic Roadmap Star (PRM*, [55]) and Expansive Space Trees (EST, [45]).

$$
\begin{aligned}
&\Gamma_s = \{ \\
&\text{PRMStarSchema}: \\
&(any\_opt\_obj \rightarrow sampler\_space \rightarrow PRMStar) \cap \\
&(any\_opt\_obj \rightarrow sampler\_valid\_state \rightarrow PRMStar) \cap \\
&(any\_opt\_obj \rightarrow sampler\_informed \rightarrow PRMStar), \\
&\\
&\text{ESTSchema}: \\
&obj\_path \rightarrow sampler\_valid\_state \rightarrow EST, \\
&[...] \\
&\}
\end{aligned}
\tag{6.4}
$$

PRM* is capable of utilizing space samplers or valid state samplers. Hence, it can incorporate informed samplers, which are a subclass of space samplers. The semantic type also encodes that PRM* is an optimizing planner that can consider various OMPL optimization objectives (e.g., path clearance or state cost integral). On the other hand, the EST planner only supports

the path length optimization objective and requires a valid state sampler. These encodings were manually collected for all considered planners and formulated as type expressions.

Planner schemes include the templating schemes for the planner's definition and the configuration of the sampling strategy. Suppose the inhabitation algorithm can not find a matching argument term for the planner combinator. In that case, it will not assemble a planner scheme, and CLS will fail to find result terms. This situation indicates an invalid combination of planner and sampler, and the active learning loop will consider this information for successive iterations of the optimization run.

6.7 Collision Detection

For sampling-based motion planning, collision detection is used for mainly two tasks. These are validity checks for a given state and a motion, represented by linear interpolation of one start state to a target state. The validation of a state incorporates applying a corresponding transformation to the robot model, and the state validity arises from the collision situation of this transformed object regarding the environment model. The basis of motion validation is either a discrete motion validation or continuous motion detection. A discrete motion validator uses the state validation at a specified resolution. It evaluates motions by validating interpolated states and yielding the overall result.

For the **continuous collision detection** (CCD) of motions, libraries use multi-staged collision detection procedures based on geometric properties. For this reason, there is a *broad phase* and a *narrow phase* collision detection. The broad-phase computation uses space partitioning techniques and data structures, such as bounding boxes, that allow rapid assessment of states and motions. These data structures can detect if a collision is not imminent because objects are not close to each other. This step can also provide a rough estimate for motion intervals that require more thorough collision checks. On the other hand, narrow-phase collision detection is a precise computation that can detect and analyze collisions, for instance, by exposing the intersecting volumes. Some strategies for narrow-phase collision detection include bounding volume hierarchies, proximity queries, or feature tracking algorithms such as the Lin-Canny method [64] or V-Clip [68]. These techniques usually come with an additional computational cost.

While validations of motions are usually more costly, they expose additional information that the planner might use. For instance, the collision detection for a motion often includes the determination of the point of impact. Some planners can regard this information in learning the connectivity of $\mathscr{C}_{free}$.

The **Flexible Collision Library** integrates various data structures and algorithms for collision detection. It has been well known in the robotics community since it was part of ROS, and it is now an active constituent of the MoveIt! library. For this work, the motion and state validators use the Python-FCL library, an unofficial Python interface for the Flexible Collision Library.

Motion validation with continuous collision detection can lead to false negatives, i.e., it wrongly considers invalid motions collision-free, resulting in incorrect paths. Apart from bugs in the collision detection framework, a coarse collision detection resolution can be the source of error. Therefore, an additional evaluation after computation checks the resulting paths with a discrete collision detection with a high resolution. This evaluation step does not contribute to the measured computation time of the algorithm configuration.

6.8 State Cost and Optimization Objectives

While path length has been the primary objective for the quality of motion planning results for a long time, the focus of research shifted towards a differentiated evaluation with growing maturity. An early adaption of this principle was finding minimum clearance paths [37]. In this work of Geraerts and Overmars, a post-processing step manipulates the nodes of the resulting path, aiming to respect a given minimum clearance constraint.

Optimizing planners often use customized state cost functions that map a state of the configuration space to a numeric value. The D* algorithm featured an early proposition of this concept [90]. Jaillet, Cortés, and Siméon presented a general adaption to sampling-based motion planning in 2010 [50]. They proposed the use of a cost map and a corresponding state transition cost to optimize planning results. This cost map described cluttered terrain, and the state transition cost represented difficult travel conditions for the robot.

The introduction of a cost map requires the adaption of the planning algorithm's optimization objective. One example is minimizing the integrated state cost over a path.

6.9 Results

6.9.1 Problem instance "Abstract"

The problem instance "Abstract" in Fig. 6.5 is an unordered composition of simple geometric forms. The robot is an L-shaped 3D rigid body that has to move from one tube to another. In this particular environment, there are many geometric homotopy classes of solution paths. Moreover, the starting point resides in a narrow space that puts high requirements on the planner as it demands a sequence of translations with simultaneous rotations. The environment also incorporates a large free space that can impact the performance of planners.

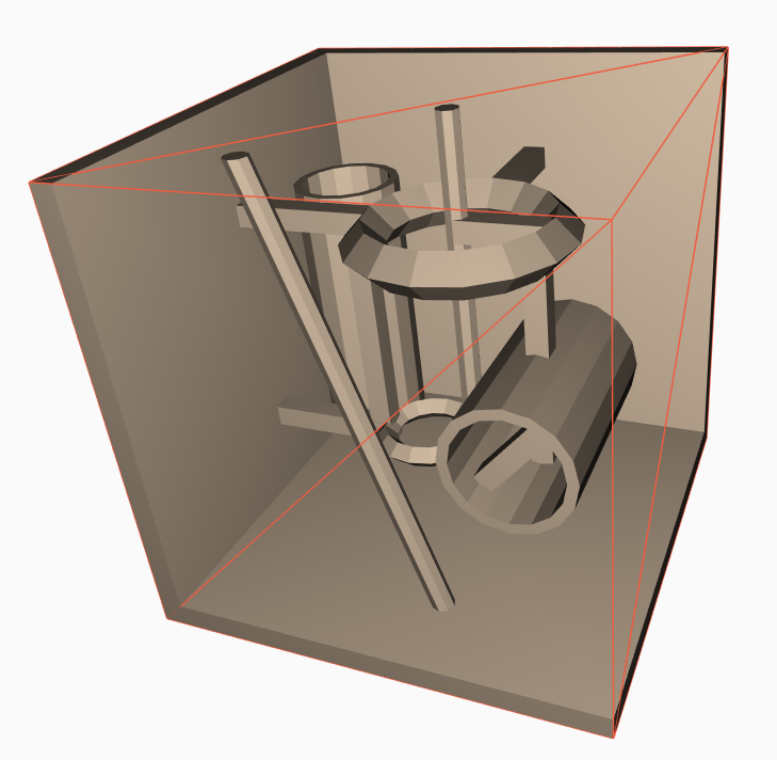


Figure 6.5: Problem instance "Abstract"

Fig. 6.6 shows the most relevant data of successful computations for an optimization run. Every point represents a single iteration result, i.e., the averaged result of 10 program instances for a specified algorithm configuration. The position of a measurement illustrates the optimization criteria computation time and path length while the shape and color depict the selected planner and sampling strategy. The result only considers valid algorithm configurations that yielded exact solutions, i.e., approximate solutions are assessed as a failure.

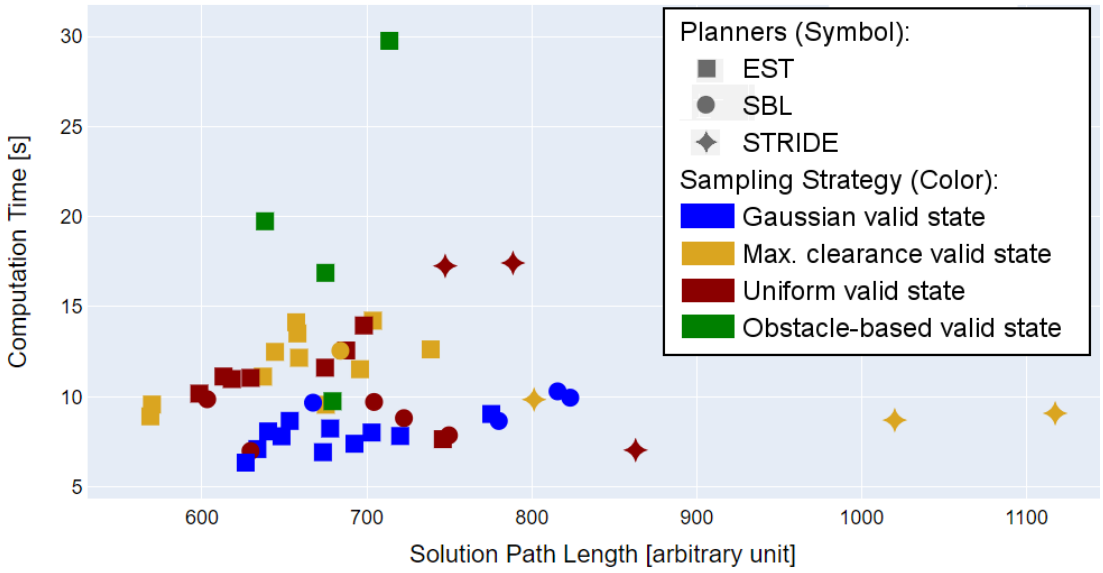


Figure 6.6: Result for the Problem instance "Abstract"

The minimal observed averaged length of simplified paths is 580 length units, corresponding to the lower spectrum of the results documented in the OMPL Planner Arena[3], an OMPL-provided benchmark database. The minimum time to solve the planning task is at least 7 seconds, including simplification. However, the measured times are not comparable to the OMPL planner arena data due to differences in the experimental setup, such as utilized hardware and collision detection parameterization.

For the given experimental setup, the machine learning procedure determined the planners, Expansive Space Trees (EST, [45]) and SBL ([84]) to be well-performing for the given planning problem. The common characteristic of these planners is the lazy evaluation of motion validity. In this experiment, the discrete motion validator has a sample resolution of 0.15 length units. Accordingly, the admissible point of collision error for the continuous collision detection is 0.15 length units. Due to this high resolution, motion validity is a precise calculation that is costly regarding computation time. This influence of the motion detection parameterization explains why these planners gain an advantage over other planner implementations. The result also shows the impact of the selected sampling strategy. Gaussian valid state sampling positively impacts computation times, while maximum clearance valid state sampling and uniform valid state sampling expose good results regarding the path length.

[3] http://plannerarena.org/

6.9.2 Problem instance "Alpha 1.5"

The problem instance "Alpha 1.5" resembles a geometric puzzle (Fig. 6.7). The aim is to disengage the red shape from the yellow object using a collision-free path. For these shapes, the rotation of the movable object has a high impact on the validity of a state. Thus, the resulting path in $\mathcal{C}_{free}$ traverses a long narrow passage that leads to a large free area.

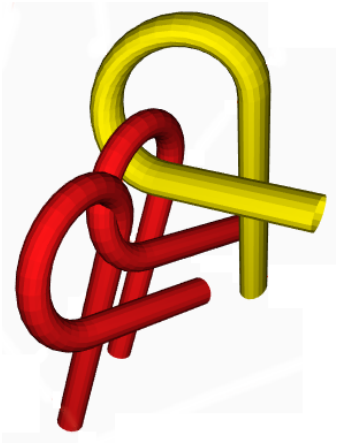


Figure 6.7: Problem instance "Alpha 1.5"

Fig. 6.8 shows the plot for this second experiment. Here, the planning algorithms EST and SBL are prevalent for the same reasons as for the Abstract problem instance. The utilization of SBL leads to the detection of the shortest paths. Moreover, the STRIDE planner also yielded strong results. It appears to find solutions in a short amount of time but with a longer path length. While planners seem to impact the path length and computation time, the sampling strategy does not significantly influence the outcome of a motion planning program.

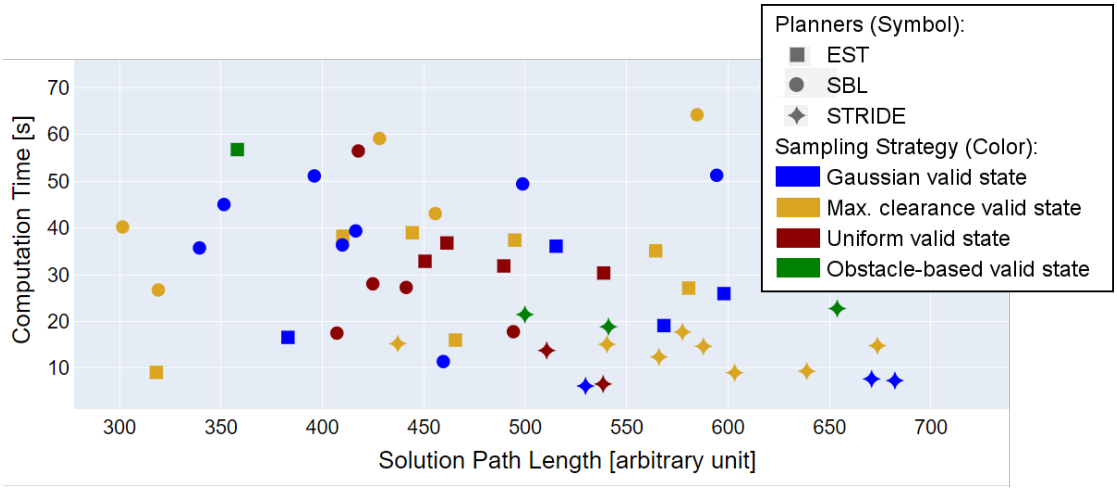


Figure 6.8: Result for the Problem instance "Alpha"

6.9.3 Problem instance "Home"

Fig. 6.9 depicts the problem instance "Home," in which a result path has to represent a collision-free transport of the red table. The resulting path is a long, narrow path that involves rotation to avoid collisions. This experiment incorporates different collision detection parameters that allow a lower collision detection resolution of 1.0 length units. Moreover, the motion validator implementation performs a simplified motion check as a first step, using a sphere with a fixed radius of 1.0 length units instead of the robot model. If the motion validator does not detect a collision, it transforms the robot's collision detection data structure for a precise collision detection query.

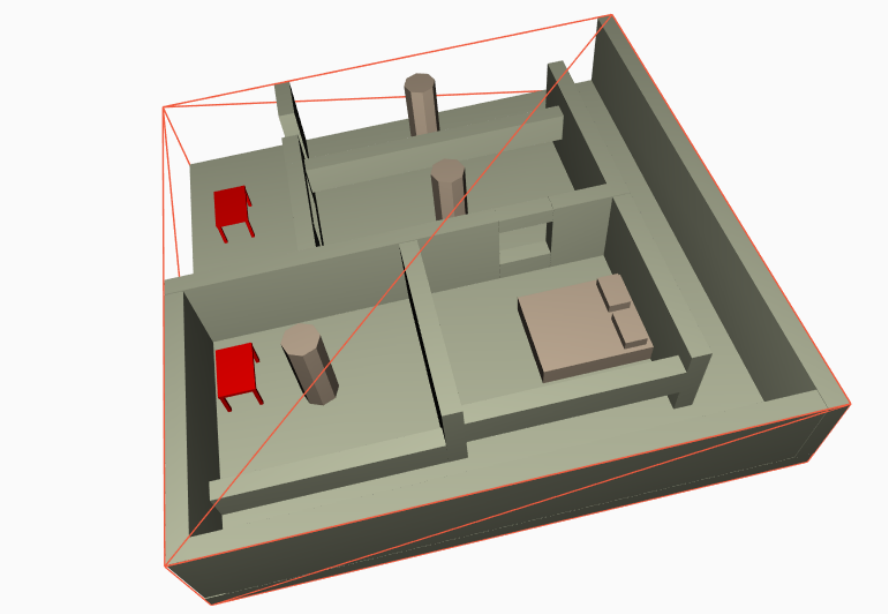


Figure 6.9: Problem instance "Home"

The result in Fig. 6.10 shows that the family of PRM-based planners performed well. The active learning loop has no initial information about the conceptual proximity of the planners PRM, PRM*, and Lazy PRM*. Just as in the previous results, this permits conclusions regarding planners' operational strengths and similarities for a given planning problem. The results show that the machine learning procedure can identify families of planning algorithms without explicit knowledge of the design space. Moreover, the experimental design enables the comparative analysis of optimizing planners (PRM*, Lazy PRM*) and geometric planners (PRM).

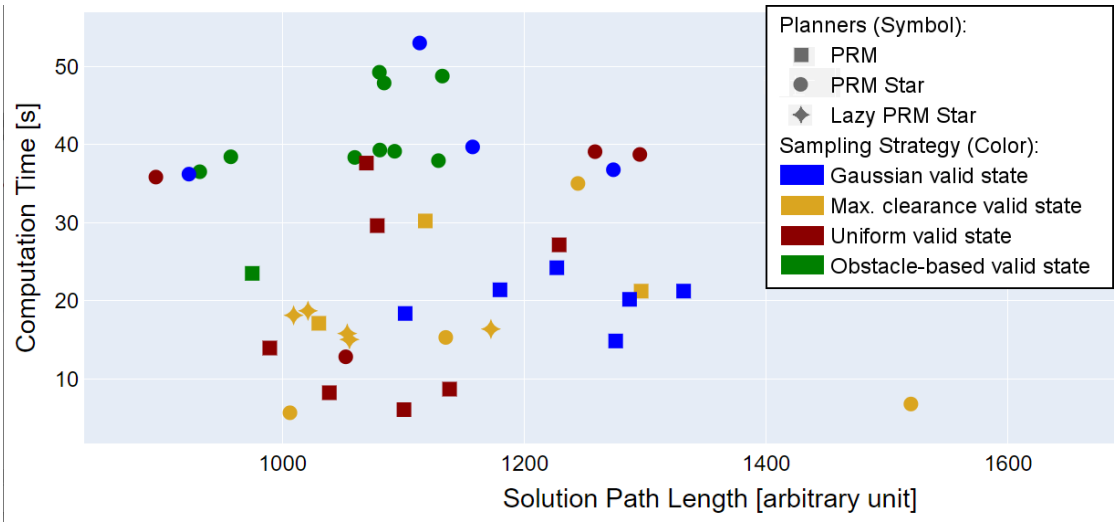


Figure 6.10: Result for the Problem instance "Home"

6.9.4 Success Rate

The probability of finding a feasible program configuration is an indicator for the positive impact of the applied optimization techniques. Each iteration executes and evaluates 10 generated program instances. The number of successful path computations per iteration is computed based on the output and evaluated as a success percentage. A simple moving average is calculated for every optimization run, using the unweighted mean value of 10 previous iterations. That way, the probabilistic nature of the algorithmic family of sampling-based motion planning algorithms can be mitigated. Fig. 6.11 illustrates the trend of the success rate for multiple optimization runs.

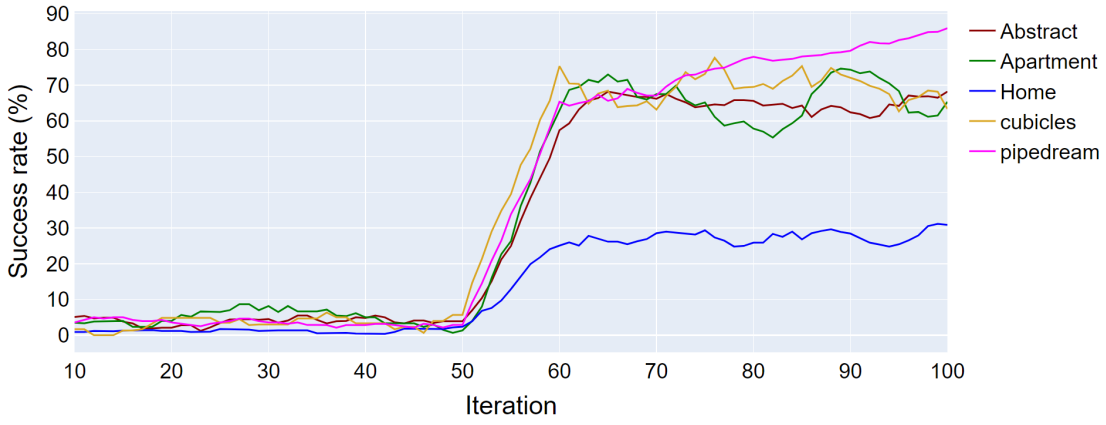


Figure 6.11: Moving average over the success rates of the iterations for multiple optimization runs

The results show a significantly growing success rate at the end of the DoE phase after 50 iterations. This finding suggests that the employed methodology works in the domain of sampling-based motion planning. The exploration involves sampling unknown, possibly invalid, algorithm configurations, which leads to a success rate lower than 100%. The "Home" planning problem is challenging due to the long solution path and narrow passages. Thus, the maximal allowed planning time of up to 90 seconds results in a lower success rate.

6.9.5 Parameter Importance

The utilized optimization framework can inform about the importance of the features. Table 6.1 shows a corresponding overview of the impact of the parameter selection for given objectives. This metric covers a complete optimization run for the problem instance "Abstract" as described in Section 6.9.1. As expected, the planner selection has the biggest impact on the different optimization objects. The low importance of the motion validator selection states that both variants, i.e., continuous motion validation and discrete motion validation, work equally well for the given problem. Therefore, this value confirms the benefits of the planners' sparse use of collision checks for motions.

objective	planner	sampler	motion validator	max. planning time
path length	0.60	0.15	0.04	0.21
computation time	0.50	0.19	0.02	0.29
failures	0.44	0.28	0.03	0.24

Table 6.1: Parameter importance for the problem instance "Abstract"

6.10 Related Work

In a recent work from 2019, Jamshidi et al. presented a concept to explore the configuration space for self-adapting robots [51]. An offline machine learning phase determines the set of Pareto optimal configurations concerning localization error and energy consumption estimates. The robotic system's configuration points include sensor usage or localization algorithms with a configurable number of particles, and the performance for selected configurations is measured.

The findings from the offline learning phase enable the robotic system to adapt to changes in the environment or inaccurate measurements of the battery state. The work demonstrates the integration in a task planning context in which energy consumption and timeliness of task completion are the primary planning objectives. Model-checking determines the selection of

the robot configurations during run-time. For instance, a robot may switch to a configuration with a decreased localization accuracy to preserve the battery.

The work shows that incorporating machine learning is a viable approach to handle highly configurable systems. The general idea of the offline learning phase has similarities to the methodology presented in this chapter. However, the work focuses on task planning, i.e., motion decisions based on tasks and adaption to changes in the robotic systems state or the environment. The study assumes the availability of a weighted graph representing the environment.

Morales et al. select suitable sampling-based motion planning algorithms for environments by incorporating geometric feature extraction [69]. Based on recognized features, they use machine learning to assign planners accordingly. The approach aims to yield a general-purpose planner, which divides the planning environment into subregions. The corresponding partial roadmaps are connected to form an overall roadmap for graph traversal. However, their work covers only a small space of possible planner variations as it uses PRM [56] for accessible spaces and OBPRM [5] for cluttered regions.

In [83], Saeedi et al. use a domain-specific framework for the design space exploration of simultaneous localization and mapping (SLAM) systems, a subdomain of robot navigation. The set of design parameters describes the algorithmic configuration, compiler settings, or hardware properties. The study considers the absolute trajectory error and execution time as optimization objectives.

6.11 Discussion

CLS is used to generate Python scripts, and the methodology is extendable to further programming languages. It is language-agnostic as the code of any programming language can be produced and manipulated with synthesized Scala programs. The support of C++ planning programs could benefit a broad range of developers and allow access to various planning instruments such as further collision detection libraries.

The repository for sampling-based motion planning algorithms consists of 70 combinators, and it supports algorithm configuration with dynamic repository construction or the use of type variables. Seven type variables yield a substitution space with 6.912 distinct variable assignments, which allows estimating the number of synthesizable programs. The subset of the repository for this study represents a space consisting of 264 unique feature configurations. While currently limited to OMPL programs for rigid body planning, additional combinators that emerge from real-life scenarios can extend this repository, allowing the applicability to a

broader range of problems and robot models. The exploration of different design spaces, e.g., similar to those considered in [83], could study the general applicability of this methodology. This requires the adaption of the algorithm feature space, combinators, algorithm feature model, and Python templates. At the same time, the structure of the approach remains as depicted in Fig. 6.1. CLS components are reusable and can provide a fast setup of experimental evaluation procedures. This way, the learning-based exploration of the algorithmic feature space could help to develop and evaluate planners, sampling strategies, or collision detection techniques.

While Hypermapper offers a great range of functionality that benefits the given experimental design, it is possible to use different optimization frameworks by setting up framework-specific black-box functions or by formulating models that integrate a synthesis-based evaluation. The loose coupling between black-box function, inhabitation, and evaluation facilities supports alternative optimization techniques. Due to this flexibility, the methodology is transferable to other domains with inherent variability.

The machine learning-based approach demonstrates the exploration of the design space of algorithmic families. Moreover, this procedure allows a hybrid search that considers the parameterization of the generated programs by specifying the maximal allowed computation time. An assessment of numerical parameters is also possible by introducing different combinators with preset parameters. In this work, this principle allowed different values for the collision detection resolution. These results can form the basis of an automated exploration and exploitation approach. The first step (exploration) collects a set of candidate solutions from the design space. In a second step, a candidate-specific optimization improves the parameterization of this particular algorithmic configuration.

Chapter 7

Conclusion

This work builds and analyzes algorithmic spaces with component-based Combinatory Logic Synthesis. Motion planning algorithms were selected as the field of study, and CLS's ability to handle structural variability permits the configuration of algorithmic variants.

Application-specific requirements and constraints drive innovations in this particular field of robotics. Moreover, planning plays an essential role in digitalization processes, and it benefits from automation. The corresponding demand for configurable software solutions arises from problem-specific constraints, objectives, performance indicators, and the algorithmic design space with high variability.

There are several reasons which motivate the selection of motion planning as the domain of interest. The domain comprises several sub-problems that expose different requirements. Moreover, the algorithmic family builds upon several years of research, which resulted in various available techniques. The fields of application have been selected accordingly, and the problem-specific generation of runnable programs led to the design of methodologies to handle spaces of high variability.

The work covers the planning problem subdomains of sampling-based motion planning and path coverage planning with domain-specific constraints. Both domains expose a rich set of variations. While motion planning exposes algorithmic variations, the variations in the tool path planning work represent particular sequences of planning operations.

7.1 Results

This work includes three proof-of-concept studies that use CLS in the following ways to generate runnable programs:

- Configuration of algorithms by precise specification with type variables
- Incorporation of general type specifications that lead to a space with high structural variability
- Configuration of algorithms by building a problem-specific repository

Every study included in this work yielded promising results. The substitution of variables precisely guides the synthesis results for combinatorial motion planning algorithms. The brute-force search for machining strategies can handle complex shapes and consider different planning problem objectives. The semantic layer of the repository encodes predefined machining sequences and the suitability of planning components for different materials. The machine learning approach handles the generation of sampling-based motion planning algorithms and effectively samples the design space. It can manage hard planning problems, and the user is no longer required to implement programs manually to find a well-suited configuration for a given problem. The optimization results give an insight into the planning problem at hand and the influence of particular elements of the planning program.

7.2 Combinatorial Complexity

The component-based approach of CLS allows the definition of spaces with high variability based on combinators. This work includes several ways of managing the resulting combinatorial complexity to achieve viable inhabitation runtimes. For instance, a restrictive substitution of type variables can reduce the search space. Moreover, including only relevant components for a specific inhabitation request helps to speed up the synthesis, and the dynamic configuration of the repository can significantly reduce the search space for the inhabitation. An alternative approach allows the use of general type specifications that yield a considerable number of inhabitants. It applies the framework-provided lazy enumeration function to implement a brute-force search.

The experiments demonstrated several ways to manage a large number of variations. With Combinatory Logic Synthesis, the developer can fine-tune the granularity of the components, encapsulating multiple terms or building more significant predefined terms that are likely to fit

the current problem. This aspect could inspire a multi-staged synthesis approach in which the first step generates components that form the basis for a consecutive synthesis procedure. For instance, a preceding synthesis step for the machining experiment could involve generating planning components with particular properties regarding tool wear.

7.3 Incorporated Methods

This work presents methodologies to use CLS in optimization contexts, using machine learning or an automated domain-specific brute-force search. Both approaches benefit from CLS's capability to capture the variability of selected subdomains. The applicability and performance of different planning algorithms concerning the planning environment inspired the development of a machine learning approach with CLS, and this principle is transferable to other planning domains.

The identification of the domain's variability points directed the implementation of corresponding combinators. The design of the repositories that generate synthesis goals according to types must match the given planning problem. The problem-specific evaluation of inhabitants involves running the resulting programs and analyzing the result.

This work demonstrated that CLS could handle large spaces with the support of appropriate methods. Moreover, meta-programs can profit from a heterogeneous set of libraries from different programming languages, including Java, Scala, Python, and C++.

7.4 Outlook

The proposed concepts can inspire a sophisticated optimization framework that considers algorithmic families' design space and numeric parameterization. While this work presented a straightforward method to map a selected point in the problem domain to a corresponding point in the design space of algorithms, this principle can be adapted further, allowing the integration of alternative optimization procedures or design of a multi-layered optimization approach. A CLS-based framework should allow easy adaption to new problem instances and should incorporate different optimization procedures.

The developed concepts can extend to other domains of interest. While motion planning is an appealing domain from an algorithmic perspective, integrating task planning from logistics or manufacturing can benefit practical applications. For instance, the generation of simulation models and an automated evaluation procedure could adapt the machine learning

procedure proposed in this work. The application to real-life problems can require the design of methodologies to integrate parameterizable components or weighted objectives.

The approaches in this work relate to some of the search strategies employed in other synthesis techniques, such as enumerative search or data-driven learning. Further methods from this research field could inspire a search technique for the result set of inhabitants based on the CLS-generated grammar. Such an approach would no longer rely on the enumeration of terms and could use machine learning methods to determine a problem-specific ranking for inhabitants.

Appendix A

Inhabitant Trees for the Examples in Section 4.5

A.1 Tree for the Grid Approximation Example

```
AkkaGraphSceneSeg
    UnityMqttAkkaSourceScene
    UnityMqttAkkaSourceTask2D
    CmpTopLevelCombinator
        SceneToScenePoly
            CubeToPoly
                RectangleVertices2D
                ApplyAffineTransform2DVertexList
            ObstacleSceneBoundingCut2D
        GridCombinator
        RoadmapCombinator
            Neighbours2D
            CellToCentroidCellNaive
            WithoutNewNodes
            WithoutNewNodes
            EdgesCentroidOnly
            NodesEdgesStartEndNearest
        DijkstraSpCombinator
    UnityMqttAkkaSinkSceneSegGraphPath
```

A.2 Tree for the Triangulation Example

```
AkkaGraphSceneSeg
    UnityMqttAkkaSourceScene
    UnityMqttAkkaSourceTask2D
    CmpTopLevelCombinator
        SceneToScenePoly
            CubeToPoly
                RectangleVertices2D
                ApplyAffineTransform2DVertexList
            ObstacleSceneBoundingCut2D
        TriangulatePolyParametrized
        RoadmapCombinator
            Neighbours2D
            CellToCentroidCellNaive
            WithoutNewNodes
            WithConnectorNodes
            EdgesCentroidToCellVertices
            NodesEdgesStartEndNearest
        DijkstraSpCombinator
    UnityMqttAkkaSinkSceneSegGraphPath
```

A.3 Tree for the VCD Example in Section 4.5

```
AkkaGraphSceneSeg
    UnityMqttAkkaSourceScene
    UnityMqttAkkaSourceTask2D
    CmpTopLevelCombinator
        SceneToScenePoly
            CubeToPoly
                RectangleVertices2D
                ApplyAffineTransform2DVertexList
            ObstacleSceneBoundingCut2D
        VcdLineSweepJTS
        RoadmapCombinator
            Neighbours2D
            CellToCentroidCellNaive
            WithCellVertices
            WithConnectorNodes
            EdgesAllCellVertices
            NodesEdgesStartEndNearest
        DijkstraSpCombinator
    UnityMqttAkkaSinkSceneSegGraphPath
```

Appendix B

Remarks on Published Work

Title:
A Synthesis-based Tool Path Planning Approach for Machining Operations

Authors:
Tristan Schäfer, Jim A. Bergmann, Rafael Garcia Carballo, Jakob Rehof, Petra Wiederkehr

Published in:
Procedia CIRP 104C (2021) pp. 917-922

Contribution of the author:
The author is responsible for the initial idea of the component-based tool path planning approach and implemented the software system. This includes selecting and implementing planning components, data structures, brute-force search, evaluation procedure, generation of machining instructions, and the Scala implementation of the acceleration model and calculation of machining times for resulting paths. The author performed the experiments, data visualization, and the design and transformation of geometric models to suitable data representations.

The author did **not** participate in determining the tool parameterization and the selection of materials used in the paper. Moreover, the author did **not** conduct the validation with the geometric physically-based simulation system.

Comparison to Chapter 5 of this dissertation:
This dissertation further addresses the mechanics of planning components, the underlying data structure, the semantic layer of the repository, and the incorporation of the acceleration model and the evaluation procedure. Moreover, it contains additional experimental results and a more precise visualization of the resulting machining strategies.

The work done by other researchers was not included in this thesis unless it was necessary for understanding the experimental setup. In the case of tool parameterization, a remark in the corresponding section indicates the work of other researchers.

Appendix C

Inhabitant Trees for the Machining Experiments in Section 5.7

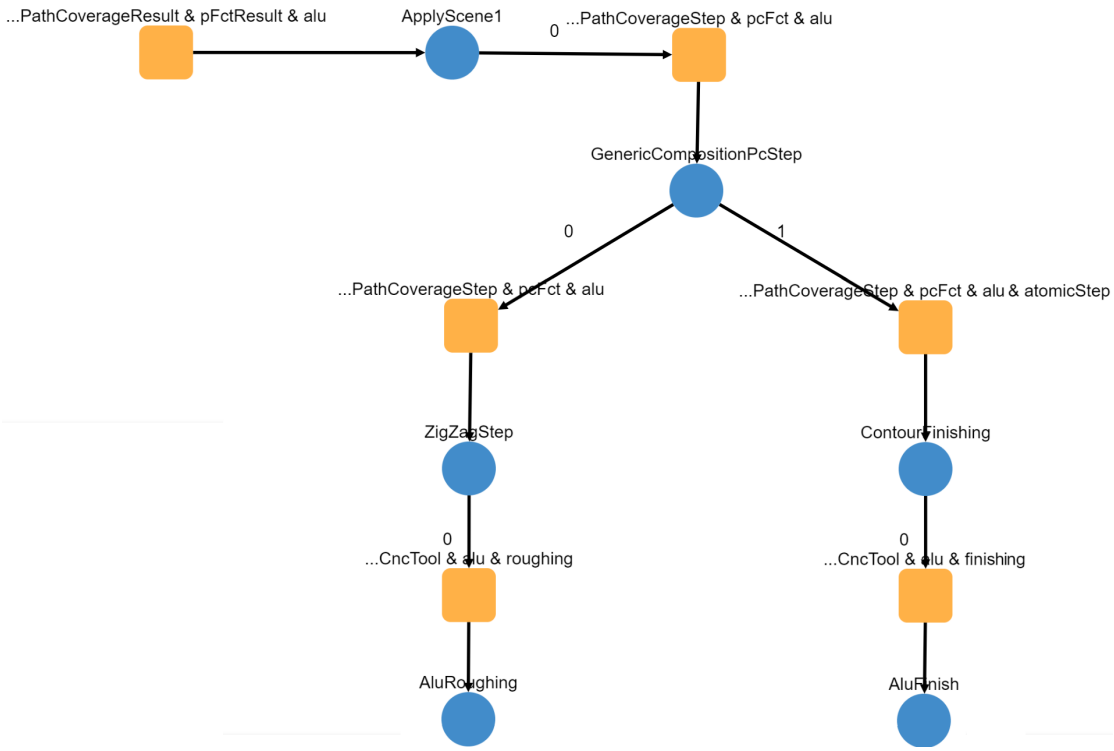


Figure C.1: Inhabitant tree structure for solution candidate 1.1

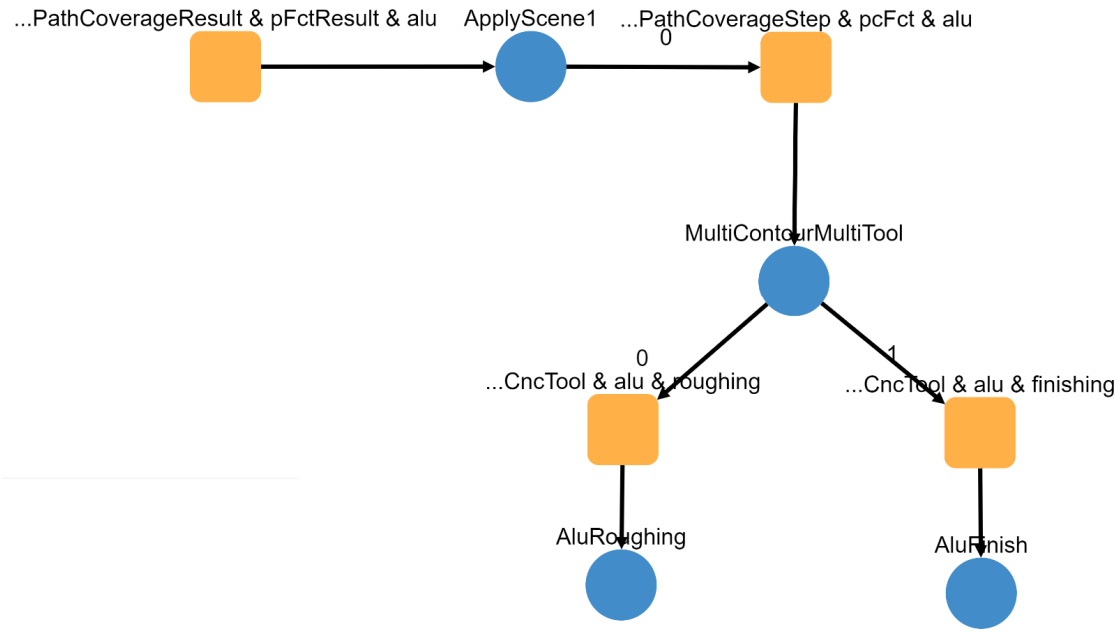


Figure C.2: Inhabitant tree structure for solution candidate 1.2

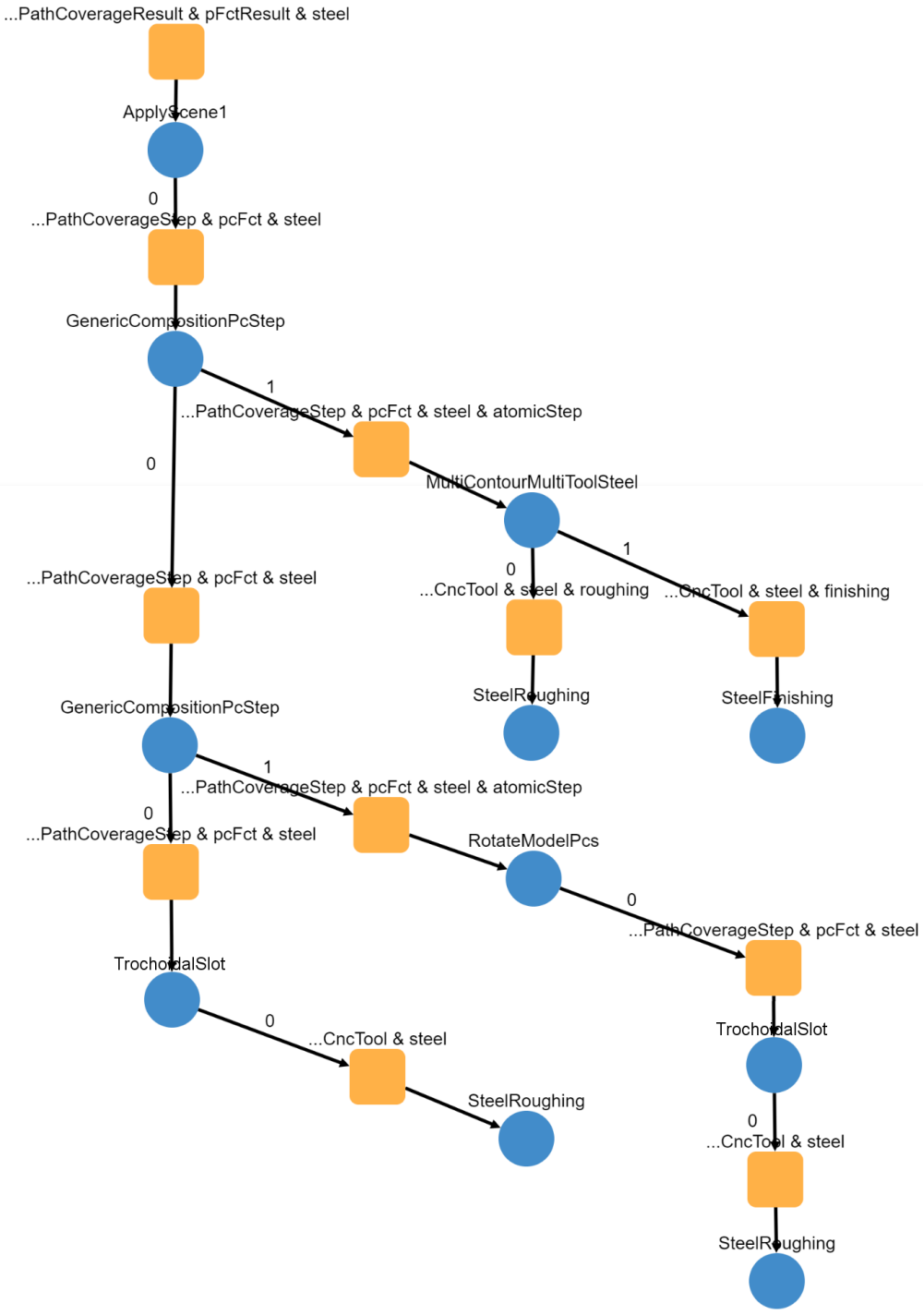


Figure C.3: Inhabitant tree structure for solution candidate 1.3

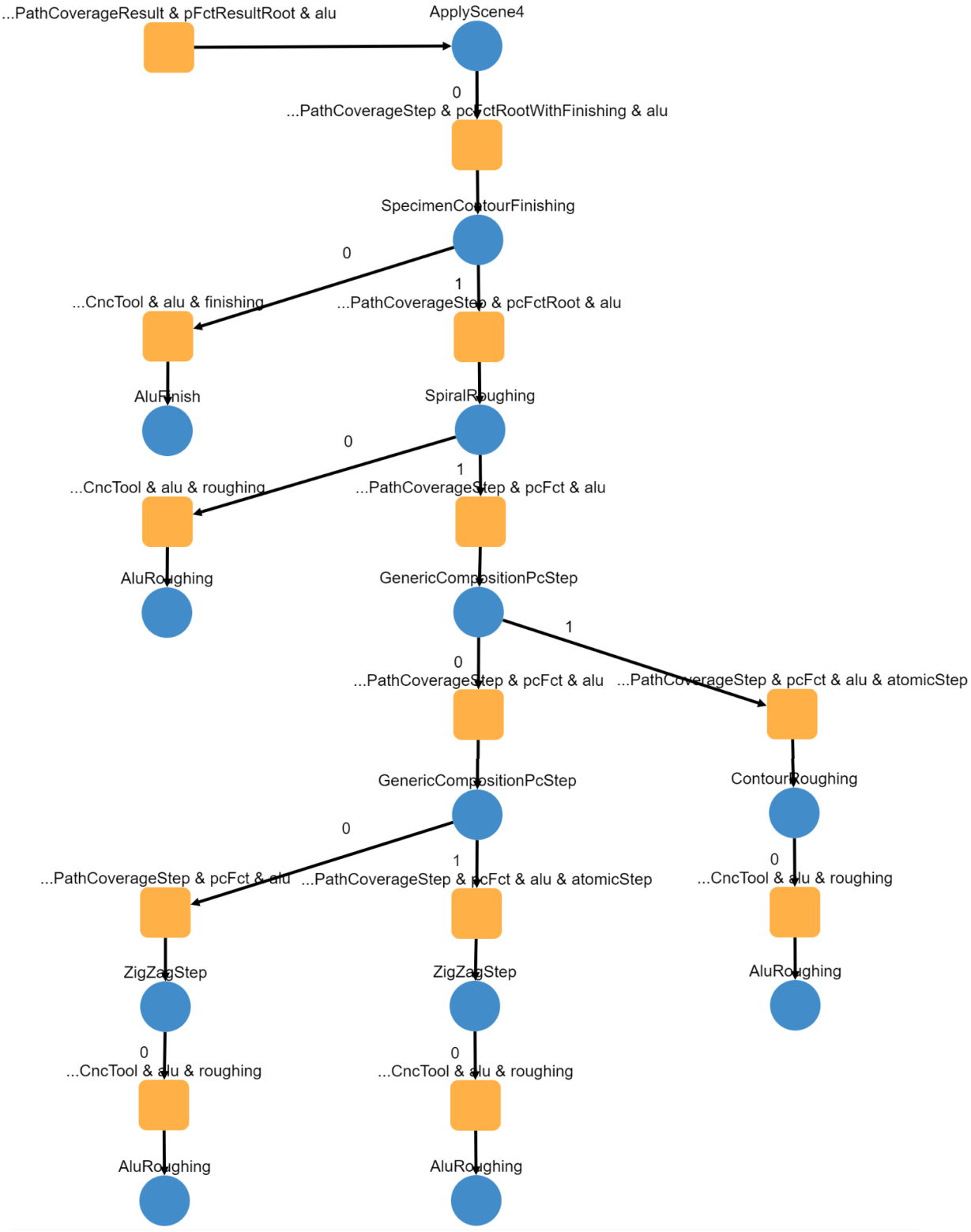


Figure C.4: Inhabitant tree structure for solution candidate 2.1

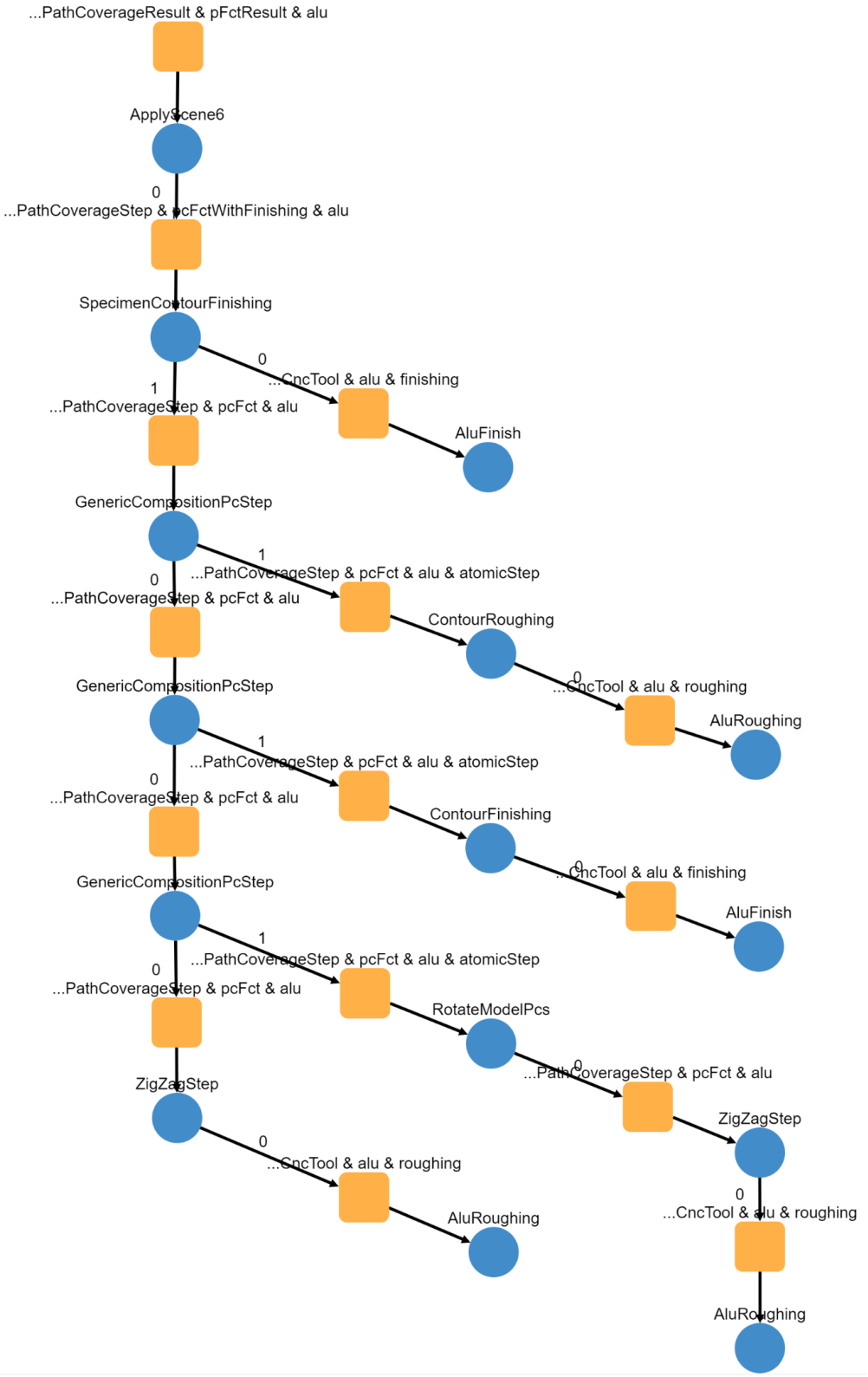


Figure C.5: Inhabitant tree structure for solution candidate 3.1

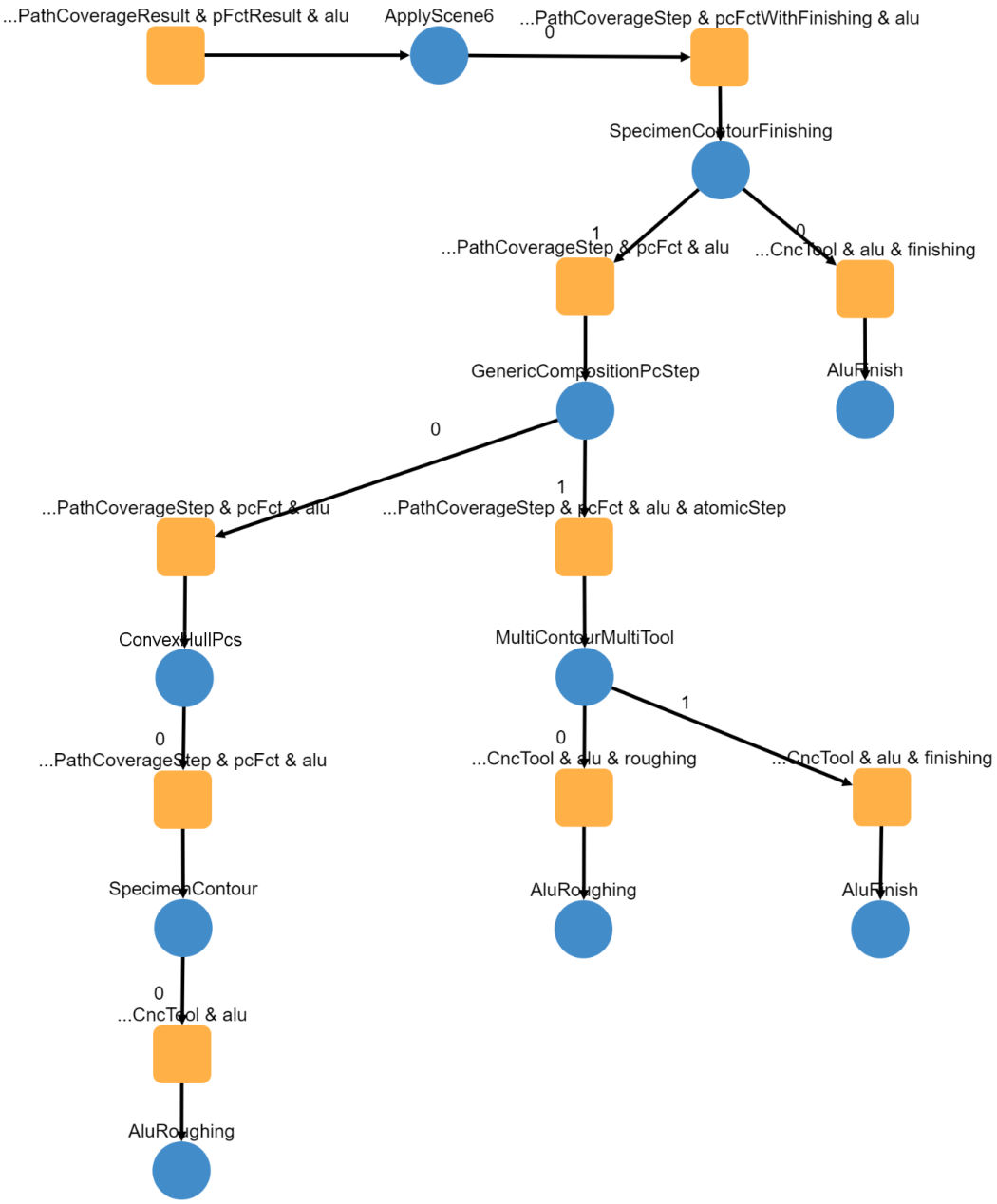


Figure C.6: Inhabitant tree structure for solution candidate 3.2

Appendix D

Planners and Samplers used in Chapter 6

D.1 List of Planners

- Probabilistic Roadmap Method (PRM)
- Probabilistic Roadmap Star (PRMStar)
- Lazy Probabilistic RoadMap Planner (LazyPRM)
- Lazy Probabilistic RoadMap Star Planner (LazyPRMStar)
- Rapidly-exploring Random Trees (RRT)
- Optimal Rapidly-exploring Random Trees (RRTStar)
- Lazy RRT (LazyRRT)
- Lower Bound Tree RRT (LBTRRT)
- Transition-based RRT (TRRT)
- RRT-Connect (RRTConnect)
- Sparse Stable RRT (SST)
- Expansive Space Trees (EST)
- Single-query Bi-directional Lazy collision checking planner (SBL)
- Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE1)
- Lazy Bi-directional KPIECE (LBKPIECE1)

- Bi-directional KPIECE (BKPIECE1)
- Search Tree with Resolution Independent Density Estimation (STRIDE)
- Path-Directed Subdivision Trees (PDST)
- Fast Marching Tree Algorithm (FMT)
- Bidirectional Fast Marching Tree Algorithm (BFMT)
- Optimal Rapidly-exploring Random Trees Maintaining An Optimal Tree (RRTsharp)
- Optimal Rapidly-exploring Random Trees Maintaining A Pseudo Optimal Tree (RRTXstatic)
- Informed RRTstar (InformedRRTstar)
- Batch Informed Trees (BITstar)

D.2 List of Samplers

- Uniform Valid State Sampler
- Obstacle-based Valid State Sampler
- Gaussian Valid State Sampler
- Maximum Clearance Valid State Sampler
- Uniform Space Sampler
- Gaussian Space Sampler

Bibliography

[1] H. Abdullah, R. Ramli, and D. A. Wahab. Tool path length optimisation of contour parallel milling based on modified ant colony optimisation. *The International Journal of Advanced Manufacturing Technology*, 92(1-4):1263–1276, 2017. ISSN 0268-3768. doi: 10.1007/s00170-017-0193-5.

[2] B. Afsharizand, A. Nassehi, V. Dhokia, and S. T. Newman. Formal Modelling of Process Planning in Combined Additive and Subtractive Manufacturing. In *Enabling Manufacturing Competitiveness and Economic Sustainability*, pages 171–176. Springer, 2014. doi: 10.1007/978-3-319-02054-9_29.

[3] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013. doi: 10.1109/FMCAD.2013.6679385.

[4] N. M. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning. In *Proceedings 1996 IEEE International Conference on Robotics and Automation*, pages 113–120, Piscataway, April 1996. IEEE. ISBN 0-7803-2988-0. doi: 10.1109/ROBOT.1996.503582.

[5] N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones, and D. Vallejo. OBPRM: An Obstacle-Based PRM for 3D Workspaces. In *Robotics: The Algorithmic Perspective*, pages 165–178. A K Peters/CRC Press, 1998. doi: 10.1201/9781439863886-19.

[6] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. Finding Tours in the TSP. 1999.

[7] A. Aramcharoen and P. T. Mativenga. Critical factors in energy demand modelling for CNC milling and impact of toolpath strategy. *Journal of Cleaner Production*, 78:63–74, 2014. ISSN 0959-6526. doi: 10.1016/j.jclepro.2014.04.065.

[8] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. DeepCoder: Learning to Write Programs. In *Proceedings of ICLR'17*, 2017. URL https://www.microsoft.com/en-us/research/publication/deepcoder-learning-write-programs/.

[9] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.

[10] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[11] J. Bessai. *A type-theoretic framework for software component synthesis*. PhD thesis, Technical University of Dortmund, Germany, 2019.

[12] J. Bessai and A. Vasileva. User Support for the Combinator Logic Synthesizer Framework. In P. Masci, R. Monahan, and V. Prevosto, editors, *Proceedings 4th Workshop on Formal Integrated Development Environment, F-IDE@FLoC 2018, Oxford, England, 14 July 2018*, volume 284 of *EPTCS*, pages 16–25, 2018. doi: 10.4204/EPTCS.284.2.

[13] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. Combinatory Logic Synthesizer. In *ISoLA (1)*, volume 8802 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2014. doi: 10.1007/978-3-662-45234-9_3.

[14] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. Combinatory Process Synthesis. In *ISoLA (1)*, volume 9952 of *Lecture Notes in Computer Science*, pages 266–281, 2016. doi: 10.1007/978-3-319-47166-2_19.

[15] J. Bessai, M. Roidl, and A. Vasileva. Experience Report: Towards Moving Things with Types - Helping Logistics Domain Experts to Control Cyber-Physical Systems with Type-Based Synthesis. *Electronic Proceedings in Theoretical Computer Science*, 310(1):1–6, 2019. doi: 10.4204/EPTCS.310.1.

[16] M. B. Bieterman and D. R. Sandstrom. A Curvilinear Tool-Path Method for Pocket Machining. *Journal of Manufacturing Science and Engineering*, 125(4):709–715, 2003. ISSN 1087-1357. doi: 10.1115/1.1596579.

[17] R. Bohlin and L. E. Kavraki. Path planning using lazy PRM. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, pages 521–528 vol.1. IEEE, 24.04.2000 - 28.04.2000. ISBN 0-7803-5886-4. doi: 10.1109/ROBOT.2000.844107.

[18] V. Boor, M. H. Overmars, and A. F. van der Stappen. The Gaussian sampling strategy for probabilistic roadmap planners. In *1999 IEEE International Conference on Robotics and Automation*, pages 1018–1023, Piscataway, May 1999. I E E E. ISBN 0-7803-5180-0. doi: 10.1109/ROBOT.1999.772447.

[19] M. S. Branicky, S. M. LaValle, K. Olson, and L. Yang. Quasi-randomized path planning. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation*

(Cat. No.01CH37164), pages 1481–1487. IEEE, 21-26 May 2001. ISBN 0-7803-6576-3. doi: 10.1109/ROBOT.2001.932820.

[20] O. Brock and O. Khatib. Elastic strips: A framework for motion generation in human environments. *The International Journal of Robotics Research*, 21(12):1031–1052, 2002.

[21] T. Childs, K. Maekawa, and P. Maulik. Effects of coolant on temperature distribution in metal machining. *Materials science and technology*, 4(11):1006–1019, 1988.

[22] H. M. Choset, S. Hutchinson, K. M. Lynch, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.

[23] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen, et al. *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer, Boston, MA, 2007. doi: 10.1007/978-0-387-36797-2.

[24] M. Coppo, M. Dezani-Ciancaglini, et al. An extension of the basic functionality theory for the λ-calculus. *Notre Dame journal of formal logic*, 21(4):685–693, 1980.

[25] K. A. Desai and P. V. M. Rao. Machining of curved geometries with constant engagement tool paths. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 230(1):53–65, 2016. ISSN 0954-4054. doi: 10.1177/0954405415616787.

[26] J. Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15-16):1–35, 2006.

[27] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. ISSN 0945-3245. doi: 10.1007/BF01386390.

[28] B. Düdder, M. Martens, J. Rehof, and P. Urzyczyn. Bounded combinatory logic. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 16. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[29] A. Dumitrache, T. Borangiu, and A. Dogar. Automatic Generation of Milling Toolpaths with Tool Engagement Control for Complex Part Geometry. *IFAC Proceedings Volumes*, 43(4):252–257, 2010. ISSN 14746670. doi: 10.3182/20100701-2-PT-4011.00044.

[30] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018. ISSN 0362-1340. doi: 10.1145/3296979.3192382.

[31] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962. ISSN 0001-0782. doi: 10.1145/367766.368168.

[32] N. A. Fountas, N. M. Vaxevanidis, C. I. Stergiou, and R. Benhadj-Djilali. Development of a software-automated intelligent sculptured surface machining optimization environment. *The International Journal of Advanced Manufacturing Technology*, 75(5-8): 909–931, 2014. doi: 10.1007/s00170-014-6136-5.

[33] J. Frankle, P. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, pages 802–815. ACM, 2016. doi: 10.1145/2837614.2837629.

[34] D. Freiburg, F. Finkeldey, M. Hensel, P. Wiederkehr, and D. Biermann. Simulation of surface structuring considering the acceleration behaviour by means of spindle control. *International Journal of Mechatronics and Manufacturing Systems*, 11(1):67–86, 2018. doi: 10.1504/IJMMS.2018.091178.

[35] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, and D. Tarlow. TerpreT: A Probabilistic Programming Language for Program Induction. 2016. URL https://www.microsoft.com/en-us/research/publication/terpret-probabilistic-programming-language-program-induction/.

[36] R. Geraerts and M. H. Overmars. Creating High-quality Paths for Motion Planning. *The international journal of robotics research*, 26(8):845–863, 2007. doi: 10.1177/0278364907079280.

[37] R. Geraerts and M. H. Overmars. Clearance based path optimization for motion planning. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, pages 2386–2392 Vol.3. IEEE, 26.04.2004 - 01.05.2004. ISBN 0-7803-8232-3. doi: 10.1109/ROBOT.2004.1307418.

[38] S. Gulwani. Dimensions in program synthesis. In T. Kutsia, W. Schreiner, and M. Fernández, editors, *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming - PPDP '10*, pages 13–24, New York, New York, USA, 2010. ACM Press. ISBN 9781450301329. doi: 10.1145/1836089.1836091.

[39] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices*, 46(1):317–330, 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926423.

[40] S. Gulwani, O. Polozov, and R. Singh. Program Synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010.

[41] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2020. doi: 10.1145/3371080.

[42] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107, 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

[43] S. G. Hoggar. *Mathematics of Digital Images: Creation, Compression, Restoration, Recognition*. Cambridge university press, 2006. ISBN 9781139451352.

[44] H.-T. Hsieh and C.-H. Chu. Improving optimization of tool path planning in 5-axis flank milling using advanced PSO algorithms. *Robotics and Computer-Integrated Manufacturing*, 29(3):3–11, 2013. ISSN 07365845. doi: 10.1016/j.rcim.2012.04.007.

[45] D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *Proceedings of International Conference on Robotics and Automation*, volume 3, pages 2719–2726. IEEE, 1997. doi: 10.1109/robot.1997.619371.

[46] D. Hsu, J.-C. Latombe, and H. Kurniawati. On the Probabilistic Foundations of Probabilistic Roadmap Planning. *The international journal of robotics research*, 25(7):627–643, 2006. doi: 10.1177/0278364906067174.

[47] R. Huang, S. Zhang, X. Bai, and C. Xu. Multi-level structuralized model-based definition model based on machining features for manufacturing reuse of mechanical parts. *The International Journal of Advanced Manufacturing Technology*, 75(5-8):1035–1048, 2014. ISSN 0268-3768. doi: 10.1007/s00170-014-6183-y.

[48] S. Ibaraki, I. Yamaji, and A. Matsubara. On the removal of critical cutting regions by trochoidal grooving. *Precision Engineering*, 34(3):467–473, 2010. ISSN 01416359. doi: 10.1016/j.precisioneng.2010.01.007.

[49] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning under Noisy I/O. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 990–998. PMLR, 2017.

[50] L. Jaillet, J. Cortés, and T. Siméon. Sampling-Based Path Planning on Configuration-Space Costmaps. *IEEE Transactions on Robotics*, 26(4):635–646, 2010. ISSN 1552-3098.

[51] P. Jamshidi, J. Camara, B. Schmerl, C. Kaestner, and D. Garlan. Machine Learning Meets Quantitative Planning: Enabling Self-Adaptation in Autonomous Robots. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 39–50. IEEE, 25.05.2019 - 25.05.2019. ISBN 978-1-7281-3368-3. doi: 10.1109/SEAMS.2019.00015.

[52] F. Kallat, C. Mieth, J. Rehof, and A. Meyer. Using Component-based Software Synthesis and Constraint Solving to generate Sets of Manufacturing Simulation Models. *Procedia CIRP*, 93(1):556–561, 2020. ISSN 22128271. doi: 10.1016/j.procir.2020.03.018.

[53] F. Kallat, J. Pfrommer, J. Bessai, J. Rehof, and A. Meyer. Automatic Building of a Repository for Component-based Synthesis of Warehouse Simulation Models. *Procedia CIRP*, 104: 1440–1445, 2021. ISSN 2212-8271. doi: 10.1016/j.procir.2021.11.243. 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0.

[54] A. Kalyan, A. Mohta, A. Polozov, D. Batra, P. Jain, and S. Gulwani. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations (ICLR)*, 2018. URL https://www.microsoft.com/en-us/research/publication/neural-guided-deductive-search-real-time-program-synthesis-examples/.

[55] L. E. Kavraki, P. Svestka, J. . Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996. doi: 10.1109/70.508439.

[56] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996. ISSN 1042296X. doi: 10.1109/70.508439.

[57] P. Kersting and A. Zabel. Optimizing NC-tool paths for simultaneous five-axis milling based on multi-population multi-objective evolutionary algorithms. *Advances in Engineering Software*, 40(6):452–463, 2009. ISSN 09659978. doi: 10.1016/j.advengsoft.2008.04.013.

[58] F. Klocke and W. König. *Fertigungsverfahren 1: Drehen, Fräsen, Bohren*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-23458-6. doi: 10.1007/978-3-540-35834-3.

[59] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-guided program synthesis. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–268, New York, NY, USA, 06082019. ACM. ISBN 9781450367127. doi: 10.1145/3314221.3314602.

[60] J.-C. Latombe. *Robot Motion Planning*. Springer Science & Business Media, 2012. ISBN 9781461540229.

[61] S. M. LaValle. *Planning algorithms*. Cambridge university press, 2006.

[62] S. M. LaValle et al. Rapidly-exploring random trees: A new tool for path planning. 1998. URL https://www.cs.csustan.edu/~xliang/courses/cs4710-21s/papers/06%20rrt.pdf.

[63] I. Lazoglu, C. Manav, and Y. Murtezaoglu. Tool path optimization for free form surface machining. *CIRP Annals*, 58(1):101–104, 2009. ISSN 0007-8506. doi: 10.1016/j.cirp.2009.03.054.

[64] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pages 1008–1014. IEEE Comput. Soc. Press, 9-11 April 1991. ISBN 0-8186-2163-X. doi: 10.1109/ROBOT.1991.131723.

[65] Y. Lu, Y. Ding, and L. Zhu. Smooth Tool Path Optimization for Flank Milling Based on the Gradient-Based Differential Evolution Method. *Journal of Manufacturing Science and Engineering*, 138(8):887, 2016. ISSN 1087-1357. doi: 10.1115/1.4032969.

[66] V. J. Lumelsky and T. Skewis. Incorporating range sensing in the robot navigation function. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(5):1058–1069, 1990. ISSN 00189472. doi: 10.1109/21.59969.

[67] V. J. Lumelsky and A. A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2(1-4):403–430, 1987. doi: 10.1007/BF01840369.

[68] B. Mirtich. V-Clip: fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, 1998. ISSN 0730-0301. doi: 10.1145/285857.285860.

[69] M. Morales, L. Tapia, R. Pearce, S. Rodriguez, and N. M. Amato. A Machine Learning Approach for Feature-Sensitive Motion Planning. In M. Erdmann, M. Overmars, D. Hsu, and F. van der Stappen, editors, *Algorithmic Foundations of Robotics VI*, volume 17 of *Springer Tracts in Advanced Robotics*, pages 361–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-25728-8. doi: 10.1007/10991541{\textunderscore}25.

[70] D. Mourtzis. Simulation in the design and operation of manufacturing systems: state of the art and new trends. *International Journal of Production Research*, 58(7):1927–1949, 2020. doi: 10.1080/00207543.2019.1636321.

[71] L. Nardi, D. Koeplinger, and K. Olukotun. Practical Design Space Exploration. In *MASCOTS*, pages 347–358. IEEE Computer Society, 2019. doi: 10.1109/MASCOTS.2019.00045.

[72] A. Nassehi, S. Newman, V. Dhokia, Z. Zhu, and R. I. Asrai. Using formal methods to model hybrid manufacturing processes. In *Enabling Manufacturing Competitiveness and Economic Sustainability*, pages 52–56. Springer, 2012. doi: 10.1007/978-3-642-23860-4_8.

[73] S. T. Newman, A. Nassehi, R. Imani-Asrai, and V. Dhokia. Energy efficient process planning for CNC machining. *CIRP Journal of Manufacturing Science and Technology*, 5 (2):127–136, 2012. ISSN 17555817. doi: 10.1016/j.cirpj.2012.03.007.

[74] J. Pan, S. Chitta, and D. Manocha. FCL: A general purpose library for collision and proximity queries. In *ICRA*, pages 3859–3866. IEEE, 2012. doi: 10.1109/ICRA.2012.6225337.

[75] E. Paucksch, S. Holsten, M. Linß, and F. Tikal. *Zerspantechnik: Prozesse, Werkzeuge, Technologien*. Springer-Verlag, 2008. ISBN 9783834802798.

[76] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, pages 522–538. ACM, 2016. doi: 10.1145/2908080.2908093.

[77] N. Polikarpova, D. Stefan, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama. Liquid information flow control. *Proceedings of the ACM on Programming Languages*, 4(ICFP): 1–30, 2020. doi: 10.1145/3408987.

[78] O. Polozov and S. Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 107–126, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336895. doi: 10.1145/2814270.2814310.

[79] M. Raghothaman, A. Reynolds, and A. Udupa. The SyGuS Language Standard Version 2.0, 2019. URL https://sygus.org/assets/pdf/SyGuS-IF_2.0.pdf.

[80] J. Rehof. Towards Combinatory Logic Synthesis. In *1st International Workshop on Behavioural Types, BEAT*, 2013.

[81] J. Rehof and P. Urzyczyn. Finite Combinatory Logic with Intersection Types. In *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011. doi: 10.1007/978-3-642-21691-6_15.

[82] J. Rehof and M. Y. Vardi. Design and Synthesis from Components (Dagstuhl Seminar 14232). *Dagstuhl Reports*, 4(6):29–47, 2014. doi: 10.4230/DagRep.4.6.29.

[83] S. Saeedi, L. Nardi, E. Johns, B. Bodin, P. H. J. Kelly, and A. J. Davison. Application-oriented design space exploration for SLAM algorithms. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5716–5723. IEEE, 2017. doi: 10.1109/ICRA.2017.7989673.

[84] G. Sánchez-Ante and J. Latombe. A Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking. In *Robotics Research*, volume 6 of *Springer Tracts in Advanced Robotics*, pages 403–417, 2001.

[85] T. Schäfer, F. Möller, A. Burmann, Y. Pikus, N. Weißenberg, M. Hintze, and J. Rehof. A Methodology for Combinatory Process Synthesis: Process Variability in Clinical Pathways. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, volume 11247 of *Lecture Notes in Computer Science*, pages 472–486. Springer International Publishing, Cham, 2018. ISBN 978-3-030-03426-9. doi: 10.1007/978-3-030-03427-6{\textunderscore}35.

[86] T. Schäfer, J. A. Bergmann, R. G. Carballo, J. Rehof, and P. Wiederkehr. A Synthesis-based Tool Path Planning Approach for Machining Operations. *Procedia CIRP*, 104:918–923, 2021. doi: 10.1016/j.procir.2021.11.154. 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0.

[87] M. C. Shaw and J. Cookson. *Metal cutting principles*, volume 2. Oxford university press New York, 2005.

[88] R. Singh. BlinkFill. *Proceedings of the VLDB Endowment*, 9(10):816–827, 2016. ISSN 2150-8097. doi: 10.14778/2977797.2977807.

[89] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. R. Martonosi, editors, *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems - ASPLOS-XII*, page 404, New York, New York, USA, 2006. ACM Press. ISBN 1595934510. doi: 10.1145/1168857.1168907.

[90] A. Stentz. Optimal and Efficient Path Planning for Partially Known Environments. In M. H. Hebert, C. Thorpe, and A. Stentz, editors, *Intelligent Unmanned Ground Vehicles*, pages 203–220. Springer US, Boston, MA, 1997. ISBN 978-1-4613-7904-1. doi: 10.1007/978-1-4615-6325-9{\textunderscore}11.

[91] I. A. Şucan, M. Moll, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. doi: 10.1109/MRA.2012.2205651. https://ompl.kavrakilab.org.

[92] L. P. Swiler, P. D. Hough, P. Qian, X. Xu, C. Storlie, and H. Lee. Surrogate models for mixed discrete-continuous variables. In *Constraint Programming and Decision Making*, pages 181–202. Springer, 2014. doi: 10.1007/978-3-319-04280-0{\textunderscore}21.

[93] V. Tandon, H. El-Mounayri, and H. Kishawy. NC end milling optimization using evolutionary computation. *International Journal of Machine Tools and Manufacture*, 42(5): 595–605, 2002. doi: 10.1016/S0890-6955(01)00151-1.

[94] M. S. Uddin, S. Ibaraki, A. Matsubara, S. Nishida, and Y. Kakino. A Tool Path Modification Approach to Cutting Engagement Regulation for the Improvement of Machining Accu-

racy in 2D Milling With a Straight End Mill. *Computer-Aided Design*, 129(6):1069–1079, 2007. ISSN 00104485. doi: 10.1115/1.2752526.

[95] P. Urzyczyn. The emptiness problem for intersection types. *The Journal of Symbolic Logic*, 64(3):1195–1215, 1999.

[96] P. Urzyczyn. Inhabitation of Low-Rank Intersection Types. In *TLCA*, volume 5608, pages 356–370. Springer, 2009.

[97] X. Wang, I. Dillig, and R. Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2018. doi: 10.1145/3158151.

[98] S. Wenzel, J. Stolipin, J. Rehof, and J. Winkels. Trends in Automatic Composition of Structures for Simulation Models in Production and Logistics. In *2019 Winter Simulation Conference (WSC)*, pages 2190–2200. IEEE, 08.12.2019 - 11.12.2019. ISBN 978-1-7281-3283-9. doi: 10.1109/WSC40007.2019.9004959.

[99] S. A. Wilmarth, N. M. Amato, and P. F. Stiller. MAPRM: a probabilistic roadmap planner with sampling on the medial axis of the free space. In *1999 IEEE International Conference on Robotics and Automation*, pages 1024–1031, Piscataway, May 1999. I E E E. ISBN 0-7803-5180-0. doi: 10.1109/ROBOT.1999.772448.

[100] J. Winkels. *Automatisierte Komposition und Konfiguration von Workflows zur Planung mittels kombinatorischer Logik*. PhD thesis, Technical University of Dortmund, Germany, 2019.

[101] J. Winkels, J. Graefenstein, T. Schäfer, D. Scholz, J. Rehof, and M. Henke. Automatic Composition of Rough Solution Possibilities in the Target Planning of Factory Planning Projects by Means of Combinatory Logic. In *ISoLA (4)*, volume 11247 of *Lecture Notes in Computer Science*, pages 487–503. Springer, 2018. doi: 10.1007/978-3-030-03427-6_36.

[102] X. Xu and S. Hinduja. Determination of finishing features in 2½ D components. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 211(2):125–142, 1997. ISSN 0954-4054. doi: 10.1243/0954405971516121.

[103] J. Yan and L. Li. Multi-objective optimization of milling parameters – the trade-offs between energy, production rate and cutting quality. *Journal of Cleaner Production*, 52 (1):462–471, 2013. ISSN 0959-6526. doi: 10.1016/j.jclepro.2013.02.030.

[104] W. Zeng and R. L. Church. Finding shortest paths on real road networks: the case for A*. *International Journal of Geographical Information Science*, 23(4):531–543, 2009. ISSN 1365-8816. doi: 10.1080/13658810801949850.