A Hierarchical Planner based on Set-theoretic Models: Towards Automating the Automation for Autonomous Systems

Bernd Kast[1][a], Vincent Dietrich[1][b], Sebastian Albrecht[1][c], G. Wendelin Feiten[1][d], and Jianwei Zhang[2]

[1]*Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, 81739 Munich, Germany*

[2]*University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany*

Keywords: Hierarchy, Planning, Autonomy.

Abstract: The complexity of today's autonomous systems renders the manual engineering of control strategies or behaviors for all possible system states infeasible. Therefore, planning algorithms are required that match the capabilities of the system to the tasks at hand. Solutions to typical problems with robotic systems combine aspects of symbolic action planning with sub-symbolic motion planning and control. The problem complexity of this combination currently prohibits online planning without task specific, manually defined heuristics. To counter that we use a set-theoretic approach to model declarative and procedural knowledge which allows for flexible hierarchies of planning tasks. The coordination of the planning tasks on different levels, the classification of information and various views on data are the core functions of hierarchical planning. We propose suitable graph structures to capture all relevant information and discuss the elements of our hierarchical planning algorithm in this paper. Furthermore, we present two use-cases of an autonomous manufacturing system to highlight the capabilities of our system.

1 INTRODUCTION

Setting up autonomous systems still requires huge integration and engineering efforts. In order to make them widely applicable, we need a holistic approach that even considers aspects of component integration. This is especially important in production, where a higher degree of automation, notably for small lot sizes, can account for changing customer demands.

A limiting factor of today's automation strategies is the interwoven product and production design which requires costly manual effort even for small changes of some component (Hitomi, 2017), (Bryan et al., 2007). To reduce this effort, the design process of the product and the production system have to be decoupled (Hu et al., 2011) and then matched again by an autonomous system. Thus, the autonomous system has to solve on its own certain engineering tasks that these days are addressed manually when automating a production system, i.e. the automation of automation (Schmitz et al., 2009) is necessary for flexible autonomous systems.

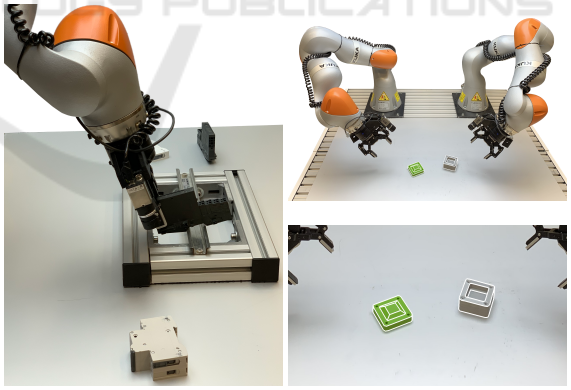

Figure 1: Images of the robotic system (upper right) and the hat-rail with components (left) and the box next to its lid (lower right).

This requires general models which characterize objects by their properties (declarative knowledge) and describe actions modifying them (procedural knowledge). These models define domains for

[a] https://orcid.org/0000-0001-7838-3142
[b] https://orcid.org/0000-0003-0568-9727
[c] https://orcid.org/0000-0002-3647-4043
[d] https://orcid.org/0000-0002-7593-6298

Kast, B., Dietrich, V., Albrecht, S., Feiten, G. and Zhang, J.
A Hierarchical Planner based on Set-theoretic Models: Towards Automating the Automation for Autonomous Systems.
DOI: 10.5220/0007840702490260
In *Proceedings of the 16th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2019)*, pages 249-260
ISBN: 978-989-758-380-3

suitable planners. Real-world manufacturing domains suffer from a huge computational complexity due to their mixed continuous and discrete nature. Discrete symbolic action planning and scheduling bring in the problem of combinatorial explosion while sub-symbolic tasks such as motion planning or control introduce time, accuracy and stability constraints. A common approach to address this curse of dimensionality is to use abstractions or hierarchies to decompose the problem. This allows to extract smaller subproblems that are easier to handle. It is especially important to deduce those hierarchies for both, procedural and declarative knowledge, automatically from the models in order to keep the task and plant description separate. Additionally, assumptions that limit the flexibility of the system have to be avoided. One example is the *downward refinement property* which demands that each coarse plan can be refined on all more detailed levels. This is in general not viable for real-world problems with decoupled models for the problem and the plant. This becomes obvious for task specific sub-symbolic properties, such as possible grasp positions, which can for the general case not be modeled on an abstract, purely symbolic level.

Throughout this paper, we illustrate our approach with an industrial assembly use-case, cf. Figure 1, which is modeled for maximal flexibility. This means in particular, that not only the actions of the robot, but even the interplay between software components is planned, cf. section 8. We present a planner that benefits from models that integrate declarative and procedural knowledge and allow for an automatic calculation of abstractions. This planner uses the graph structures of our models on different abstraction levels to factorize the problem, integrates special single-level planners for individual subproblems but does not depend on the downward refinement property.

The paper is structured as follows: In section 2 we present related work regarding hierarchical symbolic planning, combined task and motion planning in robotics and autonomous production systems. After that, we discuss a set-theoretic approach for declarative and procedural models that natively enables the deduction of hierarchical structures in section 3. The data structures for the single-level planning are described in section 4 and the respective single-level planner is discussed in section 5. In section 6 we extend these concepts for the hierarchical planning context. Finally, we present the hierarchical planning method in section 7 and demonstrate its features on two examples in section 8.

2 RELATED WORK

Planning algorithms are often tailored to modeling languages and rely on their expressiveness. In this section we discuss both symbolic and sub-symbolic planning, as well as the application specific aspects related to for autonomous production systems.

2.1 (Hierarchical) Action Planning

The key for flexible systems is a modeling language that formalizes the description of the domain. On the symbolic level, there are many languages such as PDDL (McDermott et al., 1998) and its dialects that focus on the procedural knowledge of a domain. A long series of planning competitions have resulted in a large set of fast planners for PDDL domains such as (Helmert, 2006). Extensions to PDDL add support for certain sub-symbolic properties such as time. An example is PDDL+ (Fox and Long, 2002) that introduces events and processes to model exogenous change and support domains with mixed discrete and continuous dynamics. Corresponding solvers, such as (Cashmore et al., 2016) or (Piotrowski et al., 2016), make use of this representation and handle domains with nonlinear continuous change. They approximate the dynamics of the system and handle the resulting discretized model with uniform time steps and step functions. Other PDDL dialects, e.g. (Dornhege et al., 2009), extend the formalism by using semantic attachments that allow for an evaluation of externally specified functions. Though, this requires modifying standard PDDL planners and to create domain-specific PDDL action choosing the specific external functions.

Another approach is partial-order planning (POP) (Young et al., 1994) which involves partially specified action decompositions. In this approach, the planner is only allowed to fill in missing pieces of a fixed plan template which hugely reduces the search space. However, it is difficult to ensure the separation of hardware and task description with this approach, which reduces the flexibility.

A large community addresses hierarchical task networks (HTN) that refine each abstract skill by a network of sub-methods, e.g. (Castillo et al., 2006), (Goldman, 2006), (Nau et al., 2003). In their basic form they were state-oriented and could not deal with time constraints or concurrent actions. Nevertheless, the formalism is more expressive than that of first principle planners (Erol et al., 1994), HTN methods can improve planning times (Nau et al., 2003) and even support plan reparation (Gateau et al., 2013). In (Bercher et al., 2016) an overview of HTN meth-

ods is provided that discusses the expressiveness of hierarchical planning formalisms as well as implications of preconditions and effects of abstract methods. A mandatory and limiting condition for HTNs is the downward refinement property that enforces refinements to all abstract solutions (Bacchus and Yang, 1994). Yet, the generation of suitable abstract models for a domain is a challenging or even impossible task and often results in a coupling between task and hardware description.

This becomes obvious in (Marthi et al., 2008). They propose expressive models, which allow for the definition of domains with the downward refinement property. In this approach, an underlying semantic that defines preconditions and effects even for abstract tasks, ensures that abstract plans can always be refined to a primitive solution. However, this demands that all relevant effects and preconditions of the primitives have to be considered even on the most abstract level. This is only possible if all properties can be described symbolically and hugely limits the benefits of the hierarchical abstractions.

To overcome the limitations induced by the downward refinement property, several variants of HTN planning and hybrid planning (Kambhampati et al., 1998), (Schattenberg, 2009) have been proposed. The hierarchical partial-order planner, introduced in (Bechon et al., 2014), uses additional knowledge that describes sets of abstract actions with optional methods to increase flexibility during refinement. Another approach to improve versatility is the combination of HTN and POP in a domain-specific planner with a strict separation between several hierarchical levels (Castillo et al., 2003). In both approaches the effects of the downward refinement property are mitigated at the cost of models which are dependent on the hardware and the task at the same time.

Temporal reasoning within an HTN planner is discussed in (Castillo et al., 2006) and an HTN extension for (nonlinear) continuous temporal environments can be found in (Molineaux et al., 2010), in which the SHOP2 planner (Nau et al., 2003) is matched to features from PDDL+ (Fox and Long, 2002). Those approaches offer a suitable performance even in domains with sub-symbolic properties but don't support general geometric constraints such as collisions. An overview of HTN planning can be found in (Georgievski and Aiello, 2015).

Each of those approaches extends the application area. Summarizing we state that manually drafted hierarchies, the strict requirement of the downward refinement property or a limited support of continuous properties prevents an application in our general manufacturing domain.

2.2 Combined Task and Motion Planning in Robotics

A challenging and widely discussed subproblem for autonomous systems in production is task and motion planning. In (Srivastava et al., 2014) existing task planners and motion planners are combined by the introduction of new symbolic abstractions. Motion planning is used to refine the plans from symbolic planning to a sub-symbolic level. The general idea is to add further abstract poses to the symbolic planning problem every time the refinement failed because no collision-free path could be found. This allows to consider continuous properties on an abstract level without manual prior modeling. However, the computational complexity is shifted to the abstract level. The advantages of the hierarchy are additionally reduced by the limited number of abstractions that are possible with this approach.

Instead of combining of two separate planners (Garrett et al., 2015) extends the symbolic planners and their heuristics to motion planning and thus lifts the sub-symbolic world to the symbolic planning. The heuristic is based on domain-dependent literals that represent reachability for example, which can be evaluated lazily on demand. The planner maintains a corresponding reachability graph of sampled configurations. Geometric constraint-satisfaction problems (CSP) determine plan templates with unbound variables like robot and object poses (Lozano-Pérez and Kaelbling, 2014). The central disadvantages of these approaches are that a discretization has to be specified in advance and that they run into scalability issues if a fine discretization is needed.

Another possibility is to add the symbolic properties to the sub-symbolic path planning problem. If a simple symbolic planner is used to generate action sequences, large optimization problems can be formulated that determine optimal intermediate and final states (Toussaint, 2015). They demonstrated the strength of their approach for a stacking problem in which towers of cylinders and plates have to be assembled. However, it is computationally too demanding for online planning and scales exponentially with the number of involved objects.

The ScottyActivity planner (Fernandez-Gonzalez et al., 2018) is one of the few approaches that consider dynamics and/or temporal constraints along the manipulation problem. It combines strong heuristics with convex optimization and relaxed plan graphs. However, the absence of obstacles and the limitation to linear dynamics for the robots prohibits the applicability for real-world problems.

In (Schmitt et al., 2017) an asymptotically optimal

manipulation planner is proposed that considers both, nonlinear dynamics and collisions. Without any hierarchical decomposition or heuristics, their approach, which extends sampling-based roadmap planners to explore configuration spaces, scales exponentially.

2.3 Autonomous Production Systems

There are three main challenges for autonomous robotics in production: planning to generate a coarse, symbolic plan, mapping the plan to the actual hardware and controlling it accordingly. The pioneering system (Kaufman et al., 1996) is one of the few who proposed an integrating approach to target all three challenges at a time. However, this approach relied on substantial task specific manual modeling of subassemblies, which limits flexibility. Additionally, the approach is tested in simulation only and therefore it neglects sensor input and generates a nominal control code only. In (Thomas and Wahl, 2010) an improved version is presented that uses CAD data and other models to compute assembly sequences that comply with the desired goal configuration. The optimal plan is chosen, mapped to the robot's skills and executed on the hardware. Due to the lack of a hierarchical decomposition, this approach, however, suffers from the curse of dimensionality.

3 FORMAL MODELS

The planning process requires models for declarative and procedural knowledge. For our hierarchical planning approach, it is especially important that both types are naturally hierarchically structured and that these hierarchies match each other. We accomplish that with the following set-theoretic definitions, which have been introduced in more detail in (Kast et al., 2019).

3.1 Concepts

In this work we use the term *concepts* for elements of the declarative knowledge. An example of such a concept is a simple box. Each specific box is an instance of the abstract concept "box", but the properties of boxes differ in detail. For example, the shape might be different, as one box is cubic and the other cylindrical. Thus, we have an abstract concept of a box and two specializations capturing subsets. This notion of concepts can be formally grounded by following set-theoretic definitions:

- A concept base B_Γ is the set of instances, not necessarily finite.
- A concept C is a subset of B_Γ, i.e., $C \subseteq B_\Gamma$.
- A concept class Γ is the set of concepts C_i that have a common concept base B_Γ, i.e., $\forall\, C_i \in \Gamma, b \in C_i : b \in B_\Gamma$.
- A partial order $\mathcal{M}$, describing *more detailed than*, can be defined on Γ: $(C_i, C_j) \in \mathcal{M}$ iff $C_i \subset C_j$.

Extending the previous example: In most cases you do not only want to know the shape of the box but also the characteristics of the dimensions. Further properties, like the weight or the current location, might also be interesting. This directly leads to special types of concepts, which we call *composite concepts*.

The following definitions introduce composite concepts formally:

- Given an ordered set $\mathcal{R}$ of identifiers, e.g. strings, its elements $r \in \mathcal{R}$ are called roles. Thus, for each subset $\mathcal{R}_i \subseteq \mathcal{R}$ with $n_{\mathcal{R}_i} := |\mathcal{R}_i|$ there exists a bijective mapping $\mathcal{J}$ to $\mathbb{N}_{n_{\mathcal{R}_i}} := \{1, \ldots, n_{\mathcal{R}_i}\}$, i.e., $\mathcal{J}(r) \in \mathbb{N}_{n_{\mathcal{R}_i}}\ \forall r \in \mathcal{R}_i$ and $\mathcal{J}^{-1}(n) \in \mathcal{R}_i\ \forall n \in \mathbb{N}_{n_{\mathcal{R}_i}}$.
- A composite, recursively defined concept $C = \Pi_{r \in \mathcal{R}_C} C_r$ with $\mathcal{R}_C$ being the specific set of roles for this composite concept

This definition of a composite concept corresponds to a (directed) graph, cf. Figure 2: the nodes correspond to (sub-)concepts and the edges point from the composites to their sub-concepts.

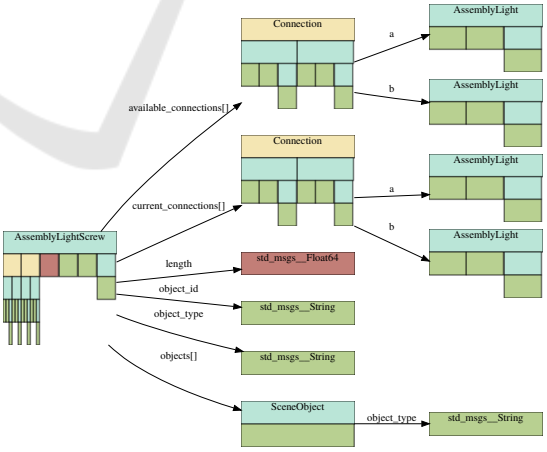

Figure 2: Example of a composite concept that describes objects in the later examples. Here, only the composite structure of the first level of sub-concepts is additionally depicted. Note that the small images for each node visualize the composite structures. A common color is used for each concept class. The edges are attributed with roles.

Having two composite concepts of one concept class $C, \overline{C} \in \Gamma$, the partial order $\mathcal{M}$ can be deduced

recursively:

$$(C,\overline{C}) \in \mathcal{M} \text{ iff } (C_r,\overline{C}_r) \in \mathcal{M}\ \forall r \in \mathcal{R}_{\overline{C}},$$

which requires that C has all the roles of $\overline{C}$. This means that (see Figure 3):

$$C \cong \Pi_{r\in\mathcal{R}_C\cap\mathcal{R}_{\overline{C}}} C_r \times \Pi_{r\in\mathcal{R}_C\setminus\mathcal{R}_{\overline{C}}} C_r \subseteq \overline{C} \times \Pi_{r\in\mathcal{R}_C\setminus\mathcal{R}_{\overline{C}}} B_{\Gamma(r)}.$$

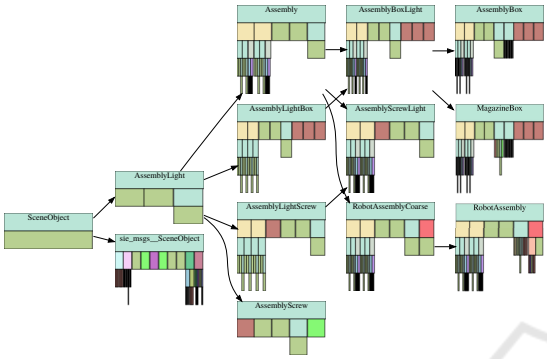

Figure 3: Example hierarchy of the concept class for objects in the later examples. The graph is directed and acyclic, but not necessarily a tree.

By definition the leaves of a concept are considered to be atomic at the modeling detail of the provided concept. Each element of a concept is called an *instance*. For composite concepts this results in the necessity to specify values for each leaf concept of a composite concept.

In order to answer the question whether two instances of the same concept represent the same thing, we assume that for each concept C a relation, the so-called *compare relation* F_C, is defined. This means two instances b_i, b_j are similar with respect to a concept C if $(b_i,b_j) \in F_C$. The product representation allows to derive similarity of two composite concepts when all their corresponding leaves are similar. Note, that similarity does not mean that the instances are identical, but rather that the information provided by the second instance does not add any additional information on a given level of abstraction of a domain. This compare relation will help to introduce *compact domains*, in which different results are mapped to the same graph node if the corresponding instances are similar.

3.2 Operators

Declarative knowledge is hardly useful if elements of the procedural knowledge, so-called *operators*, do not use them. We define operators as follows (see Figure 4):

- An operator $\pi \in P$ is a mapping of given input concepts I_{r_i} to output concepts O_{r_j} with given input roles $r_i \in \mathcal{R}_{\pi,I}$ and output roles $r_j \in \mathcal{R}_{\pi,O}$, i.e., $\pi : \Pi_{r_i\in\mathcal{R}_{\pi,I}} I_{r_i} \rightarrow \Pi_{r_j\in\mathcal{R}_{\pi,O}} O_{r_j}$.
- The output instances can be explicitly specified by symbolically-representable mappings (mathematical formulas) or implicitly as the result of some calculation or experiment in simulation or the real world, operating on input instances
- Certain operators describe modifications of instances, i.e., elements of knowledge are invalidated when such an operator is executed; such inputs are called *consumed*. The set of roles corresponding to inputs that are consumed by the operator is denoted by $\mathcal{R}^c_\pi \subseteq \mathcal{R}_{\pi,I}$.
- Operators are fully functional, i.e., they do not have an internal state.

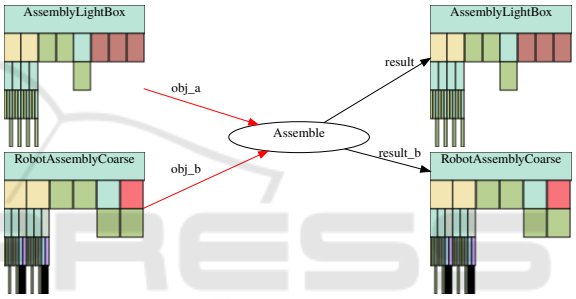

Figure 4: Example of an operator with multiple concepts as in- and outputs. A consumed input is depicted by a red edge.

Since operators are elements of a functional space, they can also be modeled as instances of an operator concept on a higher level of abstraction that is often called *meta level*. In addition to the input and output structures and a reference to the executable code, an instance of this operator-concept can contain *meta information* like key performance indicators (kpi).

A hierarchy of operators is obtained from the hierarchical structures of concepts. An operator π_1 is more detailed than another π_2 in case the following conditions hold true (see Figure 7):

- all input and output roles of operator π_2 are elements of the role set of operator π_1: $\mathcal{R}_{\pi_2,I} \subset \mathcal{R}_{\pi_1,I}$ and $\mathcal{R}_{\pi_2,O} \subset \mathcal{R}_{\pi_1,O}$,
- all (common) input concepts $I_{r,1}$ and outputs $O_{r,1}$ of operator π_1 are more detailed than those of operator π_2: $(I_{r,1},I_{r,2}) \in \mathcal{M}\ \forall r \in \mathcal{R}_{\pi_2,I}$ and $(O_{r,1},O_{r,2}) \in \mathcal{M}\ \forall r \in \mathcal{R}_{\pi_2,O}$,
- the meta information, e.g. key performance indicators, of operator π_1 are more detailed than those of operator π_2.

4 CONCEPTS AND GRAPHS FOR PLANNING

In our context planning is based on the evaluation of operators, working on the set of currently available instances, to reach one or multiple goal instances. The operators can use or consume available instances and generate new ones. Many application domains pose problems combining symbolic and sub-symbolic instances. However, standard approaches run into the curse of dimensionality. The approach of hierarchical planning is based on the hierarchies of concepts and operators. It tries to solve the planning task on an abstract level and to refine the plan step by step and abstraction level by abstraction level until a plan on the most detailed level is obtained.

To formalize the structures obtained during planning we first introduce formal models for a planning task, a planner and a plan.

4.1 Planning Task

A planning task $PT(X_{\text{init}}, X_{\text{goal}}, X_{\text{op}})$ specifies the combination of three sets: the set of initial instances $X_{\text{init}} := \{b_i,\ i = 1,\ldots,n_{init}\}$, the set of goal instances $X_{\text{goal}} = \{b_j,\ j = 1,\ldots,n_{goal}\}$ and the set of available operators $X_{\text{op}} := \{\pi_k,\ k = 1,\ldots,n_{op}\}$ with $n_{init} \in \mathbb{N}_0$, and $n_{goal}, n_{op} \in \mathbb{N}$. Consequently, a planning task is a concept in the meta domain.

4.2 Plan Family

We introduce the concept (in the meta domain) of a *plan family* that contains the following information: the initial instances, the executed operators, the input instances of executed operators with their roles, and the output instances of executed operators.

To represent this plan family, we use an annotated bi-partite graph $\mathcal{G}_{\text{PF}}(V, E, E^{\text{con}})$, in which all annotated edges in $E^{\text{con}} \subseteq E$ correspond to consumed operator inputs. The one node set V_B only contains nodes representing instances and the other V_π consists of operators only. The bi-partite property ensures that $V = V_B \cup V_\pi$, $V_\pi \cap V_B = \emptyset$ and $\forall e \in E:\ e = (v_1, v_2)$ with $\{v_1, v_2\} \not\subseteq V_\pi$ and $\{v_1, v_2\} \not\subseteq V_B$. The representation mapping $f_v : V \to B \cup P$ of nodes in $\mathcal{G}_{\text{PF}}$ to instances and operators is assumed to be bijective.

Adding an Executed Operator to a Plan Family. For every operator that is executed during planning, all new nodes are added to the graph as long as the graph does not already contain the information. The result is the bi-partite graph $(\overline{V}, \overline{E}, \overline{E}^{\text{con}})$ with $V \subseteq \overline{V}$ and $E \subseteq \overline{E}$. In more detail, this means that, with the given operator $\pi : \Pi_{r_i \in \mathcal{R}_{\pi,I}} I_{r_i} \to \Pi_{r_j \in \mathcal{R}_{\pi,O}} O_{r_j}$, the execution results in the following mapping between instances:

$$\{b_r \mid r \in \mathcal{R}_{\pi,I}\} \mapsto \{\overline{b}_r \mid r \in \mathcal{R}_{\pi,O}\}.$$

The graph is only modified if the entropy (number of different results for a given input set) of the operator, is not yet exhausted by former executions. Thus, a node v_π is added to V_π and all inputs are connected via a directed edge to this new node. Edges corresponding to consumed inputs are marked by adding them to E^{con}. Additionally, nodes for all output instances, i.e., $\{v_{f_v^{-1}(\overline{b}_r)} \mid r \in \mathcal{R}_{\pi,O}\}$, are added to V_B and v_π is connected to all these outputs via directed edges.

We introduce a so-called *compact domain*, in which new nodes are only added for instances that carry new information, i.e., node $v \in V$ is added if $\forall v_i \in V_B$ with $v_i \neq v : (f_v(v_i), f_v(v)) \notin \mathcal{M}$ or $(f_v(v), f_v(v_i)) \notin \mathcal{M}$. If the plan family already contains a node related to an instance with the same information as the output instance, the mapping f_v points to that node instead. Thus, compact domains have a considerably smaller number of nodes enabling better scaling. In particular, the number of considered operators during planning can exceed the number of nodes in the corresponding plan family (see Table 1 for the graph sizes in the robotic example). Note that in consequence a plan family in a compact domain can have cycles, since the operator outputs can be mapped to ancestor nodes of the operator node v_π.

Thus, the following equations hold for the sets of the plan family:

$$\begin{aligned}
\overline{V}_\pi &= V_\pi \cup \{v_\pi\},\\
\overline{V}_B &= V_B \cup \left\{ v_{f_v^{-1}(\overline{b}_r)} \;\middle|\; r \in \mathcal{R}_{\pi,O} \right\},\\
\overline{E} &= E \cup \left\{ \left(v_{f_v^{-1}(b_r)}, v_\pi \right) \;\middle|\; r \in \mathcal{R}_{\pi,I} \right\}\\
&\quad \cup \left\{ \left(v_\pi, v_{f_v^{-1}(\overline{b}_r)} \right) \;\middle|\; r \in \mathcal{R}_{\pi,O} \right\},\\
\overline{E}^{\text{con}} &= E^{\text{con}} \cup \left\{ \left(v_{f_v^{-1}(b_r)}, v_\pi \right) \;\middle|\; r \in \mathcal{R}^{\text{c}}_{\pi,I} \right\}.
\end{aligned}$$

4.3 Planner State Graph

Planning considers sets of instances X_i, $i \in \mathbb{N}$. The initial set is X_{init}. A plan is available if, by applying operators, a set X_m is generated that fulfills the goal set X_{goal}, i.e.,

$$\forall b_i \in X_{\text{goal}}\ \exists b_j \in X_m : (b_j, b_i) \in F_{C_i},$$

in which $b_i \in C_i$. Each set of instances X_i is obtained by executing one operator $\pi_k \in X_{\text{op}}$, $k \in \mathbb{N}$, on one selection of instances $(b_1, \ldots, b_{n_{\pi_k}})$ with $n_{\pi_k} := |\mathcal{R}_{\pi_k}|$

from a previous set of instances X_j, $j \in \{0,\dots,j-1\}$ (with $X_0 := X_{\text{init}}$):

$$X_i = (X_j \cup \pi_k(b_1,\dots,b_{n_{\pi_k}})) \setminus \{b_{\hat{n}} \mid \mathcal{J}_{\pi_k}^{-1}(\hat{n}) \in \mathcal{R}_{\pi_k}^{c}\},$$

in which the instances have to be an element of the correct concept type $b_{\mathcal{J}(r_l)} \in C_{r_l} \cap X_j$.

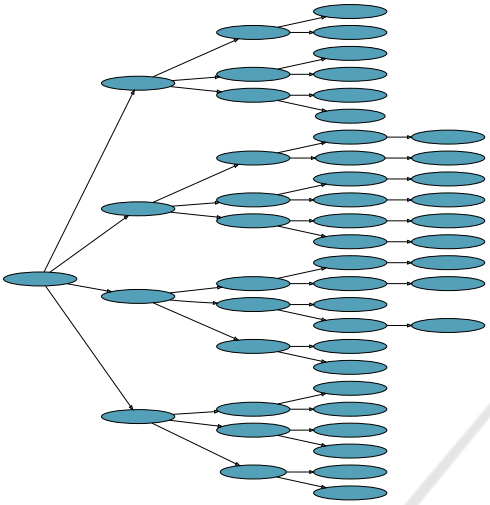

Figure 5: Part of the planner state graph for the later example of a hat-rail assembly.

Therefore, we introduce a directed graph $\mathcal{G}_{\text{PS}}(V,E)$ to describe the planner state as shown in Figure 5. Each node $v_i \in V$ corresponds to a set of instances X_i (and a subgraph of the plan family). There exists a directed edge $e = (v_i, v_j) \in E$ iff X_j was obtained from X_i by applying an operator. We assume a compact graph structure similar to a compact domain, i.e., $\forall v_i, v_j \in V, v_i \neq v_j : X_i \setminus X_j \cup X_j \setminus X_i \neq \emptyset$. This property reduces the size of the planning space considerably, because plan duplication is avoided for similar paths leading to the same intermediate planner state X_i. Additionally, this structure allows to detect dead ends, which is later used by the backtracking techniques for hierarchical planning.

5 SINGLE-LEVEL PLANNING

The classical planning problem is named *single-level planning* (SLP) as it doesn't consider hierarchies of concepts or operators. A planning task $PT(X_{\text{init}}, X_{\text{goal}}, X_{\text{op}})$ is solved in a constructive manner by applying operators from X_{op} to available instances.

A planning step is comprised by three actions:

1. choose the planning node v_i with the corresponding instances X_i to proceed from,
2. select an operator $\pi_j \in X_{\text{op}}$ with its input concepts I_r for $r \in \mathcal{R}_{\pi_j,I}$,
3. single out instances from X_i and map them to the inputs of π_j such that $b_r \in I_r \; \forall r \in \mathcal{R}_{\pi_j,I}$.

This general structure is valid for all kinds of planners ranging from PDDL to motion planning. Even classical graph algorithms like depth-first and breadth-first search can be used to realize such a single-level planner. In our implementation we let the search algorithm determine the next node v_i and iterate over all operators and all possible combinations of corresponding input instances for that state. More evolved planners have either better heuristics and sampling strategies or use suitable problem reformulations for specific sub-problems. Nevertheless, the core elements of forward planners correspond to the three steps described above.

Since each node v_i corresponds both to a set of instances X_i and a subgraph of the plan family, this subgraph captures the respective plan if the goal instances X_{goal} are met by the instances of X_i.

6 GRAPHS FOR HIERARCHICAL PLANNING

The computational complexity of the planning problem, which results from the mixture of symbolic and sub-symbolic properties, can be handled if hierarchies of suitable concepts and operators are considered.

The core idea is to generate a plan on an abstract level where only a few instances and operators exist. Then each step of this plan has to be recursively refined until the maximal level of detail is reached for all operators. In case a plan step cannot be refined on a more-detailed level, a back-tracking procedure is required that is described in subsection 7.2. This hierarchical divide and conquer approach ideally scales logarithmically with the number of required plan steps. Nevertheless, it is only beneficial with the right number of layers, if abstract plans sort out steps that are less promising on a more detailed level and if the downward refinement property holds often, thus most steps can be refined later on. In turn this requires the models of the concepts and operators to be suitably structured. Determining such models is a hard task on its own. However, the set-theoretic approach of the introduced formal models for concepts and operators helps to manually or automatically define them.

6.1 Parent Child Mapping

A central aspect of hierarchical planning is to refine an operator by posing a new planning task on a more-detailed level. This mainly corresponds to a selec-

tion of suitable operators. Thus, we introduce the set-valued parent child mapping $\mathcal{F}_{\text{PC}} : P \rightrightarrows P$ such that $\mathcal{F}_{\text{PC}}(\pi_i) := \mathcal{F}_{\text{PC}}(\{\pi_i\}) = X_{\text{op}}$.

Currently, we assume that such a mapping is provided and engineered during modeling of operators. For some domains an algorithmic extraction of that mapping just from the sets of concepts and operators was successfully tested. For general domains, however, this is a problem to be addressed in future research.

In the following we introduce two graphs for a formal description of dependencies between plans on different levels of abstraction: the *layer graph* $\mathcal{G}_{\text{LG}}$ addresses the hierarchical dependencies between plan steps and planning tasks, and the *extended planner state graph* $\mathcal{G}_{\text{EPS}}$ represents temporal dependencies. These two combined with the previously described plan family, which captures the causal dependencies between instances and operators, could be represented in a single multi-layer graph. In this paper we stick to a presentation with separate graphs though as it is easier to understand.

6.2 Layer Graph

Since our planning approach does not assume a fixed number of hierarchy levels, each planning task PT defines some *layer*. Note that consequently the number of layers can vary during the planning process.

We introduce the (directed) bi-partite layer graph $\mathcal{G}_{\text{LG}}(V,E)$ to capture dependencies between planning states and layers, cf. Figure 6. Each node within the first set V_l corresponds to a planning task. As described in subsection 4.3, several planner states result during (single-level) planning for a given planning task. Each of these states is represented in the layer graph by a node in the set V_p. Note that the two sets V_l and V_p are disjoint.

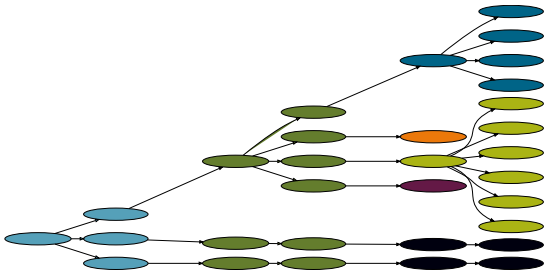

Figure 6: Part of the layer graph for the box-lid-example (subsection 8.2). The node colors match the temporal order during planning, compare Figure 8.

The set of vertices E represents dependencies of two kinds, which are discriminated by the direction relative to layer nodes. The first type connects planner state nodes obtained during planning with their respective layer nodes which represent planning tasks, i.e., $\forall e = (v_1, v_2) \in E,\ v_1 \in V_l :\ v_2 \in V_p$. The second type connects a planner state to exactly one layer node if the layer node specifies the planning task to refine the operator that led to the planner state (cf. subsection 4.3), i.e., $\forall e = (v_1, v_2) \in E,\ v_1 \in V_p :\ v_2 \in V_l$ and $\forall v_1 \in V_p :\ |\{e \mid e = (v_1, v_2) \in E\}| \leq 1$.

Consequently, the layer graph $\mathcal{G}_{\text{LG}}$ is tree-structured and captures the hierarchical refinements of plans. For later references, the mapping of each planning state to its corresponding layer is given by: $\mathcal{F}_{\text{PS,L}} : V_p \to V_l$ with $\mathcal{F}_{\text{PS,L}}(p) := l$ such that $(l, p) \in E$.

6.3 Blacklist

The planning task that refines a plan step, has a reduced set of instances for two reasons. First, the computational complexity is reduced, as less combinations are possible with fewer instances. Second, this assures that the plan is in accordance with the higher-level plan. This is particularly important for domains that are persistent and thus can't be set to an arbitrary state (e.g. real-world execution).

For example, assume that one out of two boxes has to be moved to a different working surface. In the coarse plan the smaller box is picked and then transported. If this pick is refined, only the smaller box is available. Otherwise, the more detailed planner could choose the bigger box for this task and an inconsistency between the plans on the different levels of abstraction would result.

The idea is to set all instances on a black list X_{Black} that belong to a concept corresponding to an operator input on the more abstract level. Let the coarse planning task $PT(X^{\text{c}}_{\text{init}}, X^{\text{c}}_{\text{goal}}, X^{\text{c}}_{\text{op}})$ and a corresponding plan be given as a plan family (V, E, E^{con}):

$$\forall v_1 \in V_B, v_2 \in V_\pi, (v_1, v_2) \in E :\ v_1 \in X_{\text{Black}}$$

and

$$\forall v_1 \in V_B, v_2 \in X_{\text{Black}}, v_1, v_2 \in C :\ v_1 \in X_{\text{Black}}.$$

Let $v_2 \in V_\pi$ be one operator in the plan that has no preceding operators, i.e., $\forall v_1 \in V_B$ with $(v_1, v_2) \in E$ holds $v_1 \in X^{\text{c}}_{\text{init}}$, then the corresponding planning task for refinement is:

$$PT_{v_2}\big(\quad X^{\text{c}}_{\text{init}} \setminus X_{\text{Black}} \cup \{v_1 \mid (v_1, v_2) \in E\},\ \{v_3 \mid (v_2, v_3) \in E\},\ \mathcal{F}_{\text{PC}}(\pi_{v_2})\big).$$

If the operator has predecessors, the set of initial instances has to be replaced by resulting instances of the refinement of that preceding operator. In order to formally capture these dependencies, we have to extend the planner state graph $\mathcal{G}_{\text{PS}}$ of the single-level planning.

6.4 Extended Planner State Graph

Hierarchical planning requires encoding of temporal relations between instances of different layers. Therefore, we extend the planner state graph $\mathcal{G}_{\text{PS}}$ and discuss the related blacklists.

A single-level plan that resulted from a refinement step, is defined by the sequence of planning states in the $\mathcal{G}_{\text{PS}}$ and a blacklist $X^{\text{c}}_{\text{Black}}$. The planning task corresponds to the first planning state p_0 in that sequence as it maps to the layer node in the layer graph. Solving this task results again in a separate planner state graph. However, the goal is to combine all planner state graphs in one large graph, the extended planner state graph $\mathcal{G}_{\text{EPS}}$, for unified data handling during hierarchical planning. Thus, we extend the node information of the (one-layer) planner state graphs by their layer information given by $\mathcal{F}_{\text{PS,L}}$ to avoid mixing of layers.

Assuming that a refining plan for the planner state p_i is found, the definition of the planning task corresponding to the layer $l_{i+1} = \mathcal{F}_{\text{PS,L}}(p_{i+1})$ has to use the resulting instances of that plan. Therefore, we add one additional node to $\mathcal{G}_{\text{EPS}}(V,E)$ for each planner state that reached the goal set X_{goal}. Let p_j be such a planning state and v be the corresponding additional node, then a matching edge is added $(p_j, v) \in E$. The set of instances corresponding to the node v contains the union of instances from the preceding node $X(p_j)$ and the blacklisted instances in the corresponding layer $X_{\text{Black}}(l)$: $X(v) := X(p_j) \cup X_{\text{Black}}(l)$.

In order to refine the next planner state of the coarse plan, one of the plans refining the preceding node has to be selected, leading to the node with blacklisted instances v. Let the planner state p_{i+1} correspond to the operator π in the plan family $\mathcal{G}_{\text{PF}}$. Now, the set of blacklisted instances for the next layer can be specified:

$$X_{\text{Black}}(l_{i+1}) := (X(v) \cap X^{\text{c}}_{\text{Black}}) \setminus \{v_k \mid (v_l, \pi) \in E_{\mathcal{G}_{\text{PF}}}, (v_k, v_l) \in F\}.$$

Then the set of initial instances for the next layer l_{i+1} results: $X_{\text{init}}(l_{i+1}) := X(v) \setminus X_{\text{Black}}(l_{i+1})$.

This means that for each plan refining the previous planner state p_i a new set of initial instances results and a corresponding new layer is generated. To also keep track of these dependencies, we add further edges to the extended planner state $\mathcal{G}_{\text{EPS}}$ linking the node v with the added blacklist elements to the planner state p corresponding to the initial instances $X_{\text{init}}(l_{i+1})$.

7 HIERARCHICAL PLANNING

The goal of hierarchical planning is to find a sequence of planner states in $\mathcal{G}_{\text{EPS}}$ such that no corresponding operator has a further refinement. For this task, the graphs of the plan family $\mathcal{G}_{\text{PF}}$, the extended planner state $\mathcal{G}_{\text{EPS}}$ and the layers $\mathcal{G}_{\text{LG}}$ are used.

The basic idea to reduce the fanout and thus counter the curse of dimensionality, is to use prior knowledge that is encoded by abstractions, like visualized in Figure 7. Once a plan was found on an abstract level, each step in this plan defines a new planning task that can either be refined by a single, more detailed operator or poses a new planning task on its own.

However, there is freedom in choosing the order of refining planning states on different levels of abstraction. The performance of such a refinement strategy depends highly on the domain at hand. If, for instance, the downward refinement property holds, one can always refine all operators to the most detailed level before proceeding to the next step. On the other hand, if some refinements repeatedly fail even on a rather coarse level, it is suboptimal to refine the first steps to maximal detail before checking the refinements of later steps on coarser levels.

If a refinement fails previous planning steps need to be adapted accordingly. Such backtracking methods are the key to real-world scenarios in which the coarser level cannot consider all details.

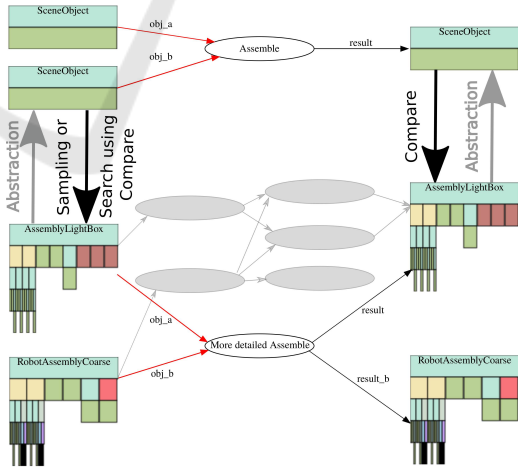

Figure 7: Visualization of the basic planning idea to reduce the fanout. An abstracted planning task is obtained based on the original instances and goals (arrows pointing upwards). New tasks in the refined level are posed by each step in the coarse plan. Either a primitive, single operator plan exists (white ellipse) or a planner has to be used to find a suitable plan (gray ellipses). The compare function is used to ensure that the abstractions of available instances and goals comply with the instances of the course plan.

7.1 Hierarchical Strategies

The two most basic strategies follow the spirit of depth- and breadth-first search and define the extrema of possible approaches. Naturally, intermediate strategies, possibly including heuristics, could increase planning performance for specific domains, however this is a topic for further research.

The breadth-first strategy refines all operators of one level once.

7.2 Backtracking

Only the downward refinement property could assure that all plan steps can be refined until the finest level is reached. However, this property cannot be guaranteed for all real-world problems. Therefore, the hierarchical strategies have to be complemented by backtracking procedures.

Such a backtracking method triggers re-planning on former layers. Since the overall problem is assumed to be solvable, there exists a set of layers and some selection of corresponding plans such that all operators are fully refined.

In most cases it is reasonable to choose the backtracking procedure that inverts the hierarchical (forward) search strategy. In the case of a breath-first search, tasks that are predecessors within the extended planner state $\mathcal{G}_{\text{EPS}}$ of the failing step are considered first. Only, if there exist no alternative plans in these layers that lead to a succeeding refinement, the more abstract layer, i.e., an ancestor in $\mathcal{G}_{\text{LG}}$ of the node given by $\mathcal{F}_{\text{PS, L}}$, is addressed.

Re-planning means that the original goals are blacklisted and the (single-level) planner continues its search then.

8 EXAMPLE: AUTONOMOUS ASSEMBLY CELL

Our example setup consists of two robot arms with seven degrees of freedom each, cf. Figure 1. The workspaces of the arms have considerable overlap to enable cooperation and wrist-mounted cameras to perceive the environment. To demonstrate the capabilities of our approach, two different use-cases are analyzed. The first one considers a setup for the assembly of control cabinets: circuit breakers and other components have to be automatically assembled on a hat-rail. The focus of this example is the scalability with an increasing number of necessary steps. The second use-case is simpler, thus we can discuss the backtracking algorithm on the real graph structures.

The difficulty for both scenarios is the combination of high dimensional continuous properties (14 degrees of freedom for the robotic arms and up to five times three dimensions for the objects) and a even higher number of discrete options. Those discrete options not only result from the actions directly visible on the robot, such as perception, grasping, clicking, pushing or releasing, but also from different sequences of varying operators that calculate parameterizations such as grasp configurations, view poses or initialize the hardware. The operators in our models are kept rather atomic, to ensure flexibility, easy extension and composability as demanded for autonomous manufacturing systems. Additionally, these requirements result in subsection 4.3 in the definition of planner states being the combinations of available instances. In consequence, inhomogeneous planner states have to be considered during planning in opposition to most motion planner that assume a homogeneous state for performance reasons.

8.1 Hat-rail Assembly

For the hat-rail example four components and the hat-rail are initially placed in the working area of the robot arm. The precise configuration of the parts is not known in the beginning, thus a perception step is required. The goal of the task is specified on an abstract level, which only states the final position of the components on the rail.

To actually control the robotic system the sequence of the four assembly operations of the coarsest plan have to be refined three times. The first refinement adds further symbolic properties to the first domain. Then, the second refinement actually simulates all necessary robot motions assuring collision-free paths and reachability. Finally, the actual control of the robotic hardware is the most-detailed layer.

In this experimental setup we use the breadth-first strategy to hierarchically refine the coarse plan. Our numerical result (cf. Table 1) shows that the hierarchical approach scales almost linearly in the number of pieces to be assembled, which is the best we can expect without (automatically generated) additional abstraction levels for longer tasks. Note, that those planning times include all execution times of called operations. This includes computation and simulation of controllers for both robot arms considering the full nonlinear robot dynamics ($\sim$ 0.3 [s] per motion task). A considerable offset for the planning times is caused by the loading and construction of the geometric scene ($\sim$ 1.2 [s] once). For comparison we used a simple breadth first search on the same problem. For a single piece that is assembled, this algorithm is one

order of magnitude slower than our hierarchical approach. For two pieces it took at least three orders of magnitude longer than the hierarchical algorithm and did not find any solution before we ran out of memory. Arguably a faster algorithm can be found to solve this problem, which will be faster at least for the single piece assembly. However, the interesting point is how the solution will scale with increasing numbers of pieces. One can expect, that non hierarchical approaches run into the curse of dimensionality.

Table 1: Planning times t, and number of nodes in the plan family and planner state over the number of pieces to assemble. The first block states the results of the hierarchical planner and the second block a single-level approach which highlights the complexity of the task.

	1 pc.	2 pcs.	3 pcs.	4 pcs.
t [s]	3.7	5.5	7.9	10.5
$\lvert\mathcal{G}_{\text{EPS}}\rvert$	159	330	502	688
$\lvert\mathcal{G}_{\text{PF}}\rvert$	89	200	314	439
t [s]	79.2	$> 7 \cdot 10^3$	*	*
$\lvert\mathcal{G}_{\text{EPS}}\rvert$	32,203	$> 2 \cdot 10^6$	*	*
$\lvert\mathcal{G}_{\text{PF}}\rvert$	981	$> 4 \cdot 10^3$	*	*
|Plan|	16	36	56	76

These results demonstrate that the hierarchy successfully handles the curse of dimensionality. Note that the models for this example fulfill the downward refinement property, thus no backtracking is needed, which explains the almost perfect results.

8.2 Box Lid Assembly

In order to show the backtracking features of our planning approach, we consider a simple assembly process, in which a lid has to be placed on top of a box.

The coarse level has to choose which robot arm should pick the lid and assemble it on the box. However, the box is only reachable by one of the arms, as it is placed outside of the workspace overlap of the two robots. If the coarse level now chooses the wrong arm to do this task, there exists no valid refinement on the sub-symbolic level. Therefore, backtracking is needed. Note that only the second step of the coarse plan cannot be refined, since both arms can pick the lid, but only one can put it on the box.

Consequently, the backtracking procedure tries to find an alternative plan in the temporal preceding step first and then chooses a new plan on the coarser layer. See Figure 8 for a visualization of the planner state of this example.

The orange area shows the part where no plan that refines the second step in the coarse plan is found. The backtracking approach then tries to find an alternative plan in the preceding layer (red area). However, there exists no plan to pick the lid in a way that allows assembly, with the left arm, as the box is out of reach. Thus, the backtracking goes to the more abstract level (dark green area) and selects an alternative plan that uses the right arm instead of the left one. Here, the backtracking stops, and the breadth-first search is restarted. In this second coarse plan both actions, i.e., picking and assembling, can be refined successfully (green and petrol area, respectively).

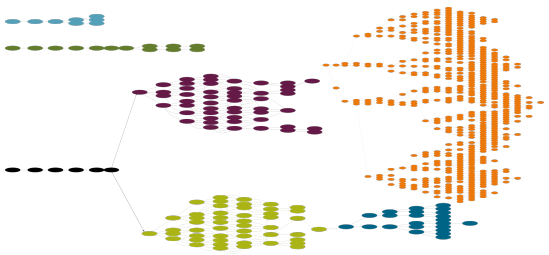

Figure 8: The planner state $\mathcal{G}_{\text{PS}}$ with backtracking. No refined plan is found (orange area), therefore backtracking considers the preceding layer (red area) first. A second plan on the coarser level (dark green area) can then be successfully refined (green and petrol area).

9 CONCLUSIONS

The approach to ground the declarative and procedural knowledge on set theory enables to flexibly reason about hierarchies. Based on these structures we introduce a hierarchical planner that allows to seamlessly traverse different abstraction levels and obliterate the differences between symbolic and sub-symbolic domains. This approach additionally allows for modeling robot and task description independently. This leads to highly flexible and composable systems. We discussed examples of autonomous production systems with real hardware that is controlled with this hierarchical approach. Our examples show that hierarchy can help to avoid the curse of dimensionality and backtracking allows to handle cases that do not have the downward refinement property.

REFERENCES

Bacchus, F. and Yang, Q. (1994). Downward refinement and the efficiency of hierarchical problem solving. *Ar-*

The presented research is financed by the TransFit project which is funded by the German Federal Ministry of Economics and Technology (BMWi), grant no. 50RA1701, 50RA1702, and 50RA1703.

tificial Intelligence, 71(1):43–100.

Bechon, P., Barbier, M., Infantes, G., Lesire, C., and Vidal, V. (2014). Hipop: Hierarchical partial-order planning. In *STAIRS*, pages 51–60.

Bercher, P., Höller, D., Behnke, G., and Biundo, S. (2016). More than a name? on implications of preconditions and effects of compound htn planning tasks. In *ECAI*, pages 225–233.

Bryan, A., Ko, J., Hu, S., and Koren, Y. (2007). Co-evolution of product families and assembly systems. *CIRP Annals*, 56(1):41–44.

Cashmore, M., Fox, M., Long, D., and Magazzeni, D. (2016). A compilation of the full PDDL+ language into SMT. In *AAAI Workshop: Planning for Hybrid Systems*.

Castillo, L., Fernández-Olivares, J., Garcia-Perez, O., and Palao, F. (2006). Efficiently handling temporal knowledge in an htn planner. In *ICAPS*, pages 63–72.

Castillo, L., Fernández-Olivares, J., and Gonzalez, A. (2003). Integrating hierarchical and conditional planning techniques into a software design process for automated manufacturing. In *ICAPS*.

Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. (2009). Semantic attachments for domain-independent planning systems. In *Conf. on Automated Planning and Scheduling*.

Erol, K., Hendler, J., and Nau, D. (1994). Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128.

Fernandez-Gonzalez, E., Williams, B., and Karpas, E. (2018). Scottyactivity: Mixed discrete-continuous planning with convex optimization. *Artificial Intelligence Research*, 62:579–664.

Fox, M. and Long, D. (2002). Pddl+: Modeling continuous time dependent effects. In *Int. NASA Workshop on Planning and Scheduling for Space*, volume 4.

Garrett, C., Lozano-Pérez, T., and Kaelbling, L. (2015). Ffrob: An efficient heuristic for task and motion planning. In *Algorithmic Foundations of Robotics XI*, pages 179–195. Springer.

Gateau, T., Lesire, C., and Barbier, M. (2013). Hidden: Cooperative plan execution and repair for heterogeneous robots in dynamic environments. In *IROS*, pages 4790–4795. IEEE.

Georgievski, I. and Aiello, M. (2015). Htn planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156.

Goldman, R. (2006). Durative planning in htns. In *ICAPS*, pages 382–385.

Helmert, M. (2006). The fast downward planning system. *Artificial Intelligence Research*, 26:191–246.

Hitomi, K. (2017). *Manufacturing Systems Engineering: A Unified Approach to Manufacturing Technology, Production Management and Industrial Economics*. Routledge.

Hu, S., Ko, J., Weyand, L., ElMaraghy, H., Lien, T., Koren, Y., Bley, H., Chryssolouris, G., Nasr, N., and Shpitalni, M. (2011). Assembly system design and operations for product variety. *CIRP Annals*, 60(2):715–733.

Kambhampati, S., Mali, A., and Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. In *AAAI/IAAI*, pages 882–888.

Kast, B., Albrecht, S., Feiten, W., and Zhang, J. (2019). Bridging the gap between semantics and control for industry 4.0 and autonomous production. In *CASE*. IEEE.

Kaufman, S., Wilson, R., Jones, R., Calton, T., and Ames, A. (1996). The archimedes 2 mechanical assembly planning system. In *ICRA*, volume 4, pages 3361–3368. IEEE.

Lozano-Pérez, T. and Kaelbling, L. (2014). A constraint-based method for solving sequential manipulation planning problems. In *IROS*, pages 3684–3691. IEEE.

Marthi, B., Russell, S., and Wolfe, J. (2008). Angelic hierarchical planning: Optimal and online algorithms. In *ICAPS*, pages 222–231.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). Pddl - the planning domain definition language. *The AIPS-98 Planning Competition Comitee*.

Molineaux, M., Klenk, M., and Aha, D. (2010). Planning in dynamic environments: extending htns with nonlinear continuous effects. In *Conf. on Artificial Intelligence*, pages 1115–1120. AAAI Press.

Nau, D., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *Artificial Intelligence Research*, 20:379–404.

Piotrowski, W., Fox, M., Long, D., Magazzeni, D., and Mercorio, F. (2016). Heuristic planning for hybrid systems. In *Conf. on Artificial Intelligence*, pages 4254–4255. AAAI Press.

Schattenberg, B. (2009). *Hybrid planning and scheduling*. PhD thesis, University of Ulm, Germany.

Schmitt, P., Neubauer, W., Feiten, W., Wurm, K., v. Wichert, G., and Burgard, W. (2017). Optimal, sampling-based manipulation planning. In *ICRA*, pages 3426–3432. IEEE.

Schmitz, S., Schluetter, M., and Epple, U. (2009). Automation of automation — definition, components and challenges. In *Conf. on Emerging Technologies & Factory Automation*. IEEE.

Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S., and Abbeel, P. (2014). Combined task and motion planning through an extensible planner-independent interface layer. In *ICRA*, pages 639–646. IEEE.

Thomas, U. and Wahl, F. (2010). Assembly planning and task planning—two prerequisites for automated robot programming. In *Robotic Systems for Handling and Assembly*, pages 333–354. Springer.

Toussaint, M. (2015). Logic-geometric programming: An optimization-based approach to combined task and motion planning. In *IJCAI*, pages 1930–1936.

Young, R. M., Pollack, M., and Moore, J. (1994). Decomposition and causality in partial-order planning. In *AIPS*, pages 188–194.