Alex Maestrini, BSc

Flexible Automation of an Industrial Assembly Task

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Steinbauer, Gerald, Assoc.Prof. Dipl.-Ing. Dr.techn.

Institute for Softwaretechnology
Head: Wotawa, Franz, Univ.-Prof. Dipl.-Ing. Dr.techn.

Graz, June 2020

Affidavit

Acknowledgments

I would first like to thank my thesis advisor Prof. Gerald Steinbauer of the Institute of Software Technology at Graz University of Technology. The door to Prof. Steinbauer's office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this work to be my own, but steered me in the right direction whenever he thought I needed it. I would like to thank the Institute of Production Engineering for their support and guidance, whenever it was needed.

I would also like to express my very profound gratitude to my family and my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Abstract

Due to the increasing demand of highly customizable, mass produced parts offered for a relatively low price, the concept of fully automated but still flexible production is becoming more important. In order to address this usually orthogonal objectives, robot systems that are equipped with intelligent planning and plan execution systems can be used.
In this thesis, a proof of concept for a system is presented, that is able to generate a full assembly plan from an appropriate initial representation of the to be assembled product with the least amount of user involvement. Moreover, the system is capable to execute the plan using a given production setting with for instance two robot arms without manual tuning. This would lead to a higher degree of automation, the reduction of preparation time and mainly a shift of freedom and responsibility from the machine operator to the product developer. Being able to reuse knowledge such as assembly related skills by sharing and collecting them, would also lead to a decrease of the workload and an increase of the flexibility of the automation engineer. The work presented in this focuses mainly on geometrical and semantical understanding of the assembly task and the involved components, derivation of an assembly plan using inferred knowledge as well as autonomous and robust execution of the planned actions based on the actual configuration in the real environment. In order to achieve this goal, a set of guidelines for the computer-aided design (*CAD*) model have been developed, that allow automated understanding of the underlying assembly process. Specially designed algorithms allow the extraction of a graph-like structure from that model, used to generate a sequential list of actions representing the full assembly plan. The entire pipeline for this automated assembly was designed, the necessary representations were defined and prototypes of tools for processing of the data were implemented. The proposed approach was applied to different types of assemblies and the resulting plans were evaluated for their feasibility and robustness. The evaluation showed that

the proposed process for obtaining a feasible assembly plan from a *CAD* drawing of the product works. Future developments may include further reduction of the user involvement as well as finalization and improvements on stability of the developed simulation.

Contents

List of Figures

List of Algorithms and Code

1 Introduction

Smart factories are being built all over the world as an important outcome of the latest industrial revolution. Whether it is computerizing assembly lines with cyber-physical systems or interconnecting independent entities and structures with others as well as with Big Data to create the Internet of Things or Internet of Systems, Industry 4.0 is affecting many different aspects of everyday life.

Led by the increasing demand of highly customizable, mass produced parts offered for a relatively low price, the concept of fully automated but still flexile assemblies is becoming more important. There are four increasing degrees of adaptability for assembly automation: fixed, programmable, lean and flexible.

Fixed assembly lines were one of the major resulting technologies of Industry 2.0, which allowed the degree of mass production that we are used to today. Aside a high initial investment cost in custom engineered, special-purpose equipment, this type of part construction is made for long product life cycles and lead to a high production rate. The downside is a low variety in production, since the production steps are fixed sequences of operations. An example of a fixed assembly line can be seen in Figure 1.1.

Programmable assembly starts with a high investment in general purpose equipment designed to accommodate a specific class of production changes. An example of such a production cell can be seen in Figure 1.2. With physical changes to the setup and changes to the executed program, different products can be assembled batchwise, which only allows a medium production volume. This leads to an inverse-proportional relation between variety and quantity.

Lean automation is a human-robot collaboration (*HRC*) focused method of assembly with a singular modular cell or a conjunction of modular cells

Figure 1.1: Fixed assembly line of Volkswagen Beetle. Source: `https://w.wiki/PXi`

with robots solving parts of the task while being aided, refilled or surveyed by humans. This is generally seen as an efficiency increase [1], since there are some tasks that the robot and the human individually are better at then the other. Since the assembly cells can either be fixed or programmable, most downsides persist. Figure 1.3 shows an exemplary *HRC* workspace.

In comparison, flexible assemblies have the same sort of high investment level as programmable assemblies but allow a continuous production flow of a mixed product variety. This can lead to slightly lower production rates on a per product basis but taking into consideration the time and effort it takes to switch batches and change the program on the machines, the overall production rates are higher, as represented in Figure 1.5. An illustration of a possible flexible assembly setup is depicted in Figure 1.4.

Regarding the variety-quantity-ratio, flexible automation has the right balance between fixed and programmable automation. It would be possible to detect product changes or even completely novel products and react to them as fast as possible with no or minimal effort regarding adaptations in from of manual labor.

Figure 1.2: Programmable assembly line. Source: `https://w.wiki/PXj`

1.1 Motivation

State of the art robot assembly knows four different methods of teaching the robot the task it needs to solve: preprogramming of the task, learning from demonstration, human-robot collaboration and declarative programming.

The first includes knowledge about offsets, distances, angles and velocities of both robot and product in combination with a perception system that can identify individual components and their pose. Most components in an industrial assembly are tiny and made of glossy, reflective metals. Object recognition providing the required level of robustness needed for such tasks, is topic of current research. Commonly, fixture arrangements of known dimensions are used to position components in a manner which facilitates the assembly. Such assembly lines do not need highly developed perception systems to identify location and orientation of parts. Instead, methods such as inductive sensors, simple *RGB* cameras or photoelectric barriers are used to identify when a picking slot is filled. The components are brought to the

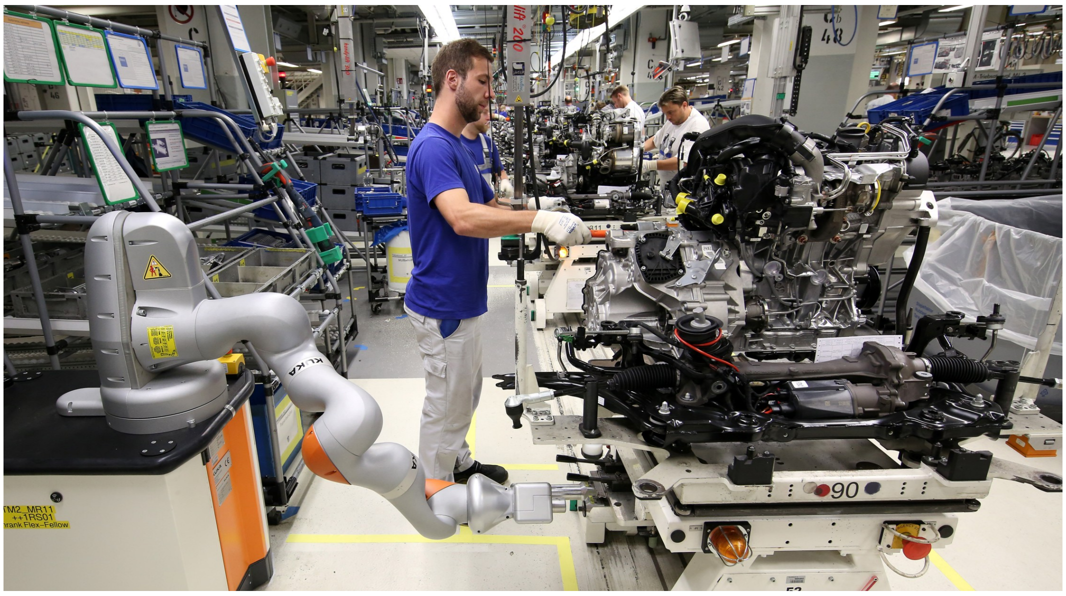


Figure 1.3: Human-robot collaboration. Source: [1]

needed locations using mechanical contraptions that constantly ensure a correct orientation for upcoming steps of the assembly task.

A second method is robots learning from demonstration. Here an instructor grabs the end effector of the robot and performs the action manually. The robot then imitates recorded joint angles and velocities to achieve the same result as previously shown. This is called kinesthetic teaching. Alternatively, teleoperation is also a possible interaction method with the robot to teach an action. The learned trajectories can be generalize with the help of machine learning methods to apply them to different kinds of object constellations.

The third possibility is human-robot collaboration. This can range from simple, repetitive patterns of action to a complex collaboration between robot and human in the same space. If the task is simple enough, safety zones and closing mechanisms are secure enough, but for an interaction in the same space, the robot needs additionally a perception system that is able to successfully detect humans.

Figure 1.4: Exemplary flexible assembly setup. Source: `https://w.wiki/PXt`

Lastly, a method of assignment description called declarative programming can be used do describe instructions. The characterization of individual task does not specify implementation details, rather it expresses *what* should be done instead of *how*. The solutions to each step are gathered automatically in form of robot skills using reasoning. Each skill needs to be implemented beforehand by the developer.

The work presented in this thesis contributes to the field of smart factories by developing a methodology and system that can extract an assembly action plan from some initial representation of the task, without the system being specifically tuned or set up for it.

This would lead to a higher degree of automation, the reduction of preparation time and mainly the shift of workload from the machine operator to the developer. Being able to reuse knowledge such as assembly related skills by sharing and collecting them, would also lead to a decrease of the workload for the developer.

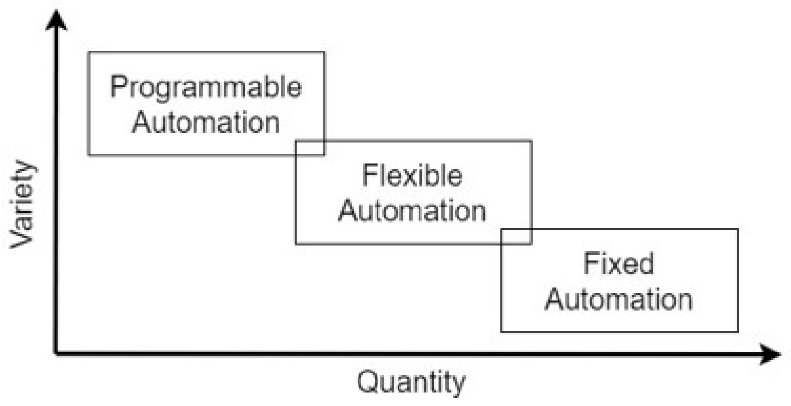


Figure 1.5: Quantity to variety ratio for flexible assembly.

1.2 Goals and Challenges

The overall goal of this thesis is to develop a proof of concept for a system that is able to generate a full assembly plan out of an initial representation of the task with the least amount of user involvement using a general dual-arm robot setup. This work focuses mainly on the following three aspects:

1. Geometrical and semantical understanding of the assembly task and the involved components. This includes knowledge and reasoning about shape, properties, function and relation of items.
2. Derive an assembly plan using inferred knowledge and subdivide it into sequences of actions in combination with the planning of concrete motions of tools and parts while minimizing manual interference and prior information about the final product or any assembly sequences.
3. Autonomous and robust execution of the planned actions based on the actual configuration in the real environment utilizing a perception system with visual sensors being able to identify and precisely localize the involved components.

Since it is intended to have as few as possible additional prior information about the task, such as knowledge about individual subtasks or additional properties of components, the utmost amount of information should be taken from the initial representation.

1.3 Contribution

In this work a methodology for the extraction of action sequences for automated assembly tasks is presented. The developed algorithms and set of rules are useful for different assembly use cases and can be integrated in Industry 4.0 applications to achieve a higher degree of automation.

The entire workflow, ranging from the set of rules guiding the computer-aided design (*CAD*) model creation to the execution of the individual actions in a simulated environment is developed and implemented as a proof of concept.

Using said guidelines, a *CAD* model with the needed information can be created, from which a directed graph-like structure of components as nodes and connections as edges is deduced. This reduces the workload of the user to creating or modifying *CAD* files. While prototyping an assembly, *CAD* models are created in every case. The imposed guidelines and rules are easy to get used to and do not imply more effort than normally once accustomed to.

This hierarchical information is then processed to obtain a sequential plan of actions. Each action is represented by a connection between two individual components of the assembly. These directed connections contain additional information about the related parts in their final position, such as a relative transformation between parent and child or the actual distance between the related surfaces. Furthermore, special constraints are added to enforce a specific sequence of action inside the plan, which can be applied using a plugin that was developed during this thesis. This can be used by the *CAD* modeler to enforce a wanted assembling order or to denote the need of moving specific components after a certain point in time during the assembly. Examples for such situations include shifting already inserted objects inside a slot to create pressure on other parts or tightening or losing a screw to be able to move already fixated objects.

As an evaluation environment, a scene with the initial state of the assembly and two UR5 arms with RG6 grippers, inspired by the Assembly Challenge from the World Robot Summit 2018, is set up in a virtual environment simulated in Gazebo. The arm and gripper motions are planned and executed using the *MoveIt!* framework. Due to time constrictions some final issues could not be overcome, which prevented the whole assembly in its final form from being built inside of the simulation. Even so, individual aspects can be visualized and used for evaluation purposes.

The main evaluation of this work was done by applying the proposed approach to different types of assemblies and by discussing the resulting plan, looking for shortcomings and further improvements. The plan extraction is developed in form of an AutoDesk Inventor 2020 plugin written in C# on Windows 10. The plan execution is developed and tested in Ubuntu 16.04 and Robot Operating System (*ROS*) [2] Kinetic Kame.

1.4 Outline

The remainder of the thesis is organized as follows. In Chapter 2 prerequisites are presented. This includes the robotic framework *ROS* [2], the simulation framework Gazebo, the motion planning and execution software *MoveIt!* and the used visualization tool rViz, the skill execution framework SkiROS [3] and the used visualization tool RQt, the UR5 robot arm model, the RG6 gripper model and the Inventor application programming interface (*API*). In Chapter 3 related research is discussed. This includes human-robot collaboration, visual perception systems, knowledge representations, planning, reasoning, dynamic motion primitives and dual arm kinematics. At the end flexible assembly and its state of the art is discussed. The problem formulation is presented in Chapter 4, including the main aspects geometrical and semantic understanding, plan derivation and robust plan execution. Moreover, the assembly task used for this work, a Belt Drive Unit (*BDU*), is described in more detail. In Chapter 5 the overall concept is presented in detail, as well as a set of rules and guidelines for the *CAD* modeler and all algorithms developed to extract and execute the plan. Chapter 6 describes in detail the implementation of the system and the resulting challenges. The evaluation of the work is presented in Chapter 7. This includes the evaluation of the plan derivation, but also of the simulation. The end of this thesis is formed by discussion about conclusions and potential future expansions in Chapter 8.

2 Prerequisites

This chapter describes concepts and software modules used for realizing the approach presented in this thesis.

2.1 ROS

The Robot Operating System is an open source robotic framework described in [2] and can be viewed as the standard for frameworks in robotics. It provides a structured communication layer mainly above Linux-based platforms. This layer is composed of a collection of master nodes and many modularly separated nodes. *ROScore*[1], a base group of nodes and programs necessary for running a *ROS*-based system, must be started before any other node and represents the core of the system. It contains a *ROS* master, a parameter server and a logging node called *rosout*. The master provides a yellow page service for managing all active partners in a *ROS* system.

The area of responsibility of individual nodes usually includes smaller tasks and calculations, while communicating with other components using messages. These messages are transmitted using a publish-subscribe protocol and represent a clearly defined data structure. The publisher notifies the master node about the existence of a topic, which is a named reference to the advertised communication channel. Other nodes can subscribe to this topic by requesting it from the master node. Every subscriber to such a topic will receive the published messages. This is a way of system-wide message transmission without sending all messages to every existing node. Since multiple nodes can publish messages on the same topic, this can be used as a method of synchronization between many nodes using services.

[1]`http://wiki.ros.org/roscore`

A node can provide a service and advertise it under a unique name. If a node sends a request to such a service, it waits for the reply of the service before continuing with the execution. Opposed to such blocking calls, non-blocking calls are implemented via a small *API*, which is presented in form of an additional *ROS* package called *action lib*[2]. It provides a preemptible task, where a node can implement an action server with many different nodes implementing an action client. The client can call accessible actions that are provided server-side via messages. Last, a parameter server is accessible system-wide, providing access to stored static values like configuration parameters.

To run a specific configuration of nodes and a master node at once, so called launch file can be created. A launch file consists of the Extensible Markup Language (*XML*) format[3].

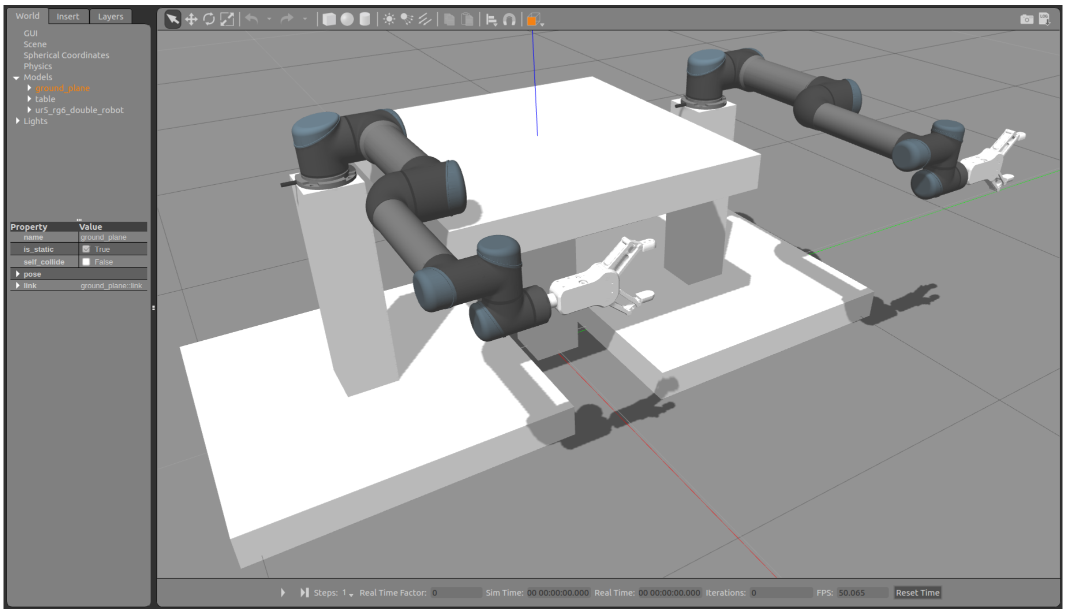
More detailed information can be found in [2] and the latest documentation is available at the *ROS* Wiki[4].

2.2 Gazebo

Gazebo is a open-source simulator used for 3D rigid body simulations of robots in a virtual environment with support for different physics engines. To be able to work with stand-alone Gazebo, a set of *ROS* packages named *gazebo_ros_pkgs* is presented by Gazebo[5].

These packages provide wrappers to access core functionalities of the simulator such as accessing data of the scene or spawning models using Unified Robot Description Format (*URDF*) for robots or Simulation Description Format (*SDF*) for other components of the simulation. These file types represent the full model with all the needed meshes for visual and collision masks, joints with offsets and rotations between links, inertial parameters, mass,

[2]https://wiki.ros.org/actionlib
[3]https://wiki.ros.org/roslaunch
[4]https://wiki.ros.org/
[5]http://gazebosim.org/tutorials?tut=ros_overview



Property | Value
name | ground_plane
is_static | True
self_collide | False
pose |
link | ground_plane::link

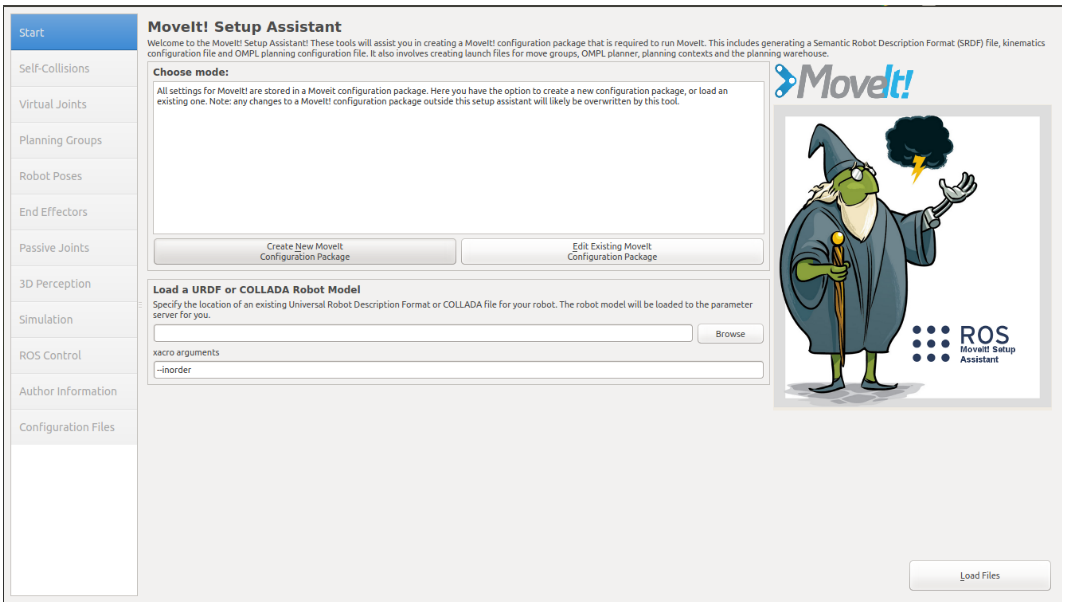
Steps: 1 Real Time Factor: 0 Sim Time: 00 00:00:00.000 Real Time: 00 00:00:00.000 Iterations: 0 FPS: 50.065

Figure 2.1: An example of a Gazebo simulation representing a possible two-arm setup for the assembly tasks.

center of mass and many more. The documentation of the *URDFormat*[6] and the *SDFormat*[7] can be found online.

Gazebo integrates with *ROS* using *ROS* messages and services. Many messages and plugins are included to provide access to data of simulated devices such as depth cameras, bumpers or laser scanners. Also, a set of predefined worlds and scenes are included in the package.

Figure 2.1 shows a possible two-arm setup for the assembly task. It contains two UR5 arms on pillars with RG6 grippers mounted on each arm, with a table in between.



MoveIt! Setup Assistant

Welcome to the MoveIt! Setup Assistant! These tools will assist you in creating a MoveIt! configuration package that is required to run MoveIt. This includes generating a Semantic Robot Description Format (SRDF) file, kinematics configuration file and OMPL planning configuration file. It also involves creating launch files for move groups, OMPL planner, planning contexts and the planning warehouse.

Choose mode:

All settings for MoveIt! are stored in a Moveit configuration package. Here you have the option to create a new configuration package, or load an existing one. Note: any changes to a MoveIt! configuration package outside this setup assistant will likely be overwritten by this tool.

Load a URDF or COLLADA Robot Model

Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot. The robot model will be loaded to the parameter server for you.

xacro arguments

--inorder

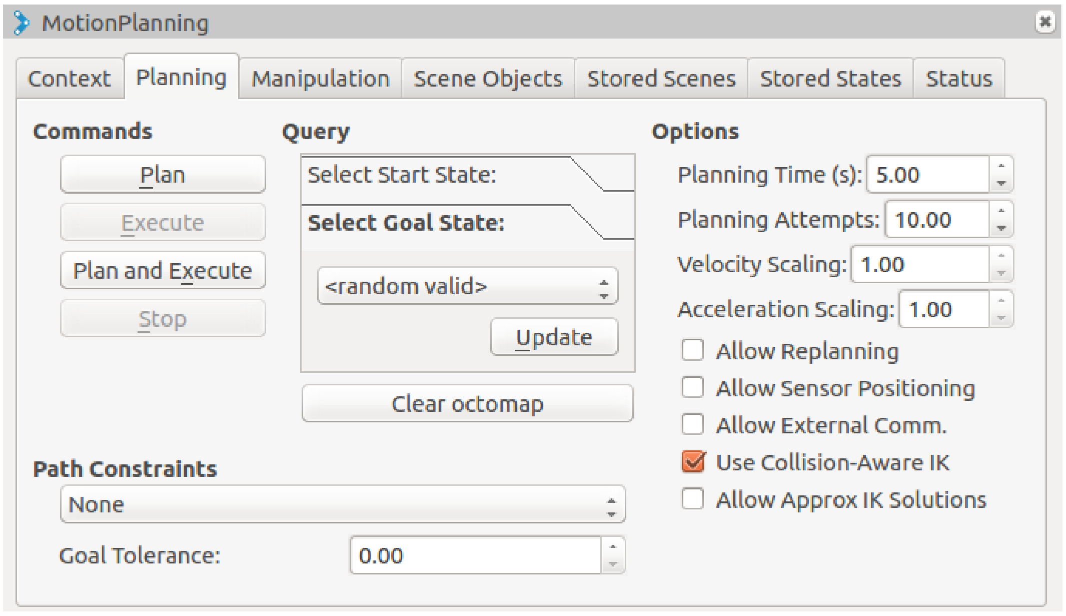
Figure 2.2: Screenshot of the *MoveIt!* setup assistant.

2.3 MoveIt!

MoveIt! is a motion planning and execution framework for manipulation of robotic arms presented in [4]. It was implemented on the arm_navigation packages in *ROS*. It contains a Planning Scene Monitor, which provides an environmental representation, including libraries for motion planning as well as the ability to monitor motion execution. It is representing the environment in two formats: a voxel grid representing most of the obstacles, and geometric primitives as well as mesh models for objects recognized and registered by object detection routines.

Motion planning takes the needed information from the planning scene. A planning scene is a kinematic and semantic description of the robot. These are presented to the framework in form of *URDF* (for the kinematic

[6]https://wiki.ros.org/urdf
[7]http://sdformat.org/spec



<random valid>

Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00

☐ Allow Replanning
☐ Allow Sensor Positioning
☐ Allow External Comm.
✓ Use Collision-Aware IK
☐ Allow Approx IK Solutions

Path Constraints: None
Goal Tolerance: 0.00

Figure 2.3: Screenshot of the *MoveIt!* rViz plugin.

representation) and Semantic Robot Description Format (*SRDF*), both represented in *XML*. The *SRDF* complement the *URDF* by specifying joint groups, default robot configurations, additional collision checking information and transforms that might be necessary to fully specify a robot's pose. In combination a *ROS* package called *xacro*[8], which can be used to simplify, shorten or modularly connect *XML* formatted files.

Using these representations of the robot, the motion planning module in conjunction with prior information about the collision objects computes into a collision-free motion plan. By default the Open Motion Planning Library (*OMPL*) is used, but other libraries can be included instead. After smoothing it, the generated trajectory is sent to a specific controller. Each separate controller in the system gets its own move group assigned. The controller executes and monitors the trajectory and reports aborted executions.

8 https://wiki.ros.org/xacro

Using the *MoveIt!* setup assistant any robot can be configured for use with *MoveIt!* by generating a *SRDF* file and other configuration files used in the planning pipeline. This step also validates the *URDF* file. A screenshot of this application can be found in Figure 2.2.

MoveIt! provides a Graphical User Interface (*GUI*) implemented as a rViz plugin. Most *MoveIt!* features are accessible and can be activated manually. This includes planning and executing of a trajectory for the given robot by either using a predefined pose or moving the end effector marker freely. A screenshot of this plugin is visible in Figure 2.3.

2.4 rViz

rViz is a 3D visualizer for the *ROS* framework. It can visualize most of the common *ROS* message types, such as point clouds or future positions of planning actions. The main use case for rViz in this project is the visualization of the *MoveIt!* planning scene using the *MoveIt!* rViz plugin. This includes the planned path before execution and the objects spawned in Gazebo in form of green colored collision objects. Additionally, it allows the general control of *MoveIt!* move groups through drag and drop of a marker, including planning and execution of end effector positions or predefined poses from the *SRDF*, which can be seen in Figure 2.3.

2.5 SkiROS

SkiROS was presented in [3] as a skill-based *ROS* platform for autonomous mission execution based on autonomous, goal directed task planning and knowledge integration.

The *SkiROS* architecture is represented in Figure 2.4, where squares represent *ROS* nodes and rectangles represent plugins. The nodes are provided by the framework, but the plugins need to be implemented by the developer. The task goals or information about the scene can be specified using a *GUI* tools or by directly accessing the *ROS* topics. The direct access and

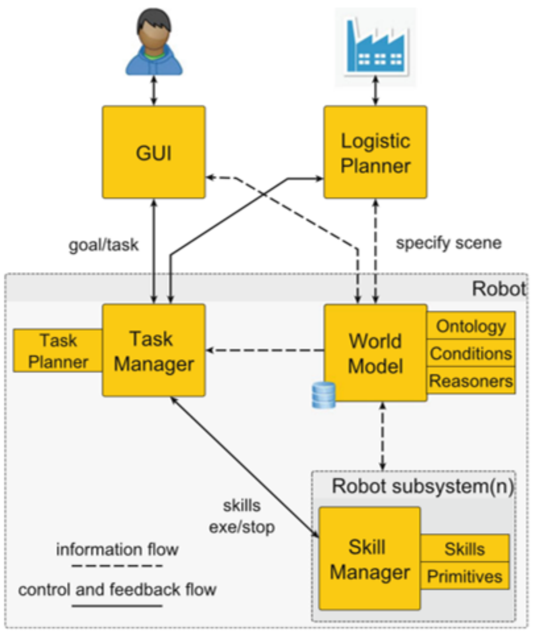


Figure 2.4: General structure of the *SkiROS* system. Source: [3]

all related *ROS* topics are described in [3] . The three major *ROS* node types regarding a robot in the *SkiROS* system are the task manager, the world model and one skill manager per robot subsystem. The task manager dispatches generated plans based on the world model knowledge to the skill mangers of the robot. The skill managers each coordinate a subset of capabilities while keeping the world model updated. The world model is the center point of the system. All knowledge is collected and shared through it.

The internal representation of a skill in the *SkiROS* system can be seen in Figure 2.5. The input of such a skill is defined by some skill-specific parameters and the initial state of the sub-module, which is controlled the skill manager. Each skill starts with a precondition check. If it is valid, the skill will be executed and simultaneously a prediction of the outcome defined by the input parameters is calculated. After the execution the final state of the robot subsystem is compared to the prediction and, if valid, returned to the world model.

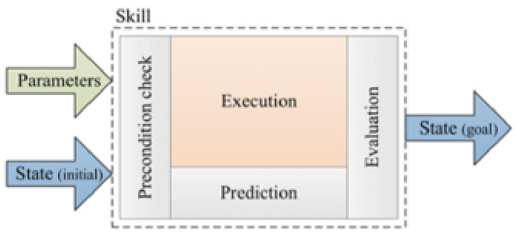


Figure 2.5: General concept of a *SkiROS* skill. Source: [3]

The robots knowledge is represented in the W3C Web Ontology Language (*OWL*)[9] standard, defined in Description Logic (*DL*), a specialization of first-order logic, which is designed to simplify the description of definitions and properties of categories. The three major aspects contained in this knowledge are the objects in the world, the robot hardware and the capabilities available to the robot, which are skills and primitives. In addition to the ontology, the following software pieces have to be created or added by the programmer during the development phase: skills, primitives, conditions, a discrete reasoner and a task planner.

Skills are actions with pre- and post-conditions that can be concatenated to form a complete plan for a task. Primitives are simple actions without pre- and post-conditions that support hierarchical composition. They are concatenated manually from the programmer inside a skill. The task planner has to support Planning Domain Definition Language (*PDDL*), which is further described in Section 3.5 and originally presented in [5].

This initial version of *SkiROS* was implemented on top of the C++ version of *ROS*. During the development of the work presented in this thesis, a second version of *SkiROS* got published[10], which is only at its demo state. The newer version is implemented on top of the python version of *ROS* and includes the concept of behavior trees, expanding the non-reactive, deliberative system purely based on *PDDL*. In conjunction with the extended behavior tree model presented in [6] a reactive behavior tree can be generated using planning. This leads to a reactive-deliberative hybrid system. The new version of *SkiROS* also fixes issues related to the execution monitoring of

[9]https://www.w3.org/TR/owl-features/
[10]https://github.com/Bjarne-AAU/skiros-demo

action servers, which prevented the original version from working correctly in conjunction with *MoveIt!*. Additionally, the skills and primitives no longer need to be represented in the ontology, only the robot knowledge and the system knowledge.

2.6 RQt

The *GUI* implemented in *SkiROS* is built in *RQt*[11], which is a *Qt*-based framework for *GUI* development in *ROS*. This software implements various *GUI* tools for *ROS* in form of plugins and is a useful cross-platform alternative to creating a graphical interface by hand.

2.7 UR5

Universal Robots launched the *UR5*[12] collaborative robot arm for industrial environments. This is a lightweight, adaptable collaborative robot that tackles medium-duty applications. It can reach 850 mm and carry 5 kg while having a footprint of 149 mm diameter and weighs 20.6 kg. This robot is especially useful in human-robot collaboration environments, since it automatically detects any type of unwanted collisions and stops execution immediately. An image of such a robotic arm can be seen at Figure 2.6.

An implementation of many Universal Robots device configurations for the control of the robotic arms via *MoveIt!* can be found at on the *ROS-Industrial* GitHub page[13], including the configuration for the UR5 robotic arm.

[11] `https://wiki.ros.org/rqt`
[12] `https://www.universal-robots.com/products/ur5-robot`
[13] `https://github.com/ros-industrial/universal_robot`

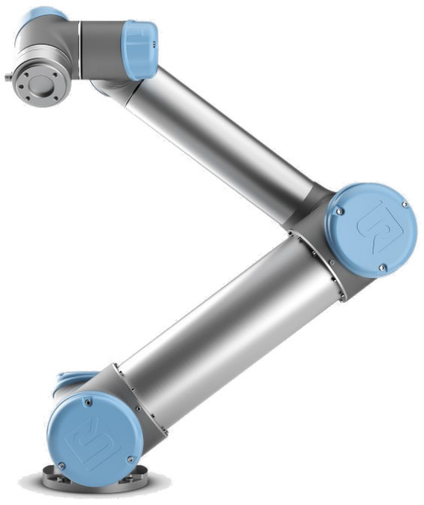


Figure 2.6: UR5 robotic arm by Universal Robots. Source: https://www.universal-robots.com/3d/ur5.html

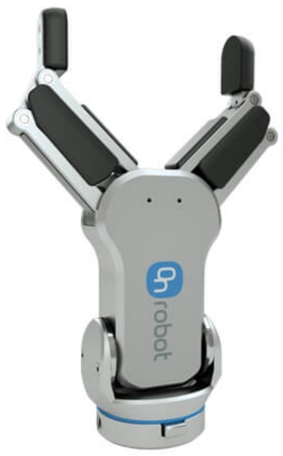


Figure 2.7: RG6 robotic gripper by OnRobot. Source: https://onrobot.com/en/products/rg6-gripper

2.8 RG6

There are many different types of grippers on the market. Given the initial intention of a future expansion of this system on real robots, a commonly used gripper was selected. It was released by OnRobot and is called *RG6*[14]. An image of such an gripper can be seen at Figure 2.7. It is a 6 kg payload robot arm gripper with up to 150 mm stroke. The fingertips can be customized and replaced, maximizing the robot utilization. A kinematic version of this gripper does not exist online, only a 3D drawing can be found on the OnRobot web page[15]. Therefore a kinematic version had to be created during this project. More information in regard to the process of creating the kinematic model can be found in Section 6.2.4.

[14]https://onrobot.com/en/products/rg6-gripper
[15]https://onrobot.com/en/downloads

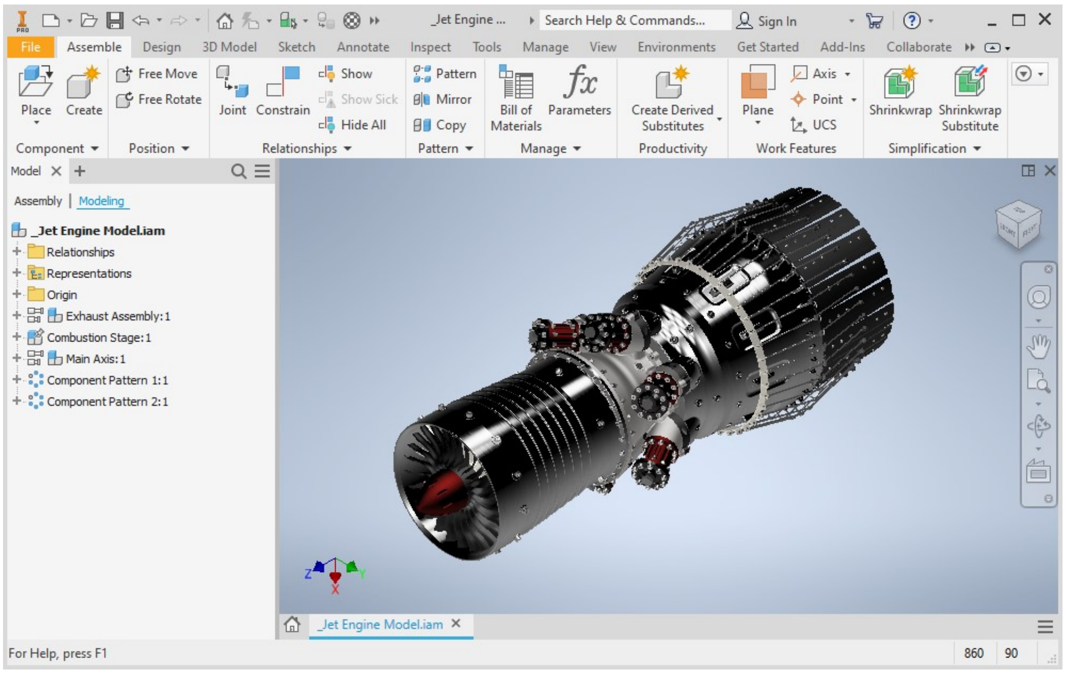


Figure 2.8: Screenshot of AutoDesk Inventor with an example assembly. Source: Inventor sample files[16]

2.9 AutoDesk Inventor

Inventor is a mechanical design and 3D *CAD* model software for Windows. This software was selected because of its free availability for educational purposes and because it provides an *API* to access almost any internal information about the assembly model and its components. Figure 2.8 shows a screenshot of AutoDesk Inventor Professional 2020.

Inventor allows access to most information over internal *API* calls to Component Object Model (*COM*) objects of the application. The documentation of the *API* can be found locally only after installing the application. Online, a help portal with guides, tutorials and additional information on Inventor in

[16] https://knowledge.autodesk.com/support/inventor/downloads/caas/downloads/content/inventor-sample-files.html

general and regarding the *API* can be found on the AutoDesk Help page[17]. *COM* is a platform-independent, object-oriented interface standard for creating binary software components that can interact. These components are programming language independent and can be accessed system wide.

2.9.1 Inventor API

The *API* allows direct access to the internal document classes. There are four document class types, where the first two are relevant to this project: *AssemblyDocument*, *PartDocument*, *DrawingDocument* and *PresentationDocument*, which are explained in detail in the locally installed documentation or summarized in form of a AutoDesk Inventor API Object Model[18].

The main access point to the assembly is possible via instances of the *AssemblyDocument* class. From there, all information regarding the file can be accessed. This includes the document name, a unique internal identification number, *ComponentDefinitions*, references to included documents such as subassemblies or the individual part files and more.
A *AssemblyComponentDefinition* contains all occurrences of joints, constraints and components of the assembly, which will be referenced in upcoming chapters of the thesis. Inventor specific information regarding each part, like name, internal id, type and more, can be accessed using the referenced *PartDocument* class, which again contains a *PartComponentDefinition* housing boundary representation and geometric feature constraints, inertial parameters, mass properties or the bounding box of each individual part.

2.9.2 Inventor iLogic

An alternative to creating a full plugin would be the use of a tool called Inventor iLogic. This is a concept of allowing access to most *API* functionality directly inside Inventor via Virtual Basic (*VB.NET*) programming.

[17] https://help.autodesk.com/view/INVNTOR/2020/ENU/

[18] https://knowledge.autodesk.com/search-result/caas/simplecontent/content/autodesk-C2-AE-inventor-C2-AE-api-object-model-reference-document-pdfs.html

There are major downsides in using the iLogic for this project. Debugging and IntelliSense do not work for iLogic. In comparison, Microsoft Visual Studio allows both to some extend. Some specific *API* functionality like exporting components into stereolithography files (*STL*), which are needed for spawning objects in ROS, is not accessible this way. Lastly the plan needs to be generated and exported, which might be possible using iLogic and *VB.NET*, but the API and C# are preferred for easier handling and the possibility of expanding the plugin with useful functionality like graph visualization tools.

2.9.3 Parts, Joints and Constrains in Inventor

In an Inventor assembly, parts or other assemblies are connected using joints or constraints, which create relationships that determine component placement and allowable movement. They are used to position components in order to gradually eliminate degrees of freedom (*DOF*s). Inventor recommends the use of joints to position a component and fully define its motion, because it reduces the complexity of component relationships. It is generally possible to produce the same outcome with either joints or constraints, but fewer connections are required using joints. It is possible to add limits for both types of relationships, defining the allowable range of motion for components that move or rotate.

Each component in an assembly has six *DOF*s, three translational and three rotational *DOF*s. These include being able to move in X, Y and Z and to rotate around the X, Y and Z axes. When placing a constraint between two pieces of geometry, a position is established and one or more *DOF*s are removed. When placing a joint between two piece of geometry, the *DOF*s and the location are explicitly defined, thus requiring fewer relationships than constrains and reducing the complexity of the assembly.
Since each joint type fully defines the location and motion of components, it can be used to uniquely identify the relation between parts. For this reason joints are preferred over constraints in this project.

The first component selected in Inventor, when placing a relation, is moved towards the second component. In this relationship, the first part is the child

and the second part is the parent. The notation for such a relation during this work is (a, b) where a is the parent and b the child.
Additionally, in this thesis two different methods of stacking components on top of each other will be used. Assuming an example setup with bolt a, spacer b and nut c, the fist possibility of connecting all parts is placing the relations face-to-face. The resulting relationships would be (a, b) and (b, c). Alternatively, it is possible to define relations, that have the same reference point, such as (a, b) and (a, c). Connecting components face-to-face is the more common approach for *CAD* modelling. In the upcoming chapters situation will be described, where the presented methodology requires either one or the other type of relation placement between multiple components.

Inventor poses two different approaches on eliminating all degrees of freedom of a component, by *locking* or *grounding* it. Locking a component allows it to change location when related parts move. Grounding, on the other hand, fixes the component position in space, making it unmovable.

Six different types of joints are supported by Inventor:

1. rigid - where all *DOF*s are removed;
2. rotational - where one rotational *DOF* can be specified;
3. slider - where one translational *DOF* can be specified;
4. cylindrical - where one rotational and one translational *DOF* can be specified;
5. planar - where two translational *DOF* can be specified and one rotational *DOF* results open;
6. ball - where three rotational *DOF* can be specified.

This project does not need to allow ball joints or free translational *DOF*s, since all assembly actions are based on pick and place actions. Therefore, it is recommended to only use rigid and rotational constraints. It is possible to use joints with specifiable translational *DOF*s for a more kinematic *CAD* model, but then it is important to make sure the default position along said *DOF* is where the components should be at the end of the assembly.

When a relation is placed between components, Inventor tries to place the the two parts in question next to each other by aligning them. To do so, it finds two similar surfaces, preferably circles, one for each component. These

surfaces are then used to define the orientation of the child component in relation to the parent. The normals of the geometries, used to align the parts, are either parallel or inverse to each other. The distance of a joint is therefore clearly defined as the distance between the two starting points of the aligned normals.
To align two components connected by a constraint, Inventor uses the exact selected geometries and aligns them using the positioning method dictated by the constraint type. The distance of a constraint is not as clearly defined, since Inventor displays the minimum distance, which can be selected out of the minimum distances calculated between vertices, edges or faces.

In regard to constraints, Inventor allows six different types:

1. mate - which positions selected faces normal to each other, with coinciding center points;
2. flush - which positions selected faces next to each other with surface normals pointing in the same direction;
3. insert - which places either a mate or a flush constraint between the components planar faces and a mate between the axes with one rotational *DOF* remaining open;
4. angle - which positions edges or planar faces at a specified angle;
5. tangent - which causes faces, planes, cylinders, spheres and cones to contact at the point of tangency;
6. symmetry - which positions two objects symmetrically according to a plane or planar face.

If it is needed to use constraints to place components, only mate, flush and both types of insert are allowed, since they align using the selected geometries, similar to joints, and have a defined interpretation. The presented system is not intended to have different aligning surfaces for relations between the same two components, since the interpretation of multiple constraints is not implemented.

Additional information can be attached to a relation via suffix to its name, such as whether a relation is pre-satisfied - meaning the referenced components come pre-assembled, it is supposed to be handled as a screwing motion on insertion or is part of a group referencing the same multi-contact component.
For pre-satisfied relations the suffix *_PRESAT* must be added to the their

name. All rigid joints are interpreted to be screws. If that is not supposed to be the case for a specific rigid, this can be noted by adding the suffix *_NS* to the relation name. If relations are referencing the same multi-contact component, they need to be identified as related by adding the same suffix to their names, for example *_G1*. This suffix can be chosen arbitrarily, as long as it does not contain one of the other suffixes in the naming scheme. It is possible to have all three suffix types for a single relation. Additionally, the presented plugin, used to generate and export the plan, allows the possibility of selecting and deselecting screws from a list of all rigid relations. These design decisions are described in more detail in Section 6.1.2.

3 Related Research

3.1 Human Robot Collaboration

The interest in robots cooperating with humans for domestic and industrial use has increased in recent years. The combination of the cognitive capacity of humans with the dexterity and physical strength of robots can lead to many applications in various fields. But enabling robotic systems to collaborate with humans is challenging on different levels of abstraction. Human robot collaboration (*HRC*) is a complex field that combines robotics, artificial intelligence, cognitive science, computer science, engineering, human computer interaction, psychology and social science [7]. In addition, there are four different degrees of human robot interaction, referenced in increasing complexity: (1) coexistence – where both agents (human and robot) work on different tasks in the same workspace; (2) assistance – where the robots passively aids the human in a task; (3) cooperation – where both agents work simultaneously on different tasks on the same work piece; and (4) collaboration – where both agents perform coordinated actions on the same task. [8]

Using perception, planning and reasoning, such systems need to understand the context under which they operate. Akkaladevi et al [8] proposed a mental model for perspective taking capabilities, that considers human preferences, the knowledge of the task and the capabilities of both human and robot. This model forms the basis of their cognitive architecture to perceive, reason and plan in the human-robot collaborative scenario. Possible actions that can be performed, based on decisions of this architecture, are 'picking', 'showing', 'placing' and 'handover' actions on real world objects. To be able to seamlessly interact with humans, the robot should know about object properties, but also about capabilities, beliefs, goals and desires of humans. It is also important to correctly recognize human actions and track components over time.

Sadrfaridpour et al. [9] consider a computational model of a human worker's trust in the robot partner. The intention was to design a system that is able to keep pace with the human worker while maintaining a level of security and embeddedness in design such that robot actions become acceptable and comfortable for humans. They use the trust evaluation as a constraint in an optimal control problem, where the robot speed needs to be controlled in a way that enables the robot path progress to follow that of the human.
Mizanoor Rahman et al. [10] presented a computational model for trust including a method to measure and display the trust in real-time inside a handover motion planning strategy. They conclude that the consideration of robot trust in humans and an adaptation of the tasks based on the level of trust significantly improve human-robot interaction and assembly performance through increasing safety, human trust in robot, handover success rate and the overall assembly efficiency.

Ajoudani et al. [11] found that, despite the inclusion of several technologies, e.g. force feedback, augmented reality, etc., the amount of information the robot should communicate to the human is still an open research topic. Concepts like impedance, force, admittance or hybrid approaches still appear to be in a premature state. When developed they will most likely not only enhance the robot adaptivity to the human and the environment but may also lead to reduction of task-related pre-programming.

3.2 Visual Perception

Around fifteen years ago, most of the open questions regarding visual object recognition where: viewpoint variation, illumination, occlusion, scale, deformation, background clutter and intra-class variation [12]. Bigger and cleaner datasets allowed a more detailed data extraction in regard to specific problems or features, which, in combination with deep learning and convolutional neural networks (*CNN*), proved the data driven models to be more effective in comparison to previous explicit model-driven concepts [13]. With this new era of deep *CNN*s most of the previously mentioned problems can be seen as at least partially solved [13], leaving occlusion and deformation as the two major open questions.

Current research is also trying to exploit multiple viewing angles [14] or fusing cameras with depth or Light Detection and Ranging (*LIDAR*) sensors for better object recognition, resulting in new image datasets [15]. For example, in [16] Skotheim et al. presented a system containing a complementary robot solely for vision purposes that acquires 3D point clouds by performing sweeps with a laser triangulation sensor. Because the sensor is mounted on a robot, an optimal viewpoint can be chosen.

Occlusions can be identified to a certain degree if they are part of the image dataset and appear frequent enough, but it is increasingly unlikely to be able to cover all occlusion configurations in all datasets. Attempts are made to detect multiple objects from a single image and then reason about occlusions between objects in a projected 3D space [17].
Similarly to deformable objects, if humans or animals would not have rigid shapes, *CNN*s would have difficulties identifying them. In [18] a method to identify flexural rigidity and the initial curvature of a deformable belt object from its static images was proposed.

More related to object recognition than object detection, reasoning about the 3D structure of objects [19] is the topic of current research, since the level of geometric detail is limited to qualitative representations or rough boxes because object recognizers are tuned towards robust 2D bounding box matching rather than accurate 3D geometry [20, 21]. Similarly, how objects are laid out in a scene and therefore understanding the scene as a whole with all its components and their poses [17, 22, 23] is also state-of-the-art computer vision research.

A geometric surface primitive patch segmentation approach based on Hough transforms to get accurate surface normal estimations from 3D point clouds is presented in [24], which is focusing on the application of bin picking in automotive subassembly automation while taking advantage of preexisting *CAD* data. These primitive surface patches are extracted from automotive *CAD* models, which are simplified to entities such as planar polygons, vertices and lines. The final matching of such primitives seems to be highly accurate regarding pose and object class estimation.

3.3 Knowledge Base

The enrichment of *CAD* models with semantic information has been approached in different research projects. OntoCAD [25] is an ontological annotation approach that is based on labeling geometry elements of *CAD* models using concepts from a *CAD* ontology. In [26] an ontology for describing *CAD* models in the Drawing Exchange Format (*DXF*) is presented. Tessier et al. [27] created an ontology-based approach of semantically describing *CAD* data and features using a rule system to automatically classify *CAD* features. This is in order to provide compatible mapping between different *CAD* systems.

OntoBrep [28] introduces an approach for leveraging *CAD* descriptions to a semantic level, in order to link additional knowledge to *CAD* models. To do so, a description language, based on *OWL*, is designed to define boundary representations of objects. This involves describing geometrical components in a semantically meaningful way, usually corresponding with the minimal mathematical description of a shape. As an example, a circle would be represented by a center point and a radius instead of a set of polygons. This covers not only shapes and objects, but also geometric constraints between components. Constraints can be specified down to single edges or faces in *CAD* files. This is especially useful for shape-based object recognition or constraint-based task descriptions.

Opposed to geometrical representations of *CAD* models, within the European *FP7*, the Seventh Framework Programme of the European Union research and development funding program, project ARUM Adaptive Production Management[1] presented a task description ontology design, which tries to increase effectiveness of production of highly complex and individualized products. It is aimed at the optimization of production ramp-up and the general workflow description of full processes.

[1] `https://cordis.europa.eu/project/id/314056/de`

3.4 Dynamic Movement Primitives

Imitation learning has been a big part of humanoid robotics for a long time [29] and are widely used [30]. In search of a method for representing modular actions for later reuse and abstraction to similar situations, motion primitives were developed.

Calinon et al. [31] presented a programming by demonstration framework for generically extracting relevant features of a task and generalizing learned concepts to different contexts. They conclude, that their imitation metric is an optimization of object-hands relationships, hands paths and joint angle trajectories, where the importance of the variables is dependent on the task. The different degrees of importance are extracted by observation of the task executed by a human expert. In a goal-directed framework, these three variables have also different levels of relevance [31]. While manipulating an object, the first variable gains relevance. Depending on the scene, whether it is full or empty, the second variable is more prominent. And if the reproduction of a gesture is relevant, for example for motions like waving a hand or knocking on a door.
Bekkering et al. showed in [32] that imitation is goal-directed. They also suggest different levels of importance for object-hands relationships and hands paths depending on the age of the subject. Infants focus on a single level, while adults use multiple levels and simultaneously assign preferences to different levels hierarchically. This was shown in a previous work of Calinon et al. presented in [33] regarding how the prioritization of these variables can speed up the extraction of task constraint.
The system presented by Calinon et al. can generalize over variations of joint angles, hands paths, hands-object relationships and the opening and closing of hands, by projecting the data onto a latent space and encoding the results into Gaussian and Bernoulli Mixture Models. The variation and correlation information can be used to find a solution to the inverse kinematics, so the system is able to generalize over multiple contexts.

Ijspeert et al. developed a framework of Dynamic Movement Primitives (*DMP*) for robust motion generation [34, 35] which can represent a demonstrated movement in form of a set of differential equations [36, 37]. This system of equations allows the addition of a disturbing force without loosing stability of the movement, since perturbations can be automatically corrected by the dynamics of the system. In [38] this framework was extended by Park et al. such that arbitrary movements in the end-effector space can be represented. They also added online obstacle avoidance by including previously established potential field algorithms [39, 40], where obstacles are represented as repellent forces centered around the obstacle. This way the robot was able to avoid obstacles while still being attracted to the original demonstrated movement. Also, the use of a dynamic potential field, dependent on the relative velocity between obstacle and end-effector, proved to produce more smooth movements, especially when the end-effector directly approached the obstacle.

Movement primitives as a modular and reusable system of actions are well-established [30] and there are many different variations that are being created. Paraschos et al. presented in [41] a probabilistic approach, allowing for the derivation of new operations essential for blending between motions, adapting to altered task variables and activating multiple motion primitives at the same time.
In [42] Deniša et al. address the problem of simultaneously achieving low trajectory tracking errors while maintaining compliant control without the necessity of explicit mathematical models. They propose a new movement representation, which encodes position trajectory and the corresponding torque and can learn from a single demonstration. Their presented control framework allows the robot to remain compliant and therefore safe for *HRI* even if high trajectory tracking accuracy is needed.
Zhang et al. [43] developed a movement primitive for force interaction skills that can be programmed with kinesthetic teaching and combines machine learning for adaptability and reusability by estimating the probability of success.

3.5 Planning and Reasoning

Classical planning revolves around finding sequences of actions that turn the initial state into a state, in which it satisfies the given goal. Commonly such problems are defined by an initial state, a set of possible actions and a goal.

Today, such problems are usually described in *PDDL* and specific planners are used to solve these problems. One of the first automatic planners that up until today is describing the basis for most problem domains is called Stanford Research Institute Problem Solver (*STRIPS*) and was presented by Fikes et al in [44]. It is designed to work with large numbers of rules and represents the world model as an arbitrary collection of first-order predicate calculus formulas. Later on, the term *STRIPS* was changed to reference the description language needed to input information for the planner.
Action Description Language (*ADL*) [45] is considered an advancement of *STRIPS* by allowing the effects of an operator to be conditional. Contrary to *STRIPS*, which assumed unknown events to be false, known as the closed-world assumption, everything not occurring in the conditions is assumed to be unknown, which is called the open-world assumption. *ADL* also allows the use of negative literals and disjunctions, whereas *STRIPS* only allowed for positive literals and conjunctions.
A partial order planner named *UCPOP* is described in [46] and handles a subset of ADL actions. It operates with actions that have conditional effects and universally quantified preconditions, effects and goals. Based on *STRIPS*, *ADL* and especially *UCPOP*, among others, a new closed-world assumption based concept called *PDDL* was introduced.

PDDL is a predictive language for the description of planning problems and was presented in [5]. It unifies basic *STRIPS*-style actions with conditional effects, domain axioms and universal quantification. Including the specification of safety constraints and actions composed of sub-actions and sub-goals, allows for a management of many problems in multiple domains using different subsets of language features. This enables the sharing of domains across different planners that handle different levels of expressiveness. On the other hand, most planners are not expected to contain the full *PDDL* language. This means a planner can skip over all definitions connected with subsets of features that are not handled.

In [47] a framework to construct complex robotics application called *Task Compiler* was presented, that automatically generates state-machine based behavior executives from a high-level symbolic description of a robotic task. To achieve this, they pursued to develop a compiler that translates task descriptions in *PDDL* to *SMACH*, a state machine based execution and coordination system. Behavior state machines provide a robust behavior control with failure recovery features for interactions in a human environment.

Various methods for intuitive task level programming have been proposed over the years and have become a fundamental point of interest for industrial applications. Many of these implementations are *PDDL*-based systems.

Cashmore et al. presented the *ROSPlan* framework in [48], which allows to link existing ROS components to a planning tool. It provides a collection of tools for AI Planning in a *ROS* system, such as planning, problem generation and plan execution. Their architecture is intended as a standard method to embedding a *PDDL*-based task planner into a robotic system, while focusing on providing a standardized access to modern heuristic search planners through *ROS*. Through knowledge discovery initial and goal state descriptions are autonomously reconstructed after each replanning cycle, since the planner is in constant communication with the knowledge base. Additionally, the integration in *ROS* allows the application designer to focus on the definition of the *PDDL* domain description.

During the European *FP7* project TAPAS[2], short for Tools for Assessment and Planning of Aquaculture Sustainability, the *skill paradigm* was introduced. It facilitates task-level programming of mobile manipulators by providing the robot with a set of movement primitives, skills and tasks. This allows a four tiered architecture separation into the layers of hardware abstraction, multi-sensory control, object-level abstraction and planning, related respectively to proxy, primitive, skill and task descriptions. This allows the implementation of a flexible, highly modular system for the development of cognitive robot tasks and is used in other *FP7* projects like Sustainable and Reliable Robotics for Part Handling in Manufacturing Automation, short *STAMINA*, or CooperAtive Robot for Large Spaces manufacturing, short *CARLOS*.
Based on investigation on multi-tiered robotic architectures, a major research effort was made to form a generalized robot middleware, *ROS*. Two examples of such multi-leveled architectures are *MoveIt!* [4], which was explained in Section 2.3, and ROS commander (*ROSco*) [49]. The latter is focused on home environment robots where expert users build a behavior in form of

[2]https://cordis.europa.eu/project/id/678396

a Hierarchical Finite State Machine (*HFSM*) out of generic, parameterized building blocks.

Rovida et al. presented in [50] a Skill-based Robot Operating System (*SkiROS*), which aims to extend the capabilities of *MoveIt!* with high level instructions similar to *ROSco*. SkiROS is explained in more detail in Section 2.5. The initial version was based entirely on *PDDL* planning, a non-reactive, deliberative system, whereas the second version includes the concept of behavior trees. In conjunction with the extended behavior tree model presented in [6] a reactive behavior tree can be generated. At run-time, the robot can use the more abstract skills to plan a sequence using a PDDL planner, expand the sequence into a hierarchical tree and re-organize it to optimize the time of execution and the use of resources. This leads to a reactive-deliberative system hybrid.

3.6 Dual-Arm Kinematics

Due to a more widespread availability of commercial single-arm robots, the popularity of industrial dual-arm robotic systems increased. They are considered as promising tools for industrial automation, because of their range of characteristics discussed in a survey by Weng et al. in [51]. They conclude, that the anthropomorphic manipulators provide operators with a more intuitive way to manipulate. The nature of their kinematics make stiffness adjustable when both arms are contacting a common object. High redundancy improves the flexibility and versatility in task manipulation and synchronized manipulation shortens the cycle time of task execution. All these characteristics make dual-arm robotics the a promising concept regarding the next generation of industrial robots.
Depending on not only the features of the robot kinematics but also the demands of the assigned task, dual-arm manipulation definitions can vary. Multiple single-arm robots organized together can handle tasks that exceed the difficulty and complexity of general single-arm robot capabilities.

The work presented in [52] classifies dual-arm manipulation into uncoordinated and coordinated manipulation. Uncoordinated manipulation indicates that individual motion is assigned to each arm. Coordinated manipulation can again be subclassified into goal-coordinated manipulation and bi-manual manipulation. Goal-coordinated manipulation has both arms assigned to the same task, where each arm has an independent motion assigned to it. Bi-manual manipulation refers to the situation, in which both arms are acting on the same object simultaneously in either symmetric or asymmetric or else congruent or non-congruent form. Goal-coordinated manipulation is seen as the most difficult and complex category which may require long-term practice for a human, while bi-manual manipulation is intuitive for humans. In combination with clear and precise taxonomies and representations of robotic assembly skills, an assembly task can be divided into a series of actions. Perception and compliance, which have been successfully implemented on assembly task for single-arm robots, can be used to realize dual-arm manipulation on assembly tasks.

Suomalainen et al. show how different choices regarding compliance affect a dual-arm assembly task. Compliant motions are helpful to mitigate errors in pose estimation and can be learned from human demonstration. They present analytical background and experimental results on how to enhance the convergence region of alignment tasks. In [53] Tarbouriec et al. propose a sparse kinematic control strategy, that minimizes the number of joints actuated for a coordinated task between two arms, while avoiding joint limits. An alternative approach to developing industrial dual-arm robots for flexible assembly is presented in [54] by Chen et al. They started by introducing a kinematic feature and reachability map of a seven *DOF* industrial robots. By adjusting the distance and angle of two identical robot arm, the bi-manual workspace of the dual-arm setup is analyzed. The concluding cooperative reachability map is meant to visually evaluate all possible models of the dual-arm robots by uniting and intersecting the reachability index of the two separate arms and was used as a criterion to evaluate the design of the dual-arm robot for an assembly task. In doing so, they designed the *PMC 14-axis dual arm robot*.

3.7 Flexible Assembly

In [55] Thomas et al. presented an assembly sequence planner able to generate feasible sequences for building a desired assembly. While taking geometrical, physical and mechanical constraints into account, the planner considers the feasibility of grasps during the planning process and considers work-cell specific constraints. For the planning AND/OR-graphs are used, which are generated by using a specialized graph cut algorithm. AND/OR-graphs represent the reduction of problems to conjunctions and disjunctions of subproblems.
AND/OR-graphs or Liaison-graphs are often used as inputs for assembly sequencing systems. Liaison-graphs describe the precedence relationships between parts but are very time consuming if inferred with the aid of humans. In [56, 57] Thomas et al. present their own method of automatically obtaining such a graph from geometric computations. It is applied to efficiently analyze the geometric feasibility of an assembly path. Pairwise, parts are mapped for their possible assembly directions into $2\frac{1}{2}$D distance maps. The generated graphs are further evaluated by considering the feasibility of grasping subassemblies and individual parts. Since the sequence and grasp planner are generic, their proposed solution can be applied to arbitrary assemblies of rigid parts. Tolerances for parts' shapes are allowed, as well as pose uncertainties [56]. Since the grasp planner is integrated during evaluation of the graph, verification of existence of feasible and robust grasp poses for subassemblies is possible.

The FlexRoP project [58] tries to overcome current limitations by developing a flexible skill-based robot programming middleware and improved *GUI* technologies for robot assistance. The system is equipped with a seven *DOF* robotic arm and was intended to have universal grippers and force closure for handling and manipulation of objects. Tests disproved the applicability of several universal grippers for accuracy and process stability reasons. The diversity of requirements resulted in a complex tool design with force torque sensors, *RGBD* and 2D cameras, two electric grippers and an automatic tool changer for spanning additional, ordinary hand tools articulated by pneumatic actuators.
The robot assistant is required to be programmable without special training. For this reason, XRob [59] is introduced as an layer of abstraction for all

hardware, such as cameras, sensors or robots, and software components, like object pose recognition or path planning. Additionally, a real time interface for kinesthetic skill learning and the force torque sensor is required.
A motion assessment primitive is responsible for providing an evaluation of kinesthetically taught DMP based skills. While recording both the joints' states and the exerted forces and torques on the end-effector, the trajectory recorded through kinesthetic teaching is exploited. Machine learning techniques make it possible to map joint states to exerted forces and torques and can therefore predict which forces and torques to expect at specific joint states.
The resulting conclusions by Ikeda et al. [58] contain the assessment, that programming in the worker's domain without kinesthetic manipulation of the robot itself remains desirable. Compared to kinesthetic teaching, the so-called embodiment problem needs to be solved, since the robot has different reach and multiple kinematic configurations that can be used to position a tool.
The second major issue is the low success rate, which could be caused by *DMP*s, since they create a single model for each *DOF*, which means the correlation between joints' states and the exerted forces and torques may be lost. This can be dealt with using all the sensory inputs to create a single model. Moreover, motion assessment would need to be done during execution to minimize security issues and component or robot damage.

4 Problem Formulation

An Industrial Assembly Task can have different degrees of automation, depending on a variety of factors, including production lifetime, volume and flexibility. This thesis will focus on flexible automation, the higher degree of automation. The idea is that the automated assembly system becomes an autonomous intelligent system able to derive an assembly plan from a description of the product automatically and to execute this assembly plan autonomously without human intervention such as teaching. Flexible automation, in comparison to fully pre-programmable automation, has the advantage of being able to detect a wide range of changes in products, up to even completely novel products, on its own and can subsequently react with no or a minimum of delay time, adaptations or manual intervention.

Generally, the input should be some kind of representation which fully describes the assembly and should lead to a plan of actions that can be successfully executed on a failsafe dual-arm robotic setup.

In order to enable the system to perform this autonomous assembly it has to master three central challenges: geometrical and semantic understanding of the given representation, derivation of a plan and robust execution of a set of skills leading to a successful assembly.

4.1 Geometrical and Semantic Understanding

First, the system needs to be able to geometrically and semantically understand a representation of the assembly task and the involved components and tools provided by the engineers (e.g. *CAD* models). This involves knowledge and reasoning about the shape, the properties, the function, and the relation of items (e.g. component models, ontologies, dynamical description of process steps).

4.2 Plan derivation

Second, the system needs to be able to derive an assembly plan from the descriptions and knowledge received. This plan represents a sequence of predefined and modular actions available in form of combinations of skills. Here the system needs to plan on an abstract as well as task and motion planning on a continuous level. The reasoning about the actions needed to assemble a given product requires to be combined with the planning of the concrete motions of the tools and parts. The assembly task itself should be divided into subtasks automatically, where every subtask can either be a basic skill or again subdivided into moderately complex subtasks. At the end, every action should be based on combinations of already pre-acquired skills. In state-of-the-art flexible automation robotics these skills are usually acquired through kinesthetic teaching or machine learning. An important aspect here is, that in order to reuse and combine these skills flexibly instead of using fixed teached-in posed the skills need to refer to the component models and their attached coordinate frames and anchor (e.g. boreholes). The idea is to minimize manual interference by using inferred knowledge of the single components used in the assembly task with as little as possible prior information about the final product, intermediate product or assembly sequences.

4.3 Robust execution

Third, the system needs to be able to execute the assembly plan in a real environment autonomously and robustly. This requires that the system is able to identify and precisely localize the involved components and tools using some kind of perception system. Moreover, the system needs to ground (e.g. link abstract object positions in the planning level to concrete object positions in the motion level) and execute the planned skills based

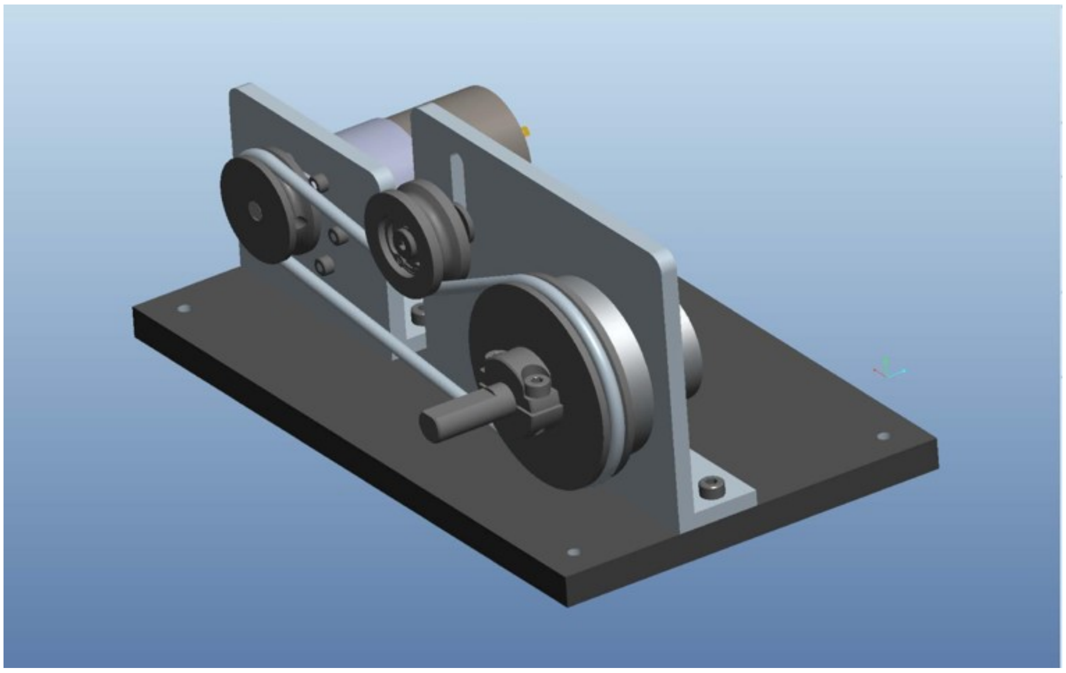


Figure 4.1: Belt Drive Unit from World Robot Summit - Assembly Challenge. Source: [60]

on the actual physical configuration in the environment. In order to gain robustness, the system needs monitoring capabilities in order to track the progress of skills and assembly steps and be able to re-plan assembly steps on the fly if a previously obtained plan does not work anymore in the environment.

4.4 Assembly Task

As a running example and a test setup the Assembly Challenge from the Industrial Robotics Category, presented at the World Robot Summit 2018, is used, in which a Belt Drive Unit needs to be assembled. Figure 4.1 shows the front view of the assembly, while Figure 4.2 shows the same assembly in an exploded view. A summary of the competition in 2018 can be found at [61].

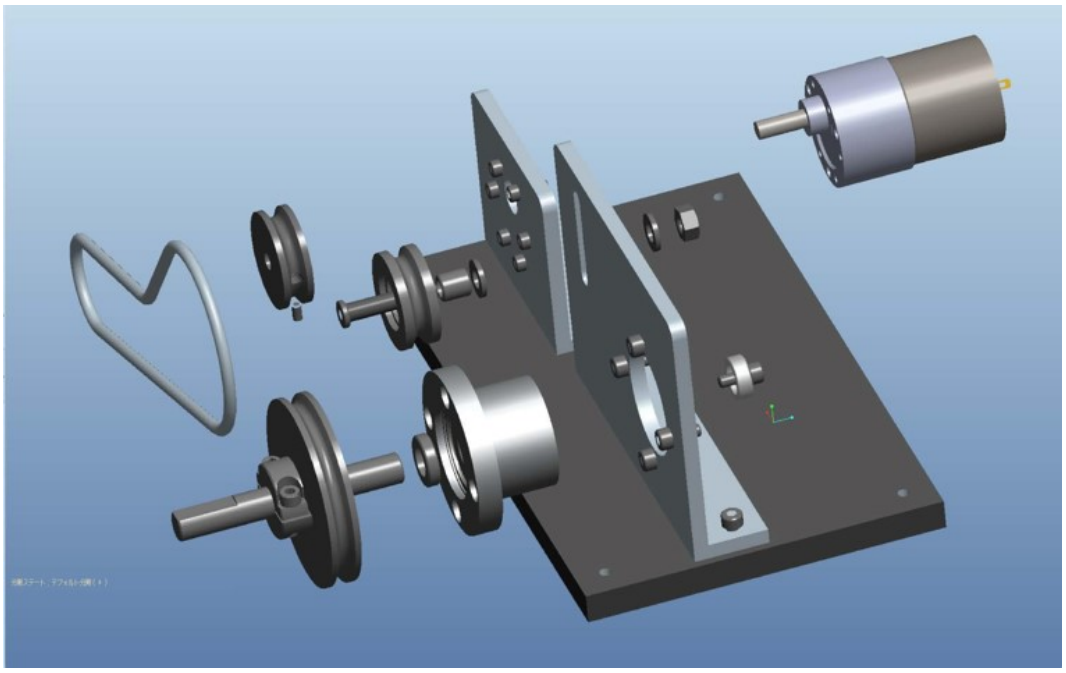


Figure 4.2: Exploded view of Belt Drive Unit from World Robot Summit - Assembly Challenge. Source: [60]

All components of the assembly are listed in [60] and the 3D models needed for the simulation were found online at the respective reseller, except the 3 mounting plates. These needed to be manually created.

Most relations between parts in this assembly can be executed using bin-picking and peg-in-the-hole actions, which makes the knowledge representation for most objects limited to abstract relations like *has peg* and *has hole*. Most of the concrete data-properties are geometrical information like translation, orientation, offset, diameter or length. These descriptions are called static. They need to be represented in form of knowledge and must be available prior to planning or execution.

Dynamic descriptions used inside such a task would be *is attached* or *is inserted*, where sensing in conjunction with a constantly updated world model needs to be used to get knowledge about the current environment.

This version of a Belt Drive has many different parts. Some need to be combined before some specific other actions can be executed. Others need to be assembled inside of a slot at a specific position, then other components need to be placed, only for the previously placed components needing to be moved within their slot afterwards.

5 Concept

The general concept behind the work presented in this thesis is shown in Figure 5.1. The input to the system is a high level task representation in the from of a *CAD* model, sensor data and the state of the robot. The sensory information and the robot state are only relevant to the task execution. The three-dimensional depiction of the assembly in form of a *CAD* file is used to generate an alternative representation of the task. Via a set of rules, described in more detail in Section 5.1.1, it is possible to generate a *CAD* file, from which a given set of algorithms can extract a sequence of actions. These algorithms are discussed in more detail in Section 5.1.4.

The extracted information is expressed using two separate ontologies. The main class structure of the system is designed as part of this project in form of a *base ontology*. A detailed representation of the full class namespace can be found in the shape of a class diagram in Figure 5.11 and is further discussed in Section 5.1.3. Summarized, a *plan* is a list of ordered *actions*. Actions reference to a specific *connection* between two parent and child components, that is described by its *relations*. A relation can either be a joint or a constraint between two parts and contains the transformation between the two objects.

The extracted plan contains instantiations of all *named individuals*, including their *data* and *object properties*, relevant to the task. This information is modeled based on the previously mentioned, defined class structure and is exported purely from the *CAD* file with no sensing needed. This ontology file is automatically generated using a plugin created within the scope of this thesis.

The next step of the workflow is to interpret the plan, which generally depends on the environment configuration. Depending on the number of available robot arms, the types of grippers and the available skills of the

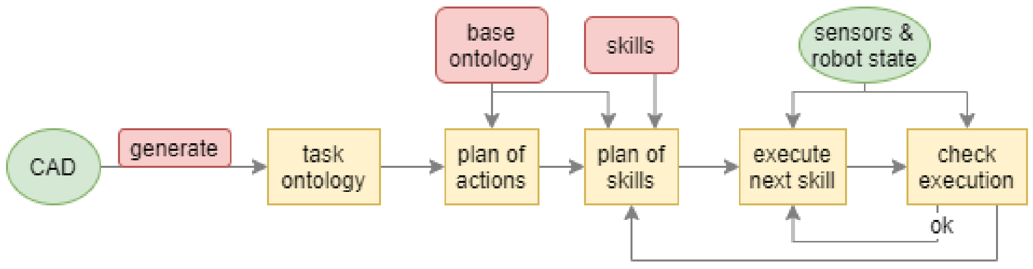


Figure 5.1: Main concept of the system. Green marks the input, red marks components presented in during this work and yellow marks outputs or intermediate steps generated by the algorithms.

system, specific tasks can be executed differently. A possible plan interpretation algorithm for the chosen dual-arm robot setup is presented in Section 5.1.5.

Lastly it is important to ensure a safe and robust execution of each individual action. Section 5.1.6 discusses a possible hierarchical setup of modular skills and primitives needed to solve general assemblies. This includes generic *bin-picking* operations, screwing components into place and moving parts after loosening others. The *BDU* is used as a running example for the construction of an assembly with moving parts and a deformable belt.

Since the execution of the assembly is handled inside of a simulation, there are two separate aspects to the data extraction from the *CAD* model. So far only the main purpose of export tool was described, namely the plan extraction. A more detailed description can be found in the following Section 5.1. The second functionality, which was implemented in the Inventor plugin, is the ability to extract an initial scene for the simulation based on an additional *CAD* model. More information about the initial scene generation can be found at Section 5.2.

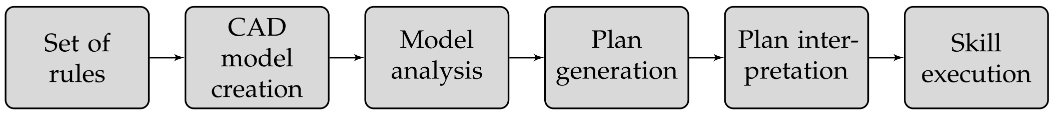


Figure 5.2: General workflow for preparation and plan extraction.

5.1 Assembly

The general workflow for the system, no matter whether within a simulated or real environment, can be subdivided into the six segments, which are represented in Figure 5.2 and described in detail in this section. No matter what environment, throughout this work two main coordinate systems are referenced: (1) the world coordinate system with a user defined origin and all assembly components, the robot and the rest of the scene refer to it; (2) the component coordinate system, which has its origin at the component's center of mass. The relative transformations of each part, such as the location of each grasp pose or the transformation between related components, are using each component's center of mass as reference point.

5.1.1 Set of rules

The guidelines are split up into preparation of parts used for the assembly and into rules for constructing said assembly and its relations correctly. The regulations are separated into three paragraphs: assembly rules, subassembly rules and a method for distinguishing between these mutually exclusive situations. This set of rules must be followed to create the *CAD* model that can be built by the presented system.

5.1.1.1 Prepare parts for assembly

Since the initial scene is arbitrary and predefined, for each component of the assembly, the ideal grasp point needs to be identified. This means, that the location needs to be found, where the part should be grasped ideally.

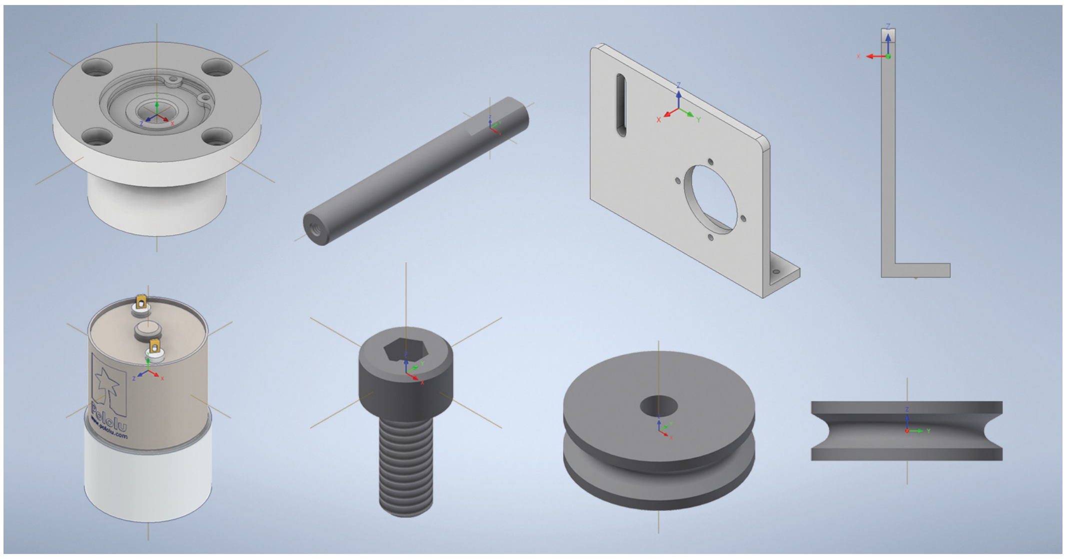


Figure 5.3: Grasp points of different components from the Drive Belt Unit. The two most right parts are presented from two different viewing angles.

Additionally, it is important to identify the direction from which the gripper should approach each part.

The grasp pose is defined inside the coordinate system of individual components. This means that the internal origin and the rotation compared to the Z axis of the components coordinate system define the grasp point. The position is identified by taking the center of the two ideal gripper contact points. The center of mass of the object needs to be shifted, so that the origin of its coordinate system represents the chosen grasp position. The orientation of the grasp point is defined as the positive direction of the Z axis of the object. This means that the direction, from which the gripper will approach the part, is coming from $Z \rightarrow \infty$ and is going towards $Z \rightarrow 0$ inside of the part's local coordinate system. The component needs to be rotated in its own coordinate system for this previously identified grasp direction to overlap with the Z axis of the simulation. Examples of possible grasp poses can be found in Figure 5.3. Please note that a parallel gripper is intended to open along the X axis of the grasp pose.

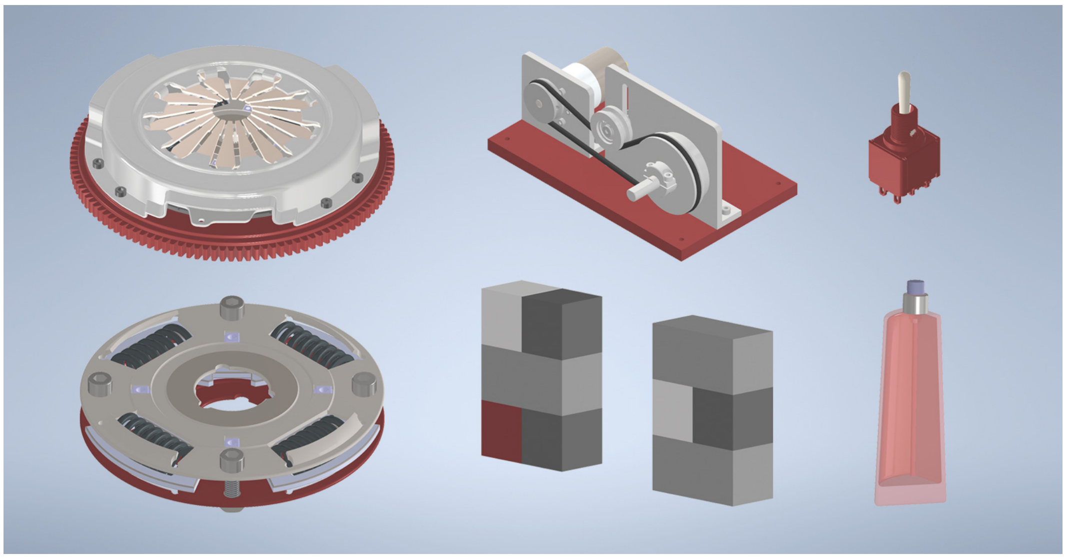


Figure 5.4: Grounded parts of examples from Chapter 7 highlighted in red color.

Furthermore, the grounded part of the whole assembly needs to be identified. Generally there must be exactly one grounded component, since all other parts can be placed referencing this single starting point or components connected to it. A grounded part will not be moved by the robot arms and thus is the starting point of the assembly. If a component in an assembly is marked as grounded, the model has all it's degrees of freedom removed and therefore can no longer be moved inside many *CAD* environments, including Inventor.

In general this object is the most intuitive starting point of the assembly. Most of the time the assembly is built on top of said component, which makes it the object with the lowest minimal *Z* coordinate. The grounded part can also be the component that is most covered by adding all other assembly components. Examples of grounded parts for all assemblies used in Chapter 7 are highlighted in Figure 5.4 and more detailed step-by-step solutions are visualized in Chapter 7. The final pose of the grounded piece is dependent on the rest of the assembly.

5.1.1.2 Component place order

The first object to be placed in a new and empty scene is the grounded part g. The floor of the simulation is assumed to be the $X-Y$ plane of the simulation and only the positive quadrant of the 3D scene is allowed to contain any components. This is necessary for the transformations to work as expected. g must therefore be placed inside and parallel to the positive quadrant of the assembly's $X-Y$ plane, so that the rest of the assembly is built towards the positive Z direction of the simulation without components entering any other quadrant.

```
inputs : $c$ ... last placed component
HandlePart ($c$)
    $I_c \leftarrow$ find all insertion locations on $c$;
    $P_c \leftarrow$ find all not placed parts that need to be attached to $I_c$;
    foreach insertion $i \in I_c$ do
        $p \leftarrow$ part $\in P_c$ with smallest offset to $i$;
        if $p$ can be placed on $c$ without any collision then
            Assembly rules apply on $p$;
        else
            Subassembly rules apply on $p$;
    if $\exists$ part that has not been placed yet then
        foreach part $p \in P_c$ do
            HandlePart($p$);
```

Algorithm 5.1: How to place components in a correct order

Initially the grounded part g must be placed in the scene. Algorithm 5.1 describes how to handle a component that was placed, so the following workflow starts by calling Algorithm 5.1 with g as parameter c.
Next, all parts P_c related to c need to be found and placed in the correct order. Each component of P is inserted along an axis towards a center point of a surface of g, which is called an insertion location. To find all insertion locations I_c and all parts P_c that need to be attached to I, see Section 5.1.1.3. For each insertion i in I_c, the part p out of P_c with the smallest offset to

i needs to be determined. The offset is defined by the distance between the two center points of the related surfaces and was already discussed in Section 2.9.3.
The general assembly rules described in Section 5.1.1.4 apply, if p can be placed on c without any collision. That means the component is not allowed to intersect with any other already placed part during any point of the insertion trajectory. A simple example of a pure assembly without such collisions can be found in Section 7.1.1.
Else, a subassembly is formed by p and any other component that is connected to p, including already placed parts. In this case the in Section 5.1.1.5 described subassembly rules apply. A simple example of a subassembly, an assembly with colliding insertions, can be found in Section 7.1.2.
This concludes with all components in P_c attached to g. Then each object p in P_c is handled as a new reference part c and all steps so far are repeated for it. This procedure stops when all components are placed.

5.1.1.3 Identifying related parts

To identify all related parts P_c of a component c, all insertion locations I_c need to be determined. All components P_c connected to c are referencing a specific surface of c with the relation placed between them. Insertion locations I_c are defined as an axis facing towards a center point of such a surface.

Components, that have already been placed inside the assembly are excluded from P_c. If P_c is empty, c does not have any parts that need to be attached to it anymore.

5.1.1.4 Assembly rules

The following paragraphs describe rules that apply for the general *CAD* model creation.

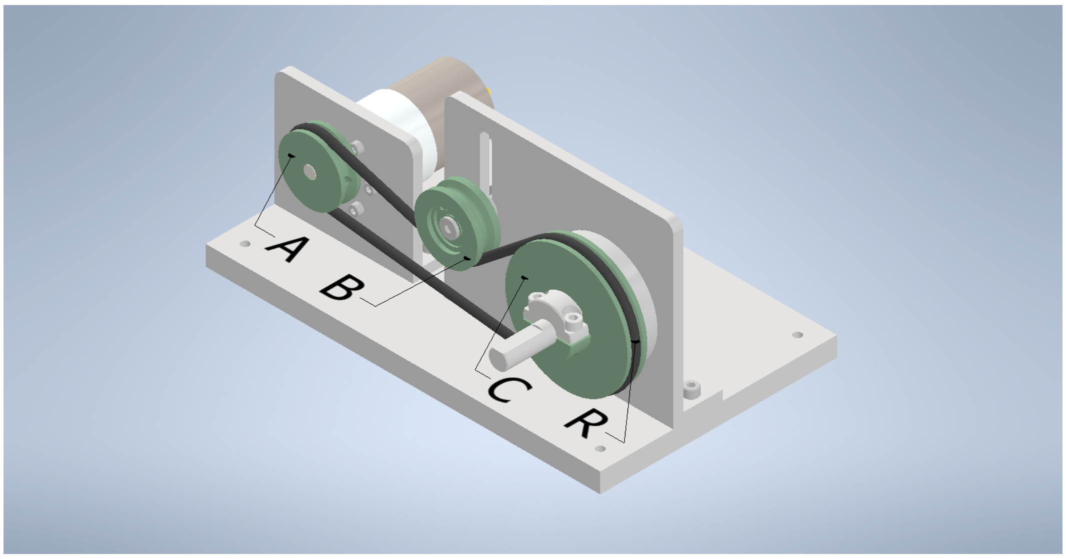


Figure 5.5: Rubber band R from Drive Belt Unit is connected to parts A, B and C, highlighted in green color. The resulting connections are between $(R, A), (R, B), (R, C)$.

Constraining two parts using as few relations as possible To connect two components via relation in the assembly it is encouraged to use the least amount of constrains or joints possible, preferably one per connection. Joints should be preferred over constraints in an assembling context, since they combine multiple constraints into a single, more high level constraint type. The only joints allowed in this work are rigid and rotational. A rigid generally represents a screw, unless defined otherwise.

Constraining a single part with many To connect a component with multiple parts it is recommended to use constraints instead of joints, since high-level joints might over-constraint the scene if the parts are not placed perfectly. As an example the rubber band from the *BDU* can be seen in Figure 5.5. In the image the constraints to three other parts are highlighted. In this case three circular geometries define the three contact points of the rubber band. This method is used to model shape-adapting or elastic components.

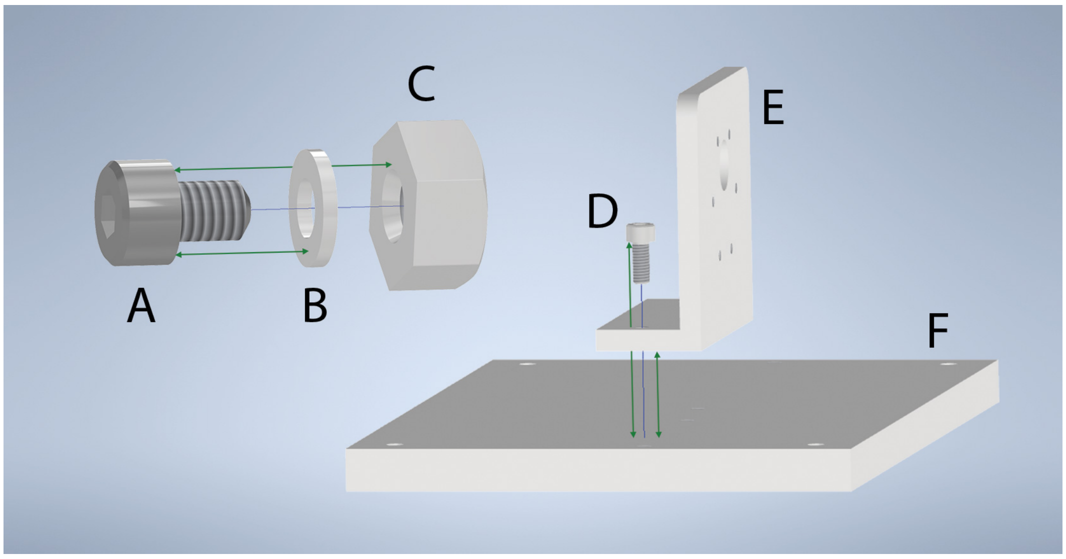


Figure 5.6: Examples for intended non-face-to-face (left), non-intended face-to-face (center) and intended face-to-face (right) connections between two parts. The letters are relevant to the description inside the running text.

Choosing the correct parts to connect Relations should be placed between parts that are *intended* to be connected. What intended means in this context can be best explained using examples. If a component is inserted *into* another part, the relation is intended. But if a object C is inserted *through* another component b into a third part A, where A is grounded, both C and B are related to A. Figure 5.6 shows examples of such intended connections, with a screw A, a spacer B and a nut C on the left side, where A is the grounded part. The resulting relations are (A, B), (A, C). Adding a relation (B, C) instead of (A, C) would mean the nut is being connected to the spacer using a face-to-face relation. But the nut is meant to be connected with the screw to fixate objects, that where previously loosely inserted on the screw. This is valid as long as more than two parts insert on the same axis towards the same point, which can be seen on the right half of 5.6. Else, a standard face-to-face combination between the two components can be applied.

Grounded parts are parents Grounded parts are always parents in their relations to other assembly objects. Thus everything that is being attached to grounded components must be the child in the joint or constraint placed between them. Following the approach described above, all children of grounded parts become the parents for new relations added to the next set of inserted assembly objects. This is repeated until either the assembly runs out of components needing to be attached and is successfully assembled, or a subassembly is identified. More information on the identification of subassemblies can be found in Section 5.1.1.5. When interpreting the represented data inside the *CAD* model as a tree-like representation, this rule enforces a *directed* graph structure, meaning all edges have a specific direction they point to. If multiple grounded parts would be allowed, this individual graph would become a forest, which might improve usability for use cases such as the example described in Section 7.1.4. Because each following component can be placed referencing a single grounded part, this is not necessary. The graph can be traversed as long as there is a singular starting point, meaning there can only be one grounded component per assembly.

Please note that in regard to Inventor, the child is always selected first when adding a new joint or constraint. An example for the selection method of Inventor can be seen in the left image of Figure 5.4. The child is already selected and highlighted in blue.
The right image shows all three components of an electrical switch, an evaluation example discussed in more detail in Section 7.1.1. The parts A, B, and C referenced in the image are related to each other, such that A is parent to B and B is parent to C.

Choosing the correct constraint surfaces Joints or constraints should always be placed between surfaces with a defined center point. The current interpretation of the geometry is able to identify the insertion direction of components using the normal of each surfaces' center point. Examples of a valid and a invalid relation surfaces are depicted in Figure 5.8. The surface's inverse of the normal is interpreted as the insertion direction of the components, which implicitly restricts all components of the assembly

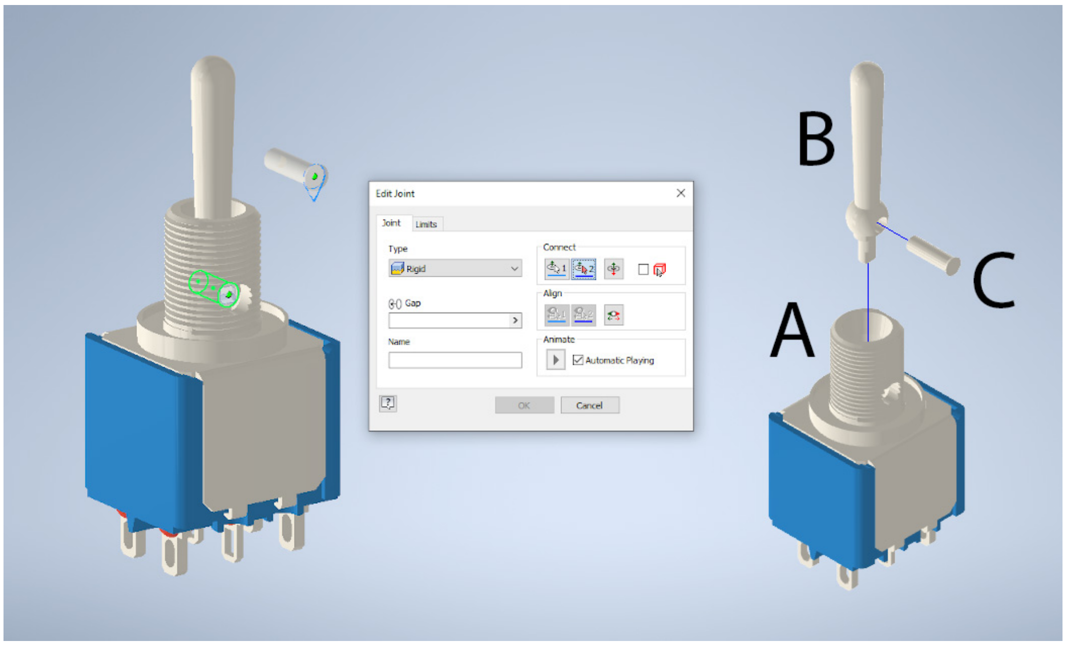


Figure 5.7: Examples for the parent-child order in regard to grounded parts. Left and Center: Inventor interface for adding joints. Right: A three-component parent-child hierarchy of a simple electrical switch from Section 7.1.1 consisting of a *case A*, a *buton B* and a pin *C*.

to have a straight insertion trajectory, meaning components like bolts, shafts, slots and holes are only allowed to have the shape of cylinders.

Pre-assembled components Some components come pre-assembled in the initial scene, but they need to be handled separately inside the assembly. Possibly some parts need to be moved or tightened after being inserted into the assembly. The joints or constraints between these objects need to be marked as pre-satisfied relations.

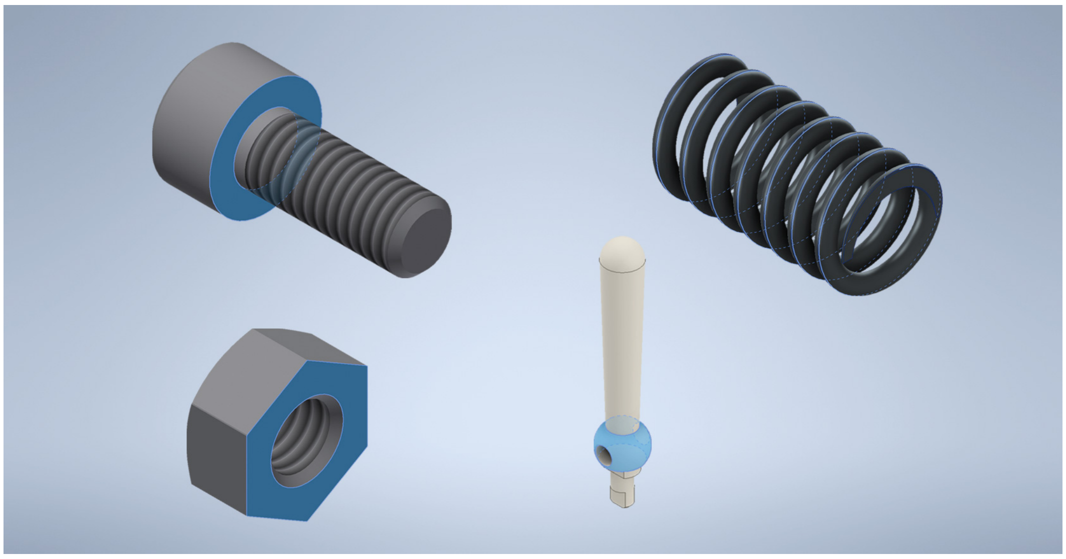


Figure 5.8: Examples of valid and invalid relation surfaces. Left: Valid relation surfaces of a screw (top) and nut (bottom). Right: Invalid relation surface of a spring (top) and a sphere (bottom).

5.1.1.5 Subassembly rules

In the following paragraphs a second set of rules is discussed, used to model subassemblies. A subassembly is formed by a parent and multiple child components, including already placed parts. All child parts must reference the same insertion location of the parent. A simple example of a subassembly can be found in Section 7.1.2.

When to create a subassembly As previously mentioned, a subassembly needs to be applied if the insertion of components in the order presented in Section 5.1.1.4 leads to objects colliding with already placed components and passing through them when being inserted into the assembly. If the subassembly still leads to collisions, either the wrong parent component was selected or not the right group of components are part of the subassembly.

Parent and child are switched around Following the normal assembly rules, new components are always added as children. But this is not the case within subassemblies. During a subassembly the parent-child hierarchy model is switched around. This means that the newly inserted component P, that would have been the child of an already inserted part c, now becomes an additional parent of C. P is now the parent of the subassembly. All components that need to be stacked on top of P, including C, must be children of P when inserted.

Choosing the correct constraint surfaces Inside a subassembly, children must always refer to the same reference surface of the parent. The insertion order of children within a subassembly is defined by the offset from the child-surface to said parent-surface. Therefore it is important to make sure that all referenced child-surfaces are chosen so that the offset to each child is increasing. In doing so, it represents the correct placing sequence of all subassembly components. Figure 5.9 shows examples for both a correct and incorrect selection of child-surfaces between a bolt A, a spacer B, a wheel C and a nut D. The same component C is shown twice on the right side of the image. The one on the bottom presents the correct execution sequence (A, B), (A, C), (A, D), whereas the top image leads to an incorrect handling order (A, C), (A, B), (A, D), because the selected surface of C is closer to the A then the one from B.

When not to create a subassembly A situation might occur, in which a subassembly does not stack on a single shaft, but rather on multiple. This is for example the case for a series of components stacked on top of each other, held together by attaching multiple screws and nuts between the two outermost objects. The only difference to a subassembly is the amount of shafts on which the components stack. During a normal subassembly there is one single central insertion location for all components. But in this case, if seen as a subassembly, multiple components would stack on many insertion locations. This problem arose during the evaluation phase in Section 7.1.3 and more information about it can be found there. Generally, there are two different types of solutions applicable to this scenario, but neither of them is a subassembly. Applying the subassembly rules would imply that all screws

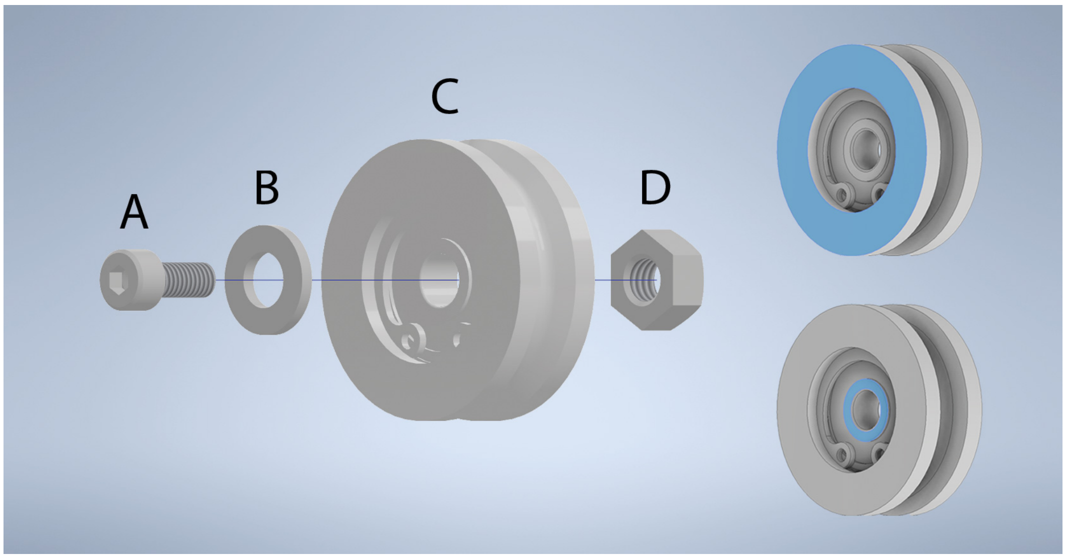


Figure 5.9: Left: Subassembly between a bolt *A*, a spacer *B*, a wheel *C* and a nut *D*. Right: Examples of invalid (top) and valid (bottom) child surface selections for the subassembly.

fixating the individual parts together should be parents to all components of the subassembly. This either needs a separate robotic arm for each parent, a single gripper being able to hold many tiny objects far apart or the ability of the system to stack components on a non-centered shaft without causing any components to rotate because of gravity and the holding orientation. The first solution is using the Inventor plugin to enforce a specific execution order by creating an assembly with different steps in time, which is discussed in more detail in Section 5.1.2.2. This can be useful, if the closure of the previously mentioned example fails because one of the two sides of the fixating screws is no longer accessible. This can happen, since the subsection of the assembly is built by placing the first component at its final location and therefore blocking access to bolts or nuts.

Secondly, it is theoretically possible to create nested assemblies. This solution is not currently necessary, since a functional workaround is to assemble this pseudo-subassembly beforehand and interpreting it as a single part. The inclusion of an assembly inside an assembly, also called nested assemblies, is part of future expansions and will be further discussed in Chapter 8.

Choosing the correct reference surface of parents A cylinder has two directions from which components can be inserted onto it. One of them is in close proximity to the grasp pose and is thus generally blocked by the gripper. This is obvious for a subassembly parent that is a screw, because one of two sides is blocked by its head. Also, the head of the screw is where the grasp position is located. Therefore, the surface of the subassembly parent, that is referenced by all relations of subassembly children, is found at the same end of the insertion axis where the grasp pose usually is. This can be seen in the examples presented in Figure 5.10.

Please note that the child, that is a screw and closes this subassembly chain, should have the largest offset to the parent, in comparison to all other children of the same subassembly. Looking again at a screw as an example, the thread on the screw most commonly resides at the opposing end of the components head.

5.1.2 CAD model creation

Every component needs to be placed in the assembly following the guidelines presented in Section 5.1.1 to comply with the norms defined during this work. Once each part is inserted correctly and the assembly is constructed, the creation of snapshots needs to be taken into consideration. A snapshot represents the current state of the assembly in time, including the necessary information about all involved relations and components. A sequence of snapshots can therefore be used to model specific order of actions inside an assembly.

5.1.2.1 Preparation of relations for snapshots

A single snapshot of the final assembly needs to be added using the Inventor plugin, if changes to the model in form of shifting or tightening relations

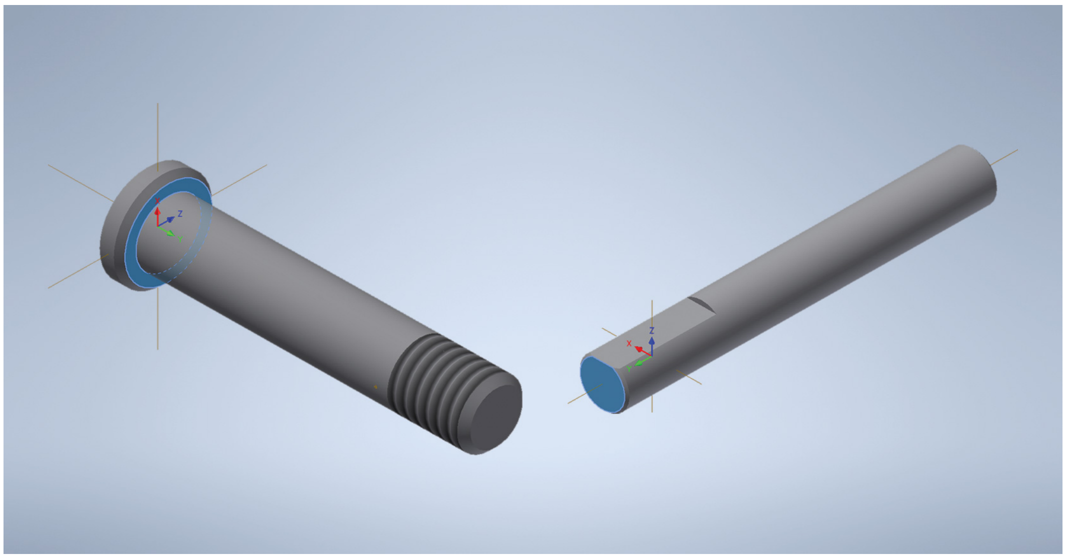


Figure 5.10: Examples of the correct reference side for a bolt (left) and a shaft (right) in a subassembly. Grasp positions are shown as small markers on the left side of each component.

are not needed. Also one snapshot is enough if no specific execution order of actions during the assembly needs to be enforced. More detail regarding the creation of snapshots can be found in Section 5.1.2.2.

Depending on the task and the skills involved, intermediate snapshots might be necessary to enforce a specific order of execution between individual steps of the assembly. There are mainly two different reasons for the creation of intermediate snapshots, which both need different preparations.

Changes of relations between the same parts Sometimes a joint or constraint needs to be changed during the assembly. A possible scenario is the tightening of pre-satisfied screws after the insertion of a different component. This situation appears during the belt drive assembly and is further expanded upon in Section 7.1.
Also, some components might be moved during the assembly after the insertion of additional components. Such a case can again be found during the composition of the World Robot Summit assembly challenge, where the

middle wheel needs to be pushed down to create tension on the rubber band, and is also described in Section 7.1.

For both examples additional relations need to be added to the *CAD* model instead of changing the existing ones, since most *CAD* environments do not allow temporal changes to their joints or constraints. After hiding the already existing relations representing the initial state, new relations representing the end state should be placed between the same components with the final offset. The old connection will be replaced by the new relation using a snapshot. See Section 5.1.2.2 for the creation of the individual snapshots using the Inventor plugin.

Adding new parts after specific changes Some components should not be placed by the robots until a specific relation was changed to its final position. An example for such a case can be found during the execution of the *BDU*, where a pre-satisfied screw needs to be fixated before the rubber band covers the access to the screw's head. More information is available in Section 7.1. In this case no additional joint or constraint needs to be added, only a correct snapshot creation is necessary, as discussed in Section 5.1.2.2.

5.1.2.2 Creation of snapshots

To create a snapshot, all to this snapshot not relevant parts and all their relations must be suppressed, meaning made hidden inside the *CAD* file. Parts might be irrelevant because they are added during a later snapshot. Relations might be irrelevant because they are also added later on or because they either have changed or will change. All the needed components and their connections to other needed parts should be unsuppressed, visible inside the three-dimensional representation. Then, using the Inventor plugin, a new snapshot can be created using the UI interface described in Section 6.1.2.

Every assembly needs at least one snapshot If there is no part that needs to be moved and no relation that needs to be changed after the insertion of an additional component to the assembly or after being connected with another object, the assembly does not need any further manipulation. Only a single snapshot of the full assembly, with all relevant components and connections unsuppressed, needs to be created using the Inventor plugin.

Changes of relations between the same parts There are many examples of relations that need to be changed during the *BDU* assembly, evaluated in Section 7.1. Most of them are pre-satisfied relations from parts that come pre-assembled, where for example a screw meant for fixing a wheel on a rod is already inserted in the wheel by the manufacturer or a previous assembly process. The only thing for the assembly left to do with said screw, is screwing it in further once the wheel is inserted onto the shaft.

One of the connections that need to be changed along the assembly process of the *BDU* is a subassembly in form of a spinning wheel on a bolt. This subassembly needs to be pushed down along a oblong slot after the rubber band is inserted around the other two wheels. This way, external pressure using this third wheel being pushed down can create tension on the belt. A more detailed explanation of this scenario can be found in Section 7.1.

Depending on the assembly all changes can be put in place using a two snapshots. Alternatively, an individual snapshot for each change can be created. Assuming an example with only one moving part, the first snapshot would contain the connection restricting the moving component to its initial insertion point. The second snapshot would no longer contain the first relation of the movable component, but rather the second connection, constraining it to its final position in the assembly.

Adding new parts after specific changes It is possible for a component, once inserted, to block the access to another part. For example in the *BDU* a screw inside one of the three wheels can only be accessed as long as the rubber band is not placed. Assuming the screw was not pre-assembled but has to be inserted, the assembly would be defined by only two snapshots. The first one would contain the insertion of the screw, the second one the

placement of the rubber band. A further expansion on this example can be found in Section 7.1.
If a placed component blocks the access to parts that need to be changed because they move or they need to be tightened, the change must happen in the snapshot before adding the new component.

Blocked access to parents or to new children As mentioned in Section 5.1.1.5, when talking about *when not to create a subassembly*, the example from Section 7.1 is mentioned. In that case, a specific segment of the assembly contains stacked components fixated together through many radially arranged equidistant screws between the two most outer components *a* and *b*. While building the assembly the time arrives, where part *a* needs to be placed. Since *b* would be placed later than *a*, the screws would point from *a* to *b* or vice versa. This means that either the access to the head of the screw or the access to the nut closing the other side of the screw is blocked. One solution to this problem would be to add a snapshot containing only *a*, *b*, all parts in between and the screws used to fixate the components before any other snapshot. An alternative solution presented in form of a future expansion in Chapter 8 is the addition of nested assemblies.

5.1.3 Model analysis

The following section revolves around model analysis and data representation after extraction from Inventor. The presented class structure is used for the plan generation while also including the generated plan itself.

5.1.3.1 Data representation

The definition of the data model for the knowledge base resides in a base file, *assembly_base.owl*, and was developed as part of this work. A second knowledge file regarding the plan is automatically generated by instantiating the classes and properties defined in the base to individuals with attributes using the presented Inventor plugin. A graphical representation of the base

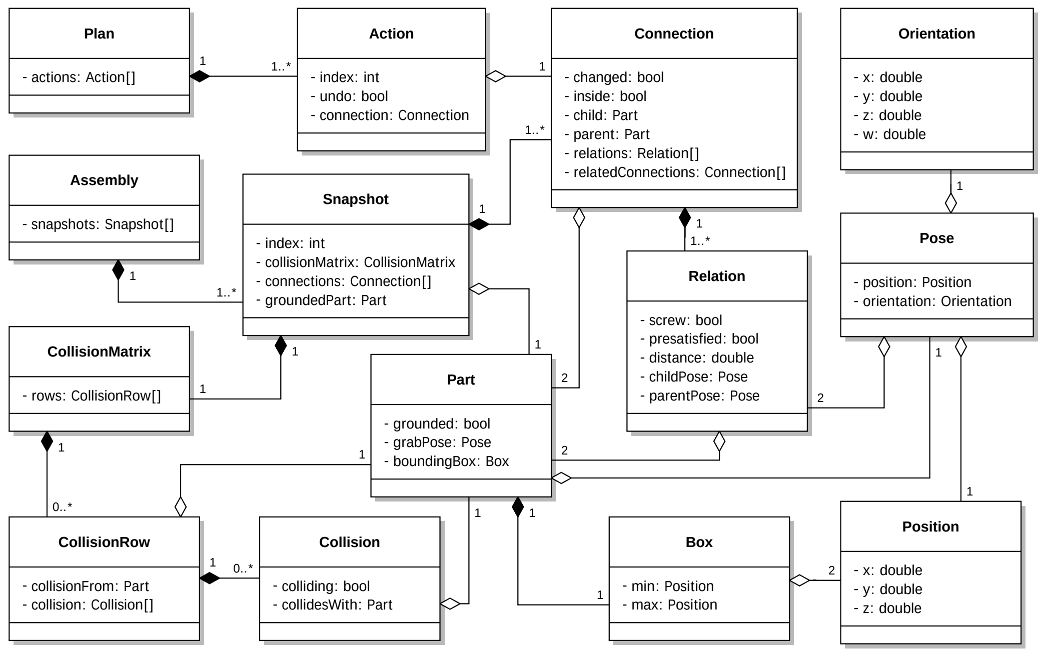


Figure 5.11: Class diagram representation of the data structure of the concept.

classes is presented in Figure 5.11 in form of a Unified Modeling Language (UML) class diagram.

5.1.3.2 Data structure

The data structure represented in Figure 5.11 can be categorized into the following two aspects, where some information is used in both data models. The class names used in the upcoming paragraphs also reference to the names used in Figure 5.11.

Analysis of CAD model data The topmost class representing the full assembly is named *Assembly*, which contains one or more so called *Snapshots*. Guidance on how to create snapshots can be found in Section 5.1.2.2. Each snapshot includes one or more *Connections* and a grounded part as a starting point for the assembly. A snapshot also contains a collision matrix A denoting in a binary manner the existence of collisions between all parts, such as

$$A = \begin{bmatrix} a_{00} & \cdots & a_{0j} \\ \vdots & \ddots & \vdots \\ a_{i0} & \cdots & a_{ij} \end{bmatrix} \quad | \quad a_{ij} = \begin{cases} 1, & \text{if part } i \text{ is in collision with part } j \\ 0, & \text{otherwise.} \end{cases}$$

Parts are the components of the assembly and contain information like a grasp pose, a bounding box and the binary information on whether they are grounded or not. There can only be exactly one connection between the same two components, so connections also contain references to the parent and child components and a list of one or more relations between the same parent and child. *Relations* are the joints or constrains placed between two components, have a distance, also called offset, contain the transformation between parent and child, can be of a screwing type and / or pre-satisfied. A *Pose* contains both *Orientation* and *Position*. *Boxes* are simplified to be the bounding box inside the local coordinate system of the component consisting of six values, referencing positive and negative distances on all three axes from the center of mass of the object.

Generation of plan data In regard to plan generation, some data needs to be extracted from the *CAD* model to enrich the graph-like structure presented by the combination of parts using connections. The full extracted information is then converted to a sequential *plan* of *actions* with a fixed order. A detailed explanation on plan generation can be found at Section 5.1.4.

5.1.3.3 Data calculation

To extract the necessary information about the model, the Inventor *API*, described in Section 2.9.1, is used. The following four sections contain explanations about what information is kept how and where by Inventor.

Parts For each existing component inside the assembly, a individual part instance is initialized. The component in the *CAD* file, that is marked in Inventor grounded, is the only part with that property set to true. All others are initialized as not grounded. The grasp pose is either defined by the user or using a heuristic approach as the center of each component's local coordinate system, as explained in Section 5.1.1.1. The position of the component's actual grasp pose is calculated as the three-dimensional distance from the center of mass to the user defined or heuristicly reached grasp position. The actual grasp orientation is given by the rotation of the parts local coordinate system, which is also defined by the user grasp pose. The *API* property *RangeBox* inside an *occurrence* is defined in absolute coordinates. Therefore, the bounding box is calculated by subtracting the center of mass from the minimum value of the *RangeBox* or vice versa by subtracting the maximum value from the center of mass.

Connections For each pair of parts connected by one or more joints or constraints, a connection is added to the current snapshot. Who of the two involved components is the parent and who is the child, is defined by the correct placement of the joint or constraint, as described in Section 5.1.1. Changes from one snapshot to the next are denoted by the property named *changed*. It is identified using the algorithm *IdentifyAffected* described in Section 5.1.4.10.

Only partially implemented is the use of the list *relatedConnections*. If a connection is referenced as related to other connections, it means that all related connections need to be satisfied before the connection to their common child can be satisfied. The property *inside* is an example for a possible future expansion of handling multi-contact components using related connections and is further expanded upon in Section 7.1.5 and in Chapter 8 as a future expansion. Multi-contact connections make only sense if some sort of deformable object is present, like a belt. In that case the property denotes whether the parent is on the inside or the outside of such a component, which can be used to automate the insertion sequence of deformable components.

Relations All joints or constraints with the corresponding parent and child are added as relations to each connection. Whether a relation is pre-satisfied or representing a screw is determined by the user. The transformation between parent and child after being placed inside the assembly is encoded using the child's pose and the parent's pose. The distance property references the Euclidean distance between the related surface's center points and is presented by Inventor separately from the transformation.

Collisions To find collisions inside a Inventor assembly file, the analyze tool described in Section 2.9.1 is used via the *API*. The resulting collisions are not the expected outcome since Inventor only identifies overlapping components as collisions. Therefore the only results are delivered for threads in nuts and on bolts and for mistakes in the assembly, since no components surface should intersect with any other surface. To get the needed type of collision, meaning *which component is in contact with which component*, a heuristic is chosen, where intersecting bounding boxes are considered as collisions.

5.1.4 Plan generation

Based on the information, rules and conditions presented until now, a fully automatic action sequence generation can be applied to a collection of snapshots representing an assembly, which results in a plan. As previously mentioned in more detail, this plan represents of a sequential list of actions leading to a successful assembly. The plan extraction is presented in form of a set of algorithms followed by detailed descriptions.
The generated action plan is then interpreted for a specific scene, which consists of the previously mentioned dual-arm robotic setup and a table for the assembly components. Depending on the scenario and the given robot skills, interpretations with as many arms as given by the system can be implemented. More on this topic can be found in Section 5.1.5.

For readability reasons, compactness and better understanding, the main algorithms are split up into different smaller algorithms, which are denoted as function calls inside other algorithms.

5.1.4.1 Generate plan

```
inputs : $S$ ... ordered list of all snapshots of the assembly
output: $\pi$ ... sequential plan of actions of the assembly
GetPlan ($S$)
    initialize $z$ as $null$;
    foreach $snapshot\ s \in S$ do
        initialize $A$, $T$ as $\emptyset$;
        $g \leftarrow s.groundedPart$;
        $A \leftarrow$ GetExecutionOrder($g$, $A$, $s$, $T$, $null$);
        if $s = first\ snapshot$ then
            append execution order $A$ to $\pi$;
        else
            $s \leftarrow$ FindDifferences($s$, $z$);
            $U$,$R \leftarrow$ IdentifyAffected($A$, $s$, $z$);
            $\pi \leftarrow$ AddAffected($U$, $R$, $\pi$);
        $z \leftarrow s$;
    return $\pi$;
```

Algorithm 5.2: Get the plan for the full assembly

The general workflow of extracting a full plan is represented in Algorithm 5.2. The expected input consists of a list of all snapshots S representing the whole assembly sorted by their creation order. The output is the final plan, an ordered list of actions π needed to successfully finish the assembly.

After the initialization of needed local variables, an empty list of actions A and an empty list of already traversed parts T, each snapshot is analyzed. For each snapshot s, the grounded component g is identified. Then the correct execution order of actions A is extracted by stepping through the assembly connection graph, starting from the grounded component (see Section 5.1.4.2).

If s is the first snapshot in S, then all actions in A should be appended to the output π.
Otherwise, differences between the current snapshot s and the previous

snapshot, given by changes in relations or components, need to be identified (see Section 5.1.4.9). Actions inside the current execution order A, that are affected by the differences, have to be found (see Section 5.1.4.10) and added to the plan π (see Section 5.1.4.11). The affected differences are two lists of lists of affected connections separated in actions to undo U and actions to redo R. Each list in U and R corresponds to a group of actions arising from an individual change between snapshots.

5.1.4.2 Get execution order

```
inputs : $p$ ... root part of the search,
         $s$ ... current snapshot,
         $T$ ... already traversed parts,
         $A$ ... list of actions representing the plan,
         $h$ ... halting part for recursive parent handling
output: $A$ ... modified list of actions representing the plan

GetExecutionOrder $(p, s, T, A, h)$
    if $h \neq null$ and $p = h$ then return $A$;
    $R \leftarrow$ get all connections of $p \in s$;
    $G \leftarrow$ Group($R$, $s$);
    $C \leftarrow$ Sort($G$);
    $\hat{P}, \hat{C} \leftarrow$ GetParentsAndChildren($p$, $C$, $A$);
    append $p$ to $T$;
    $A \leftarrow$ handleParents($\hat{P}$, $s$, $T$, $A$, $p$);
    foreach child $c \in C$ do
        if $\neg$ ShouldWait($c$, $A$) and $c.parent = p$ then
            append new action pointing at $c$ to $A$;
    $A \leftarrow$ handleChildren($\hat{C}$, $s$, $T$, $A$, $h$);
    return $A$;
```

Algorithm 5.3: Get execution order of snapshot actions

Algorithm 5.3 describes the extraction of the execution order of an individual snapshot. It is the most relevant step of the plan extraction for single-snapshot assemblies, since in that case the output represents to the full plan. The inputs for this sub-algorithm are a root part p, which is the starting point of the graph search, the current snapshot s and a list of root parts T already traversed in previous recursive calls of this algorithm. The output is a list of actions sorted by the correct execution order inside the current snapshot.

The component h represents a halting point for some of the recursive expansions of this algorithm. When h exists and is equal to p, the halting point is reached for the current branch and this recursive execution is interrupted.

All connections R of the initial root part p inside the s need to be found. This includes connections with both p as parent and as child. The connections in R are grouped by collisions (see Section 5.1.4.3). The resulting groups G are a list of lists of connections. Each list in G corresponds to components that are in collision with each other. Within these groups the connections are then sorted by their largest relation offset and finally all groups are concatenated to a single list of connections C (see Section 5.1.4.4).

For the recursive expansion of the algorithm to all nodes, it is important to split child parts of p and their parent parts into separate lists (see Section 5.1.4.5), since they must be handled differently, where $\hat{P}$ contains parents and $\hat{C}$ contains child components. At this point the root component p can be viewed as traversed and should be added to T. Before adding the list of sorted connections C to the output, all parents of p need to be handled (see Section 5.1.4.7). Handling parents means traversing the graph from a different root point until it reaches the original component connecting the parent subgraph with the main branch. This is defined in the halting point h, which is *null*, until a parent component needs to be handled. Algorithm 5.8 identifies the connecting component between the sub and the main branch and passes it to the recursive call of Algorithm 5.3 as h.

After handling the parent components, for each connection c in C, it is important to check if c should not wait for related connections (see Section 5.7) and if the parent component of c is equal to p. If both are true, a new action pointing to c is added to the output A. Finally, all children of p need to be handled (see Section 5.1.4.8).

5.1.4.3 Group connections by collision

```
inputs : $C$ ... connections that need to be grouped by collisions,
         $s$ ... current snapshot
output: $G$ ... groups of colliding connections

Group ($C, s$)
    add all connections $\in C$ to pool;
    add first element $e$ of pool to a new group $\bar{G}$;
    remove $e$ from pool;
    while length of pool $> 0$ do
        $e \leftarrow$ get next pool element;
        if $\exists$ collision $\in s$ between $e$ and any element of $\bar{G}$ then
            add $e$ to $\bar{G}$ and remove from pool;
        if all elements of pool are checked once then
            if no changes in pool after all elements checked once then
                add $\bar{G}$ to $G$ and clear $\bar{G}$;
                add first element of pool to $\bar{G}$;
            else if no changes in pool after all elements checked twice then
                add pool to $\bar{G}$ and clear pool;
                add $\bar{G}$ to $G$ and clear $\bar{G}$;
    return $G$;
```

Algorithm 5.4: Group connections by collision

Algorithm 5.4 describes how to group connections based on collisions of their parent or child components. The inputs are connections C that need to be grouped by collisions and the current snapshot s. The output for this sub-algorithm are groups G of connections that are in collision with each other.

All connections C, that should be grouped by collisions, are added to the *pool* during initialization. Additionally, an empty list of groups G needs to be set up. The first element of the pool is added to a new group $\bar{G}$.

As long as *pool* contains elements, the following algorithm loop will not stop. If the next element e in the *pool* is in collision with any element of $\bar{G}$, then e is added to $\bar{G}$. As a side node, the element after the last one in *pool* is the first one again.

If e is the last element in *pool*, then the following subroutine is entered. If *pool* did not change after each element was checked once, $\bar{G}$ is added to G, $\bar{G}$ is cleared and the first element of *pool* is added to the new $\bar{G}$. Else, if *pool* did not change after each element was checked twice, all elements in *pool* are added to $\bar{G}$, $\bar{G}$ is added to G and both $\bar{G}$ and *pool* are cleared.

5.1.4.4 Sort connections within groups

```
inputs : $G$ ... groups of connections in collision
output: $S$ ... list of sorted connections

1 Sort ($G$)
2     foreach group $g \in G$ do
3         initialize $\varnothing$ list of $D$ connection-distance tuples;
4         foreach connection $c \in g$ do
5             $d \leftarrow$ get largest distance of relations of $C$;
6             append tuple of $c$ and $d$ to $D$;
7         sort $D$ by smallest distance from each tuple;
8         append connection of each tuple $\in D$ to $S$;
9     return $S$;
```

Algorithm 5.5: Sort connections within groups

For Algorithm 5.5 the inputs are the groups of connections G that are in collision. These connections are sorted by the distance of their most far away relation and then concatenated into a single list of sorted connections S, which is the output of the algorithm.

Each connection contains references to relations, which themselves contain distances between parent and child. These longest distance of all relations inside a connection is used. Then the connections are sorted in-place by this distance and afterwards added to the output S.

5.1.4.5 Get parents and children list

```
inputs : $p$ ... root part not to append,
         $C$ ... sorted connections,
         $A$ ... list of already handled actions
output: $\hat{P}$ ... parents list,
        $\hat{C}$ ... children list
1  GetParentsAndChildren ($p$, $C$, $A$)
2      initialize $\hat{C}$, $\hat{P}$ as $\varnothing$;
3      foreach connection $c \in C$ do
4          if $\neg$ ShouldWait($c$, $A$) then
5              if $c.child \notin \hat{C}$ then
6                  append $c.child$ to $\hat{C}$;
7              $B \leftarrow$ connections where $child = c$;
8              foreach connection $d \in B$ do
9                  if $d.parent \neq p$ and $d.parent \notin \hat{P}$) then
10                     append $d.parent$ to $\hat{P}$;
11     return $\hat{P}$, $\hat{C}$;
```

Algorithm 5.6: Split children and their parents into separate lists

Algorithm 5.6 describes the procedure of separating children components and their parent parts of the sorted connections into separate lists. The inputs are the root part p that should not be appended to any of the lists, a list of sorted connections C and a list of already handled actions A representing the plan. The outputs are a list of parents $\hat{P}$ and a list of children $\hat{C}$ connected to p.

Separate and empty lists for children $\hat{C}$ and their parents $\hat{P}$ must be initialized. For each connection c in C, it needs to be checked if c should wait for related connections (see Section 5.7). If that is not the case, the part of the connection that is not p needs to be added to one of the lists $\hat{C}$ or $\hat{P}$. If the part in regard is the child of C, it must be added to $\hat{C}$. If the component is the parent, it must be added to $\hat{P}$. For each connection d where c is the connections child component, if the parent component of d is not p and also not already in $\hat{P}$, then it is added to $\hat{P}$. On the other hand, if the child object of c is not already in $\hat{C}$, it is added to $\hat{C}$.

5.1.4.6 Should wait for dependent connections

```
inputs : $c$ ... connection that is checked for dependencies,
         $A$ ... already handled actions
output: $true$ or $false$

1 ShouldWait ($c$, $A$)
2     foreach connection $r \in c.relatedConnections$ do
3         initialize $h$ as $false$;
4         foreach connection $a \in A$ do
5             if $r \in a.connection.relatedConnections$ then
6                 $h \leftarrow true$;
7         if $h = false$ then return $true$;
8     return $false$;
```

Algorithm 5.7: Should wait for related connections

The check, if all combined relations are already handled, is described in Algorithm 5.7. The inputs are the connection c that needs to be checked and a list of already handled actions A. The output is either $true$ or $false$.

Each connection contains at least one relation, which can reference to a set of combined relations. These need to be handled, before their common child can be traversed. A connection counts as already handled if it is part of A.

5.1.4.7 Handle parents

Algorithm 5.8 represents the process of finding the correct execution order for parent components. The inputs are the current parents P which need to be handled, the list of connections C from which P was extracted, the current snapshot S, a list of already traversed parts T, the list of actions

```
inputs : $P$ ... parents that will be used to find top-most parents,
         $S$ ... current snapshot,
         $T$ ... already traversed parts,
         $A$ ... list of actions representing the plan,
         $r$ ... root part of current iteration step
output: $A$ ... modified list of actions representing the plan
HandleParents ($P, S, T, A, r$)
    foreach parent $p \in P$ do
        $\hat{p} \leftarrow$ find top-most parent of $p$ that $\notin T$;
        $C \leftarrow$ connections where $parent = p$;
        foreach connection $c \in C$ do
            $\hat{c} \leftarrow c.child$;
            $D \leftarrow$ connections where $child = \hat{c}$;
            foreach connection $d \in D$ do
                if $d.parent = r$ then
                    $A \leftarrow$ GetExecutionOrder($\hat{p}, S, T, A, \hat{c}$);
    return $A$;
```

Algorithm 5.8: Handle parents for correct execution order

A representing the full plan and the current root part r for which the connections are currently being handled in Algorithm 5.3. The output is the expanded list of actions A.

The main idea of handling a parent connection is to find the top-most parent and then to recursively run the execution order algorithm for the top-most parent's sub-tree until it reaches the halting point. The two inner loops are used to find this halting point, defined as the meeting point of the main graph and the sub-branch.

For each parent p in P, the top-most parent $\hat{p}$ that is not in T is found. This will be the new root part for the next recursive call of *GetExecutionOrder*, defined in Algorithm 5.3). Next, all connections C where p is the parent component need to be found. Each child $\hat{c}$ of connection c in C needs to be identified and with it all connections D where $\hat{c}$ is the child component. For each connection d in D, if d is equal to the root component r, the correct halting point is found as $\hat{c}$ and the recursive execution can continue, starting a sub-branch search.

5.1.4.8 Handle children

```
inputs : $C$ ... children that will be root parts for recursive calls,
         $s$ ... current snapshot,
         $T$ ... already traversed parts,
         $A$ ... list of actions representing the plan,
         $h$ ... halting part for recursive parent handling
output: $A$ ... actions sorted by execution order

1 HandleChildren ($C, s, T, A, h$)
2     foreach child $c \in C$ do
3         if $c \notin T$ then
4             $A \leftarrow$ GetExecutionOrder($c, s, T, A, h$);
5     return $A$;
```

Algorithm 5.9: Handle children for correct execution order

Algorithm 5.9 describes how to continue the recursive execution order extraction for child parts of the root component of Algorithm 5.3. The inputs are the current children C which need to be handled, the current snapshot s, a list of already traversed parts T, the list of actions A representing the full plan and the current halting part h. The output is the expanded list of actions A.

For each child c in C, if c is not in T, the recursive call of *GetExecutionOrder*, defined in Algorithm 5.3), is run with c as the new root part.

If the current search is on the main branch with the assembly's grounded part as root component, h is $null$. Otherwise, h is the connecting component between the main and the sub-branch, where the sub-branch search is based on a different root component than the assembly's grounded part. The latter situation means, that Algorithm 5.8 was called and currently a subassembly is being handled.

5.1.4.9 Find differences between snapshots

```
inputs : $s$ ... current snapshot,
         $z$ ... previous snapshot
output: $s$ ... current snapshot with changes
1 FindDifferences ($s$, $z$)
2     foreach part $p \in s$ do
3         if $p \notin z$ then
4             foreach connection $c$ regarding $p$ do
5                 $c.changed \leftarrow true$;
6     foreach connection $c \in s$ do
7         if $c$ has different relations compared to equivalent $\in z$ then
8             $c.changed \leftarrow true$;
9     return $s$;
```

Algorithm 5.10: Find differences between snapshots

To find the differences between two snapshots, Algorithm 5.10 can be used. The inputs are the current snapshot s and the previous snapshot z. The output is the current snapshot s with changes marked inside the connections themselves. This algorithm manipulates the current snapshot by setting the property *changed* for each modified connection to *true*.

Two situations result in a connection being marked as *changed*. Some components might appear, while some connections might have changes in their relations.

For each part p in s, if p is new, meaning not in z, the property *changed* needs to be set to *true* for all connections of s that reference p as either parent or child.
For each connection c in s, if c references different relations than the corresponding connection in the previous snapshot z, *changed* should be set to *true* for c.

5.1.4.10 Identify affected connections

```
inputs : $E$ ... sorted execution order,
         $s$ ... current snapshot,
         $z$ ... previous snapshot
output:  $U$ ... affected connections in form of undos,
         $R$ ... affected connections in form of redos
IdentifyAffected $(E, s, z)$
    initialize $\hat{U}$, $\hat{R}$ lists of connections as $\varnothing$;
    initialize $U$, $R$ lists of lists of connections as $\varnothing$;
    foreach connection $c \in s$ do
        $d \leftarrow$ counterpart of $c \in z$;
        if $c.changed$ then
            if length of $\hat{R} > 0$ then
                append $\hat{R}$ to $R$, append $\hat{U}$ to $U$ and clear $\hat{U}$ and $\hat{R}$;
            append $c$ to $\hat{R}$;
            if $\exists\ d$ then append $d$ to $\hat{U}$;
            else append $\varnothing$ to $\hat{U}$;
        else if length of $\hat{R} > 0$ then
            if $\exists\ d$ then
                if max offset of $c >$ max offset of $d$ and $(c.parent = d.parent$ or $c.child = d.child$ or $c.parent = d.child$ or $c.child = d.parent)$ then
                    append $c$ to $\hat{R}$;
                    append $d$ to $\hat{U}$;
                else append $\hat{R}$ to $R$, append $\hat{U}$ to $U$ and clear $\hat{U}$ and $\hat{R}$;
            else append $c$ to $\hat{R}$, append $\varnothing$ to $\hat{U}$;
    return $U$,$R$;
```

Algorithm 5.11: Identify affected connections

After having identified all differences between snapshots and marked them as *changed* connections inside each snapshot, it is important to identify all connections that are affected by these changes. These connections represent all actions necessary in order to satisfy the changes. Algorithm 5.11 describes this procedure. The inputs are the sorted execution order E of connections, the current snapshot s and the previous snapshot z. The outputs are two lists of lists of connections separated in actions to be undone U and actions to be redone R.

The nesting of lists in a list is used to distinguish between related sequences of affected connections. Each element in U and R is a list of affected connections $\hat{U}$ and $\hat{R}$ respectively, which together symbolize all actions necessary to satisfy a individual change in s.
It is important to keep separate lists for undos $\hat{U}$ and redos $\hat{R}$, since actions that need to be undone are from the previous snapshot and redos are from the current snapshot. This means that the relations could be different and must be handled separately.
Empty lists of connections for undos $\hat{U}$ and redos $\hat{R}$ and need to be initialized, as well as empty lists of lists U and R.

For each connection c in s the property $changed$ of c can either be $true$ or $false$, which leads to the following two separations.

If c did change, meaning the property $changed$ is set to $true$, the following part of the algorithm is entered. If $\hat{R}$ contains any elements, then both $\hat{R}$ and $\hat{U}$ need to be appended to U and R respectively and $\hat{R}$ and $\hat{U}$ need to be cleared.
With now in any case empty lists $\hat{R}$ and $\hat{U}$, c is added to $\hat{R}$ and the corresponding counterpart in z is appended to $\hat{U}$, if it exists. Else, an empty element is added to $\hat{U}$. Without it, the lists of undos and redos have different lengths, which would need to be taken into consideration in this algorithm and in Algorithm 5.12.

Else, if c did not change but $\hat{R}$ contains any elements, a different part of the algorithm is entered.
If the counterpart of c does not exist in z, then c is appended to $\hat{R}$ and an empty element is added to $\hat{U}$. If it does exist, the following considerations are taken into account. If the maximal offset of the relations in c is greater then the maximal offset of the relations in d and either parent or child of c is equal to either parent or child of d, then c and d should be added to $\hat{R}$ and

$\hat{U}$ respectively. Else the current lists $\hat{R}$ and $\hat{U}$ are finished and need to be added to R and U respectively. Afterwards $\hat{R}$ and $\hat{U}$ need to be cleared.

5.1.4.11 Add affected connections

```
inputs : $U$ ... affected undos,
         $R$ ... affected redos,
         $\pi$ ... list of actions representing the plan until this point
output: $\pi$ ... list of actions representing the plan including changes
AddAffected ($U$, $R$, $\pi$)
    for $i \leftarrow 0$; $i <$ length of $R$; $i \leftarrow i + 1$ do
        $\hat{U} \leftarrow i^{th}$ element of $U$;
        $\hat{R} \leftarrow i^{th}$ element of $R$;
        append each connection $\in \hat{U}$ to $\pi$ in inverse order if $\neg \varnothing$;
        append each connection $\in \hat{R}$ to $\pi$;
    return $\pi$;
```

Algorithm 5.12: Add affected connections to plan

Algorithm 5.12 presents how to append the affected connections to the current plan. The inputs are the affected undos U, the affected redos R and a list of actions A representing the plan until the previous snapshot. The output is the modified list of actions A extended by the affected changes of the current snapshot.

For each list of list of connections U and R, each list of connections $\hat{U}$ and $\hat{R}$ is handled. First, all elements of $\hat{U}$ are added to A in inverse order. Then $\hat{R}$ is appended to A.

5.1.5 Plan interpretation

The resulting sequence of actions can then be executed by all different kinds of bin-picking-esque robotic setups, as long as the given skill sets of each robot allow it to pick up and place down all involved components.

The following interpretation is formulated around the assumption, that the system has two arms with generic two-finger grippers.

One of the major challenges for such an assembly cell is gravity. It forces some components to move after they are released by the robot, if they are not held in place by other components or do not fit seamlessly into their inserted slot. Also orientation is important and might need to be changed for parts at some point during the assembly. A more detailed description of a possible future expansion, in regard to building a system capable of understanding gravity and its effects, can be found in Chapter 8.

Within this project, the plan interpretation aims to mitigate this problem by having additional rules describing when to keep components held by an arm and when to release the part that is currently being held. During the general assembly, components needed to be held until there is a subsequent component being pressed against it or a screwing relation holding everything together.
That is not possible for subassembly steps, because one arm is holding the shaft, on which the other components stack. This means that in a subassembly every part is released immediately after it is placed at its final location, since the other arm is holding the subassembly parent while the current arm is needed for the next component. Placing a component at its final location limits the situations in which gravity affects the assembling process, but it does not prevent it. To prevent any movement of the placed component after releasing it, the system would need to understand how a specific set of components interact with each other while under the effect of gravity, which is part of future expansions in Chapter 8.

After screwing a component into place, both the screw and all fixed components can always be released.

Please note, that bold and italic commands in the following algorithms, including grasp, insert, screw, unscrew, shift and release, represent elementary skills. They are known to the robot and known to *SkiROS*. These skills are discussed in more detail in Section 5.1.6.1.

```
inputs : $O$ ... ontology containing the plan
1 Assemble ($O$)
2     decide on main arm $r_1$ and secondary arm $r_2$;
3     $\pi \leftarrow$ read plan of actions from $O$;
4     $z \leftarrow$ read user defined assembly location from $O$;
5     ExecutePlan($\pi$, $r_1$, $r_2$, $z$);
```

Algorithm 5.13: Assemble full sequence plan

5.1.5.1 Assemble

The interpretation of the plan is presented in Algorithm 5.13 and has only the Ontology O as input.

The workflow starts with the general decision about which is the main arm r_1 and which is the secondary arm r_2. If all arms are equal, the choice of main and secondary arm can be random, otherwise it might be important to choose the more skilled robot arm as the main one. Next, the plan π, given in form of a series of actions, is read from the ontology. *SkiROS* loads the ontology files when starting the system makes the knowledge available through the world model. The full plan can be found by looking for the object with the class type *plan* using predefined functions of the *SkiROS* framework to access the world model. Additionally, the pre-defined position of grounded component is requested from the world model. This position is defined by the developer in a specific ontology, together with the scene setup including the robots, and is further described in Section 6.2.6.
And finally, the plan gets executed (see Section 5.1.5.2).

5.1.5.2 Execute plan

The method of extraction of the entire plan can be seen in Algorithm 5.14. The inputs of the system are the plan of actions π, the main robotic arm r_1, the secondary robotic arm r_2 and the pre-defined position for the grounded component. It is important to keep track of three issues: which arm is

```
inputs : $\pi$ ... plan of actions to execute,
         $r_1$ ... main robot arm,
         $r_2$ ... secondary robot arm,
         $z$ ... pre-defined assembly location
ExecutePlan ($\pi, r_1, r_2$)
    initialize $r_{2,l}$ as $null$, $x_l$ as $null$, $l$ as $null$;
    Keep track of which arm holds what part in $H$;
    Keep track of the grounded parts $G$;
    Keep track of subassembly parts $S$;
    foreach $action\ a \in \pi$ do
        if $\neg\ l.undo$ and $a \neq last \in \pi$ then
            $n \leftarrow$ action after $a \in \pi$;
            if $a.undo$ and $\neg\ n.undo$ and $a.connection.parent =$
              $n.connection.parent$ and $a.connection.child =$
              $n.connection.child$ then
                $S, G, r_{2,l}, x_l \leftarrow$ ExecuteAction($a, l, S, G, r_1, r_2, r_{2,l}, x_l,$
                  $H, true$)
        if $a.undo$ or $\forall\ a.connection.relations \nexists\ relation.presatisfied$
          then
            $q \leftarrow a.parent$;
            if $q.grounded$ and $q \notin G$ then
                $G \leftarrow$ HandleGrounded($a, G, r_1, z$)
            $S, G, r_{2,l}, x_l \leftarrow$ ExecuteAction($a, l, S, G, r_1, r_2, r_{2,l}, x_l, H,$
              $false$);
        else
            $S, G, r_{2,l}, x_l \leftarrow$ ExecuteAction($a, l, S, G, r_1, r_2, r_{2,l}, x_l, H,$
              $true$)
        $l \leftarrow a$;
    go to home position with both arms $r_1$ and $r_2$;
```

Algorithm 5.14: Execute full plan

holding what part, all parts G connected to grounded components and all subassembly parts S.

For each action a in π the following aspects need to be considered. If a previous action l and a next action n exist for a, l and n are not an *undo*, but a is, and the parent and child of a and n are equal, this means that the current step is an *undo* which gets redone in the next action. In that specific case a is skipped and n is interpreted as a shift skill (see Section 5.1.5.3). For more information about each skill refer to Section 5.1.6.1.
If a is either *undo* or its connection does not contain any relation that is $presatisfied$, the action a gets executed (see Section 5.1.5.3), since also $presatisfied$ relations might need to be undone. Otherwise, the action gets skipped, since $presatisfied$ connections do not need to be executed (see Section 5.1.5.3). If the parent of a is the grounded component and is not in G, an action placing the grounded component needs to be run before executing a (see Section 5.1.5.4).

After all actions in π are executed, the assembly is finished. Both robot arms release the parts they are still holding and return to their home position.

5.1.5.3 Execute action

Algorithm 5.15 presents how to handle an action, which includes releasing components if necessary, identifying main and secondary arm for the current action and deciding whether to execute or to skip a skill. The inputs are the action a that is being executed, the action l previous to a, a list of current subassembly parts S, a list of grounded parts G, the main robot arm r_1 and the secondary robot arm r_2, a reference to a non-acting arm $r_{2,l}$ of the previous action, the component x_l grasped in the previous step and a binary value whether to *skip* action a or not. The outputs of the algorithm are the list of grounded parts G expanded by the placed part, the list of current subassembly parts S, the non-acting robot arm $r_{2,a}$ of the current action execution and the component x that was grasped during this iteration.

A list M is used to keep track of skipped components with unsupported multi-contact and must be initialized as empty. First, the part x that needs to be grasped and the part y that needs to be held are found (see Section 5.1.5.5). Then the previous action execution is checked on whether some components can be released (see Section 5.1.5.6). It is important to then identify the acting and the non-acting arm. The acting arm will be in contact

```
inputs : $a$ ... action to execute,
         $l$ ... previous action,
         $S$ ... list of current subassembly parts,
         $G$ ... List of grounded parts,
         $r_1$ ... main robot arm,
         $r_2$ ... secondary robot arm,
         $r_{2,l}$ ... previous non-acting arm,
         $x_l$ ... previous component to grasp,
         $H$ ... which arm is holding what part,
         $skip$ ... whether to skip action $a$ or not
output:  $S$ ... modified list current of subassembly parts,
         $G$ ... modified list of grounded parts,
         $r_{2,a}$ ... current non-acting robot arm,
         $x$ ... component to grasp
ExecuteAction ($a, l, S, G, r_1, r_2, r_{2,l}, x_l, H, skip$)
    initialize $M$ list of components as $\varnothing$;
    $x, y, G, S \leftarrow$ GetParentAndChild($a, G, S$);
    CheckReleasePrevious($a, l, S, r_1, r_2, x_l$);
    $r_{1,a}, r_{2,a} \leftarrow$ IdentifyArms($x, y, r_1, r_2, H$);
    if $\neg$ $skip$ then
        $c \leftarrow a.connection$;
        $C \leftarrow c.relatedConnections$;
        if $C = \varnothing$ and $c.parent \notin M$ and $c.child \notin M$ then
            ExecuteSkill($a, l, G, x, y, r_{1,a}, r_{2,a}, H$);
        else
            $m \leftarrow$ find common $child$ of $C$;
            if $m \notin M$ then append $m$ to $M$;
        CheckReleaseCurrent($a, l, S, r_{2,l}$);
    return $S, G, r_{2,a}, x$;
```

Algorithm 5.15: Execute individual action

with x and the non-acting arm will be holding y.
If a should not be skipped, then the necessary skills for the execution of a are started consecutively (see Section 5.1.5.8). If the connection of a has

related connections, then the execution is also skipped, since multi-contact relations are not handled yet and are part of future improvements (see Chapter 8). Following actions could depend on such a skipped component, which could lead to failure during the rest of the assembly. The implemented workaround for this issue consists of finding the common child of all related connections of a and adding them to a list M. This adds the condition, that neither parent nor child of any connection in an action can be in M, for the action to be executed. Additional workarounds for this issue, except for adding skills to support multi-contact actions, include removing multi-contact relations from the *CAD* model or stopping the assembly process after the first appearance of a multi-contact action.

The final step is checking if the execution of a allows letting go with any arm (see Section 5.1.5.6).

Please note that the *BDU* example uses the rubber band only as a child component, meaning that there are no following actions adding components that dependent on the rubber band. The only following action is shifting the position of a subassembly in order to create pressure on the rubber band. If there is no rubber band, this action can still be executed, but the assembled *BDU* will not function properly without the rubber band in place.

5.1.5.4 Handle grounded part

Handling the grounded component of the assembly means executing a series of skills to place the component on a pre-defined location, as described in Algorithm 5.16. The inputs consist of the action a that is being executed, a list of grounded components G, a robot arm R and the pre-defined position for the grounded component. It can be any robotic arm of the system that is empty. The output of the algorithm is the expanded list of grounded components G.

Initially, parent P of the connection in a is added to G. If placing the grounded component is needed, P must be grasped by arm R and then inserted onto the pre-defined assembly location. Afterwards, R should release of part P.

```
inputs : $a$ ... action to execute,
         $G$ ... list of grounded parts,
         $r$ ... robot arm,
         $z$ ... pre-defined assembly location
output:  $G$ ... modified list of grounded parts

HandleGrounded ($a, G, r, z$)
    $p \leftarrow a.connection.parent$;
    append $p$ to $G$;
    if placing grounded component is needed then
        Grasp $p$ with $r$;
        Insert $p$ onto $z$ using $r$;
        Release with $r$;
    return $G$;
```

Algorithm 5.16: Handle grounded component during assembly interpretation

Placing the grounded component is not necessary for this work, since the simulation starts with the grounded component already at the pre-defined assembly location.

5.1.5.5 Get parent and child components

Algorithm 5.17 describes how to identify whether the parent or the child of an action is the part that needs to be grasped and actively moved towards the other component. The inputs are the Action a that is being executed, a list of grounded parts G and a list of the current subassembly parts S. The outputs of the algorithm are the component x that needs to be grasped, the component y to which x should be attach to, the extended list of grounded parts G and the modified list of current subassembly parts S.

To identify which part needs to be actively moved and which part needs to be held in place or is already grounded, the indices p_I and p_I in G for both the parent p and the child c of the connection in a need to be found.
If p can be identified in G and either c can not be located or c_I is greater than p_I, the child c of a must be actively moved towards the parent. That

```
inputs : $a$ ... action to execute,
         $G$ ... list of grounded parts,
         $S$ ... list of current subassembly parts
output:  $x$ ... component to grasp,
         $y$ ... component to attach to,
         $G$ ... modified list of grounded parts,
         $S$ ... modified list of current subassembly parts
GetParentAndChild ($a, G, S$)
    $p \leftarrow a.connection.parent, c \leftarrow a.connection.child;$
    $p_I \leftarrow$ index of $p \in G$, $c_I \leftarrow$ index of $c \in G$;
    if $\exists\, p_I$ and ($\nexists\, c_I$ or $p_I < c_I$) then
        $x \leftarrow c, y \leftarrow p;$
    else if $\exists\, c_I$ and ($\nexists\, p_I$ or $c_I < p_I$) then
        $x \leftarrow p, y \leftarrow c;$
    else
        if $p \notin S$ then append $p$ to $S$;
        if $c \notin S$ then append $c$ to $S$;
        return $c, p, G, S$
    append $x$ to $G$;
    if $S \neq \varnothing$ then append $S$ to $G$ and clear $S$;
    return $x, y, G, S$;
```

Algorithm 5.17: Get parent and child component from action

means x is assigned to be c and y is assigned to be p.
Else, if vice versa c can be found in G and either p can not be found or p_I is greater than c_I, x becomes p and y becomes c.
Else, if neither p nor c can be found in G, the current component is part of a subassembly. In that case, both p and c, if they are not found in G, are respectively appended to S. Then the algorithm returns with x equal to c and y equal to p.

In the case that p or c are in G, the component assigned to x is appended to G. If S is not empty, which means there was an subassembly that ended up connecting to a grounded part, all components in S are appended to G and S is cleared.

5.1.5.6 Check if arms should release because of previous action

```
inputs : $a$ ... action to execute,
         $l$ ... previous action,
         $S$ ... list of current subassembly parts,
         $r_1$ ... main robot arm,
         $r_2$ ... secondary robot arm,
         $x_l$ ... previous component to grasp
1 CheckReleasePrevious ($a, l, S, r_1, r_2, x_l$)
2     if $\forall$ $l.connection.relations$ $\exists$ $relation.screw$ then
3         if $\forall$ $a.connection.relations$ $\nexists$ $relation.screw$ or ($a.undo$ and $S \neq \varnothing$) then
4             Release with both arms $r_1$ and $r_2$;
5     else if $S \neq \varnothing$ and $\exists$ $x_l$ and $a.parent \neq x_l$ then
6         if $r_1$ contains $x_l$ then Release with arm $r_1$;
7         else if $r_2$ contains $x_l$ then Release with arm $r_2$;
```

Algorithm 5.18: Check if any arm should release from previous action

The procedure to check if there is any component from a previous action, that can be released, is presented in Algorithm 5.18. There are no outputs and the inputs are an action a that is being executed, the action l previous to a, a list of current subassembly parts S, the main robot arm r_1, the secondary robot arm r_2 and the component x_l that was grasped in the previous action.

If the connection of the previous action l contains any relation that is *screw* and if the connection of the current action a does not contain any relation that is *screw*, or a is *undo* and S is not empty, then both arms r_1 and r_2 can release what they are holding. This means a independent group of actions is started since in an assembly all series of actions finish with screws.

Consecutive screw actions do not allow to release parts, the only exception is during a subassembly, which is when S is not empty.
Else, if l does not contain any relation that is *screw*, then the following part of the algorithm is entered. If S is not empty, meaning the current actions are part of a subassembly, and x_l exists and is not the parent of a, then the arm that contains x_l should release it, since the part is not needed for a.

5.1.5.7 Identify acting and supporting arm

```
inputs : $x$ ... component to grasp,
         $y$ ... component to attach to,
         $r_1$ ... main robot arm,
         $r_2$ ... secondary robot arm,
         $H$ ... which arm is holding what part
output:  $r_{1,a}$ ... acting robot arm,
         $r_{2,a}$ ... non-acting robot arm
IdentifyArms ($x$, $y$ $r_1$, $r_2$)
    if any arm is holding $x$ then
        $r_{1,a} \leftarrow$ arm that is holding $x$;
    else
        if any arm is empty then
            $r_{1,a} \leftarrow$ arm that is empty;
        else
            if any arm is holding a part that is not in collision with $x$ then
                $r_{1,a} \leftarrow$ arm that is holding the with $x$ colliding part;
    if any arm is holding $y$ then $r_{2,a} \leftarrow$ arm that is holding $y$;
    else $r_{2,a} \leftarrow$ arm $\neq r_{1,a}$;
    return $r_{1,a}$, $r_{2,a}$;
```
Algorithm 5.19: Identify acting and supporting arm

Depending on which arm is already holding what component, a acting arm and a non-acting arm are identified using Algorithm 5.19. The inputs are the component x that needs to be grasped, the component y to which x should be attach to, the main robot arm r_1 and the secondary robot arm r_2. The outputs of the algorithm are the acting robot arm $r_{1,a}$ and the non-acting robot arm $r_{2,a}$. $r_{1,a}$ is executing the action, whereas $r_{2,a}$ is supporting the action by holding components that need to be held in place during execution.

If any arm is holding x, is empty or is holding a part that is not in collision with x, that arm is assigned to be $r_{1,a}$. If no arm is holding y, $r_{2,a}$ is an empty arm that is not $r_{1,a}$. Else, $r_{2,a}$ is the arm that is holding y.

5.1.5.8 Execute skill

Algorithm 5.20 presents the execution of individual skills of the system. The inputs are the action a that is being executed, the action l previous to a, a list of grounded parts G, the component x that needs to be grasped, the component y to which x should be attach to, the acting robot arm $r_{1,a}$ and the non-acting robot arm $r_{2,a}$. There is no output, since an elementary skill is repeated until success or else the execution of the whole assembly fails.

The major differentiation is initially done by investigating if a needs to be undone.

On one hand, if a is *undo* but its connection does not contain any relation that is *screw*, the action is skipped, since only screws need to be undone. On the other hand, if a is *undo* and its connection contains any relation that is *screw*, then x should be grasped by $r_{1,a}$, if the arm does not already hold the part. Also, y should be grasped by $r_{2,a}$, if the arm does not already hold the part and if y is not already in the set of grounded parts G. Afterwards, x should be unscrewed from y using $r_{1,a}$.

Else, if a is not *undo*, then both x and y need to be grasped, depending on the same conditions as previously mentioned. x should be grasped by $r_{1,a}$, if the arm does not already hold the part. Also, y should be grasped by $r_{2,a}$, if the arm does not already hold the part and if y is not already in the set of grounded parts G. If a contains any relation that is *screw*, then x should be screwed into y using $r_{1,a}$. Else, if l is *undo*, then x should be shifted within y using $r_{1,a}$. Else, x should be inserted into y using $r_{1,a}$.

```
inputs : $a$ ... action to execute,
         $l$ ... previous action,
         $G$ ... list of grounded parts,
         $x$ ... component to grasp,
         $y$ ... component to attach to,
         $r_{1,a}$ ... acting robot arm,
         $r_{2,a}$ ... non-acting robot arm,
         $H$ ... which arm is holding what part
1  ExecuteSkill ($a, l, G, x, y, r_{1,a}, r_{2,a}, H$)
2      if $a.undo$ then
3          if $\forall$ $a.connection.relations$ $\exists$ $relation.screw$ then
4              if $r_{1,a}$ does not hold $x$ then
5                  Grasp $x$ with $r_{1,a}$;
6              if $r_{2,a}$ does not hold $y$ and $y \notin G$ then
7                  Grasp $y$ with $r_{2,a}$;
8              Unscrew $x$ from $y$ with $r_{1,a}$;
9          else skip, since disassembly is not needed;
10     else
11         if $r_{1,a}$ does not hold $x$ then
12             Grasp $x$ with $r_{1,a}$;
13         if $r_{2,a}$ does not hold $y$ and $y \notin G$ then
14             Grasp $y$ with $r_{2,a}$;
15         if $\forall$ $a.connection.relations$ $\exists$ $relation.screw$ then
16             Screw $x$ into $y$ using $r_{1,a}$;
17         else if $l.undo$ then Shift $x$ within $y$ using $r_{1,a}$;
18         else Insert $x$ into $y$ using $r_{1,a}$;
```

Algorithm 5.20: Execute individual skill or set of skills

5.1.5.9 Check if arms should release because of current action

Algorithm 5.21 presents a method of checking if there is any component to release. The inputs are an action a that was just executed, the action l

```
inputs : $a$ ... action to execute,
         $l$ ... previous action,
         $S$ ... list of current subassembly parts,
         $r_{2,l}$ ... previous non-acting arm
1 CheckReleaseCurrent ($a, l, S, r_{2,l}$)
2     if no arm is empty and $S = \varnothing$ and $\neg$ a.undo and $\neg$ l.undo and $\forall$
        l.connection.relations $\nexists$ relation.screw then
3         Release with $r_{2,l}$;
```

Algorithm 5.21: Check if any arm should release from current action

previous to a, a list of current subassembly parts S and the non-acting arm $r_{2,l}$ of the previous action l. This algorithm does not have outputs, since the executed skill is repeated until success or else the execution of the whole assembly fails.

The next action execution needs an empty arm. So if no arm is currently empty, but S is empty, the connection of l does not contain any relation that is *screw* and neither a nor l are *undo*, then the previous non-acting arm $r_{2,l}$ should release the component it is holding, since the previous action was not part of a subassembly and was only stacking components consecutively face-to-face.

5.1.6 Skill execution

5.1.6.1 Skills

Skills are modular, hierarchical combinations or sequences of primitives. The following skills are implemented in Python building upon the basic structure presented in [3], as mentioned in Section 2.5 describing SkiROS.

The grasp skill is essential to most other skills, since components need to be grasped before for example inserting them or screwing them into place. This dependency is depicted in the following graphs by the first step being in a less dark color then the other steps. It is important to note, that sometimes a component is already grasped, because a previous action needed it to be.

Grasp This skill is is supposed to grasp a part. It can either be a component, that was not yet grasped and is still in the kitting box, or an already placed object. It is possible, that the performing arm is already holding something, which is no longer needed. Because of that, every grasp action starts with opening the gripper first. This is always safe, since the system has determined that the arm should grab a component and for that it needs to be empty. The second step is to move to a pose, which is assumed to be not in collision with the object and in line with the grasp axis. The calculation of this pose is described alongside the primitives in Section 5.1.6.2. From here on, the gripper can reach the final grasp pose with the correct orientation by keeping its current orientation constant. Lastly, the gripper closes after reaching the final destination. See Figure 5.12 for a representational graph.

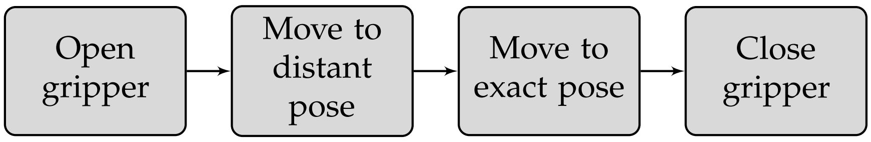


Figure 5.12: Primitive structure of the grasp skill.

Insert The insertion of a component into another along a specific axis is similar to the grasp skill, with the only difference, that the gripper should neither be opened nor closed, since it is assumed that the correct component is already grasped. This leaves reaching the distant pose outside the bounding box and then moving towards the final location as the two needed steps for the insert skill (see Figure 5.13).

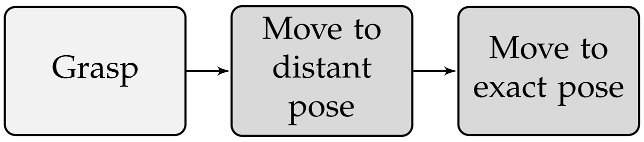


Figure 5.13: Primitive structure of the insert skill with a prior grasp skill call.

Screw Screwing some component into place is similar to inserting it into another component under the assumption that the gripper is holding the screw, with the major difference, that while reaching the exact pose, the gripper is rotating clockwise around its own axis (see Figure 5.14).

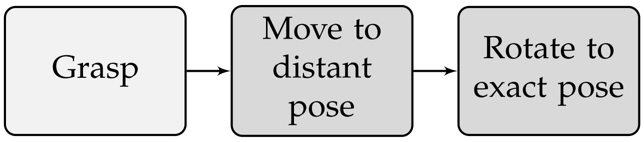


Figure 5.14: Primitive structure of the screw skill with a prior grasp skill call.

Unscrew Unscrewing a component means loosening a screw, so that for example a previously fixed component can be moved inside a slot. To achieve this, the component needs to be grasped and then rotated counterclockwise for a single turn (see Figure 5.15), while its screw-counterpart is also being held. Since no components can be removed from the assembly, as dictated by the presented guidelines, it is safe to loosen the screw by a single rotation, which should make components loose enough for example to be shifted inside slots. Without the inclusion of the information about the depth of the nut's hole or the length of the screw's shaft, it is unsafe to unscrew further, since that could completely undo the screw.

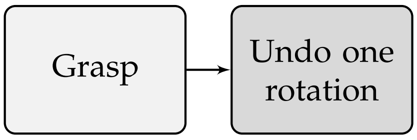


Figure 5.15: Primitive structure of the unscrew skill with a prior grasp skill call.

Shift To shift a component after its counterpart was unscrewed, means that the component is already being held by a gripper and is now loose. The grasped component is then moved to the new exact location (see Figure 5.16). The previously unscrewed component would be screwed in place again afterwards.

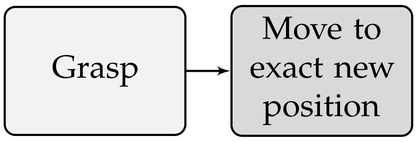


Figure 5.16: Primitive structure of the shift skill with a prior grasp skill call.

Release Sometimes it is important to release components at a specific point in time during the assembly, as described in the introduction of Section 5.1.5. Using this skill will make an arm open its gripper and then navigate to a predefined home position (see Figure 5.17). A unused, empty arm can block access to target locations of other arms, therefore navigating to the home position is added, so that the empty arm moves out of the way.

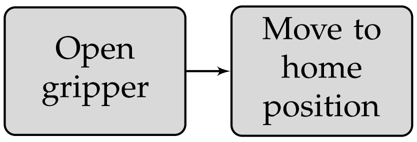


Figure 5.17: Primitive structure of the release skill.

5.1.6.2 Primitives

Move arm This primitive is responsible for the arm movement and represents the bridge between SkiROS and the motion planning and execution in *MoveIt!*. There are two different methods of moving an arm to a target: (1) exact and (2) distant. The used method is decided by a parameter when calling the primitive. An additional parameter allows constraining the gripper's pointing-direction to the current one, making it possible to approach the current goal while keeping the same orientation as reached by a previous move call.
The exact pose is given by the plan, whereas the distant pose is calculated. The exact pose's orientation is also used as distant orientation and the distant position is calculated by intersecting the grasp pose direction vector with a marginally bigger bounding box as the one described in Section 5.1.3, since it is important to assure being outside the component. The distant position of the final pose is given by this intersection, but the orientation

to reach is the inverse of the grasp direction used for the intersection calculation. A formal definition and a visual representation can be found in Section 7.3.
Lastly, a differentiation between a call started from a grasp skill or a insert skill must be given. When grasping a part, only the component's transformation is relevant. For an insertion of one component into another, both actual component's transformations and the requested transformation between the two parts, denoted in current action that is being executed, are relevant for the final pose of the gripper and therefore of the component being inserted. A possible expansion for the heuristic of finding the distant pose could be the consideration of the bounding boxes of components that are attached to the part that is already grasped and currently being inserted. This is not necessary, since the component's shapes are known to the *MoveIt!* planning scene and taken into consideration when trying to find a collision free trajectory.

Inside the simulation an additional differentiation between grasp and insert is relevant for linking or unlinking all assembly components with a collision with the grasped part. More on this topic can be found in Section 6.2.7.

Move gripper This primitive controls the grippers *MoveIt!* move group, allowing the system to theoretically move the arm and open the gripper concurrently. This primitive has only one parameter, which is to open or close the gripper. Furthermore, for the simulation it is important to distinguishing between grasping and inserting. Initially all components are attached to the empty space of Gazebo, the so called world frame. When grasping a component, it needs to be detached from the world and attached to the grippers frame. In contrast, when a component is inserted, it needs to be detached from the gripper and attached to the world frame.

5.2 Initial scene

Since this project is implemented in the context of a simulation, it is important to be able to model the initial state of the scene in Inventor and export it, so that it does not need to be set up manually for *ROS* and Gazebo. For

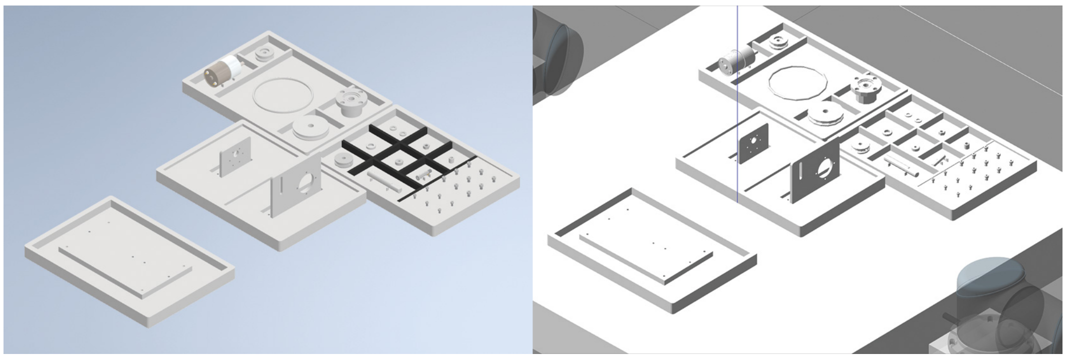


Figure 5.18: Example setup of a possible initial scene, in this case for the *BDU*. The left image is from Inventor, the right one is from Gazebo after spawning the components.

this, an additional mode in the Inventor plugin is implemented, which is further described in Section 6.2.3.

5.2.1 Set of rules

Generally, constraints or joints between components in the initial scene in the *CAD* file are interpreted as pre-satisfied relations. All grounded parts of the initial scene are assumed to be kitting trays, fixtures or generally starting containers for components. All relations between assembly components and grounded parts will be ignored. This way it is possible to use joints or constraints to place each individual assembly part in the kitting tray, without those relations being interpreted as pre-satisfied. The final poses will be used to later spawn the initial scene.

Same as in the assembly *CAD* model, the three axes X, Y and Z correspond to the simulation with the Z axis pointing along the inverse direction of gravity. A screenshot of an example setup inside Inventor can be found in Figure 5.18.

5.2.2 Scene analysis

An Inventor Assembly (*IAS*) file contains references to all used components in form of existing Inventor Part (*IPT*) files. The global pose of each component inside the *IAS* file is used for the placement parameters defined in the *ROS launch* file containing all parametrized spawning commands for each part for Gazebo and *ROS*. In addition, inertial parameters, mass and the center of mass in regard to the user defined grasp pose are exported from the individual *IPT* file and become relevant to the upcoming file generation, described in the next section.

Lastly, previously identified pre-satisfied relations are used to generate links between already combined components via *ROS* and the fake perception system for Gazebo described in Section 6.2.7.

5.2.3 Model exportation

IPT files can be exported to *STL* files, standing for *stereolithography*, using the Inventor *API*. This file type encodes geometrical information about the raw surface of a model in form of detailed triangle meshes. It is commonly used for 3D printing and is, together with the Collaborative Design Activity (*COLLADA*) file format, one of the only two possible file types compatible with Gazebo and *ROS* Kinetic Kame. *COLLADA* files have the additional capability of describing the material which is used. That is not possible for *STL* files, but the conversion to *COLLADA* files is not supported by Inventor and visual textures are not relevant to a successful execution of the simulation.

To complete preparations for spawning individual components in Gazebo, the exported model files need to be aggregated into a *SDF* file each, containing all needed information about each model as mentioned in Section 2.2. Each *SDF* model can be spawned multiple times to create many instances of the same model at different locations or with different orientations.

Finally, all generated *SDF* files must be referenced inside a *ROS launch* file to be loaded and instantiated into individual parts. Additionally, all pre-satisfied relations lead to linking of components, which needs to be called after everything has spawned. For this, two *launch* files are generated, which need to be placed, together with the *STL* and the *SDF* files, in the corresponding folders of the *ROS* package implemented during this project. More information about the general setup of the actual simulation can be found in Section 6.2.1.

6 Implementation Details

The following chapter summarizes implementation details of the approach presented in this thesis and is split into two main parts. The first part deals with deriving a plan while the second part describes executing it inside a simulation. Figure 6.1 shows an overview of all involved and developed components, that are necessary for the presented workflow.

6.1 Plan derivation

The first part of this chapter is is focusing on the plan extraction aspect of the proposed approach. This involves all preparations and steps needed to successfully export a sequence of actions representing the assembly in form of an ontology. Not only are ontologies the necessary file format for defining the inputs for *SkiROS*, they will also allow for reasoning to be applied in future expansions to further automate the skill execution.

6.1.1 Setup

To derive a plan from a CAD model, previously prepared by using the guidelines presented in in Section 5.1.2, AutoDesk Inventor and Microsoft DotNet (*.NET*) Framework are used to create a plugin that is able to export the ontology.

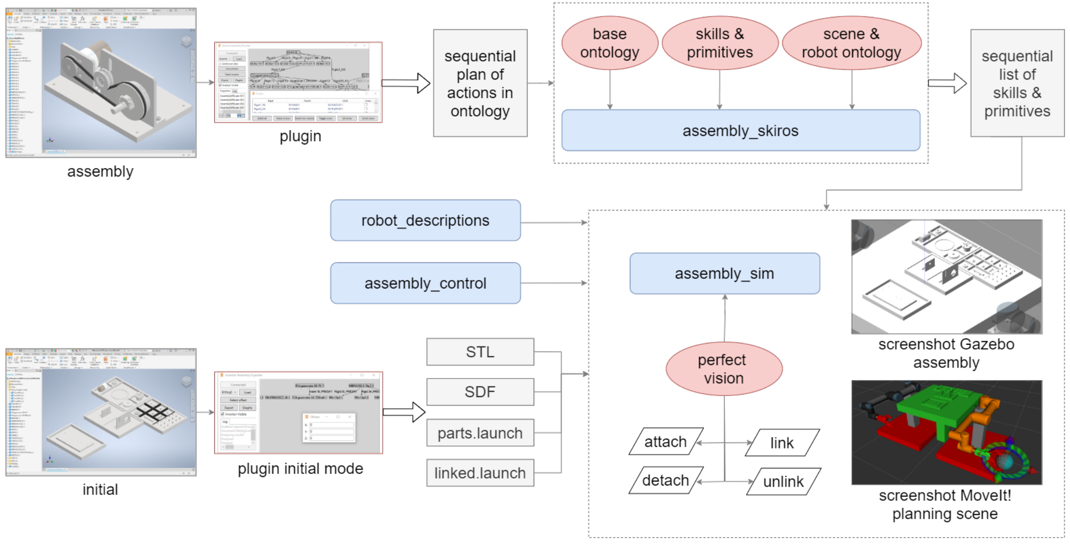
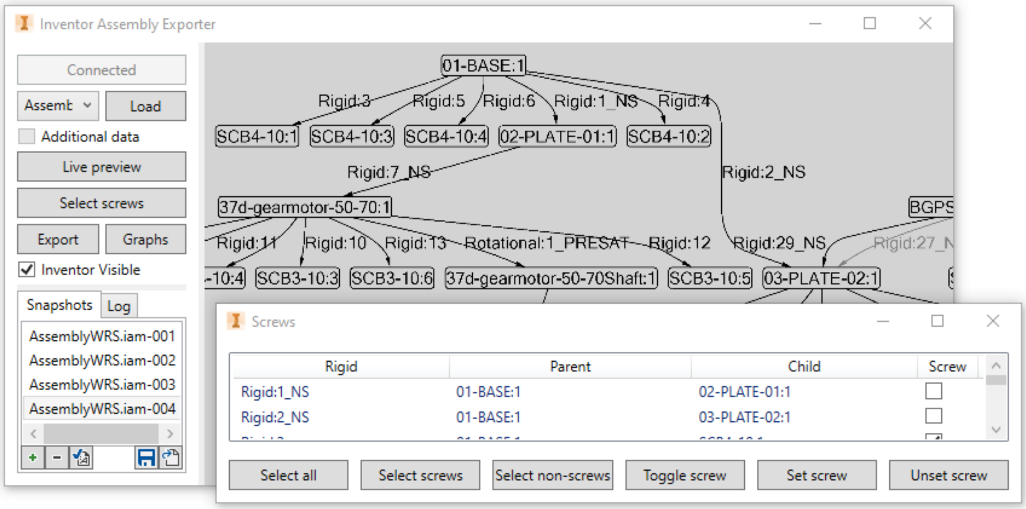


Figure 6.1: Implementation details of this work including all created *ROS* packages in blue, all developed plugins in red, the created services as parallelogram, generated components in form of gray boxes and the two input files in form of assembly *CAD* file and initial scene *CAD* file.

6.1.2 Inventor Plugin

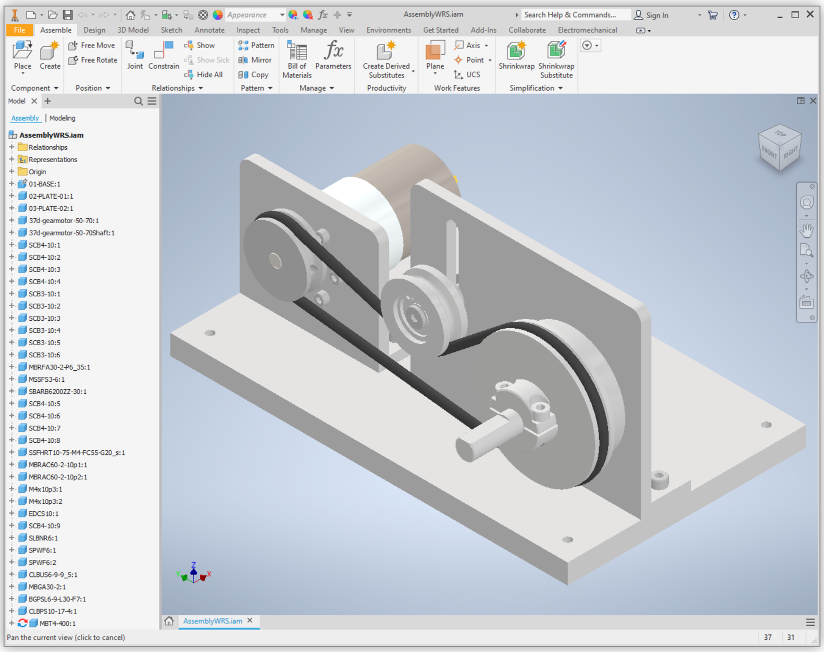
The plugin was created in C# and accesses the necessary information using the Inventor *API* as described in Section 2.9.1 and Section 5.1.3. To create a visual representation of an individual assembly snapshot as a graph, the Microsoft Automatic Graph Layout[1] (*MSAGL*) library was used. All exported files are generated by the plugin itself, except for the *STL* files, which are generated using the Inventor *API*. The internal data structure is almost identical to the class diagram in Figure 5.11, it is only missing the plan and the actions, but contains all snapshots, relations, components and collisions. The algorithms in use when generating and exporting the plan with the plugin are described in Section 5.1.4. Figure 6.2 shows the Inventor plugin analyzing the *BDU* example. It contains a screenshot of the plugin

[1] https://www.microsoft.com/en-us/research/project/microsoft-automatic-graph-layout/



Rigid	Parent	Child	Screw
Rigid:1_NS	01-BASE:1	02-PLATE-01:1	☐
Rigid:2_NS	01-BASE:1	03-PLATE-02:1	☐

(a) Screenshot of the Inventor plugin with highlighted *GUI* representing the *BDU* example.



(b) Screenshot of the Inventor window of the *BDU* example.

Figure 6.2: Screenshots of the Inventor plugin with highlighted *GUI* elements and the corresponding Inventor window representing the *BDU* example.

itself in Figure 6.2a and a screenshot of the corresponding Inventor window in Figure 6.2b.

After starting the plugin, any of the currently opened documents of the latest Inventor instance can be selected or a model can be opened via the plugin itself (see 1 in Figure 6.2a). If the plugin is run without Inventor already opened in the background, a new instance of the *CAD* program is run. This Inventor window can be hidden by setting the according checkbox (see 2 in Figure 6.2a).

For each snapshot that is added using the plugin (see 3 in Figure 6.2a), the entire assembly is analyzed, including all parts, joints and conditions. Then grasp poses, bounding boxes and collisions are extracted for each component using the Inventor API as described in Section 5.1.3. Relations are created from the resulting data and combined into connections, for which the relative parent-child transformation is calculated. Lastly, the plugin creates a directed graph from the given data and visualizes it (see 4 in Figure 6.2a).

After adding a snapshot, changes can be made to the *CAD* file if needed, like making components of future steps visible or replacing relations of moving parts. Then the next snapshot can be added. This cycle can be repeated as often as needed. When finished, the plan can be generated by exporting it (see 5 in Figure 6.2a). The only supported export format currently is *OWL*. The ontology file creation is done without any additional library by iterating over all aspects that need to be exported and writing predefined structures filled with the corresponding values of each individual. The encoded structure in the ontology file is represented by Figure 5.11. Alternatively, each individual graph can be exported (see 6 in Figure 6.2a).

It is possible to save or load the created snapshots using the plugin (see 7 in Figure 6.2a), which is implemented by serialization and deserialization of the full data structure. The editing capability of such an assembly file and its snapshot is limited to the necessary functionality of deleting selected snapshots and adding new ones. A possible future expansion could be the capability of editing individual aspects of single snapshots or changing the order of snapshots. A live preview of the assembly graph is also available (see 8 in Figure 6.2a).

Before exporting, all relations that represent screws inside the assembly need to be set accordingly. This was either already done inside the *CAD* file by adding the suffix *_NS* to a relation or can be done before exporting the plan using the plugin. The button indicated with the number 9 in Figure 6.2a opens up the dialog also shown in the same image, which allows to select or deselect all rigid joints as screws.

6.2 Simulation

The second part of this chapter describes all the necessary steps to be able to execute the plan inside the given simulation with the presented interpreter.

6.2.1 Setup

A GitHub project[2] contains all components for the plan execution setup in form of a united catkinised *ROS* package and some installation scripts. To prepare Ubuntu 16.04 for the use with the simulation, a single installation script is presented inside the root folder of the project. It installs all dependencies, such as general *ROS* packages, a python libraries for quaternions, the Gazebo simulator including all necessary *ROS* packages and files for the UR5, *SkiROS* and *MoveIt!*. It also sets up the environment variables, so that the current *catkin*[3] workspace can be found. To compile a *catkin* workspace, the command *catkin_make* must be executed inside folder *catkin_ws*. This makes the in Section 6.2.2 presented *ROS* packages available to the system.

After a successful setup and compilation of the workspace, additional information, such as initial pose, mass and inertial parameters, for the simulation regarding the initial scene must be extracted as described in Section 6.2.3.

[2]https://github.com/maestrini/Thesis
[3]http://wiki.ros.org/catkin

The simulation can be executed by launching the *simulation.launch* file found inside the *assembly_sim ROS* package.

6.2.2 ROS packages

The following packages have been developed to individually cover the robot control, the simulation process, the necessary additions for *SkiROS* and the general robot description for *ROS*.

assembly_control This metapackage[4] contains the *ROS* packages necessary to control the dual-arm robotic setup, presented in Section 6.2.4. The main package inside is *ur5_rg6_double_moveit_config*, which describes the configuration of the dual-arm robot. Additionally, packages for UR5[5] control inside the simulation and drivers for non-simulated UR5 arms[6] are collected. Finally, to solve an issue with closed loop joint chains in *URDF* files described in Section 6.2.4, a package for correct joint handling[7] is included.

assembly_sim This *ROS* package contains the simulation and all its components, including the Gazebo world, the 3D mesh and model files, the perception system described in Section 6.2.7, *ROS* service definitions and *launch* files. The *launch* file that connects all presented packages is called *simulation* and is the only file that needs to be launched to start the simulation.

assembly_skiros This metapackage contains two *ROS* packages. One is the implementation of the second version of *SkiROS*. The other is also named *assembly_skiros* and contains the pre-developed skills, ontologies and *launch* files to run *SkiROS* with the correct parameters and inputs together with the ontology file exported using the plugin.

[4]http://wiki.ros.org/Metapackages
[5]https://github.com/ros-industrial/universal_robot
[6]https://github.com/ros-industrial/ur_modern_driver
[7]https://github.com/roboticsgroup/roboticsgroup_gazebo_plugins

robot_descriptions This package contains the information necessary for the individual components of the dual-arm robot, such as configurations, controllers, descriptions, models and *launch* files.

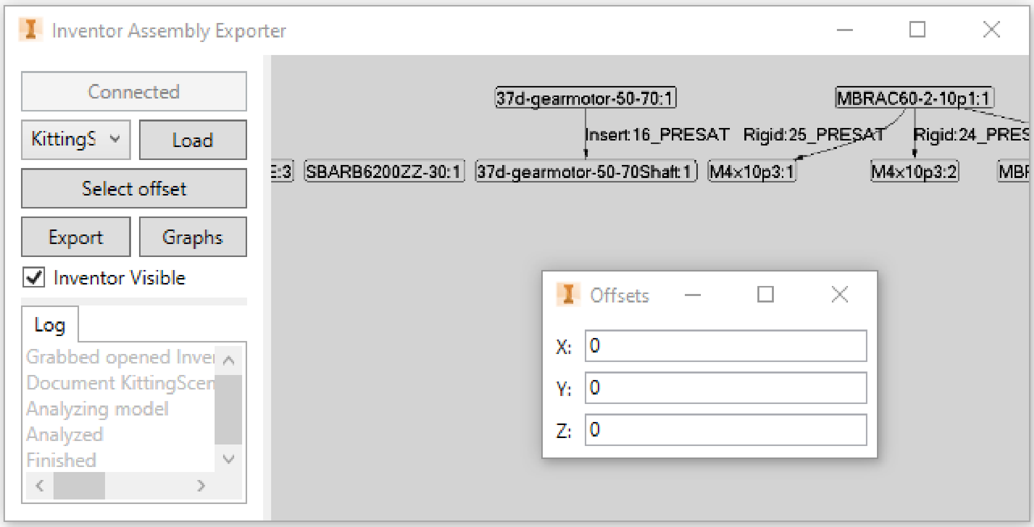
6.2.3 Inventor plugin for initial scene export

During execution it is possible to switch the plugin into *initial scene* mode. This mode no longer enables to create snapshots, but was developed to allow the exportation of the assemblies initial situation from a *CAD* file, as described in Section 5.2. Guidelines for the preparation of this model are presented in Section 5.2.1. Figure 6.3 shows the Inventor plugin analyzing the initial scene of the *BDU* example. It contains a screenshot of the plugin itself in Figure 6.3a and a screenshot of the corresponding Inventor window in Figure 6.3b.

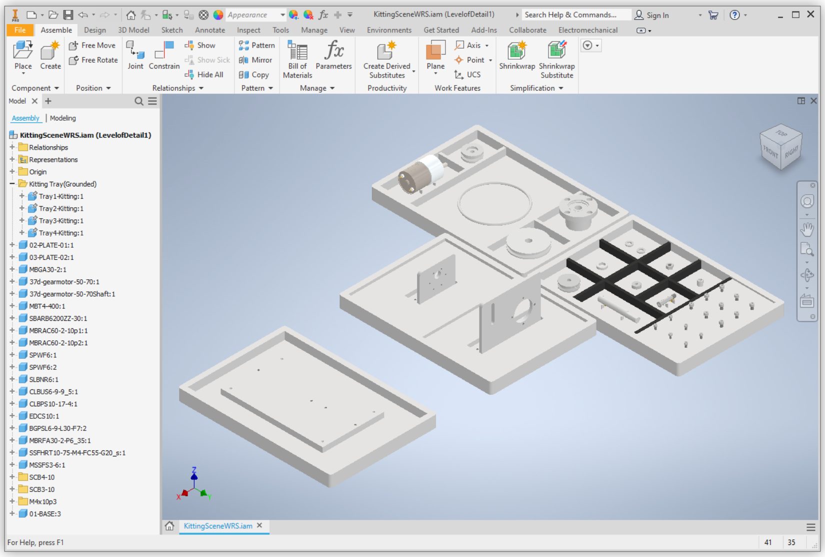
During the secondary mode of the plugin, a Inventor document can be selected (see 1 in Figure 6.3a). The plugin immediately analyzes the scene and presents the corresponding graph (see 2 in Figure 6.3a). Only components with pre-satisfied relations should have connections in the graph, every other part should be unconnected. Grounded components are ignored, since they are assumed to be part of the kitting scene. The kitting scene is part of the initial scene and contains all components from the initial scene that are not part of the final product that is to be assembled.

As described in Section 5.2.3, the accumulated information can be exported to *ROS launch* files containing all spawn commands for each component (see 3 in Figure 6.3a). This includes the importation of each component type using its *SDF* file, the spawning of each instance and the linking of pre-satisfied parts. The two files *parts.launch* and *link_parts.launch* need to be placed inside the *launch* folder of the *assembly_sim* package.

The necessary *SDF* files and the corresponding *STL* files can also be exported using the plugin by changing the file type (see 3 in Figure 6.3a). As mentioned in Section 5.2.3, the *STL* files can be converted from the initial *IPT* files using the Inventor *API*. The *SDF* files can not be extracted automatically and are therefore written to a file without any additional library by having a basic template filled with each components values for each file separately.



(a) Screenshot of the Inventor plugin in *initial scene* mode with highlighted *GUI* representing the *BDU* example.



(b) Screenshot of the Inventor window of the *BDU* example's initial scene.

Figure 6.3: Screenshots of the Inventor plugin in *initial scene* mode with highlighted *GUI* elements and the corresponding Inventor window representing the *BDU* example.

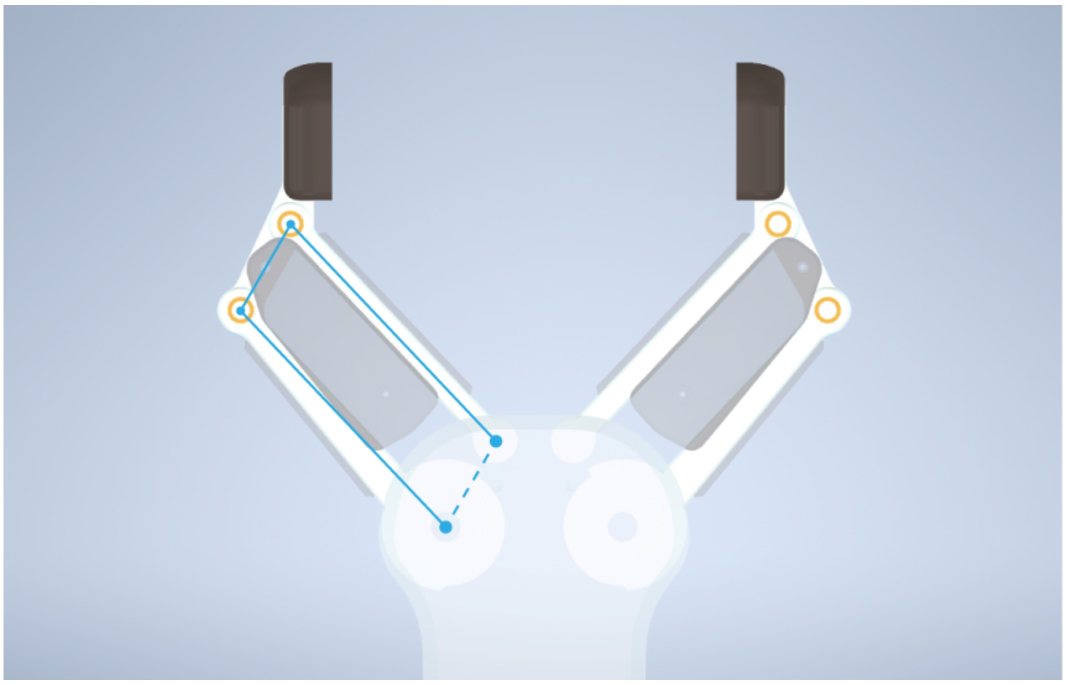


Figure 6.4: Screenshot of the lateral cut of a RG6 gripper.

The *STL* files need to be placed inside the *parts* folder, which resides inside the *meshes* folder of the *assembly_sim* package. The *SDF* files belong inside the *description* folder of the *assembly_sim* package.

It is possible to apply an offset to the initial scene using the button number 4 in Figure 6.3a. This opens a dialog, that allows entering an *X*, *Y* and *Z* offsets to the center point of the assembly. If the *CAD* file, represented by the initial state, is created without the initial scene of the actual robotic setup in mind, this functionality enables to translate the assembly location and therefore the initial scene to a different position, allowing for example the usage of a table with a different height.

6.2.4 Dual-Arm Robot

Originally the idea was to spawn individual robots separately using the pre-existing UR5 configuration. This solution proved to be infeasible, since it meant that separate *MoveIt!* instances with different planning scenes had to be initialized. This caused the individual robots to not know about each others position during planning, leading to arms colliding with each other. This was solved by creating a single dual-arm controller that contained both arms as separate *MoveIt!* move groups with each a RG6 gripper as their end-effector.

The RG6 gripper was chosen because of its out-of-the-box compatibility with UR5 arms. Due to technical issues in the available real robot system, the decision was made to switch to a simulation. Since OnRobot does only provide a triangle mesh in form of a *STL* files for all sub-components of its gripper, the gripper model had to be manually combined into a kinematic model, which is understood by Gazebo and *ROS*. This includes finding out which joints need to be added to the gripper, exporting the individual links as *STL* files and then creating a *URDF* or *xacro* file that combines all these links via joints. Unfortunately, *URDF* faces a major hurdle with graph-like intersections in joint models, because it only allows robots to be designed in tree-like hierarchies. As can be seen in Figure 6.4, non-parallel grippers, such as the RG6, have two joints, which connect from and to the same parent and child link. In this specific case four joints form a closed loop cycle, where only one of them is actively driven, while the other three are passive and dependent on the active joint.
To solve this issue, a plugin for Gazebo called *MimicJointPlugin*[8] is used. The idea is to remove one of the passive joints in each finger, making the *URDF* valid again, while at the same time actively controlling each passive joint of the closed loop depending on the only active joint. Essentially, the plugin allows to set varying degrees of dependability between joints.
The only driven joint of each finger is controlled using a GripperActionController[9], which connects to a transmission of a *PositionJointInterface*[10].

[8]https://github.com/roboticsgroup/roboticsgroup_gazebo_plugins
[9]http://wiki.ros.org/gripper_action_controller
[10]https://github.com/ros-controls/ros_control/wiki/hardware_interface

6.2.5 Initial Scene

As spawning scene, a simple table was created using two boxes linked together, one as a surface and one as table leg. The components that are exported using the plugin, are spawned on the table. There is an issue with just spawning components into the scene, since Gazebo does not handle freely moving components and their inertia well enough for parts to be intertwining without being connected by joints. This works well enough for flat surfaces or simple primitive geometrical shapes, but round surfaces never completely stand still. This end in components colliding, intersecting and then finally getting launched far away from each other. To counteract this issue, individual components that are supposed to be intersecting, connected or fixed to each other are spawned or placed by linking them together. Spawning components must therefore be done with the simulation on pause until all parts are added and linked successfully. Since this procedure can take an arbitrary amount of time, the parameter *delay* for the *simulation.launch* file must be set depending on the system performance. It denotes the duration of delay for linking pre-satisfied components before un-pausing the simulation.
The process of detaching components from the Gazebo world and attaching them to a gripper and vice versa is presented as *ROS* services, which are called *attach* and *detach*. Linking and unlinking components is also available as a *ROS* service and are called *link* and *unlink*. These four services have been developed during this work are further described in Section 6.2.7

6.2.6 SkiROS

On top of to the previously described basic ontology file (*base.owl*) file needed for the assembly representation inside *SkiROS*, an ontology file defining the robot had to be added for *SkiROS* to function properly. This file includes the default position on which the grounded part of the assembly should be placed, which can be used to place the grounded component if the assembling process does not start with the grounded component already in place. The ontology file generated by the Inventor plugin needs to be placed inside the *owl* folder of the *assembly_skiros* package, together with the just described, pre-defined ontology files.

SkiROS also needs all skills and primitives, that should be available to the system, implemented and configured by the developer. All necessary skills and primitives for this work are presented in Section 5.1.6.1. They have been implemented based on the template classes presented by the documentation of the second version of *SkiROS*[11]. Such a *SkiROS* skill or primitive base class gives access to all input parameters and the world model. Based on this information, the correct parametrization of *MoveIt!* is found and executed inside the skill. The initial version of *SkiROS* had issues with the interaction with *MoveIt!*. To prevent the *SkiROS* skill manager from crashing because of that, an action server that connects to the *MoveIt!* services was implemented. This was only possible with the newer version of *SkiROS*.

The skill that can be executed to start the assembly process, is called *assemble*. The execution of that skill contains the entire plan interpretation process described in Section 5.1.5. The information, given in form of ontologies, is requested by the skill from the world model of *SkiROS*. The extracted information is then used to initiate a sequence of pre-implemented skills leading to a successful assembly.

6.2.7 Perception system and Gazebo

The existing simulation implementation of the UR5 contains a Kinect. The resulting data can be accessed in form of a point cloud. Using this data to identify all components and their orientations goes beyond the scope of this thesis.

Alternatively, a perfect vision system is added in form of a Gazebo plugin. It intercepts when new components are spawned and publishes the resulting information as collision objects to *MoveIt!*'s planning scene. These components are colored in green in Figure 6.5. The resulting poses are also converted into and published as static transforms, to allow for an easy

[11]https://github.com/Bjarne-AAU/skiros-demo/tree/master/skiros2.wiki

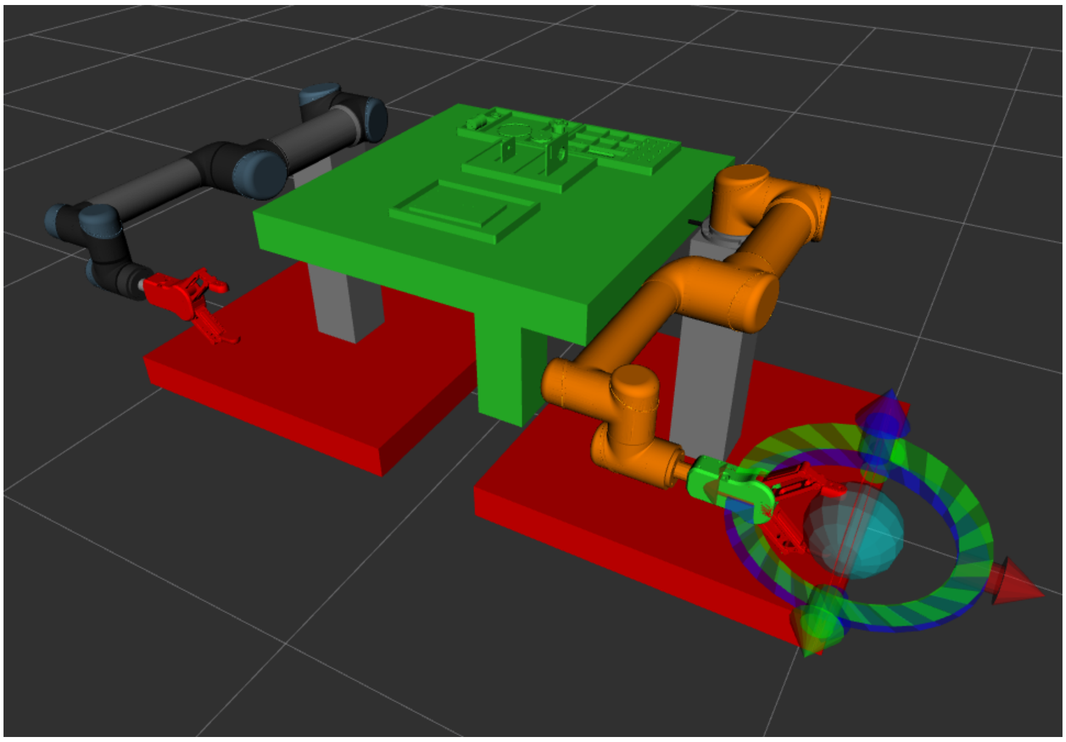


Figure 6.5: Screenshot of the *MoveIt!* planning scene including spawned components, represented in green color.

access to the component's poses during the execution of different skills and primitives.

When a component is grasped or placed inside the simulation, it is important to link or unlink it to or from the component it is related to. This can be done using the *ROS* services *link* and *unlink* developed during this work. The service *link* is used at the time of spawning the components to combine parts with pre-satisfied relations before unpausing Gazebo. The joints created between the components are rotational joints with a minimum limit equal to the maximum, which makes it a rigid connection. The services are located inside the perfect vision system, since they take advantage from the fact, that the vision system knows about each joint and link of all components, because are specified when spawning the *SDF* files using *ROS*.

Two additional, similar *ROS* services, *attach* and *detach*, have been developed during this work and describe the process of detaching components from the Gazebo world and attaching them to a gripper or vice versa. These services are located inside the perfect vision system, since that is the system that intercepts the spawning of components and creates collision objects for the *MoveIt!* planning scene. These collision objects need to be converted into attached collision objects for *MoveIt!* to recognize that a component is attached to a gripper.
A correct handling of planning scene components in *MoveIt!*, allowing consecutive changes to the assembly scene, goes beyond the scope of this thesis because of an unidentified problem and is part of future expansions elaborated in Chapter 8.

6.3 Implementation summary

The practical aspect of this work consist of four major implementations:

1. the plan and simulation input generation, implemented in C# and run on Windows, that connects to Inventor using the Inventor *API*;
2. the plan interpretation, implemented in Python and run on Ubuntu 16.04, that interacts with *SkiROS*;
3. the skill execution, implemented in Python and run on Ubuntu 16.04, that uses *MoveIt!* to interact with *ROS*;
4. the perfect vision system, implemented in C++ and run on Ubuntu 16.04, that closes the gap between Gazebo, *MoveIt!* and the skill execution.

The following sections describe each individual aspect, the generated output and the correlation to other implementations.

6.3.1 Plan and simulation-input generation

To export all necessary data from Inventor and make it available to the simulation, a Inventor plugin was developed. This program was written in C# using the *.NET* Framework. The *GUI* and the events, that are triggered on different buttons being pushed, have been created using the integrated development environment (*IDE*) Visual Studio, allowing to focus the implementation on four different facets: (1) the extraction of geometrical and semantical information from the assembly *CAD* model using the Inventor *API*; (2) the handling of multiple snapshots and general usability of the plugin; (3) the plan generation and (4) the file export from the plugin.

The Inventor *API* allows access to the internal representation of an assembly or a component using *COM* objects, which was already described in more detail in Section 2.9. The needed geometrical information of individual parts and semantical correlations between components can be found using this *API* is specified in Section 5.1.3.3. The internal representation of this accumulated data is similar to the class diagram shown in Figure 5.11. The only missing component from that figure is the plan itself, which consists of a sequential list of actions.

The discussed data extracted from the *CAD* model symbolizes an individual snapshot of the assembly in time. Such a snapshot is visualized as a graph using the *.NET* library *MSAGL* from Microsoft. It allows to define nodes and edges between nodes and then automatically generates the graph layout. In addition, the API was used to create usability features, such as setting the assembly to the state of a created snapshot by double-clicking it or highlighting screws inside Inventor that are selected in the plugin. The main usability aspect of the plugin is the creation of multiple snapshots. Each time a snapshot is created, the entire assembly is analyzed at its current state and saved in a list of snapshots.

This list of snapshots is used to generate the sequential plan of actions. The algorithms from Section 5.1.4 have also been implemented in C# to find the individual actions of each snapshot dependent on changes in regard to the previous snapshot. This results in a full plan of actions representing the assembly.

After this plan is generated, it is exported to an ontology file, which also includes the previously mentioned geometrical and semantical data extracted from the *CAD* model. The class structure of the ontology file is defined in Figure 5.11. The *XML*-like structure of the ontology output file is created without any additional library. The plugin uses hardcoded templates of all different data and property types needed for the presented class structure and fills them with each object's individual values. This information is then exported using C#'s native ability of writing text to files. This ontology file needs to be added to the *assembly_skiros* package, allowing *SkiROS* to load it. To ensure a correct format of the generated ontology files, the generated files were successfully opened using the ontology file editor Protégé[12], which validates ontologies on load.

To recreate a initial scene for the simulation that contains all needed components, an additional mode is implemented within the plugin. This mode allows the user to analyze a separately created *CAD* file representing the initial state of the assembly and export it to data formats, that can be used by the simulation. Since no snapshots are necessary for the creation of this representation, the *CAD* file is analyzed once, as described in Section 5.2. The Inventor *API* allows to export the *IPT* file of each component to *STL* files. Neither *ROS* nor Gazebo allow the direct usage of *STL* files to spawn components. These files are used to define links inside a *SDF* model file. The same direct strategy for creating the ontology output files is used to created the *SDF* files, since their file structure is also based on *XML*. Two additional files regarding the initial scene can be exported. The first file, called *parts.launch*, contains the commands used to import the *SDF* object models into *ROS* and to subsequently spawn them in Gazebo. The second file, called *link_parts.launch*, contains commands for pre-satisfied components to be linked together before Gazebo is unpaused.

The ontology file generated by the Inventor plugin needs to be placed inside the *owl* folder of the *assembly_skiros* package, where other pre-defined ontology files are located. The *STL* files need to be placed inside the *parts* folder, which resides inside the *meshes* folder of the *assembly_sim* package. The *SDF* files belong inside the *description* folder of the *assembly_sim* package. The *launch* files need to be placed inside the *launch* folder of the *assembly_sim* package.

[12]https://protege.stanford.edu/

6.3.2 Implementation of plan interpretation

The interpretation of the plan is implemented in Python using the *SkiROS* framework as a basis. It allows the implementation of modular skills and primitives that can easily be concatenated or hierarchically combined to create more complex behavior. The entry point for starting the interpretation of the plan inside the ontology is given by the *assemble* skill. This skill requests the plan from the world model using the skill manager and executes the algorithms presented in Section 5.1.5, which have also been implemented in Python. These algorithms then execute other skills, which are described in detail in Section 5.1.6.1. *SkiROS* handles the passing of parameters between skills and primitives and allows to define pre- and post-conditions. These conditions are yet to be implemented in a meaningful way inside *SkiROS*, since post-conditions are not used at all and pre-conditions only allow for checks regarding whether a skill or a primitive should be executed or skipped. *SkiROS* is yet to present a functional addition to their system, that derives the use and order of skills and primitives depending on their pre- and post-conditions.

6.3.3 Implementation of skill execution

As mentioned, there are two different type of primitives, that each skill can use: (1) moving an arm towards a target, which behaves differently depending on whether the robot arm is already holding a component and therefore inserting a part rather than grasping it; (2) opening or closing a gripper. Each skill and primitive is implemented in Python using the base classes for skills and primitives presented by the *SkiROS* framework. Since each skill consists of multiple primitives concatenated together and primitives represent atomic actions, the executed series of skills leads to a sequence of primitives interacting with the scene. Section 5.1.6.2 describes the two types of primitives available to the system: (1) moving an arm (2) and opening or closing the gripper.

Moving an arm requires a target component. If the executing skill is not grasping a new component but rather inserting an already grabbed one, the part held by the gripper is passed as a separate parameter. The global transformations of these two components, in regard to the simulated world, are published by the perfect vision system. These transformations, together with each components grasp poses and the relative transformation between the parts themselves, are used inside the arm motion primitive in order to compute a transformation chain that describes the end-effectors final pose when trying to grasp or insert a component. This pose is also used to calculate a more distant location, to which the arm will move first, in order to ensure a constant orientation while approaching a component, explained in more detail in Section 7.3. This procedure is implemented by calling the same arm motion primitive twice in a row and distinguish each call with the use of an additional parameter. This parameter tells the skill to either approach the distant pose with an unconstrained gripper orientation during the movement or to move towards the exact location with a fixed orientation. This way there is no need to implement two separate arm motion primitives for such similar behavior. The pose that should be reached and the constraints posed on the arm during the robot motion are defined by using the *MoveIt!* move group commander. This allows to set the final pose, orientational constraints for specific joints of the arm and constraints limiting each joint's reach.

Opening or closing the gripper also makes use of the *MoveIt!* move group commander to send gripper-specific commands. These commands contain a single value, which represents the current angle of the only active joints of a gripper.

An additional *ROS* packages and a metapackage have been developed and reused during this work. The package, *robot_descriptions*, contains the controllers, descriptions, models and launch files in regard to the dual-arm robot to work properly with *ROS* and *Gazebo*. The metapackage, *assembly_control*, contains necessary communication packages for the UR5 arm[13] for simulated and real environments, the *MimicJointPlugin*[14] for closed loop joints in *URDF* files and the dual-arm *MoveIt!* configuration. The setup for *MoveIt!* is based on the official implementation[15] of a *MoveIt!* configura-

[13] https://github.com/ros-industrial/universal_robot

[14] https://github.com/roboticsgroup/roboticsgroup_gazebo_plugins

[15] https://github.com/ros-industrial/universal_robot/tree/kinetic-devel/ur5_moveit_config

tion for a single UR5 arm and was modified to contain two individually controllable UR5 robot arms with each a RG6 gripper as end-effector.

6.3.4 Perfect vision Gazebo plugin

In order to close the gap between Gazebo and *MoveIt!*, a Gazebo plugin was developed. It is based on an existing implementation[16] of a Gazebo plugin. When loaded, this plugin binds a function to the Gazebo *ConnectWorldUpdateBegin* event. During the Gazebo world update event, this function looks for components inside the Gazebo world. Each component is registered as a *CollisionObject* and published to the *MoveIt!* planning scene when first recognized. On every following update, the pose of each *CollisionObject*[17] is updated, based on the current pose of the component inside the Gazebo world. The *CollisionObject* requires the triangle mesh for each component's link, in order to represent the part correctly. The component's model in Gazebo can be accessed, allowing to identify all joints and their meshes.

In addition, the plugin also contains four *ROS* services:

1. *attach*, used to detach a *CollisionObject* from the simulated world and attach it as an *AttachedCollisionObject*[18] the gripper in action;
2. *detach*, used to detach an *AttachedCollisionObject* from the gripper in action and attach it as a *CollisionObject* to the simulated world;
3. *link*, used to create a rigid joint between two components, and
4. *unlink*, used to remove a connection, previously created with the *link* service.

Rigid joints are technically not supported by the Gazebo physics engine, thus rotational ones are used. These joints have a minimum limit equal to the maximum, which makes it a rigid connection.

[16] http://docs.ros.org/hydro/api/gazebo_plugins/html/gazebo__ros__moveit__planning__scene_8cpp_source.html

[17] http://docs.ros.org/melodic/api/moveit_msgs/html/msg/CollisionObject.html

[18] http://docs.ros.org/melodic/api/moveit_msgs/html/msg/AttachedCollisionObject.html

7 Evaluation

The main focus of this chapter is on the evaluation of the proposed automation of the assembly task. Multiple assembly plans from different example assemblies of varying complexity will be exported with the presented methodology and the resulting sequence of actions will be evaluated. This includes the evaluation of the plan resulting from the *BDU* assembly. The interpretation of each plan is also evaluated in regard to the feasibility of each skill. Moreover, the transformations, poses and skills used to execute the plan inside the simulation are evaluated by partial execution of individual assembly actions in simulation.

7.1 Plan evaluation

In the following section, different assemblies with varying levels of complexity are evaluated for the feasibility of the generated sequence of actions. The generated sequence is presented alongside the graph of each involved snapshot and a visual representation of the full assembly in either step-by-step solutions, exploded views or a mixture of both. The feasibility of each plan is evaluated by discussing each individual action, groups of actions or important actions regarding the type of scenario. The different evaluated examples include: (1) a simple assembly of a toggle switch, consisting of three components, to evaluate a general assembly; (2) a simple subassembly of a perfume bottle, consisting of four components, to evaluate a general subassembly; (3) a more complex assembly of a clutch, containing a nested assembly and the need for a snapshot, to evaluate snapshots and the general assembly approach for many components; (4) a block world example, where the component order needs to be changed, to evaluate changes inside an assembly and to evaluate the applicability of the presented methodology to a simple dis- and re-assembly; (5) the running example of a *BDU*, to evaluate the full methodology presented throughout this work. This includes general assemblies and subassemblies, moving components, some parts blocking the access to others and deformable multi-contact components.

7.1.1 Simple assembly

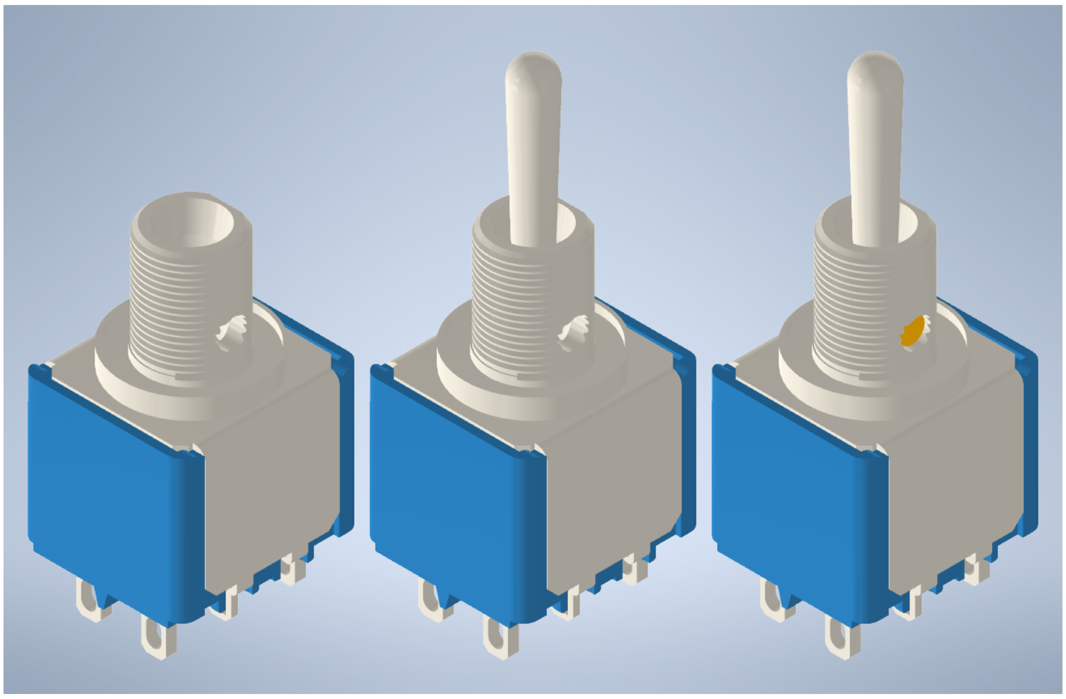
The first example is a simple assembly of a toggle switch based on three components, a *case*, a *baton* and a *pin*. The *case* of the switch is assumed to be pre-assembled in a separate step and is the grounded component of the assembly.
Figure 7.1 contains all representations regarding the simple assembly example. Figure 7.1a shows a step-by-step solution of the assembly using the *CAD* models of all parts. Figure 7.1b shows the connection graph between the involved components generated by the Inventor plugin when creating a snapshot based on the *CAD* file in Figure 7.1a. The plan, that is exported using the Inventor plugin after all snapshots are created, leads to a plan with two separate actions, which are represented in the list in Figure 7.1c.

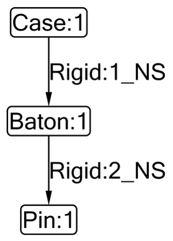
The first action is inserting the *baton* into the top slot of the *case*. Secondly, this *baton* needs to be fixed to the *case*. In order to do so, a small *pin* is inserted into the *baton* through a side slot in the *case*.

This example is used to visualize the general concept of a linear assembly process, in which component are connected face-to-face with the next.

7.1.2 Simple subassembly

The second example is a simple subassembly of a perfume bottle based on five components, a *bottle*, a *straw*, a *spring*, the *top* part used as a pushable element and a *brace* closing up the assembly. The *bottle* is the grounded component.
Figure 7.2 contains all representations regarding the simple assembly example. Figure 7.2a shows a step-by-step solution of the assembly using the *CAD* models of all parts. Figure 7.2b shows the connection graph between


(a) Step-by-step solution of the simple assembly example (left to right).

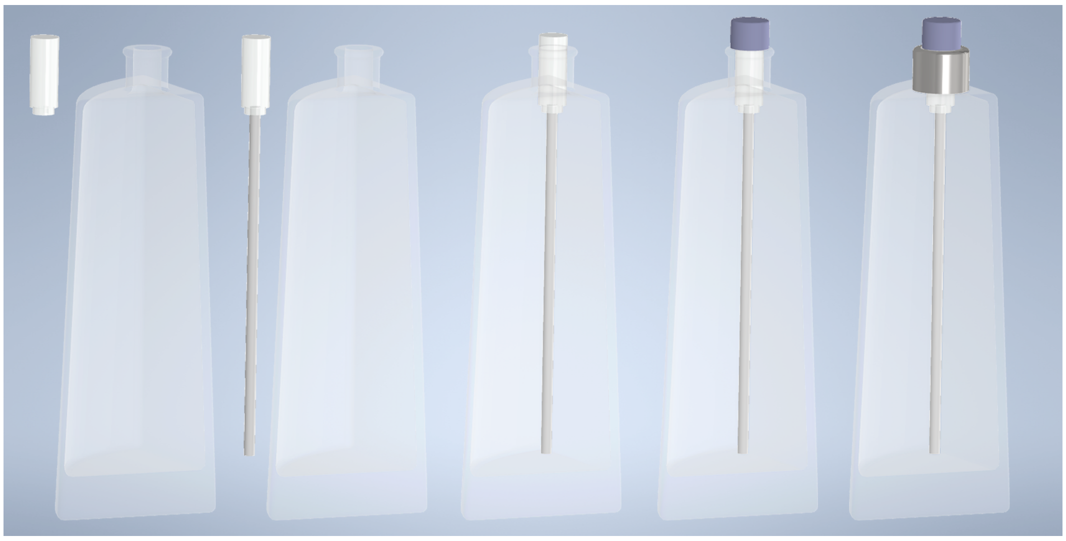


(b) Connection graph of the simple assembly example.

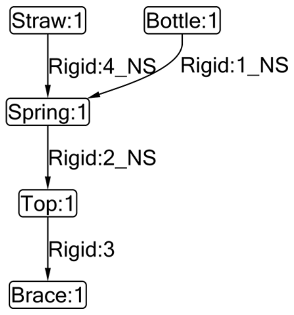
#	Parent	Child	undo	presat	screw
1	Case:1	Baton:1	no	no	no
2	Baton:1	Pin:1	no	no	no

(c) List of actions of the simple assembly example.

Figure 7.1: Representations of a simple assembly example, including step-by-step instructions, the graph and the list of actions resulting from the generated plan.



(a) Step-by-step solution of the simple subassembly example (left to right).



(b) Connection graph of the simple subassembly example.

#	Parent	Child	undo	presat	screw
1	Straw:1	Spring:1	no	no	no
2	Bottle:1	Spring:1	no	no	no
3	Spring:1	Top:1	no	no	no
4	Top:1	Brace:1	no	no	yes

(c) List of actions of the simple subassembly example.

Figure 7.2: Representations of a simple subassembly example, including step-by-step instructions, the graph and the list of actions resulting from the generated plan. Screws where selected using the Inventor plugin.

the involved components. The resulting plan leads to four separate actions, which are represented in the list in Figure 7.2c.

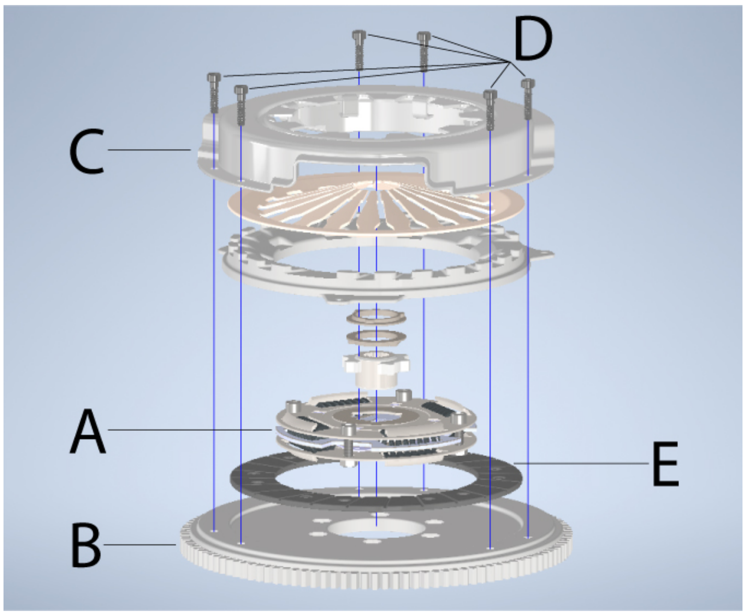
The first action is the actual subassembly part of the assembly, which is inserting the the *straw* into the *spring*. This *spring* is then inserted into the opening of the *bottle*.
The importance of applying subassembly rules can be shown by assuming the opposite scenario. If normal assembly rules are used to assemble the perfume bottle, the resulting plan would have the order of the first and the second action flipped, since the assembly process would start by placing the component related to the grounded part, which is the *spring*. When the *straw* would need to be inserted into the opening on the bottom of the *spring*, the *spring* would already be inside the *bottle* and the opening for the *straw* would already be fully enclosed by said *bottle*. This means there would be no possible way for the robot to place the *straw* without colliding with the bottle.

The third and fourth actions are attaching the *top* part onto the *spring* and then fixating all components with a *brace*, which is stacked onto the *top* part. The last action is marked as being of the screw type.

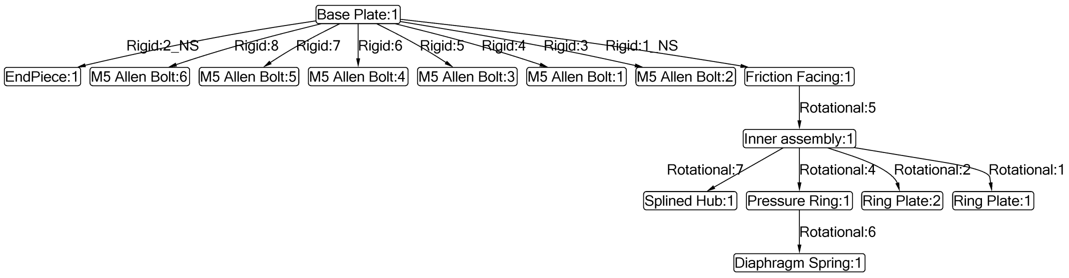
This example is used to visualize the general concept of a subassembly process. During creation of the *CAD* model it became obvious that some components needed to be assembled before being attached to already placed components.

7.1.3 Nested assembly

The third example is a more complex assembly of a clutch. It consists of many different components that mostly stack on top of each other and is split into an *outer assembly* (represented by Figure 7.3a) and an *inner assembly* (*A* in Figure 7.3a). Figure 7.3 contains all representations regarding the *outer assembly*, whereas Figure 7.4 contains all representations in of the *inner assembly*. The grounded parts of the two assemblies are the *base plate* (*B* in Figure 7.3a) and the *spring hub* (*G* in Figure 7.4a) respectively.



(a) Exploded view of the *outer assembly* of the nested assembly example. The letters are relevant to the description inside the running text.

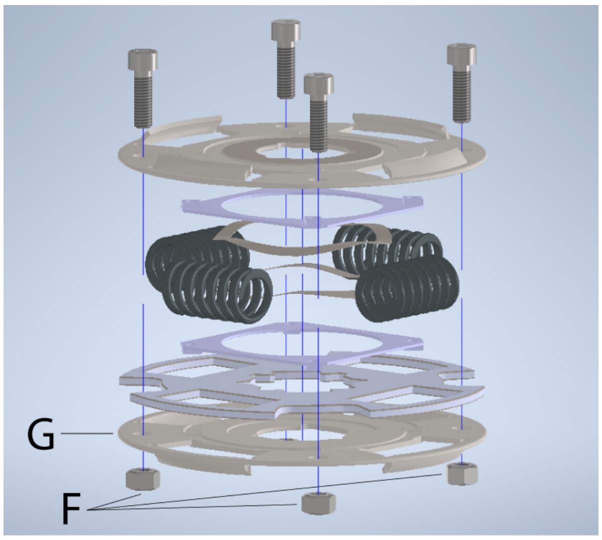


(b) Connection graph of the *outer assembly* of the nested assembly example's second snapshot, including the *end piece* and all *M5 Allen Bolts*.

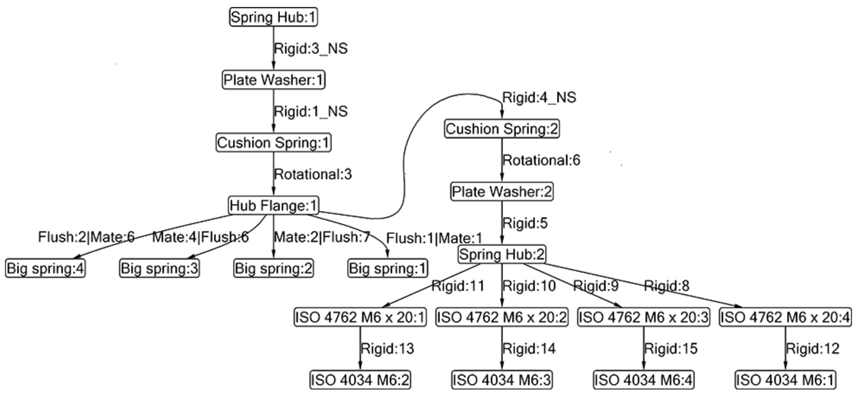
#	Parent	Child	undo	presat	screw	snapshot
1	Base Plate:1	Friction Facing:1	no	no	no	1
2	Friction Facing:1	Inner assembly:1	no	no	no	1
3-4	Inner assembly:1	Ring Plate:1-2	no	no	no	1
5	Inner assembly:1	Pressure Ring:1	no	no	no	1
6	Inner assembly:1	Splined Hub:1	no	no	no	1
7	Pressure Ring:1	Diaphragm Spring:1	no	no	no	1
8	Base Plate:1	End Piece:1	no	no	no	2
9-14	Base Plate:1	M5 Allen Bolt:1-6	no	no	yes	2

(c) List of actions of the *outer assembly* of nested assembly example.

Figure 7.3: Representation of the *outer assembly* of the nested assembly example, including the graph and the list of actions resulting from the generated plan.



(a) Exploded view of the *inner assembly* of the nested assembly example. The letters are relevant to the description inside the running text.



(b) Connection graph of the *inner assembly* inside the nested assembly example.

#	Parent	Child	undo	presat	screw
1	Spring Hub:1	Plate Washer:1	no	no	no
2	Plate Washer:1	Cushion Spring:1	no	no	no
3	Cushion Spring:1	Hub Flange:1	no	no	no
4	Hub Flange:1	Cushion Spring:2	no	no	no
5-8	Hub Flange:1	Big spring:1-4	no	no	no
9	Cushion Spring:2	Plate Washer:2	no	no	no
10	Plate Washer:2	Spring Hub:2	no	no	no
11-14	Spring Hub:2	ISO 4762 M6 x 20:1-4	no	no	no
15-18	ISO 4762 M6 x 20:1-4	ISO 4034 M6:1-4	no	no	yes

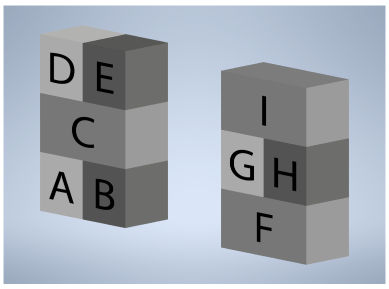
(c) List of actions of the *inner assembly* inside the nested assembly example.

Figure 7.4: Representation of the *inner assembly* of the nested assembly example, including the graph and the list of actions resulting from the generated plan.

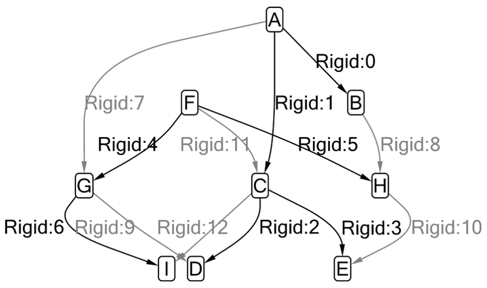
Figure 7.3a shows an exploded view of the *outer assembly*'s *CAD* model. Starting with the grounded part, all following components, including the *inner assembly*, should be stacked on top of each other, until no parts other than the *end piece*(*C* in Figure 7.3a) and all six *MS Allen Bolt*s (*D* in Figure 7.3a) are left. Then the *end piece* needs to be placed on top of the *base plate* and all components are fixated together using the left-over bolts that need to be inserted through the *end piece* into the *base plate*. To achieve this, two separate snapshots need to be added to the assembly, first without, then with the *end piece* and the *MS Allen Bolt*s. If only one snapshot is used, the *end piece* and the *MS Allen Bolt*s would be added immediately after or possibly even before the *friction facing* (*E* in Figure 7.3a) is placed, thus blocking the access for all following components. Figure 7.3b shows the full connection graph between the involved components, represented by the second snapshot. The graph of the first snapshot would not contain any connections from or to the *base plate*, except to the *friction facing* part. The resulting plan leads to the list of actions represented in Figure 7.3c.

Figure 7.4a shows an exploded view of the *inner assembly*'s *CAD* model, which shows the importance of assembly nesting for this example. As can be seen in Figure 7.4a, the access to the *ISO 4034 M6* (*F* in Figure 7.3a) nuts in the *inner assembly* is blocked, if the grounded component is not elevated by a specifically designed holding fixture. Therefore, if this segment would not be handled as a separate assembly, the placement of the *spring hub* onto the *friction facing* part would prevent the placement of the *ISO 4034 M6* nuts. Figure 7.4b shows the linear connection graph of the *inner assembly* between the involved components. The resulting plan leads to the list of actions represented in Figure 7.4c.

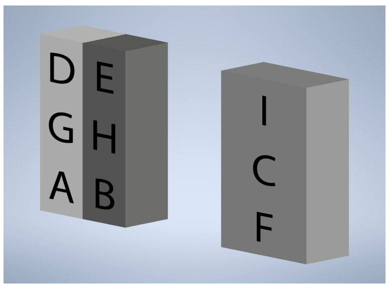
This example is used to visualize the two major concepts of snapshot creation and nested assembly processes. The snapshots were used to place enclosing components last and the necessity of nesting became obvious during the creation of the *CAD* model to ensure full access to all components during insertion.



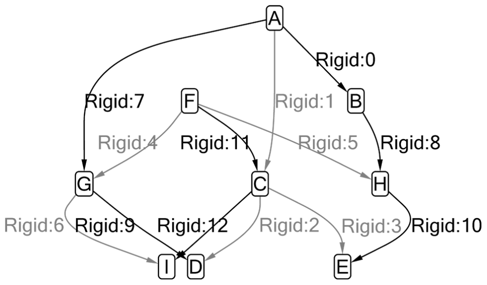
(a) Screenshot of the initial block world example.



(b) Connection graph of the initial block world example.



(c) Screenshot of the changed block world example.



(d) Connection graph of the changed block world example.

#	Parent	Child	undo	presat	snapshot
1	A	C	no	no	1
2	A	B	no	no	1
3	C	D	no	no	1
4	C	E	no	no	1
5	F	G	no	no	1
6	F	H	no	no	1
7	G	I	no	no	1
8	G	I	yes	no	2
9	F	H	yes	no	2
10	F	G	yes	no	2
11	C	E	yes	no	2
12	C	D	yes	no	2
13	A	C	yes	no	2
14	A	G	no	no	2
15	G	D	no	no	2
16	B	H	no	no	2
17	H	E	no	no	2
18	F	C	no	no	2
19	C	I	no	no	2

(e) List of actions of the full block world example.

Figure 7.5: Step-by-step representation of a block world example, including the graph and the list of actions resulting from the generated plan.

7.1.4 Block world

As a fourth example a block world example is presented. The main focus for this scenario lies in presenting adaptations regarding the plan extraction algorithms. The previously presented methodology is not able to fully undo or extract already placed components and does not handle changes between snapshots in a manner, that allows undoing all actions of the previous snapshot necessary to successfully assemble the next one. Screws do not exist in this context and multiple grounded components are used during this example to show that it is possible, but unnecessary.

The block world example is represented by Figure 7.5 and is divided into two separate constellation of components using snapshots. The first snapshot, representing the initial scene, is shown in Figure 7.5a as a *CAD* model. The second snapshot, representing the changed scene, can be seen in the *CAD* model in Figure 7.5c. The corresponding connection graphs between the involved components can be found in Figure 7.5b for the first and in Figure 7.5d for the second snapshot respectively. The resulting plan leads to the list of actions represented in Figure 7.5e.

Algorithms 5.10, 5.11 and 5.12 are not necessary for the block world scenario. Being able to ignore most corner cases of an assembly, a simple solution to finding changes in the block world example is to look for the earliest action, that contains changes, within the list of the previous snapshot's actions. All actions after this point need to be undone in an inverted order than previously executed, since blocks are placed loosely on top of each other. Then, all actions from the new snapshot need to be added, ignoring the same, unchanged actions common to both snapshots.

This example is used to visualize the necessary changes to the presented methodology in order to enable the system to dis- and re-assemble an assembly in a different manner.

7.1.5 Belt Drive Unit

The BDU assembly was chosen as a running example for this thesis, because of its higher level of complexity. It contains multiple subassemblies, a

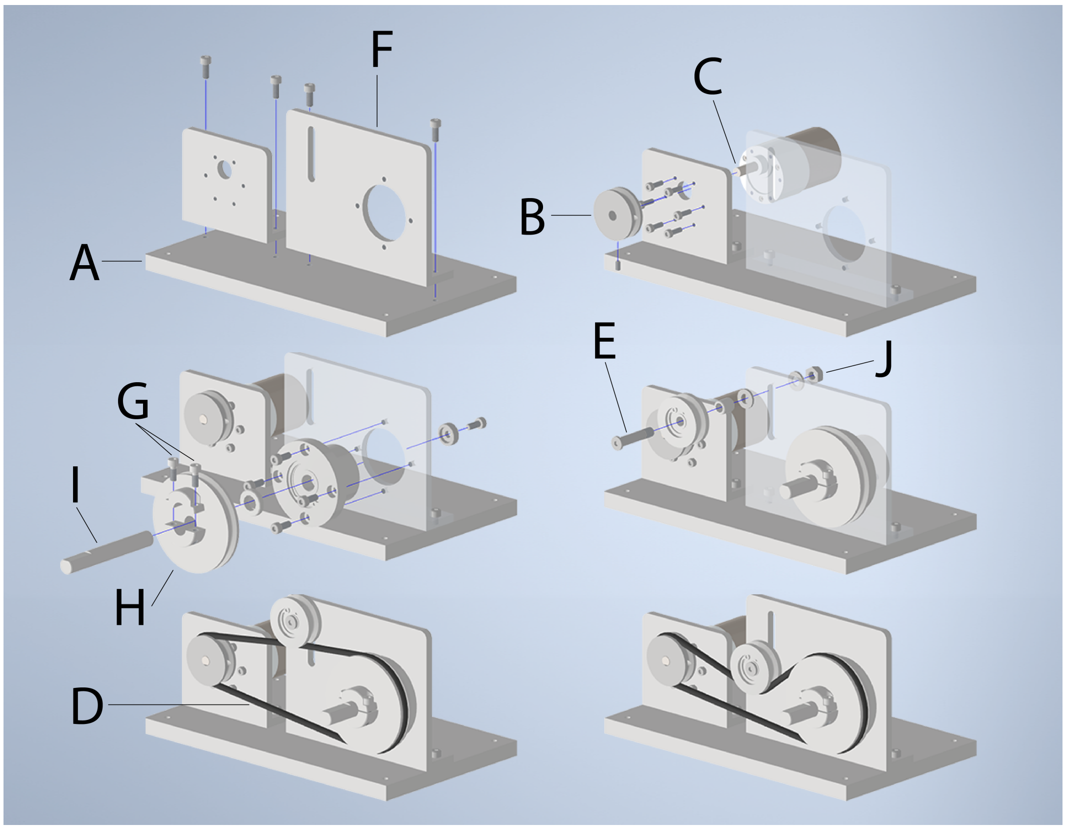
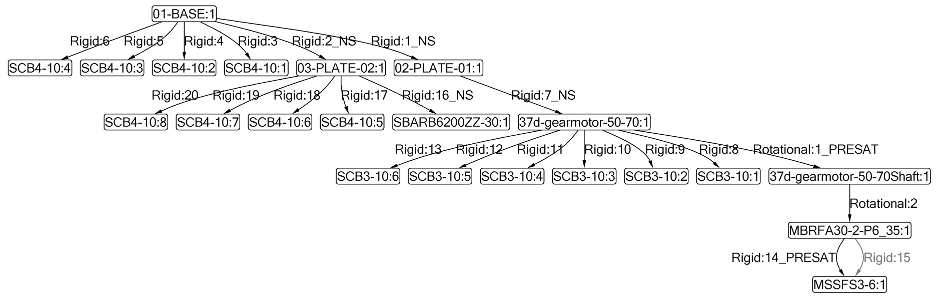


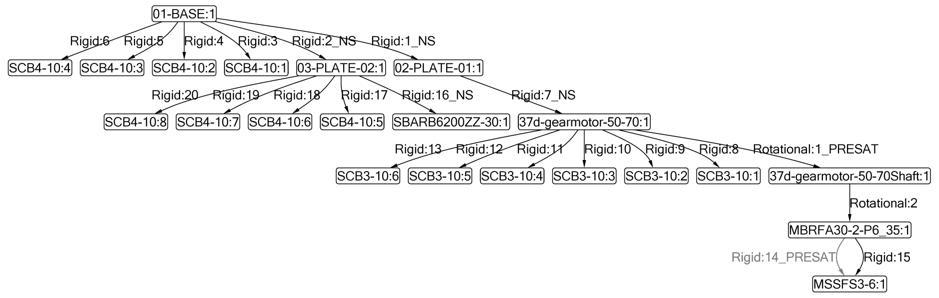
Figure 7.6: Step-by-step solution of the *BDU* example (left to right and top to bottom). The letters are relevant to the description inside the running text.

moving part, a deformable belt and components with blocked access after specific parts are placed. The *base plate* (*A* in Figure 7.6) is the grounded component of the assembly. The final result of the assembly can be seen in Figure 4.1. An exploded view can be found in Figure 4.2. Figure 7.6 shows a step-by-step solution of the assembly using exploded views of the main assembly steps inside the *CAD* model. Figure 7.8 shows the connection graph between the involved components of each snapshot separately. The resulting plan leads to the list of actions represented in Figure 7.9.

The first snapshot is represented in 7.7a and contains an assembly without subassemblies. Each component is either inserted into or placed onto an

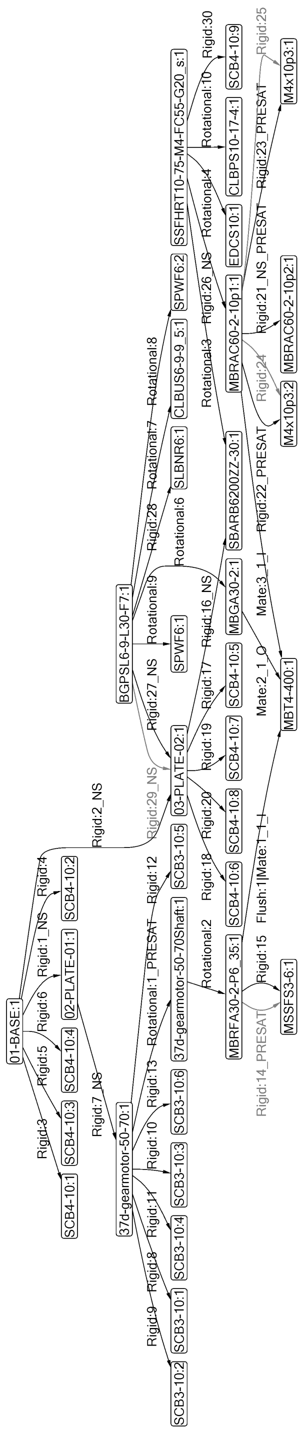


(a) Graph of the *BDU* example's first snapshot.

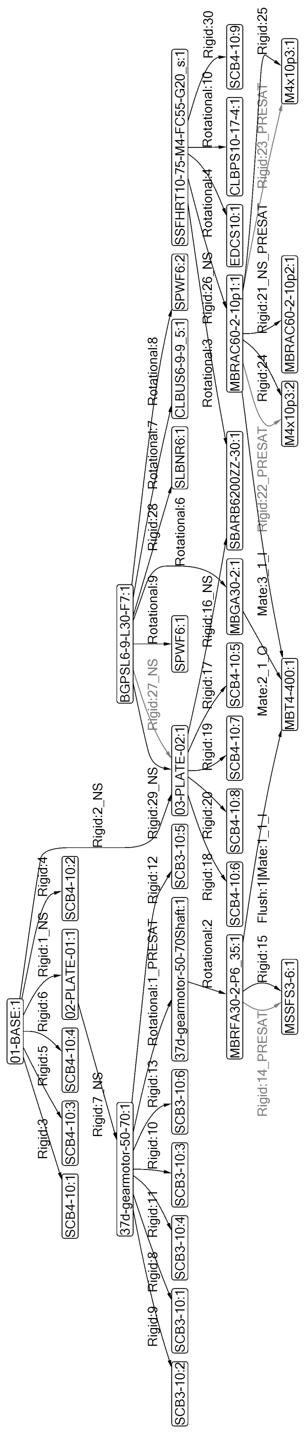


(b) Graph of the *BDU* example's second snapshot.

Figure 7.7: Graphs of the *BDU* example representing the first two snapshots.



(a) Graph of the *BDU* example's third snapshot.



(b) Graph of the *BDU* example's fourth snapshot.

Figure 7.8: Graphs of the *BDU* example representing the last two snapshots.

already previously placed part. These actions are visualized by the two images at the top of Figure 7.6.

The only difference to the second snapshot is, that a pre-satisfied relation is screwed further into the opening to fixate the disk *MBRFA30-2P6_35* (*B* in Figure 7.6) onto the *gearmotor shaft* (*C* in Figure 7.6). The second snapshot can be seen in Figure 7.7b. The action of this snapshot is part of the top right image of Figure 7.6.

That the third snapshot contains all the other components, including the subassemblies, has two reasons. Firstly, the access to the pre-satisfied relation, which changed in the previous snapshot, would be blocked by the rubber band *MBT4-400* (*D* in Figure 7.6) once inserted. Secondly, to simplify the scenario, the simulation is not intended to build subassemblies and put them aside until one of the subassembly components needs to be attached to the assembly. For this reason, the subassemblies are added using a snapshot after one of its components is already attached to the assembly. This is the case after the first two snapshots. The connection graph of snapshot three is shown in Figure 7.8a. The actions of this snapshot are visualized by the two middle images and the bottom left image of Figure 7.6.

The last snapshot contains the changes regarding the component *BGPSL6-9-L30-F7* (*E* in Figure 7.6). This shaft needs to be moved along an oblong slot inside *03-PLATE-02* (*F* in Figure 7.6), which means the relation between the shaft and the plate has changed. Also the pre-satisfied screws *M4x10p3* (*G* in Figure 7.6) are tightened, in order to fixate the disk *MBRAC60-2-10_p1:1* (*H* in Figure 7.6) to the rod *SSFHRT10-75-M4-FC55-G20_s* (*I* in Figure 7.6). The actions of this snapshot are visualized by the bottom right image of Figure 7.6.

Snapshot one only adds new components, just as every first snapshot does, and generates the first 21 actions in the final list of actions shown in Figure 7.9. The differences between snapshot one and two result in actions 22 and 23. The interpreter is able to identify such a pair of actions as a changed connection, since parent and child are the same components and the first action of the two is an undo. Undoing a component and then placing it again on the same part is interpreted as a shift of a component. For this reason action 22, an undo action with a screw relation, can be skipped and

#	Parent	Child	undo	presat	screw	snapshot
1	01-BASE:1	02-PLATE-01:1	no	no	no	1
2-3	01-BASE:1	SCB4-10:3-4	no	no	yes	1
4	01-BASE:1	03-PLATE-02:1	no	no	no	1
5-6	01-BASE:1	SCB4-10:1-2	no	no	yes	1
7	02-PLATE-01:1	37d-gearmotor-50-70:1	no	no	no	1
8	37d-gearmotor-50-70:1	37d-gearmotor-50-70Shaft:1	no	yes	no	1
9-14	37d-gearmotor-50-70:1	SCB3-10:1-6	no	no	yes	1
15	37d-gearmotor-50-70Shaft:1	MBRFA30-2-P6_35:1	no	no	no	1
16	MBRFA30-2-P6_35:1	MSSFS3-6:1	no	yes	yes	1
17	03-PLATE-02:1	SBARB6200ZZ-30:1	no	no	no	1
18-21	03-PLATE-02:1	SCB4-10:5-8	no	no	yes	1
22	MBRFA30-2-P6_35:1	MSSFS3-6:1	yes	yes	yes	2
23	MBRFA30-2-P6_35:1	MSSFS3-6:1	no	no	yes	2
24	BGPSL6-9-L30-F7:1	MBGA30-2:1	no	no	no	3
25	BGPSL6-9-L30-F7:1	CLBUS6-9-9_5:1	no	no	no	3
26	BGPSL6-9-L30-F7:1	SPWF6:2	no	no	no	3
27	BGPSL6-9-L30-F7:1	03-PLATE-02:1	no	no	no	3
28	BGPSL6-9-L30-F7:1	SPWF6:1	no	no	no	3
29	BGPSL6-9-L30-F7:1	SLBNR6:1	no	no	yes	3
30	SSFHRT10-75-M4-FC55-G20_s:1	MBRAC60-2-10_p1:1	no	no	no	3
31	SSFHRT10-75-M4-FC55-G20_s:1	CLBPS10-17-4:1	no	no	no	3
32	SSFHRT10-75-M4-FC55-G20_s:1	SBARB6200ZZ-30:1	no	no	no	3
33	SSFHRT10-75-M4-FC55-G20_s:1	EDCS10:1	no	no	no	3
34	SSFHRT10-75-M4-FC55-G20_s:1	SCB4-10:9	no	no	yes	3
35	MBRAC60-2-10_p1:1	MBT4-400:1	no	no	no	3
36	MBRAC60-2-10_p1:1	MBRAC60-2-10_p2:1	no	yes	no	3
37-38	MBRAC60-2-10_p1:1	M4x10p3:1-2	no	yes	yes	3
39	BGPSL6-9-L30-F7:1	SLBNR6:1	yes	no	yes	4
40	BGPSL6-9-L30-F7:1	03-PLATE-02:1	yes	no	no	4
41	BGPSL6-9-L30-F7:1	03-PLATE-02:1	no	no	no	4
42	BGPSL6-9-L30-F7:1	SLBNR6:1	no	no	yes	4
43	MBRAC60-2-10_p1:1	M4x10p3:2	yes	yes	yes	4
44	MBRAC60-2-10_p1:1	M4x10p3:2	no	no	yes	4
45	MBRAC60-2-10_p1:1	M4x10p3:1	yes	yes	yes	4
46	MBRAC60-2-10_p1:1	M4x10p3:1	no	no	yes	4

Figure 7.9: List of actions of the *BDU* example.

the following action, that also has a screw relation, is going to be interpreted as tightening the screw.

Snapshot three, just as the first one, only adds new components to the assembly, represented by actions 24 to 38.
All actions after 38 are part of the fourth snapshot. Action 39 is the loosening of the screw *SLBNR6* (*J* in Figure 7.6) in order to move the shaft *BGPSL6-9-L30-F7*, whereas action 42 is fastening the screw again. In between these two, actions 40 and 41 are again an example of two consecutive actions that are undone and redone between the same components, which instead results in skipping the first action and interpreting the second as some sort of change in an already existing relation. Since these actions do not contain a screw relation, action 41 is interpreted as shifting a component inside a slot, instead of fastening a screw. The same as for actions 22 and 23 is valid for the actions 43 and 44 and the last two actions 45 and 46.

This example is used to visualize all concepts presented in thesis used to automatize a complex assembly process, where some components are stacked on top of each other, others are part of a subassembly, some have to be moved and others block access to parts after they are placed.

7.2 Plan interpretation evaluation

In the following section, the plans resulting from each of the previously evaluated assemblies are interpreted for the presented dual-arm robotic setup and evaluated for the feasibility of the generated sequence of skills. The process of evaluation is of a quantitative nature and is executed by domain experts. The considered aspects are, whether arms are always empty before they are used to grasp a component, whether an arm blocks the access to the grasp or insert location of the next arm or whether any skills will be executed in the wrong order. The arms of the dual-arm robotic setup are named *left* and *right* for the following sections.

#	Skill	Arm	Target	Other target	Action
1	Grasp	left	Baton:1	-	1
2	Insert	left	Case:1	Baton:1	1
3	Grasp	right	Pin:1	-	2
4	Insert	right	Baton:1	Pin:1	2
5	Release	left	Baton:1	-	2
6	Release	right	Pin:1	-	2

Figure 7.10: List of skills of the simple assembly example.

7.2.1 Simple assembly interpretation

The plan for the simple assembly example from Section 7.1.1 is interpreted for a dual-arm robotic system and the resulting list of skills is presented in Figure 7.10. The list shows the name of the executed skill, the arm that executes it, the target component to approach, the other target component if a component is already grasped and the action it was generated from.

The first two skills represent grasping the *baton* with the *left* arm and inserting it into the *case*. The next two skills represent grasping the *pin* with the *right* arm and inserting it into the *baton*, which is still being held, else it could fall to the side. The last two skills are releasing both grasped components because the assembly is finished.

All components are grasped when they need to be and the execution order is correct. If the *grasp* poses are chosen accordingly, no arm is blocking the other.

7.2.2 Simple subassembly interpretation

The plan for the simple subassembly example from Section 7.1.2 is interpreted for a dual-arm robotic system and the resulting list of skills is presented in Figure 7.11. Please note that the *screw* skill releases components after its execution. This *release* is not executed as a separate skill, but is part of the *screw* skill and thus not part of the list.

The first action contains two *grasping* skills, since none of the components have been placed yet. The reason for this is, that the assembly process starts

#	Skill	Arm	Target	Other target	Action
1	Grasp	left	Spring:1	-	1
2	Grasp	right	Straw:1	-	1
3	Screw	left	Spring:1	Straw:1	1
4	Screw	left	Bottle:1	Spring:1	2
5	Grasp	left	Top:1	-	3
6	Screw	left	Spring:1	Top:1	3
7	Grasp	left	Brace:1	-	4
8	Screw	left	Top:1	Brace:1	4

Figure 7.11: List of skills of the simple subassembly example.

with a subassembly between two parts, where none of the two components are grounded. Skill 3 represents the straw being *screwed* into place and therefore the *right* arm is free again. The left arm still contains the *spring*, which is *screwed* into the bottle during execution of skill 4. Skills 5 and 6 represent *grasping* the *top* part and *screwing* it onto the spring. Similarly, skills 7 and 8 indicate the *grasping* of the *brace* and the *screwing* of it onto the top part. The *left* arm is free after each *screwing* skill, which results in almost all skills being executed by the *left* arm, except for the very first action, where two components need to be grasped.

All components are grasped when they need to be and the execution order is correct. No arm can be blocking the other, since they collaborate initially and then only one is in use.

7.2.3 Nested assembly interpretation

The plans for the nested assembly example's *outer assembly* and *inner assembly* from Section 7.1.3 are interpreted for a dual-arm robotic system and the resulting lists of skills are presented in Figure 7.12 for the *outer assembly* and in Figure 7.13 for the *inner assembly*. Figure 7.12 also includes the snapshot number the action was generated from.

The first twenty skill executions of the *outer assembly* follow the same pattern: (1) *grasp* **b**; (2) *insert* **b** into **a**; (3) *release* **a**; (4) *grasp* **c**; (5) *insert* **c** into **a** or **b**; (6) *release* **b** and so on. The only deviation is, that initially **a** is the grounded component, which is already in place and therefore does not need to be *released*. Skill 21 represents *releasing* the second arm, because a component

#	Skill	Arm	Target	Other target	Action	Snapshot
1	Grasp	left	Friction Facing:1	-	1	1
2	Insert	left	Base Plate:1	Friction Facing:1	1	1
3	Grasp	right	Inner assembly:1	-	2	1
4	Insert	right	Friction Facing:1	Inner assembly:1	2	1
5	Release	left	Friction Facing:1	-	2	1
6	Grasp	left	Ring Plate:1	-	3	1
7	Insert	left	Inner assembly:1	Ring Plate:1	3	1
8	Release	right	Inner assembly:1	-	3	1
9	Grasp	right	Ring Plate:2	-	4	1
10	Insert	right	Inner assembly:1	Ring Plate:2	4	1
11	Release	left	Ring Plate:1	-	4	1
12	Grasp	left	Pressure Ring:1	-	5	1
13	Insert	left	Inner assembly:1	Pressure Ring:1	5	1
14	Release	right	Ring Plate:2	-	5	1
15	Grasp	right	Splined Hub:1	-	6	1
16	Insert	right	Inner assembly:1	Splined Hub:1	6	1
17	Release	left	Pressure Ring:1	-	6	1
18	Grasp	left	Diaphragm Spring:1	-	7	1
19	Insert	left	Pressure Ring:1	Diaphragm Spring:1	7	1
20	Release	right	Splined Hub:1	-	7	1
21	Release	left	Diaphragm Spring:1	-	7	1
22	Grasp	left	End Piece:1	-	8	2
23	Insert	left	Base Plate:1	End Piece:1	8	2
24	Release	left	End Piece:1	-	8	2
25,27,...,35	Grasp	left	M5 Allen Bolt:1-6	-	9-14	2
26,28,...,36	Screw	left	Base Plate:1	M5 Allen Bolt:1-6	9-14	2

Figure 7.12: List of skills of the nested example's *outer assembly*.

was inserted, that does not have any connections to parts that have not been placed yet. The remaining skills, skill 22 to skill 36, also repeat the same pattern, where the component in regard is *grasped*, *inserted* and *released*.

The execution of the first 41 skills of the *inner assembly* is repeating the same pattern as the *outer assembly*. The last eight skills represent *screwing* the *ISO 4034 M6* nuts onto their respective screw, while also *releasing* the the last components, that is being held from the previous skills, after the first *ISO 4034 M6* nut is *inserted*.

All components are grasped when they need to be and the execution order is correct. The last three *ISO 4034 M6* nuts can be *grasped* and *inserted* with any arm, since both are free. If the *grasp* poses are chosen accordingly, no arm is blocking the other.

#	Skill	Arm	Target	Other target	Action
1	Grasp	left	Plate Washer:1	-	1
2	Insert	left	Spring Hub:1	Plate Washer:1	1
3	Grasp	right	Cushion Spring:1	-	2
4	Insert	right	Plate Washer:1	Cushion Spring:1	2
5	Release	left	Plate Washer:1	-	2
6	Grasp	left	Hub Flange:1	-	3
7	Insert	left	Cushion Spring:1	Hub Flange:1	3
8	Release	right	Cushion Spring:1	-	3
9	Grasp	right	Cushion Spring:2	-	2
10	Insert	right	Hub Flange:1	Cushion Spring:2	4
11	Release	left	Hub Flange:1	-	4
12	Grasp	left	Big Spring:1	-	5
13	Insert	left	Hub Flange:1	Big Spring:1	5
14	Release	right	Cushion Spring:2	-	5
15	Grasp	right	Big Spring:2	-	6
16	Insert	right	Hub Flange:1	Big Spring:2	6
17	Release	left	Big Spring:1	-	6
18	Grasp	left	Big Spring:3	-	7
19	Insert	left	Hub Flange:1	Big Spring:3	7
20	Release	right	Big Spring:2	-	7
21	Grasp	right	Big Spring:4	-	8
22	Insert	right	Hub Flange:1	Big Spring:4	8
23	Release	left	Big Spring:3	-	8
24	Grasp	left	Plate Washer:1	-	9
25	Insert	left	Cushion Spring:1	Plate Washer:1	9
26	Release	right	Big Spring:4	-	9
27	Grasp	right	Spring Hub:2	-	10
28	Insert	right	Plate Washer:1	Spring Hub:2	10
29	Release	left	Plate Washer:1	-	10
30,33,36,39	Grasp	left,right,...	ISO 4762 M6 x 20:1-4	-	11-14
31,34,37,40	Insert	left,right,...	Spring Hub:2	ISO 4762 M6 x 20:1-4	11-14
32,35,38,41	Release	left,right,...	Spring Hub:2,ISO 4762 M6 x 20:1-3	-	11-14
42,45,47,49	Grasp	right	ISO 4034 M6:1-4	-	15-18
43,46,48,50	Screw	right	ISO 4762 M6 x 20:1-4	ISO 4034 M6:1-4	15-18
44	Release	left	ISO 4762 M6 x 20:4	-	15

Figure 7.13: List of skills of the nested example's *inner assembly*.

7.2.4 Block world interpretation

The plan for the block world example from Section 7.1.4 is interpreted for a dual-arm robotic system. Because of the presented changes to the plan generation for this simplified scenario of an assembly that is dis- and then re-assembled differently, the resulting list of skills can be categorized by two different cases: (1) adding a component, which means *grasping*, *inserting* and *releasing* it or (2) removing a component, which means *grasping* it and *releasing* it back on the ground.

The first snapshot only consists of components being added. The second snapshot initially removes all components, that have changed or depend on a changed part and need to be moved for the re-assembly, and adds them again in a different order.

Because of the modular, independent nature of the possible cases, the dual-arm robotic system always uses the same arm for this example, since only one arm is needed.

All components are grasped and held for only one action at a time and the execution order is correct. If the *grasp* poses are chosen accordingly, each component can be placed correctly.

7.2.5 Belt Drive Unit interpretation

The plan for the *BDU* assembly example from Section 7.1.5 is interpreted for a dual-arm robotic system. The resulting list of skills in regard to the first two snapshots is presented in Figure 7.14, for the third snapshot in Figure 7.15 and for the fourth in Figure 7.16. The skill numbering is continuous from one table to the next.

The first eight skills of the first snapshot are related to each other, where a plate is *grasped* and *inserted* on the grounded component and is then *screwed* in place by two screws. The plate is held with the *left* arm while the each screw is *grasped* with the *right* arm, which is free after the first screw is in place. The inserted plate is *released* after the second screw is added, which means both grippers are empty an thus both arms are available. The same

#	Skill	Arm	Target	Other target	Action	Snapshot
1	Grasp	left	02-PLATE-01:1	-	1	1
2	Insert	left	01-BASE:1	02-PLATE-01:1	1	1
3,5	Grasp	right	SCB4-10:3-4	-	2-3	1
4,6	Screw	right	01-BASE:1	SCB4-10:3-4	2-3	1
7	Release	left	02-PLATE-01:1	-	3	1
8	Grasp	left	03-PLATE-02:1	-	4	1
9	Insert	left	01-BASE:1	03-PLATE-02:1	4	1
10,12	Grasp	right	SCB4-10:1-2	-	5-6	1
11,13	Screw	right	01-BASE:1	SCB4-10:1-2	5-6	1
14	Release	left	03-PLATE-02:1	-	6	1
15	Grasp	left	37d-gearmotor-50-70:1	-	7	1
16	Insert	left	02-PLATE-01:1	37d-gearmotor-50-70:1	7	1
17,19,. . . ,27	Grasp	right	SCB3-10:1-6	-	9-14	1
18,20,. . . ,28	Screw	right	37d-gearmotor-50-70:1	SCB3-10:1-6	9-14	1
29	Release	left	37d-gearmotor-50-70:1	-	14	1
30	Grasp	left	MBRFA30-2-P6_35:1	-	15	1
31	Insert	left	37d-gearmotor-50-70Shaft:1	MBRFA30-2-P6_35:1	15	1
32	Release	left	MBRFA30-2-P6_35:1	-	16	1
33	Grasp	left	SBARB6200ZZ-30:1	-	17	1
34	Insert	left	03-PLATE-02:1	SBARB6200ZZ-30:1	17	1
35,37,39,41	Grasp	right	SCB4-10:5-8	-	18-21	1
36,38,40,42	Screw	right	03-PLATE-02:1	SCB4-10:5-8	18-21	1
43	Release	left	SBARB6200ZZ-30:1	-	21	1
44	Grasp	left	MSSFS3-6:1	-	22+23	2
45	Screw	left	MBRFA30-2-P6_35:1	MSSFS3-6:1	22+23	2

Figure 7.14: List of skills of the *BDU* example's first two snapshots.

applies for the next eight skills, skills 8 to 14, with the only difference, that a different plate is being attached. Skills 15 to 29 are similar, because the *gearmotor* is *inserted* into one of the previously added plates and then *screwed* to it using six screws. The *gearmotor* is held by the *left* arm, while each screw is handled by the *right* arm. After the last screw is added, the *gearmotor* is *released* with skill 29. The disk *MBRFA30-2P6_35*, that is being attached to the *gearmotor shaft*, is *grasped* with skill 30 and *inserted* with skill 31. The only remaining relation of *MBRFA30-2P6_35*, that has not been handled yet, is a multi-contact relation with the *rubber band*. The multi-contact relation is skipped, because it contains *relatedConnections* that have not been reached yet. If a component does not have any following parts that need to be inserted, all currently held components can be released. Thus, disk *MBRFA30-2P6_35* is *released* with skill 32. Skills 33 to 43 represent an almost identical scenario as skills 15 to 29, where a component is *inserted* into one of the plates by one arm and *screwed* to it using four screws. All

#	Skill	Arm	Target	Other target	Action	Snapshot
46	Grasp	left	MBGA30-2:1	-	24	3
47	Grasp	right	BGPSL6-9-L30-F7:1	-	24	3
48	Insert	left	BGPSL6-9-L30-F7:1	MBGA30-2:1	24	3
49	Release	left	MBGA30-2:1	-	24	3
50	Grasp	left	CLBUS6-9-9_5:1	-	25	3
51	Insert	left	BGPSL6-9-L30-F7:1	CLBUS6-9-9_5:1	25	3
52	Release	left	CLBUS6-9-9_5:1	-	25	3
53	Grasp	left	SPWF6:2	-	26	3
54	Insert	left	BGPSL6-9-L30-F7:1	SPWF6:2	26	3
55	Release	left	SPWF6:2	-	26	3
56	Insert	right	03-PLATE-02:1	BGPSL6-9-L30-F7:1	27	3
57	Grasp	left	SPWF6:1	-	28	3
58	Insert	left	BGPSL6-9-L30-F7	SPWF6:1	28	3
59	Release	left	SPWF6:1	-	28	3
60	Grasp	left	SLBNR6:1	-	29	3
61	Screw	left	BGPSL6-9-L30-F7:1	SLBNR6:1	29	3
62	Release	right	BGPSL6-9-L30-F7:1	-	29	3
63	Grasp	left	MBRAC60-2-10_p1:1	-	30	3
64	Grasp	right	SSFHRT10-75-M4-FC55-G20_s:1	-	30	3
65	Insert	left	SSFHRT10-75-M4-FC55-G20_s:1	MBRAC60-2-10_p1:1	30	3
66	Release	left	MBRAC60-2-10_p1:1	-	30	3
67	Grasp	left	CLBPS10-17-4:1	-	31	3
68	Insert	left	SSFHRT10-75-M4-FC55-G20_s:1	CLBPS10-17-4:1	31	3
69	Release	left	CLBPS10-17-4:1	-	31	3
70	Insert	right	SBARB6200ZZ-30:1	SSFHRT10-75-M4-FC55-G20_s:1	32	3
71	Grasp	left	EDCS10:1	-	33	3
72	Insert	left	SSFHRT10-75-M4-FC55-G20_s:1	EDCS10:1	33	3
73	Release	left	EDCS10:1	-	33	3
74	Grasp	left	SCB4-10:9	-	34	3
75	Screw	left	SSFHRT10-75-M4-FC55-G20_s:1	SCB4-10:9	34	3
76	Release	right	SSFHRT10-75-M4-FC55-G20_s:1	-	34	3
77	Multi	-	MBT4-400:1	MBRAC60-2-10_p1:1	35	3

Figure 7.15: List of skills of the *BDU* example's third snapshot.

screws are handled by the other arm. The component is then *released* after the last screw was added.
The second snapshot contains two actions, an *undo* and a *redo* of the same two parts. If such a combination of actions appears consecutively, they are interpreted as a single action which describes a change of an already satisfied connection. If the two actions represent a *screw*, the new action is still executed using the *screw* skill. Otherwise, two consecutive *insert* actions, where the first represents an *undo*, are executed using the *shift* skill. This results in two skills for snapshot two, skills 44 and 45.

#	Skill	Arm	Target	Other target	Action	Snapshot
78	Grasp	left	SLBNR6:1	-	39	4
79	Grasp	right	BGPSL6-9-L30-F7:1	-	39	4
80	Unscrew	left	BGPSL6-9-L30-F7:1	SLBNR6:1	39	4
81	Release	left	SLBNR6:1	-	39	4
82	Shift	right	03-PLATE-02:1	BGPSL6-9-L30-F7:1	40+41	4
83	Grasp	left	SLBNR6:1	-	42	4
84	Screw	left	BGPSL6-9-L30-F7:1	SLBNR6:1	42	4
85	Release	right	BGPSL6-9-L30-F7:1	-	42	4
86	Grasp	left	M4x10p3:2	-	43+44	4
87	Screw	left	MBRAC60-2-10_p1:1	M4x10p3:2	43+44	4
88	Grasp	left	M4x10p3:1	-	45+46	4
89	Screw	left	MBRAC60-2-10_p1:1	M4x10p3:1	45+46	4

Figure 7.16: List of skills of the *BDU* example's fourth snapshot.

Snapshot three contains two subassemblies and a multi-contact component in form of a *rubber band*. The first subassembly contains skills 46 to 62 and the second reaches from skill 63 to 76. In both cases, the sequence is similar. The first two components of the subassembly are *grasped* by different arms, the child is *inserted* into the parent using the *left* arm and then *released*, because the arm is needed for the next skill. The next child of the subassembly is *grasped* by the *left* arm again, *inserted* onto the parent and *released*. This is repeated until the parent, currently in the *right* hand, needs to be connected to a component, that is already placed inside the assembly. This is the case for skill 56 for the first subassembly and skill 70 for the second. In these two cases, the parent is *inserted* into the already placed component, but not released yet. Then the subassemblies are continued as before, until the last *screwing* action of each subassembly is executed. After the screw or the nut is *screwed* to the parent, which is skill 61 for the first subassembly and 75 for the second, all components can be *released* by the arms, executed by skill 62 and 75 respectively, and the subassemblies are finished. The multi-contact component of action 35 has skill 77 dedicated to it as proof of concept, but is not actually executed, since the system does not support multi-contact relations yet.

The fourth and final snapshot contains three changes to the assembly. The first moving component is *BGPSL6-9-L30-F7*, which is a bolt that needs to be *shifted* along the oblong slot in *03-PLATE-02* to create tension on the *rubber band*. The *rubber band* has not been placed into the assembly, but the bolt can be *shifted* anyway. Skills 78 and 79 execute the *grasping* of the bolt, that needs to be moved, and the nut *SLBNR6*, that is holding it in place. The nut is *unscrewed* by skill 80, which means it has been loosened enough for the bolt to be moved. After *SLBNR6* is *released* in skill 81, the two consecutive actions 40 and 41 are interpreted as a single *shifting* of *BGPSL6-9-L30-F7* along *03-PLATE-02* in skill 82. Skill 83 executes *grasping* of the *SLBNR6* nut again and skill 84 *screws* the nut fully to the *BGPSL6-9-L30-F7* again, which can now be released, as represented in skill 85. Currently, both grippers are empty. The skill pair 86 and 87 an the next pair, 88 and 89, both represent a *M4x10p3* screw that needs to be fastened onto *MBRAC60-2-10_p1*. Each screw is *grasped* and *screwed* further into the same parent component. The grippers are empty again at the end and no explicit *release* skill is needed.

All components are grasped when they need to be and the execution order is correct. The *grasp* poses have been chosen accordingly and thus, no arm is blocking the other.

7.3 Transformation evaluation

The valid execution of an individual action depends on the transformation chain used to calculate grasp and insert locations of components in the actual scene. This chain consists of the absolute locations of parent and child of the action's connection, each components transformation from its center of mass to the grasp pose and the relative transformation between both components. For each action this data is used to calculate specifically the final position and orientation of each robotic arm's end-effector. This spatial information, which represents the gripper's pose when finished grasping or inserting into the target of the current action, is used as the goal description for *MoveIt!*. There are two separate methods of calculating the final position and orientation of the end-effector: (1) grasping components and (2) inserting or screwing components into other parts.

Equation 7.1 describes the calculations for the final end-effector pose **E** inside the world coordinate system. The transformations in use are visualized in Figure 7.17. The transformation **T** describes the location of the target component inside the simulation space. The target's grasping pose is

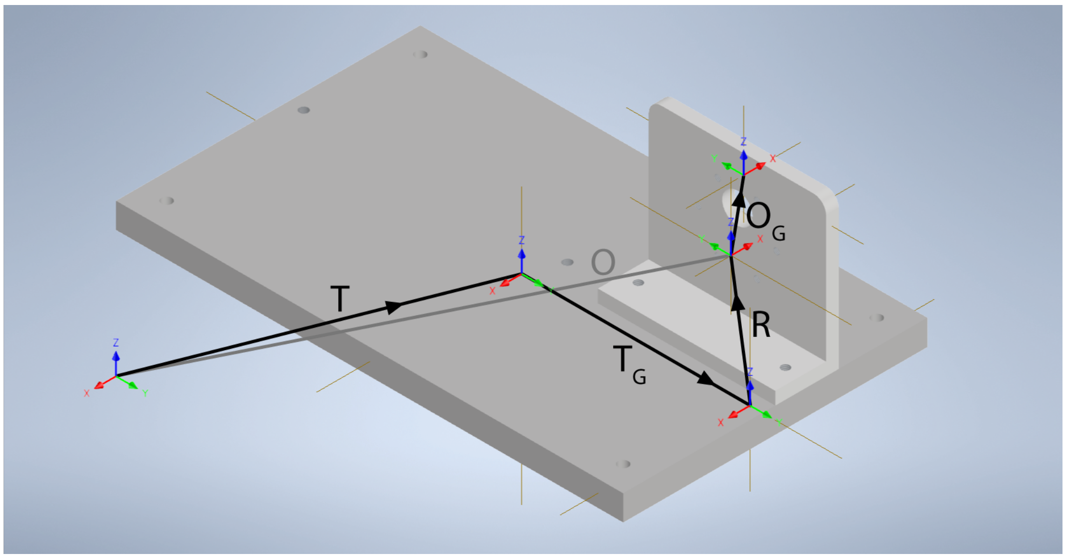


Figure 7.17: Geometrical representation of the transformation chain described in Equation 7.2.

defined by the transformation $\mathbf{T}_G$, which is relative to $\mathbf{T}$. The transformation $\mathbf{O}$ describes the location of the other target component inside the simulation space.
If the current skill involves two components, like *insert* or *screw*, then the other components grasping pose is relevant. It is described by the transformation $\mathbf{O}_G$, which is relative to $\mathbf{O}$. The relative transformation between the two components needs also to be included, which is defined by $\mathbf{R}$. This scenario is described by Equation 7.2.
In both cases, an additional fixed rotation $\mathbf{F}$ needs to be applied, which represents the assumption posed during the guidelines, that the default insertion direction is along the negative Z axis of the world coordinate system. A quaternion representing such a transformation is shown in Equation 7.3.

$$\mathbf{E} = \mathbf{T}_G \cdot \mathbf{T} \cdot \mathbf{F} \tag{7.1}$$

$$\mathbf{E} = \mathbf{O}_G \cdot \mathbf{R} \cdot \mathbf{T}_G \cdot \mathbf{T} \cdot \mathbf{F} \tag{7.2}$$

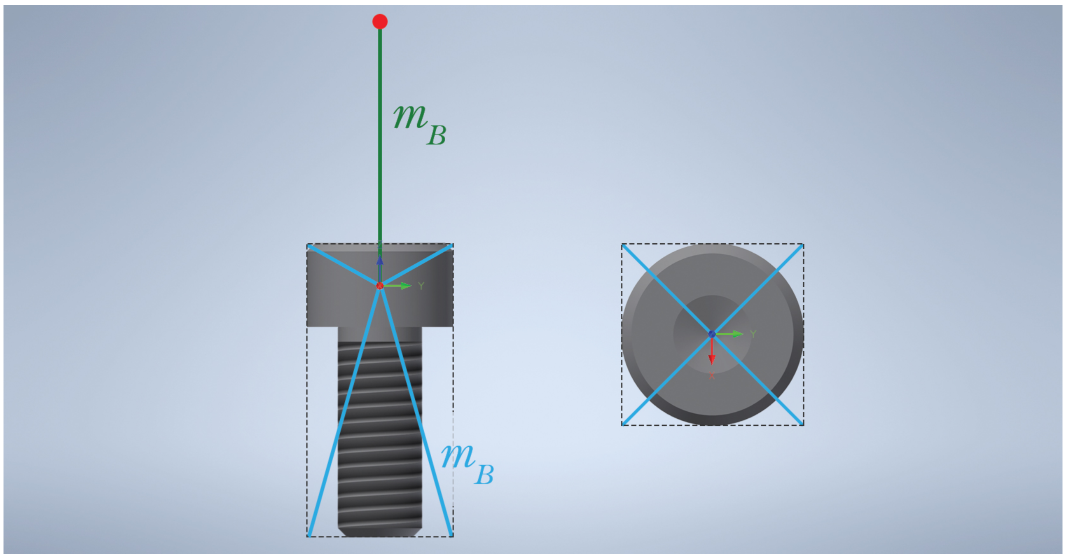


Figure 7.18: Geometrical representation of the distant pose calculation for approaching a component with the gripper oriented correctly.

$$\mathbf{F} = \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0.7071067811865474 6 \\ 0 \\ 0.70710678118654746 \\ 0 \end{bmatrix} \tag{7.3}$$

Most skills, such as *grasp*, *insert* or *screw*, demand that the involved gripper approaches the final location with a constant orientation along a specific axis. To achieve this, a pre-contact location must be determined, from which the final pose can be accessed while keeping a constant orientation. Figure 7.18 shows the calculation of such a distant pose. First, the end-effector approaches a location outside of the component's bounding box, that is also along the insertion axis of the action. This position can be reached without constraining the gripper orientation during the movement of the arm, as long as the gripper's final orientation at the pre-contact location is defined as facing towards the exact destination. Equation 7.4 shows how the distant position $\mathbf{P_D}$ outside of the bounding box is calculated. The position $\mathbf{P}_E$ and orientation $\mathbf{O}_E$ come from the transformation $\mathbf{E}$ shown in either Equation 7.1 or Equation 7.2. m_B describes the largest distance from the center of the bounding box to one of the corners. The calculation for the normalized direction vector $\hat{\mathbf{D}}$ is shown in Equation 7.5. The direction vector $\mathbf{D}$ is calculated using Equation 7.6, which describes the rotation of $\mathbf{O}_E$ by $\mathbf{Q}$. $\mathbf{Q}$ is a *pure* quaternion, which represents a rotation in 3D space, since the w component is always zero. Since $\mathbf{D}$ represents the inverted direction vector of $\mathbf{O}_E$, the value chosen for $\mathbf{Q}$ is constant and can be seen in Equation 7.8. The function $\mathbf{k}$ in Equation 7.7 is used to remove the unwanted first dimension of a quaternion to convert it to a three-dimensional vector.

$$\mathbf{P}_{\mathbf{D}} = \mathbf{P}_E + m_B \cdot \hat{\mathbf{D}} \tag{7.4}$$

$$\hat{\mathbf{D}} = \frac{\mathbf{D}}{||\mathbf{D}||} \tag{7.5}$$

$$\mathbf{D} = \mathbf{k}(\mathbf{Q}^* \cdot \mathbf{O}_E \cdot \mathbf{Q}) \tag{7.6}$$

$$\mathbf{k} : \begin{cases} \mathbb{R}^4 \to \mathbb{R}^3 \\ \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} \mapsto \begin{bmatrix} x \\ y \\ z \end{bmatrix} \end{cases} \tag{7.7}$$

$$\mathbf{Q} = \begin{bmatrix} w \overset{!}{=} 0 \\ x \\ y \\ z \end{bmatrix} \overset{!}{=} \begin{bmatrix} 0 \\ -1 \\ 0 \\ 0 \end{bmatrix} \tag{7.8}$$

After the pre-contact destination is reached successfully, the second motion primitive is activated, which is constraint by the gripper's current orientation. During this movement, the gripper travels along the insertion axis to the final, exact location while always facing the same direction.

Additional constraints for some arm joints are needed to prevent the robot from planning a trajectory for the distant position, that would lead to a collision between the elbow of the robotic arm and the ground when trying to reach the exact destination. For this, constraints for the first joint, the shoulder, and third joint, the elbow, are introduced. They only allow movement in half of the available joint range. It is not enough to only

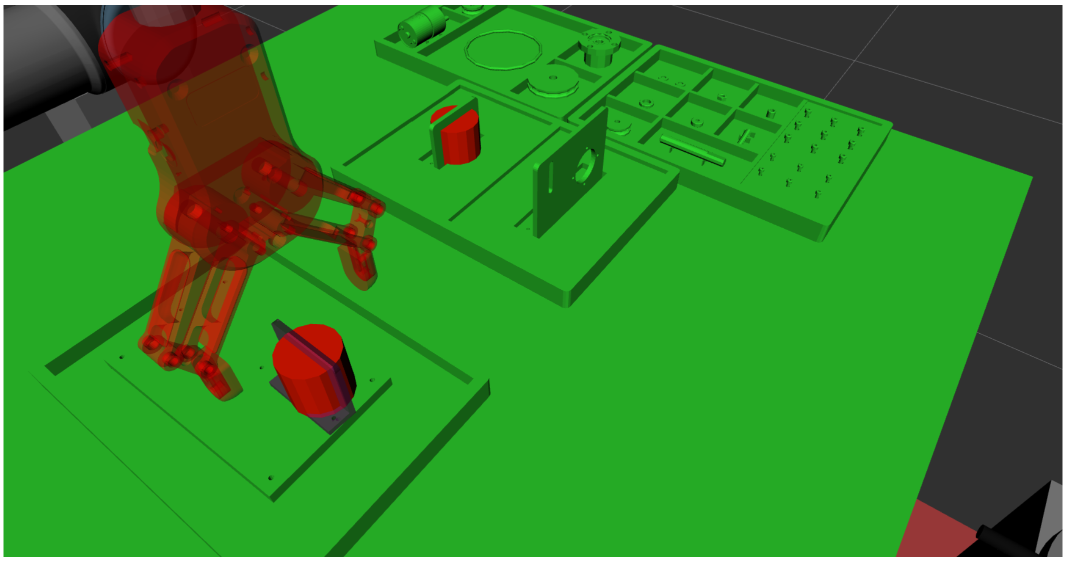


Figure 7.19: Screenshot of the execution of a single action inside the simulation, containing a component being grasped and placed. The red pillars are big arrows facing the ground representing grasping poses. The green component in such a red arrow is at the original position in the kitting box and the violet part with the gripper above is at the final location of the component placement.

constrain the elbow, because, if the shoulder joint rotates by 180°, the elbow joint constraints would need to be flipped around.

Figure 7.19 shows a screenshot of the simulation, where the first component of the assembly is being grasped and placed on top of the grounded component. The red pillars are big arrows facing the ground and represent the initial and final grasping poses of the gripper, published to rViz. The green component in such a red arrow is at the original position in the kitting box and the violet part, with the gripper above, is at the final location of the component placement. The pose of the violet component represents the correct final pose of the part in regard to the grounded component, which shows the correct calculation of transformation and position for the action.

8 Conclusion and Future Work

The overall goal of this thesis was to develop a proof of concept for a system that is able to generate a full assembly plan out of an initial representation of the assembly task with the least amount of user involvement using a general dual-arm robot setup. The work mainly focused on geometrical and semantical understanding of the assembly task and the involved components, deriving an assembly plan using inferred knowledge, subdividing it into sequences of actions and autonomously and robustly executing the planned actions based on the actual configuration in the real environment. The utilized perception system, comprised of visual sensors, was intended to be able to identify and precisely localize the involved components. Since it was intended to need as few as possible additional prior information about the task, the utmost amount of information was taken from the initial representation.
A methodology for the extraction of action sequences for automated assembly tasks was presented. It contains a set of guidelines and rules to prepare and combine individual components in order to build a *CAD* representation of the assembly task. The imposed guidelines and rules are easy to be used by enginners and do not imply more effort than normally once accustomed to. A directed graph-like structure of components as nodes and connections as edges is deduced from the *CAD* model. Using the developed set of algorithms described in this thesis, the hierarchical information in from of said graph is then processed to obtain a sequential plan of actions. Each action is represented by a connection between two individual components of the assembly. These directed connections contain additional information about the related parts in their final position, such as a relative transformation between parent and child or the actual distance between the related surfaces. Furthermore, special constraints in form of snapshots are added to enforce a specific sequence of action inside the plan. This can be used by the *CAD* modeler to enforce a wanted assembling order or to denote the need of

moving specific already mounted components after a certain point in time during the assembly. An additional group of algorithms is presented in this thesis, which represents a possible interpretation of such a plan for a dual-arm robotic setup. This setup consists of two combined UR5 robotic arms with general-purpose parallel two-finger RG6 grippers as end-effector of each arm. The presented scenario is executed inside a Gazebo simulation using *ROS* Kinetic as a robot software framework. The *MoveIt!* framework was used for motion planning and execution for the arms and grippers. The plan interpretation leads to a series of modular and predefined skills and primitives, developed during this thesis. The skills and primitives are implemented using a skill-based *ROS* platform called *SkiROS*. It enables autonomous mission execution based on autonomous, goal directed task planning and knowledge integration.
The *CAD* environment of choice, in order to model an assembly for this work, is Inventor. The presented algorithms for the extraction of a sequential list of actions were implemented in form of an Inventor plugin. This plugin allows the user to create snapshots of the current state of the assembly. Multiple snapshots allow the user to enforce a specific temporal order, if necessary. The only external input for the algorithms consists of such a collection of snapshots. The exported representation of the sequential list of actions is added as input in form of an ontology to *SkiROS*, enabling the skills and primitives implemented based on *SkiROS* to access the plan and all its collected information. For this reason, the second set of algorithms were implemented in form of such skills interacting with the simulated environment.
The initial state of the assembly needed to be spawned inside the simulation. For this purpose, the functionality of the Inventor plugin was expanded to include the ability of analyzing such an initial representation of the assembly. This allows identifying all components' physical properties, such as mass, center of mass or inertial parameters, and exporting them together with the triangle mesh of each part as a *ROS* and Gazebo conform model. Two additional files can be exported, which contain the commands to spawn the correct amount of instances for each part model and the commands for linking components spawned with pre-satisfied connections.

The evaluation process of this thesis was focused on the correct interpretation of each plan and the feasibility of each skill. The main evaluation example was inspired by the Assembly Challenge from the World Robot Summit 2018 and consists of a Belt Drive Unit (*BDU*) with many components, two subassemblies, some moving parts and a deformable objects. The evaluation of the presented proof of concept was also comprised of less complex examples to facilitate the understanding and the evaluation of the *BDU* example. The *ROS* services *attach* and *detach*, developed to move simulated components inside the simulation using a robot arm, contain an unidentified issue that went beyond the scope of this work and prevented the full sequence of actions to be executed inside the simulation. Thus, the process of evaluation is of a quantitative nature and was executed by domain experts. The considered aspects are, whether arms are always empty before they are reused, whether an arm blocks the access for the next arm or whether any skills happen in the wrong order. The evaluation shows, that if all guidelines and rules are followed as described in this thesis, the assembly process does not contain problems. The three major aspects of the presented methodology, that entail a heavy workload and represent the most common source of mistakes in following the guidelines, are finding suitable grasp poses, combining components using the proper connection type, surface and hierarchical order and combining the right components with each other. During a future expansion, ideally, these decisions would be automated, to further eliminate user workload and possible sources of errors.

During development a few specific limitations were encountered. The first issue was found in the *URDF* format in form of unsupported closed loop chains of joints. This problem is known and as such, solutions in form of *ROS* packages and Gazebo plugins exist. This project used the *MimicJointPlugin*[1] to circumvent this problem. An alternative solution, released more recently, is a *ROS* package called *closed_loop_plugin*[2]. Both solutions change the robot model after it is converted to the *SDF* format, which is generated from the *URDF* model. The remaining issue is, that the *URDF* model is used by rViz to display the robot. This leads to a discrepancy of the passive gripper joints between Gazebo and rViz. That difference can cause issues, since the

[1] https://github.com/roboticsgroup/roboticsgroup_gazebo_plugins

[2] https://wiki.ros.org/Angel_jj/closed_loop_plugin

trajectory planning of *MoveIt!* gets its information from the planning_scene, which also uses the *URDF* model of the robot. During the creation of the movement primitives in *SkiROS*, major flaws in the connection to *MoveIt!* became apparent. This led to research concluding that the initial version of *SkiROS* presented in [3] was deprecated and a newer version, *SkiROS2*[3], was published. To prevent the *SkiROS* skill manager from crashing, an action server that connects to the *MoveIt!* services was implemented. This was only possible with the newer version of *SkiROS*.

Additionally, there are general problems with the simulation, such as handling small, detailed or circular components with Gazebo. On collision, components accelerate drastically and fly out of the scene. This is a prevalent issue, since most components are made to fit tightly into each other. It can be overcome by creating un-movable links between two connected components and all its colliding parts.
Because of an unidentified reason the execution of the simulation takes a random amount of attempts until it is able to connect to the execution_trajectory of *MoveIt!*. This sometimes prevents the system from connecting to the move_group, which causes the execution to either fail or jump to the next action while still executing a previous one. More research needs to be done to identify the cause of this problem.

The major issue preventing the simulation to function successfully, which was not yet overcome, is transporting components by attaching them to the gripper. This problem can and will be overcome with some additional work, but was put aside, because it went beyond the scope of this project.
Gazebo does not support deformable objects, so no method for multi-contact components, such as the rubber band from the *BDU* example, was introduced yet and will be part of future expansions for real environments.
The concept of nested assemblies, which came up during the evaluation phase of this work and is currently handled by separating the workflow into two different assemblies, will also be further expanded upon to increase the usability of the system.
The handling of screws was simplified during this work and needs either future improvements of the skills or even a new, different tool than the universal two finger grippers.

[3]`https://github.com/Bjarne-AAU/skiros-demo`

Future expansions may also include an automatized identification of connections between components without needing to rely on the joints and constraints of the *CAD* file. This would eliminate user guidelines for the model creation and further automate the plan extraction, including the necessity for consideration of gravity for arm actions. The addition of grasp evaluation could eliminate the need for user defined grasping poses, allowing for further user workload reduction.
Currently, a set of presented algorithms is used to deduce the individual group of skills needed for each action, depending on previous and future actions. This plan interpretation is only valid for dual-arm systems. A future development goal of *SkiROS* in form of a more sophisticated pre- and post-condition evaluation, could automate this process for different kinds of robots and setups.

The stability of the simulation could also be further improved by addition of concurrent and continuous state estimation of scene components. The integration in a execution monitoring would increase the reactiveness and safety of such a system for real life environments.

Community wide sharing of motion primitives for standardized architectures, adaptable to different kinds of hardware, would decrease future development workload even further and allow for standardized research and uniform refinement of standards in robotics.

Appendix

Bibliography

[1] Johannes Kurth. “Human-Robot Collaboration makes companies more profitable.” In: *LinkedIn* (Aug. 2019). URL: https://www.linkedin.com/pulse/human-robot-collaboration-makes-companies-more-profitable-kurth (cit. on pp. 2, 4).

[2] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. “ROS: an open-source Robot Operating System.” In: *ICRA Workshop on Open Source Software*. Vol. 3. Jan. 2009 (cit. on pp. 8–10).

[3] Francesco Rovida, Matthew Crosby, Dirk Holz, Athanasios S. Polydoros, Bjarne Großmann, Ronald P. A. Petrick, and Volker Krüger. “SkiROS—A Skill-Based Robot Control Platform on Top of ROS.” In: *Robot Operating System (ROS): The Complete Reference (Volume 2)*. Ed. by Anis Koubaa. Cham: Springer International Publishing, 2017, pp. 121–160. ISBN: 978-3-319-54927-9. DOI: 10.1007/978-3-319-54927-9_4. URL: https://doi.org/10.1007/978-3-319-54927-9_4 (cit. on pp. 8, 14–16, 91, 152).

[4] Sachin Chitta, Ioan Sucan, and Steve B. Cousins. “MoveIt! [ROS Topics].” In: *IEEE Robotics Automation Magazine* 19.1 (Mar. 2012), pp. 18–19. ISSN: 1558-223X. DOI: 10.1109/MRA.2011.2181749 (cit. on pp. 12, 32).

[5] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. “PDDL - The Planning Domain Definition Language.” In: *Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control* (Aug. 1998) (cit. on pp. 16, 31).

[6] Francesco Rovida, Bjarne Grossmann, and Volker Kruger. “Extended behavior trees for quick definition of flexible robotic tasks.” In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2017, pp. 6793–6800. DOI: 10.1109/IROS.2017.8206598 (cit. on pp. 16, 33).

[7] Kerstin Dautenhahn. “Methodology & Themes of Human-Robot Interaction: A Growing Research Field.” In: *International Journal of Advanced Robotic Systems* 4 (Mar. 2007). DOI: 10.5772/5702 (cit. on p. 25).

[8] Sharath Akkaladevi, Matthias Plasch, Andreas Pichler, and Bernhard Rinner. “Human Robot Collaboration to Reach a Common Goal in an Assembly Process.” In: *STAIRS*. July 2016. DOI: 10.3233/978-1-61499-682-8-3 (cit. on p. 25).

[9] Behzad Sadrfaridpour, Hamed Saeidi, and Yue Wang. “An integrated framework for human-robot collaborative assembly in hybrid manufacturing cells.” In: *2016 IEEE International Conference on Automation Science and Engineering (CASE)*. Aug. 2016, pp. 462–467. DOI: 10.1109/COASE.2016.7743441 (cit. on p. 26).

[10] S. M. M. Rahman, Y. Wang, I. D. Walker, L. Mears, R. Pak, and S. Remy. “Trust-based compliant robot-human handovers of payloads in collaborative assembly in flexible manufacturing.” In: *2016 IEEE International Conference on Automation Science and Engineering (CASE)*. Aug. 2016, pp. 355–360. DOI: 10.1109/COASE.2016.7743428 (cit. on p. 26).

[11] Arash Ajoudani, Andrea Maria Zanchettin, Serena Ivaldi, Alin Albu-SchÃ¤ffer, Kazuhiro Kosuge, and Oussama Khatib. “Progress and Prospects of the Human-Robot Collaboration.” In: *Autonomous Robots* (Oct. 2017). DOI: 10.1007/s10514-017-9677-2 (cit. on p. 26).

[12] Li Fei-Fei. “Generative Models for Visual Objects and Object Recognition via Bayesian Inference.” In: *Autumn School 2006: Machine Learning over Text and Images - Pittsburgh* (Sept. 2006). URL: http://videolectures.net/mlas06_li_gmvoo/?q=fei-fei (cit. on p. 26).

[13] Derek Hoiem and Silvio Savarese. “Representations and Techniques for 3D Object Recognition and Scene Interpretation.” In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 5.5 (2011), pp. 1–

169. DOI: https://doi.org/10.2200/S00370ED1V01Y201107AIM015 (cit. on pp. 26, 27).

[14] Xiaozhi Chen, Huimin Ma, Ji Wan, Bo Li, and Tian Xia. "Multi-view 3D Object Detection Network for Autonomous Driving." In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. July 2017, pp. 6526–6534. DOI: 10.1109/CVPR.2017.691 (cit. on p. 27).

[15] Pushmeet Kohli Nathan Silberman Derek Hoiem and Rob Fergus. "Indoor Segmentation and Support Inference from RGBD Images." In: *ECCV*. 2012 (cit. on p. 27).

[16] Øystein Skotheim, Morten Lind, Pål Ystgaard, and Sigurd Aksnes Fjerdingen. "A flexible 3D object localization system for industrial part handling." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2012, pp. 3326–3333. DOI: 10.1109/IROS.2012.6385508 (cit. on p. 27).

[17] Yu Xiang and Silvio Savarese. "Object Detection by 3D Aspectlets and Occlusion Reasoning." In: *Proceedings of the IEEE International Conference on Computer Vision Workshops*. Dec. 2013, pp. 530–537. DOI: 10.1109/ICCVW.2013.75 (cit. on p. 27).

[18] Hidefumi Wakamatsu, Masayuki Aoki, Eiji Morinaga, Eiji Arai, and Shinichi Hirai. "Property identification of a deformable belt object from its static images toward its manipulation." In: *2012 IEEE International Conference on Automation Science and Engineering (CASE)*. Aug. 2012, pp. 448–453. DOI: 10.1109/CoASE.2012.6386360 (cit. on p. 27).

[19] Zeeshan Zia, Michael Stark, and Bernt Schiele. "Detailed 3D Representations for Object Recognition and Modeling." In: *IEEE transactions on pattern analysis and machine intelligence* 35 (Nov. 2013), pp. 2608–23. DOI: 10.1109/TPAMI.2013.87 (cit. on p. 27).

[20] Mohsen Hejrati and Deva Ramanan. "Analyzing 3D Objects in Cluttered Images." In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 593–601. URL: http://papers.nips.cc/paper/4680-analyzing-3d-objects-in-cluttered-images.pdf (cit. on p. 27).

[21] Yu Xiang and Silvio Savarese. "Estimating the aspect layout of object categories." In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 3410–3417 (cit. on p. 27).

[22] Zeeshan Zia and Michael Stark. "Towards Scene Understanding with Detailed 3D Object Representations." In: *International Journal of Computer Vision* 112 (Nov. 2014). DOI: 10.1007/s11263-014-0780-y (cit. on p. 27).

[23] Luca Del Pero, Joshua Bowdish, Bonnie Kermgard, Emily Hartley, and Kobus Barnard. "Understanding Bayesian Rooms Using Composite 3D Object Models." In: *Proceedings / CVPR, IEEE Computer Society Conference on Computer Vision and Pattern Recognition. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. June 2013, pp. 153–160. DOI: 10.1109/CVPR.2013.27 (cit. on p. 27).

[24] Sukhan Lee, Jaewoong Kim, Moonju Lee, Kyeongdae Yoo, Leandro G. Barajas, and Roland Menassa. "3D visual perception system for bin picking in automotive sub-assembly automation." In: *2012 IEEE International Conference on Automation Science and Engineering (CASE)*. Aug. 2012, pp. 706–713. DOI: 10.1109/CoASE.2012.6386359 (cit. on p. 27).

[25] Chunlei Li, Chris McMahon, and Linda Newnes. "Progress with OntoCAD: A Standardised Ontological Annotation Approach to CAD Systems." In: *International Conference on Product Lifecycle Management (PLM), Eindhoven, Netherlands*. 2011 (cit. on p. 28).

[26] Ramos Luis Enrique García. "Ontological CAD Data Interoperability Framework." In: *SEMAPRO 2010* (Mar. 2020) (cit. on p. 28).

[27] Sean Tessier and Yan Wang. "Ontology-based feature mapping and verification between CAD systems." In: *Advanced Engineering Informatics* 27 (Jan. 2013), 76â€“92. DOI: 10.1016/j.aei.2012.11.008 (cit. on p. 28).

[28] Alexander Perzylo, Nikhil Somani, Markus Rickert, and Alois Knoll. "An Ontology for CAD Data and Geometric Constraints as a Link Between Product Models and Semantic Robot Task Descriptions." In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Sept. 2015, pp. 4197–4203. DOI: 10.1109/IROS.2015.7353971 (cit. on p. 28).

[29] Stefan Schaal. "Is imitation learning the route to humanoid robots?" In: *Trends in Cognitive Sciences* 3 (1999), pp. 233–242 (cit. on p. 29).

[30] Brenna Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. "A survey of robot learning from demonstration." In: *Robotics and Autonomous Systems* 57 (May 2009), pp. 469–483. DOI: 10.1016/j.robot.2008.10.024 (cit. on pp. 29, 30).

[31] S. Calinon, F. Guenter, and A. Billard. "On Learning, Representing and Generalizing a Task in a Humanoid Robot." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 37.2 (Apr. 2007), pp. 286–298. ISSN: 1941-0492. DOI: 10.1109/TSMCB.2006.886952 (cit. on p. 29).

[32] Harold Bekkering, Andreas Wohlschläger, and Merideth Gattis. "Imitation of Gestures in Children is Goal-directed." In: *The Quarterly Journal of Experimental Psychology: Section A* 53 (Mar. 2000), pp. 153–64. DOI: 10.1080/713755872 (cit. on p. 29).

[33] Sylvain Calinon, Florent Guenter, and Aude Billard. "Goal-Directed Imitation in a Humanoid Robot." In: *Proceedings - IEEE International Conference on Robotics and Automation*. Vol. 2005. Jan. 2005, pp. 299–304. DOI: 10.1109/ROBOT.2005.1570135 (cit. on p. 29).

[34] Stefan Schaal, Jan Peters, Jun Nakanishi, and Auke Ijspeert. "Control, planning, learning, and imitation with dynamic movement primitives." In: *Workshop on Bilateral Paradigms on Humans and Humanoids: IEEE International Conference on Intelligent Robots and Systems (IROS 2003)*. Jan. 2003, pp. 1–21 (cit. on p. 29).

[35] Stefan Schaal. "Dynamic Movement Primitives: Framework for Motor Control in Humans and Humanoid Robotics." In: *Adaptive Motion of Animals and Machines* (Jan. 2006). DOI: 10.1007/4-431-31381-8_23 (cit. on p. 29).

[36] A.J. Ijspeert, Jun Nakanishi, and Stefan Schaal. "Movement imitation with nonlinear dynamical systems in humanoid robots." In: *Proceedings - IEEE International Conference on Robotics and Automation*. Vol. 2. Feb. 2002, pp. 1398–1403. ISBN: 0-7803-7272-7. DOI: 10.1109/ROBOT.2002.1014739 (cit. on p. 29).

[37] A.J. Ijspeert, Jun Nakanishi, and Stefan Schaal. “Learning Attractor Landscapes for Learning Motor Primitives.” In: *Advances in Neural Information Processing Systems*. Vol. 15. Jan. 2002, pp. 1523–1530 (cit. on p. 29).

[38] Dae-Hyung Park, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. “Movement reproduction and obstacle avoidance with dynamic movement primitives and potential fields.” In: *Humanoids 2008 - 8th IEEE-RAS International Conference on Humanoid Robots*. Dec. 2008, pp. 91–98. DOI: 10.1109/ICHR.2008.4755937 (cit. on p. 30).

[39] Bruce Krogh. “A Generalized Potential Field Approach to Obstacle Avoidance Control.” In: *Proc. SME Conf. on Robotics Research: The Next Five Years and Beyond, Bethlehem, PA, 1984*. 1984, pp. 11–22 (cit. on p. 30).

[40] Johann Borenstein and Yoram Koren. “Real-Time Obstacle Avoidance for Fast Mobile Robots.” In: *Systems, Man and Cybernetics, IEEE Transactions on* 19 (Oct. 1989), pp. 1179–1187. DOI: 10.1109/21.44033 (cit. on p. 30).

[41] Alexandros Paraschos, Christian Daniel, Jan R Peters, and Gerhard Neumann. “Probabilistic Movement Primitives.” In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger. Curran Associates, Inc., 2013, pp. 2616–2624. URL: http://papers.nips.cc/paper/5177-probabilistic-movement-primitives.pdf (cit. on p. 30).

[42] Miha Deniša, Andrej Gams, Aleš Ude, and Tadej Petrič. “Learning Compliant Movement Primitives Through Demonstration and Statistical Generalization.” In: *IEEE/ASME Transactions on Mechatronics* 21.5 (Oct. 2016), pp. 2581–2594. ISSN: 1083-4435. DOI: 10.1109/TMECH.2015.2510165 (cit. on p. 30).

[43] Xiang Zhang, Athanasios S. Polydoros, and Justus H. Piater. “Learning Movement Assessment Primitives for Force Interaction Skills.” In: *CoRR* abs/1805.04354 (2018). arXiv: 1805.04354. URL: http://arxiv.org/abs/1805.04354 (cit. on p. 30).

[44] Richard E. Fikes and Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. IJCAI'71. London, England: Morgan Kaufmann Publishers Inc., 1971, pp. 608–620 (cit. on p. 31).

[45] Edwin P. D. Pednault. "ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus." In: *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 324–332. ISBN: 1558600329 (cit. on p. 31).

[46] J. Scott Penberthy and Daniel S. Weld. "UCPOP: A Sound, Complete, Partial Order Planner for ADL." In: *KR*. 1992 (cit. on p. 31).

[47] Kei Okada, Yohei Kakiuchi, Haseru Azuma, Hiroyuki Mikita, Kazuto Murase, and Masayuki Inaba. "Task compiler : Transferring high-level task description to behavior state machine with failure recovery mechanism." In: *In IEEE International Conference on Robotic and Automation*. May 2013 (cit. on p. 31).

[48] Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcís Palomeras, Natàlia Hurtós, and Marc Carreras. "Rosplan: Planning in the robot operating system." In: *Proceedings International Conference on Automated Planning and Scheduling, ICAPS* 2015 (Jan. 2015), pp. 333–341 (cit. on p. 32).

[49] Hai Nguyen, Matei Ciocarlie, Kaijen Hsiao, and Charles Kemp. "ROS commander (ROSCo): Behavior creation for home robots." In: *Proceedings - IEEE International Conference on Robotics and Automation*. May 2013, pp. 467–474. ISBN: 978-1-4673-5641-1. DOI: 10.1109/ICRA.2013.6630616 (cit. on p. 32).

[50] Francesco Rodiva, Casper Schou, Rasmus Andersen, Jens Damgaard, Dimitrios Chrysostomou, Simon Bøgh, Mikkel Pedersen, Bjarne Grossmann, Ole Madsen, and Volker Krüger. "SkiROS: A four tiered architecture for task-level programming of industrial mobile manipulators." In: *International Workshop on ROS-Industrial in European Research Projects*. July 2014 (cit. on p. 33).

[51] Ching-Yen Weng, Wei Tan, and I-Ming Chen. “A Survey of Dual-Arm Robotic Issues on Assembly Tasks.” In: *CISM International Centre for Mechanical Sciences, Courses and Lectures* (Jan. 2019), pp. 474–480. DOI: 10.1007/978-3-319-78963-7_59 (cit. on p. 33).

[52] Dragoljub Surdilovic, Y. Yakut, T.-M Nguyen, X.B. Pham, Axel Vick, and Roberto Martin-Martin. “Compliance control with dual-arm humanoid robots: Design, planning and programming.” In: Jan. 2011, pp. 275–281. DOI: 10.1109/ICHR.2010.5686273 (cit. on p. 33).

[53] Sonny Tarbouriech, Benjamin Navarro, Philippe Fraisse, André Crosnier, Andrea Cherubini, and Damien Salle. “Dual-arm relative tasks performance using sparse kinematic control.” In: *IROS 2018*. Oct. 2018. DOI: 10.1109/IROS.2018.8594320 (cit. on p. 34).

[54] Chien-Pin Chen. “Developing Industrial Dual Arm Robot for Flexible Assembly Through Reachability Map.” In: *Precision Machinery Research and Development Center* (June 2015). DOI: 10.13140/RG.2.2.16688.56321 (cit. on p. 34).

[55] Ulrike Thomas, Theodoros Stouraitis, and M. A. Roa. “Flexible assembly through integrated assembly sequence planning and grasp planning.” In: *2015 IEEE International Conference on Automation Science and Engineering (CASE)*. Aug. 2015, pp. 586–592. DOI: 10.1109/CoASE.2015.7294142 (cit. on p. 35).

[56] Ulrike Thomas and Friedrich M. Wahl. “Assembly Planning and Task Planning - Two Prerequisites for Automated Robot Programming.” In: vol. 67. Nov. 2010. ISBN: 978-3-642-16784-3. DOI: 10.1007/978-3-642-16785-0_19 (cit. on p. 35).

[57] Ulrike Thomas, Mark Barrenscheen, and Friedrich M. Wahl. “Efficient assembly sequence planning using stereographical projections of C-Space obstacles.” In: vol. 2003. Aug. 2003, pp. 96–102. ISBN: 0-7803-7770-2. DOI: 10.1109/ISATP.2003.1217194 (cit. on p. 35).

[58] Markus Ikeda, Srinivas Maddukuri, Michael Hofmann, Andreas Pichler, Xiang Zhang, Athanasios Polydoros, Justus Piater, Klemens Winkler, Klaus Brenner, Ioan Harton, and Uwe Neugebauer. “FlexRoP - flexible, assistive robots for customized production.” In: *Austrian Robotics Workshop 2018*. July 2018, pp. 53–58. DOI: 10.15203/3187-22-1-11 (cit. on pp. 35, 36).

[59] Sharath Akkaladevi, Matthias Plasch, and Andreas Pichler. “Skill-based learning of an assembly process.” In: *e & i Elektrotechnik und Informationstechnik* 134 (Sept. 2017). DOI: 10.1007/s00502-017-0514-2 (cit. on p. 35).

[60] The Industrial Robotics Competition Committee. “Industrial Robotics Category - Assembly Challenge.” In: *Rules and Regulations 2018*. World Robot Summit, 2018. URL: https://worldrobotsummit.org/download/rulebook-en/rulebook-Assembly_Challenge.pdf (cit. on pp. 39, 40).

[61] Yasuyoshi Yokokohji, Yoshihiro Kawai, Mizuho Shibata, Yasumichi Aiyama, Shinya Kotosaka, Wataru Uemura, Akio Noda, Hiroki Dobashi, Takeshi Sakaguchi, and Kazuhito Yokoi. “Assembly Challenge: a robot competition of the Industrial Robotics Category, World Robot Summit - summary of the pre-competition in 2018.” In: *Advanced Robotics* 33.17 (2019), pp. 876–899. DOI: 10.1080/01691864.2019.1663609. eprint: https://doi.org/10.1080/01691864.2019.1663609. URL: https://doi.org/10.1080/01691864.2019.1663609 (cit. on p. 39).