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

Tristan Schäfer, Jan Bessai, Constantin Chaumet, Jakob Rehof, and Christian Riest

TU Dortmund University, 44227 Dortmund, Germany
{tristan.schaefer, jan.bessai, constantin.chaumet, jakob.rehof, christian.riest}@tu-dortmund.de

Abstract. Motion Planning is widely acknowledged as a fundamental problem of robotics. Due to the continuous efforts of the scientific community, various algorithmic families emerged that have different strengths and weaknesses. Finding a suitable motion planning program is often not trivial for real-world problems, as various domain-specific factors must be considered. An obvious example is a potential trade-off between path length, computation time, and resource constraints. We propose a technique to systematically explore the space of suitable programs, aiming to find Pareto optimal algorithm configurations. Our approach makes use of Combinatory Logic Synthesis to perform component-based software composition. Software components are injected with domain-knowledge, effectively restricting the solution space of synthesizable programs. We synthesize sample-based global planning programs that make use of the Open Motion Planning Library (OMPL) and evaluate the produced programs to yield numeric result vectors. These steps are encapsulated in a black-box function which is used with a multi-objective optimization tool (Hypermapper) to yield an automatic, learning-based search procedure for a given feature space. We validate our approach with a series of experiments that demonstrate the extensibility and transferability of our methodology regarding different robotic systems and planning instances.

1 Introduction

Motion planning aims to find a continuous path for a movable object from a starting configuration state to a goal state while avoiding obstacles. The general piano movers problem was proven to be PSPACE-hard [27], and additional requirements of real-life applications and the high-dimensional configuration space of the moving object (in the following assumed to be a robot) form the need for various heuristics.

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). This is relevant for solving challenging problems in practical applications with high-dimensional planning domains. In this work, we perform a

search in the space of global sampling-based motion planning programs with path post-processing. The domains of interest include geometric planning for rigid bodies in the special Euclidean group $SE(3)$ and planning in real vector spaces representing states of robotic arms. The considered family of planning algorithms performs searches 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 are often asymptotically optimal as the solution path converges to a global optimum with a growing number of iterations. In practice, limited computation time prevents optimal planning results. However, these results can be used as an initial solution that is supplemented by local planning techniques to smooth and simplify paths [8] or account for dynamic changes in the environment [6]. Real-life applications often introduce the need for specialized planners with specific characteristics, such as their suitability for narrow passages [16] or real-time optimal motion planning capabilities in dynamic environments [32]. Consequently, the variety of planners grows continuously, and finding a suitable planning approach for a given problem can be challenging.

Thus, we introduce a novel automated configuration approach for planning programs by using Combinatory Logic Synthesis (CLS) and subject the space of configurable programs to an exploration procedure. The main contribution of the present paper is twofold. First, we apply component-based synthesis to sampling-based motion planning. Second, we incorporate an optimization procedure by embedding CLS into an active learning loop.

The strength of the CLS approach lies in its ability to automate the assembly of multiple variants of complete programs by composing existing software artifacts (referred to as combinators in CLS). In software engineering terms, we automatically generate members of a product line [12] from repositories (libraries) of components. In our application to motion planning we are able to automatically generate families of planner programs using the Open Motion Planning Library (OMPL [31]) to form a set of combinators that represents building blocks for planners, samplers, and further code artifacts that are required for setting up an executable planning program. Our planner combinators implement code templates which are used by CLS to generate meaningful Python code. The family of generated programs can be extended by adding new combinators to the repository without any need to change existing parts of the system. The automated generation of programs with CLS incorporates a modular and flexible way of specifying the space of valid program variations, which captures a user-defined design space of motion planning programs. A valid variant complies with the rules of composition defined by the user. In sampling-based motion planning, these rules might consider admissible combinations of planning algorithms and sampling strategies. Moreover, the algorithmic variations can involve problem-specific adaptions. For instance, some planning problems require specialized techniques to find valid samples in the configuration space. Thus, they mandate the use of components implementing a suitable technique.

The repository forms an algorithmic space that enables the use of optimization procedures to find valid structural variances instead of performing a search procedure on numerical parameters. For this purpose, we define a feature vector that represents points in the feature space of sampling-based motion planning programs. We use CLS to resolve points of this space and generate programs that comply with the given algorithm configuration. We consider motion planning as a multi-objective optimization problem subject to an optimization procedure. This approach requires an optimization technique that is capable of black-box optimization and supports categorical variables. For these reasons, we chose the optimization tool Hypermapper [22] to perform experiments for single query planning tasks for specific problem instances. Our approach finds a Pareto front that can be analyzed to reason about the suitability of algorithmic families for a given problem instance. The approach might be a step towards an evaluation platform of use to developers and researchers. We hope that the observation of performance characteristics compared to existing artifacts can support further development of planners, samplers, or collision detection techniques.

2 Related Work

2.1 Type-Directed Synthesis

Software synthesis – producing a program from a high-level specification – is an active field of study in computer science [10]. In type-directed synthesis, types are used for the high-level description of programs and for guiding the synthesis process. However, this technique is limited to small functional programs and, to the best of our knowledge, has not been used to solve non-trivial motion planning problems. In our novel approach, we apply the CLS framework [24, 5] which is type-directed but also component-oriented [26]. While many synthesis approaches require synthesizing software from scratch (i.e., producing the entire code without any human intervention), component-oriented techniques use pre-existing components as building blocks. Components may be manually engineered, e.g., to be readable and efficient, and components can encapsulate logic that is otherwise infeasible to formally specify. Semantic type expressions extend standard program types to expose what components do instead of describing how they do it. The CLS framework, which will be described in more detail in Section 3.3, can handle a broad set of application domains ranging from small-scale software composition [2], large-scale software product lines [3, 11, 4], to schedule-oriented planning and factory planning [9, 33]. Recent studies demonstrate the applicability of CLS for engineering disciplines such as warehouse simulation modeling [19] or tool path planning for machining operations [30]. While being a type-directed framework, the programming language of CLS (Scala) does not limit the programming language usable in target artifacts. Instead, CLS enables components to act as code generators of arbitrary languages, which is crucial for the generated code (in our case, Python using C++ bindings) to access comprehensive motion planning libraries. It does not apply a synthesis

algorithm to find a plan but instead composes families of specialized algorithmic configurations to solve the problem.

2.2 Learning and Design Space Exploration

The challenge of having a high-dimensional solution space for a given problem is widespread. As a result, a standard method is to do a design space exploration to find suitable solutions. Because it is expensive to do a full exploration, approaches like machine learning are used to find suitable solutions [35, 7]. For example, Ipek et al. trained an artificial neural network with a small sample of design space points as training data to create an approximation of the design space to predict other points' results with high accuracy [17]. Another approach by Hosny et al. uses reinforcement learning to generate a flow of logic synthesis optimizations from a set of transformation algorithms without human interactions [14]. In the motion planning domain, Morales et al. use machine learning to find regionally optimal algorithms, based on spatially varying features of a given environment [21]. Saeedi et al. use design space exploration to select from four different design spaces to find suitable SLAM algorithms [28]. In contrast, we use a combination of machine learning techniques to select from an algorithmic feature space of software components and synthesis techniques to generate suitable programs. Xiang et al. provide a novel approach to sample algorithm configurations for large software product lines [34]. They use SAT solvers to find an initial set of valid sample points (i.e., algorithm configurations) and a search procedure that uses a distance metric to build a uniform sampling set incrementally. The resulting samples are a reduced representation of the software product line, enabling the application of techniques such as optimization procedures or performance prediction. Jamshidi et al. proposed a concept to explore the space for self-adapting robot configurations [18]. A machine learning procedure determines potential Pareto optimal configurations regarding localization error and estimates for energy consumption. In contrast, our approach examines algorithmic spaces as CLS can describe large design spaces and synthesize runnable programs. Moreover, we study the application of a black-box optimization procedure to the domain of sampling-based motion planning programs, which has not been examined before.

3 Preliminaries

3.1 Multi-Objective Optimization Problem

The multi-objective optimization problem (MOP) is an optimization problem to find optimal solutions regarding multiple (possibly conflicting) optimization criteria. The problem can be formalized as finding x to minimize

$$f(x) = (f_1(x), \ldots, f_m(x)) \tag{1}$$

where x is a point from the feature space $\mathbb{X}$ and $f : \mathbb{X} \to \mathbb{R}^m$ is an objective function mapping from the feature space $\mathbb{X}$ to objective space $\mathbb{R}^m$. Each of the m

vector components is a function defining the value for an optimization criterion at a given point in the feature space. Because of the possible conflicts between the criteria, it is unlikely to find only one optimal point. Instead, a MOP requires searching a set of dominating points. A point x_1 dominates another point x_2 written $x_1 \prec x_2$ precisely if

$$\forall i \in 1,\ldots,m : f_i(x_1) \leq f_i(x_2) \quad \text{and} \quad \exists j \in 1,\ldots,m : f_j(x_1) < f_j(x_2) \quad . \tag{2}$$

If any other point does not dominate a point, it is called Pareto optimal. The set of all Pareto optimal points is called Pareto set, and the image of this set under f is called Pareto front. For our study, we defined the following four dimensions for $\mathbb{X}$: (1) planner, (2) sampler, (3) motion validator, and (4) maximal allowed planning time. The planner describes an algorithm, the sampler provides configuration space samples for the given algorithm, and the motion validator checks if a linear motion of the robot involves collisions with the environment. The objective space is defined by the three objectives: (1) averaged solution path length, (2) averaged computation time, and (3) number of failed computation attempts. Planner, sampler, and motion validator are feature dimensions represented by categorical variables. The maximal allowed planning time, on the other hand, is a numerical variable. It is used to specify a termination condition that will cause a planning program to stop the execution after the given amount of time. That way, we can compare optimizing planners with other geometric planners that terminate instantly when a first valid solution is found.

3.2 Black-Box Optimization

It is common to determine the Pareto front for a given objective function by using analytical methods. In our case, the algorithmic feature space of motion planning algorithms lacks derivative information, which prevents using these methods. Black-box optimization (also referred to as derivative-free optimization) is a technique that requires only objective function values. To solve the multi-objective optimization problem, we use HyperMapper [22]. The framework determines the Pareto front for a given multi-objective function by using multiple random forests to learn a model abstracting the MOP function. The model is then used to find the Pareto front by determining an initial front in the warm-up phase. The active learning phase consists of multiple iterations of two steps. First, the model is improved based on the determined Pareto front. If possible, the Pareto front is improved in the second step.

3.3 Combinatory Logic Synthesis

The underlying formalism of CLS is combinatory logic [13] with intersection types [25, 5, 1]. Synthesis in CLS is guided by the three rules of the type system depicted in Equation 3.

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}(\mathsf{Var}) \quad \frac{\Gamma \vdash M : A \quad A \leq B}{\Gamma \vdash M : B}(\leq) \quad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}(\mathsf{MP}) \tag{3}$$

Types are assigned by the ternary relation $\vdash$, where the first place is a collection of combinators, Γ, the second place is the term, which can either be a combinator x or an application of terms MN and the third place is the type to be assigned. The first rule (Var) allows using any component in a repository of combinators Γ, where we write $\Gamma(x) = A$ if the repository holds information that combinator x has type A. The second rule allows casting types from special to general according to a preorder $\leq$, the rules of which coincide with a set-theoretic view of types. That is: types are considered to classify sets of programs, and the preorder $\leq$ on types reflects the inclusion preorder $\subseteq$ on sets. We have $A \leq B$ precisely if the set of programs classified by A is included in the set of programs classified by B. Finally, the rule of modus-ponens (MP) allows applying functions $A \to B$ to arguments of type A, where the result of function application MN will have type B. For these rules, CLS solves the problem of relativized type inhabitation [25]: Given a repository Γ and a target type A, find the terms M (called inhabitants) such that $\Gamma \vdash M : A$. Combinators in Γ can then be interpreted as programs and their application as function calls. Note that this setting does not mention the language in which programs (combinators) are written. This allows CLS to be language agnostic. In practice, programs of CLS are code generators for programs in any other language, in our case Python. Their types consist of two parts. The first part, called native type, is the type of the code generator, which is programmed in Scala. The second part, called semantic type, is an additional domain-specific classification. These two parts are combined by the intersection type operation $A \cap B$. Semantically, the type $A \cap B$ classifies all the programs in the intersection of the sets of programs classified by type A and type B. Since the repository Γ is user-specified and intersections only add further restrictions on the number of programs classified by a type, it is possible to add semantic information freely. This way, the user can express a domain-specific taxonomic structure without adhering to the rigid and more generic rules of native programming language type systems. An example for this will be explained in Section 4.1, where additional semantic types classify sets of programs representing different motion planning algorithms. The algorithmic details are available in the literature [25, 1], and beyond the scope of this paper.

4 Architectural Overview

Fig. 1 displays a schematic overview of the parts used to perform the search for Pareto optimal planning programs. The main parts are the active learning loop and the black-box function designed for this study.

A set of categorical variables is defined based on the dimensions of the feature space, forming the search space for the optimization problem at hand. The active learning loop provided by the Hypermapper guides the traversal of the search space. It uses regression and classification models to select specific samples in the search space, which are transformed to an n-tuple and used as an argument for the black-box function. When executed, this function yields a real vector that contains values for the path length, computation time, and the num-

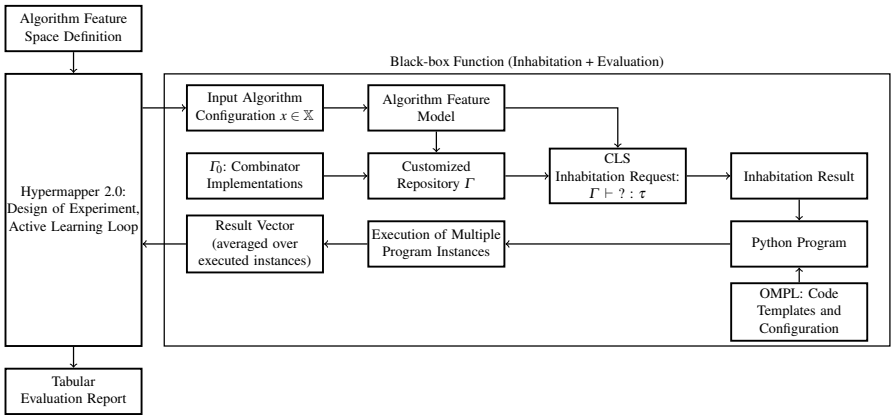

Fig. 1. Overview of the approach: Hypermapper selects samples from the algorithm feature space. Samples are forwarded into the black-box function, which generates Python programs that are executed to obtain evaluation information that is fed back to close the learning-loop.

ber of failed computation attempts. The resulting artifact is a tabular output of the iterations that are plotted, filtering invalid attempts. The black-box function generates and runs the Python program for a given algorithm configuration and problem instance. Every iteration, a new repository Γ is dynamically built from the general repository Γ_0, which contains all combinator implementations. A point in the feature space only requires a subset of Γ_0, and the new specialized repository Γ will only contain relevant combinators. The inhabitation looks for well-typed terms that represent planning programs and can be built from this generated repository. In general, CLS allows the enumeration of multiple terms that comply with the target specification. For these specialized repositories only one inhabitant is found due to the restricted combinator selection. If a requested point in the feature space does not comply with the domain knowledge encoded in the combinator types, CLS yields an empty inhabitation result. This case may occur when the planner does not support the requested sampler. For instance, some planners rely on space sampling and do not use valid space sampling, which we determined by reviewing the OMPL code for every planner. The black-box function exposes invalid samples based on the inhabitation result, to allow the optimizer to proceed with space-exploration. Multiple planning program instances are executed to compute the average-case running time and path length, mitigating the variations of result values caused by randomized algorithms. The generation of the Python program will be performed by a Scala program that is generated by inhabitation. It makes use of Python code templates which can be complemented with code fragments, eventually building an executable Python program. The actions required for this build process are held in combinator implementations which are typed components encapsulating a piece of Scala code.

These combinators take care of various steps such as:

- Parsing of problem definition files and referencing the 3D models
- Setup of samplers, state validators and motion validators
- Definition of geometric space, start and goal states, planner
- Substitution of code fragments according to a substitution schema
- Execution of the Python programs and parsing of the console output

4.1 A Repository for Sampling Based Motion Planning

An essential part of the methodology is the repository for the component-based synthesis of motion planning programs set up as a proof-of-concept. The software components are implemented as CLS combinators which contain Scala code and a user-defined semantic type expression. For this study, 22 planner combinators, six combinators for samplers, and three combinators for state and motion validation were defined. Equation 4 displays three of these combinators and their user-defined semantic type signatures. They were selected to describe the basic principles of Scala programs that perform the generation, execution, and result extraction for Python scripts. The semantic repository Γ_s contains only the semantic types from Γ_0 for better readability.

$$
\begin{aligned}
&\Gamma_s = \{\\
&\text{PlannerAssembly: } \mathit{any\_planner} \to \mathit{any\_state\_validator} \to \mathit{any\_motion\_validator} \to\\
&\mathit{any\_simplification} \to \mathit{sbmp\_input} \to \mathit{sbmp\_program},\\
&\\
&\text{PRMStarSchema}: \ (\mathit{any\_opt\_obj} \to \mathit{sampler\_space} \to \mathit{PRMStar}) \ \cap\\
&(\mathit{any\_opt\_obj} \to \mathit{sampler\_valid\_state} \to \mathit{PRMStar}) \ \cap\\
&(\mathit{any\_opt\_obj} \to \mathit{sampler\_informed} \to \mathit{PRMStar}),\\
&\\
&\text{ESTSchema}: \ \mathit{obj\_path} \to \mathit{sampler\_valid\_state} \to \mathit{EST},\\
&[...] \ \}
\end{aligned}
\tag{4}
$$

The combinator PlannerAssembly is a top-level component that constructs the specified Scala program. According to the rule of modus-ponens in Section 3.3, CLS can form a program described by *sbmp_program* when it builds or finds terms that comply with the semantic types *any_planner*, *any_state_validator*, *any_motion_validator*, *any_simplification*, and *sbmp_input*. It will yield a function typed ProblemDefinitionFile → List[List[Float]] which loads an OMPL configuration file and executes the specified motion planning program. List[List[Float]] represents the result as a list of states that can be transformed to a continuous path by linear interpolation. CLS provides means to define a subtyping relation for semantic types by using a semantic taxonomy. For instance, the semantic type *any_planner_type* is a base type that matches every semantic type describing a planner. However, the repository is dynamically built to represent a single

algorithm configuration and will only contain one planner combinator. This semantic structure allows efficient search for inhabitation results even for large feature spaces.

The implementation resolves substitution schemas found by inhabitation to generate the Python OMPL scripts. They contain string mappings used for templating and mappings for file paths of template and output files. The combinator makes use of a wrapper class that invokes the templating, holds the file path to the main Python script and performs execution and parsing of the generated program. The parse function can be adapted to handle planner-specific output, different state representations and to accept approximate solutions. A data substitution schema loads problem-specific data according to the contents of a configuration file and has the native type ProblemDefinitionFile $\rightarrow$ SubstitutionSchema. It produces the corresponding substitution schema, which contains Python code to define the configuration space as well as the start and goal state. Moreover, it ensures that the problem-specific geometric models for the environment and the robot model are loaded as bounding volume hierarchy objects using the Flexible Collision Library (FCL [23]). These can be accessed as global variables, allowing collision checks to be performed inside OMPL state and motion validator implementations. The collision detection combinators handle the allocation of correct state and motion validators for the OMPL planning instance.

4.2 Encoding Domain Knowledge in Semantic Types

CLS semantic types are often used for the encoding of domain knowledge. In this work, the semantic layer expresses which specific planners can use samplers and optimization objectives. The type signatures displayed in Equation 4 contain this information for the OMPL implementations of the two planners Probabilistic Roadmap Method Star (PRM*, [20]) and Expansive Space Trees (EST, [15]). PRM* is capable of utilizing space samplers or valid state samplers. We also allow the use of an informed sampler, as it is a subclass of space sampler. 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.

In case the specified sampler is not used by the configured planner, CLS will not find an inhabitant. The inhabitation result will eventually be empty, signaling that the requested algorithm configuration is invalid. These encodings were collected for all considered planners and formulated as type expressions.

5 Experiments

5.1 3D Rigid Body Planning

The first part of the experimental evaluation deals with 3D rigid body planning. We selected the OMPL provided planning problems[1] and performed design space

[1] http://omplapp.kavrakilab.org/

exploration for every suitable problem instance. The experiments were conducted with a design of experiment (DoE) phase with 50 randomly sampled algorithm configurations, followed by 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. Planning program instances that fail to find an exact solution after a given time are considered unsuccessful program executions in the result set. The solution paths of the sample-based motion planning programs are subjected to OMPL simplification procedures as a post-processing step with a simplification time of 2.0 seconds. Fig. 2 shows the optimization criteria computation time and path length for an experiment on the problem instance *Abstract*. Every point represents a single iteration result, i.e., the averaged result of 10 program instances for a specified algorithm configuration.

The design space includes the complete set of OMPL planners and samplers with Python bindings. However, the plot contains only valid algorithm configurations selected by the exploration procedure, and we only consider motion planning programs that yielded exact solutions. 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. The computation took at least 7 seconds, including simplification. This is, however, not comparable to the OMPL planner arena data due to differences in the experimental setup. In this particular experiment, our methodology determined the planners Expansive Space Trees (EST, [15]) and SBL ([29]) to be well-suited for the given planning problem. Gaussian valid state sampling appears to have a positive impact on computation times, while maximum clearance valid state sampling and uniform valid state sampling yielded strong results in the path length dimension.

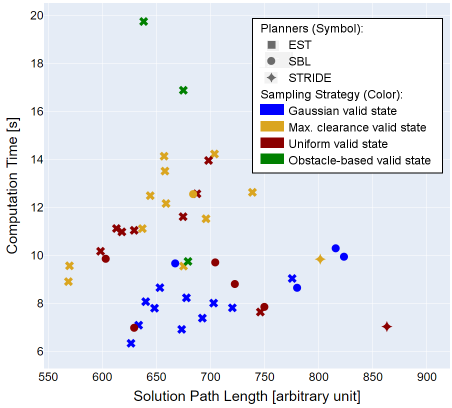

Fig. 2. Results for the motion planning problem `Abstract`

Hypermapper can inform about the importance of the features. This metric is displayed in table 5.1 and was retrieved at the end of the optimization run. As expected, the planner selection has the most impact on the different optimization objects. The low importance of the motion validator selec-

objective	planner	sampler	motion validator	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 1. Parameter importance for the problem instance `Abstract`

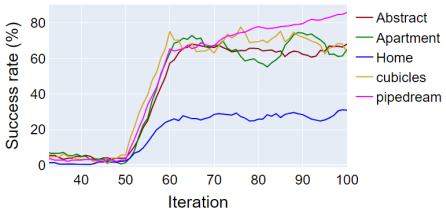

Fig. 3. Success rate for various problem instances

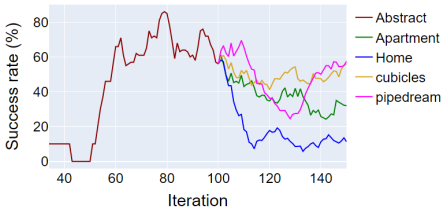

Fig. 4. Success rate for a model transfer between problem instances

tion could be explained by the planners' sparse use of collision checks and the simple geometric model of the *Abstract* problem instance.

We study the ability of our methodology to find valid algorithm configurations. The trend of the probability of finding a valid program can be considered as an indicator of the positive impact of the applied optimization techniques. The experiments were performed with an asynchronous execution of ten generated program instances per iteration. Thus, the number of successful program runs is in the interval [0.0, 10.0], which was normalized to a percentage. The number of successful path computations per iteration is computed based on the output. 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. The resulting trend of the success rate for multiple optimization runs is displayed in Fig. 3.

We can observe a growing success rate with a significant increase after the DoE phase (50 iterations). This finding suggests that our proposal is valid for the given domain. The exploration involves the sampling of 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. We incorporated a pre-learned model for the problem instance "Abstract" (using 50 DoE and 50 active learning iterations) to explore the algorithmic feature space for different problem instances. This approach allows examining the interchangeability of the problem-specific models, using the success rates as a metric. Fig. 4 illustrates the averaged measurements for the experiments, which begin at iteration 101. The captured data suggests that an interchange of models is viable to some extent. As expected, the recorded success rates are generally lower than in Fig. 3. The "Home" instance shows degraded success rates close to the performance of random sampling due to its distinctive characteristics. The remaining planning problems appear to be more receptive to the transferred model as they show success rates in the range of 30% to 60%.

5.2 Motion Planning for Configurable Robotic Arms

An additional series of experiments demonstrates the extensibility of our methodology to other robotic systems and planning instances, using planning problems for robotic arms as an example. The abstract notion of state validity allows integrating other robotic systems by setting up suitable state validators and space definitions. Component-based synthesis guides users to design modular components and also eliminates the effort for combining these modules. The planning problem introduces a new configuration space consisting of a real vector, where each dimension describes an angular position of the robot's joints. The configuration space utilizes the euclidean distance between samples, which results in optimal paths being defined as paths with minimal joint movement. Additional software components enable the code generation of space definition, sampling, and definition of goal and start state. Moreover, the problem requires a state validator fitted to this robotic family. The validation finds the position and orientation for each segment of the robot arm by using forward kinematics, which involves the generation and transformation of FCL collision objects for each arm segment. A series of collision checks examine the state validity regarding the environment and other parts of the robotic system. A state is valid if all checks confirm a collision-free pose of the associated robot segment. The state validation instance automatically reads the robotic system's degrees of freedom (DoF) and its geometric models from URDF files.

The robotic arms utilized are synthesized from a separate repository consisting of individual components using CLS. The individual components are motors, their corresponding mounting brackets, and structural parts. The synthesized robotic arms have an arbitrary number of revolute joints. For each synthesized robotic arm, a corresponding URDF file and MoveIt! package for visualization is generated. These are utilized to obtain example planning problems (Fig. 5) for varied robotic arms and visually verify the resulting motion plans. Due to the robotic arms being synthesized, a wide range of robotic arms are used in the experiments. Robotic arms with DoF between four and ten are utilized. Different environments that vary in their complexity and restrictiveness are tested.

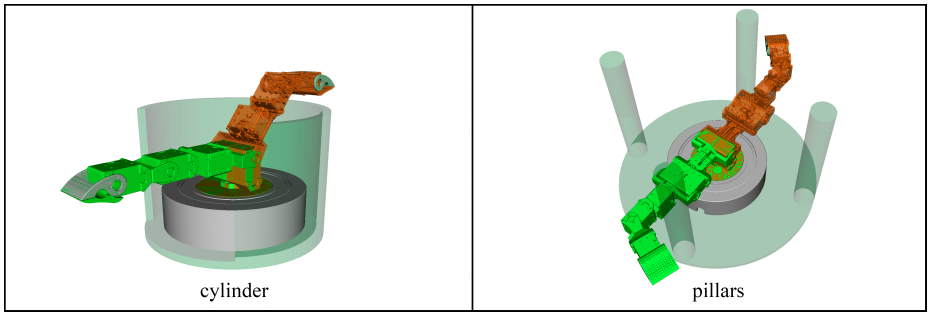

Fig. 5. Visual representation of problem instances

We study the influence of path simplification by comparing experiments with and without the post-processing simplification step provided by the OMPL built-in simplification procedures. The plots in Fig. 6 and Fig. 7 stem from two different optimization runs for the same planning problem. We can observe a notable shift of the Pareto front as post-processing leads to shortened paths and higher overall runtime of the motion planning programs. Moreover, the problem definition favors a different class of planners. As such, Fig. 7 shows a significant accumulation of good data points for RRT Connect with uniform space sampling.

Fig. 8 and Fig. 9 show the optimization results for the same environment (pillars) with different robotic systems (5 and 6 DoF). The differences in the robot models affect the number of configuration space dimensions and the computational cost of state validation. The plots imply that the suitability of planners may change according to the underlying planning problem. Additional experiments for challenging problems involving robotic arms with 10 DoF were only solvable with the Lazy RRT planner. These findings imply that the learned optimization models are, in fact, robot-specific and thus offer limited transferability to new robotic systems. In general, the given problem instances appear to be favorable for planners using uniform space sampling.

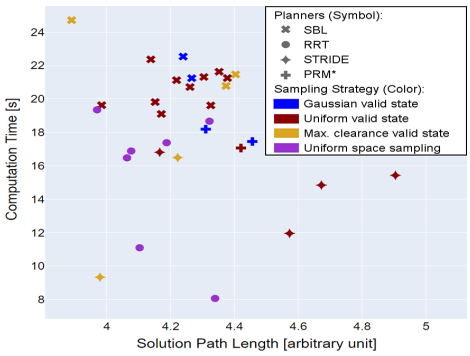

Fig. 6. cylinder with path simplification

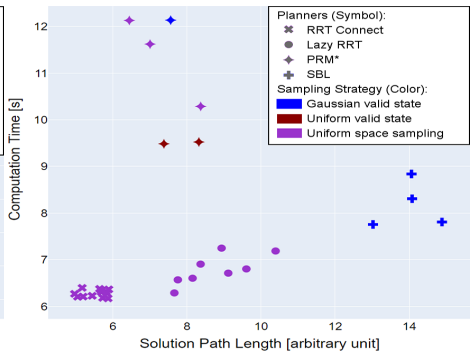

Fig. 7. cylinder without path simplification

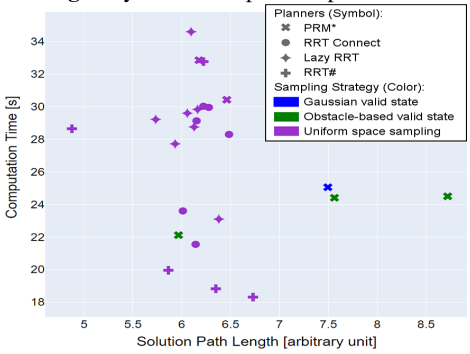

Fig. 8. pillars with 5 DoF

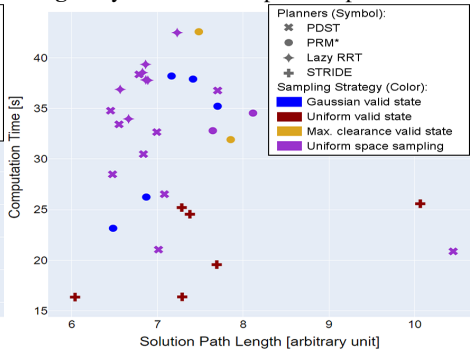

Fig. 9. pillars with 6 DoF

6 Discussion

While many algorithm configuration techniques try to find a parameter configuration for a given algorithm, CLS composes different algorithms. Our methodology allows a hybrid search that considers different algorithms and their parameterization. Moreover, it is possible to handle planner-specific parametrization in a combinatory way by introducing multiple combinators with preset parameter values for a single planner. That way, the impact of these configurations on the optimization objectives could be observed with only minor changes to the experimental design.

In this work, CLS is used to generate Python scripts and the methodology is extendable to further programming languages. In general, our methodology 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 collision detection libraries.

We used a small repository that represents a space consisting of several hundred unique feature configurations in this work. While currently limited to OMPL programs, additional combinators emerging from real-life scenarios can enrich the repository, allowing applicability to a broader range of problems. The adaption to new planning problems involves minor additions to the algorithm feature space, combinators, algorithm feature model, and Python templates. At the same time, the structure of the approach may remain as depicted in Fig. 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 develop and evaluate planners, sampling strategies, or collision detection techniques.

Hypermapper offers a great range of functionality that benefits the component-based synthesis. Nevertheless, it is possible to use alternative optimization frameworks by setting up specific black-box functions. The loose coupling between black-box function, synthesis framework, and evaluation facilities supports easy integration of alternative optimization techniques.

References

1. Bessai, J.: A type-theoretic framework for software component synthesis. Ph.D. thesis, Technical University of Dortmund, Germany (2019). URL http://hdl.handle.net/2003/38387
2. Bessai, J., Chen, T., Dudenhefner, A., Düdder, B., de'Liguoro, U., Rehof, J.: Mixin composition synthesis based on intersection types. Logical Methods in Computer Science **14**(1) (2018). DOI 10.23638/LMCS-14(1:18)2018
3. Bessai, J., Düdder, B., Heineman, G.T., Rehof, J.: Combinatory synthesis of classes using feature grammars. In: FACS, *Lecture Notes in Computer Science*, vol. 9539, pp. 123–140. Springer (2015). DOI 10.1007/978-3-319-28934-2_7
4. Bessai, J., Düdder, B., Heineman, G.T., Rehof, J.: Towards language-independent code synthesis (2018). URL https://popl18.sigplan.org/details/PEPM-2018/12/Towards-Language-independent-Code-Synthesis-Poster-Demo-Talk-. Poster and Talk

5. Bessai, J., Dudenhefner, A., Düdder, B., Martens, M., Rehof, J.: Combinatory logic synthesizer. In: ISoLA (1), *Lecture Notes in Computer Science*, vol. 8802, pp. 26–40. Springer (2014). DOI 10.1007/978-3-662-45234-9_3
6. Brock, O., Khatib, O.: Elastic strips: A framework for motion generation in human environments. The International Journal of Robotics Research **21**(12), 1031–1052 (2002)
7. Campigotto, P., Passerini, A., Battiti, R.: Active learning of pareto fronts. IEEE Transactions on Neural Networks and Learning Systems **25**(3), 506–519 (2014). DOI 10.1109/TNNLS.2013.2275918
8. Geraerts, R., Overmars, M.H.: Creating high-quality paths for motion planning. The international journal of robotics research **26**(8), 845–863 (2007)
9. Graefenstein, J., Winkels, J., Lenz, L., Weist, K., Krebil, K., Gralla, M.: A hybrid approach of modular planning - synchronizing factory and building planning by using component based synthesis. In: HICSS, pp. 1–9. ScholarSpace (2020). DOI 10.24251/HICSS.2020.806
10. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1-2), 1–119 (2017). DOI 10.1561/2500000010
11. Heineman, G.T., Bessai, J., Düdder, B., Rehof, J.: A long and winding road towards modular synthesis. In: ISoLA (1), *Lecture Notes in Computer Science*, vol. 9952, pp. 303–317 (2016). DOI 10.1007/978-3-319-47166-2_21
12. Heineman, G.T., Hoxha, A., Düdder, B., Rehof, J.: Towards migrating object-oriented frameworks to enable synthesis of product line members. In: D.C. Schmidt (ed.) Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015, pp. 56–60. ACM (2015). DOI 10.1145/2791060.2791076. URL https://doi.org/10.1145/2791060.2791076
13. Hindley, J.R., Seldin, J.P.: Lambda-calculus and Combinators, an Introduction. Cambridge University Press (2008)
14. Hosny, A., Hashemi, S., Shalan, M., Reda, S.: Drills: Deep reinforcement learning for logic synthesis. In: ASP-DAC, pp. 581–586. IEEE (2020). DOI 10.1109/ASP-DAC47756.2020.9045559
15. Hsu, D., Latombe, J., Motwani, R.: Path planning in expansive configuration spaces. Int. J. Comput. Geom. Appl. **9**(4/5), 495–512 (1999). DOI 10.1142/S0218195999000285
16. Ichter, B., Schmerling, E., Lee, T.W.E., Faust, A.: Learned critical probabilistic roadmaps for robotic motion planning. In: 2020 IEEE International Conference on Robotics and Automation (ICRA), pp. 9535–9541 (2020). DOI 10.1109/ICRA40945.2020.9197106
17. İpek, E., McKee, S.A., Caruana, R., de Supinski, B.R., Schulz, M.: Efficiently exploring architectural design spaces via predictive modeling. SIGPLAN Not. **41**(11), 195–206 (2006). DOI 10.1145/1168918.1168882
18. Jamshidi, P., Camara, J., Schmerl, B., Kaestner, C., Garlan, D.: 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), pp. 39–50. IEEE (25.05.2019 - 25.05.2019). DOI 10.1109/SEAMS.2019.00015
19. Kallat, F., Pfrommer, J., Bessai, J., Rehof, J., Meyer, A.: Automatic building of a repository for component-based synthesis of warehouse simulation models. Procedia CIRP **104**, 1440–1445 (2021). DOI https://doi.org/10.1016/j.procir.2021.11.243. 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0

20. Kavraki, L.E., Svestka, P., Latombe, J.., Overmars, M.H.: 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
21. Morales, M., Tapia, L., Pearce, R., Rodriguez, S., Amato, N.M.: A Machine Learning Approach for Feature-Sensitive Motion Planning, pp. 361–376. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). DOI 10.1007/10991541_25
22. Nardi, L., Koeplinger, D., Olukotun, K.: Practical design space exploration. In: MASCOTS, pp. 347–358. IEEE Computer Society (2019). DOI 10.1109/MASCOTS.2019.00045
23. Pan, J., Chitta, S., Manocha, D.: FCL: A general purpose library for collision and proximity queries. In: ICRA, pp. 3859–3866. IEEE (2012). DOI 10.1109/ICRA.2012.6225337
24. Rehof, J.: Towards combinatory logic synthesis. In: 1st International Workshop on Behavioural Types, BEAT (2013)
25. Rehof, J., Urzyczyn, P.: Finite combinatory logic with intersection types. In: TLCA, *Lecture Notes in Computer Science*, vol. 6690, pp. 169–183. Springer (2011). DOI 10.1007/978-3-642-21691-6_15
26. Rehof, J., Vardi, M.Y.: Design and synthesis from components (dagstuhl seminar 14232). Dagstuhl Reports **4**(6), 29–47 (2014). DOI 10.4230/DagRep.4.6.29
27. Reif, J.H.: Complexity of the mover's problem and generalizations. In: 20th Annual Symposium on Foundations of Computer Science (sfcs 1979), pp. 421–427 (1979). DOI 10.1109/SFCS.1979.10
28. Saeedi, S., Nardi, L., Johns, E., Bodin, B., Kelly, P.H.J., Davison, A.J.: Application-oriented design space exploration for slam algorithms. In: 2017 IEEE International Conference on Robotics and Automation (ICRA), pp. 5716–5723 (2017). DOI 10.1109/ICRA.2017.7989673
29. Sánchez-Ante, G., Latombe, J.: A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In: ISRR, *Springer Tracts in Advanced Robotics*, vol. 6, pp. 403–417. Springer (2001)
30. Schäfer, T., Bergmann, J.A., Carballo, R.G., Rehof, J., Wiederkehr, P.: A synthesis-based tool path planning approach for machining operations. Procedia CIRP **104**, 918–923 (2021). DOI https://doi.org/10.1016/j.procir.2021.11.154. 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0
31. Şucan, I.A., Moll, M., Kavraki, L.E.: The Open Motion Planning Library. IEEE Robotics & Automation Magazine **19**(4), 72–82 (2012). DOI 10.1109/MRA.2012.2205651. https://ompl.kavrakilab.org
32. Wang, J., Meng, M.Q..H., Khatib, O.: Eb-rrt: Optimal motion planning for mobile robots. IEEE Transactions on Automation Science and Engineering **17**(4), 2063–2073 (2020). DOI 10.1109/TASE.2020.2987397
33. Wenzel, S., Stolipin, J., Rehof, J., Winkels, J.: Trends in automatic composition of structures for simulation models in production and logistics. In: WSC, pp. 2190–2200. IEEE (2019). DOI 10.1109/WSC40007.2019.9004959
34. Xiang, Y., Huang, H., Zhou, Y., Li, S., Luo, C., Lin, Q., Li, M., Yang, X.: Search-based diverse sampling from real-world software product lines. In: 2022 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), International Conference on Software Engineering. Proceedings. IEEE (2022). Not yet published as of 04/02/2022
35. Zuluaga, M., Krause, A., Püschel, M.: e-pal: An active learning approach to the multi-objective optimization problem. Journal of Machine Learning Research **17**(104), 1–32 (2016)