School of Information Technologies

Department of Computer Systems

Yevhen Bondarenko

Development of Digital Twin-based Learning from Demonstration System for Industrial Robots

Master's Thesis

Supervisors: Vladimir Kuts
Ph.D.

Eduard Petlenkov
Ph.D.

TALLINN 2021

Declaration of Originality

Declaration: I hereby declare that this thesis, my original investigation and achievement, submitted for the Master's degree at Tallinn University of Technology, has not been submitted for any degree or examination.

Deklareerin, et käesolev diplomitöö, mis on minu iseseisva töö tulemus, on esitatud Tallinna Tehnikaülikooli magistrikraadi taotlemiseks ja selle alusel ei ole varem taotletud akadeemilist kraadi.

Yevhen Bondarenko

Date: May 19, 2021

Abstract

Learning from Demonstration (LfD) is an approach to robot programming where the machine aims to replicate the task presented by a human without being explicitly programmed to execute this task. While being an effective way to create complex robot routines even for users without coding skills, most LfD implementations heavily rely on sensors for the robot to capture the state of the surrounding world and the task being demonstrated. This work presents an alternative LfD system based on a fully simulated 3D environment. Simulation eliminates the need for real-life sensors on the robot, serves as a unified medium for recording demonstrated tasks, and facilitates sharing of the produced solutions between different types of robotic cells with minimal to no reconfiguration. The proposed system also features a Virtual Reality interface which allows the operator to interact with the environment in a natural way when recording task demonstrations. The system is built on top of commonly available software such as Unity engine, Robot Operating System, ROS-Industrial and MoveIt motion planning framework.

The thesis is in English and contains 67 pages of text, 8 sections, 12 figures and 2 tables.

Annotatsioon

Õppimine demonstratsioonist (LfD) on lähenemine robotite programmeerimisele, kus masina eesmärk on korrata ehk imiteerida inimesega esitatud ülesannet ilma, et oleks selle ülesande täitmiseks robot protsessid detailselt programmeeritud. See on tõhus viis keerukate robotirutiinide seadistamiseks ka programmeerimisoskusteta kasutajate jaoks, ning enamik LfD rakendusi põhineb robotiga liidestatud anduritele, mis analüüsivad ja salvestavad ümbritseva maailma olekut ja aitavad demonstreerida masinale ülesannet. Antud töös esitatakse alternatiivse LfD süsteemi, mis põhineb täies mahus simuleeritud 3D-keskkonnale. Simulatsioonil pole vajadust robotile paigaldatud füüsilise anduri järel ning see toimib ühtse keskkonnana LfD ülesannete salvestamiseks ja teostab tehtud lahenduste jagamist erinevate tüüpi robot süsteemide vahel minimaalse või ilma lisakonfigureerimise vajadust. Pakutud süsteemil on ka Virtuaalreaalsuse (VR) liides, mis võimaldab operaatoril ülesannete demonstratsioonide salvestamisel keskkonnaga loomulikul viisil suhelda tänu VR prillidele. Süsteem on ehitatud laias kasutuses oleva tarkvara tööriistade baasil, nagu näiteks Unity mängumootor, robotite operatsioonisüsteem (ROS), ROS-Industrial ja MoveIt eri-liikumise planeerimiste teekart.

Antud diplomitöö on kirjutatud inglise keeles ning koosneb 67 tekstilehest, 8 peatükkist, 12 joonisest ja 2 tabelitest.

Nomenclature

3D	Three-dimensional
CAD	Computer aided design
Cobot	Collaborative robot
DoF	Degrees of freedom
EEF	End-effector
FK	Forward kinematics
GUI	Graphical user interface
HMD	Head-mounted display
IK	Inverse kinematics
IP	Internet Protocol (can also refer to Internet Protocol address)
LfD	Learning from demonstration
MCP	MoveIt Configuration package
ML	Machine learning
MPS	MoveIt Planning Scene
MSA	MoveIt Setup Assistant
MTDF	Manipulation Task Description Format
NN	Neural network
ROS	Robot Operating System

SRDF Semantic Robot Description Format

TCP Tool center point

TCP Transmission Control Protocol

UI User interface

URDF Universal Robot Description Format

XML Extensible Markup Language

Contents

1 Introduction **11**

1.1 Research domain 11

1.2 Room for improvement 12

1.3 Thesis goals 13

1.4 Thesis contents 13

2 Background **15**

2.1 Note on the definitions 15

2.2 Categories of LfD 15

2.2.1 Kinesthetic teaching 16

2.2.2 Teleoperation 17

2.2.3 Passive observation 18

2.2.4 Comparison of demonstration approaches 19

2.3 Potential of passive observation in digital twin environment 20

2.4 Existing simulation-based implementations and their limitations . . . 22

3 System implementation overview **24**

3.1 Design philosophy 24

3.2 Software overview 24

3.2.1 Unity 25

3.2.2 ROS 25

3.2.3 ROS-Industrial 26

3.2.4 MoveIt 26

3.3 Architecture overview 27

4 ROS: motion planning and robot control servers **29**

4.1 Robot representation in ROS 29

4.1.1 URDF and XACRO 29

4.1.2 URDF in ROS-Industrial standard 30

4.1.3 SRDF and MoveIt configuration packages 30

4.1.4 Gripper description for grasping pipeline 32

4.2 Environment representation in ROS 33

4.3 Motion planning 35

4.3.1 IK service 35

4.3.2 MoveIt Task Constructor 36

4.3.3 Manipulation Task Description Format 37

4.3.4 Task planner node 39

4.3.5 Motion planning server . . . 39
4.4 Physical robot control . . . 40
4.4.1 ROS-Industrial drivers . . . 40
4.4.2 Control server . . . 41

5 Unity: task recording interface **42**
5.1 Communication between Unity and ROS . . . 42
5.2 Robot representation in Unity . . . 43
5.2.1 URDF import . . . 43
5.2.2 Robot control scripts . . . 46
5.3 Environment synchronization . . . 46
5.4 Task recording interface . . . 50

6 Use-case: testing LfD system prototype **52**

7 Discussion and future developments **55**
7.1 On the achieved goals . . . 55
7.2 Directions of future research . . . 55
7.2.1 Automatic deployment pipeline for ROS motion planning servers 56
7.2.2 Extending MTDF . . . 56
7.2.3 Migration to ROS 2 . . . 57
7.2.4 System performance evaluation experiment . . . 57

8 Summary **58**

References **59**

A Geometry parameters for *grasp_data.yaml* files for two-finger grippers **64**

B Example of *grasp_data.yaml* (from use-case) **65**

C Example of MTDF YAML (from use-case) **66**

List of Figures

1 *Three different approaches to robot demonstrations[1]* 16
2 *Architecture of the LfD system prototype proposed in this work* . . . 27
3 *Motoman GP8 loaded in MoveIt Setup Assistant GUI* 31
4 *Configuration parameters of ROSConnection component* 42
5 *"ROS Message Browser" window showing 5 MoveIt action types compiled to C# classes* 43
6 *Motoman GP8 robot imported into Unity Scene from its ROS-Industrial URDF file. Note that the robot's structure in Hierarchy tab (on the right) follows the one from URDF, including geometry containers for "Visuals" and "Collisions"* 44
7 *ArticulationBody and UrdfJointRevolute attached to "link_1_s" of the imported Motoman GP8* 45
8 *Interface of MoveItPlanningSceneSynchronizer component. Five objects marked for synchronization with MoveIt can be seen in the "Synchronized Objects" list at the bottom of the image* 48
9 *Robot workcell with object primitives in Unity (left) mirrored into MoveIt Planning Scene in ROS (right)* 50
10 *Initial layout of the recorded task* 53
11 *Steps of the recorded task* 53
12 *Motoman GP8 (left) and ABB IRB 1600 (right) executing trajectories generated from the recorded task's MTDF* 53

List of Tables

1 *Strengths of each demonstration approach* 19
2 *Mapping of collision geometry from Unity to ROS* 49

1. Introduction

Finding an effective way to teach the machine to execute required task has been one of the main challenges in robotics since the emergence of the field. How to split a complex industrial problem into the set of simple, easily programmable sequences of motions? How to keep the produced algorithms reusable for solving similar problems in the future, possibly using robots of different type? And finally, how to allow people with the good understanding of the task at hand, but little experience in writing code to teach robots as effectively as the experienced operators can do?

1.1. Research domain

One of the approaches to robot programming capable of answering the questions above is called Learning from Demonstration (LfD). The essence of LfD is the following: instead of directly programming the robot motions to solve certain task, the user can personally demonstrate the execution of this task and let the machine infer the sequence of actions required to replicate it. Though it may sound like LfD needs advanced Machine Learning (ML) algorithms in order to work, the concept actually dates back to 1980s[2], when robot LfD was done by kinesthetic teaching (where the operator physically moves the robotic manipulator to desired poses, forming a trajectory which is recorded and played back later to repeat the full task). Modern LfD algorithms, however, employ a wide array of techniques to improve the robot's perception of the human actions being demonstrated, achieve better generalization of the output solutions and provide more user-friendly ways to record new task examples (sometimes even freeing the operator from the need to be present next to the real robot hardware at all).

The importance of LfD has been growing in the recent decades thanks to the ongoing democratization of the robotics field, which spans from personal hobby projects to serious industrial systems. Emergence of more affordable robot systems on the market[3, 4] and open-source projects like Robot Operating System (ROS)[5] brings more people into robotics and increases the potential benefit of simpler, more unified programming methods. Researchers are actively investigating the possibilities of integrating LfD principles into the robot programming software in order to lower the barriers of entry for new robotics specialists as well as to make

the experience of robot programming more universal and vendor-agnostic. This interest is reflected in the constantly growing number of publications related to the topic: in the past decade, on average 18% more LfD papers were published each consecutive year[1]. The rate of LfD adoption in the commercial robotics field, however, is slower. This is mainly reasoned by the potential financial risks connected with switching from the proprietary programming languages to a more open and universal LfD paradigm, but the situation is slowly changing with the spread of collaborative robots (cobots). Several robot models on the market already come provided with basic LfD programming functionality out-of-the-box. Examples of commercial systems with support for LfD include CoBlox environment by ABB[6] and TM-Flow by Omron[7] (box allow to use kinesthetic teaching for recording robot poses). Given such tendencies, it is very likely that LfD principles will get integrated into more commercial systems in the upcoming future. That being said, the majority of innovations in the field are still exclusive to scientific research carried out in the laboratories, and the question of an industrially feasible system for LfD programming still remains open.

1.2. Room for improvement

To date, almost all LfD systems proposed in literature heavily rely on expensive hardware sensors. Although there exist studies about using simulated environments for LfD, they still utilize the real robotics hardware in certain parts of the teaching process. In addition, many proposed LfD systems rely on deep learning models, which require time-consuming data collection and training to be effective in custom tasks. This seriously limits the potential of adopting LfD in real-world production scenarios, especially for small and medium enterprises which cannot afford long integration times. Thus, making an LfD solution which is easily reproducible, does not require physical hardware and is capable of adapting to custom tasks with minimal time investment can produce great value for the industry. This thesis work aims to lay the foundation for such system by creating a hardware-agnostic LfD interface running in a simulation based on standardized and commonly available software.

Though LfD can be applied to any type of the robot, this work focuses specifically on arm manipulators. Such choice was made for several reasons. Firstly, the typical tasks carried out by this kind of robots, e.g. pick-and-place or trajectory following, are easy to demonstrate for the human operator. Secondly, these types of tasks

can be replicated in simulation without time or force parametrization while still providing realistic results, which simplifies initial development of the system. Finally, arm manipulators constitute the major part of the robotics market and are available in most robotics research facilities, which provides a reliable ground for testing the proposed solution.

1.3. Thesis goals

Given the defined aim of developing an LfD system running entirely in a simulated environment, this work has to achieve the following goals:

- Present the benefits of using simulated environments for LfD programming.
- Establish an implementation guideline for the hardware-agnostic simulation-based LfD system for industrial robots.
- Develop a Virtual Reality (VR) interface for intuitive recording of task demonstrations.
- Validate system universality by demonstrating its ability to adapt to different robot models.
- Draw suggestions regarding the future developments based on the outlined system.

The project developed in the process of this research is also to be made available in an online repository[8] with the intention to ease its adoption.

1.4. Thesis contents

This thesis consists of eight sections. The first section is an introduction stating the motivation behind the work and the goals set for the research. The second section provides the background for practical findings of this work, giving an overview of the current state of the art in robotic LfD and indicating room for improvement. The third, fourth and fifth sections together provide a thorough explanation of the

system implemented over the course of this research. The sixth section presents a practical use-case validating the system's prototype. The seventh section touches on the future research directions opened by the proposed practical implementation. The eighth section closes the thesis with a short summary and conclusion to the accomplished work.

2. Background

This section provides an insight on the current state of LfD in robotics, including the overview of existing approaches and solutions, as well as further explains the motivation behind this research work.

2.1. Note on the definitions

Before starting a deep dive into research literature connected with the topic, let us make the definitions clear. As of 2021, there is still no standard name for LfD, so the same robot programming approach may be referenced in literature as "Imitation Learning", "Programming by Demonstration", "Teaching by Example" etc. However, according to the field overview published in 2020[1] (which is the most recent one as of the time of writing of this work), over 36% of related research articles published in between the years 2008-2018 tend to use the term "Learning from Demonstration" (with the next most popular option being "Imitation Learning", at 29%). In the light of this, it was decided to stick to "Learning from Demonstration" as the most widely used naming.

2.2. Categories of LfD

In the generic meaning of the term, LfD is focused on algorithms which can generate program code based on the demonstration of the desired outcome by the user. This paradigm has found different uses in computer software, from basic things like defining macros in office software[9] to more complex ones like generating SQL queries based on desired data description[10].

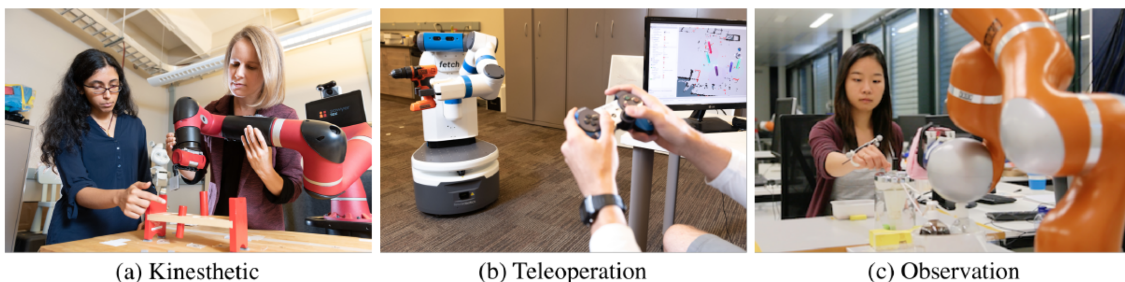
However, LfD in robotics holds a special place, largely thanks to the physicality of the domain. Most tasks carried out in software are abstract, like manipulating information in a table or establishing communication link between two networked machines. Conversely, tasks executed by the robots assume interaction with the physical world, which is much more natural for humans. This fact inherently provides us, as human operators, with more choices when deciding how to demonstrate the

task to the machine.

Existing approaches to robotic LfD can be divided into 3 main categories based on the demonstration type*, as described in [1]:

- Kinesthetic teaching
- Teleoperation
- Passive observation

Proper understanding of these categories will help correctly differentiate the system presented in this work. Thus, each of them is explained in detail in the following subsections.



(a) Kinesthetic (b) Teleoperation (c) Observation

Figure 1. *Three different approaches to robot demonstrations[1]*

2.2.1. Kinesthetic teaching

Historically, kinesthetic teaching was the first type of robotic LfD. This method allows the user to physically move the robot's links into desired poses, sequentially building the full trajectory required for executing the task. All motions of the robot during the teaching procedure get recorded by the on-board sensors, storing values of joint angles and/or torques. The resulting set of data can be used directly to playback the motions required to execute the task, or be utilized as a training set for ML model which can help generalize the solution better[11].

*It is worth noting that there also exists categorization of LfD methods based on the learning outcome (i.e. what is generated as the learning output - low-level joint trajectories, abstract task policy etc.). This classification is not be detailed here, because it is already explained well in the existing overviews ([1]) and is not crucial for the description of the system presented in this work (presented system, in fact, produces both low-level and high-level learning outcomes in its different stages; the implementation focuses on the improvement of the demonstration method).

Kinesthetic teaching gained significant popularity with industrial arm manipulators, because it requires little training and makes the programming process intuitive even for new operators. On top of that, kinesthetic approach does not require additional external sensors, because robot motions get recorded by the internal sensors in the robot joints' motors. Direct recording of robot poses during the task also entirely eliminates the problem of mapping LfD task inputs to robot motions, also known as the correspondence problem[12].

Due to its simplicity, kinesthetic teaching also falls victim to several limitations. Human operator must be physically present next to the machine to manipulate it. It also becomes harder to apply this approach as the robots complexity grows: some tasks, such as legged locomotion, require posing multiple limbs of the robot simultaneously, which may be challenging or even impossible to execute for a single operator; same goes for complex grippers, such as multi-finger robotic hands, where many joints have to be adjusted to achieve a good grasp. Another drawback is that the quality of the recorded motions depends on the experience and dexterity of the operator who demonstrates the motions, and produced trajectories often have to be post-processed to achieve smooth results. Finally, most of the industrial robot's joints have too high resistance to be moved by humans without aid. Internal sensors required to detect human's motion intent and move the servos accordingly are usually installed only on cobots, while for traditional industrial robots these sensor systems have to be purchased separately (for example [13]). This puts a price premium on using kinesthetic teaching in the practical industry scenarios.

2.2.2. Teleoperation

Teleoperation allows for somewhat greater flexibility than traditional kinesthetic teaching. This LfD approach relies on controlling the robot through an arbitrary external input, which can be a simple joystick, graphical user interface (GUI) or an immersive VR application[14]. Teleoperation is similar to kinesthetic teaching (and can even be seen as an evolution of the latter) in a sense that it allows human operator to directly manipulate the robot for achieving the intended result.

Teleoperation has an obvious advantage which comes by definition: ability to control the taught robot over distances. Thanks to the modern Internet, robotic systems can now be reliably controlled by remote operator in almost real time, even between

continents[15]. An important implication of remote access is that it can be used for crowdsourcing multiple demonstrations without bringing people to facility with the robot (successfully utilized in [16, 17]). This allows to collect larger datasets faster, leading to better training results in case if ML models are used for LfD. Teleoperation LfD also does not have to deal with the correspondence problem, as the robot motions are directly mapped to the executed task during the demonstration phase.

The downside of the teleoperation approach is the fact that it requires additional effort to develop the remote interface for controlling the robot. The developed solutions are also rarely standardized to be applicable to different types of robots, mainly due to the relative novelty of this approach. The standardization aspect, however, begins to improve with the introduction of popular open-source solutions like ROS.

2.2.3. Passive observation

In passive observation LfD, the learning is done by observing the operator manually execute the intended task. In this case, the real robot does not participate in the task during its demonstration. Passive observation-based LfD is definitely the most complex of the three categories discussed in this section (in terms of implementation), but it can greatly simplify the demonstrations for the user.

Serious advantage of passive observation is the intuitiveness of the demonstrations. In contrast to kinesthetic teaching and teleoperation, the user no longer has to learn how to manipulate/control specific robot - instead, the task is shown by the teacher in a natural way, and the software handles mapping the recorded actions to robot motions needed to perform them. This method is applicable to robots with high degrees of freedom (DoF), as it does not require the teacher to manually pose every joint of the robot. Finally, the separation of the demonstration from the robot hardware learning to perform the task means that the same task recording can be potentially reused for teaching different robots (e.g. manipulators of different brands can be taught to perform the same pick-and-place operation). This can, for example, save significant amount of time when comparing the efficiency of different robots when designing a new manufacturing line.

However, the perceived simplicity for the operator comes at the cost of extra com-

plexity under the hood of the system. Passive observation LfD has to deal with the correspondence problem head-on, because the recorded actions of the operator have to be somehow mapped to the robot's motions, and the procedure of this mapping will vary depending on the recording method used. Sensor noise and occlusion in recordings also have to be dealt with to avoid affecting the mapping result. But if these challenges are handled properly when designing the LfD system, passive observation has the potential to become the most efficient way of robot programming for an average user.

2.2.4. Comparison of demonstration approaches

Summary of the strengths and weaknesses of the described demonstration approaches is shown in Table 1 (based on [1]).

Table 1. *Strengths of each demonstration approach*

Demonstration approach	Ease of demonstration	High DoF robots	Ease of mapping
Kinesthetic teaching	✓		✓
Teleoperation		✓	✓
Passive observation	✓	✓	

It can be seen that passive observation approach is the only one which combines both ease of demonstration and support for high DoF robots. The first factor is important when the adoption potential of the LfD system is concerned, as easier demonstrations equal to lower barrier of entry for new users. When compared to kinesthetic teaching, which also makes demonstration intuitive to the user, passive observation has an edge in terms of iteration times: executing the task manually is usually faster than by moving the physical robot.

Kinesthetic and teleoperation approaches do avoid the correspondence problem, simplifying the mapping of recorded human actions to the robot. However, it can be argued that "Ease of mapping" criteria is only significant during design time of the LfD system, and, if automated properly, does not concern the user when recording the demonstrations. For example, in pick-and-place tasks, once the algorithm is able to automatically generate grasps and approach/retract motions for the objects repositioned during the task, it will be sufficient for the operator to show how and which objects have to be moved in order to generate the program.

However, even if the algorithm can be designed to properly handle the mapping of recorded human motions to the ones of the robot, another question still remains: how to perform the recording? Most implementations to date have relied on real-life sensors (mainly cameras) which monitor the motions of the human and/or objects of interest in the real scene[18]. Such approach makes the LfD algorithm dependent on specific sensor hardware, limiting its universality and increasing the costs. In this thesis, it is proposed that recording observations in the digital twin of the real environment instead can be a universal and scalable solution for the correspondence problem in passive observation LfD.

2.3. Potential of passive observation in digital twin environment

The concept of the digital twin (DT) was initially framed in the NASA research paper in 2012[19]. The definition has subsequently evolved into a more concise form as presented by Chang in [20]:

"*A digital twin is a computerized model of a physical device or system that represents all functional features and links with the working elements.*"

Utilizing the digital twin of the industrial robot's workcell for recording LfD task demonstrations can provide two important benefits, namely:

- Easy access to the full state of the scene at any moment in time
- Unified interface for recording and sharing task demonstrations

Let us explore both points in detail. First, the simulation inherently holds the data about the state of every object contained within it. Consequently, every interaction of the user with the elements of the virtual workspace can be easily recorded and reproduced. Modern computer hardware is capable of running physics simulations in real time, assuring the validity of the recorded scene state.

This leads to the second point, making the simulation a unified interface for recording and sharing the task demonstrations. Such approach eliminates the need for hardware

sensors at task recording stage, and avoids black-box ML models mapping sensor inputs directly to generated robot motions [18, 11]. Instead, the whole flow of the task can be described in a hardware-agnostic manner through 3D geometry and its changes in terms of simulated physics. It must be noted that this approach to recording tasks *does not aim to eliminate the correspondence problem*: the algorithm still has to map the changes in the scene to low-level motions of the robot required to execute them; *instead, it provides a universal base condition* off of which the correspondence problem can be solved, eliminating human operator and real-world noise from the equation. This can effectively counter the drawbacks of the passive observation approach, as described in the previous section.

Of course, the benefits from above would be immediately halted if recording tasks in the simulation would limit the performance of the teacher. This may hold true for traditional 2D interfaces residing on computer screens, but nowadays we have more immersive technologies to choose from. Modern VR hardware provides an intuitive interface between the human operator and digital twin environment, enabling the former to naturally perform physical interactions such as grasping and moving virtual objects, while also enhancing the experience with world-space UI elements. This makes VR a good potential candidate for building a pure simulation-based LfD system.

Nevertheless, challenges also exist in the proposed digital twin approach. Time has to be invested into modelling the virtual environments prior to simulation, including all the objects relevant to the task. However, this undertaking becomes easier if we take into consideration that most modern industrial products and parts are available as 3D computer-aided design (CAD) models. Incorporating digital twin of the programmed robot into the simulation also becomes easier thanks to the existence of the official support packages from manufacturers, facilitated by initiatives like ROS-Industrial. 3D scanning techniques like photogrammetry can be used to quickly generate the full 3D representation of the static geometry of the real robotic cell.

Another possible challenge of digital twin-based LfD lies in dealing with highly dynamic environments. For example, LfD for mobile robots would likely require different algorithms than for manipulation tasks, and would have to rely on some representation of hardware sensors used for localization to keep the simulated environment up-to-date in real time. For this reason, and in order to keep focus during the initial development of the novel digital twin-based LfD concept, this thesis limits

its scope to industrial manipulators and tasks associated with them.

To conclude this section: even with the outlined challenges, pure simulation-based LfD has potential benefits worth investigating, which is done in this thesis. Next section presents a concise overview of the existing related solutions to provide practical background before proceeding to the implementation sections of this work.

2.4. Existing simulation-based implementations and their limitations

Existing implementations of LfD mostly utilize simulation and VR interfaces for more convenient teleoperation and crowdsourcing demonstrations.

For example, multiple studies have focused on using VR as an immersive teleoperation interface[14, 11, 21]. In [14] the researchers use VR rig (head-mounted display (HMD) and handheld controllers) as an input source for controlling dual-arm robot in real time (Baxter by Rethink Robotics). In this implementation, the user hands' movements are directly mapped to the hands of the robot, allowing for intuitive teleoperation. In the virtual environment, the user is presented with real-time colored point cloud feed from the camera installed on the robot, allowing to see the environment in which the robot operates. However, it relies on the depth-sensing camera which is not included in most commercial robots by default (in contrast to Baxter robot used in that article). Although the study does not focus on LfD capabilities of the system, potentially it can be used for collecting demonstration data with teleoperation approach. The authors also assess their teleoperation interface's performance for pick-and-place tasks in the follow-up research[22]. This system is built on popular software: Unity and ROS, which is good because it can ease the adoption. Integration with ROS also simplifies the adaptation of other robot models with the existing teleoperation interface. The system developed in this thesis follows a similar vision regarding the used software components.

In [11], a similar teleoperation system is introduced, albeit it is utilized for the actual LfD. The authors use the same principle of VR teleoperation: VR rig motions are mapped to the hands of the robot in real-time, and the real environment is streamed to the simulation through a colored point-cloud from the depth-sensing camera installed on the robot. The operator can utilize this teleoperation interface for

recording task demonstrations. Recorded data is used to train a custom convolutional neural network (NN), which produces robot arm poses as its output. The system is using PR2 robot by Willow Garage and its control logic is implemented solely in Unity (in C programming language). Such architecture certainly requires more effort for integrating other robot models when compared to previous approach (manufacturers often provide their robot communication packages for ROS, but for Unity those have to be implemented manually). Moreover, to make system work after changing the robot model to another one will require colleting new set of demonstration data and retraining the NN, as it would have to produce solutions for different kinematics chain. Thus, this solution is hardly scalable.

The example of crowdsourcing demonstrations through simulated environment is found in [17]. In this study, the researchers explore so-called *wide-area pick and place* tasks, where the mobile robot with attached manipulator has to arrange the objects on different surfaces in the domestic environment. Crowdsourcing was implemented through a web interface, where the users could see the simulated living apartment and arrange pickable objects in it using their mouse pointer. The collected data was used to train an ML model for inferring the optimal placement of the objects in the environment (e.g. the plate with cutlery has to be placed on the dining table). The study achieves success in generating the proper object placements from abstract task description. It does not explore low-level trajectory generation for the robot, as the authors claim that the robot already had logic for automatic detection and grasping of the objects implemented. Nevertheless, this implementation can be attributed to the category of passive observation learning - as the system aggregates the data by recording the users' actions in the environment. The authors have not disclosed details as to the simulation environment implementation, so it is not possible to draw claims about the scalability of the system.

As it can be seen from the research examples presented above, existing implementations have certain limitations, either relying on hardware to work or being based on hard-to-scale software architecture. To the author's best knowledge, a system providing observation-based LfD programming cycle entirely in simulation based on Unity/ROS has not been described in the literature to date. Such system is introduced in the next section of this work.

3. System implementation overview

This section details the prototype implementation of the digital twin-based LfD system envisioned in this work. The overview of the utilized software is presented, followed by the thorough description of the system's architecture and its individual parts.

3.1. Design philosophy

To ensure that the built prototype is not a once-off research endeavor, but can be grown into a practically useful system, it was crucial to follow two principles during development. The prototype had to be:

- Easy to reproduce, i.e. based on commonly available and maintained software.
- Hardware-agnostic, i.e. require minimal to no changes in code to program industrial robots of different models.

These two criteria serve as the cornerstones for all things described further in this work.

3.2. Software overview

Quality of a passive observation-based LfD implementation is determined by at least two of its parts: interface for making demonstrations and power of the algorithms handling mapping of the task to the robot. Thus it was decided to use best-suited software for each of these parts. Visualization and VR interface for recording the demonstrations were implemented in Unity game engine, while the "brains" behind the task mapping and robot kinematics were built in ROS. Long-term support (LTS)* versions of software were favored for implementation (where available). Each piece of software is presented in the individual sub-section below.

*Long-term support label marks the given stable release of the software as the version to be supported/updated by its developer over the extended period of time (usually several years).

3.2.1. Unity

Unity is a platform for creating interactive, real-time 3D applications[23]. While originally a game engine, in recent years it has gained significant popularity in other fields such as cinematography and industrial visualization. Unity is a perfect fit for developing VR apps, as it has the leading number of supported VR platforms in industry and multiple official and community libraries to boost the development. The engine also uses NVidia PhysX as its default physics solver, which is capable of running decent robotics simulations[24].

Unity uses C# as its main programming language. There exists an open-source integration between C# and ROS in the form the ROS (ROS-Sharp) project by Siemens[25]. Unity team has recently revealed a demo project which uses the engine to visualize the robot models simulated in ROS[26]. This demo also served as an inspiration for this thesis work.

Thus, Unity was chosen as the visualization interface for our LfD system. In this prototype, we use Unity version 2020.4 LTS.

3.2.2. ROS

Robot Operating System is a modular framework for developing robot software[5]. It is an open-source project with over 10 years of history, and is actively supported by research and industrial community around the world. The power of ROS lies in its modular publisher-subscriber architecture, which splits application logic into dedicated modules called *nodes* which communicate with each other through the network. Over the years, ROS has aggregated multiple community packages and smaller projects dedicated to solving any robotics-related tasks. Several of them are utilized in this work. Hundreds of custom and commercial robot models are supported by ROS nowadays[27].

ROS nodes can be written in either C++ or Python programming languages. The core functionality of ROS available to both languages is identical, so community packages may be based on either language. C++ is usually preferred for performance-intensive code, while Python interfaces are provided for faster prototyping.

Currently, the most widespread version of ROS is ROS 1. The newer version, ROS 2, is coming with multiple improvements over the original, but it is still in active development and lacks documentation and proper community support. The latest version of ROS 1, Noetic Ninjemys, is considered LTS software and will stay an industry standard for several years to come.

Thus, ROS 1 was chosen as a software for handling robot simulation and communication in out LfD system. In this prototype, we use ROS Noetic (ROS 1) as our distribution.

3.2.3. ROS-Industrial

ROS-Industrial is an open-source initiative aimed at providing official ROS support packages for industrial robots[28]. This project aims to provide a standard interface for communicating with industrial robot controllers from ROS, facilitating hardware-agnostic software development. Standardized support packages also include the files describing geometry of the robot, which can be directly incorporated into simulation. Several vendors have already joined the ROS-Industrial initiative, including ABB and Motoman[29].

Supporting ROS-Industrial standard in our system has the potential to extremely simplify the creation of digital twins for robots, so it was taken into account during development.

3.2.4. MoveIt

MoveIt is a motion planning library for ROS[30], originally introduced in [31]. It provides extensive functionality for planning collision-aware trajectories, automatic generation of pick-and-place motions for different object types, calculating forward and inverse kinematics (FK and IK) of the robot. MoveIt architecture is based on plugins, so many core features (for example, IK solver) can be easily replace with the implementation more relevant to the task at hand. MoveIt is used by several industry-leading companies and is also recognized by ROS-Industrial as a standard tool for handling robot kinematics and planning.

It was decided to use MoveIt capabilities in our LfD system to handle mapping of recorded task actions to robot trajectories required for their execution.

3.3. Architecture overview

The proposed system consists of the three core modules:

A. Task recording interface in Unity

B. Motion planning server in ROS

C. Robot control server in ROS

Functionality of the modules and relations between them are shown in the system diagram in Figure 2.

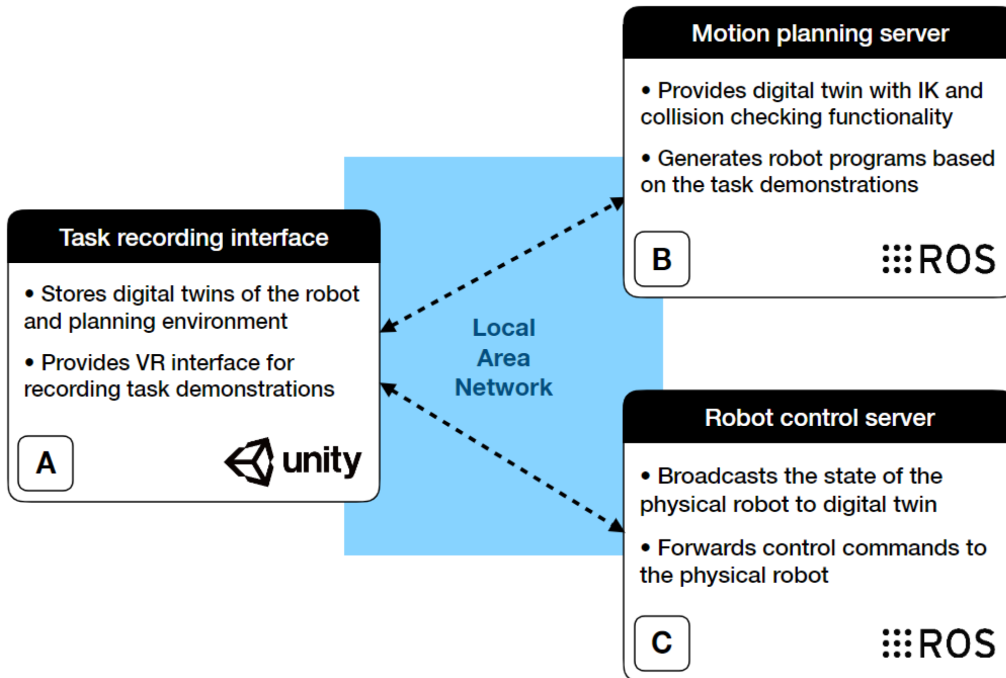


Figure 2. *Architecture of the LfD system prototype proposed in this work*

Modules A and B provide all functionality of the LfD system, and are not dependent on C. Module C is used only when the generated programs have to be executed on the physical robot.

The two sections that follow comprehensively explain the inner workings of each module. It shall be noted that many features, like simulated environment definition, are used in both ROS and Unity sides of the application, and their full value shines when both sides are understood. Thus, descriptions of certain features used in ROS have references to their counterparts in Unity, and vice versa, to aid in navigating the explanation.

4. ROS: motion planning and robot control servers

4.1. Robot representation in ROS

In order to make the system compatible with manipulators of different types, a way of representing the robots in simulation had to be carefully planned. ROS-Industrial standard was chosen as a way to ensure easy integration of new robots into the system, because it provides a unified ROS interface for robots from different brands. It was in turn combined with MoveIt as the component for simulating the robot model. Let us explore each element of this approach in detail.

4.1.1. URDF and XACRO

The Universal Robot Description Format (URDF) is a specification used to describe robots in ROS[32] based on Extensible Markup Language (XML). It can provide information about the robot's visual and collision geometry, kinematics (e.g. joint articulation limits) and dynamics (e.g. joint speed limits). Each URDF file has "robot" XML tag at its root, which in turn can contain "link" and "joint" tags. "Link" tags are used for describing physical links of the robot and their geometry. "Joint" tags are used to describe how links are connected with each other, as well as articulation and speed limits of those connections. Currently, the main limitation of the format is that it does not support closed kinematic loops, which rules out all parallel robots. However, URDF still can be used for most industrial arm robots, as only minority of the models on the market possess parallel links.

XACRO (derived from "XML Macros") is a macro language which allows to reuse XML files as an expandable macro expressions[33]. It often accompanies the URDF in ROS, as XACRO allows to "assemble" robots from several files — something that cannot be done with plain URDF. This feature is especially useful for combining industrial arms with custom grippers without having to manually rewrite the whole URDF.

In this work URDF/XACRO files are used to provide a coherent robot representation across ROS and Unity. On the ROS side, robot models are imported into MoveIt

simulation from XACRO files. For URDF import in Unity, please see subsection 5.2.

4.1.2. URDF in ROS-Industrial standard

Controversially, the advantage of using ROS-Industrial standard instead of generic URDFs lies in the restrictions it imposes. According to ROS-Industrial, each robot support package created by a manufacturer *must* include the URDF of the robot it describes. On top of this, the standard defines strict guidelines regarding the content of these URDF files, namely:

- URDF must define correct articulation limits and speeds of the joints of the robot.
- URDF must include several standard links:
  - "base_link" representing the base point of the robot arm,
  - "flange" representing the attachment point of an end-effector (EEF),
  - "tool0" representing tool central point (TCP).
- Corresponding XACRO file must be provided together with the URDF.

These restrictions create a common interface for working with ROS-Industrial URDFs. This commonality becomes very useful when automating the import of industrial robots to the simulated environment. Having standard components in robot structure, such as "base_link" and "tool0", allows to automatically set up FK and IK algorithms and to generate semantic robot data used by MoveIt library. It shall be noted that the developed prototype still works with generic URDF files (i.e. the ones not following ROS-Industrial conventions), but such files require varying amount of manual input from the user to set up (for example, the user has to manually show the system which link of the robot should be treated as TCP if "tool0" link is missing).

4.1.3. SRDF and MoveIt configuration packages

Semantic Robot Description Format (SRDF) is used to store semantic information about the robot that complements what is already available from the robot's URDF

file[34]. Among other features, SRDF allows to define named groups of robot joints or links (to use in motion planning) and mark selected links to be ignored in collision checks (to optimize planning performance). SRDF files are used by MoveIt library to handle motion planning of the robot. In ROS, the most straightforward way to create SRDF for the given robot is to generate a MoveIt configuration package.

MoveIt configuration packages (MCPs) are special ROS packages which store parameters and code required to use the given robot with MoveIt motion planning pipeline. While MCPs can be created manually, the most straightforward way of creating them in ROS is using MoveIt Setup Assistant (MSA)[35]. MSA is a ROS node which provides a GUI aimed at generating a MCP based on the robot's URDF or XACRO file (seen in Figure 3).

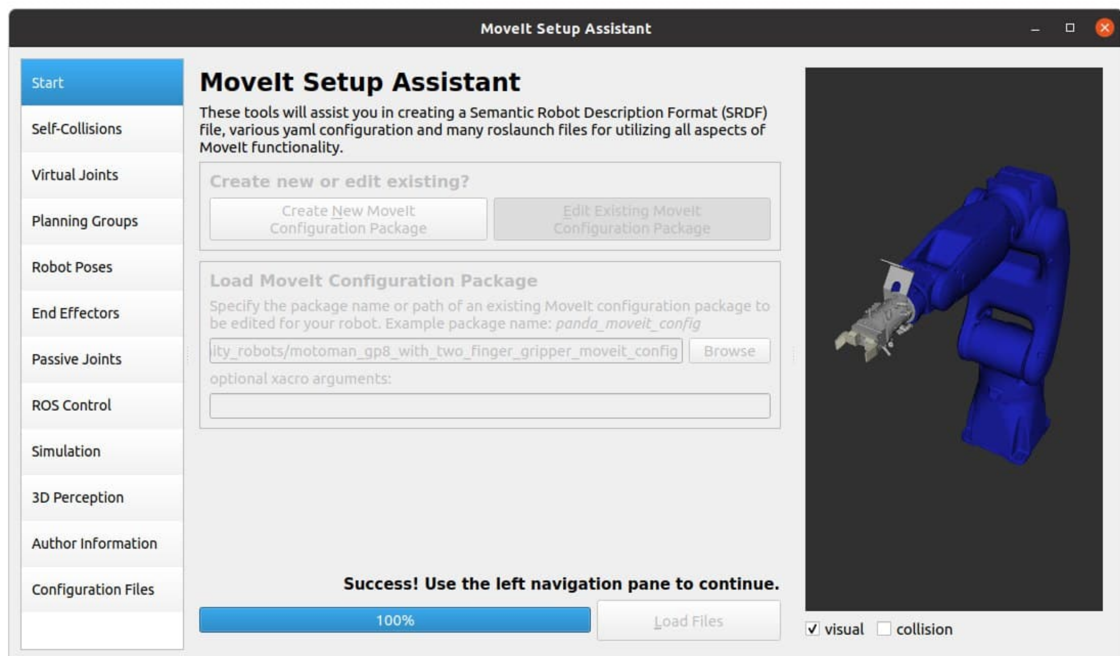


Figure 3. *Motoman GP8 loaded in MoveIt Setup Assistant GUI*

All steps required to generate a MCP in MSA are available in [35] and will not be listed here for clarity.

MCPs of the robots used for testing this prototype were generated through MSA as a part of initial research process. From the author's experience, generating a new MCP for industrial robot in MSA takes under 5 minutes (after initial acquaintance with the approach). However, it is worth noting that in the future our architecture can allow to generate MCP in a more user-friendly manner on Unity side without the need to interact with ROS (discussed in section 7.2.1).

4.1.4. Gripper description for grasping pipeline

Automated generation of pick-and-place trajectories would be impossible without a way to determine gripper motions required to grasp and release the object moved in the task. MoveIt provides this exact functionality in the form of the grasp generation plugins, and this exact feature was used in our system prototype. Grasp plugins are responsible for generating grasp and release motions given the gripper configuration and the model of the object that has to be grabbed. The plugins can be split into two categories:

- Procedural (represented by MoveIt Grasps[36], published in [37])
- Deep learning-based (represented by MoveIt Deep Grasps[38])

MoveIt Deep Grasps plugins use NNs to infer the correct position of the gripper. Two options are available:

- Grasp Pose Detection library plugin[39] (samples grasps from the point cloud using convolutional NN).
- Dex-Net plugin[40] (samples grasps from the camera images using convolutional NN).

Both NN-based grasp libraries take sensor data as input. While sensors data (both point clouds and images) can be generated in the virtual environment, such approach is cumbersome and may require retraining the NNs with new, simulated datasets to achieve optimal performance. Thus, in line with our "hardware-agnostic" principle, it was decided to favor the procedural MoveIt Grasps plugin, which is able to compute grasp poses for simulated object primitives without the use of NNs. MoveIt Grasps plugin also supports grasp generation for both two-finger and suction grippers, which makes is a good fit for typical industrial pick-and-place tasks.

In order to configure MoveIt Grasps, two files have to be included in the MCP of the robot for which the grasps will be planned:

- *ROBOT_grasp_data.yaml* which describes robot's gripper geometry.

- *moveit_grasps_config.yaml* which configures behavior of grasps generator, including debug features.

The contents of each configuration file are detailed in [36]. Illustrations explaining gripper geometry parameters (as should be defined in *grasp_data.yaml* file) are included in Appendix A. An example of *grasp_data.yaml* file for the gripper used in our prototype is provided in Appendix B.

With MoveIt Grasps pipeline configured, the setup of the robot model in ROS is complete and can be used for solving LfD task definitions supplied from recordings, as explained further in this work.

4.2. Environment representation in ROS

In MoveIt framework, the representation of the virtual world is called MoveIt Planning Scene (MPS)[41]. MPS contains the robot model as well as all the objects used for collision checking when planning motions. MPS provides C++/Python interfaces for adding, deleting and updating objects in the scene, and a ROS application programming interface (API)[42] providing the same functionality.

In our prototype, MPS is the source of digital twin environment state on the ROS side. However, manually building this environment in ROS would be a suboptimal approach. The only graphic way of editing MPS in ROS is through MoveIt GUI in Rviz (ROS 3D visualization tool[43]), which was originally created for displaying, but not for composing 3D content. Object manipulation tools available in Rviz are not efficient for an average user when editing complex scenes with many objects. Unity, on the other hand, is a software which have been perfected for authoring 3D scenes over the years, providing a variety of tools and scripts to streamline the process. Thus, supplying ROS with scene state from Unity was a straightforward decision.

Before we can proceed forward, let us touch on how the data can flow between different parts of a ROS application. ROS nodes use *messages* and *topics* as the means of communication between each other. Topic is a destination address with a certain name. Messages define the type of data accepted and published by the topic. ROS topics provide *stream* communication: messages published to the given topic arrive

to all subscribers listening to it. Topics are usually used for streaming constantly changing data; for example, a point cloud from a laser sensor. However, ROS also provides *request-response* communication pattern in the form of services: when a ROS service takes a request message, it does certain work and sends reply with the results back to the requesting client. Services are normally used for executing quick, granular calculations or tasks; for example, retrieving the IK solution for the given pose of robot's EEF. Finally, ROS has *actions*, which can be thought of as a "heavy-duty" version of services: actions also work using request-response communication pattern, but can provide progress updates while the request is being processed. Thus, actions are usually used for executing time-consuming logic; for example, commanding a mobile robot to move to certain point and waiting until the objective is reached. Custom messages, services and actions in ROS are correspondingly defined in **.msg*, **.srv* and **.action* files. These are plain text files which declare the structure and data types used in each message. Keeping message description in plain text allows to automatically generate C++ and Python data structures for corresponding message type.

MoveIt library includes dozens of message types used in communication with the framework (the full list is available in [44]). ROS API for MPS consists of two ROS services* which can be called to interact with the scene:

- *apply_planning_scene* service takes an *ApplyPlanningScene* message[44], which contains a list of named 3D objects with corresponding commands to add, remove or update their position in the scene; the service returns a boolean showing whether the scene changes have been applied successfully.
- *get_planning_scene* service takes a *GetPlanningScene* message[44], which contains a binary mask used to specify which components of the scene are being queried (this allows to query the data selectively to preserve bandwidth); the service returns a *PlanningScene* message containing the data asked for in the request.

The digital twin of the environment used for the task recording is originally created in Unity, and is then synchronized with MoveIt by sending the geometry data of all objects relevant to the task to the *apply_planning_scene* ROS service. Such approach allows to build complex 3D scenes using Unity's rich toolset and synchronize selected

objects from them with MoveIt for motion planning. The logic of this synchronization on Unity side is covered in subsection 5.3.

4.3. Motion planning

This subsection explains how the motion planning capabilities of MoveIt were used in the prototype to automatically generate robot trajectories from the recorded tasks, thus handling the correspondence problem of observation-based LfD.

4.3.1. IK service

IK is the basic function required for motion planning with any robotic arm. It is possible to do IK calculations in Unity, as the kinematic structure of the robot is available in the scene and the engine has multiple IK libraries designed for it. However, IK calculations are essential in ROS, because in our prototype exactly ROS is responsible for motion planning. Implementing IK on Unity side would add an extra layer of complexity and risk of inconsistency with ROS because of different IK solvers used in each module. Instead, it was decided to allow Unity to use ROS's IK solver by querying a service provided by MoveIt — *compute_ik*. The logic of the service is straightforward: request takes *PositionIKRequest* message, which contains the EEF pose for which IK solution has to be found; response provides *RobotState* message which contain individual joint's poses corresponding to the IK solution, or *MoveItErrorCodes* message in case the solution was not found.

Like the grasping solvers, IK in MoveIt is provided in the from of plugins. The list of options is vast, complimented by a possibility to create custom plugins. Due to the timeframe limitations for this research work, only 2 solvers were explored, namely:

- *KDL*, a default solver[45].
- *IKFast*, provided by the OpenRAVE framework[46].

MoveIt can be configured to use specific IK plugin through *kinematics.yaml* file located in the robot's MPC (the process is detailed in [47]). The prototype system

was initially tested with the default KDL solver, but it turned out to be relatively slow and often provided unstable solutions. In an attempt to improve the situation, it was replaced with IK-Fast plugin, which resulted in better performance and extremely stable solutions. It has to be noted that IK-Fast plugin has to be compiled through OpenRAVE library based on the robot's URDF, which is not an optimal approach (though it can be automated). Another limitation is that OpenRAVE cannot be compiled and used on ARM architecture-based processors, which restrains from using it on most modern single-board computers (like Raspberry Pi). *TRAC-IK* plugin developed by Traclabs[48] has a promise to deliver the same performance as IK-Fast with easier installation and no dependency on URDF, but at the moment of writing of this thesis it was not available for Noetic, the version of ROS used in this work. For the prototype needs, IK-Fast was chosen as a viable option, with plans to evaluate more IK plugin alternatives reserved for future research.

4.3.2. MoveIt Task Constructor

With the robot model, virtual environment and IK solver set up, we have all the prerequisites for generating motion plans from the recorded LfD tasks. In the prototype, the mapping of the recorded virtual objects' movements to the robot actions is accomplished by MoveIt Task Constructor (MTC)[49]. This is a novel high-level planning module introduced to MoveIt framework in 2019 with the aim to replace the older pick-and-place pipeline. MTC allows to plan complex routines consisting of multiple interdependent subtasks. Each such subtask in MTC is called a *stage*. The stages are divided into two classes:

- *Primitive stages*, which are dedicated to solving a singular problem (e.g. generating plan for EEF motion between two points).
- *Container stages*, which are used to encapsulate several primitive stages and post-process their results (e.g. a container stage wrapping a primitive IK solver can be used to filter computed IK solutions based on the given constraints).

Any manipulation task can be represented as a combination of primitive and container stages in MTC. A feature worth noting is that MTC allows the solutions of individual stages to propagate both forward and backward through the task. For example,

the solver can account for the pose of the gripper required for placing the object when deciding how the object should be grabbed in the first place. This allows to successfully plan complex pick-and-place tasks.

MTC is written in C++ programming language. In order to make MTC compute the solution for the manipulation task, the programmer must correctly define the sequence of stages be to solved and their corresponding inputs in code. In the case of our system, we aim to select the required stages automatically based on the sequence of actions recorded during the user's demonstration in Unity. This raises a need for a way to store information about the recorded task, and pass it from Unity to ROS for automatic planning.

4.3.3. Manipulation Task Description Format

For our prototype, we introduce the concept of Manipulation Task Description Format (MTDF). It is a file specification intended for describing a sequence of object motions in 3D space. The format does not specify the means by which the objects should be moved, making it agnostic to the configuration of the robot (or the operator) which will be executing the manipulation task. Although the implementation in this work is limited to describing pick and place tasks, the author hopes that in the future the format can be extended to support more object manipulation scenarios (e.g. using machining tools and interacting with controls such as valves and levers).

In this work MTDF was implemented using YAML data serialization language. Such choice was made because YAML is easily readable by humans, aiding in debugging and improving the format during initial development. It shall be noted that MTDF can be implemented in other languages (e.g. JSON or XML), and the default language used may change based on the future research.

Now let us break down the structure of an MTDF file. It consists of two main parts:

- Object descriptors
- Keyframes

Object descriptors is a list of strings uniquely identifying 3D objects being manipulated

in the task. It must be noted that MTDF format is not fully self-contained: it does not store the information about the initial configuration of the scene before task execution, nor does provide any information about the 3D geometry of the manipulated objects. Such design decision was made on purpose, leaving the identification of the objects by the descriptor to implementation, and allowing to reuse the same high-level task in different starting conditions. In the presented prototype, each descriptor string is the actual name of MPS object loaded from Unity. Nonetheless, the author believes that in the future this concept can be extended: instead of a name, object descriptor may reference a list of rules used to identify the object in the virtual (or real) environment. For example, certain descriptor can refer to "red part with the given QR code, located in the robot's work envelope". However, implementation of such generic object descriptors requires extensive research on its own and is out of scope of this thesis work (more on this in subsubsection 7.2.2).

Keyframes is a list containing data about how and which objects get manipulated. Following our "hardware-agnostic" priciple, keyframe only describe the movements of the objects, not the manipulator which executes them. In the prototype implementation, MTDF has two types of keyframes:

- *Pick* — describes "pick" action; contains a single object descriptor (object to be picked).
- *Place* — describes "place" action; contains the descriptor of the object to be placed, and 3D pose in which to place the object (position and orientation).

These two keyframe types are sufficient for describing basic pick-and-place tasks, and were utilized for testing the prototype of our LfD system. It shall be noted that object poses in MTDF (such as the object pose in Place keyframe type) are recorded relative to the "task origin" — a custom reference frame defined by the user when recording the task. This allows to plan the task at an arbitrary pose relative to the robot. After MTDF is recorded from the user's VR demonstration in Unity, the starting environment state gets synchronized with MPS, and MTDF gets sent to the ROS node which uses it to generate the motion plan for the taught robot. Functionality of this node is described in the next subsection. Also, an example of the MTDF file generated from demonstration in Unity is included in Appendix C.

4.3.4. Task planner node

A custom ROS C++ node was developed for planning robot motions based on the MTDF task recordings. The node was appropriately named "task planner". It provides a service called *compute_motion_plan_from_task*, which operates using the following request/response types:

- *TaskPlanningRequest*, which contains: *task_description*, the YAML string of MTDF; and *task_origin*, the pose of the task relative to the base link of the robot.
- *TaskPlanningResponse*, which contains:*trajectory*, a JointTrajectory message with the motion plan for the robot to execute the given task; and *error_code*, which indicates whether the solution has been found successfully.

Demonstrations recorded using Unity VR interface are saved in MTDF format, and forwarded to compute_motion_plan_from_task service. The node parses the incoming MTDF from the YAML string, and generates MTC stages based on the keyframes included in it. YAML parsing was implemented using the open-source library called yaml-cpp[50]. After all stages of the task are generated, MTC computes the solution, producing a motion plan for the robot, which is then converted into *JointTrajectory* message and sent back to Unity as a part of the service response. Generated trajectories can be used in Unity to run the program on the digital twin of the robot, thus completing the cycle of LfD.

4.3.5. Motion planning server

All motion planning capabilities described in the above subsections are combined together in a single instance of ROS, which becomes a standalone module for solving LfD tasks. We call this module a "motion planning server". Thanks to MoveIt framework, the server can be configured to work with any industrial robot (using its URDF) while providing the same planning API. Because Unity application can communicate with multiple ROS instances over the network (detailed in subsection 5.1), the planning server can be run on a separate Linux machine or inside a Docker container on any OS. This completes the first module of our system.

4.4. Physical robot control

The second ROS module in our prototype is responsible for running the generated programs on the physical robot. While the LfD system does not depend on it (generated programs can be run on the simulated robot), this module becomes necessary when it comes to executing the task on the real hardware.

4.4.1. ROS-Industrial drivers

Control of the physical robot is another area where ROS-Industrial brings big value. The standard includes the specification of a universal interface for industrial robot drivers[51]. Robot manufacturers are responsible for integrating their robot control code with *industrial_robot_client* ROS package[52], which serves as a base implementation of the specified driver interface. Multiple vendors have already implemented such packages for their flagship controller boards.

ROS-Industrial driver package is supposed to be installed directly onto the controller board of the industrial robot to have immediate access to the machine. It communicates with the control node in ROS instance connected to the robot through the local network, and the control node provides a common API which is used to send commands to the robot and monitor its state. While all API features are outlined in [51], the ones important to our implementation of robot control are:

- *joint_states*, a topic providing stream of up-to-date joint positions of the robot (based on URDF joint names).
- *joint_path_command*, a service accepting the list of trajectory points to be executed on the robot (based on URDF joint names).
- *stop_motion*, a service which allows to stop the execution of the current robot motion.

Using the features listed above, it is possible to upload the trajectories generated from the LfD task to the physical robot and monitor their execution, while keeping our code hardware-agnostic (as the robot structure is dynamically loaded from URDF).

4.4.2. Control server

ROS-Industrial robot control node can be combined with MoveIt simulation for collision-aware control of the robot. This feature is officially supported by ROS-Industrial, and manufacturers are encouraged to provide MCP for each of their robots; however, this requirement is not strict and some robot support packages come without MCP. Nevertheless, MCP can be generated based on the URDF provided with each ROS-Industrial robot package. This was done in our prototype.

Robot control node from ROS-Industrial and MoveIt simulation are combined in a single instance of ROS, which we call "robot control server". It serves as an access point for cotrolling a single physical robot, allowing us to test the generated LfD solutions on the real hardware connected over the network. While not necessary for the LfD cycle itself, having robot control server module in our prototype allowed us to validate the generated programs on the real hardware, as described in section 6.

5. Unity: task recording interface

5.1. Communication between Unity and ROS

ROS does not provide out-of-the-box means to communicate with software written in C programming language. Thus, to let our Unity application interact with ROS motion planning and robot control servers, a custom solution had to be found. It was decided to utilize an open-source project called *ROS-TCP-Connector*[53], developed by Unity for this purpose.

ROS-TCP-Connector is based on ROS#[25], and allows to communicate with ROS's topics, services and actions directly from C# code. As reflected in the name of the library, connection with ROS gets established through Transmission Control Protocol (TCP), which guarantees that all messages passed through the communication tunnel get delivered to the other side. To start the communication, Unity application has to be provided with Internet Protocol (IP) address and port. In ROS-TCP-Connector, each individual connection is represented by a *ROSConnection* component (i.e. a Unity C# script), which can be configured to connect to specific IP (see Figure 4).

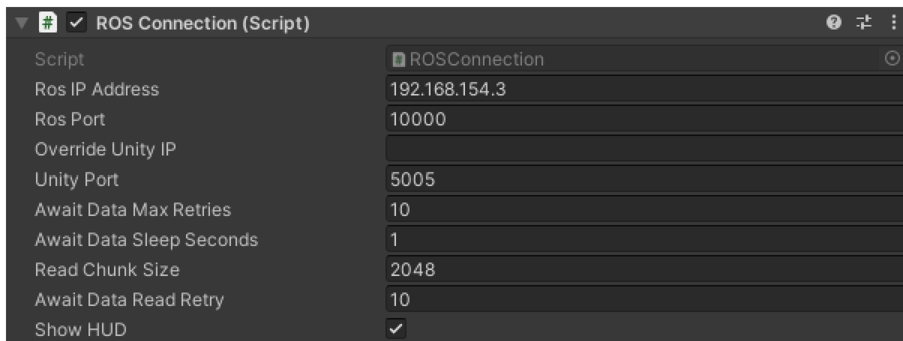


Figure 4. *Configuration parameters of ROSConnection component*

To correctly serialize the messages sent to and received from ROS in C#, each message type has to be represented as a C# class. ROS-TCP-Connector takes this burden off the programmer by automatically generating C# classes based on existing ROS **.msg*, **.srv* and **.action* files (see more about these file types in subsection 4.2). Automatic generation can be done using "ROS Message Browser" tool window (included with ROS-TCP-Connector, accessed through "Robotics/Generate ROS Messages..."; see Figure 5).

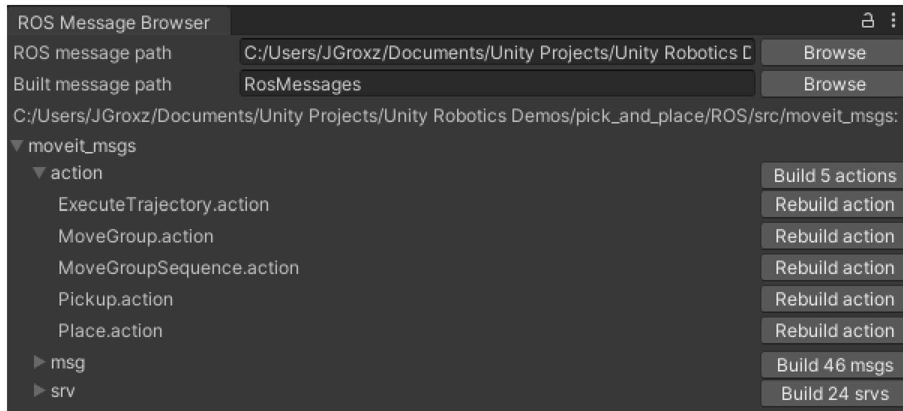


Figure 5. *"ROS Message Browser" window showing 5 MoveIt action types compiled to C# classes*

After the connection settings were set and message classes were generated, the prototype was able to successfully communicate with our ROS servers in the local network.

5.2. Robot representation in Unity

While ROS robot representation is responsible for motion planning (see subsection 4.1), its counterpart in Unity carries out two other tasks: visualization and simulation of planned motions in the digital twin environment. At the same time, it was crucial to make the process of importing robot model to Unity as frictionless as possible. Functionality used to achieve these criteria is described below.

5.2.1. URDF import

The main design goal for Unity robot representation was to make it automatically generatable from the same URDF file used in ROS (as described in subsection 4.1). This was successfully achieved using "URDF Importer" tool by Unity[54], based on ROS#. This tool generates a model of the robot in Unity's 3D scene based on the kinematic structure and models provided in URDf file, as can be seen in Figure 6.

The virtual robot in Unity is constructed from *GameObjects* — primary building blocks in Unity engine, which represent 3D Transforms with optional components (i.e. C# scripts) attached to them to define extra functionality. In order to preserve

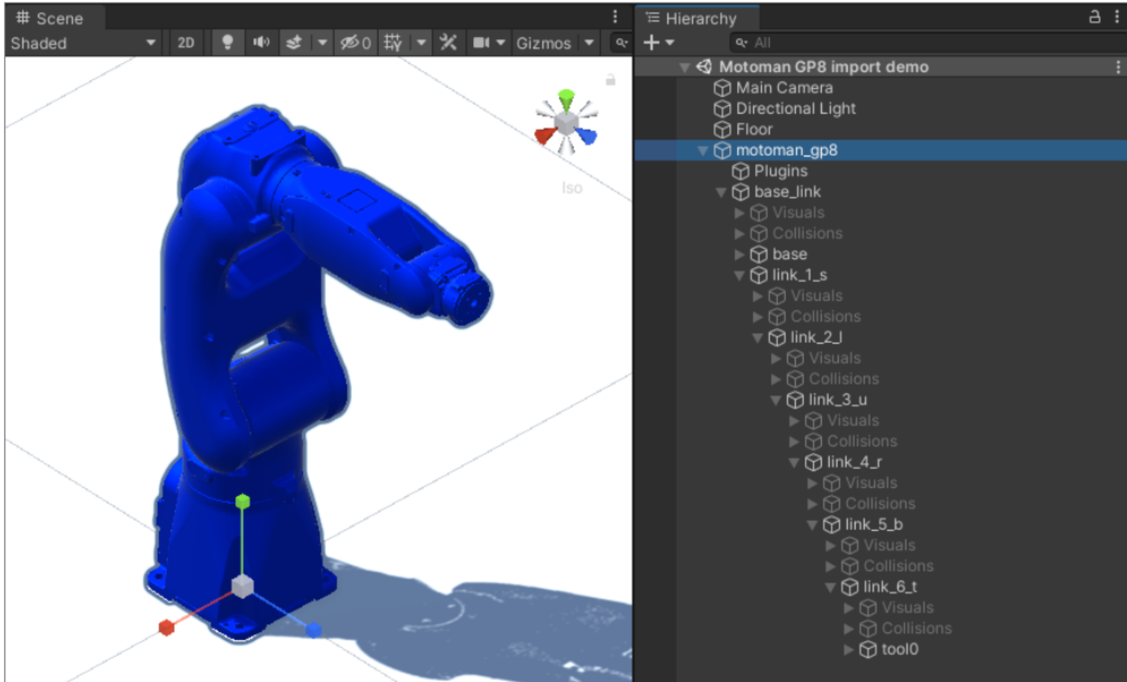


Figure 6. *Motoman GP8 robot imported into Unity Scene from its ROS-Industrial URDF file. Note that the robot's structure in Hierarchy tab (on the right) follows the one from URDF, including geometry containers for "Visuals" and "Collisions"*

information from the original URDF file, the generated robot model has several components attached to it. These include:

- *UrdfRobot*, manager script which holds reference to the original URDF file and provides multiple controls for adjusting the imported model. It also provides a function to *export* the robot back into URDF.
- *UrdfLink*, which holds data about a single robot link.
- *UrdfJoint*, which hold data about a single robot joint. There exist several subclasses of *UrdfJoint* used to represent *Revolute*, *Linear* and *Fixed* joint types from URDF specification.
- *ArticulationBody*, script responsible for physical simulation of an individual joint. It defines joint articulation limits and effort used in the simulation, corresponding to the original values set in the URDF. *ArticulationBody* scripts, along with collision geometry attached to the links, provide a way to simulate movement and interactions of the robot with objects in the virtual environment (e.g. grasping).

An example of the robot's link GameObject with attached components can be seen in Figure 7.

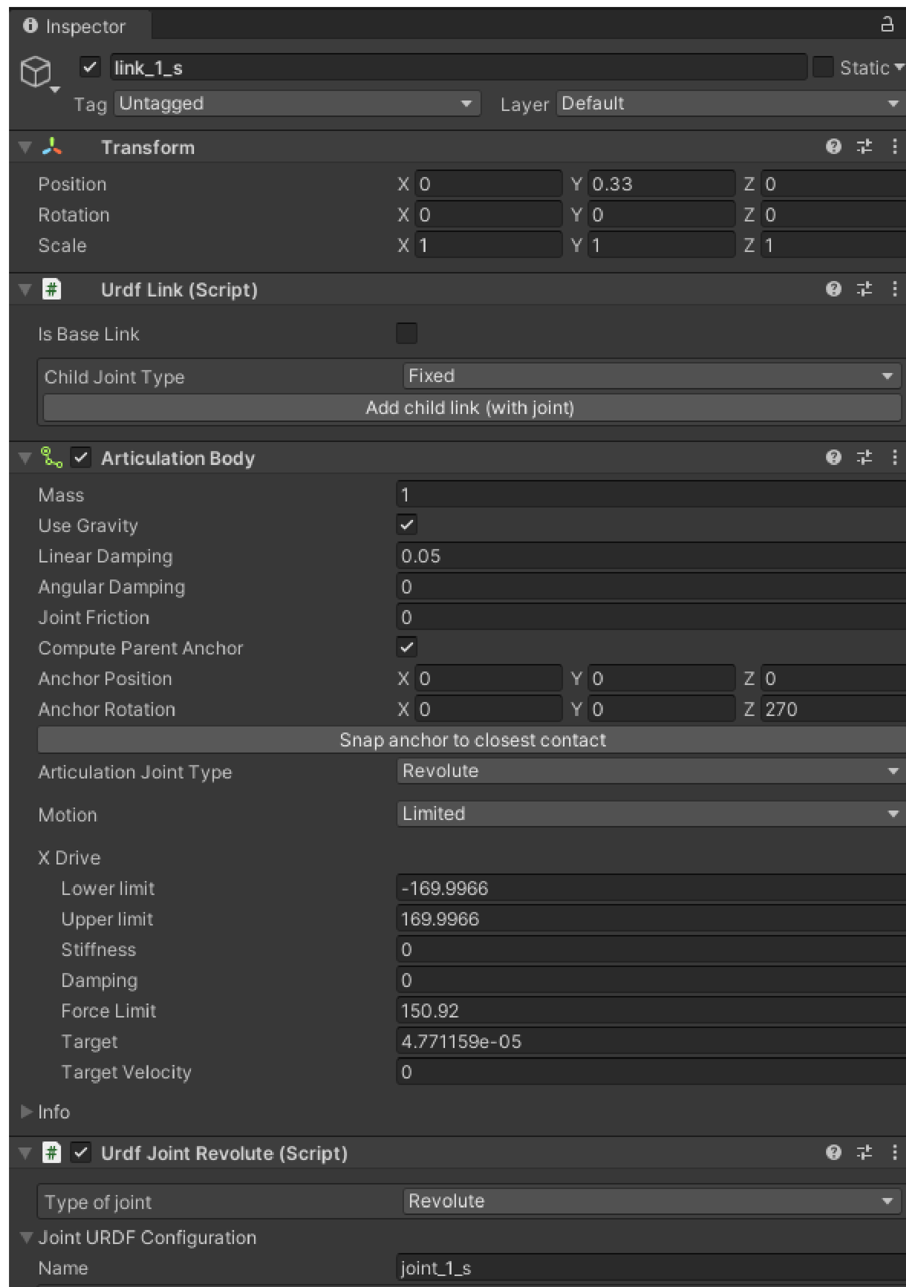


Figure 7. *ArticulationBody and UrdfJointRevolute attached to "link_ 1_ s" of the imported Motoman GP8*

To make it easier for the robot control scripts to access kinematic information of the robot (i.e. its joints and links), we created another component called *RosRobotKinematicsDataProvider*. This script automatically collects the lists of references to joints, links and their respective names, which makes it easy to query and set the state of each part of the virtual robot from other C code. This component is utilized in robot

control scripts, which are described next.

5.2.2. Robot control scripts

Following our "hardware-agnostic" criteria, it was compulsory to provide functionality for testing the programs generated by the motion planning server on the virtual robot, independently from the physical hardware. For this purpose, a script called *VirtualRosRobotController* was created. This script was based on *RobotController* family of scripts proposed in the author's previous research on digital twins in Unity[55], with certain modifications for easier interoperability with ROS. *VirtualRosRobotController* provides methods to execute arbitrary motions on the virtual robot by providing it with ROS *JointTrajectory* messages (the same message type is used by MoveIt planning pipeline and ROS-Industrial robot drivers). The trajectories get executed on the virtual robot using the *ArticulationBody* scripts attached to the model, ensuring that movement is coherent with Unity's physics simulation and allowing the robot to interact with virtual objects (e.g. pick them up using its gripper).

To allow for digital twin operation mode, i.e. when the virtual robot mirrors the motions of the physical one, another script called *DigitalTwinRosRobotController* was added to the system. It is based on the communication with ROS-Industrial robot control node API (described in subsection 4.4). The script listens to the state of the physical robot through *joint_states* topic, and updates virtual robot joint positions accordingly; control commands passed to *DigitalTwinRosRobotController* methods get forwarded to *joint_path_command* service, which executs them on the physical robot. In the prototype, digital twin control mode was used to test the generated programs on the real hardware, while observing the robot from the VR environment (which also allows for remote operation scenarios).

5.3. Environment synchronization

As explained in subsection 4.2, it is more efficient to construct the MPS in ROS by automatically synchronizing it with the Unity Scene instead of building the environment from scratch in Rviz. To take the most out of such approach, the synchronization system had to be:

- Selective — it should be possible to filter which objects get synchronized with MoveIt; the digital twin environment in Unity may be much vaster than the working envelope of the single robot being programmed, and adding the objects not reachable by this robot to MPS would be a waste of processing power and bandwidth.
- Efficient — the system should preserve bandwidth where possible to allow for faster synchronization; geometry already uploaded to MPS should not be sent again on the next synchronization — instead, it is sufficient to send only what changed since the last update (e.g. if the object was moved in Unity, send the command to update the position of the mesh in MPS instead of reuploading the whole object's mesh in the new position).

Let us see how these two criteria were met in our implementation.

Before writing the synchronization logic, it is necessary to decide what has to be synchronized — i.e., derive a mapping between Unity's and MoveIt's representation of a 3D scene. In MPS, 3D objects used for collision checking during motion planning are called "collision objects", and are represented by *CollisionObject* ROS messages. *CollisionObject* message contains data about solid primitives and meshes constituting the object, as well as their poses relative to the object's origin. Collision objects of MPS are directly visualized in Rviz. In Unity, visible geometry is represented by *MeshRendereer* components, which can efficiently render complex meshes but do not take part in physics simulation, and thus do not represent the physical form of virtual environment. Instead, collision volumes used in Unity physics engine are represented by *Collider* components, which can use either solid primitives or mesh geometry. Thus, it was decided to map Unity *Collider* components onto *CollisionObject* messages and upload them to MPS.

To mark individual objects for synchronization, a component called *MoveItCollisionObjectAlias* was created. When attached to a GameObject, this script automatically collects all Collider components belonging to this GameObject and its children. All collected colliders are stored in a list, which can be mapped to a single *CollisionObject* message for use in MPS. *MoveItCollisionObjectAlias* provides a straightforward way to mark selected GameObjects for synchronization, thus fulfilling the "Selective" criteria pointed out above.

The synchronization logic is contained in a separate script called *MoveItSceneSyn-*

chronizer. This script handles conversion *MoveItCollisionObjectAliases* into MoveIt's *CollisionObjects* and their uploading to ROS. Each instance of *MoveItSceneSynchronizer* holds reference to its own *RosConnection*, MPS name, robot digital twin in the form of *RosRobotKinematicsDataProvider* and a list of *MoveItCollisionObjectAliases* to be uploaded into the named MPS on the respective ROS server. Having configurable parameters allows to have several synchronizers dedicated to different robots (and to respective motion planning servers) in the same digital twin environment. An example of a configured synchronizer can be seen in Figure 8.

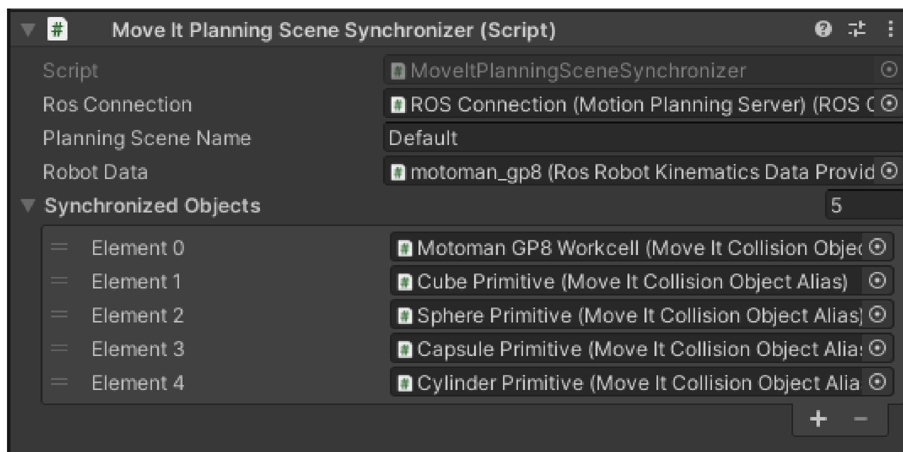


Figure 8. *Interface of MoveItPlanningSceneSynchronizer component. Five objects marked for synchronization with MoveIt can be seen in the "Synchronized Objects" list at the bottom of the image*

An important aspect to take care of when converting Unity's 3D geometry and transforms into ROS is that the two applications use different coordinate systems. Unity coordinate system is left-handed (X right, Y up, Z forward), while the most common coordinate system used in ROS is right-handed (X forward, Y left, Z up). Fortunately, ROS-TCP-Connector package includes code for converting 3D positions and rotations between the two coordinate systems, so it was utilized in the prototype.

To correctly construct *CollisionObject* messages, each type of Unity's Collider had to be mapped to the corresponding primitive or mesh in ROS. Elements of *CollisionObjects* in MPS are described using *Shape* message type from the standard ROS shape_msgs package[56]. Certain primitive shapes supported in *Shape* message are not available in Unity, namely a cylinder and a cone. Unity has a standard cylinder mesh included in the engine library, so if any MeshCollider is using it, it gets mapped to *SolidPrimitive.Cylinder* type. However, Unity does not have a cone mesh in its default library, so it is not possible to reliably map a collider to SolidPrimitive.Cone without analyzing the underlying mesh; thus, any cone meshes are treated as generic

Table 2. *Mapping of collision geometry from Unity to ROS*

Unity *Collider*	ROS *Shape*
BoxCollider	*SolidPrimitive.Box*
SphereCollider	*SolidPrimitive.Sphere*
CapsuleCollider	*Mesh* (capsule)
MeshCollider	*Mesh*
MeshCollider (cylinder)	*SolidPrimitive.Cylinder*
MeshCollider (plane)	*Plane*
-	*SolidPrimitive.Cone*

MeshCollider and mapped to *Mesh* type in ROS. On contrary, Unity supports capsule primitive, which is not available in ROS. Thus, for each *CapsuleCollider*, a standard capsule mesh from Unity gets uploaded to *Mesh* in ROS. Full mapping of the 3D collision geometry, as it was implemented in *MoveItSceneSynchronizer*, can be seen in Table 2.

After each marked *Collider* is converted into ROS *Shape* message, it is included in the corresponding *CollisionObject* together with its pose in 3D space (converted to ROS coordinate system). All poses of the objects are given relative to the base link of the robot, because in MPS the robot base is located at the world origin point, i.e. [0;0;0]. Once all *MoveItCollisionObjectAliases* are converted into *CollisionObject* messages, they can be sent to the MPS. Triggering the synchronization from the current state of Unity Scene can be done simply by calling *Synchronize()* method on *MoveItSceneSynchronizer*. The synchronizer automatically keeps internal track of the objects already uploaded to the MPS, and only synchronizes the changes (e.g. an updated position of the object) with ROS, thus fulfilling the "Efficient" criteria. It should also be noted that the *Synchronize()* method is not getting called automatically by the synchronizer itself — instead allowing to toggle MPS update from other scripts when needed. In our prototype, synchronization happens exactly before the MTDF is sent to the motion server for processing. Result of the demo environment synchronization can be seen in Figure 9.

In the prototype, *MoveItSceneSynchronizer* has successfully demonstrated that MPS can be generated automatically based on the existing 3D environment in Unity. This approach allows to use complex environments for planning in MoveIt without compromising the convenience and speed of authoring them in Unity Editor.

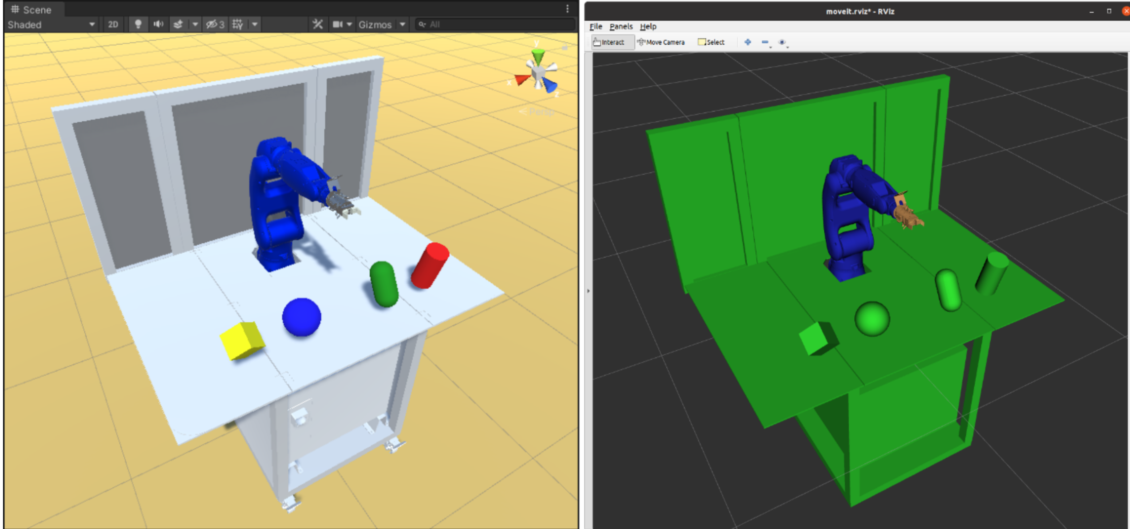


Figure 9. *Robot workcell with object primitives in Unity (left) mirrored into MoveIt Planning Scene in ROS (right)*

5.4. Task recording interface

The proposed interface for recording LfD demonstrations consists of the recording script, status UI and VR system which allows the user to interact with the environment.

VR functionality was implemented using Tilia framework[57] — the successor of Virtual Reality Toolkit, a substantial open-source framework for VR development in Unity. The framework is independent from hardware and works with any VR headset supported by Unity. In our prototype, Tilia provides input handling for VR HMD and hand controllers, user's avatar locomotion and logic for interactable objects in the environment. Interactables objects, often referred to as "interactables", are 3D objects which can be grabbed by the user using a hand controller. Grabbing functionality is the main advantage of using VR for demonstrations, because it allows the user to rearrange the virtual scene in a natural fashion. The moments when the user grabs or releases an interactable can be precisely detected in code, and directly mapped to "pick" and "place" keyframes of MTDF. This brings us to the recording logic.

The recording script, called *MtdfRecorder*, contains the core functionality for interpreting user actions into MTDF format. *MtdfRecorder* automatically collects references to objects which are both interactable and have *MoveItCollisionObjectAlias*

script attached to them. After the digital twin environment is set up and the user starts a demonstration by clicking "Start recording" button in the UI, the script starts monitoring. As the user grabs and releases the tracked objects, each such interaction gets mapped to the keyframe of MTDF in the recorder's memory. Once the demonstration is complete, the user clicks "Complete recording" button in the UI and *MtdfRecorder* saves the generated MTDF data into a Scriptable Object (SO) asset in the Unity project. *ScriptableObject* is a type of C# script which makes it convenient to serialize and edit custom data in the Unity project. It was intentionally chosen for storing MTDF in Unity, as it allows to easily load recorded MTDFs for analysis in other scripts. Nevertheless, each MTDF SO contains a method to convert it to a YAML string, which is exactly what happens when sending MTDFs to ROS planning server.

The recorded MTDF SOs are handled by the script called *MtdfManager*. This script automatically keeps track of the available MTDF SOs in Unity application, and provides methods for reviewing them and converting them to robot programs. *MtdfManager* requires a task reference frame (i.e. *Transform*) for MTDF, a reference to *RosRobotKinematicsDataProvider* of the robot for which the program must be generated, and a *RosConnection* linked to the motion planning server. Once these parameters are set, the manager can convert any MTDF to the motion plan for execution on the given robot. It must be noted that before sending MTDF to the task planner node, the manager also triggers synchronization with MPS to set up correct collision environment before planning.

The status UI of the prototype provides minimal functionality sufficient for using the features described in the above paragraphs. User can use the UI to start and stop task recordings, review the created MTDFs and convert them into robot motion plans. UI also displays a live list of generated MTDF keyframes during the recording for user's reference.

6. Use-case: testing LfD system prototype

In order to validate the developed prototype, it was tested with two robots supported by ROS-Industrial: Motoman GP8 and ABB IRB-1600. This specific choice of robot models was based on the fact that both robots were available in the Industrial Virtual and Augmented Reality (IVAR) Laboratory in TalTech. This provided the possibility to validate the generated programs not only in simulation, but also on the physical hardware. In addition, a digital twin of the IVAR Laboratory environment was available as a part of the ongoing project on the digialization of the said laboratory.

The following steps were carried out to set up the system for each of the robots:

- Locate ROS-Industrial support package for the robot.
- Import robot's URDF into the digital twin environment in Unity.
- Add custom gripper to the robot and export it back to URDF.
- Generate MCP based on the exported URDF.
- Mark objects in the robot's work envelope in Unity for synchronization with MPS.
- Deploy motion planning server in the local network.

With the virtual robots set up, an example task was recorded, where the user stacked digital twins of 4 colored wooden boxes. An A4 paper-sized template layout was created to easily replicate the task starting condition when executing it in the real world. The template can be seen in Figure 10. Each step of the task is shown in Figure 11.

After the MTDF was generated, it was processed on both robots' motion servers to demonstrate that the system is capable of producting solutions for different robot models based on MTDF from a single demonstration. Motion plans were successfully generated for both robots, and snapshots of the programs' execution can be seen in Figure 12.

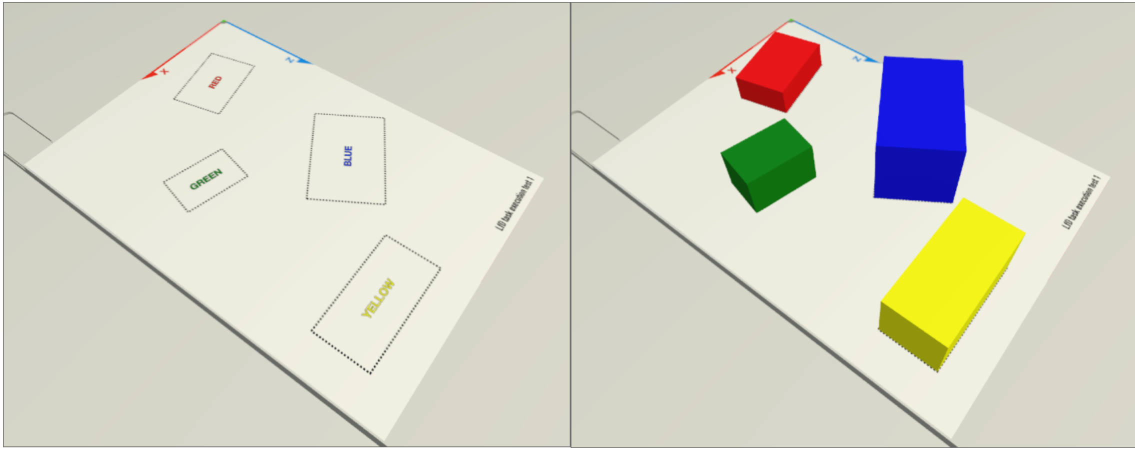


Figure 10. *Initial layout of the recorded task*

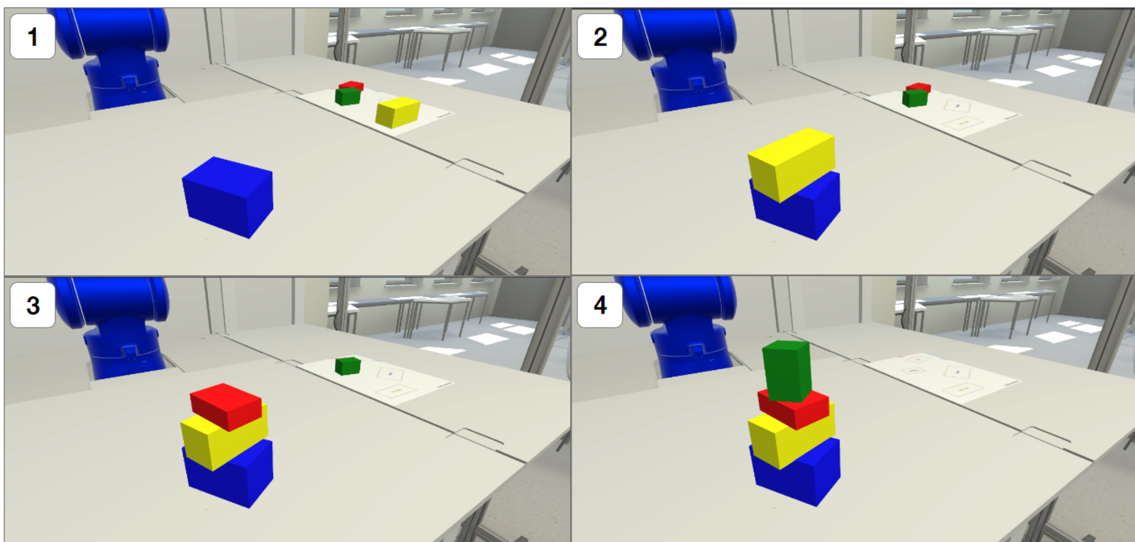


Figure 11. *Steps of the recorded task*

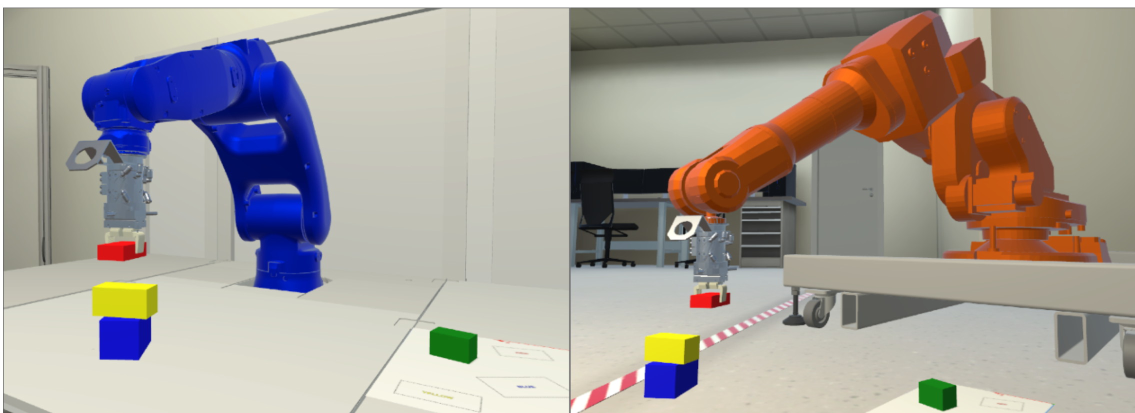


Figure 12. *Motoman GP8 (left) and ABB IRB 1600 (right) executing trajectories generated from the recorded task's MTDF*

Finally, the generated program was validated on the physical Motoman GP8 robot, The result was a successful execution of the demonstrated task in the real world. In shall be noted that the physical execution was tested only on Motoman because the ABB robot available in the laboratory lacked a gripper. Virtual version of ABB was equipped with the same gripper used on the physical Motoman robot, which demonstrates how LfD in simulation can be used for testing different robot configurations in absence of the physical hardware.

7. Discussion and future developments

This section draws suggestions regarding the future of ideas presented in this work based on the achieved results.

7.1. On the achieved goals

Several goals were defined in the beginning of this work (subsection 1.3). They have been successfully achieved:

- The prototype vividly presents the benefits of LfD programming in a simulated environment: software-only setup, independence from hardware sensors, unified mechanism for recording object manipulation tasks in a simulated 3D environment.
- The practical part of this thesis serves as detailed guideline for implementing similar digital-twin based LfD systems.
- Presented VR interface allows the user to record demonstrations by intuitively manipulating the object in virtual environment, while the system automatically handles mapping of the user's actions to the keyframes relevant to the task.
- The prototype has successfully used a single demonstration recording to generate programs for two different industrial robot models, which were imported to the system without any modifications to the code.

Now, let us complete the final goal by setting the course for the future development.

7.2. Directions of future research

A working prototype of hardware-agnostic simulation-based LfD system has been complete. However, the author believes that the presented system has a lot of room for improvement. Thus, let us outline potential improvements for the system which deserve to be implemented.

7.2.1. Automatic deployment pipeline for ROS motion planning servers

One of the key motivations behind this research was the desire to make the process of robot programming simple and accessible to an average user without coding skills. The prototype requires the user to manually configure the ROS server, and work with ROS command line tools to generate MCP for the robot and launch MoveIt nodes needed for planning. However, with more automation, the system architecture presented in this thesis has the potential to provide a no-code programming experience. Configuration of the robot's URDF with custom grippers can already be done in Unity. Additional semantic data, such as motion planning groups, can be added to the model using Unity components. This means that the data for generating MCP is available on Unity side, and can be used to automatically configure motion planning server without demanding the user to directly interact with ROS. One can develop a ROS node which will generate MCP based on the URDF and semantic data uploaded from Unity, and launch the planning environment for the given robot. Furthermore, this node can be packaged in a Docker image with ROS and MoveIt preinstalled. This would result in a drop-in motion planning module which can run aside our Unity LfD application on any OS, without manual configuration.

7.2.2. Extending MTDF

The concept of MTDF, introduced in this work, has the potential to be greatly extended. As mentioned in subsection 4.3.3, object descriptors can be expanded to reference sets of rules used to identify objects in the virtual environment. This would allow to flexibly define the objects relevant to the task, using arbitrary features for identification (for example, the object could be identified by color or based on point cloud shape from a simulated sensor). Next, the library of keyframes can be extended to allow for more complicated manipulation scenarios. One of the possible suggestions is to introduce a "trajectory" keyframe type, which would define a list of poses which have to be followed by the manipulated object. This could be useful for tasks where the robot has to execute specific motions with a tool (for example, grab a marker and execute a drawing with it). With more object descriptors and keyframe types added in the future, MTDF can become a generic format for high-level description of manipulation tasks, which can then be translated to low-level motions of any robotic manipulator.

7.2.3. Migration to ROS 2

Although ROS 1 is currently the most stable and widespread version of the software, ROS 2 will inevitably become the new industry standard in the upcoming years. Migration to ROS 2 will bring several benefits which can be harnessed in our implementation. Among them is improved performance, which can allow for faster robot and scene synchronization as well as quicker generation of LfD task solutions by MTC library. Another crucial upcoming feature is the ability to launch the nodes in explicitly defined order — it will simplify the deployment of ROS-TCP-Connector nodes, which in our case depend on the topics published by the motion planning pipeline. The software projects from our solution are already have development versions supporting ROS 2, including MoveIt and ROS-Industrial packages. Thus, it is planned to adapt the developed system to ROS 2 in the close future.

7.2.4. System performance evaluation experiment

In order to objectively assess the efficiency of digital twin-based LfD approach, a experiment comparing it to conventional methods of robot programming should be performed. While, theoretically, the prototype system can provide significant boost to programming speed for the users when compared to traditional coding approach, the time required to set up the digital twin environment has to be accounted for. The efficiency of the system will also likely depend on the frequency of reprogramming required in a specific industrial scenario. Due to the time constraints and pandemic situation at the time of writing of this thesis, the performance evaluation experiment has not been attempted, thus leaving it to the future research.

8. Summary

In this thesis, a concept and implementation of hardware-agnostic digital twin-based LfD system for industrial robots were suggested. An overview of the state of the art in LfD literature was presented, outlining strengths and weaknesses of the existing solutions. Based on the collected data, passive observation was chosen as the most efficient, albeit complex, LfD approach. It was reasoned that the limitations of passive observation LfD can be countered by using VR environment for task demonstrations. An architecture of the system was proposed, relying on the common software, with the focus on replicability and independence from hardware. The prototype of the system was successfully implemented and validated with two different robot models in the digital twin of TalTech IVAR Laboratory. Finally, the possible directions of further development were defined.

The result of this work confirms the assumption that passive observation approach to LfD can be empowered by running demonstrations in a pure simulated environment, and does not have to be restricted by the robotic hardware available to the user. Combined with the fact that it was developed using common and open-source software components, the proposed system is indeed a novel outlook on the LfD paradigm, which is also friendly to being reproduced and extended by other developers.

As a conclusion, the author sincerely hopes that this work will pave the way for a whole set of future research endeavors crossing the areas of LfD and digital twins, which will in turn lead us to more accessible an effective robot programming.

References

[1] H. Ravichandar, A. S. Polydoros, S. Chernova, and A. Billard, "Recent advances in robot learning from demonstration," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 3, no. 1, pp. 297–330, may 2020, overview of current state of Learning from Demonstration in robotics.

[2] D. Halbert, "Programming by example," Ph.D. dissertation, University of California, Berkeley, 1984.

[3] "Niryo one," an affordable 6 DoF robot arm. [Online]. Available: https://niryo.com/product/niryo-one/

[4] "Pi arm," an affordable 6 DoF robot arm based on Raspberry Pi. [Online]. Available: https://shop.sb-components.co.uk/products/piarm-the-diy-robotic-arm-for-raspberry-pi

[5] "About ros." [Online]. Available: https://www.ros.org/about-ros/

[6] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. Shepherd, and D. Franklin, "Evaluating CoBlox," in *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, apr 2018.

[7] "Omron tm-flow software." [Online]. Available: https://automation.omron.com/en/us/products/family/OmronTMSoftware

[8] Y. Bondarenko, "Simulation-based lfd demo," 2021, demo implementation of the system developed in the scope of this thesis. [Online]. Available: https://gitlab.com/IVAR_Lab/simulation-based-lfd-demo

[9] A. Sugiura and Y. Koseki, "Simplifying macro definition in programming by demonstration," in *Proceedings of the 9th annual ACM symposium on User interface software and technology - UIST '96*. ACM Press, 1996.

[10] M. M. Zloof, "Query-by-example: A data base language," vol. 16, pp. 324–343, 1977.

[11] T. Zhang, Z. McCarthy, O. Jow, D. Lee, X. Chen, K. Goldberg, and P. Abbeel, "Deep imitation learning for complex manipulation tasks from virtual reality teleoperation," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2018.

[12] K. Dautenhahn and C. L. Nehaniv, *Imitation in Animals and Artifacts.* MIT Press, 2002, "Correspondence problem" definition in Chapter 2.

[13] "Ati multi-axis force/torque sensor system," example of a sensor kit used to enable kinesthetic teaching in industrial robots. [Online]. Available: https://www.ati-ia.com/products/ft/sensors.aspx

[14] D. Whitney, E. Rosen, D. Ullman, E. Phillips, and S. Tellex, "Ros reality: A virtual reality framework using consumer-grade hardware for ros-enabled robots," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE, oct 2018.

[15] V. Kuts, G. E. Modoni, T. Otto, M. Sacco, T. Tähemaa, Y. Bondarenko, and R. Wang, "Synchronizing physical factory and its digital twin through an iiot middleware: a case study," vol. 68, p. 364, 2019.

[16] M. J.-Y. Chung, M. Forbes, M. Cakmak, and R. P. N. Rao, "Accelerating imitation learning through crowdsourcing," in *2014 IEEE International Conference on Robotics and Automation (ICRA).* IEEE, may 2014.

[17] R. Toris, D. Kent, and S. Chernova, "Unsupervised learning of multi-hypothesized pick-and-place task templates via crowdsourcing," 2015.

[18] Y. Liu, A. Gupta, P. Abbeel, and S. Levine, "Imitation from observation: Learning to imitate behaviors from raw video via context translation," in *2018 IEEE International Conference on Robotics and Automation (ICRA).* IEEE, may 2018.

[19] E. Glaessgen and D. Stargel, "The digital twin paradigm for future nasa and u.s. air force vehicles," 2012.

[20] Y. Chen, "Integrated and intelligent manufacturing: Perspectives and enablers," *Engineering*, vol. 3, no. 5, pp. 588–595, oct 2017.

[21] J. Spranger, R. Buzatoiu, A. Polydoros, L. Nalpantidis, and E. Boukas, "Human-machine interface for remote training of robot tasks." in *2018 IEEE International Conference on Imaging Systems and Techniques (IST).* IEEE, oct 2018.

[22] D. Whitney, E. Rosen, E. Phillips, G. Konidaris, and S. Tellex, "Comparing robot grasping teleoperation across desktop and virtual reality with ROS reality," in *Springer Proceedings in Advanced Robotics.* Springer International Publishing, nov 2020, pp. 335–350.

[23] "Unity engine website." [Online]. Available: https://unity.com/releases/release-overview

[24] "Nvidia physx 4.0 release announcement," physX 4.0 is the default physics solver in current versions of Unity engine. [Online]. Available: https://developer.nvidia.com/blog/announcing-physx-sdk-4-0-an-open-source-physics-engine/

[25] "Ros," gitHub repository of the ROS project by Siemens. [Online]. Available: https://github.com/siemens/ros-sharp

[26] "Unity robotics hub," repository with demo projects integrating Unity with ROS, developed by Unity themselves. [Online]. Available: https://github.com/Unity-Technologies/Unity-Robotics-Hub

[27] "Robots of ros," official public list of robots supported by ROS. [Online]. Available: https://robots.ros.org/

[28] "Ros-industrial official website." [Online]. Available: https://rosindustrial.org/

[29] "Ros-industrial supported hardware," rOS Wiki page listing industrial robot vendors supporting ROS-Industrial initiative. [Online]. Available: http://wiki.ros.org/Industrial/supported_hardware

[30] I. A. Sucan and S. Chitta, "Moveit." [Online]. Available: https://moveit.ros.org/

[31] T. Coleman David, "Reducing the barrier to entry of complex robotic software: a moveit! case study," 2014.

[32] "Universal robot description format (urdf)." [Online]. Available: http://wiki.ros.org/urdf/XML/model

[33] "Xacro (xml macros)." [Online]. Available: http://wiki.ros.org/xacro

[34] "Semantic robot description format (srdf)." [Online]. Available: http://wiki.ros.org/srdf

[35] "Moveit setup assistant documentation." [Online]. Available: https://ros-planning.github.io/moveit_tutorials/doc/setup_assistant/setup_assistant_tutorial.html

[36] "Moveit grasps documentation." [Online]. Available: https://ros-planning.github.io/moveit_tutorials/doc/moveit_grasps/moveit_grasps_tutorial.html

[37] M. Lautman, "Introducing MoveIt grasps, a manipulation framework tightly integrated with MoveIt." oct 2019.

[38] "Moveit deep grasps documentation." [Online]. Available: https://ros-planning.github.io/moveit_tutorials/doc/moveit_deep_grasps/moveit_deep_grasps_tutorial.html

[39] "Grasp pose detection library," library for detecting robot gripper grasps in point clouds using convolutional neural network. [Online]. Available: https://github.com/atenpas/gpd

[40] "Dexterity network (dex net)," convolutional neural network for extracting robot gripper grasp poses for objects from camera images. [Online]. Available: https://berkeleyautomation.github.io/dex-net/

[41] "Moveit planning scene documentation." [Online]. Available: https://ros-planning.github.io/moveit_tutorials/doc/planning_scene/planning_scene_tutorial.html

[42] "Moveit planning scene ros api." [Online]. Available: https://ros-planning.github.io/moveit_tutorials/doc/planning_scene_ros_api/planning_scene_ros_api_tutorial.html

[43] "Rviz," 3D visualization tool for ROS. [Online]. Available: http://wiki.ros.org/rviz

[44] "Moveit ros message types." [Online]. Available: http://docs.ros.org/en/api/moveit_msgs/html/index-msg.html

[45] "Orocos kinematics and dynamics library," one of the IK plugin options in MoveIt. [Online]. Available: https://www.orocos.org/kdl.html

[46] "Openrave," a motion planning library which includes IKFast, one of the IK plugin options in MoveIt. [Online]. Available: http://openrave.org/

[47] "Moveit kinematics configuration." [Online]. Available: https://ros-planning.github.io/moveit_tutorials/doc/kinematics_configuration/kinematics_configuration_tutorial.html?highlight=kinematics

[48] "Trac-ik," one of the IK plugin options in MoveIt. [Online]. Available: https://bitbucket.org/traclabs/trac_ik

[49] M. Gorner, R. Haschke, H. Ritter, and J. Zhang, "MoveIt! task constructor for task-level motion planning," may 2019.

[50] "yaml-cpp," c++ library for working with YAML format. [Online]. Available: https://github.com/jbeder/yaml-cpp.git

[51] "Ros-industrial robot driver specification." [Online]. Available: http://wiki.ros.org/Industrial/Industrial_Robot_Driver_Spec

[52] "industrial_robot_client package from ros-industrial." [Online]. Available: http://wiki.ros.org/industrial_robot_client

[53] "Ros-tcp-connector." [Online]. Available: https://github.com/Unity-Technologies/ROS-TCP-Connector

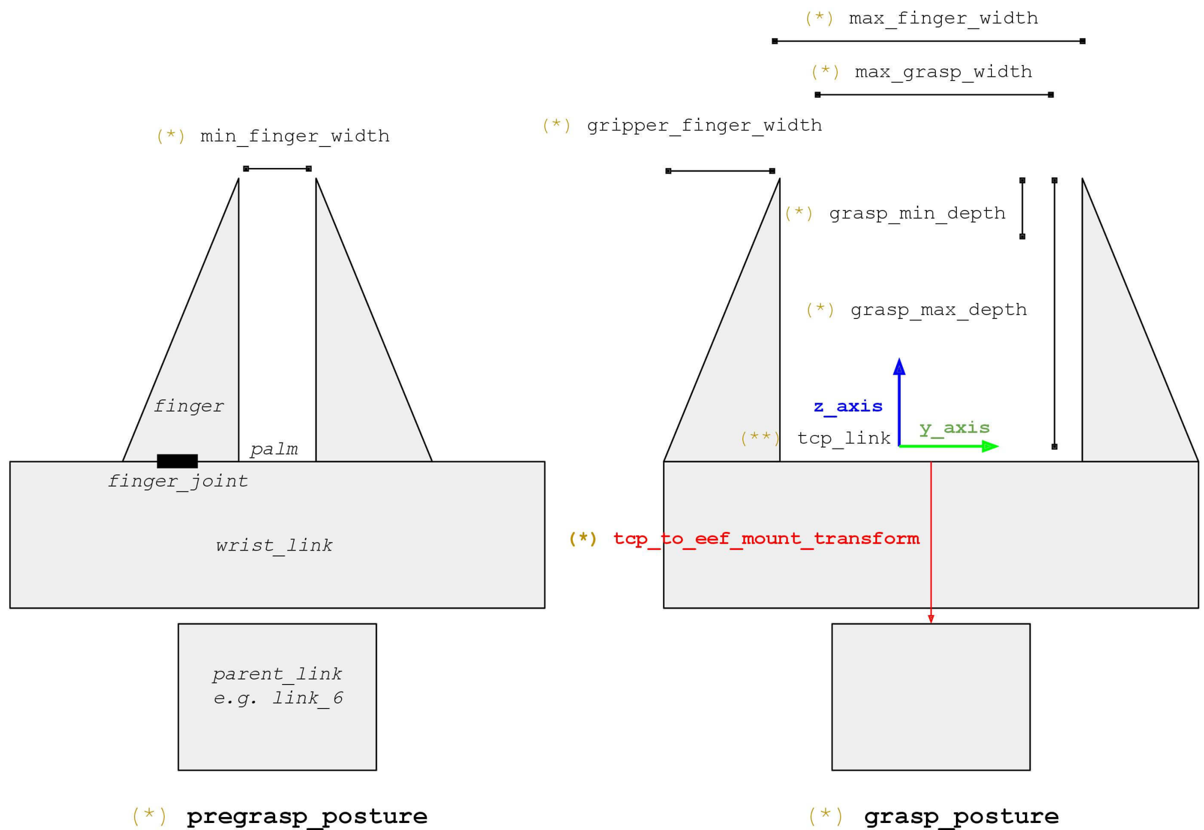
[54] "Unity urdf importer." [Online]. Available: https://github.com/Unity-Technologies/URDF-Importer

[55] Y. Bondarenko, "Digital twin communication and control bridge development using virtual reality – industrial robot use case," 2018. [Online]. Available: https://digikogu.taltech.ee/en/Item/bc59417d-007b-439d-9e52-2bb266d4b1b8

[56] "Ros shape_msgs package." [Online]. Available: http://wiki.ros.org/shape_msgs

[57] "Tilia," a framework for the development of spatial computing applications in Unity engine. [Online]. Available: https://www.vrtk.io/tilia.html

A. Geometry parameters for *grasp_ data.yaml* files for two-finger grippers



The original image is available in [36].

B. Example of *grasp_ data.yaml* (from use-case)

```
base_link: world
hand:
    # Name and EEF type
    end_effector_name: hand
    end_effector_type: finger

    # Finger gripper parameters
    max_finger_width: 0.049
    min_finger_width: 0.030
    max_grasp_width: 0.044
    gripper_finger_width : 0.006

    # Joint names and state values for the EEF
    joints : [ joint_left_finger, joint_right_finger ]
    pregrasp_posture : [ 0.04, -0.04 ]
    grasp_posture : [ 0.0, 0.0 ]

    # Distance from the mount to the palm of EEF [x, y, z, r, p, y]
    tcp_to_eef_mount_transform :  [ 0, 0, 0, 0, 0, 0 ]

    # Planning resolution parameters
    grasp_resolution : 0.05
    angle_resolution : 90
    grasp_max_depth : 0.006
    grasp_min_depth : 0.003
    grasp_depth_resolution : 1.0

    # Grasp motions parameters
    approach_distance_desired: 0.05
    retreat_distance_desired: 0.05
    lift_distance_desired: 0.01
    grasp_padding_on_approach: 0.003
    pregrasp_time_from_start : 0.0
    grasp_time_from_start : 0.0
```

C. Example of MTDF YAML (from use-case)


```
object_descriptors:
  - red_box
  - green_box
  - blue_box
  - yellow_box

keyframes:
  - { type: pick, object_descriptor: blue_box }
  - { type: place, object_descriptor: blue_box,
      place_pose:
      { position: [ 0.534, 0.0237, 0.321 ]
        orientation: [ 1.4756, -0.2618, -6.0335, 0.9651 ] }
    }
  - { type: pick, object_descriptor: yellow_box }
  - { type: place, object_descriptor: yellow_box,
      place_pose:
      { position: [ 0.5332, 0.069, 0.3241 ]
        orientation: [ 0.0007, 0.5797, -0.0007, 0.8149 ] }
    }
  - { type: pick, object_descriptor: red_box }
  - { type: place, object_descriptor: red_box,
      place_pose:
      { position: [ 0.5345, 0.0988, 0.3279 ]
        orientation: [ 0.0005, 0.9934, 0.0004, 0.1143 ] }
    }
  - { type: pick, object_descriptor: green_box }
  - { type: place, object_descriptor: green_box,
      place_pose:
      { position: [ 0.5360, 0.1329, 0.3290 ]
        orientation: [ -0.651, -0.275, -0.2767, 0.6511 ] }
    }
```