CSP-Based Integrated Task & Motion Planning for Assembly Robots

Jan Kristof Behrens[1] Ralph Lange[2] Michael Beetz[3]

***Abstract*— To increase the flexibility in industrial manufacturing, two-arm robots that may quickly take over typical human assembly tasks have been developed. One important issue for the easy and quick preparation for new assembly tasks is integrated task and motion planning for such robots. We propose a CSP-based planning approach based on a general constraint model for assembly tasks. Our model combines both, the space and collision-free motion aspects as well as a general graph-based approach to represent the assembly goal and the possible assembly actions, in one formalism. We explain the implementation of our model in the constraint programming language MiniZinc and present several optimizations to reduce the computing time of common solvers significantly.**

I. Introduction

Customer-driven product variants and small lots sizes are a clear trend in manufacturing. This trend causes a high effort in today's shop floors. Work tasks may change from day to day and the volume of work varies frequently. Two-arm robots with human-like working spaces (e.g., ABB YuMI) that may take over typical assembly tasks (including pick-and-place tasks) have been developed to meet this challenge for flexibility.

A crucial issue for the success of such robots is that they can easily and quickly be prepared for a new task. Classical industrial robot programming is far too time-consuming and very inflexible. Such easy, quick set up raises a number of research questions for example on perception, dexterous manipulation, safety, skill learning, and task and motion planning.

In this paper, we focus on integrated task and motion planning for industrial assembly tasks for two-arm robots. As a running example, we consider a workstation in an assembly line for automotive window wiper motors, illustrated in Figure 1. At this workstation, the rotors are already inserted into workpiece holders (A) on a conveyor system, arriving in groups of five. The stators with the brushes and the electric interfaces are delivered in plastic blisters (B) from a supplier. A stator cannot be mounted directly on a rotor, but only after a cone-shaped tool has been placed on the motor shaft. After the assembly of stator, this tool has to be removed. Once five stators have been assembled on the five rotors, the workpiece holders move to the next workstation. The label (C) marks the home position of the cone-shaped tool available.

[1]Jan Kristof Behrens is in an industrial PhD program with Robert Bosch GmbH and is supervised by Prof.Michael Beetz from the Institute for Artificial Intelligence of University of Bremen Jan.Behrens@de.Bosch.com

[2]Ralph Lange is with Robert Bosch GmbH, Corporate Sector Research and Advance Engineering ralph.lange@de.bosch.com

[3] Prof. Michael Beetz is head of the Institute for Artificial Intelligence of the University Bremen

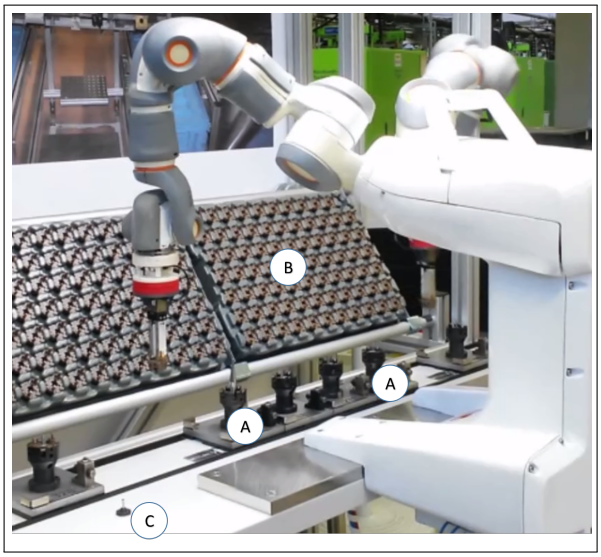

Fig. 1. Two-arm robot on our test assembly line station

This example illustrates three important characteristics of industrial assembly tasks with regard to task and motion planning:

1) The assembly actions are not strictly ordered but their order may be optimized with regard to time.
2) The two manipulators offer a huge potential for parallelization, but require comprehensive motion planning – in particular due to the narrow spatial layout of the workstations (for monetary and ergonomic reasons).
3) The workstations constitute a rather controlled environment (compared to domestic homes). The arms have to reach a finite, predefined set of poses only and in many scenarios the obstacles in the work space are statically known from 3D scans and/or CAD models.

In the past, a number of approaches for integrated task and motion planning have been proposed. According to [1] they can be classified into (1.) motion planning guided by task planning, (2.) task planning querying motion planning and (3.) task planning first, then motion planning (iterated). The above three characteristics show the strong interplay between task and motion planning in industrial assembly tasks. Therefore, a planning approach that treats both kinds of planning in one integrated formalism and by one single search procedure is highly desirable. In doing so, this approach exploits the fact that the poses are known in advance, which allows to pre-compute the configurations for each pose and for each manipulator as well as the space that is swept when moving from one to another pose.

A very promising approach is formulating such highly integrated task and motion planning problems as constraint optimization problems (COP), which we will follow in this paper. Formalizing planning problems as constraint optimization problems has been proposed in several recent works [2], but such modeling and solving is always highly domain-dependent. Our contributions are:

- A general constraint model that combines the representation of a two-arm robot, its workspace, the motion of its manipulators and the resulting occupancy of the work space ...
- ... in one formalism together with a general graph-based approach for representing an assembly goal, the possible assembly actions and the intermediate states. The graph-based approach is derived from an ontology-based domain model.
- The implementation of this model in MiniZinc, which is a high-level, solver-independent constraint modeling language.
- Hints and experiences learned about the MiniZinc implementation and user-defined search strategies for prevalent MiniZinc solvers to reduce computational cost.

The remainder of the paper is structured as follows: In Section II we discuss related work. In Section III we describe the general modeling of the planning problem as constraint optimization problem, before we explain the formulation in MiniZinc in Section IV. Finally, the paper is concluded in Section V with a summary and an outlook to future work.

II. Related Work

In many robotic scenarios, task and motion planning has to be closely integrated because decisions on task level might render a motion planning problem infeasible or a motion planner chooses a solution, which conflicts with a later action. Bidot et al. divide the integrated task and motion planning approaches into three groups [1]: (1.) motion planning guided by task planning, (2.) task planning querying motion planning, and (3.) task planning first, then motion planning iterated.

This work is closest to the second group regarding the degree of integration of information about the continuous part of the state space into the discrete planner. A sequence of actions is planned by a discrete/symbolic planner and if a geometric precondition or effect is involved, a geometric reasoner or motion planner is called. Dornhege et al. propose the use of semantic attachments, which is essentially calling external reasoning procedures to use the results on the symbolic level [3]. In [4] they propose a subsumption caching and lazy evaluation technique to postpone geometric calculations and save the result for later reuse. This is an interesting addition to their previous work on semantic attachments as they stated before that a bottom up approach is prohibitive because of excessive space and time consumption whereas a top down approach (first task planning, then refinement with motion planners) might often fail due to geometric preconditions not considered by the symbolic planner. We believe that in our domain, caching or precomputation is a feasible approach as we have to consider a small number of object placements and robot configurations only.

Gaschler et al. emphasize the importance of volumes as intermediate representation in robotics [5]. They use a decomposition into simple convex volumes of robot shapes and swept volumes to make the geometric reasoning invoked by a task planner tractable. We rely on a similar representation of space. However, we integrate task and motion planning closer as we model the volumes and the possible collisions in a single CSP-based planning problem.

In [6], Ghallab et al. describe a general approach to formulate task panning problems and hybrid problems as constraint optimization problems. Similar approaches are presented by Bartak et al. in [2] and for the UPMurphy planner for which Della Penna et al. formulate hybrid planning as explicit model checking [7]. Mansouri et al. consider the hybrid problem of mining vehicles with motion constraints and 2D footprint [8] using the meta-CSP framework [9]. Although they did not apply the meta-CSP approach to robotic manipulation tasks, it might be a promising target formulation for addressing the temporal aspects of integrated task and motion planning for assembly robotics. In this paper, we focus on the modeling of the robot and the assembly domain logic.

III. CSP-based Model for Integrated Task and Motion Planning

In this section, we present our generic model for integrated task and motion planning by constraint optimization for assembly tasks. A constraint optimization problem is specified by a CSP and a cost function. A CSP, in turn, is generally specified by a triple (X, D, C), where X is a set of variables, D a set of domains (one for each variable), and C a set of constraints.

For our constraint model, we first introduce *types*. A type represents a physical or virtual concept (class) of the robot or the workstation. Examples are the robot's arms (Arms) and the relevant end-effector poses (Poses). Each type is a finite set.

In our model, a variable $x_i \in X$ is not a scalar, but a time-dependent function to a type (or a union of types). For example, the configuration of an arm r at time t is given by the variable $\mathrm{loc}(r, t)$. Hence, the domain $d_i \in D$ is the set of all possible function graphs for x_i. In doing so, we discretize time. This is a common approach in CSP-based planning, but requires some strong assumptions – here in particular on the movement between poses. We will explain these assumptions together with the corresponding variables.

The values of all variables at a given time t represent a state in the planning problem, i.e. the variables describe the states in a factored way (just as in classical planning with PDDL or STRIPS).

We distinguish four categories of constraints, inspired by the classification in [10]:

- *State constraints* are time-invariant and statically reduce the overall set of states to a subset of allowed states.

- *Initial state constraints* refer to the states at $t = 0$ only and make the planning problem congruent with the initial, perceived state of the world.
- *Goal constraints* specify the allowed final states.
- *Dynamic constraints* connect variables of successor states to ensure only valid actions/changes occur between states.

To simplify the formulation of such constraints we introduce *rigid relations* as "helper-structures". Rigid relations specify fixed, given relations between two or more types and can thus be used as logical predicates. Often they are even functions such as the reaching relation, which specifies the end-effector poses for the relevant configurations of an arm.

Next, we describe the most relevant types, variables, rigid relations and constraints for collision-free motion of a two-arm robot at a workstation. Then, we present our approach for modeling an assembly task. At the end of this section, we discuss the definition of a cost function for optimal planning regarding time, considering non-intuitive effects by CSP-based planning. For readability, we omit the time in the variable definitions.

A. Space Representation and Collision-Free Motion

We model the functional components of the robot, which are the manipulators here, by the type $\mathsf{Arms} = \{r_{\text{left}}, r_{\text{right}}\}$. The type Cells provides a 3D decomposition of the robot's working space into small cuboids and the type Poses specifies the poses in the working space that are relevant for the use-case. For each pose, we precompute a set of joint configurations for each arm in reach to that pose and store them in the type Configurations. The rigid relation

$$\mathsf{applicable} \subseteq \mathsf{Configurations} \times \mathsf{Arms}$$

specifies which configurations belongs to which arm. Similarly, the rigid relation blocking specifies the cells that are blocked by each configuration. Then, we introduce the variable $\mathrm{loc} : \mathsf{Arms} \to \mathsf{Configurations}$ to model the manipulators' configurations over time.

The variable $\mathrm{occupant} : \mathsf{Cells} \to \mathsf{Arms} \cup \{empty\}$ defines each cell as renewable resource of capacity one for the manipulators. The state constraint

$$\forall c \in \mathsf{Cells} : \forall r \in \mathsf{Arms} : \mathsf{blocking}(c, \mathrm{loc}(r)) \leftrightarrow \mathrm{occupant}(c) = r$$

ensures that the occupancy of the cells is consistent with the blocked cells by the manipulators' configurations. This constraint and the capacity limitation in occupant together prevent collisions between the two manipulators. By extending occupant to other types, other dynamic obstacles can be integrated easily.

The movements between Configurations are constrained by the rigid relation connected. We allow movements only between those configurations of an arm that take less than the time discretization. By refining the time and space discretization and introducing additional poses and configurations, the optimality with regard to motion planning may be traded off against the computational cost for planning.

The rigid relation sweptBlocking specifies the cells that are blocked when moving from one configuration to another configuration and is used by a dynamic constraint to prevent collisions during the movement from a configuration c_i to c_j. In order to avoid the explicit representation of sweptBlocking, the union of blocking at c_i and c_j may be used, depending on the spatial discretization.

B. Representation of Parts and Assembly Graphs

We first developed a domain ontology (see Fig. 2) for representing the states during assembly processes. From this ontology, we derived the following types, variables and rigid relations.

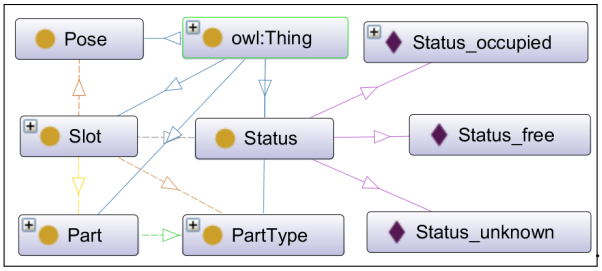

Fig. 2. Excerpt from ontology describing Parts and Slots

Parts and Slots. The type Parts denotes the range of manipulatable things at the workstation. Parts have a PartType encoded in the rigid relation $\mathsf{ofPartType} \subseteq \mathsf{Parts} \times \mathsf{PartTypes}$.

We further introduce Slots as concept for potential locations of parts. Examples for slots are workpiece holders, the manipulators' grippers, and the home positions of the tools. A slot s can hold one single part b, where $\mathsf{ofPartType}(b) \in \mathsf{ofSlotTypes}(s)$, where $\mathsf{ofSlotTypes} \subseteq \mathsf{Slots} \times \mathsf{PartTypes}$ lists all PartTypes the slot can hold. The capacity is implicitly enforced by having a single valued variable

$$\mathrm{slotOcc} : \mathsf{Slots} \to \mathsf{Parts} \cup \{empty\}$$

to represent the content of each slot at each point in time. Conversely, the variable

$$\mathrm{pos} : \mathsf{Parts} \to \mathsf{Arms} \cup \mathsf{Slots}$$

denotes the location of each part in terms of slots. The constraint

$$\forall b \in Parts : slotOcc(pos(b)) = b$$

makes the variables pos and slotOcc consistent.

We connect the assembly logic (i.e. the assembly states and possible assembly actions) and the robot and space representation by assigning a pose to each $s \in \mathsf{Slots}$ through the rigid functional relation.

$$\mathsf{situated} \subseteq \mathsf{Slots} \times \mathsf{Poses}$$

Grasping. Grasping is modeled implicitly. We post a (dynamic) constraint that $\mathrm{pos(b)}$ of a part b can change its value from time step t_i to $t_i + 1$ if and only if the slots have the same pose at time t_i. To model the capacity and type

constraints for the grippers, we introduce dedicated slots for them. However, instead of situated, we use the variable loc and the rigid relation reaching to determine the poses of the grippers' slots.

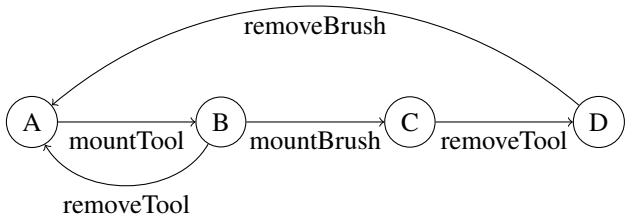

Fig. 3. Assembly graph for our example task

Assembly graph. The states during an assembly process and the possible assembly actions can be generally represented by a graph. Figure 3 depicts the states and actions of our running example:

(A) Workpiece empty
(B) Tool mounted on workpiece
(C) Part and tool mounted on workpiece
(D) Part mounted on workpiece

Note that some actions are possible, but usually not needed. Normally, the robot removes the tool to get from node (C) to (D). However, if one tool is not available, the other tool could be exchanged between the arms by placing it in a slot reachable for both arms.

Slot groups. The nodes in the assembly graph resemble the combinatorial states of the two slots on a workpiece holder. To be able to refer to multiple slots that logically belong together, we introduce the type SlotGroups. We refer to a SlotGroup for mainly two reasons: (1.) to formulate dynamic constraints on the state of the group to comply with the allowed assembly actions and (2.) to frame goal constraints regarding the assembly task.

The belonging of Slots to SlotGroups is specified by the rigid relation

$$\mathsf{slotGrouping} \subseteq \mathsf{SlotGroups} \times \mathsf{Slots}.$$

Each SlotGroup has a SlotGroupType denoting its class specified by the rigid relation

$$\mathsf{ofGroupType} \subseteq \mathsf{SlotGroups} \times \mathsf{SlotGroupTypes}.$$

We enumerate the allowed states $gs \in \mathsf{GroupStates}$ for $g_i \in \mathsf{SlotGroups}$ (state constraint) as well as the allowed successor states for each group (dynamic constraint) to model the assembly graph.

Assembly Goal. As the robot should accomplish an assembly task, we introduce a goal constraint in terms of the variables introduced in this section, which requires all workpieces to be finished in the last time step.

$$\forall g \in \mathsf{SlotGroups} : \mathsf{grouptype}(g, \mathsf{workpiece}) \rightarrow \mathrm{groupState}(g) = \mathsf{finState}(\mathsf{grouptype}(g))$$

with the rigid relation

$$\mathsf{finstate} \subseteq \mathsf{SlotGroupTypes} \times \mathsf{GroupStates}$$

marking the final state per SlotGroupTypes.

C. Planning by Constraint Optimization

A CSP according the generic model in the previous subsections already allows to compute a valid plan for an assembly task using a CSP solver. However, such plan will likely not be optimal with regard to execution time due to the lack of an optimization goal and a corresponding search strategy.

In our case, the executing time is represented by the plan length k. This length is defined as the number of states (at times $t_0, t_1, \ldots, t_{k-1}$) in this plan.

Thus, we search for a valid plan with minimal length k. Such $k_{\min}$ might still not be of satisfactory quality – for example if one manipulator performs useless movements while waiting for the other manipulator. Therefore, we define a cost function which takes the plan length as well as the sum of individual movements of the arms into account:

$$\mathrm{cost} = |\mathsf{Arms}| \cdot k^2 + \sum_{t=1}^{k-1} \sum_{r \in \mathsf{Arms}} \xi(r,t)$$

where $\xi(r,t)$ detects the individual movements in the plan by

$$\xi(r,t) = \begin{cases} 0 & \text{if } \mathrm{loc}(r,t) = \mathrm{loc}(r,t-1) \\ 1 & \text{else} \end{cases}$$

In this formula, the term $|\mathsf{Arms}| \cdot k^2$ ensures that the plan length dominates over the sum of individual movements.

IV. Implementation in MiniZinc

MiniZinc is a medium-high-level constraint programming language, which enables modeling very close to the mathematical formulation. In particular, we can use operators like $\forall, \exists, \neg, \wedge, \vee, =, \neq, \leq, \geq, >, <, \rightarrow, \leftarrow$ and $\leftrightarrow$. Furthermore, MiniZinc provides a catalogue of global constraint types, which encapsulate common constraint patterns such as *alldifferent* or *cumulative*.

The definition of a MiniZinc problem consists of two kinds of files. Modelfiles (mzn) hold the definitions of sets, parameters, variables and constraints to define a problem domain while datafiles (dzn) hold the problem instance dependent values for parameters.

MiniZinc implementations are not read directly by corresponding solvers, but such solvers take the lower-level input language FlatZinc. The MiniZinc-to-FlatZinc compiler takes care of the optimized translation to FlatZinc by unrolling the constraints and substitution of common subexpressions.

Next, we first explain the translation of our CSP model for integrated task and motion planning to MiniZinc. Then, we discuss several optimizations and hints for the MiniZinc implementation.

A. Translation of Modeling Concepts

First, the types are translated into named disjunct integer sets, i.e. each type member is represented by a unique integer in MiniZinc. Then, for each variable we create an array to represent its value over time. For example, the variable loc is implemented as

```
array[steps,arms] of var confs: loc;
```

and thus results in a two-dimensional array of MiniZinc variables – one row for each time step and a column for every manipulator in the type Arms. By including our initial state constraints and goal constraints, we make the MiniZinc implementation congruent with our CSP model. For example, with

```
constraint
  loc[steps[1],arms[1]] == confs[1] /\
  loc[steps[1],arms[2]] == confs[3];
```

we determine the configurations of the arms at the first time step. The state constraints (see the examples below) and dynamic constraints ensure that only valid and reachable states can be part of a solution. We save the rigid relations as a multi-dimensional arrays into the dzn file for the use in a constraints. Finally, we instruct the solver to prove satisfiability or search for an optimal solution regarding a goal function. To validate a solution, we generate a sequence

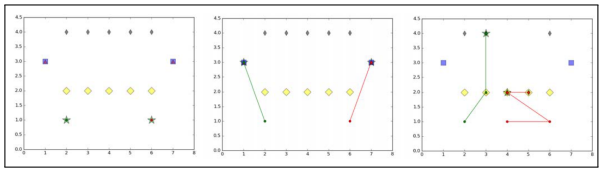

Fig. 4. State snapshots visualization for generated plans

of images as visualization (see Fig. 4). Another helpful approach in testing the constraint model and its MiniZinc implementation is to specify test cases in the form of known valid or invalid solutions. In particular, this allows to test whether the constraints are too tight or loose.

Our model represents various aspects using unary resources. An important example are the Slots. To ensure that the capacity of such a resource is not violated, we introduce a variable for each resource and constrain it to have the value of the occupying object. In the case of a slot, this is the variable $slotOcc(s) \in Parts \cup \{empty\}$. Then, for consistency, we connect the slot with the respective occupying part by constraining the dual variable pos, to point to each other (see lines 4-5 in Fig. 5). This has to hold if and only if the slot is occupied, i.e. $slotOcc \neq 0$.

By extending this constraint, we can help the solver in the constraint propagation step to prune the search space more efficiently. Therefore, we add the redundant constraint that no part may be positioned in a slot iff $slotOcc = 0$. We do this by negation the global constraint exists (see lines 7-9 in Fig. 5). The third part of this constraint comes from the

```
1 constraint
2   forall(t in steps)(
3     forall(s in slots)(
4         (slotOcc[t,s] != 0 <->
5            pos[t,slotOcc[t,s]] == s)
6         /\
7         (slotOcc[t,s] == 0 <->
8            not exists(p in parts)
9              (pos[t,p] == s))
10    )
11    /\
12    alldifferent_except_0(row(slotOcc,t))
13  );
```

Fig. 5. MiniZinc constraint: Slot capacity

insight that every part can occupy one slot only.[1] Therefore, the value of all $SlotOcc$ variables must be pairwise different at each point in time or zero. We can express this nicely using the *alldifferent-except-0* constraint.

```
1 constraint
2   forall(t in steps)(
3     forall(p in parts)(
4         slotOcc[t,pos[t,p]] == p
5         \/
6         armSlotOcc[t,pos[t,p]] == p)
7     /\
8     alldifferent(row(pos,t))
9   );
```

Fig. 6. MiniZinc constraint: Part location

Conversely, we can formulate the constraint from the parts perspective. Parts have to have be either in a slot or a gripper, which in turn points to them (see lines 4-6 Fig. 6). As we made the assumption that we only have capacity-one-slots for parts in our domain, all the values of pos must be pairwise different for each time step, which we model using the *alldifferent* constraint.

The constraint in Figure 7 ensures that a part fits in a suitable slot only, which means that its PartType has to be in the list of PartTypes the slot can hold (see lines 4-5 in Fig. 7). Note that we use the rigid relations ofPartType and ofSlotTypes, as introduced in Section III-B.

B. Optimizing the MiniZinc Implementation

The time for computing an optimal plan by a CSP solver highly depends on the exact implementation of our model in MiniZinc. We discovered several optimizations and general hints.

Optimizing Constraints. As we saw in the previous Subsection, we can express the same constraint in orthogonal ways. For example the model of slots is a dual to the part

[1]Note that this does not apply to all kinds of occupancy: For example, a manipulator generally occupies multiple cells.

```
1 constraint
2   forall(t in steps)(
3     forall(p in parts)(
4      member(slots,pos[t,p]) ->
5       member(row(ofSlotTypes,pos[t,p]),
6              ofPartType[p])
7     )
8   );
```

Fig. 7. MiniZinc constraint: Slot- and PartType have to be consistent

model. By chaining them together, we help the solver to use the strengths of each implementation.

We introduce redundant constraints to provide additional hints to the solver. We can use any invariant of the problem for this, e.g. the insight that two parts cannot occupy the same slot. Nevertheless, the formulation of some of those invariants rely on costly operators like $\neg, \exists, \cup$ or $\rightarrow$, on which the solver has to take many decisions. As there is a trade-off between the additional information by such constraints and the overall implementation length and complexity, we tested multiple versions of such constraints before settling for one.

Cost Function. Common solvers (e.g. chuffed, gecode or choco) offer different options to search for a solution while minimizing the proposed cost function. The greedy option, with which the solver performs a binary search on cost often results in computing times over 30 minutes. To reduce the computational costs, we define a search strategy for the solver, i.e. we partly define the order of the variables the solver works on.

Our problem has the substructure of a combinatorial configuration problem and a scheduling problem. In a valid goal configuration each part has to be placed according to the final and state constraints, i.e. a part must be on each workpiece and the tools have to be removed. The scheduling problem consists of how to transform the initial state into a goal state using the manipulators . Therefore, we instruct the solver to search for a valid goal state before it solves the scheduling problem. By searching for plans of increasing length, we guarantee to find a plan of minimal length first. By then searching over an upper bound for the number of individual arm movements, we minimize those as well. In this way, we find an optimal plan regarding our cost function.

V. Conclusion and Future Work

Two-arm robots that may take over typical human assembly tasks have been developed to increase the flexibility in industrial manufacturing. For the easy and quick preparation for new assembly tasks, we proposed a CSP-based integrated task and motion planning approach based on a general constraint model. Our model uses a spatio-temporal discretization to ensure collision-free motion between predefined poses. An assembly goal and the possible assembly actions are represented by a graph-based approach, using three generic concepts named parts, slots and slot groups to describe the intermediate states. The proposed cost function ensures that a constraint solver computes not only a plan with minimal length but also prevents useless movements in case that a manipulator has to wait for the other one.

We implemented our model in the constraint programming language MiniZinc and presented several optimizations (e.g. by redundant constraints) to reduce the computing time of common MiniZinc solvers. We also explained how to further reduce the computing time by selecting a proper search strategy that first searches for a valid goal state before solving the logistics aspects of the planning problem.

In the future, as a next step to reach the overall goal to ease and accelerate the preparation for new assembly tasks, we will investigate how to integrate our approach with a robotic knowledge processing system such as KnowRob [11]. This integration shall allow to create instances of our constraint model - and thus to solve planning problems in assembly robotics from a comprehensive knowledge representation, without manual coding in MiniZinc.

References

[1] J. Bidot, L. Karlsson, F. Lagriffoul, and A. Saffiotti, "Geometric backtracking for combined task and motion planning in robotic systems," *Artificial Intelligence*, 2015.

[2] R. Barták, M. A. Salido, and F. Rossi, "Constraint satisfaction techniques in planning and scheduling," *Journal of Intelligent Manufacturing*, vol. 21, pp. 5–15, Nov. 2008.

[3] C. Dornhege, P. Eyerich, T. Keller, S. Trg, M. Brenner, and B. Nebel, "Semantic attachments for domain-independent planning systems," in *Towards service robots for everyday environments*, pp. 99–115, Springer, 2012.

[4] C. Dornhege, A. Hertle, and B. Nebel, "Lazy evaluation and subsumption caching for search-based integrated task and motion planning," in *IROS workshop on AI-based robotics*, 2013.

[5] A. Gaschler, R. P. A. Petrick, M. Giuliani, M. Rickert, and A. Knoll, "KVP: A knowledge of volumes approach to robot task planning," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 202–208, Nov. 2013.

[6] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[7] G. Della Penna, I. Benedetto, M. Daniele, and M. Fabio, "UPMurphi: a Tool for Universal Planning on PDDL+ Problems," pp. 106–113, AAAI Press, 2009.

[8] M. Mansouri, H. Andreasson, and F. Pecora, "HYBRID REASONING FOR MULTI-ROBOT DRILL PLANNING IN OPEN-PIT MINES," *Acta Polytechnica*, vol. 56, p. 47, Feb. 2016.

[9] S. Stock, M. Mansouri, F. Pecora, and J. Hertzberg, "Hierarchical hybrid planning in a mobile service robot," in *Proc. of the 38nd German Conf. on on Artificial Intelligence (KI)*, 2015.

[10] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*. New York: Cambridge University Press, Apr. 2010.

[11] M. Tenorth and M. Beetz, "KnowRob – Knowledge Processing for Autonomous Personal Robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4261–4266, 2009.