FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

General Purpose Task and Motion Planning for Human-Robot Teams

Nuno Resende da Costa

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Gil Gonçalves

Second Supervisor: Liliana Antão

March 30, 2022

Resumo

No ambiente industrial atual, a customização de produtos e flexibilidade dos processos têm passado a tomar um papel central. As equipas colaborativas entre operadores humanos e robôs, ou *human-robot teams*, emergiram para responder a esta realidade, combinando capacidades humanas e robóticas e adaptando-se a diferentes cenários de uma forma otimizada. Os mais recentes desenvolvimentos no domínio de planeamento de tarefas oferecem soluções para este problema apresentando formas de melhorar os processos industriais. No entanto, muitas vezes não é dada a devida atenção ao primeiro passo no planeamento de tarefas, a discretização e formalização das mesmas, que pode ter de ser parcialmente ou completamente manual devido ao seu potencial para elevada variabilidade e complexidade de soluções. Por outro lado, os planos de tarefas obtidos podem nem sempre se traduzir em soluções viáveis, devido a limitações do ambiente. Consequentemente, ter em consideração algum planeamento dos movimentos necessários para efetuar as operações e avaliar a sua viabilidade torna-se essencial.

Para combater este problema, é proposta uma *framework* para o planeamento de tarefas e movimento, que usa uma abordagem *bottom-up* para a definição e formalização das tarefas, baseada em regras e configurações obtidas por uma entrada que contém uma abstração do resultado final pretendido. Subsequentemente, são gerados grafos baseados nas diferentes formalizações obtidas, a partir dos quais planos de tarefas podem ser obtidos e de seguida escrutinados por um módulo de simulação e planmeanto dos movimentos robóticos. A saída final do programa deve conter os planos viáveis mais eficientes.

Esta abordagem foi testada usando um caso de estudo de assemblagem de mobília modular do Yale Social Robotics Laboratory, denominado HRC model set. Resultados foram obtidos de dois objetos propostos no caso de estudo, uma mesa e uma móvel de prateleiras, apresentando diferentes níveis de complexidade. Os principais resultados mostram que apesar de não ser tão robusto para tarefas mais complexas como a assemblagem do móvel de prateleiras, para cenários mais simples como a mesa o sistema é capaz de definir muitas estratégias diferentes para a montagem do objeto pretendido, resultando em planos apropriados para o operador e o robô no ambiente colaborativo.

Abstract

In the current industrial environment, product customization and process flexibility have taken a central role. Human-robot teams try to answer this demand by coupling human and robot skills, adapting to the different scenarios in an optimized way. Recent developments in task planning offer solutions for this problem by highlighting ways of improving industrial processes. However, the first step in task planning is many times overlooked: task's discretization and formalization. This phase of the process is mostly performed partially or completely manually due to its immense variability potential and complexity of solutions in some scenarios. Resulting task plans alone may not translate into feasible solutions, due to environment constraints. Consequently, motion planning is essential for the evaluation of the tasks' validity and for obtaining decent outcomes.

To combat this problem, a task-motion planning framework is proposed. The implementation uses a bottom-up approach for the definition and formalization of the task, based on constraints and configurations obtained from an input that holds an abstraction of the desired outcome. The results of this formalization can be seen as different strategies of assembling the desired object. Subsequently planning graphs are generated based on the different formalizations, where task plans can be obtained and scrutinized by a motion planning module that simulates the robotic movements. The output should include the most time-efficient viable plans.

This approach was tested using a furniture assembly case study, the Yale Social Robotics Laboratory HRC model set. Results were taken from two prototypical objects suggested by this case study, a table and a shelf, with different levels of complexity. The main results show that despite not being as robust for more complex tasks like the shelf assembly, for simple scenarios such as the table the system is able to lay out many different strategies to address the assembly of the object, resulting in suitable task plans for the operator and the robot in the collaborative environment.

Acknowledgements

First of all, I would like to thank my supervisors for giving me the opportunity to work on this project and dissertation, with a special note of gratitude to Liliana for her help, patience and positivity. It was a long journey, with a lot of highs and lows. I'd like to extend my gratitude toward everyone who has been a part of it, one way or the other, for all the support and lessons learned throughout this time. I could not finish off without expressing a feeling of eternal appreciation for all those who make you laugh at the best of times and always have a kind word for you in the worst of times. It was a without a doubt a pleasure.

Nuno Costa

“In this terrifying world, all we have are the connections we make.”

BoJack Horseman

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

3D	Three-Dimensional
API	Application Programming Interface
CC-HTN	Clique/Chain-Hierarchical Task Network
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
FF	Fast-Forward
HAN	Hierarchical Action Networks
HPN	Hierarchical Planning in the Now
HR	Human-Robot
HRC	Human-Robot Collaboration
HRI	Human-Robot Interaction
HRP	Humanoid Robotics Platform
HTN	Hierarchical Task Networks
IK	Inverse Kinematics
JSON	JavaScript Object Notation
LPL	Linear Planning Logic
OMPL	Open Motion Planning Library
PDDL	Planning Domain Definition Language
POMDP	Partially Observable Markov Decision Process
POP	Partial-Order Planning
RG2	Robot Gripper Two
ROS	Robot Operating System
STL	Standard Triangle Language
STRIPS	Stanford Research Institute Planning System
TM	Task-Motion
TMP	Task-Motion Planning
UR5	Universal Robots Five
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

The industry has suffered several revolutions that introduced many impactful technologies. In the third industrial revolution, robots started to emerge in factories, creating automated assembly lines that revolutionized the industry. In today's industrial environment, automation is an essential step toward success for any company. In a wide variety of sectors, countless functions that used to be executed by human operators are now almost entirely performed by robots. These machines have increasingly become more efficient and reliable and have proven a cost-effective replacement to improve productivity and profits. In the last decades, such robotic systems have been a growing research area with significant signs of progress.

Despite the inevitable trend towards using machines in all fields, with the fourth industrial revolution (Industry 4.0), the industrial reality is being transformed. This new reality is leading towards a more customer-oriented production, where the human operator is indispensable in executing specific roles that robots cannot yet perform. Humans can offer the process useful skills like the ability to improvise, to understand time delays, to manage unscheduled events, to plan sequences of tasks, and to react to errors. At the same time, robots are significantly more susceptible to fail to respond to unusual situations appropriately. Working on feasible improvements for the robots is extremely important to ensure the tasks are performed efficiently and securely.

Additionally, customer-oriented production entails high product variability, where some tasks may be too challenging to be fully achieved by robots or too expensive to be fully automated. With all these factors, there was a shift in robotics towards using Human-Robot teams to perform these tasks, taking advantage of the benefits each offer. The pair works in a complementary way to achieve a common goal, combining robotic precision, reliability, and strength with human resilience, intelligence, and improvisation. Improving the cooperation between humans and robots, i.e., Human-Robot Collaboration (HRC), is one of the most critical challenges in this field.

Most of the current research on Human-Robot teams aims to ensure the operators' safety. However, other important problems in HRC have not yet received the same attention, even though they are also essential to guarantee fewer stoppages, higher productivity, and human comfort.

The efficiency of collaboration between team members has a direct and significant impact on the quality of the task being performed by the team; therefore, more flexible and automatic robots need to emerge in an environment ever more oriented towards product customization and synergetic operations.

1.2 Motivation

Given the expanding presence of machines in industrial processes, optimizing the robots' performance is a clear goal. While human operators may understand an entire process and efficiently work out the most sensible sequence of operations, given a specific set of process parameters, a robot's knowledge is limited to its programming. A wide range of robots is programmed based on existing code, which has to be adapted to different operations and process parameters. Changes in the parameters may cause the optimal sequence of tasks to be altered. In complex situations, it may be difficult to reconfigure the robot in a satisfactory and practical way.

Complicated sets of operations require complex and effective scheduling and planning methods for different tasks. A standard solution is to have the order of operations hard-coded in the robot's programming. A more efficient solution would be to have the robot schedule and organize the planning of various activities automatically and be able to repeat that same planning exercise with new sets of tasks and parameters in different environments. Breaking down large industrial tasks into small operations can be challenging, but it is an extremely important factor in process optimization. Achieving several ways of decomposing the same task can also contribute to a more flexible work environment and facilitate product customization.

In addition to the difficulty of planning their tasks logically, there are some other obstacles in robots' way towards optimal performance in a collaborative environment context. Human executed tasks are more prone to delays or other time inconsistencies. The human factor carries a lot of unpredictability, which may disrupt the initial programmed solution and cause the robot to make mistakes. By contrast, at this point, the robot's operation would be fixed, meaning the operator would have to adjust to the robot's limitations. The robot would then be the master, dictating the process. Simultaneously, the human would have to continually guarantee the repeated and precise execution of the tasks, a strenuous and exhausting job, potentially leading to more process errors, stoppages, and lower productivity. Finding ways for the robot to be more adaptable to the operators' tendencies when necessary and learn from process errors would change this dynamic. Having the more intelligent and flexible actor in the process, i.e., the human, become the master would fundamentally improve the executed tasks' quality and efficiency.

1.3 Problem Definition

In light of the industry's most recent changes, there is a need for new approaches that can meet the requirements of increased customization and flexibility. In HR teams, it is always helpful to have robots with a certain degree of autonomy. However, the main goal has shifted towards developing partially-autonomous robots that can cooperate with human partners and adapt according to the operators' behavior. For an efficient work strategy and collaboration, robots need to be intelligent in performing tasks, which involves arranging the ideal sequences of operations and planning the trajectories associated with the performance of those same operations. Changes in the environment can require altering the outlined plans, which can become infeasible. Robots need to be capable of overcoming these changes and adapting to different realities. Thus, developing highly-specific task-motion planning tools has become expendable, while generalized approaches that can survive significant changes to process parameters have risen as the most attractive to flexible industrial environments.

The problem consists of finding a generalized task and motion planning approach focused on formulating viable planning sequences for task execution. In this process, the robot and a human operator should work in a collaborative environment, being exposed to interchanging scenarios. The problem was divided into two main areas: Task Planning and Motion Planning.

Numerous Task Planning solutions for all sorts of problems have already been implemented. However, most of those solutions do not focus too much on a necessary first step of task planning which is the breakdown of the task, i.e., task discretization. Most approaches work with the already discretized task as an input, thus opting for a manual decomposition of complex tasks, which can limit the flexibility and customization capabilities of the solution and can also steal away some variability, resulting in a smaller set of possible plans for each desired task.

Task plans are abstract sequences of operations that may not be feasible to execute in certain scenarios. Therefore, offering task plans without a proper test in the specific environment is insufficient to solve many industrial problems. In this stage of planning, it is crucial to validate the robotic trajectories and operation flows that are obtained in the task plans. This way, the plans can be analyzed and scrutinized, so only the most viable and efficient ones can be offered as a solution to the planning problem. This is where Motion Planning plays a significant role in the planning process.

As previously mentioned, a significant challenge in task planning lies in finding generalized approaches that can offer solutions to different scenarios. This challenge turns even more complex when considering the automatic discretization of tasks, given that in a high-customization context, a solution is only as good as its scalability. A generalized approach also raises difficulties on the motion planning side, which can only contribute to finding better planning solutions if there is a clear, efficient, and robust integration between the Task and Motion aspects of the problem.

1.4 Objectives

This dissertation addresses some of the robots' limitations presented above in the field of Human-Robot Teams. The project is divided into two phases, as mentioned in Section 1.3, each meant to tackle a different set of obstacles.

The first phase should provide the robot with the capability to automatically plan tasks with sequences of multiple operations. In this stage, tasks should be decomposed into small and straightforward sub-tasks or granular operations in a general and context-independent manner. A task planning algorithm should be developed using those operations. The idea is for the robot to plan logical sequences of granular operations using the algorithm without constant input from an operator or the need for hard-coded algorithms, which are not as dynamic and adaptable.

The second phase should engage in planning the robot's trajectories, associated with the planning of sequences of tasks. This stage should be able to validate the formulated task plans to provide a more complete solution to the planning of the desired task. By testing the operations' execution, process parameters should be acquired so the best viable plans can be offered as solutions.

After these two phases of the project, a robot should be able to schedule, plan, and organize sets of operations and coordinate the robotic movement with the planning of tasks. With this, the human-robot team capability of dealing with changes is improved. In sum, the following objectives should be achieved:

1. The algorithm input should contain only the rules for the discretization of the task and the necessary operation information and input process parameters;

2. The task planning algorithm should be able to break down the task according to the input rules and formulate a planning problem with the decomposed task;

3. Task plans should be generated according to the planning formulation;

4. The robotic movement and viability of the formulated task plans should be tested in simulation environments;

5. The proposed approach should be scalable to different task input configurations and scenarios;

6. The best viable and simulated task plans should be outputed.

1.5 Dissertation Structure

In addition to the Introduction, this document contains six more chapters: Literature Review (Chapter 2), Task-Motion Planning Framework (Chapter 3), Task Planning Approach (Chapter 4), Motion Planning Approach (Chapter 5), Experiments and Results (Chapter 6) and Conclusions and Future Work (Chapter 7).

Chapter 2 contains the information collected from the research on relevant topics to the dissertation, such as HR teams, task formalization and discretization methods, task and motion planning approaches and simulation platforms.

In Chapter 3, the case study that is central to the implemented solution and that is used for its validation is presented. This is followed by an overview of the proposed solution, accounting for general aspects that characterize the solution such as the solution architecture, the points of similarity with the reviewed research and some important definitions to understand the implementation.

Chapter 4 reveals the task planning approach. More specifically, it details how the task is formalized and discretized based on the inputs, how planning graphs are generated and how task plans can be formed by traversing the graphs.

Chapter 5 accounts for the motion aspect of the solution, detailing the work done to create motion plans that using the task graph information, to set up the simulation scene and the remote API client and to generate the robotic movement on the simulator side.

Chapter 6 shows an account of the experiments performed to validate the different aspects of the planning framework and the results that were obtained. The limitations of the solution are also reported.

To conclude, Chapter 7 gathers the conclusions that were reached as well as defining some relevant areas of future improvement.

Chapter 2

Literature Review

A series of articles were carefully read and analyzed to better understand the scope of the field and learn of some of the existing solutions for task and motion planning with human-robot teams. This process was helpful in determining what approaches and technologies were more desirable to use while attempting to solve the problem explained in Section 1.3. The selection of the articles was based on the quality, clarity, and relevance of their content. Initially, the search focused on broader subjects to get as much information as possible and find new study areas. At a later stage, articles that failed to meet the criteria were excluded, and the information gaps were filled by more targeted research. This chapter presents useful background information in the field, as well as a collection of related papers.

2.1 Human Robot Teams

Human-robot teams have garnered a lot of attention in today's industrial and scientific community. Research efforts in this field have tried to find innovative ways to couple human and robotic skills in a collaborative environment. Some key objectives and challenges related to human-robot collaboration are presented in [1]. A wide range of studies on the topic is collected, which is used to analyze the field in two major topics carefully: HR task planning & coordination and intuitive programming.

The same article also classifies human-robot interaction, specifically in assembly problems, according to different factors, such as:

- The type of hybrid assembly cell: based on a *workspace sharing* approach or a *workspace and time sharing* philosophy;
- Role of the human: supervisor, operator, teammate, mechanic, programmer or bystander, among others;
- Level of interaction between the human and robot, in other words, how much of the process is shared between them: *shared tasks and workspace*, *common tasks and workspace*, *common tasks and separate workspace*

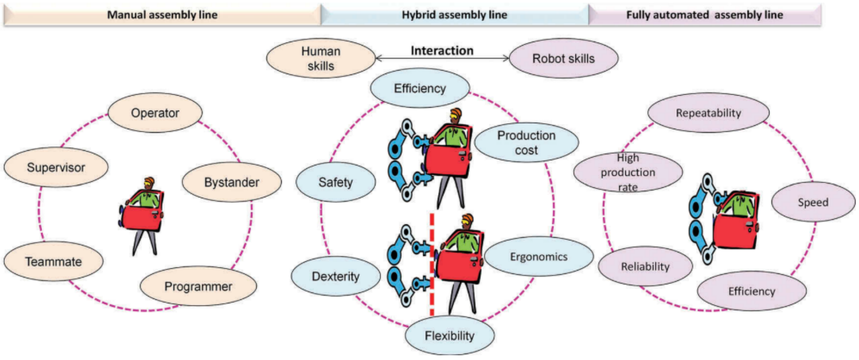


Figure 2.1: HRI in different assembly scenarios. [1]

Figure 2.1 summarizes the different approaches to an HRI assembly process, relating to the level of interaction and some of the characteristics and advantages of each one.

In the task planning field, [1] mentions different metrics that can be used for evaluating each of the two partners in a HRI scenario. On the robot's side, self-awareness, human awareness, and autonomy are indicated, on the human's side, some important factors are the situation awareness, workload, and accuracy, while for the system as a whole, effectiveness, efficiency, quality of effort and number of interruptions are among the main criteria. All of these process parameters are important to manage large tasks, given that they offer information on what operations should be assigned to the human and the robot. Equipment and environmental characteristics, as well as ergonomics factors are also considered in this operation allocation process.

2.2 Task Discretization and Formalization

Most of the progress done in task planning has come up through the evolution and adaptation of the Stanford Research Institute Planning System (STRIPS). According to their priorities and objectives, a considerable number of planning strategies based on STRIPS have been tried and tested, each weighing distinct factors differently. *Heuristic search* and *constraint satisfaction* are considered to be two of the leading strategies for efficient task planning, according to [4]. The former uses a heuristic function specified by the planning instance to guide the search through the state space so as to try and find a corresponding solution, while the latter obeys a set of defined constraints/restrictions, eventually guiding towards a viable solution. *The FF Planning System*, which was a very successful approach based on heuristic search, is presented in [5]. A popular pioneering study in constraint satisfaction can be found in [6]. A detailed definition of the constraint satisfaction approach and a number of different techniques is presented in [7].

In [8], three groups of satisfaction-based approaches are mentioned: PDDL, Hierarchical Task Networks (HTN), and Constraint Satisfaction Problems (CSP). A brief account of some of these methods' variants can be found in [9]. PDDL approaches are based on the combination of high-level task planning with low-level motion planning through semantic attachment to a PDDL planner. The low-level planner checks action applicability and computes effects when high-level actions are executed. As described in [10], fully observable and deterministic task planning problems are defined in PDDL by a tuple $\langle$A, s_0, g$\rangle$, where A is a set of parameterized actions with preconditions and effects, s_0 is the initial state of the domain and g is a set of propositions, corresponding to the goal condition. Some methods use symbolic description to represent geometric feasibility conditions and action operator preconditions.

Hierarchical task networks are based on an extensive search, requiring mechanisms to increase its speed to make it more scalable to large problems. *Hierarchical Planning in the now* (HPN) is a flexible HTN-based technique capable of generating a hierarchy dynamically so that during a transition between degrees of the hierarchy, the goal specification of the planner is defined by the preconditions of the destination node of the transition. Some HTNs have the ability to backtrack the geometric module's decisions. A constraint satisfaction problem approach, which is specified in [9], establishes the notion of an action skeleton, which is a sequence of symbolic actions with unspecified geometric parameters. The geometric parameterization of actions occurs at a stage where the skeleton has been fixed. In this approach, the geometric parameters of the skeleton need to be discretized. Other CSP approaches also turn to constraint propagation to resolve geometric constraints. These methods are all viewed by [8, 9] as insufficient for industrial environments, as some of these approaches work well in lab robotics but may not always offer the most optimal solutions.

Logic-geometric programming consists of another group of approaches cited by [9] that are more focused on finding the optimal feasible solutions. This is done by treating the problem as an optimization problem on the geometric level and controlling the constraints logically. However, these methods' sheer complexity significantly affects their scalability to some scenarios with a large number of tasks. According to [11], early attempts at logical representations were found to be infeasible. Still, more recent approaches were able to harness the upsides of logic-based methods and reduce some of its limitation's impact. Some alternative methods include Partial-Order Planning (POP) and GraphPlan.

POP is a popular method that prioritizes efficiency, searching through the space of plans instead of the state space. GraphPlan explores through the state space but uses an approximate reachability graph with mutual exclusion constraints. A first-order linear logic approach is depicted in [11], the Linear Planning Logic (LPL). The author defines linear logic as a resource-aware logic that treats resources as single-use assumptions, which can enable the encoding and reasoning of dynamic state domains. Unlike other logic-based methods, LPL combines the elegance of classical logic with the strengths of intuitionist logic, resulting in an efficient representation of the dynamic state. A particular downside lies in the inherent difficulty of working with a complex logic-based method, in addition to some resource management issues.

The definition of the planning formalism is not always a main focus in many task planning studies, and sometimes it is barely mentioned. However, as described above, it may be a deciding factor in the applicability of the method being implemented. Some approaches may be too complex and computationally heavy, resulting in scalability-related limitations. Other more simplistic known techniques may be more efficient while at the same time not always offering the most optimal possible solutions. In [12], further information can be found on different ways of representing task formalisms and knowledge.

2.2.1 HTNs and graph-based planning

As explained above, HTNs are a constraint-based method of representing a task planning problem that is based on defining hierarchical relationships between different sub-tasks and operations as well as restrictions along the network to reach viable solutions. In [13], a detailed analysis of the HTN design and features can be found.

Conventional HTNs take an initial task network as an input, consisting of a set of tasks, each with a list of relevant attributes. These attributes and the relationships between the tasks form constraints that will originate task plan solutions. Three types of tasks are defined:

- Goal tasks, which correspond to the desired result;
- Primitive tasks, achieved by directly executing the corresponding action;
- Compound tasks, which involve several primitive tasks and possibly one or more goal tasks.

Along with the initial task network, the HTN algorithm also takes a set of operators that will be responsible for determining the effects of each primitive task and a set of methods that describe how to perform non-primitive tasks. Generally, the planning algorithm tries to find non-primitive tasks along with the network and replaces them with a number of smaller tasks in accordance with the set of methods received. When no more non-primitive tasks can be found in the network, the algorithm produces a sequence of all tasks that satisfies all the constraints. To sum up, the planning problem P in the HTN problem can be defined as $P = \langle d, I, D \rangle$, where d is the initial task network, I is the initial state and D is the planning domain, containing the set of operations and methods, defined as $D = \langle Op, Me \rangle$.

As with most formalization and planning approaches, HTNs owe their origins to STRIPS, making that comparison unavoidable. This approach, in fact, includes all of the concepts that STRIPS provides while offering a richer constraint language. It also provides more flexibility and expressivity in the definition of arbitrary constraints along with multiple tasks and in handling compound task abstractions. On the other hand, HTNs can become incredibly complex when handling non-primitive tasks, which is why it is usually recommended to have as few as possible non-primitive tasks or, in cases where this is unavoidable, to make them totally ordered.

In [2], GraphHTN is introduced as a hybrid approach, meant to formalize tasks and their connections and constraints in the HTN-style format but to solve the planning problem using a common planning-graph generation approach. This way, the solution improves HTN planning by using planning graphs, and planning graphs sped up by exploiting HTN control knowledge. The approach divides the problem into two data structures:

1. The *planning tree*, which holds the task decomposition rules and constraints;

2. The *planning graph*, generated in a way that is consistent with the constraints in the planning tree.

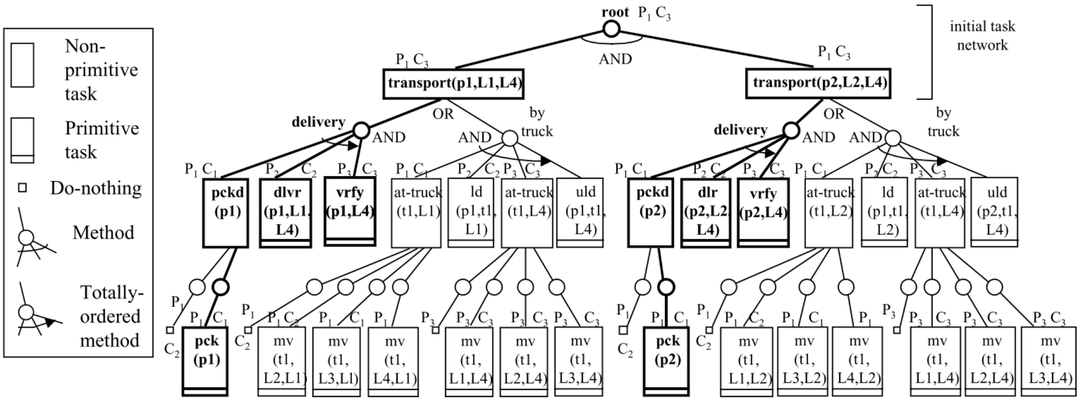


Figure 2.2: Example of the planning tree structure as described by the GraphHTN approach. [2]

The *planning tree* consists of an AND/OR graph with all the task's decompositions, and an example of the structure is presented in Figure 2.2. AND/OR graphs are Directed Acyclic Graphs (DAG) whose nodes can be *AND-nodes* or *OR-nodes*. These hierarchical structures can be a good solution for dealing with complex tasks that can be broken into smaller operations in several different ways. A more detailed account of AND/OR graphs applied to an assembly scenario can be found in [14]. In the case of the GraphHTN *planning tree*, there are two types of nodes:

- Method nodes, which correspond to AND nodes;

- Task nodes, which correspond to OR nodes, whose children are method nodes used for the decomposition of the task

The initial task network is composed of a special method node whose children are initial tasks. A solution to the planning problem will be a subtree of the planning tree. The *planning graph* is generated level by level recursively, in accordance with the *planning tree* and each iteration, the algorithm tries to find a valid plan.

An interesting work involving the construction of HTNs is described in [15]. This variation of HTNs, dubbed Clique/Chain HTNs (CC-HTNs), allows for robots to build their own hierarchical interpretations of tasks by leveraging ordering constraints and knowledge from previously known tasks to learn multiple levels of abstraction for arbitrary task networks. The presented algorithm starts out with a *task graph*, where graph nodes are states and edges are actions, and sequences of actions to perform can be formed by jumping through the nodes until task completion is guaranteed. Subsequently, a different graph structure named *Conjugate Task Graph* is formed where nodes are now subgoals of the main task and edges represent the problem constraints. Similar operations are grouped in an automatic generalization exercise from this new graph, forming a clique or chain abstraction, clique for unordered sequences, and chain for ordered sequences. To conclude, hierarchical structures are established automatically, based on a bottom-up approach that uses the generalization previously obtained.

2.3 CoppeliaSim

For connecting task and motion planning, there is a need for a platform that can take information from task plans and be able to originate the continuous actions, while keeping a realistic account of the environment effects that they produce. A common reliable and flexible way to do this is by means of a simulator.

CoppeliaSim [3] (formerly known as V-REP) is a general-purpose robot simulation framework that is highly versatile and scalable. This tool makes simulation capabilities more accessible to the general public by offering vast built-in functionalities and programming approaches. It also provides low-complexity integration tools for direct and efficient remote control.

There are plenty of simulation frameworks with immense capabilities, each with its unique approach, advantages, and weaknesses, including Gazebo [16], Webots [17], and Open HRP [18]. Compared to these and other approaches, CoppeliaSim is thought to have the more appropriate balance between the complexity of simulation setup and control, built-in functionality capabilities, and wide accessibility and scalability to compatible and open-source resources.

Figure 2.3 shows a CoppeliaSim scene, where it is possible to see the user interface. Among other elements, the interface offers a model browser with a large number of resources to use in simulations, and the scene hierarchy panel, where all objects in the scene are represented through their handle and the connections between them in the environment are established.

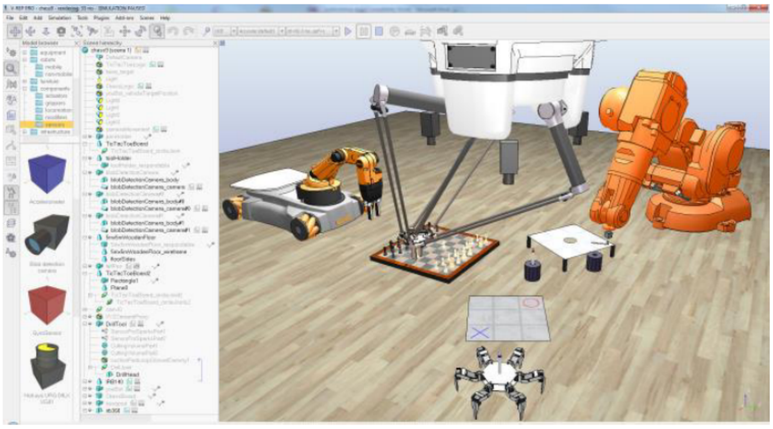


Figure 2.3: Generic CoppeliaSim scene with examples of robot models provided by the simulation platform working simultaneous, showing the wide variety of functionalities offered by the simulator. [3]

In [3], an account of the simulator functionality, properties, and supported techniques can be found, with respect to:

- Control code execution: executed on another machine, on the same machine but in a different process/thread from the main simulation loop or on the same machine and same thread as the main simulation loop;
- Programming: embedded scripts, add-ons, plug-ins, remote API clients, and ROS nodes
- Scene objects: joints, shapes, proximity/vision/force sensors, graphs, cameras, lights, paths, dummies, and mills;
- Calculation modules: kinematics, dynamics, collision detection, mesh-mesh distance calculation, and path/motion planning.

CoppeliaSim embedded scripts are written in Lua[1] code, with a main script handling the simulation functionality and calling child scripts in a cascaded way, respecting scene hierarchy. More detailed information on the simulation platform characteristics can be found on the CoppeliaSim User Manual[2]. CoppeliaSim offers a number of tutorials and simulation scenes to help any user to get acquainted with the platform and its functionalities.

The *Designing dynamic simulations*[3] page offers a brief and clear explanation on basic concepts necessary for the understanding of the dynamic properties of simulated environments and guidelines for the design of those simulations. For dynamical purposes, scene objects (shapes) are categorized according to two criteria:

- Static or non-static (dynamic): static shapes' positions relative to their parent object are fixed and not affected during the simulation, while dynamic shapes are directly affected by gravity and other environment constraints that may cause their movement;
- Respondable or non-respondable: respondable shapes are influenced by other aspects of the environment, namely other shapes they may be in contact with, while non-respondable shapes cannot be affected by other aspects of the surrounding environment.

The most realistic category of shapes is undoubtedly the dynamic respondable shape, i.e., the object that can be moved and that is affected by natural physical forces and influenced by what is happening around it. However, the non-respondable and static properties in shapes can have advantages and be useful in certain simulation scenarios.

The same CoppeliaSim user manual page also lists some necessary considerations that should be taken into account when designing a simulation scene, as well as some guidelines for dealing with dynamically enabled joints and force sensors in association with shape objects and how they interact with each other in their hierarchical configurations in the scene. It also touches on the

[1] https://www.lua.org/

[2] https://www.coppeliarobotics.com/

[3] https://www.coppeliarobotics.com/helpFiles/en/designingDynamicSimulations.htm

role of dummy links in the dynamically enabled hierarchical relations between parent and child objects.

As previously mentioned, CoppeliaSim supports many plug-ins to perform a multitude of tasks. There is a variety of material (user manual guidelines, simulation scenes, tutorials, among others) on handling inverse kinematics[4] (IK), path plannnig[5] and remote API[6] setup and control, which can all be found and/or referenced in the user manual.

To solve inverse kinematics problems for robots in the simulation platform, using the IK plug-in, the CoppeliaSim guidelines suggest the creation of an IK environment with IK groups based on which the IK calculations will be computed. Each IK group can incorporate a single or several IK elements, which in turn contain a base, a number of links and robot joints, a tip (corresponding to the robot endpoint), and a target, i.e., a dummy that sets the position and/or orientation the tip of the IK element should move to.

CoppeliaSim offers the Open Motion Planning Library (OMPL) plug-in for path and motion planning. Consisting of an open-source library with multiple sample-based motion planning algorithms, OMPL is meant to be easily integrated into systems that offer the necessary basic simulation capabilities, such as CoppeliaSim or many other simulation platforms. With regard to remote APIs, CoppeliaSim offers some easy-to-use and lightweight options, namely the ZeroMQ, the WebSocket, and the legacy remote APIs. A brief overview of the different possibilities can be found in the user manual remote API page and short scripts to test client creation, connection to the simulator, and the simulation control operating from the client-side.

2.4 Related Work

To properly address the problem, it is crucial to start by understanding the general architecture of an implementation of this kind and all the conceptual thinking behind it. In [4], a description of the existing dichotomy between task planning and motion planning is explained. Task planning uses discrete reasoning to obtain decisions on certain actions' specifications and in what order they should be executed. Motion planning uses continuous reasoning to determine how the robot should move to reach a certain target location. Combining task and motion planners can be challenging and requires some adjustments. Isolated task planners are usually prepared to pick a single plan, the best one they can find, without offering alternative plans if the chosen one turns out to be inconceivable. In addition to that, some isolated motion planners assume a fixed configuration space, failing to take into account changes in the kinematics and configuration changes caused by the motion. For these reasons, integrating the two efficiently and effectively may prove to be a demanding goal.

[4] https://www.coppeliarobotics.com/helpFiles/en/kinematics.htm

[5] https://www.coppeliarobotics.com/helpFiles/en/pathAndMotionPlanningModules.htm

[6] https://www.coppeliarobotics.com/helpFiles/en/remoteApiOverview.htm

TMP has been a widely studied field, thus, many approaches focusing on distinct problems have come up through the years. The differing methods are characterized by their specific contributions to the field, what metrics they are committed to improve and how they propose to deliver those benefits to the system. An interesting depiction of a probabilistically complete open-source task-motion planning (TMP) framework, presented in [4], summarizes the main inputs of a task-motion planner in the following way:

1. Task Domain, that defines possible actions/operations, including their preconditions and effects.

2. Motion Domain, which defines the configuration parameters of the robot and the surrounding environment.

3. Domain Semantics, responsible for the coupling between the task and motion domains, in a feasible, efficient, collision-free manner. This breakdown outlines the starting points in the implementation of a task-motion planner: the definition of a planning strategy and formalism, the motion planning specifications, and how they can be integrated with a task-motion planner.

Figure 2.4 shows a map of the architecture proposed by [4] which lays out the necessary structures, files, modules and tools to compose the planning algorithm.

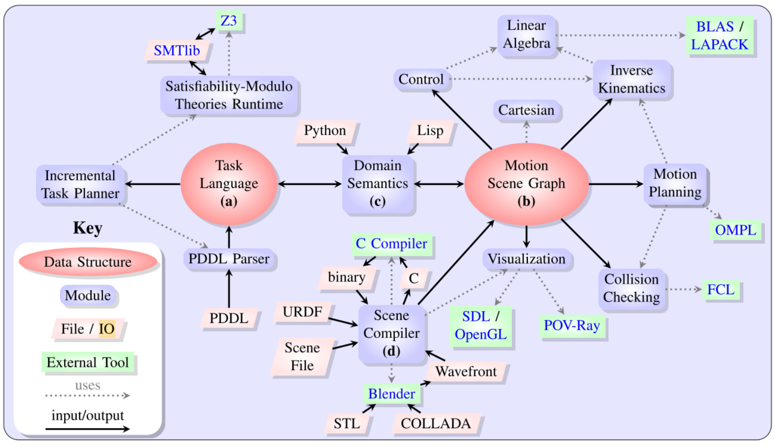


Figure 2.4: Architecture of the Task-Motion Kit, a framework that integrates multiple methods of task and motion planning. [4]

A proper motion planning system is significantly important for the optimization and validation of task planning. In realistic scenarios, a planner may reach a sequence of operations that cannot be executed due to limitations in the robot's motion. Therefore, checking whether a task plan can successfully translate into a set of actions performed by the robot is extremely important to validate and adapt (if necessary) the procedures outlined by a task planner. Some motion planning approaches include sampling-based planners, optimization-based planners, and planning through gradient descent. A sampling-based planner is the chosen approach by [4] because it offers probabilistic completeness, and therefore a solution is always found if one exists. Their planner is adaptable to different robots, scenarios, and algorithms, which they refer to as the *Task-Motion Kit*.

Some motion planning approaches have some limitations when applied to dynamic environments where objects may be moving, altering the kinematics and environment settings. In [19], a TMP approach that focuses on the benefits of combining symbolic and geometric reasoning is presented. This method uses an algorithm that optimizes over Cartesian frames defined relative to target objects instead of joint configurations, which can improve the effectiveness of the computed plans in scenarios with moving objects. This approach enhances integration with reactive controllers. It also proves to have good results in environments with inaccurate perception and imprecise control. However, the Cartesian frames approach has some limitations when compared to a more traditional joint configuration setting, especially in adequately detecting and avoiding robot collisions.

Most approaches currently used join different methods, resulting in an integrated solution for the problem. In [9], an integrated approach that uses Tabu search for task assignment, linear programming for task scheduling, and A* search for routing is mentioned, consisting of a common combination used in path planning problems that can optimize the system and backtrack to a high-level domain when the lower level cannot offer a feasible problem solution. The integrated approach proposed by the article, the *co-optimization approach*, uses a similar layered format, with high-level task planning and low-level motion planning, while adding a middle layer for action planning. The top layer finds the sequence of tasks that optimizes a certain performance metric defined by a cost function. This approach models task planning as a traveling salesman problem. The middle layer relies on a Hierarchical Action Network (HAN) to define a sequence of primitive actions for the implementation of a high-level task. The lower level generates the motion plan required to perform the primitive actions. This solution takes advantage of low-level information used in the optimization of the motion planning to improve the high-level task planning incrementally. Initially, an algorithm creates a task plan without taking into account the information from lower-levels. This preliminary plan is then updated based on a calculated cost of the feasible motion plans in the lower-level.

In [20], a modular highly-extensible HRC framework for collaborative tasks in hybrid assembly cells is presented. This approach entails a decision-making method, integrated within a Robot Operating System (ROS) framework, that can efficiently allocate tasks to the different resources it has at its disposal. Resources are defined as any human operator or robot that can participate in the execution of the tasks' operations. The task allocation process is carried out in three steps that assess different criteria for each resource:

1. Suitability of the resource to perform the operation;

2. State of availability of the resource;

3. Operation time.

To summarize, this method tends to choose the quickest available resources that can properly execute the necessary operation. The way this approach optimizes its results is by optimizing a specific process parameter, like the average resource utilization or the mean flowtime. The latter was the selected metric by the article to illustrate the functionality of the method. In trying to optimize the mean flowtime of the full task, the planner generates a number of alternative plans, each one distributing operations differently through the resources. The plan with the lowest mean flowtime would generally be the optimal plan. However, a differing choice could be taken, in case the human need to perform concurrent work in other cells. The operator communicates with the robot through body gestures that are detected by a depth sensor. This means of interaction makes it possible for the human to work in a separate space, which can ease safety measures and work constraints.

A dynamic scheduling scenario in a collaborative manufacturing assembly station is presented in [21]. Similar to the previous approach, this methodology proposes the breakdown of the work load into small operations that can be attributed to specific resources. A hierarchical model of the process is proposed, enabling the re-allocation of the lowest action units to different resources. This approach focuses on the dynamic and adaptive nature of the scheduling, offering a robust solution to unexpected events, such as resource failure or sudden environment changes.

Many current collaborative environments are failing to harness robots' full potential, while still placing a significant cognitive load on the human side of the team. A framework capable of performing role assignment and task allocation that focuses on this issue is presented in [22]. This approach is based on a transparent system, where mental models about the executing tasks are shared between the robot and the human. The idea is for the robot to perform the most strenuous tasks and for the human to have a more targeted role, mostly using his/her decision making and dexterity capabilities to more specific and unusual tasks that may prove challenging for the robots. This way, the human is left in charge of overseeing the whole process and can easily intervene in the most adequate tasks or when a problem occurs. The human is offered an intuitive interface that he/she can be comfortable with, regardless of the level of experience. Communication between the two partners is key in a transparent system. It occurs in both ways and in this case is limited to only what is truly necessary, to avoid an exhausting work load associated with the communication of every aspect of the process. However, it should be done effectively to avoid uncertainty in the system. The system may be only partially observable and understandable by the robot. However, this method also provides the robot with the ability to plan under uncertainty and in situations where the intentions of the operators are not clear. The chosen approach is based on the merging of high-level task planning with adptive planning under uncertainty modeled by POMDPs (*Partially Observable Markov Decision Process*).

Another study using a single-agent POMDP formalism in its task planning module is presented in [23]. This approach is a perception-based analytics framework that can estimate current and future states of actions performed by human operators. By doing so, it can improve the collaboration between humans and robots and assist the planning of tasks. In other words, the coupling of inference mechanisms and task planning algorithms makes it possible for the robot to anticipate

Table 2.1: Literature review articles and areas of focus.

Article	Task form. and disc.	Graph planning	HTN	Task planning	Motion	HR teams
Tsarouchi et al. [1]				x		x
Lotem et al. [2]	x	x	x	x		
Rohmer et al. [3]					x	
Dantam et al. [4]	x			x	x	
Hoffman et al. [5]	x	x		x		
Kautz et al. [6]	x	x				
Bartak et al. [7]	x			x		
Faroni et al. [8]	x			x	x	
Zhang et al. [9]	x			x	x	
Srivastava et al. [10]	x			x	x	
Kortik et al. [11]	x			x		
Sun et al. [12]	x			x		
Erol et al. [13]	x		x			
Knepper et al. [14]	x	x				
Hayes et al. [15]		x	x			
Koenig et al. [16]					x	
Michel et al. [17]					x	
Kanehiro et al. [18]					x	
Migimatsu et al. [19]				x	x	
Tsarouchi et al. [20]				x		x
Nikolakis et al. [21]	x			x		x
Roncone et al. [22]				x		x
Oshin et at. [23]				x	x	

the operators' behavior and predict their intentions, while adjusting its plans accordingly, resulting in an effective work interaction between the two partners. The approach is based on combining generative and discriminative models. These analytic models use a set of human skeletal joint locations obtained by a depth map video stream. The generative models determine a future joint position based on current and past positions, while the discriminative models calculate the probability of the future action belonging to each class within a set of action classes, displaying those probabilities in the form of a vector. The information from the models is then used by the task planning module, formulated as a single-agent POMDP, to improve the planning process. The study concluded that the approach could correctly predict future joint positions and even future actions before they start, providing a safer and more effective environment for a collaborative HR team.

To sum up the researched information, Table 2.1 can be consulted, where the most relevant articles are represented as well as the areas of research each one covers. It is also shows the areas that received the most focus, which are aligned with the priorities in the developed work.

Chapter 3

Task-Motion Planning Framework

Before going into detail over the specifications of the proposed Task-Motion Planning approach, it is important to go over the case study that is used for validation of its development, later on collecting and analysing results. Thus, this chapter presents the case study used in this project as well as an overview of the implemented algorithm and some definitions that will be used in the following chapters.

3.1 Case study

This section demonstrates the case study used for validation of the proposed solution. In [24], a model set for HRC research is introduced. This tool is presented as a unified framework to facilitate the comparison and integration of contributions to the field of HRC and it is meant to serve as a standardized collaborative task or reference collaborative setup. It also helps to provide experiments that can offer reproducibility and replicability.

The HRC model set[1] [24], consists in a common HRC scenario, a collaborative assembly of furniture. It was developed by investigators from the *Center for Engineering, Innovation and Design* and the *Social Robotics Laboratory* at Yale University. It offers a set of components that are cheap to obtain or produce and that can be easily understood and adapted to flexible scenarios of different complexity. The simple design of the components is meant to tackle the scalability problem, making it possible for the model to be used by a wide variety of robotic platforms, to simulate real-life problems and to facilitate a convergence towards a standardized task domain. The work also mentions some of the tasks that can be implemented using the model set. To summarize, the HRC model set fulfills the following requirements for a robust experiment tool:

- Easy accessibility and low cost of the components and final designs;
- Modular and scalable components, which can be re-used several times in different experiments;
- Vast applicability of the model set in the HRC field;

[1] https://scazlab.github.io/HRC-model-set/

- Scalability of the task complexity;
- Lightweight and small scale design, compatible with a wide variety of collaborative scenarios' payload requirements

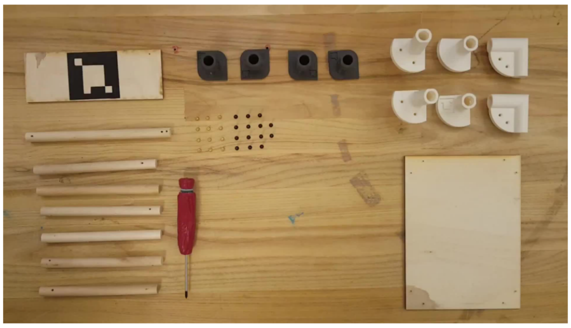
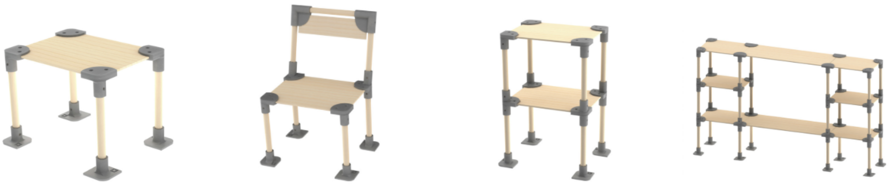


Figure 3.1: Examples of the components proposed by the model set.[2]

The model set design presents four groups of components: dowels, plywoods, brackets and screws. All of them can be acquired in a hardware store by a low price, except for the brackets which have a custom design, that was made available and can be 3D printed. According to the original proposed component features, dowels measure 1/2 in. in diameter and have varying length; plywoods are 1/8 in. thick with varying length and width; standard #4 screws are suggested with 1/2 in. and 1/4 in. lengths. The brackets should be scaled to fit the other components dimensions, i.e., the plywood thickness and the dowel diameter. This design offers vast possibilities of customization, making it possible to simulate in small scale all sorts of real-life scenarios. Figure 3.1 shows examples of the different types of components.



(a) Table (b) Chair (c) Shelf (d) Console

Figure 3.2: Prototypical objects proposed in the HRC model set.

Four prototypical objects are suggested in [24]: table, chair, shelf and entertainment console. These objects are examples of assembly configurations of the proposed components to simulate real-life objects, with distinct levels of complexity, the table being the simplest one and the console the most complex. Figure 3.2 displays the four objects at their full assembly stage. These suggested objects can be translated into proposed assembly tasks that can be used by new and already existing planning approaches so it becomes possible to have standard task domains and

[2] https://scazlab.github.io/HRC-model-set/

task planning inputs, as well as a facilitated process of result comparison and solution validation. Table 3.1 shows the list of components that take part in the assembly of each suggested object. Also in the table it is possible to see all of the specific brackets, plywoods and screws that make up the case study. Plywoods are differentiated by their dimensions and screws by their size. Brackets take several designations according to their format and scenario they belong to: chair (back, left and right), foot, shelf (90 and 180) and top (180 and simple top, which can also be defined as T90).

Table 3.1: List of components of HRC model set prototype objects.

	Brackets								**Dowels**	**Plywoods**			**Screws**	
	CB	**CL**	**CR**	**F**	**S90**	**S180**	**T180**	**T**	**D**	**6x8**	**6.75x2.5**	**6x16**	**1/2**	**1/4**
Table	0	0	0	4	0	0	0	4	4	1	0	0	8	8
Chair	2	1	1	4	0	0	0	2	7	1	1	0	14	12
Shelf	0	0	0	4	4	0	0	4	8	2	0	0	16	16
Console	0	0	0	8	12	4	4	4	24	6	0	2	48	24

3.2 Proposed solution

To address the problem presented in Section 1.3, a task-motion planning framework was implemented in an attempt at accomplishing the desired outcome. The developed approach is divided into two modules: the Task Planning Module and the Motion Planning Module. The main objectives of each module are outlined in Section 1.4. In this section, an overview of the developed algorithm can be found, as well as the general thought process behind it. As explained in the previous section, the proposed solution is inspired by the presented assembly case study. Therefore it is meant to be scalable to the generalized assembly task planning problem.

3.2.1 System overview

Regarding task planning, the approach starts by parsing a task configuration input that holds information on what components are necessary for the assembly and how they can be connected to result in the desired object. After that, the task discretization and formalization process begin by determining the sub-tasks required to achieve the task object. Two types of sub-tasks are considered: *assembly* sub-tasks at a higher level and *build* sub-tasks at a lower level. Thus, in other words, the input configures the algorithm to determine what assembly and build sub-tasks are required to accomplish the task, as well as some constraints on how these sub-tasks are to be associated and ordered. The output of this process is a list of different *recipes* for planning graphs based on the different ways to decompose the main task, which can be seen as distinct assembly strategies to construct the desired object.

Each recipe should then originate a graph from which task plans can be obtained. In this part of the process, a new lower level of the task process surges, referred to the *operation* level, holding assembly operations and build operations. These operations correspond to the more granular actions (or primitive tasks as defined in Chapter 2), that is, the actions that cannot be divided further and can directly be executed by the robot or human operator. Operation actions are set up through another input that holds ordering constraints and configurations that are used to integrate these operations with the sub-task nodes in the planning graph.

Task sequences can be generated by traversing the planning graph, following a set of rules and constraints. All node sequences are considered at this point as long as the constraints are followed, and all nodes in the graph are included in the sequence. These sequences are meant to be scrutinized through motion simulation, which will determine their validity.

The motion planning module is divided into two main parts: graph translation and movement generation. Graph translation stands for the harnessing of the necessary information from the graph for the robot's motion. Motion sequences with the relevant information are obtained from the task plans and then sent to the simulator to be executed. Movement generation refers to the robotic movement execution on the side of the simulator. The simulator is controlled by means of a remote API, where the client is responsible for sending the desired operations' orders, and the server (simulator) executes the movements it receives.

Figure 3.3 shows an overview of the developed framework, divided into the two main modules. This architecture represents the different modules that were implemented, how they interact and the logical sequence of the process, beginning with the parsing of the inputs and the definition of the task discretization possibilities and corresponding planning graphs, which is followed by the generation of task plans translated into motion plans that are tested in the simulation environment and finishing with the choice of the best task plans to be returned as the system output.

Most of the developed work was implemented through using Python, except for the *Movement Generation* which consists on a Lua script that runs in the simulator. The system inputs are XML files and the output is a list of the chosen task plans' sequences of nodes.

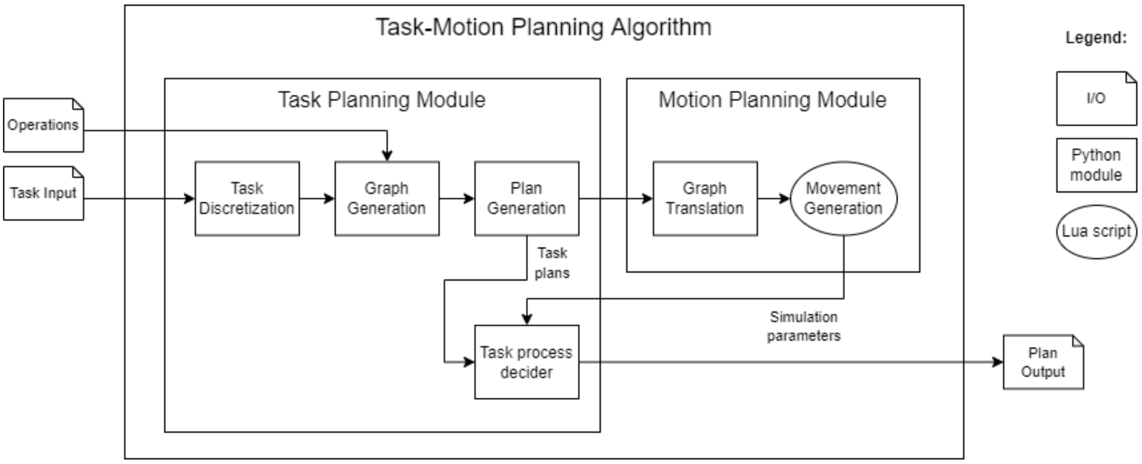


Figure 3.3: System overview.

3.2.2 General Approach and List of Concepts

To better understand the proposed solution, Table 3.2 defines a number of terms used regularly throughout the dissertation and to be consulted if any doubts surge. Some of the terms will be explained in further detail along with the document in the implementation Chapters 4 and 5. This section is meant to explain the general thought process behind the solution. To determine the specifications of the solution, the research information mentioned in Chapter 2 was taken into account.

Table 3.2: List of concepts to better understand the solution.

Concept	**Definition**
Component	Small single piece of the model set that cannot be subdivided into other pieces.
Part	Abstract concept. Formalization of an intermediate object in the assembly process of the final object. Represents a possible combination of a structural component with one or more link components.
Structural component	Component responsible for forming the final object spatial structure, e.g., a dowel forming a leg or a plywood forming a base. Main component that integrates a *Part*, also knows as the part body.
Link component	Component that is responsible for forming the connections between the structural components of the assembly task object. Secondary component integrating a *Part*. Corresponds to the *Bracket* in the presented case study,and therefore is often referred along the document as simply *bracket*.
Bracket Configuration	Characterization of a specific bracket, defining the types and amounts of connections the bracket can hold. Serves as a rule or constraint included in the system input.
Component Configuration	Holds information on *Part* creation possibilities and connection between components. Consists on a set of configurations that create an abstract realization of the desired object and offer the algorithm instructions to achieve the final object.
Recipe	Structure that contains *Elements* and *Assemblies*, holding information on how the final object was divided and how those divided pieces connect to each other. This serves as a direct recipe for the creation of the planning graph and defining its constraints.*Assemblies*, holding the information on what
Element	Practical realization of the abstract concept of *Part*. Belongs to a *Recipe*. It has a unique *ID* so it can be identified within the recipe and a fixed component configuration serving as a constraint to its attachment possibilities. It can be described as a number of components connected to each other forming an intermediate piece.
Assembly	Connection between two intermediate pieces within the recipe. In other words, it defines a match between two *Elements* that are meant to be connected to each other and the attach point in each element that partakes in the attachment.
Task	Full assembly task, resulting in the complete assembly of the desired object.
Sub-task	Subdivision of the task, obtained by the discretization process of the main task. Non-primitive task that can be discretized into even smaller operations. In the presented case study, there are two types of sub-tasks: *Assembly Sub-task* and *Build Sub-task*.
Operation	Primitive action, i.e., action that cannot be decomposed into smaller actions and therefore is executed directly either by the human operator or the robot.

The initial formalization of the planning problem was viewed through two lenses: the assembly process and the assembly object. Both approaches serve to divide the main task into smaller actions and define how they connect with each other hierarchically. The former implies the breakdown of the task into sub-tasks and operations, while the latter breaks down the final object into smaller intermediate objects and components. Figure 3.4 illustrates the two defined approaches to address the decomposition of the task planning problem.

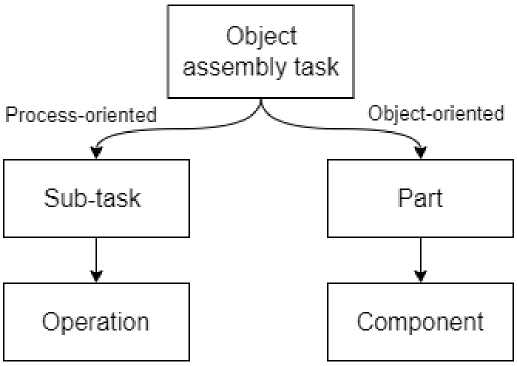


Figure 3.4: Approaches to the breakdown of the task.

This kind of decomposition is present in many papers studied and analyzed in Chapter 2, namely in [21] [15] [22]. The idea is that the task should be divided into hierarchically smaller actions that result in the formation of larger objects until the desired final object is obtained. Thus, the solution can be seen as a top-down approach to decomposing the task process-wise and a bottom-up approach to incrementally generate hierarchically higher objects.

Given the immense possibilities and potential for complexity in the presented case study, only one intermediate level object-wise was considered. This means that the small components should be attached into parts according to the system's rules and input constraints, and thereafter the parts should be assembled into the final object. These two processes are thus defined as follows:

- *Build* sub-tasks: construction of *parts*, i.e., attachment of small components into intermediate pieces of the final object;
- *Assembly* sub-tasks: connecting the created intermediate parts, resulting in the desired object.

The build and assembly sub-tasks are determined by the automatic task discretization and formalization method, using a bottom-up approach to find all possible combinations of components that lead to the task object. This approach was mostly inspired by [15], which explores the autonomous generation of HTNs.

The actions at the operation level are determined by the *operations* input. This input offers the system information, so build, and assembly sub-tasks can be subdivided into primitive actions at the operation level accordingly. Operations are directly executed either by the human operator or the robot, and the allocation is defined in the input. The solution foresees five different operations with fixed allocation. The allocation is outlined in Figure 3.5 and the operations detailed below:

- *Pick*: performed by the robot, consists in moving toward the object in question, grasping the object and moving back to a neutral position;

- *Place*: performed by the robot, consists in moving the object held by the gripper to the target location;
- *Snap*: performed by the human operator, consists in attaching two pieces together in the correct attach points;
- *Position Screw*: performed by the human operator, consists in placing the screw in the correct position to be fastened;
- *Fasten Screw*: performed by the human operator, consists in fastening the already positioned screw.

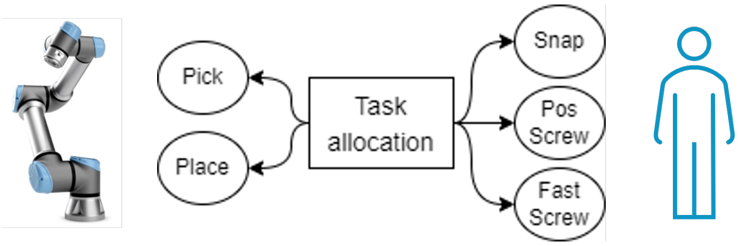


Figure 3.5: Allocation of operations: robotic (Pick and Place) and human (Snap, Position Screw and Fasten Screw).

The chosen operations are common in assembly scenarios, and similar versions can be found in many articles on the topic [21][22][24]. *CoppeliaSim* was the chosen simulation platform to generate the movement of the robot, and a simulation scene was developed to test the robotic operations.

The system supports highly changeable inputs, addressing the problem of product customization and process flexibility explained in Section 1.3. This means that the operator can alter different component types and specifications without compromising the program. It also can be applied to different objects with different configurations, making it a generalized approach.

Chapter 4

Task Planning Approach

This chapter describes the proposed process of task planning with regard to the parsing of the algorithm inputs, the task domain abstractions, the formalization of the task problem, the connection between recipes and graph creation, and the generation of planning graphs and task plans. It is divided into four parts: input presentation and description, task discretization and formalization, generation of planning graphs, and definition of task plans.

4.1 Inputs

This section aims to introduce the inputs necessary for the algorithm to function. This refers to both the task discretization and formalization module input - which supplies the algorithm with the information needed for the computation of the task assembly strategies that will, in turn, be used in the construction of planning graphs - and the additional input to the graph generation module that offers information on the most low-level operations. To distinguish the two, the former one has been labeled *task input* and the latter *operations input*.

The task input is meant to offer the generalized information of the task, focusing on the high-level process, i.e., the task and sub-task levels of the process. It consists of an XML file with the vital information for the formalization of the planning problem and task decomposition, as well as configurations of the task and problem constraints. The following topics detail the input:

- The desired task, i.e., the object that is meant to be built;
- A list of the components that make up the desired object to build, divided by type and specification, e.g., *Bracket* and *T*, respectively, and accompanied by the numbers of the specific components required to carry out the assembly;
- A configuration of the brackets, defining a set of constraints related to the connections that each specific bracket can hold;
- A configuration of the generalized structural components, defining how the components should connect to each other, thus serving as a recipe for the assembly of the task object;

- A screw configuration, defining rules for what screws are meant for each possible connection;
- The part creation mode, which can be "Automatic" or "Manual."

The operations input delivers information to the system on how the sub-tasks can be decomposed. In other words, the input provides the graph generation module with knowledge on how to subdivide the sub-tasks obtained from the task discretization and formalization module. The knowledge provided includes:

- Information on the operations that the build sub-task nodes and assembly sub-task nodes are divided into;
- Ordering constraints for the operation nodes;
- Duration of the operation execution.

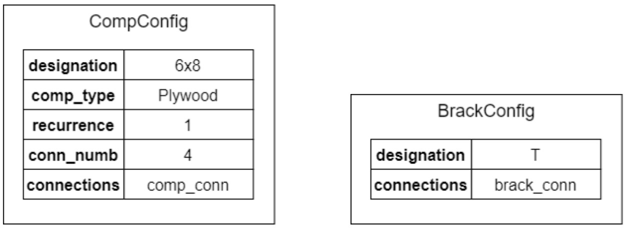
In industrial terms, the inputs provide the algorithm a bill of materials and information on product procedure. It is meant to be a set of basic rules and instructions and not a full description of the process. This justifies the deep focus on the task discretization and formalization module, given that the input is not a description of the graph itself.

4.2 Task Discretization and Formalization

In this section, the task discretization and formalization module implementation is described. Throughout the section, a lot of abstract concepts previously mentioned in Chapter 3 and Table 3.2 are explained in more detail. Before this process begins, the task input mentioned in Section 4.1 is parsed, and some configuration variables used to control the task's formalization are set up.

Figure 4.1 shows the structural component configurations and bracket configurations data structures, obtained from the input. As explained in previous sections, the bracket configurations offer the system knowledge on the bracket format, thus working as a constraint to what kind of connections are supported by each bracket specification. There is one data structure object of this kind in the program for each different bracket. The bracket configuration connections variable, *brack_conn*, is a dictionary that holds the relevant information on the bracket and its format can be seen in Figure 4.2, which shows examples for different brackets. Each bracket connection is characterized by the type of component it connects with, e.g., Dowel, and an additional connection specification serving to distinguish bracket connections and as a constraint to the assembly problem.

On the other hand, the component configurations offer the program information on how different components are supposed to be connected. Therefore, the list of all component configurations can be seen as an instruction list for the assembly of the desired object, or in other terms, an abstraction of the object that should be obtained at the end of the process. There is one component



CompConfig

designation	6x8
comp_type	Plywood
recurrence	1
conn_numb	4
connections	comp_conn

(a) Component configuration

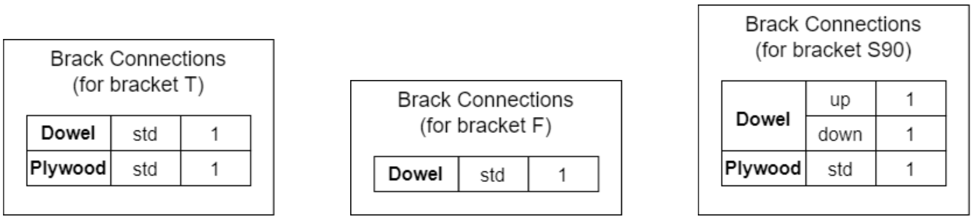
BrackConfig

designation	T
connections	brack_conn

(b) Bracket configuration

Figure 4.1: Data structures that hold the configurations for object structural components and link (bracket) components.

configuration object for each combination of connections that a specific structural component may support. Taking a shelf assembly task as an example, as defined in Section 3.1, two component configurations objects regarding the 6x8 object would be necessary, one that specifies the connections to Top (T) brackets (higher rack) and another to specify the connections to Shelf 90 (S90) brackets (lower rack). This example is illustrated in Figure 4.3.



Brack Connections (for bracket T)

Dowel	std	1
Plywood	std	1

(a) Top bracket

Brack Connections (for bracket F)

Dowel	std	1

(b) Foot bracket

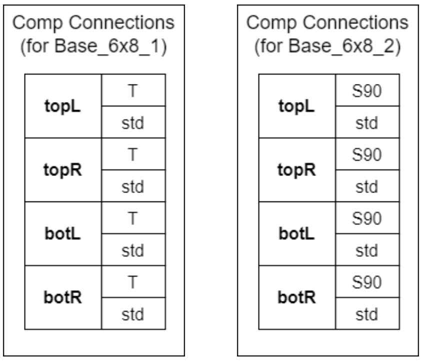
Brack Connections (for bracket S90)

Dowel	up	1
	down	1
Plywood	std	1

(c) S90 bracket

Figure 4.2: Examples of bracket configuration connections.

With the configuration variables set up, the main formalization process may begin. To understand the first step, the concept of *Part* introduced in Table 3.2 needs to be explored. A *part* can be defined as a possible intermediate object obtained from a first step of assembly of components, as is explained in Section 3.2. An example of a part object is presented in Figure 4.4. The part connections variable is similar in structure to the component configuration connection dictionary but it may not have a connection for all of the attach points.



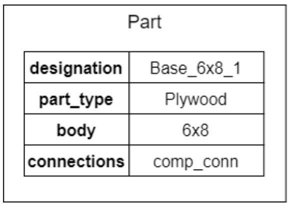
Comp Connections (for Base_6x8_1)

topL	T
	std
topR	T
	std
botL	T
	std
botR	T
	std

Comp Connections (for Base_6x8_2)

topL	S90
	std
topR	S90
	std
botL	S90
	std
botR	S90
	std

Figure 4.3: Examples of component configuration connections.



Part

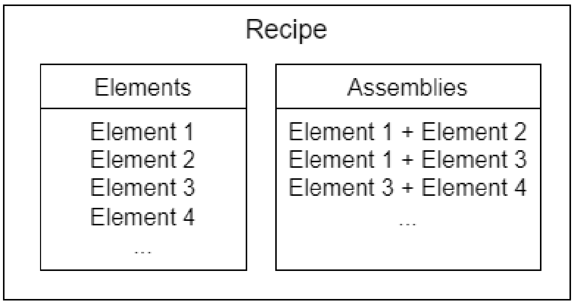
designation	Base_6x8_1
part_type	Plywood
body	6x8
connections	comp_conn

Figure 4.4: Part data structure.

Keeping this concept in mind, the first step of the bottom-up process, explained in Section 3.2, is triggered. This means the program finds all different possibilities of intermediate parts that can be obtained by combining a structural component to the link components that are meant to be placed on its attach points. This process of part creation is implemented using the *combinations* method from python's *itertools* package.

For more complex tasks with a large number of components and high variability in their configurations, an alternative may be necessary to avoid an unnecessary number of created parts. In this case, an additional input needs to be provided to the algorithm, carrying the information on the more relevant parts to be created. Whether the part creation process is fully automatic or not is defined by the given input.

The next step in the process consists in trying to attach the created parts recursively until the final object is obtained. As matches are found between different parts, they are saved into another structure mentioned in Table 3.2, the *Recipe*, which can be seen in Figure 4.5.



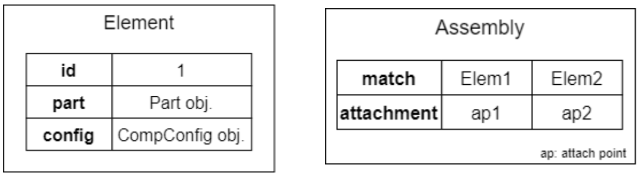
Recipe

Elements	Assemblies
Element 1	Element 1 + Element 2
Element 2	Element 1 + Element 3
Element 3	Element 3 + Element 4
Element 4	...
...	

Figure 4.5: Recipe data structure.

A recipe can be interpreted as a set of instructions for the creation of a planning graph, according to the obtained assembly strategy. Each recipe contains *Elements* and *Assemblies*. Elements correspond to build sub-tasks, responsible for putting together single components into intermediate parts, while assemblies are aligned with assembly sub-tasks, the higher level attachment of the intermediate parts that were built. Figure 4.6 details both structures.

The initial process of recipe formation starts off by setting pairs of parts and checking if there is a way to connect them. If the number of cumulative components in the pair surpasses the amount of a specific component in the final object, as defined by the input, the pair is automatically discarded. At first, no parts have an assigned configuration, so all combinations are tried. Since



Element

id	1
part	Part obj.
config	CompConfig obj.

(a) Element

Assembly

match	Elem1	Elem2
attachment	ap1	ap2

ap: attach point

(b) Assembly

Figure 4.6: Structures that integrate a recipe.

parts are connected through link components, i.e., brackets, the algorithm checks whether the missing brackets in one of the part attach points, as defined by the component configuration, can be found in the other part. If so, an additional check is required to figure out if the bracket in question can support the connection. For that purpose the corresponding bracket configuration is used. In case the bracket is available for a connection of that type and specification, the connection is possible and a match has been found. The process of match finding can thus be summarized by the following steps:

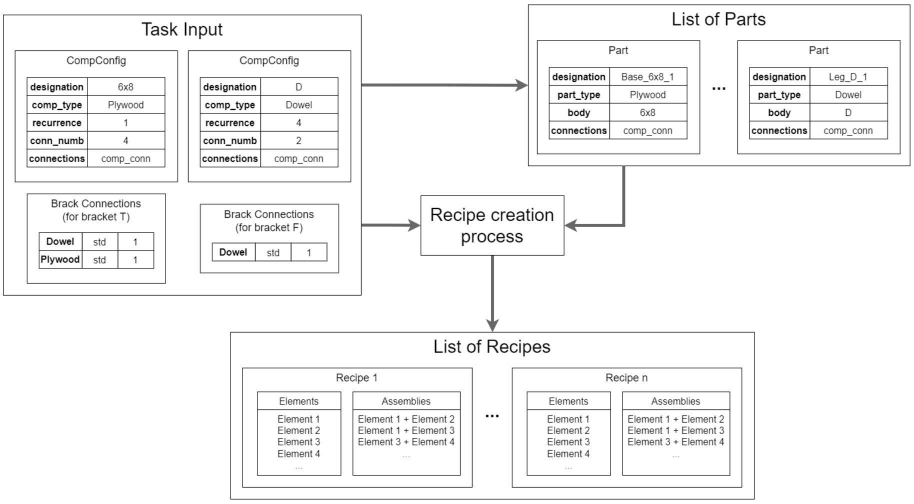
1. Generating combination pairs of all obtained parts;
2. Checking if the cumulative number of components in each pair does not surpass the input amounts;
3. Finding all the possible combination pairs of component configurations that may be compatible with each part;
4. Checking for each part which brackets are missing in its *connections* variable;
5. Determining whether the missing brackets from one of the parts can be found in the other and checking if the bracket supports the connection with the specification as defined in its bracket configuration.

When matches are found, new recipes are created, with elements carrying the information on the part and component configuration and assemblies detailing what elements are connected and through which attach points. In case two attach points in an element are indistinguishable and may form the connection to the other element, the multiple attach point pairs that may hold the connection are saved in the attachment variable of the assembly structure, shown in Figure 4.6b. In addition to the structures that will be used to form the sub-tasks, the recipe also has some internal control variables to perform the following functions:

- Controlling the component configurations attributed to each element. As it can be seen in Figure 4.1a, each component configuration has a recurrence variable which determines how many times it can be attributed to an element in the same recipe;
- Controlling the connection availability of the brackets in the recipe. Whenever a bracket holds a new connection, the control variable decrements the number of available connections of that type and specification for that bracket.

After recipes are formed in the first iteration of this process, the algorithm tries to find additional connections for those elements with the parts initially created. In each iteration, for all new matches found, a new recipe with the updated information is created. The recipes that resulted from the previous iteration are discarded, meaning that recipes that could not lead to new connections are abandoned. This process happens recursively until recipes with all the components determined by the input are obtained. The main differences between the first iteration and the subsequent ones are the following:

- Combination pairs are now formed by an element of the recipe and a part as initially defined;
- The element already has an attributed component configuration. Thus, each configuration pair contains the already assigned configuration of the element and a compatible configuration for the part;
- The algorithm needs to take into account what assemblies are already expressed in the recipe. Thus, it is not sufficient to check which brackets are missing on both sides, but also if the brackets in their specific attach points already integrate established connections in the recipe domain. This is where the recipe internal control variables come into the process.



designation	6x8
comp_type	Plywood
recurrence	1
conn_numb	4
connections	comp_conn

designation	D
comp_type	Dowel
recurrence	4
conn_numb	2
connections	comp_conn

Dowel	std	1
Plywood	std	1

Dowel	std	1

designation	Base_6x8_1
part_type	Plywood
body	6x8
connections	comp_conn

designation	Leg_D_1
part_type	Dowel
body	D
connections	comp_conn

Elements	Assemblies
Element 1 Element 2 Element 3 Element 4 ...	Element 1 + Element 2 Element 1 + Element 3 Element 3 + Element 4 ...

Elements	Assemblies
Element 1 Element 2 Element 3 Element 4 ...	Element 1 + Element 2 Element 1 + Element 3 Element 3 + Element 4 ...

Figure 4.7: Diagram showing the general functionality of the discretization process.

Figure 4.7 summarizes the make up of the discretization and formalization process. At the end of each iteration, the algorithm checks for duplicate recipes that may have been created and eliminates them. A final check is performed to determine whether there are recipes that already contain all the required components for the assembly of the final object. In this case, the final recipes are saved into a list that will serve as the output of this module. All other recipes are saved for the next iteration of the recursive loop, which runs until new recipes cannot be formed with the existing constraints.

4.3 Graph generation

This section explores the process of generation of the planning graphs based on the output of the task discretization and formalization module and the operations input, explained in Section 4.1, as well as the thought process behind the structure of the graphs.

The planning graphs' structure can be analyzed through two points of view: the process and the object. Process-wise, the graph contains distinct hierarchical layers, with an assembly task node at the very top, followed by a layer of high-level sub-tasks resulting from the decomposition of the task and finally a layer of lower-level operations resulting from the subdivision of the sub-tasks. Object-wise, the sub-task level can be subdivided into build sub-tasks, which attach components to create intermediate parts, and assembly sub-tasks, which associate those parts to form the final object. Assembly sub-tasks deal with larger pieces and therefore have a higher position in the object hierarchy and consequently in the graph as well. This can be interpreted as a top-down approach to process decomposition resulting from a bottom-up approach to the definition of the intermediate parts that will define the entire sub-task level of the graph, as explained in Section 3.2.

Thus, the graph sub-task level is defined based on the recipes obtained from the task discretization and formalization module, while the operation level is obtained from the data acquired by the system through the operation input. As can be seen in Figure 4.8, which describes the sequence of structures obtained along the task planning module, each recipe generated is responsible for originating a planning graph. Before being used by the graph generation module, the originated recipes from the previous module go through an attachment assignment process with the objective of attributing only one attachment possibility to each assembly. As explained in Section 4.2, all attach point pairs that can establish the assembly are saved. Given that in this case they are indistinguishable, they can be directly assigned. With the attachments simplified, the recipes are ready to generate graphs.

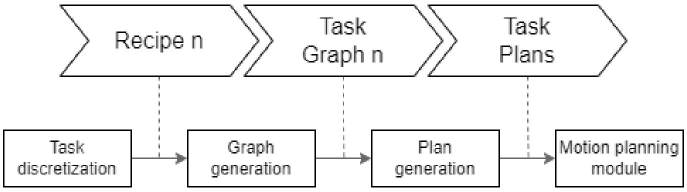


Figure 4.8: Logical sequence of structures built within the task planning module.

The graph generation process begins by creating a graph with the task node, obtained from the task input, and adding sub-task nodes according to the recipe in question. The graphs were implemented using the python package NetworkX[1], which is used for the creation and manipulation of complex networks. Each element in the recipe leads to a build sub-task node in the graph and each assembly leads to an assembly sub-task node. Directed edges determining the flow of the graph are set up between the task node and the assembly sub-task nodes, as well as between

[1] https://networkx.org/

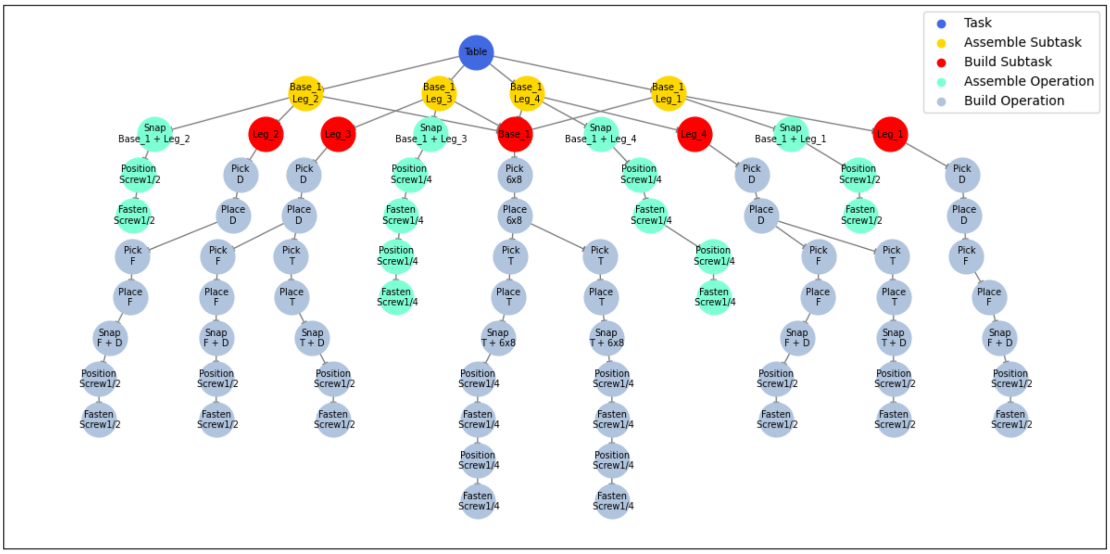


Figure 4.9: Example of generated graph for the table assembly task.

each assembly sub-task node and the two build sub-task nodes that correspond to the match in the assembly.

With the task and sub-task levels complete, the operation level is then defined. From the operations inputs described in Section 4.1, the necessary information is obtained to add to the graph the build and assembly operation nodes and corresponding edges. Build operation nodes form branches starting from build sub-task nodes and the same is true for assembly operations and sub-task nodes. The complete structure of the graph can be seen in Figure 4.9. Assembly sub-task nodes contain the following attributes:

- node_info, corresponding to the recipe assembly object that characterizes the node and carries information on the match between two recipe elements and the attach points in each element that engage in the connection;
- node_type, a string with the value "Assembly_Operation";
- object, containing a tuple of identifier strings for the two elements that form the match.

The assembly sub-task nodes' attributes contain the necessary information for the decomposition of the sub-tasks into the assembly operations and for establishing links to the build sub-task nodes. The build sub-task nodes contain similar attributes:

- node_info, corresponding to the recipe element object that characterizes the node and carries information on the intermediate parts that are initially formed;
- node_type, a string with the value "Build_Operation";
- object, containing an identifier string for the element in question.

These attributes contain the required configurations for forming branches of build operation nodes. In a general way, the attributes of both types of sub-task nodes play a similar role to the methods of task decomposition in traditional HTN non-primitive tasks, consisting of dividing a larger task into granular tasks that can be directly executed.

4.4 Plan generation and final output decision

With the planning graphs formulated, the next step in the task planning approach consists in finding sequences of nodes that may offer a solution to the assembly problem, i.e., task plans. This section explains the general rules and constraints for obtaining viable plans from the several graphs that may be obtained from the graph generation module.

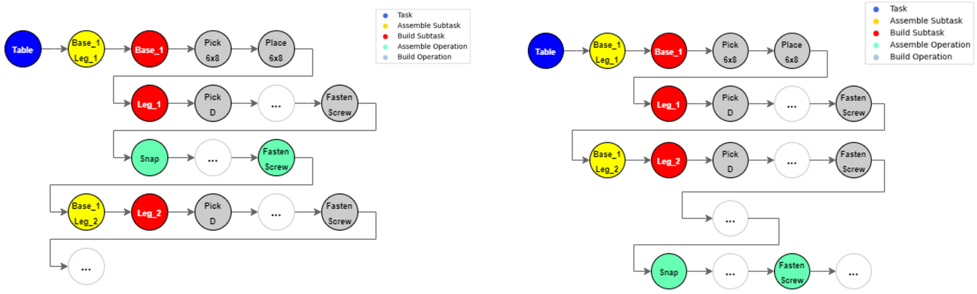
All nodes in the graph represent a task, sub-task, or operation that is required to be performed along the object assembly process. For this reason, in order to find a viable plan, the algorithm needs to traverse the entire graph; otherwise, the plans would be incomplete. Given the hierarchical nature of the graphs, which have a similar structure to trees, a good way to search through all the nodes is a branch by branch approach. Thus to start this process, the plan generation module finds all possible paths from the source (task) node to the leaves, which correspond to the build and assembly operation nodes that do not have successor nodes. These paths are then divided into two sets:

- The possible paths, corresponding to all paths from the source to the leaf build operation nodes;
- The conditional paths, corresponding to all the paths from the source to the leaf assembly operation nodes.

The set of possible paths holds all of the paths that can be pursued at a certain point in the plan sequence, while conditional paths are blocked until a specific condition occurs. When a conditional path is unblocked, it joins the set of possible paths, making it possible for the operations in the specific branch to be a part of the plan. With the bottom-up approach of the object assembly in mind, it is easy to understand that higher-level assembly sub-tasks (and their subdivided assembly operations), which are meant to attach intermediate parts, cannot be performed without those intermediate parts being assembled first. These intermediate assemblies are carried out by the build sub-task nodes (and their subdivided build operations).

Concerning the graph, this means that assembly operation nodes can only appear in the plan sequence after the related build operation nodes. In other words, the nodes that carry out the assembly of two parts can only show up in the plan when the nodes responsible for building those parts have already been included. Therefore each conditional path is unblocked only when both successors (the related build sub-task nodes) of the assembly sub-task node that belongs to the path have been included in the sequence.

With these constraints in mind, plans can be formed by iteratively adding path nodes to a defined plan sequence that is initially empty. This process occurs in a loop that ends when no more possible paths can be found. In each loop iteration, a path from the set of possible paths is selected. Thereafter, the nodes in that path already included in the plan sequence are ignored, and the remaining nodes are added to the end of the sequence. The traversal of the graph is similar to a depth-first search approach, which implicates that when a path is initially chosen, the entire branch all the way up to the source is explored. This means that all its operation subdivisions follow when a build sub-task is included in the plan. In addition to that, when an assembly sub-task is included, both its successor build sub-task nodes, corresponding to the build sub-tasks of the pair of elements of the assembly, are explored, and after that happens, the conditional path holding the assembly operation nodes associated to the assembly sub-task are unblocked.



(a) Example 1

(b) Example 2

Figure 4.10: Examples of task plan, showing different possibilities of plan formation.

The final structure of the task plan is similar to the one illustrated in Figures 4.10a and 4.10b, which show segments of plans starting with the main task node, followed by an assembly sub-task node and the two branches formed by the two successor build sub-task nodes. Immediately after the addition of these two branches to the plan, the corresponding assembly operations' branch is unblocked, i.e., added to the list of possible paths. In Figure 4.10a, the assembly operations' branch is pursued right after becoming available, while in Figure 4.10b, the plan jumps to the next assembly sub-task node and leaves the assembly operations' branch for a later stage in the sequence.

Task plan sequences are then sent to the graph translation module, which creates motion sequences to be generated in the simulator. From the simulation, the lead time of the process, based on the duration of human operations and the simulated robotic operations is determined. The task planning module then chooses the three best final plans, giving preference to the most efficient viable plans. These plan sequences serve as the final output of the program and can be exported as a JSON file.

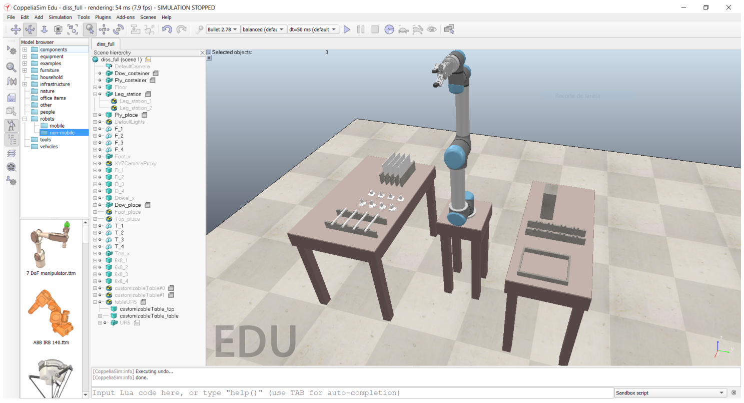
Chapter 5

Motion Planning Approach

This chapter explains the developed work around the formulation of motion sequences, the execution of robotic trajectories, and the preparation of the simulation scenarios to test task plans' validity and obtain process parameters used for their performance evaluation. The first section details the entire simulation preparation process. The second section explains the implemented method of translating the operation information from task plans into executable motion sequences. The third section describes how the movement was executed on the simulator side.

5.1 Simulation setup

To simulate the motion execution of the formulated task plans, a CoppeliaSim scene was designed, based on a simple scene configuration similar to many collaborative environments, as can be seen in Figure 5.1. This section goes through the process of building the scene and preparing the simulation. In the developed scene, there are three tables, one that holds up the robot, one that holds the case study components, and one that serves as a workstation where the assembly tasks will be executed. The simulation scene foresees that the assembly workstation is the only one the human operator has access to.



```
[CoppeliaSim:info] Executing undo...
[CoppeliaSim:info] done.
```

Figure 5.1: Developed scene, prepared for the assembly of the task object *Table*.

Figure 5.2: Universal Robots UR5e collaborative robotic arm.

The robot used for the simulation is a Universal Robots 5 (UR5[1]), which can be seen in Figure 5.2. The UR5 is a flexible and lightweight robot arm that is a perfect robotic partner for an HR team collaborative environment, given its fast setup, easy programming and safe working style. The UR5 model used in the simulation was obtained from CoppeliaSim available open-source resources[2]. For the end-effector, an RG2 gripper model was used, also from CoppeliaSim resources. The RG2[3] gripper is a 2-finger smart robot gripper with in-built force, torque and proximity sensors.

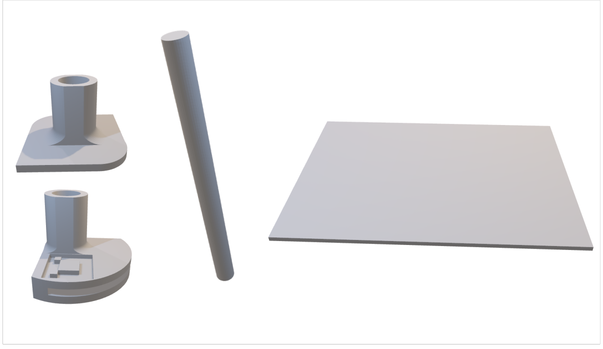
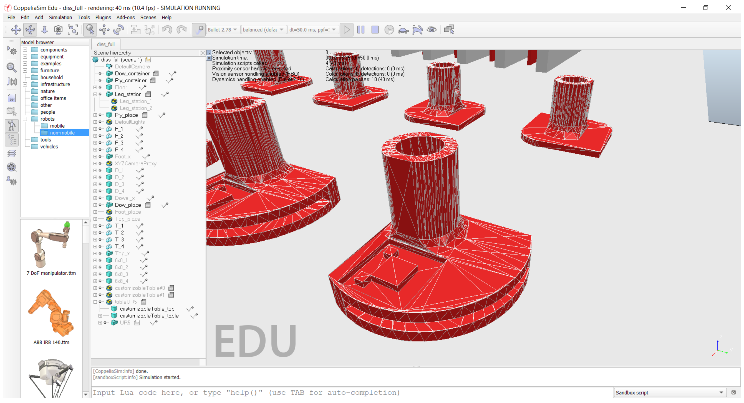


Figure 5.3: Examples of component meshes used in the simulation.

As for the components, dowel, plywood, and bracket STL files were imported into the scene as mesh shapes. The STL files of the bracket models can be found in the Yale *Social Robotics Laboratory* HRC model set GitHub repository[4]. The dowel and plywood files were created according to the case study features and recommendations and imported into the scene as well. Examples of the mesh files used to simulate the components can be found in Figure 5.3. The mesh shapes were

[1] https://www.universal-robots.com/products/ur5-robot/
[2] https://github.com/CoppeliaRobotics
[3] https://onrobot.com/en/products/rg2-gripper
[4] https://github.com/ScazLab/HRC-model-set



```
[CoppeliaSim:info] done.
[sandboxScript:info] Simulation started.
Input Lua code here, or type "help()" (use TAB for auto-completion)
```

Figure 5.4: Brackets during simulation in the dynamic content visualization mode, showing the respondable properties of the shape.

edited in the CoppeliaSim built-in triangle edit shape mode to comply with the necessary dynamic properties for the development of a robust simulation design, as described in Section 2.3.

A better understanding of the steps required to properly edit the shapes can be acquired by analysing the *Building a clean model*[5] CoppeliaSim tutorial, which suggests simplifying the mesh data by dividing the imported shapes into grouped smaller, convex shapes (or even pure shapes, e.g., cuboids, cylinders or spheres, when possible) and lays out some guidelines to do so without losing too much of the mesh original data. While dowel and plywood components have simple formats and dimensions, making them attractive for simulation computation purposes, brackets have a more complex design which requires a comprehensive editing process. Figure 5.4 exemplifies an edited bracket in the *dynamical content visualization and verification mode* (see CoppeliaSim/User Interface for more information) while the simulation is running. Screws were not considered in the simulation. A set of simple support structures were added to the scene to hold the components, both on the component storage and the assembly workstation sides.

The relevant reference positions and orientations for picking, placing or attaching objects are defined by dummies, which the robot tip should move to and align with. For pieces to stay in place when they are attached, a force sensor can be used between a pair of dummies, one from each of the pieces.

5.2 Graph translation

This Section explains how task plans originated from the task planning graphs can be used to form motion plans that can be interpreted and prepared to be sent by a remote API client to the simulation for movement simulation and scene updates. Details on the establishment of the remote API connection and the process on the side of the client and server are presented in Section 5.3, while this Section focuses on the control and handling of the task plan sequence and the operation data.

[5] https://www.coppeliarobotics.com/helpFiles/en/buildingAModelTutorial.htm

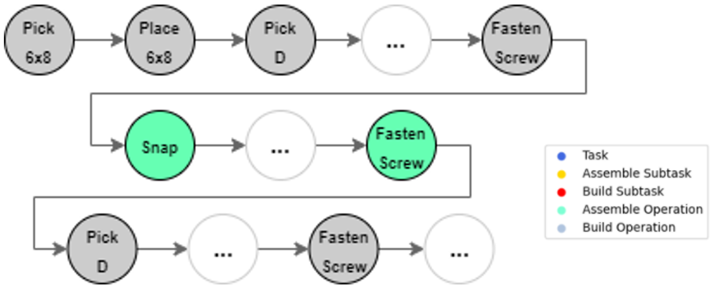


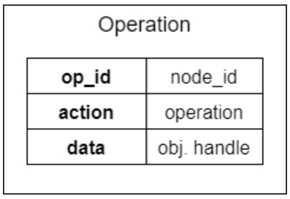
Figure 5.5: Example of motion sequence, formed in the graph translation process and containing only operation nodes.

From a task plan, i.e., node sequence, obtained in the previous module, a new node structure is created using only the operation nodes, more specifically the build operation nodes and the assembly operation nodes, and edges connecting them in the order determined by the task plan. Figure 5.5 shows a segment of a motion sequence, obtained from a task plan for the table object. Following this process, the remote API client can be created, the connection established, and the simulation started, as explained in the following Section.

The translation process begins with the simulation running, starting with the first node in the obtained motion sequence. From the node attributes, the necessary information to execute the operation in the simulation is extracted into a data structure. This information consists of the node name, action type, e.g., Pick, and additional data, which varies according to the action type, e.g., the object handle(s) that intervene in the operation.

The described data structure, which is represented in Figure 5.6 is then packed using the *packb* method from the *MessagePack* python package, which offers an efficient binary serialization format that enables data exchange among multiple programming languages. Finally, the obtained data package is sent to the server which handles the received order. The node name is also sent as a string value that is meant to be the identifier of the operation. This string value plays a part in handling the execution of the operation in the simulator and reporting to the client when the operation has terminated, by being returned as the string signal value, as explained in Sections 5.3 and 5.4.

The data package is sent by using the remote API function for python *simxCallScriptFunction*, to call a function that was implemented on the server side that unpacks the data and adds it to a simulation variable (Lua table) that holds information on all operations that have been sent and have not been executed. The node name, i.e., operation identifier, is sent by calling another server function that places the string in a queue (also a Lua table) of operations to execute, thus it represents the order for the simulator to execute the operation. After sending this request, the client goes into an infinite loop until the string signal starts streaming the value equal to the operation identifier string. In other words, after sending the operation data and execution order, the client waits for the operation termination signal to proceed.



Operation

op_id	node_id
action	operation
data	obj. handle

Figure 5.6: Structure that holds the necessary information to execute the motion operation.

After the first node's motion operation has been executed in the simulator, the algorithm skips to the successor node to repeat the same process. The algorithm loops through all the nodes in the sequence until it finds a node without a successor and the task is considered to have finished executing.

5.3 Remote API setup

This section explains the setup and functionality of the remote API client and the corresponding server on the simulation side. As explained in 2.3, CoppeliaSim supports several approaches for remote API control of the simulator. The legacy remote API option was chosen for its easy use and integration with different programming platforms. It is necessary to include some files in the directory of the python module that establishes the remote API. CoppeliaSim offers instructions to do just that, as was previously mentioned in Section 2.3.

The client is set up after the task plan translation into motion plans. The process on the client side is shown in Figure 5.7 Firstly, all open connections are closed. This is done to prevent the establishment of a connection to an undesirable server that is already opened, e.g., a different CoppeliaSim scene. Subsequently, the client is created and attempts a connection to the server, which results in the report of an error message in case of failure.

With the connection established, the client orders the simulation to start, and a string signal begins being streamed between the client and the server. The string signal is created on the server's side and contains a string value that can offer the client status information on processes it wants to control and oversee. Therefore, it can be used to communicate important information to the client, e.g., when an operation has finished being executed. Before advancing in the process, the client waits for the string signal value to be equal to the string "ready." This means the server is signaling the client that its initialization process has been complete and it is ready to receive orders from the client.

Subsequently, the graph translation module prepares the motion data from the first node and sends the data package and node name to the server as detailed in Section 5.2. The remote API client then waits for the string signal value that is being streamed to be changed to the value that was previously sent to the server, that is, the node name. If the current node has a successor, the process repeats itself for the next node. When a node without successors is reached, the client discontinues the string signal and stops the simulation. Finally, the client closes the connection to the CoppeliaSim server.

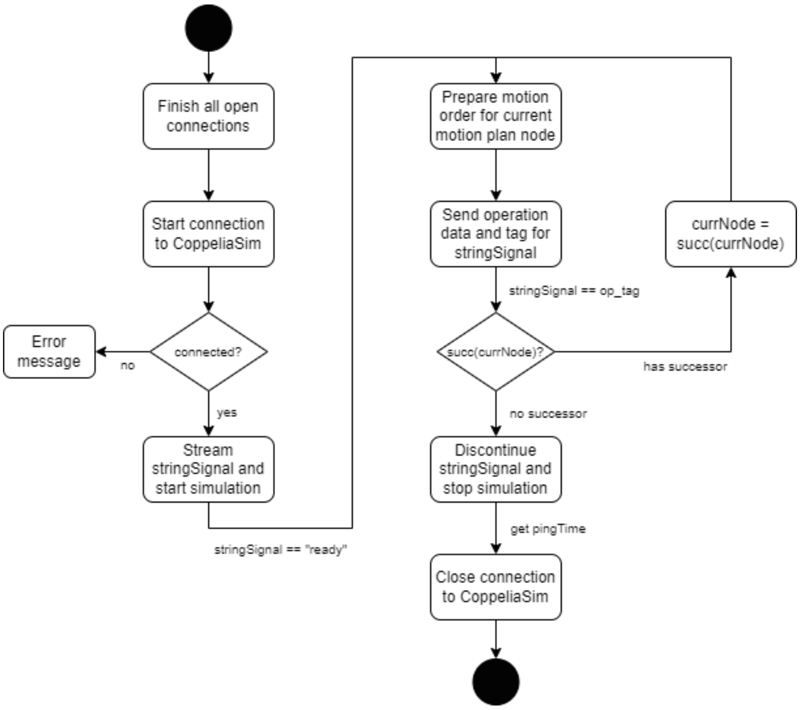

Finish all open connections
Start connection to CoppeliaSim
connected?
no
Error message
yes
Stream stringSignal and start simulation
stringSignal == "ready"
Prepare motion order for current motion plan node
Send operation data and tag for stringSignal
stringSignal == op_tag
succ(currNode)?
has successor
currNode = succ(currNode)
no successor
Discontinue stringSignal and stop simulation
get pingTime
Close connection to CoppeliaSim

Figure 5.7: Process from the remote API client side.

On the side of the server, shown in Figure 5.8, the process begins the establishment of the connection performed by the client. Thereafter, the server waits for the simulation to be started by the client. After this start, the simulation variables are initialized, and the IK environment is prepared. Once the initialization is completed, the string signal used for communicating with the client is created and set to "ready".

Subsequently, the server looks for operations that may have been sent from the client. These operations' identifier strings are stored in a queue, and their motion information is saved in an operation data Lua table, as mentioned in Section 5.2. If an operation identifier is found in the queue, the movement generation module pulls the corresponding operation information from the operation data table and executes the operation. The identifier and data are removed from the queue and table, respectively and, when the operation is terminated, the string signal value is updated with the value of the identifier. This process occurs in an infinite loop until the simulation is stopped by the client.

5.4 Movement generation

While the previous sections described the simulation control process and the remote API establishment, this section describes the motion generation on the side of the server. In other words,

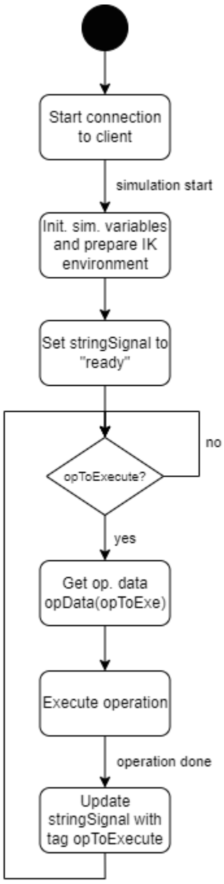

Start connection to client
simulation start
Init. sim. variables and prepare IK environment
Set stringSignal to "ready"
no
opToExecute?
yes
Get op. data opData(opToExe)
Execute operation
operation done
Update stringSignal with tag opToExecute

Figure 5.8: Process from the remote API server side.

this section explores the implemented Lua scripts on the side of the simulator. A child script attached to the UR5 parent object was developed. The main function of the script include:

- A main coroutine function, created by the sysCall_init function, where the main process occurs;
- The sysCall_init function, a built-in function which runs once at the start of the simulation;
- The sysCall_actuation function, a built-in function that runs continuously and is responsible for checking for errors that may occur in the main coroutine.

In addition to creating the main function, the sysCall_init function is responsible for initializing some simulation variables, namely the variables containing the handles of the robot joints, tip, target, and base, as explained in Section 2.3, as well as the Lua table variables that will store the orders sent from the client. The IK environment is then set up, following the guidelines also mentioned in Section 2.3. Finally, the string signal that works as the means of communication with the client is also created at this stage. The main coroutine function runs an infinite loop searching for operations in the opToExecute queue, as explained in Section 5.3.

The two functions called on the client's side to send the operation information and orders are also implemented in this script. One of the functions is responsible for storing the operation data in a Lua table (*allOpData*) with the operation identifier serving as a hash key, while the other stores the operation identifier in a queue of operations to be executed (*opToExecute*). The gripper parent object has its own child script responsible for controlling the action of the gripper actuator. Another function in the UR5 child script is responsible for sending data to the gripper control script on whether to open or close and the maximum velocity and force.

Finally, two movement generating functions were implemented, one for picking operations and one for placing operations. When an operation is received in the operation queue, the main coroutine calls the corresponding function, which passes along the data required to execute the movement. These functions use a moveToPose built-in function to generate the robotic trajectories to the intended position and orientation. For the picking operations, the robot starts by moving its tip, with the gripper opened, to an approximate pose to the object in question, then it moves to the grasping pose and closes the gripper. Finally, the robot returns to a neutral position carrying the object. A similar process takes effect for placing operations, which always occur right after picking operations. The robot starts by moving its tip close to the target pose, and then it moves to the exact placing pose and opens the gripper. The robot turns back to a neutral position with the object in place.

Chapter 6

Experiments and Results

In an effort to test and validate the developed work, two prototype objects from the HRC model set case study were chosen, the table and the shelf. The input configurations for these two objects were designed and experiments were conducted in order to obtain some relevant results and to draw informative conclusions.

6.1 Experiment definition and input design

This section presents the two cases where experiments were conducted. It demonstrates the common properties and the main differences between the two cases, as well as defining the necessary system inputs for each one. The chosen assembly objects were the following:

- Case 1: Table assembly, presented in Figure 3.2a;
- Case 2: Shelf assembly, presented in Figure 3.2c.

```
<comp_config>
  <designation>6x8</designation>
  <component_type>Plywood</component_type>
  <recurrence>1</recurrence>
  <connection_number>4</connection_number>
  <attach_point type="topL">
      <connection>T</connection>
      <specification>std</specification>
   </attach_point>
  <attach_point type="topR">
      <connection>T</connection>
      <specification>std</specification>
   </attach_point>
  <attach_point type="botL">
      <connection>T</connection>
      <specification>std</specification>
   </attach_point>
  <attach_point type="botR">
      <connection>T</connection>
      <specification>std</specification>
   </attach_point>
</comp_config>
```

```
<comp_config>
  <designation>D</designation>
  <component_type>Dowel</component_type>
  <recurrence>4</recurrence>
  <connection_number>2</connection_number>
  <attach_point>
      <connection>T</connection>
      <specification>std</specification>
   </attach_point>
  <attach_point>
      <connection>F</connection>
      <specification>std</specification>
   </attach_point>
</comp_config>
```

Figure 6.1: Component configurations for Case 1.

In both cases, the operations input as defined in Section 4.1 was the same, given that most of the focus of this work was in addressing variability at a higher level of the process, i.e., the

```
<comp_config>
  <designation>6x8</designation>
  <component_type>Plywood</component_type>
  <recurrence>1</recurrence>
  <connection_number>4</connection_number>
  <attach_point type="topL">
     <connection>T</connection>
     <specification>std</specification>
  </attach_point>
  <attach_point type="topR">
     <connection>T</connection>
     <specification>std</specification>
  </attach_point>
  <attach_point type="botL">
     <connection>T</connection>
     <specification>std</specification>
  </attach_point>
  <attach_point type="botR">
     <connection>T</connection>
     <specification>std</specification>
  </attach_point>
</comp_config>
```

```
<comp_config>
  <designation>6x8</designation>
  <component_type>Plywood</component_type>
  <recurrence>1</recurrence>
  <connection_number>4</connection_number>
  <attach_point type="topL">
     <connection>S90</connection>
     <specification>std</specification>
  </attach_point>
  <attach_point type="topR">
     <connection>S90</connection>
     <specification>std</specification>
  </attach_point>
  <attach_point type="botL">
     <connection>S90</connection>
     <specification>std</specification>
  </attach_point>
  <attach_point type="botR">
     <connection>S90</connection>
     <specification>std</specification>
  </attach_point>
</comp_config>
```

```
<comp_config>
  <designation>D</designation>
  <component_type>Dowel</component_type>
  <recurrence>4</recurrence>
  <connection_number>2</connection_number>
  <attach_point>
     <connection>T</connection>
     <specification>std</specification>
  </attach_point>
  <attach_point>
     <connection>S90</connection>
     <specification>up</specification>
  </attach_point>
</comp_config>
```

```
<comp_config>
  <designation>D</designation>
  <component_type>Dowel</component_type>
  <recurrence>4</recurrence>
  <connection_number>2</connection_number>
  <attach_point>
     <connection>S90</connection>
     <specification>down</specification>
  </attach_point>
  <attach_point>
     <connection>F</connection>
     <specification>std</specification>
  </attach_point>
</comp_config>
```

Figure 6.2: Component configurations for Case 2.

sub-task level. However, this does not mean that the planning graphs will be exactly the same at the operation level. Instead, it implies that there is a common configuration at the operation level so that an informative analysis can occur at the higher level.

As for the task input, the screw configuration aspect is the same, given that for each type of connection in the case study (bracket and plywood or bracket and dowel), the same number and size of screws are required. The bracket configuration is also the same in both experiments, given that it reflects the connection constraints of the bracket components to other structural components, which are characteristic of the case study and are not changeable. Thus, the main distinguishable factors of the two task inputs are the following:

- The list of components that form the object, which can be consulted in Table 3.1;
- The component configurations of the structural components, which hold the connection instructions for the assembly;
- The part creation mode, fully automatic for Case 1 and manual in Case 2.

For Case 1, the task input includes a component configuration for the 6x8 plywood, with the specified destined connection (Top bracket) for each attach point, as well as a component configuration for all the dowels (therefore holding a recurrence of 4) with a Top bracket on one side and a Foot bracket on the other. All connection specifications in this scenario are standard (std), given each bracket in the object only holds one connection to a dowel and one connection to a plywood, thus not requiring a distinction. Figure 6.1 shows the sections of the XML input containing this information.

For Case 2, the task input includes the same component configuration for a 6x8 plywood connected to the Top brackets with a recurrence of 1 and three additional configurations: one for a 6x8 plywood connected to S90 brackets with a recurrence of 1, one for a dowel connected to a Top bracket and an S90 bracket with a recurrence of 4 and finally one for a dowel connected to an S90 bracket and a Foot bracket, also with a recurrence of 4. In this scenario, additional specifications are used for connections between dowels and S90 brackets, given that this bracket supports two connections of this type that may need to be distinguished. Thus, the up and down specifications are used to establish this difference. The input information is represented in Figure 6.2.

Table 6.1 summarizes the main distinctions between the two objects, containing the total number of components and connections required for their assembly. Given the approach treats screws as accessory components, it is more relevant to exclude them from the total number of components when comparing different objects. Therefore, we can say the table has thirteen main components, while the shelf is made up of twenty-two main components. Larger objects with more components may lead to a higher number of created parts and more iterations necessary to assemble the object, resulting in a more complex planning problem.

Given the proposed approach to discretization, the total number of connections and the connection configurations of the brackets are also directly tied to the planning problem complexity. While the table task has only four connections to solve, the shelf task needs to handle twelve, i.e., three times the number of connections. As for the brackets, i.e., link components, while the most complex bracket type for the table (T bracket) only connects two structural components, the shelf scenario contains the S90 bracket which holds three-way connections between components.

Table 6.1: Important numbers that characterize both cases.

Assembly object	**Total number of components**		**Total number of connections**	**Bracket with max. number of connections**
	With screws	**Without screws**		
Table	29	13	4	T (2)
Shelf	54	22	12	S90 (3)

6.2 Task discretization and formalization

This section addresses the results obtained in Cases 1 and 2 with regard to the formalization of the tasks into the assembly strategies that are used to configure the sub-task nodes in the graph. As explained in Section 4.2, the first step in this process is the combination of structural components and link components to form intermediate parts, as described in Section 4.2. In Case 1, this process is automatic and the results are presented in Figure A.1. In Case 2, the parts shown in Figure A.2 were given to the system manually. Table 6.2 holds an account of the results obtained from experiments performed along this process.

Table 6.2: Summary of the results obtained from the task discretization experiments. Successful processes in green and limitations in yellow.

Test	**Table**		**Shelf**	
	Results	**Conclusion**	**Results**	**Conclusion**
Part creation	20 (Automatic)	Predicted number of parts obtained	7 (Manual)	Predicted number of parts obtained
Recipe creation	16 recipes	Predicted number of possibilities	22 recipes	Predicted number of possibilities
Recipe structure	Elements: 5 Assemblies: 4	Contains the full required structure	Elements: 10 Assemblies: 9	Missing 3 assemblies (should be 12)

The part creation process in Case 1 was proven successful, given it resulted in obtaining all possible combinations of intermediate parts. This provides the opportunity to study the best way to arrange the process at a high level. In other words, using these parts may offer us information on the best ways to attach components in multiple pieces that need to be connected to form the final object. In Case 2, due to the high number of components leading to a vast number of combinations, parts were inserted manually into the program. Although this does not provide a complete account of the domain of combinations, it makes it possible to focus the analysis on the more sensible and less redundant solutions.

With the intermediate parts formulated, the next step is the assembly of those parts to form the object fully. This process, explored in Section 4.2, takes the intermediate parts and combines them to form recipes for graph creation. The resulting recipes for Case 1 are presented in Figure A.3. The results show that all combinations of recipes were able to be created, providing the full scope of high-level strategies to decompose the final object.

Results from Case 2 were divided between Figures A.4, A.5 and A.6. The results show that despite being able to contain all of the elements required to solve the problem, the assemblies are not complete. This is due to a limitation in the implementation regarding the condition to check for finalized recipes. As they are at the moment, the recipes would lead to plans that form the intermediate parts and connect them but miss some snap and screw operations along the lower rack of the shelf. To solve this issue, the program would need to figure out the location of the missing attachments and simply add them to the current finalized recipes.

With the results taken into account, one can conclude that the developed module is fully capable of addressing lower complexity assembly tasks but still requires a few minor adjustments to be scalable to object configurations with a large number of connections between components, especially scenarios with more complex link components. With regard to the time efficiency of the algorithm, another expected conclusion was reached, given that the algorithm is fairly quick for simpler scenarios but for complex tasks, it requires a long time to compute all of the combinations. In the conducted experiments, the table assembly formalization process takes around a single second to compute, while the same process for the shelf - with an already manually limited number of parts - usually takes more than a minute. The exact times of an iteration of the code were the following: 1.0996 (table) and 64.2139 seconds (shelf).

Table 6.3: Summary of the results obtained from the graph and plan generation experiments, as well as tests in graph translation and motion simulation. Successful processes in green and limitations in yellow.

Test	Table		Shelf	
	Results	Conclusion	Results	Conclusion
Graph generation	16 graphs	All graphs generated as expected	22 graphs	All graphs generated as expected
Graph structure	Missing branches: 0	Graphs include all operations	Missing branches: 3	Incomplete graph due to recipe limitation
Plan generation	No. of nodes equal to graph Ordering constraints met	Viable plans generated	Missing some expected nodes Ordering constraints met	Incomplete plans due to recipe limitation
Graph translation	Errors in transference of motion information: 0	Full integration of task and motion modules	Yet to be tested due to the previous limitations Results expected to be similar for all objects	
Motion simulation	Simulation times: unreliable	Need for improved simulation performance		

6.3 Graph and plan generation and motion simulation

This section gives an account of the main results obtained from the whole process, starting with the use of the recipes to generate graphs and ending with the final plans. The objective of these tests was to start by presenting two of the obtained graphs from each case, making it possible to establish comparisons of graphs within the same assembly task and to compare the behavior of the graphs for different scenarios. Then, for each case, a plan should be taken from each graph and go through the graph translation and movement generation modules, after which its lead time should be calculated. The main results can be found in Table 6.3.

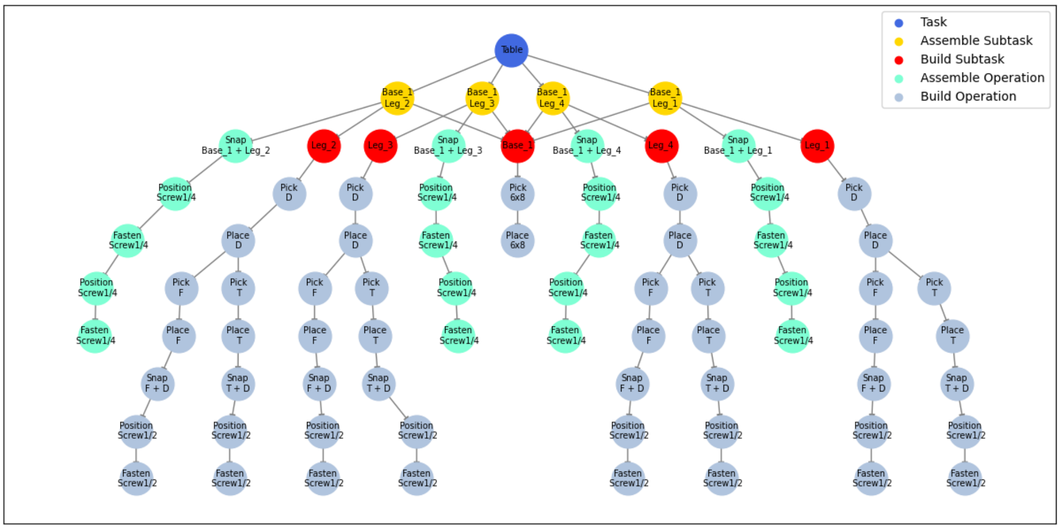


Figure 6.3: Graph A from Case 1.

To explain the results for the object in Case 1, two graphs were extracted: Graph A in Figure 6.3 and Graph B in Figure 6.4. Both graphs can solve the problem. However, they have different approaches. The graphs are generated from recipes that hold distinct elements and assemblies. The recipe setup ends up working as a graph constraint that defines how components will be attached at first, or in other words, what intermediate parts are built. Plans generated by Graph A start by

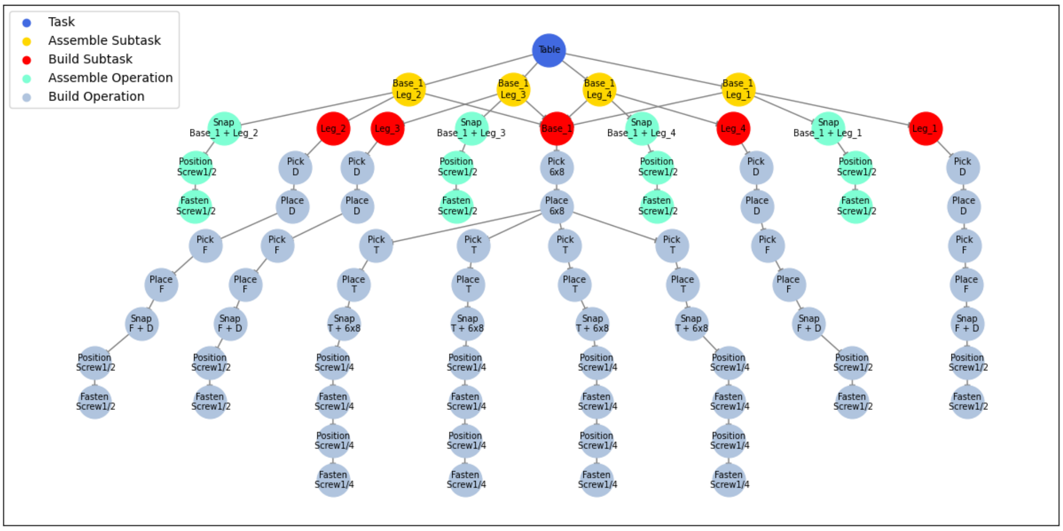


Figure 6.4: Graph B from Case 1.

building the table's legs through the attachment of the top and foot brackets to the dowel and then lead to the assembly of those parts. On the contrary, Graph B begins by building the table's base - through the attachment of the plywood and the top brackets - and separately the four simple legs - through the attachment of the foot bracket to the dowel.

The obtained results from this case correspond to the foreseen outcome: a graph with four assembly sub-task nodes - based on the recipe assemblies - and five build sub-task nodes - based on the recipe elements that are decomposed into assembly operations and build operations setup by the input configuration. Therefore, one can conclude the system succeeded in generating variability in the plans and flexibility of the process, offering different forms to break down the process, that can be adapted to different scenarios. The plan generation module was able to extract viable plans from the graphs in accordance with the graph structure, the rules for its traversal and ordering constraints, namely in regards to the unblocking of certain paths from the top node to the leaf nodes.

With regard to Case 2, even though all the necessary graphs can be obtained, they are all missing some operations necessary to generate complete task plans. This is due to a propagation of the limitation in the discretization process that creates incomplete recipes, which in turn result in the formation of incomplete graphs. We can therefore conclude that the process of graph formation is itself successful. However, it is affected by the previous limitations of the system. As Table 6.2 shows, the recipes for the shelf were three assemblies short. For this reason, three assembly sub-tasks and the corresponding branches of operations they formed were missing from the graphs generated for this object, as can be seen in Table 6.3. Going by the analysis of one of the table graphs from Case 1 in Figure 6.3, one can see that a missing assembly sub-task means a missing snap, position screw, and fasten screw operation. This is why it is crucial for the missing three recipe assemblies to be added; otherwise, the lower rack of the shelf would be connected to the top rack only through a single dowel. Thus, the correct graph structure for the shelf task should contain twelve assembly sub-task nodes and ten build sub-task nodes, as well as all of their corresponding branches of assembly operation nodes and build operation nodes. To obtain the complete graphs, one would need to fix the task discretization issue that is affecting the recipe formation.

As explained in Chapter 5, generated plans are translated into motion sequences and sent to the simulator through a remote API for robotic movement simulation. The expected result from the simulator was for it to execute the received operation orders and send back to the client information on the duration of the robotic operations. For Case 1, although the connection between the client and the server worked perfectly, e.g., communicating when an operation has been completed, and no errors in translating the motion information were found, viable operation times could not be retrieved due to an unforeseen simulator issue while running the simulation, related to the performance of the simulation and robot behavior. Therefore, the graph translation process proved to be successful, while the motion simulation process still carries some limitations. Due to time restrictions and the previously mentioned limitations, it was not possible to achieve results for the remaining part of the tests in Case 2. However, given the generalized nature of the translation process, one would expect the results to be identical for the graph translation process. On the other hand, the motion simulation is expected to have the same setbacks as in Case 1.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this dissertation, a task-motion planning framework for collaborative assembly scenarios was developed to find a generalized solution for planning in this field. The developed work focuses on some important aspects of task planning and addresses the increasing demand for flexibility and customizable processes. Research was carried out on task and motion planning existing approaches as well as task discretization and formalization methods and motion simulation. The main focus was placed on hierarchical task representation methods, namely Hierarchical Task Networks and graph-based planning.

The proposed solution consists of an integrated approach to task-motion planning, which formalizes assembly tasks from inputs that offer a build of materials and information on product procedure and constructs hierarchical planning graphs to generate task plans that can solve the problem. The robotic operations are simulated using the CoppeliaSim simulation platform. The motion simulation is meant to scrutinize the task plans and offer back the task planning module information to support the choice of the task plans.

Our formalization allows for an easy and intuitive form of providing inputs so that any assembly task is discretized and a task graph and plan formulated. From the conducted experiments, the main advantage of the developed system lies in the potential for flexibility and customization of the task planning process, which is obtained by the task formalization and graph generation modules.

The solution still has some limitations that need to be addressed when it encounters scenarios with a high number of components and complex brackets which hold a lot of connections in order to make sure all attachments between components are performed. Concerning motion, there are also some limitations in the generation of the movement on the side of the simulator that should be improved.

7.2 Future Work

Due to time restrictions and some of the limitations of the developed algorithm, the testing phase of this project went through some complications. Therefore the first thing to address regarding future work would be to fix the limitations that are holding the program back so the tests can be performed more smoothly. In addition to that, different tests should be developed to explore the main advantage of the approach: the variability of the process. Further tests using larger objects and scenarios as well as more accurate studies regarding the scalability of the solution should then be performed.

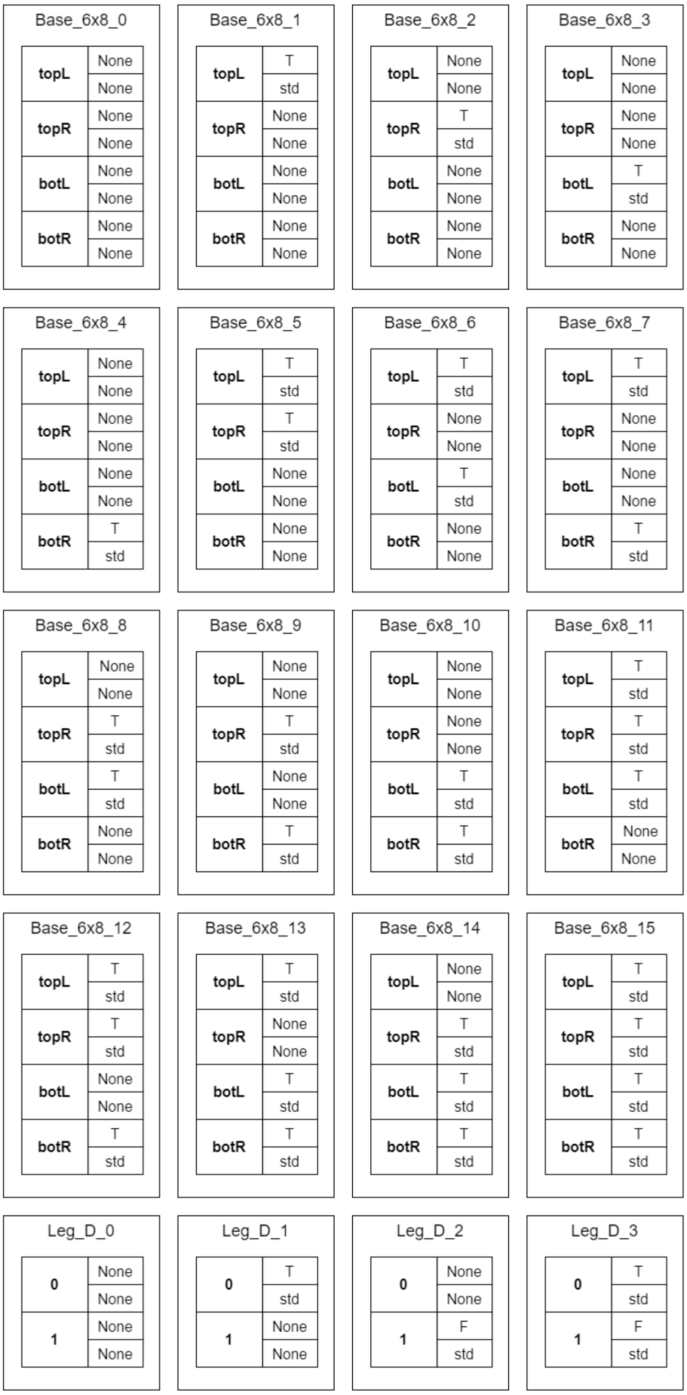
An interesting area of future development would be an algorithm that would optimize the method of part creation and detect redundant recipes that could be dismissed by the system. This could be an important tool to extend this solution to larger objects, given it would not necessarily take into account the entire state space of strategies available and would instead contemplate the ones that would offer the most variability. This addition would make the solution fully automatic even for task objects with a very large number of components and connections.

Task allocation is another aspect that should be pursued. In the conducted experiments, the input that configured the low-level operations was fixed, and therefore robotic, and human operations were the same along the process. Merging variability at the top and the bottom of the process may be the best way to achieve a truly adaptable approach for all kinds of scenarios.

With regard to the motion side of the solution, a more robust motion planning layer on top of the motion simulation should be implemented. CoppeliaSim offers the OMPL plug-in, which can be used in conjunction with the IK plug-in to generate more efficient robotic trajectories. The simulator is also very useful for collision detection, which can be an important aspect of task planning validation. Adding these features to the motion planning approach could greatly improve the plan validation capabilities of the system.

Appendix A

Appendix



Base_6x8_0	
topL	None None
topR	None None
botL	None None
botR	None None

Base_6x8_1	
topL	T std
topR	None None
botL	None None
botR	None None

Base_6x8_2	
topL	None None
topR	T std
botL	None None
botR	None None

Base_6x8_3	
topL	None None
topR	None None
botL	T std
botR	None None

Base_6x8_4	
topL	None None
topR	None None
botL	None None
botR	T std

Base_6x8_5	
topL	T std
topR	T std
botL	None None
botR	None None

Base_6x8_6	
topL	T std
topR	None None
botL	T std
botR	None None

Base_6x8_7	
topL	T std
topR	None None
botL	None None
botR	T std

Base_6x8_8	
topL	None None
topR	T std
botL	T std
botR	None None

Base_6x8_9	
topL	None None
topR	T std
botL	None None
botR	T std

Base_6x8_10	
topL	None None
topR	None None
botL	T std
botR	T std

Base_6x8_11	
topL	T std
topR	T std
botL	T std
botR	None None

Base_6x8_12	
topL	T std
topR	T std
botL	None None
botR	T std

Base_6x8_13	
topL	T std
topR	None None
botL	T std
botR	T std

Base_6x8_14	
topL	None None
topR	T std
botL	T std
botR	T std

Base_6x8_15	
topL	T std
topR	T std
botL	T std
botR	T std

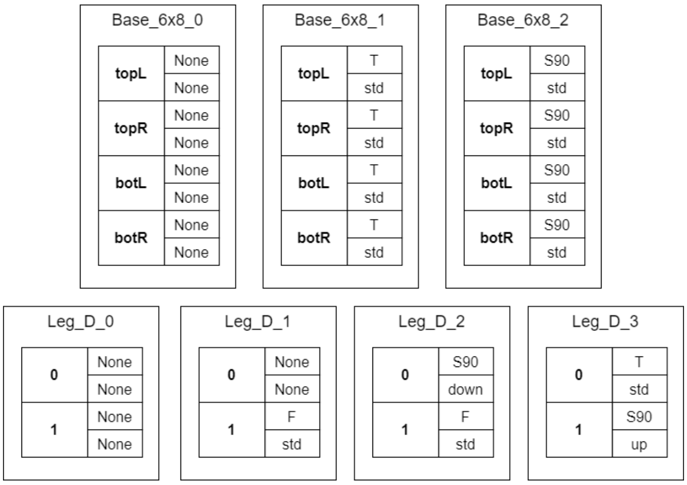
Leg_D_0	
0	None None
1	None None

Leg_D_1	
0	T std
1	None None

Leg_D_2	
0	None None
1	F std

Leg_D_3	
0	T std
1	F std

Figure A.1: Results from the automatic part creation in Case 1.



Base_6x8_0

topL	None
	None
topR	None
	None
botL	None
	None
botR	None
	None

Base_6x8_1

topL	T
	std
topR	T
	std
botL	T
	std
botR	T
	std

Base_6x8_2

topL	S90
	std
topR	S90
	std
botL	S90
	std
botR	S90
	std

Leg_D_0

0	None
	None
1	None
	None

Leg_D_1

0	None
	None
1	F
	std

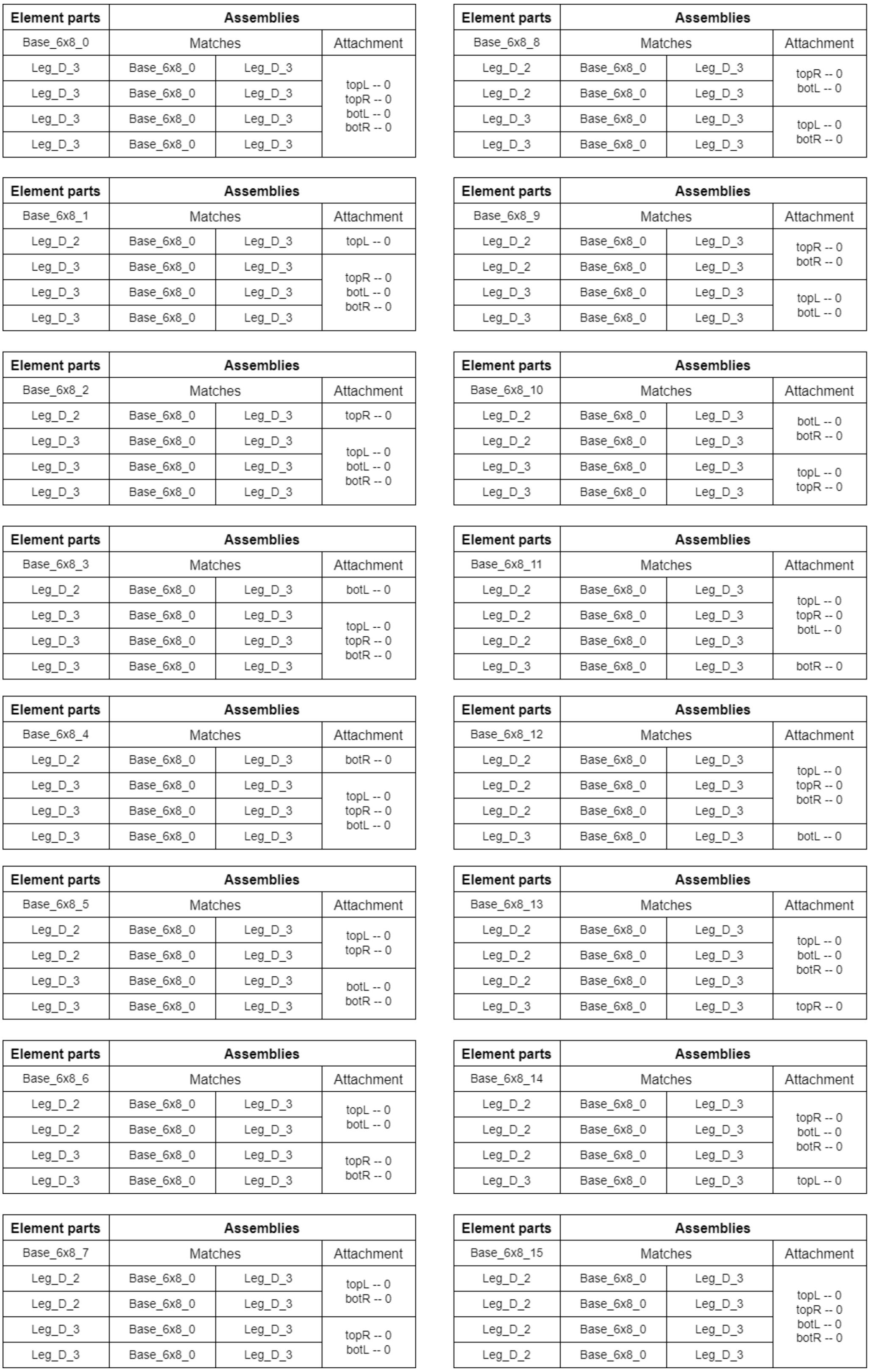
Leg_D_2

0	S90
	down
1	F
	std

Leg_D_3

0	T
	std
1	S90
	up

Figure A.2: Parts used in the formalization process in Case 2.



Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Leg_D_3	Base_6x8_0	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_1	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topL -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	topR -- 0 botL -- 0 botR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_2	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	topL -- 0 botL -- 0 botR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_3	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	botL -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	topL -- 0 topR -- 0 botR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_4	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	botR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	topL -- 0 topR -- 0 botL -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_5	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topL -- 0 topR -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	botL -- 0 botR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_6	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topL -- 0 botL -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	topR -- 0 botR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_7	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topL -- 0 botR -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	topR -- 0 botL -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_8	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topR -- 0 botL -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	topL -- 0 botR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_9	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topR -- 0 botR -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	topL -- 0 botL -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_10	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	botL -- 0 botR -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	topL -- 0 topR -- 0
Leg_D_3	Base_6x8_0	Leg_D_3	

Element parts	Assemblies		
Base_6x8_11	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topL -- 0 topR -- 0 botL -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	botR -- 0

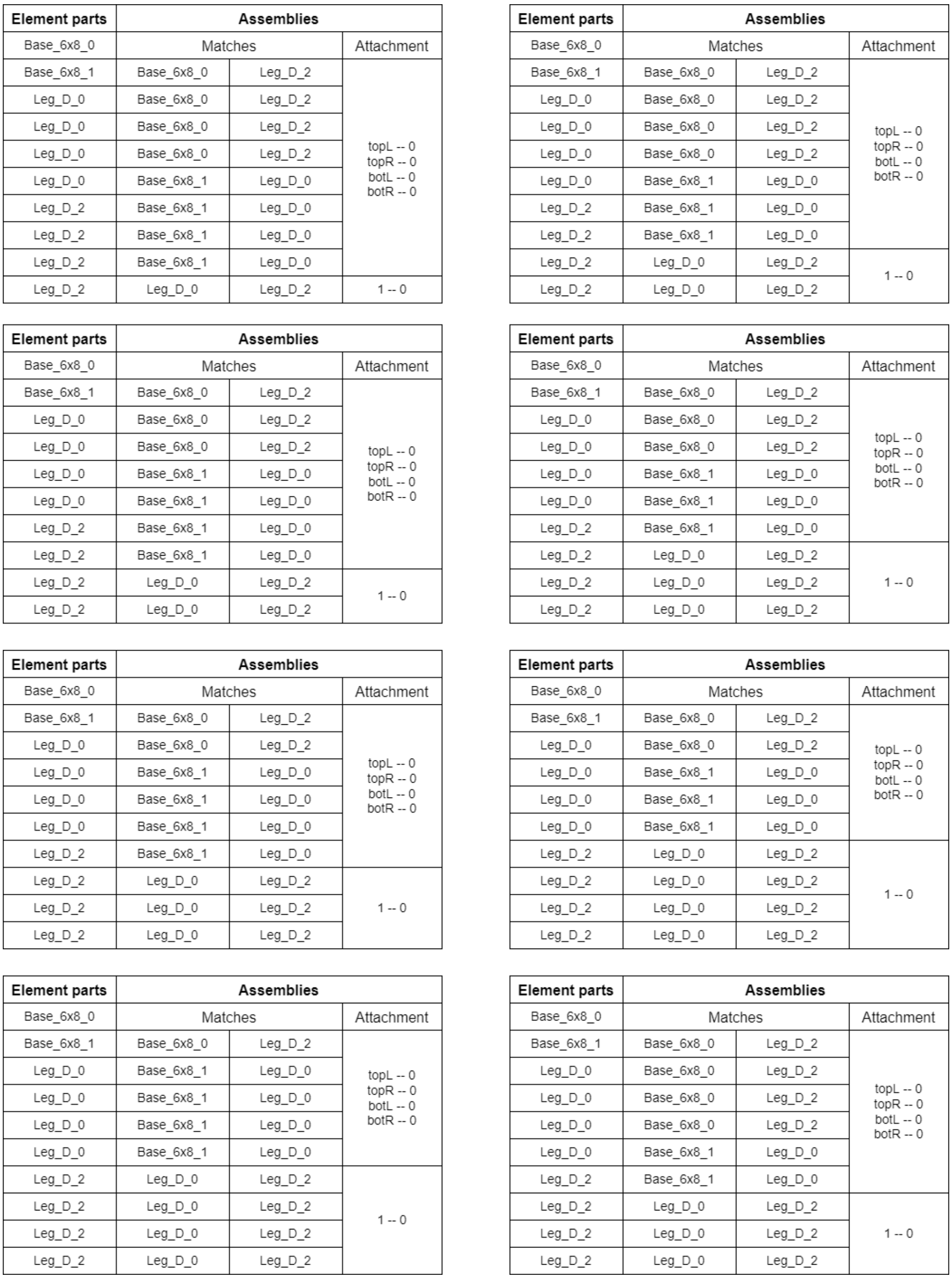
Element parts	Assemblies		
Base_6x8_12	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topL -- 0 topR -- 0 botR -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	botL -- 0

Element parts	Assemblies		
Base_6x8_13	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topL -- 0 botL -- 0 botR -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	topR -- 0

Element parts	Assemblies		
Base_6x8_14	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topR -- 0 botL -- 0 botR -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_3	Base_6x8_0	Leg_D_3	topL -- 0

Element parts	Assemblies		
Base_6x8_15	Matches		Attachment
Leg_D_2	Base_6x8_0	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_2	Base_6x8_0	Leg_D_3	
Leg_D_2	Base_6x8_0	Leg_D_3	

Figure A.3: Recipes resulting from the task discretization and formalization process in Case 1.



Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0

Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	

Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	

Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	

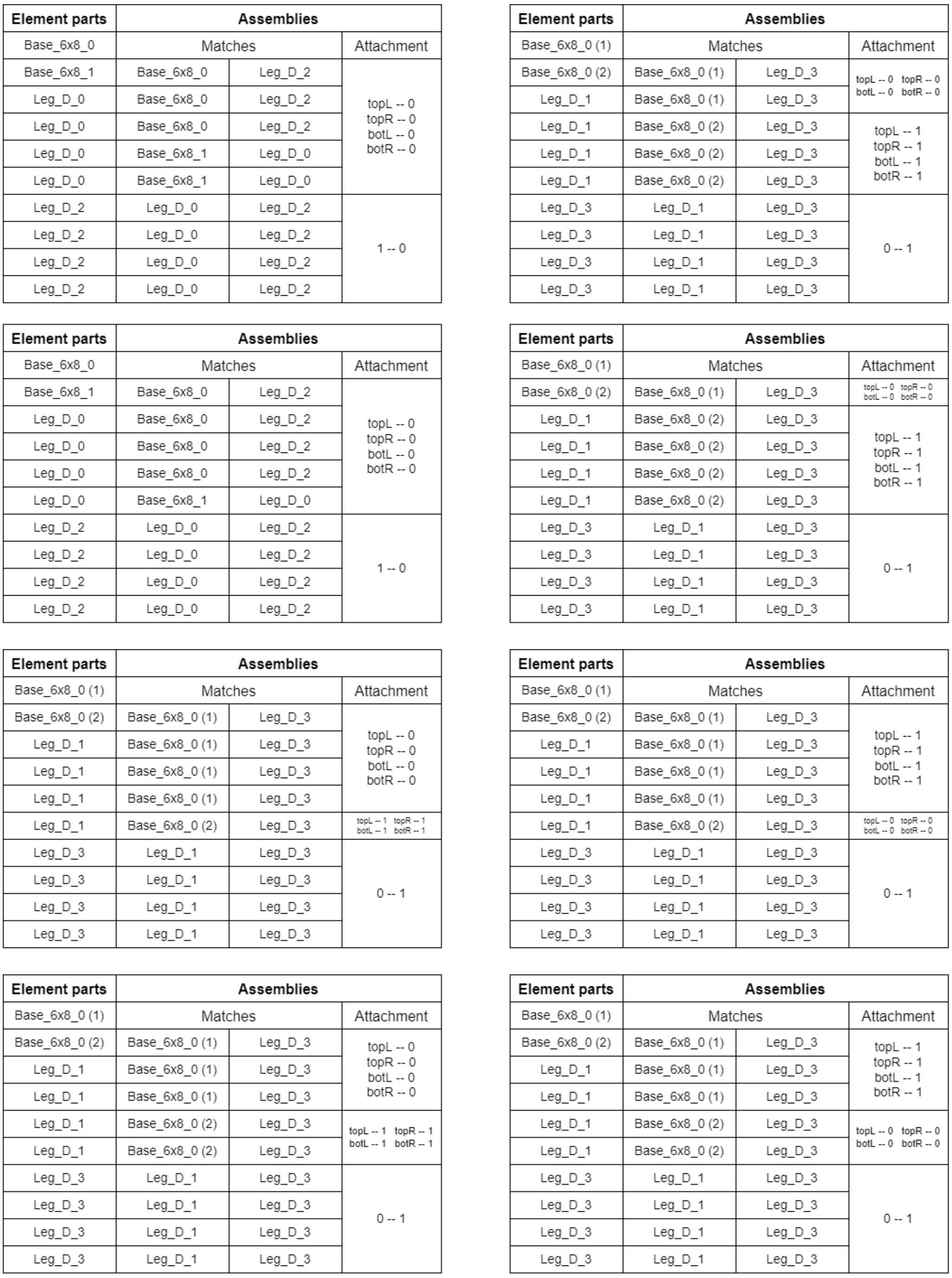
Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	

Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	

Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	

Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	

Figure A.4: Recipes resulting from the task discretization and formalization process in Case 2.



Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	

Element parts	Assemblies		
Base_6x8_0 (1)	Matches		Attachment
Base_6x8_0 (2)	Base_6x8_0 (1)	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	0 -- 1
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	

Element parts	Assemblies		
Base_6x8_0	Matches		Attachment
Base_6x8_1	Base_6x8_0	Leg_D_2	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_0	Leg_D_2	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_2	Leg_D_0	Leg_D_2	1 -- 0
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	
Leg_D_2	Leg_D_0	Leg_D_2	

Element parts	Assemblies		
Base_6x8_0 (1)	Matches		Attachment
Base_6x8_0 (2)	Base_6x8_0 (1)	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	0 -- 1
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	

Element parts	Assemblies		
Base_6x8_0 (1)	Matches		Attachment
Base_6x8_0 (2)	Base_6x8_0 (1)	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_3	Leg_D_1	Leg_D_3	0 -- 1
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	

Element parts	Assemblies		
Base_6x8_0 (1)	Matches		Attachment
Base_6x8_0 (2)	Base_6x8_0 (1)	Leg_D_3	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_3	Leg_D_1	Leg_D_3	0 -- 1
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	

Element parts	Assemblies		
Base_6x8_0 (1)	Matches		Attachment
Base_6x8_0 (2)	Base_6x8_0 (1)	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	0 -- 1
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	

Element parts	Assemblies		
Base_6x8_0 (1)	Matches		Attachment
Base_6x8_0 (2)	Base_6x8_0 (1)	Leg_D_3	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	0 -- 1
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	

Figure A.5: Recipes resulting from the task discretization and formalization process in Case 2.

Element parts	Assemblies		
Base_6x8_0 (1)	Matches		Attachment
Base_6x8_0 (2)	Base_6x8_0 (1)	Leg_D_3	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_1	Base_6x8_0 (1)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	0 -- 1
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	

Element parts	Assemblies		
Base_6x8_1	Matches		Attachment
Base_6x8_2	Base_6x8_1	Leg_D_0	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_2	Leg_D_0	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_0	Base_6x8_2	Leg_D_0	
Leg_D_1	Base_6x8_2	Leg_D_1	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_2	Leg_D_1	
Leg_D_1	Base_6x8_2	Leg_D_1	
Leg_D_1	Base_6x8_2	Leg_D_1	

Element parts	Assemblies		
Base_6x8_0 (1)	Matches		Attachment
Base_6x8_0 (2)	Base_6x8_0 (1)	Leg_D_3	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_1	Base_6x8_0 (2)	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	0 -- 1
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	
Leg_D_3	Leg_D_1	Leg_D_3	

Element parts	Assemblies		
Base_6x8_1	Matches		Attachment
Base_6x8_2	Base_6x8_1	Leg_D_0	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_2	Leg_D_0	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_0	Base_6x8_2	Leg_D_0	
Leg_D_0	Base_6x8_2	Leg_D_0	
Leg_D_1	Base_6x8_2	Leg_D_1	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_2	Leg_D_1	
Leg_D_1	Base_6x8_2	Leg_D_1	
Leg_D_1	Base_6x8_2	Leg_D_1	

Element parts	Assemblies		
Base_6x8_1	Matches		Attachment
Base_6x8_2	Base_6x8_1	Leg_D_0	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_1	Leg_D_0	
Leg_D_0	Base_6x8_2	Leg_D_0	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_1	Base_6x8_2	Leg_D_1	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_2	Leg_D_1	
Leg_D_1	Base_6x8_2	Leg_D_1	
Leg_D_1	Base_6x8_2	Leg_D_1	

Element parts	Assemblies		
Base_6x8_1	Matches		Attachment
Base_6x8_2	Base_6x8_1	Leg_D_0	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_0	Base_6x8_2	Leg_D_0	topL -- 1 topR -- 1 botL -- 1 botR -- 1
Leg_D_0	Base_6x8_2	Leg_D_0	
Leg_D_0	Base_6x8_2	Leg_D_0	
Leg_D_0	Base_6x8_2	Leg_D_0	
Leg_D_1	Base_6x8_2	Leg_D_1	topL -- 0 topR -- 0 botL -- 0 botR -- 0
Leg_D_1	Base_6x8_2	Leg_D_1	
Leg_D_1	Base_6x8_2	Leg_D_1	
Leg_D_1	Base_6x8_2	Leg_D_1	

Figure A.6: Recipes resulting from the task discretization and formalization process in Case 2.

References

[1] Panagiota Tsarouchi, Sotiris Makris, and George Chryssolouris. Human–robot interaction review and challenges on task planning and programming. *International Journal of Computer Integrated Manufacturing*, 29(8):916–931, 2016. URL: http://dx.doi.org/10.1080/0951192X.2015.1130251, doi:10.1080/0951192X.2015.1130251.

[2] Amnon Lotem, Dana S. Nau, and James A. Hendler. Using planning graphs for solving HTN planning problems. *Proceedings of the National Conference on Artificial Intelligence*, pages 534–540, 1999.

[3] E. Rohmer, S. P. N. Singh, and M. Freese. CoppeliaSim (formerly V-REP): a Versatile and Scalable Robot Simulation Framework. *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[4] Neil Dantam, Swarat Chaudhuri, and Lydia Kavraki. The Task Motion Kit. *IEEE Robotics and Automation Magazine*, 25(september):61–70, 2018.

[5] J Hoffman and B Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14(27):253–302, 2001.

[6] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. *IJCAI International Joint Conference on Artificial Intelligence*, 1:318–325, 1999.

[7] Roman Barták and Miguel A. Salido. Constraint satisfaction for planning and scheduling problems. *Constraints*, 16(3):223–227, 2011. doi:10.1007/s10601-011-9109-4.

[8] Marco Faroni, Manuel Beschi, Stefano Ghidini, Nicola Pedrocchi, Alessandro Umbrico, Andrea Orlandini, and Amedeo Cesta. A Layered Control Approach to Human-Aware Task and Motion Planning for Human-Robot Collaboration. *29th IEEE International Conference on Robot and Human Interactive Communication, RO-MAN 2020*, pages 1204–1210, 2020. doi:10.1109/RO-MAN47096.2020.9223483.

[9] Chongjie Zhang and Julie A. Shah. Co-optimizing task and motion planning. *IEEE International Conference on Intelligent Robots and Systems*, 2016-Novem:4750–4756, 2016. doi:10.1109/IROS.2016.7759698.

[10] Siddharth Srivastava, Eugene Fang, Lorenzo Riano, Rohan Chitnis, Stuart Russell, and Pieter Abbeel. Combined task and motion planning through an extensible planner-independent interface layer. *Proceedings - IEEE International Conference on Robotics and Automation*, (L):639–646, 2014. doi:10.1109/ICRA.2014.6906922.

[11] Sitar Kortik and Uluc Saranli. Robotic Task Planning Using a Backchaining Theorem Prover for Multiplicative Exponential First-Order Linear Logic. *Journal of Intelligent*

and Robotic Systems: Theory and Applications, 96(2):179–191, 2019. doi:10.1007/s10846-018-0971-9.

[12] Xiaolei Sun and Yu Zhang. A review of domain knowledge representation for robot task planning. *ACM International Conference Proceeding Series*, pages 176–183, 2019. doi:10.1145/3325730.3325756.

[13] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. *Proceedings of the National Conference on Artificial Intelligence*, 2:1123–1128, 1994.

[14] Ross A Knepper, Dishaan Ahuja, Geoffrey Lalonde, and Daniela Rus. Distributed Assembly with AND / OR Graphs.

[15] Bradley Hayes and Brian Scassellati. Autonomously constructing hierarchical task networks for planning and human-robot collaboration. *Proceedings - IEEE International Conference on Robotics and Automation*, 2016-June:5469–5476, 2016. doi:10.1109/ICRA.2016.7487760.

[16] Nathan Koenig and Andrew Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3:2149–2154, 2004. doi:10.1109/iros.2004.1389727.

[17] Olivier Michel. WebotsTM: Professional Mobile Robot Simulation. 1(1):39–42, 2004. URL: http://arxiv.org/abs/cs/0412052, arXiv:0412052.

[18] Fumio Kanehiro, Kiyoshi Fujiwara, Shuuji Kajita, Kazuhito Yokoi, Kenji Kaneko, Hirohisa Hirukawa, Yoshihiko Nakamura, and Katsu Yamane. Open architecture humanoid robotics platform. *Proceedings - IEEE International Conference on Robotics and Automation*, 1(February):24–30, 2002. doi:10.1109/ROBOT.2002.1013334.

[19] Toki Migimatsu and Jeannette Bohg. Object-centric task and motion planning in dynamic environments. *arXiv*, 5(2):844–851, 2019.

[20] Panagiota Tsarouchi, Alexandros Stereos Matthaiakis, Sotiris Makris, and George Chryssolouris. On a human-robot collaboration in an assembly cell. *International Journal of Computer Integrated Manufacturing*, 30(6):580–589, 2017. URL: http://dx.doi.org/10.1080/0951192X.2016.1187297, doi:10.1080/0951192X.2016.1187297.

[21] Nikolaos Nikolakis, Niki Kousi, George Michalos, and Sotiris Makris. Dynamic scheduling of shared human-robot manufacturing operations. *Procedia CIRP*, 72:9–14, 2018. URL: https://doi.org/10.1016/j.procir.2018.04.007, doi:10.1016/j.procir.2018.04.007.

[22] Alessandro Roncone, Olivier Mangin, and Brian Scassellati. Transparent role assignment and task allocation in human robot collaboration. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 1014–1021, 2017. doi:10.1109/ICRA.2017.7989122.

[23] Olusegun Oshin, Edgar A. Bernal, Binu M. Nair, Jerry DIng, Richa Varma, Richard W. Osborne, Eddie Tunstel, and Francesca Stramandinoli. Coupling deep discriminative and generative models for reactive robot planning in human-robot collaboration. *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, 2019-Octob:1869–1874, 2019. doi:10.1109/SMC.2019.8913974.

[24] Sofya Zeylikman, Sarah Widder, Alessandro Roncone, Olivier Mangin, and Brian Scassellati. The HRC Model Set For Human-Robot Collaboration Research. *arXiv*, 2017.