UNIVERSITY OF BIELEFELD

TECHNICAL FACULTY

Design of an Interactively Parametrizable Robot Skill Architecture for the KUKA iiwa

Hendrik Oestreich

MASTER THESIS

INTELLIGENT SYSTEMS

supervised by

Dr. Sebastian WREDE

and

Michael WOJTYNEK

May 2017

Statement of authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged. This thesis has not been presented to an examination office in the same or a similar form yet.

Bielefeld, 22nd May 2017

Hendrik Oestreich

Acknowledgement

I am grateful for the supervision of this thesis by Dr.-Ing. Sebastian Wrede and M.Sc. Michael Wojtynek. Both of them always supported my work through valuable discussions and concrete advice.

A very special gratitude goes out as well to the other members of the Cognitive Systems Engineering group for all the support and collaboration during the last months. It was great sharing the laboratory and time with you. Thanks especially to M.Sc. Johannes Wienke who took some of the photos and was a great help for improving the code.

Furthermore I would like to thank the company HARTING and their employees for their trust, all the insights and for providing the hardware which made a concrete application of my work possible.

Finally, last but by no means least, I would like to thank my family and my friends for all their support during the writing of this thesis. I do not take this for granted and I hope I can make up for everything in the future. I really deeply appreciate all your help, good words and all the sacrifices you made for me.

Abstract

This thesis presents a new approach for a skill framework which is based on BPMN. Graphical modelling of complex processes simplifies their understanding and opens up new possibilities for robot programming.
Interactive parametrization combines human robot interaction and traditional interface based configuration to provide a good usability.
The skill framework consists of three layers with different abstraction levels that support reuse, scalability and modularity.
A real industrial assembly task is used for application of the framework and to evaluate strengths and weaknesses. Embedding the skill framework in an unified automation environment shows its potential to be used in production environments with steadily changing requirements for a flexible production.
The skills were tested on a lightweight robot especially designed for collaborative robotics, the *KUKA iiwa*. As a consequence fundamentals for a basic safety level for human robot collaboration have been considered and implemented.

Contents

1 Introduction **1**
- 1.1 General introduction . . . 1
- 1.2 FlexiMiR project description . . . 2
- 1.3 Goals . . . 3

2 Related Work **5**
- 2.1 History and Origins . . . 5
- 2.2 Skill Frameworks . . . 6
- 2.3 Proprietary Implementations . . . 7

3 Concept **9**
- 3.1 Skill Primitives . . . 10
- 3.2 Skills . . . 12
- 3.3 Tasks . . . 13
- 3.4 Composition of Skills and Tasks . . . 13
- 3.5 Concept for Realization . . . 14
  - 3.5.1 BPMN . . . 15
  - 3.5.2 Camunda . . . 16
  - 3.5.3 AWAre Framework . . . 16
  - 3.5.4 Kuka iiwa . . . 16
  - 3.5.5 Kuka Sunrise.OS . . . 18
- 3.6 Interactive Parametrization . . . 18
- 3.7 Task Frames . . . 20

4 Implementation **23**
- 4.1 Framework Architecture . . . 23
- 4.2 Skill primitives - Kuka specific . . . 26
  - 4.2.1 SkillProvider - RoboticsAPIApplication . . . 26
  - 4.2.2 Skill Callbacks . . . 27
  - 4.2.3 Skill Implementation . . . 28
- 4.3 Skill primitives - Generic implementation (AWAre Plugin) . . . 28
  - 4.3.1 Skill Delegate . . . 28
  - 4.3.2 Pre- and Postconditions . . . 29
  - 4.3.3 Robot Skill Plugin . . . 29
- 4.4 Skill and Task Composition . . . 30

4.5 Task Frames . . . 31
4.5.1 Kuka Scene Graph . . . 31
4.5.2 Neo4J - Persistent Storage . . . 32
4.6 Interactive Parametrization . . . 33
4.6.1 Teach Skill . . . 33
4.6.2 Teaching Task Frames . . . 34
4.6.3 Teaching Trajectories . . . 35

5 Application **37**
5.1 Production Environment and Components . . . 37
5.2 Modelled Assembly Process . . . 39
5.2.1 Exemplary Task composition . . . 40
5.3 Plug and Produce . . . 40
5.3.1 Localization of Objects . . . 41
5.4 Simulation and Motion Planning . . . 46
5.5 Safety . . . 47
5.5.1 Risk Assessment . . . 47
5.5.2 Kuka Safety . . . 48

6 Discussion **53**
6.1 Parametrization . . . 53
6.2 Robustness . . . 54
6.3 Composition . . . 55
6.4 World Model . . . 55
6.5 Safety . . . 56
6.6 Potentials and Limitations . . . 56
6.7 Summary . . . 57

7 Conclusion and Outlook **59**
7.1 Conclusion . . . 59
7.2 Outlook . . . 59

A Appendix **61**
A.1 Camunda Modeler . . . 61
A.2 AWAre GUI - Robot Control Center . . . 62
A.3 SICK Safety Designer . . . 63
A.4 Kuka iiwa . . . 64

B BPMN Diagrams **65**
B.1 Skill example 1 . . . 65
B.2 Skill example 2 . . . 65
B.3 Task example 1 . . . 66
B.4 Task example 2 . . . 66

C Code **67**
C.1 Repositories 67
C.1.1 AWAre Skill Implementation 67
C.1.2 Kuka Skill Implementation 67
C.1.3 FlexiMiR Application Project 67
C.2 Skill Callback Example 68
C.3 Skill Delegate Example 70
C.4 Skill Condition Example 73
C.5 Teach Skill 74
C.6 RST Examples 75
C.7 Class Diagram 78
References 80

List of Figures

1.1 Interactive Robotics 2

2.1 Intera Studio IDE - Graphical Programming 8

3.1 Skill Concept (Pedersen et al., 2016) 10
3.2 Conceptual Skill Hierarchy 14
3.3 The *Kuka iiwa* (KUKA, 2016) 17
3.4 Interactive Parametrization 19
3.5 Task Frame Hierarchy and Transformations 20
3.6 *HARTING* Production Cell 21

4.1 Software Architecture 23
4.2 UML Class Diagram 25
4.3 Task Frame Hierarchy in *Neo4j* View of *AWAre GUI* 32
4.4 Joint Angle Collections in *Neo4j* View of *AWAre GUI* 35

5.1 *HARTING* Production Cell 38
5.2 *HARTING Han variants* 39
5.3 Assembly Task 40
5.4 Measurement Order and Resulting Orientation 41
5.5 Measurements for Localization 42
5.6 Translations and Rotations 45
5.7 Calculated Trajectory from Carrier to IFM Component 46

A.1 Screenshot of the *Camunda Modeler* - General Tab 61
A.2 Screenshot of the *Camunda Modeler* - Field Injections Tab 61
A.3 Screenshot of the *AWAre GUI* 62
A.4 Configured Safety Zone of Laser Scanner 1 63
A.5 Configured Safety Zone of Laser Scanner 2 63
A.6 *Kuka iiwa* with Mounted Tools 64
A.7 *Kuka iiwa* Interactive Teaching 64

B.1 Pick Skill 65
B.2 Open Frame Skill 65
B.3 Assembly Process 66
B.4 Pick, Place and Open Frame 66

C.1 Genric Part of the Implementation 78
C.2 *Kuka*-specific part of the Implementation 79

List of Tables

3.1 Cartesian PTP Movement Skill Primitive 11

5.1 Measurements Localization 44
5.2 Collaborative Operations 48
5.3 Safety Configuration Kuka 52

Listings

5.1 Safety Override 50

C.1 Callback for a Cartesian PTP Movement 68
C.2 Delegate for a Cartesian PTP Movement 70
C.3 Gripper Closed Condition 73
C.4 Teach Skill 74
C.5 Cartesian Movement Datatype 75

1 Introduction

1.1 General introduction

In 1983 Lozano Perez[1] noticed that robot programming always needed an expert. Back then, he already pointed out that this is a significant drawback because it is though not feasible for the normal shop floor worker to adapt the robot behaviour.

[1] Lozano-Perez (1983, p. 821)

Nowadays, especially in Germany, *Industrie 4.0* is a very popular topic and and experts presume that production systems have to be adaptive to fulfil the quickly changing requirements for production. In a guideline for *Industrie 4.0* it is said, that "the benefit of Industrie 4.0 unfolds with a clever combination of already existing technologies"[2]. Since 1983 different kinds of frameworks for robot programming were developed and tested in a variety of scenarios. The problem persists that most of those frameworks are not well integrated in automation environments. In the final report about *Industrie 4.0* it is explained that machines in smart factories should communicate with human beings in a natural way[3]. This leads to the requirement that also robots as a production component have to be easy to program.

[2] Anderl et al. (2015, p. 7)

[3] Kagermann, Wahlster and Helbig (2013, p. 19)

The approach of this thesis is to design and implement a robot skill framework which enables the shop floor worker to design complex robot behaviours based on a simple graphical notation. As a result, the shop floor worker is able to react to flexible production environments and consequently adapts the robot behaviour. The term skill describes a capability or behaviour that the robot is able to execute. It might be adapted by parametrization, but each "skill" has its own characteristics which define the boundaries to separate it from other skill variants. Additionally other requirements for robot skills will be integrated, like force-based movements, persisting task frames and safety precautions for human robot interaction.

The thesis is resided in the context of the *it's owl* project *FlexiMiR*. *it's owl* is an abbreviation for "Intelligent Technical Systems Ostwestfalen-Lippe (a region in Germany)". This Leading-Edge Cluster encourages cooperations between companies and universities through various projects. *FlexiMiR* is one of the 34 innovation projects and the identifier is an abbreviation for "Flexible Assembly with integrated and interactive Robotics".

1.2 FlexiMiR project description

The FlexiMiR project is a cooperation between the Bielefeld University and the company HARTING. One goal of the project was to develop an architectural approach which continuously considers the requirements of interaction and (re)configuration of robots in the whole automation context and in combination with other components.

Another goal of the project was the exploration and integration of the *Plug and Produce* concept[4]. Intelligent components should be easily integrated into the production process and the robot should be used to localize those components in the production environment. Furthermore a modelling of the components is required and is a necessary prerequisite to allow collision free path planning for the robot in the production process.

[4] cf. Naumann, Wegener and Schraft (2007)

The company was responsible for developing models and software assistant systems which allow production oriented workforce planning and calculate the possible benefit through intelligent planning functions. Another aspect of the developed concepts is the changing of employee roles in the production. Today the machine setter is the person who configures a production machine and integrates all the components needed for the production process. Following the new approach, it will be more and more the task of the shop floor worker to adapt the machine to the production and integrate new components when necessary. This has to be evaluated in the future but will not be part of this thesis.

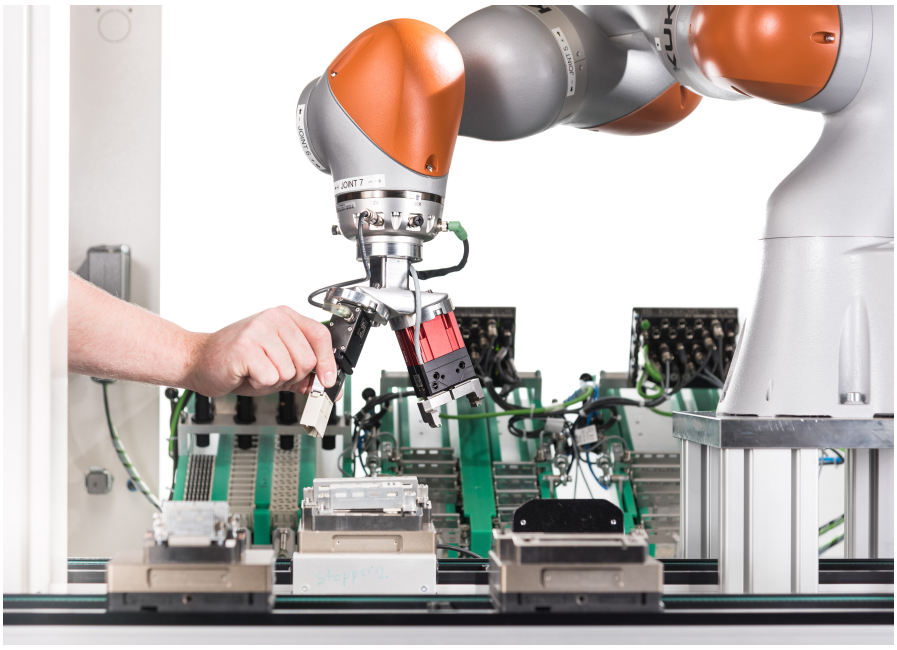


Figure 1.1: Interactive Robotics

1.3 Goals

The scope of my thesis will not include the modelling and simulation of the automation components. This as well as the path planning algorithms will be contributed by other collaborators and is only integrated through previously defined interfaces.

The thesis will focus on the design, implementation and application of the skill framework. Considering the actual requirements and goals of the framework, the following six topics are the most important ones:

Modularization: Skills should be modular. This allows reuse of skills in various contexts and improves the maintainability of the software.

Parametrization: The parametrization of skills protects the developer from designing too tailored skills for specific problems. Skills should be general and flexible, only their parametrization makes them distinct for the special context they are used in. Furthermore to provide a good usability, parametrization should be interactive. During the process modelling, the user should be guided through the parametrization by interactive teaching of positions and by being asked to provide other parameters like control modes, acceleration or velocity for a movement.

Sequencing: One robot movement will not be sufficient to realize a meaningful behaviour of the robot. Skills have to be sequenced after each other, the order must be easily and obviously definable by the user. Parallel execution must also be evaluated and should be supported by the framework.

Hierarchies: To support combination of skills to more complex behaviours, skill hierarchies must be supported. This means instances of skill primitives are combined to a skill and instances of skills can be combined to tasks. This allows the generation of skill library to simplify the design of processes for the user.

World Model: As soon as the robot interacts with its environment, the world is changed by the robot and the robot must be able to recognize changes in the environment. Without vision and active recognition of the world, there must be ways to inform the robot about positions and orientations of objects, to allow interaction with those.

Safety: Last but not least the safety is an important concern if the robot should be used in cooperation with the human or the other way around. Safety mechanisms have to be implemented to reduce the risks during interaction with the human, this can be reduced speeds, reduced allowed external forces or at least making the robot compliant in case of collisions.

2 Related Work

2.1 History and Origins

Before the invention of robot skill frameworks, robots were either programmed through guiding, at robot-level or at task-level[1]. Influenced by new requirements like compliant motions[2], in 1992 Hasegawa et. al presented a framework which proposed to describe tasks "as a sequence of skills"[3]. One year later, Archibald and Petriu[4] also published a paper about SKORP, which is an abbreviation for 'SKills ORiented Programming' and already presented an approach for graphical programming and sequencing of skills. This was extended in Archibalds PhD Thesis[5] which contained a generic skill template (cf. p. 34) but also considered low-level controller logic for sensors and actuators, similar to Morrows Thesis[6]. 1996 Bruyninckx and De Schutter formalized the concept of compliant motions and the task frame formalism[7]. They also defined that a "task continues until a stop condition, or termination condition, is fulfilled". Morrow and Khosla delivered a first list of compliant motion primitives in 1997 and also emphasized the importance of task composition[8]. A few years later in 2004 the task frame formalism got popular again and Kröger et al. demanded the development of a robot control architecture which is capable of executing the primitives with hybrid position / force control[9].

Other research was more focused on task planning, automatic decomposition or autonomous skill acquisition, which is not in the focus of this thesis. The work of Ekvall et al. was pioneering, they combined the concepts of graphical programming, skill modelling and programming by demonstration in 2006[10].

The combination of these topics was also the foundation for this thesis: The development of a skill framework which could be graphically programmed, interactive parametrized and easy to extend and use.

All of the previously mentioned research also built the foundation for the more comprehensive skill frameworks developed since 2010. More details about those approaches can be found in the following section.

[1] cf. Lozano-Perez (1983, p. 821)

[2] cf. Mason (1981, p. 419), Schutter and Brussel (1988, p. 3)

[3] Hasegawa, Suehiro and Takase (1992, p. 535)

[4] Archibald and Petriu (1993, p. 104)

[5] Archibald (1995)

[6] Morrow (1997)

[7] cf. Bruyninckx and De Schutter (1996)

[8] cf. Morrow and Khosla (1997)

[9] cf. Kröger, Finkemeyer and Wahl (2004), Kröger, Finkemeyer, Thomas and Wahl (2004)

[10] cf. Ekvall, Aarno and Kragic (2006)

2.2 Skill Frameworks

Robot programming is described on a general level by Haun and divided into different categories[11]. Based on these categories, the goal that should be reached with the development of a skill framework would be to support direct teach-in which is part of the online programming category. This means directly guiding the robot to goal positions or even recording trajectories. Additionally the programming would be a mixture of robot oriented and task oriented programming. At least on lower levels of the skill hierarchy, robot movements are directly defined. More generic skills would fall into the task oriented programming section, because they only need few parameters which are defined by the special task the skill should be used for.

[11] cf. Haun (2007, p. 173-180)

In 2011 Björkelund et al. released their work about skill frameworks which should increase productivity in production processes[12]. They widened the focus from looking at the robot to also regarding the other automation components that might be part of a production process. Their approach was to use *Model-Driven Engineering* to create composable components and separate configuration to allow better reuse.

[12] cf. Björkelund et al. (2011)

Bøgh et al. released two papers in 2012[13] which included models for the skill concept, an analysis of skills[14] relevant for a robot in industrial production and an overview on the implementation layers and roles. This work was continued and extended by Petersen et al. in 2013 in which they defined exemplary pick and place skills with pre- and postconditions and execution steps[15]. They also note that they implemented the task frame formalism (TFF) as a skill primitive, but it is not a real time TFF controller. Weidauer et al. presented a more focused approach for the TFF which was based on place transition nets and seemed more formalized but less applicable for industrial scenarios[16].

[13] cf. Bøgh, Nielsen, Pedersen, Krüger and Madsen (2012)

[14] see also Bøgh, Hvilshoøj, Kristiansen and Madsen (2012)

[15] cf. Pedersen, Nalpantidis, Bobick and Krüger (2013)

[16] cf. Weidauer, Kubus and Wahl (2014)

In his master thesis Zeiß 2014 also developed a skill framework which was based on motion nets (finite state machines) but dealt with the topic on a lower level with very formulized descriptions of skills, preconditions, completition and quality criterias and motion descriptions[17].

[17] cf. Zeiß (2014)

Another approach that should also be mentioned as a reference, is the work from Pfrommer et al. who also classified robot skills in a broader automation context but used a different model to integrate skills in the context of product, process and resource where skills and transformation were combined to describe an action[18]. This differs from the more common model of skill primitives, skills and tasks. Other interesting aspects of this work include the *Plug and Produce* concept and formalization via *AutomationML*.

[18] cf. Pfrommer et al. (2015)

More connected to the compliant motion and TFF focus, Butting et al.[19] released a paper in 2015 which describes in a very descriptive way how they implemented their skills based on UML/P Statechart language.

[19] cf. Butting, Rumpe, Schulze, Thomas and Wortmann (2015)

2.3 Proprietary Implementations

Obviously the interactive and simple ways for robot programming also found their way into proprietary solutions offered by some of the robot manufacturers today. Most of those approaches also rely on the skill concept which allows the user to parametrize the skills during the modelling of the production process. The following list provides a short insight of the current state that the manufacturers offer:

- Rethink Robotics - Sawyer
  With their 7 degrees of freedom (DOF) arm and an interactive touch display, *Rethink Robotics* offers a robot which is specially designed for human robot interaction. At the flange of the robot there is a cuff with various buttons that can be used for direct interaction. For programming, a specialized IDE called *Intera Studio* is provided (see figure 2.1) which allows graphical programming based on abstract icons and 3D visualization / simulation.

- Kuka - iiwa
  The *iiwa* is also a 7 DOF arm which is designed for human robot interaction. Force-Torque sensors in each joint can measure external forces to detect contacts and react accordingly. The robot can be ordered with a hand-guiding flange which offers one input button. For their programming environment *Sunrise.OS Kuka* announced a graphical programming environment. The interaction should be realized through the *smartPAD* which has various input buttons and a touch display.

- Franka - Emika
  As a quite new manufacturer on the market, *Franka* also offers a 7 DOF arm with lots of input elements at the end effector. The development environment called *Franka Desk* is based on visual programming and interactive teaching and parametrizing.

- FP Robotics - P-Rob
  The *P-Rob* is a 6 DOF arm which already looks a bit different than the other industrial robots which are made of metal or plastic. This robot arm has a housing made of leather which interferes safety for interaction directly. The

programming environment called *myP-Interface* also allows interactive development based on skills and teach-in.

- ABB - YuMi
  The dual arm robot has also been designed for interaction between robot and human. An IDE is called *RobotStudio Online YuMi* is freely available and also offers teaching possibilities for the process modelling.

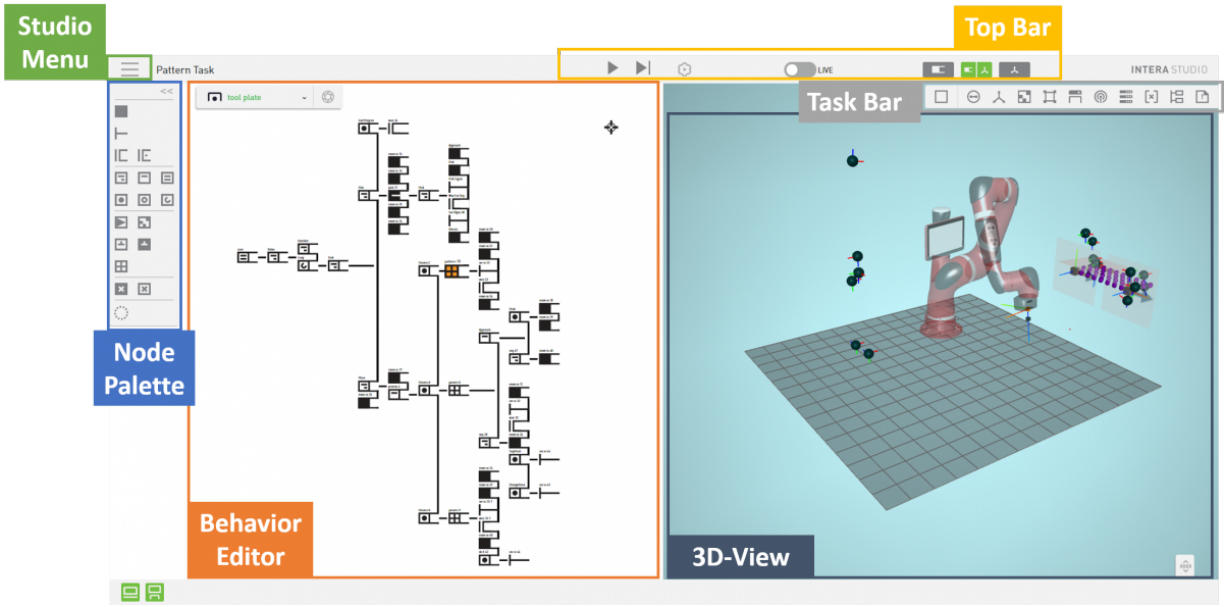


Figure 2.1: Intera Studio IDE - Graphical Programming

3 Concept

Pedersen et al. describe robot skills as high level building blocks from which a task can be composed[1]. Furthermore they formalize a skill to be object-centred rather than being connected to 3D coordinates. Following their argumentation, on the one hand this might provide a higher usability, but on the other hand it makes the skills more complex and less robust. To allow object-centred skills, a robot must be able to identify the objects in its environment which would require a vision framework and complex strategies and algorithms for gripping. In my concept, skills are position-based which allows the users to directly teach the coordinates needed for a certain task. This will require more user interaction when setting up complex processes, but it will also lead to more robustness and it also enables precise interactions.

[1] cf. Pedersen et al. (2016, p. 284)

Furthermore Pedersen et al.[1] say that a skill must be self-sustained which is explained by the following three requirements. Each skill should be:

- "parametric in its execution, so it will perform the same basic operation regardless of input parameter,
- able to estimate if the skill can be executed based on the input parameter and world state, and
- able to verify whether or not it was executed successfully."

Even though there were differences in the skill design (object/coordinates), this skill concept was used as the base concept for my implementation and is visualized in Fig. 3.1.

As an input, the skill receives a set of parameters. Which parameters are needed will be defined by the skill itself and it may contain required and optional parameters. In case of optional parameters not specified by the user, default values will be chosen to ensure a basic practicability. Some of the parameters might be defined during design time of the process, others will be taught interactively (see section 4.6) Another input would be the current state of the world, which could be for example the actual joint configuration / cartesian position of the robot tool center point (TCP) or other global parameters like the operation mode of other components in the environment.

The skill itself is subdivided into four main parts:
The preconditions check whether it is ensured that the skill may be executed (see section 4.3.2 for more information).

The execution itself will mostly be controlled by a robot controller which should abstract the control logic so that low level implementations should not be needed. This means in case of movements, the robot controller offers simple, configurable interfaces to execute those. Other examples can be tool operations which would be e.g. open gripper or close gripper.
The continuous evaluation gets important when using compliant motions, for example where a force should be tracked. Also other parameters can be tracked, like distance from start / goal position, execution time and so on. They might be used to trigger other actions or stop the execution.
Finally the postconditions check should verify whether the execution of the skill was successful and its outcome might influence the further execution of the task (see section 4.3.2 for more information).
The output of a skill is always a change in the world state. Maybe the robot has moved to a different position, did something with its attached tools or the skill sets some variables which describe the changes caused by the skill execution.

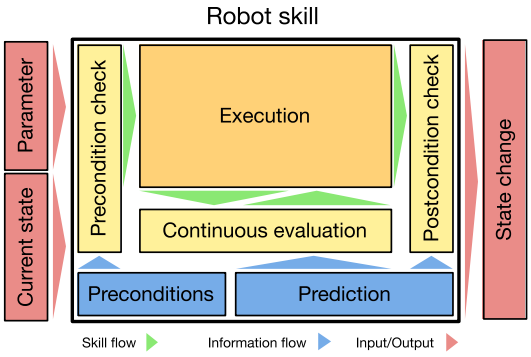


Figure 3.1: Skill Concept (Pedersen et al., 2016)

3.1 Skill Primitives

To allow composition of skills, they should be organized hierarchically and may be sequenced in the higher hierarchy layers as needed. The lowest layer should implement the skill primitives which are not composed from other primitives. They implement their full functionality and can only be parametrized to adjust their execution.

My implementation is based on the *Kuka Sunrise API* and the skill primitives encapsulate all motion primitives offered by the API. Furthermore, skill primitives have been implemented to control the attached grippers. They are controlled by simple IO port settings which are also offered through the *Kuka* API.

The skill primitives used so far are: Cartesian PTP movement, Linear PTP movement, Relative Linear PTP movement, Joint PTP movement, Cartesian PTP Batch movement, Joint PTP Batch movement, Open Gripper, Close Gripper and Teach Position. These primitives are robot-centred because they offer the different movement functionalities that the robot can execute. The developer who implements skills from those primitives needs to know how they vary in their execution and input parameters.

Representative for the other skills, the Cartesian PTP movement is presented in detail in the table 3.1:

Cartesian PTP Movement		
Preconditions	1. Robot at defined Pre-Position 2. Robot ready to execute movement	
Execution	**Required Parameters**	1. X-Coordinate 2. Y-Coordinate 3. Z-Coordinate 4. A-Orientation 5. B-Orientation 6. C-Orientation 7. Status-Parameter 8. Turn-Parameter 9. E1-Parameter 10. Motion Frame 11. Execution Mode
	Optional Parameters	1. Max. Relative Velocity 2. Max. Relative Acceleration 3. X-Force Threshold 4. Y-Force Threshold 5. Z-Force Threshold 6. X-Torque Threshold 7. Y-Torque Threshold 8. Z-Torque Threshold 9. Measure Frame
Continuous Evaluation	1. Force / Torque Monitoring	
Postconditions	1. Robot at defined Goal-Position	

Table 3.1: Cartesian PTP Movement Skill Primitive

The preconditions could be extended: If the safety should be enhanced, the robot should not only check whether it is ready to execute the movement, but maybe check as well whether the environment is ready for execution. In a pro-

[2] cf. Programmable Logic Controller

duction environment a higher control component (e.g. PLC[2]) might be requested for that.

The first nine required parameters can be consolidated to a separate data structure which defines the cartesian position and redundancy parameters to reach this position (see also section 3.7). The motion frame defines with which part of the robot the position should be reached, e.g. the flange, a TCP or even an attached workpiece. Another important parameter is the control mode: This could be either position based or impedance mode.

As already mentioned before, if the optional parameters are not specified, they will be set to a default value. If no force or torque thresholds are specified, they will not be monitored during execution. The measure frame can be stated if the force/torque measurements should be executed at a special TCP, by default the motion frame will also be used for the measurements. During execution, the force/torque values are continuously evaluated and the movement is stopped if one of the specified threshold is reached or exceeded.

At the end of the execution the current cartesian position can be compared to the goal position inside the postcondition. This can help to evaluate whether the motion was successful or whether it might have been stopped by one of the force conditions earlier than expected.

3.2 Skills

Skills are compositions of skill primitives or other skills. When composing a skill, the skill developer defines the sequence of actions that should be executed.

As an example the Pick Skill can be seen as a sequence of motions and gripper commands: The first action would be a movement (e.g. Cartesian PTP Movement) to a pre-position. The second action would be to open the gripper if not already opened. Then the third action would be a movement to the actual goal-position (e.g. Cartesian Linear Movement), where the gripper is closed (action four). And the fifth and final action would be to move the robot to a post-position which can be individually parametrized or the same as the pre-position (e.g. Cartesian Relative Linear Movement). In this example all used actions would be skill primitives.

If the skill developer combines the pick skill with an implementation of a place skill because this might often be used in combination, he simply uses both skills and creates a new skill where they are sequenced after each other. This is a good example for modular skills. They are kept small so each unit can be reused. In other scenarios, maybe something should happen between the pick and place which would not be possible if there would only be a pick-and-place skill without the modular design.

Combining Skills might allow pre-parametrization which can allow better usability because less parameters for the end user always mean a simplification and a better error tolerance.

3.3 Tasks

Tasks can use skills and skill primitives to reach a certain goal or world state. While skills and skill primitives should always stay more generic, tasks are normally more concrete and embedded in a specific context.

Thinking in a broader context, tasks might also include other components, interactions and maybe also demand more logic in the execution flow. While other approaches often only regard the robotic system in the skill execution, the interaction with other components and the user and especially the integration in an overall process is often neglected. In my concept, the task level is the place where this integration happens which offers many benefits in usability and simplifies the adaption/reconfiguration of whole processes.

The task-level programming would require the least knowledge or experience from the process designer. While skill primitive implementation will still remain the task of a robotic expert, the design of tasks should provide as much abstraction as possible and may be supported by an interactive guide. The skill composition also involves more parametrizing of skill primitives which also requires a profound understanding of robot behaviour. An interactive guide would enable the shop floor worker to adapt processes including the robot behaviour without a deep robotic knowledge.

3.4 Composition of Skills and Tasks

The composition based on BPMN[3] allows graphic modelling of the hierarchies and the borders between the different levels are quite fluent. This makes a consistent use of notation important. Skill primitives are not composable further more, they are one function block, providing different interfaces by being configurable through parameters and pre- and post conditions.

[3] Abbr. for Business Process Modelling and Notation

This means skill primitives are noted as *BMPN Tasks*. Skills are modelled in separate processes which can be saved as **.bpmn* files and may be included in other processes through the use of *BPMN Call Activities*. Tasks are higher level process models which make use of skills, skill primitives and other logic. The differentiation between skills and tasks is more conceptual than it can be closely coupled to BPMN notation. Tasks also include the integration of other components, dy-

namic changes of the user interface and further aspects (see section 3.5.3). The different layers and their notations are shown in figure 3.2.

A pick-and-place skill can for example be used in an assembly scenario where the skill is just one action which is enriched by other movements or interactions with other components. Tasks however can be sequenced for complex scenarios and processes. By this sequencing and hierarchical ordering of skill primitives, skills and composed skills, complex behaviours can be established and used in flexible ways.

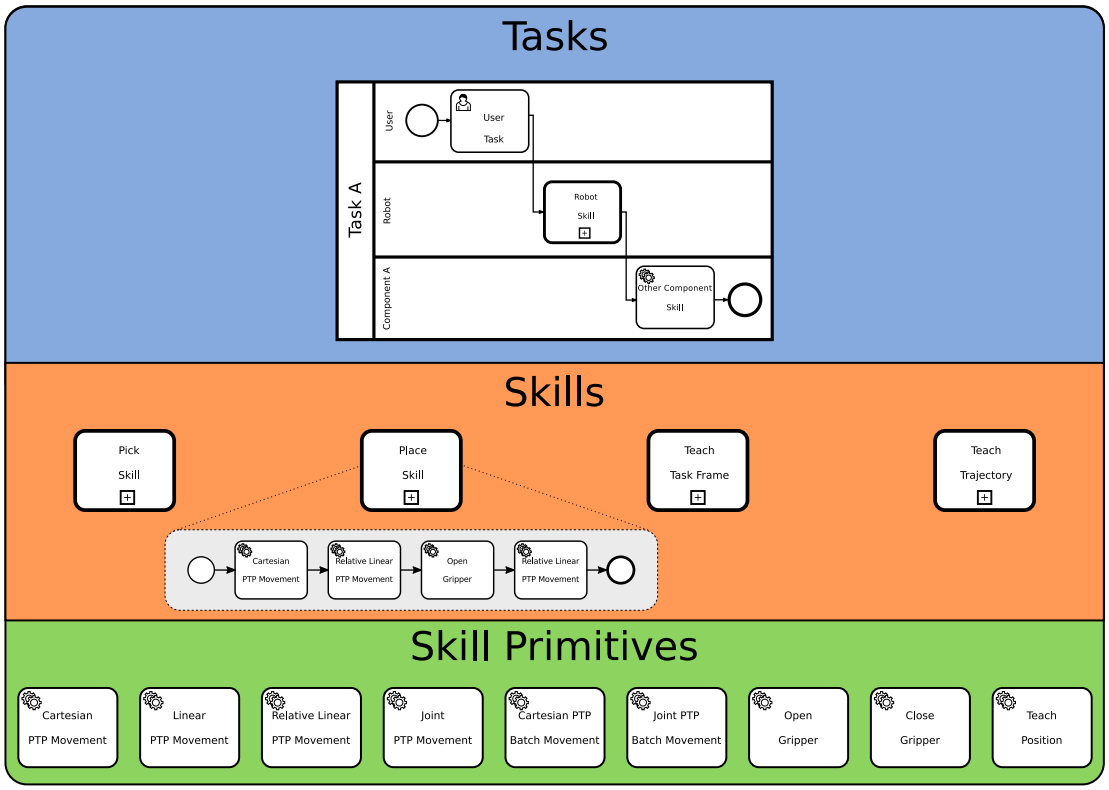


Figure 3.2: Conceptual Skill Hierarchy

3.5 Concept for Realization

To set up a skill framework different components are required: To model the skills, compose and sequence them, create a hierarchy and allow parametrization, a comprehensive framework is required. To reduce development effort on the one hand and to allow embedding the skill framework in a unified environment on the other hand, *BPMN* (see section 3.5.1) was chosen as the graphical modelling language and the *Camunda Framework* was chosen to execute the modelled plans directly (see section 3.5.2). Furthermore the *AWAre Framework* was used (see section 3.5.3), which is an extension of the *Camunda Framework*.

As a testing environment for application of the skill framework, a *Kuka iiwa* was used (see section 3.5.4) which is programmed in a *Java* environment and makes use of the *Kuka Sunrise.OS* (see section 3.5.5). This allows programming the robot on a higher level, provides abstraction and makes the implementation of the skill primitives relatively easy.

3.5.1 BPMN

BPMN is a graphical modelling language used to model and design processes which can then be automatically executed. It has a broad set of symbols which allow to model complex flows with branching of processes, hierarchical composition, combining automated and manual tasks and connecting various actions with the tasks as well as explicit error modelling and handling. The current version 2.0 was released in 2011 by the OMG[4].

[4] Abbr. for Object Management Group

The most important symbols and notations are the following:

Tasks: They can be used in various manifestations: *Service Tasks* are used to execute code that can be defined in *Java* classes or *Expressions*. *User Tasks* are used to model interaction with a user (through any kind of interface) and *Script Tasks* are used to execute inline scripts or scripts that are defined in external files (various scripting languages are supported). *Subprocesses* allow structuring code into functional units and *Call Activities* allow to include processes defined in separate files.

Events: *Events* are used to start and end processes. *Intermediate Catch Events* can also be used to wait for specific actions, *Intermediat Throw Events* can be used to trigger the sending of different events. Additionally it is also possible to attach event catching to tasks which is then called *Boundary event*. Most important *Event* types are: *Message*, *Signal*, *Cancel* and *Timer*.

Gateways: *Gateways* allow to implement logical sequence flows. *Exclusive Gateways* are used to fork a process and to join it together. *Conditions* can be used to define which *Sequence Flow* is chosen after a fork. *Parallel Gateways* can be used for modelling parallel execution, but since one execution is normally single threaded in *Camunda*, the execution is not really parallel.

Pools: *Pools* with *Lanes* are used to structure processes and order them by responsibility or belonging.

3.5.2 Camunda

The *Camunda Framework* uses the BPMN description of processes to automatically execute them using the process logic defined by the language. It allows to integrate scripts and code into various tasks and extend other tasks through code to implement process logic and extend process functionality. Processes can be executed by building standalone applications or by building and deploying web applications to a web server (e.g. *Apache Tomcat*).

The *Process Engine* is the core of the framework and responsible for deploying and executing processes. Functionalities like process querying, process modifications and execution manipulations are just some additional features. To allow tracking states and logging results, *Process Variables* can be used. They can be created and updated by *BPMN Tasks*and through *Expressions*. Each variable has a specific scope and variables can be accessed globally or locally.

3.5.3 AWAre Framework

[5]Abbr. for Assistive Workflow Architecture

The *AWAre*[5] *Framework* was developed at the University of Bielefeld to extend the *Camunda Framework* with more functionality and the ability to develop plugins for various components. This in turn makes integration of those components into BPMN processes possible. It allows a unified modelling and communication through the same interfaces.

To get access to the process engine, the *AWAre Framework* provides an interface. It allows to develop plugins which can be loaded by the process engine and may provide delegates to enrich the BPMN process with functionality.

Another feature is the graphical user interface which can be started together with the processes and can be connected to directly interact with the process, the process engine or other components. On the one hand the GUI is capable of controlling the overall process execution, on the other hand it can be used to react to events from the process or provide an interface for user interaction through *User Tasks*.

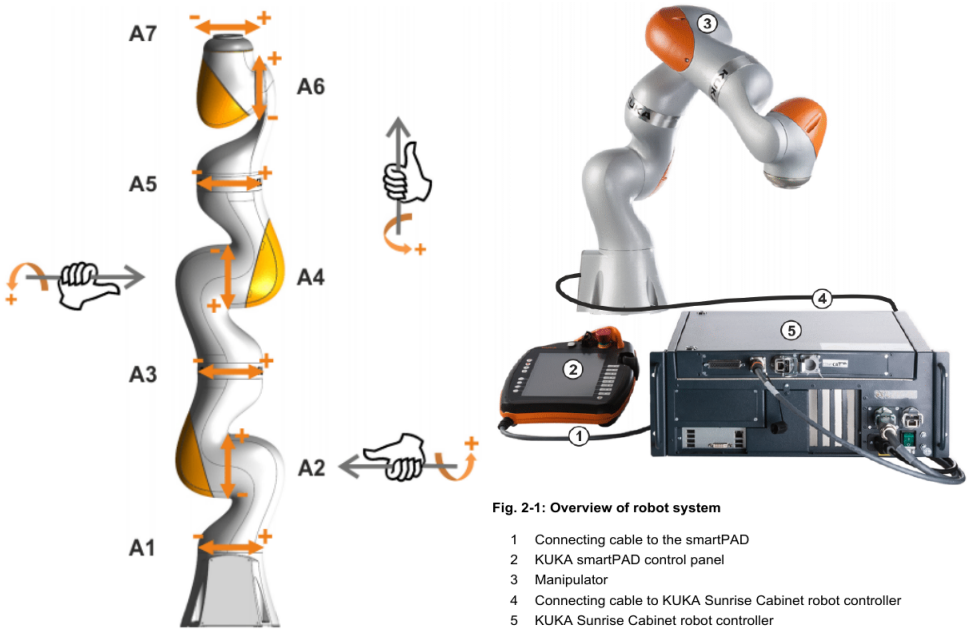
The *AWAre* plugin structure should be used to develop a robot skill plugin that provides delegates for each skill primitive. This allows parametrization at the time of designing the process or even later through parametrization processes and the graphical user interface mentioned before (see also section 3.6).

3.5.4 Kuka iiwa

The *Kuka iiwa* is a 7 DOF robot, meaning that it has seven joints that can be used to reach a position with various joint configurations (see figure 3.3a). This re-

dundancy makes it more flexible than other conventional robots with less joints. For the application tests a *Kuka iiwa 7 R800* has been used, which is capable of moving up to 7kg at its end effector and has a radius of 800mm maximum reach. One thing that makes this robot arm special and allows the interaction with it, is that force-torque sensors were integrated into every joint. This makes it possible to detect external forces at every part of the robot and it is even possible to calculate the direction from where they acted upon the robot. It can be used for user interaction which can be established through a zero gravity mode, where the robot only reacts to gravity in the first place and keeps its current position stable. Secondly the robot reacts to additional external forces and is compliant, so it can be moved wherever it is led by the user (in range of the joint limits). But the force measurements can also be used for the tasks that the robot should establish: Forces can be increased until a certain threshold is reached and then the robot can stop the movement and continue the execution without damaging parts through forces which were to high.

In figure 3.3b the hardware can be seen that belongs to the robot. Number 3 shows the robot arm itself, 5 denotes the robot controller (*Kuka Sunrise Cabinet*) which runs a *Windows CE* environment extended by a real time kernel. Number 2 shows the *SmartPAD* which can be used to control the robot, execute applications, monitor parameters and also stop the robot in case of failures. The robot controller provides a number of interfaces like Ethernet, *EtherCAT*, IO ports and furthermore.



(a) 7 joints **(b)** Hardware

Figure 3.3: The *Kuka iiwa* (KUKA, 2016)

Additionally two grippers were installed on a Y-Mount which was attached to the flange. More details can be found in section 5.1 and 5.2.

3.5.5 Kuka Sunrise.OS

[6] Abbr. for Fast Research Interface

Besides a low level interface for programming (FRI[6]), *Kuka* also offers the *Sunrise.OS* as a java framework to program the robot arm and include further logic and safety mechanisms. The *Sunrise API* is a good starting point to implement skill primitives: The different motion commands can be encapsulated as primitives and it is possible to integrate a middleware to allow communication with other software components. All information regarding the programming capabilities of the framework can be found in the official manual for system integrators[7].

[7] KUKA (2016)

3.6 Interactive Parametrization

[8] Schou, Damgaard, Bogh and Madsen (2013, p. 2)

[9] Schou, Andersen, Chrysostomou, Bøgh and Madsen (2016, p. 5)

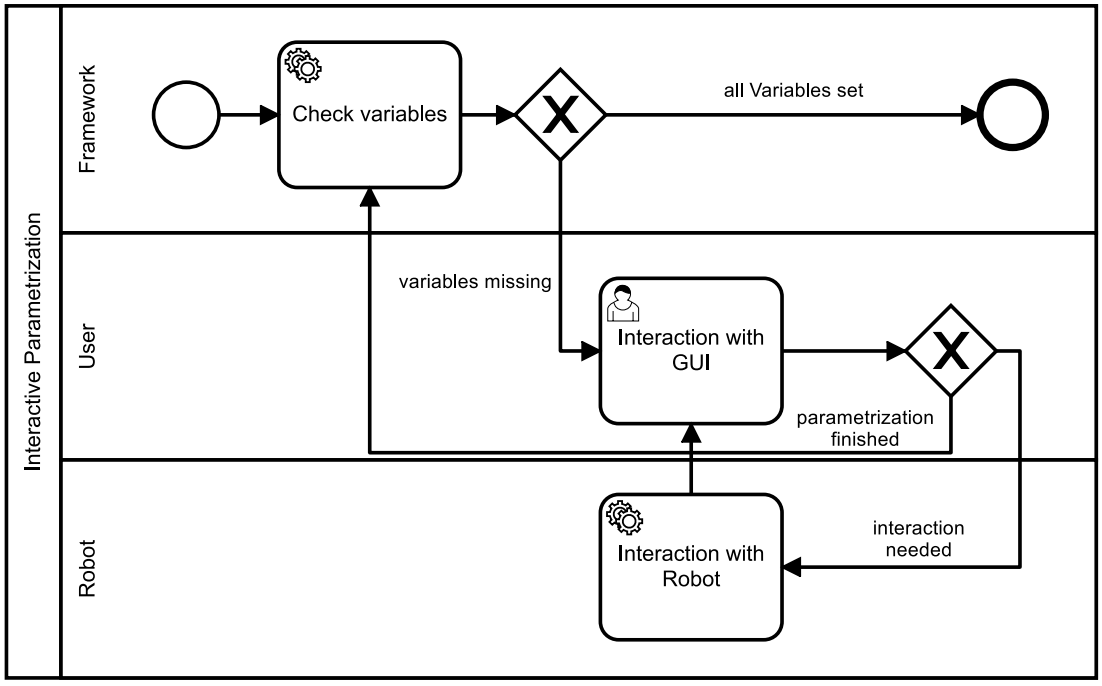
[10] cf. Ekvall et al. (2006, p. 399), Skoglund (2009, p. 2)

[11] cf. Steinmetz and Weitschat (2016, p. 282)

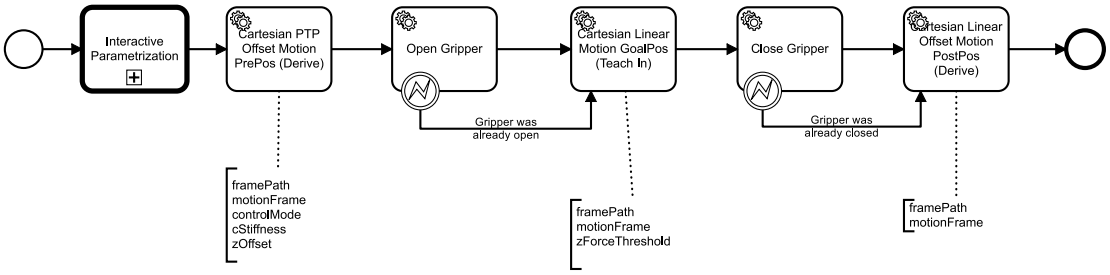
Skill Parametrization is an important aspect of a skill framework. In 2013 Schou et al.[8] wrote that if a robot "is to be programmed by an operator with limited robotics knowledge at the shop floor during production runtime, the programming interface must be easier and faster to use, than conventional robot programming interfaces." They propose that the programming has to be done at a higher level of abstraction and that it might be divided into two phases: Phase 1 is called *Specification phase* where the sequence of skills is modelled and skills are partly parametrized. This phase is also called *Offline specification phase*[9]. Phase 2 is called *Online teaching phase* and in the phase the interaction between human worker and robot is important. Locations can be trained through kinaesthetic teaching[13], sometimes also called *Programming by Demonstration (PbD)*[10](see also figure A.7). 2016 Steinmetz and Weitschat presented an approach to integrate the parametrization process explicitly in a skill[11]. In their example of a screw skill, a check whether the screw pose has already been taught is always executed and if this is not the case, the process automatically forks into a branch where the pose is taught interactively and confirmed by the user.

Embedded in the BPMN concept, parametrization can be done through *Input/Output Variables* and *Field Injection* on *BPMN Tasks*. Through this, it is possible to implement the phase 1 as mentioned above: Parametrization during specification. Depending on the parameters, some can be taught interactively, e.g. positions and trajectories. This can be modelled explicitly in the skill as shown in figure 3.4b. The idea is based on Steinmetz's approach but my concept tries to generalize the parametrization so it can easily be included in every skill. The concept shown in figure 3.4a is simple, the framework checks whether all variables are

already configured. Therefore it is able to request the database, process variables and maybe even use communication with other components. If parameters are missing, the user will be asked for help with the graphical user interface of the *AWAre* framework. Maybe some skills can be parametrized through GUI elements directly, for others direct interaction with the robot might be needed, which will then be triggered directly and can return its results to the process. If the user decides that the parametrization is finished (supported by the assistant interface), a second check will occur and normally the interactive parametrization process will be finished and the skill might be executed.



(a) Concept Interactive Parametrization



(b) Interactive Parametrized Skill

Figure 3.4: Interactive Parametrization

3.7 Task Frames

The concept of task frames was first introduced by De Schutter and Van Brussel in 1988[12] and formally defined by Bruyninckx and De Schutter in 1996[13] to describe tasks with compliant motions. In 2004 Kröger et al. presented an approach for a notation and representation of task frames and their relations[14]. Another paper was published by Blumenthal et al. in 2013 which deals with similar topics but has a broader focus and introduces the concept of a scene graph[15].

In the skill framework positions have to be stored somehow to allow different robot programs and other components to access them. This is done in the world model which is realized as a graph database (see section 4.5.2). Inside the database different objects can be stored as nodes and relations between them can be added and removed as necessary. This allows a dynamic capturing of the world and follows roughly the requirements as mentioned by Blumenthal et al[15].

[12] cf. Schutter and Brussel (1988, p. 4)

[13] cf. Bruyninckx and De Schutter (1996, p. 581)

[14] cf. Kröger, Finkemeyer and Wahl (2004, p. 5219) and Kröger, Finkemeyer, Thomas and Wahl (2004, p. 2)

[15] cf. Blumenthal, Bruyninckx, Nowak and Prassler (2013, p. 453)

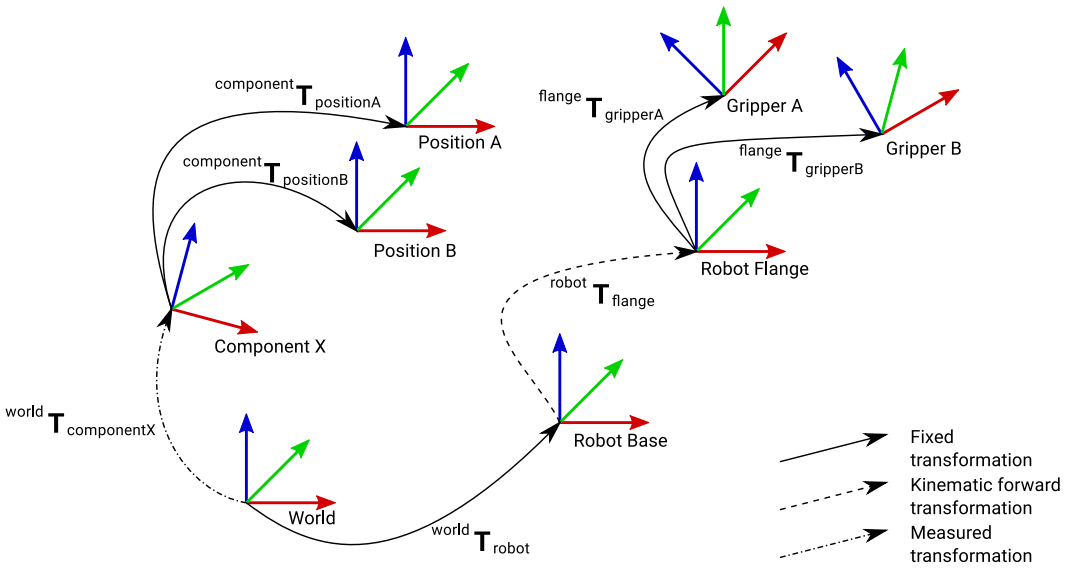


Figure 3.5: Task Frame Hierarchy and Transformations

Figure 3.5 shows some exemplary task frames and their relations. Base of all coordinate systems is the *World* task frame. It defines the orientation of all world coordinates. Through a transformation from world to robot the *Robot Base* is defined. A transformation consists of a translation and rotation. The kinematic forward transformation is able to determine the position of the robot flange at any time. Relative to the flange further task frames like a gripper can be defined through transformations. An ongoing chain can describe the root of the gripper, then the tip of the gripper and maybe even an inserted workpiece.

Other components are also placed in the robot environment and normally their position is related to the world as well. The transformation may be determined through manual measurement or measurements with the robot. Sometimes it may also just be virtual for simulation purposes. Each component can also deliver its own task frames which are normally fixed to the base frame of the component. As an example some task frames are visualized in the simulation shown in figure 3.6. In the upper left corner of the production cell the *WorldRoot* task frame is defined. The robot *RootFrame* is located at its base and the other components show some exemplary positions called *M1* and the according pre-positions for robot movements called *PreM1*. As well the frame *FlangeFrame* is shown between the mounted tool and the robot flange. The frames *Gimatic_Tip* and *Afag_Tip* are placed at the center points of the two grippers.

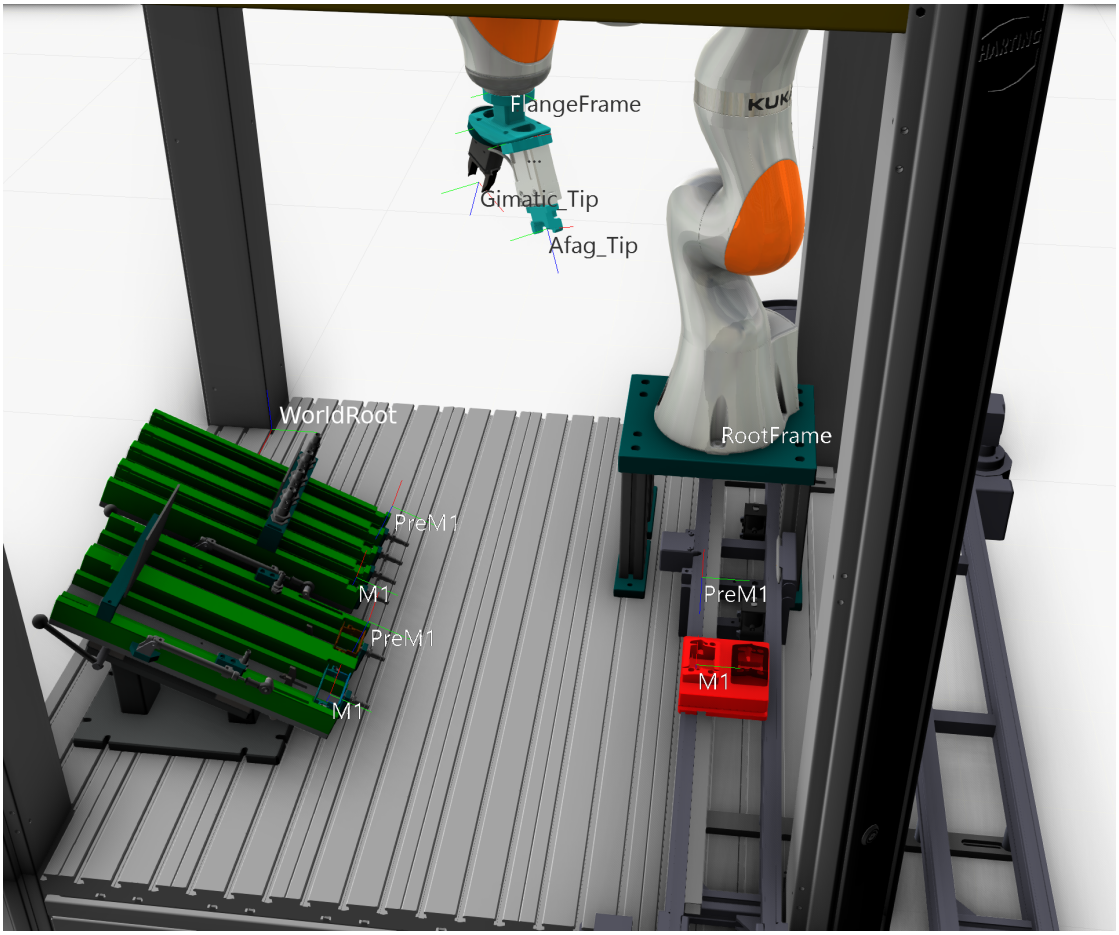


Figure 3.6: *HARTING* Production Cell

4 Implementation

To implement the skill concept described in chapter 3, a distributed architecture was chosen. Existing technologies and frameworks are extended and integrated to set up a framework which is modular and scalable.

4.1 Framework Architecture

The description of the framework architecture should give an overview on the particular components and afterwards the skill implementation will be explained in more detail in section 4.2 and 4.3.

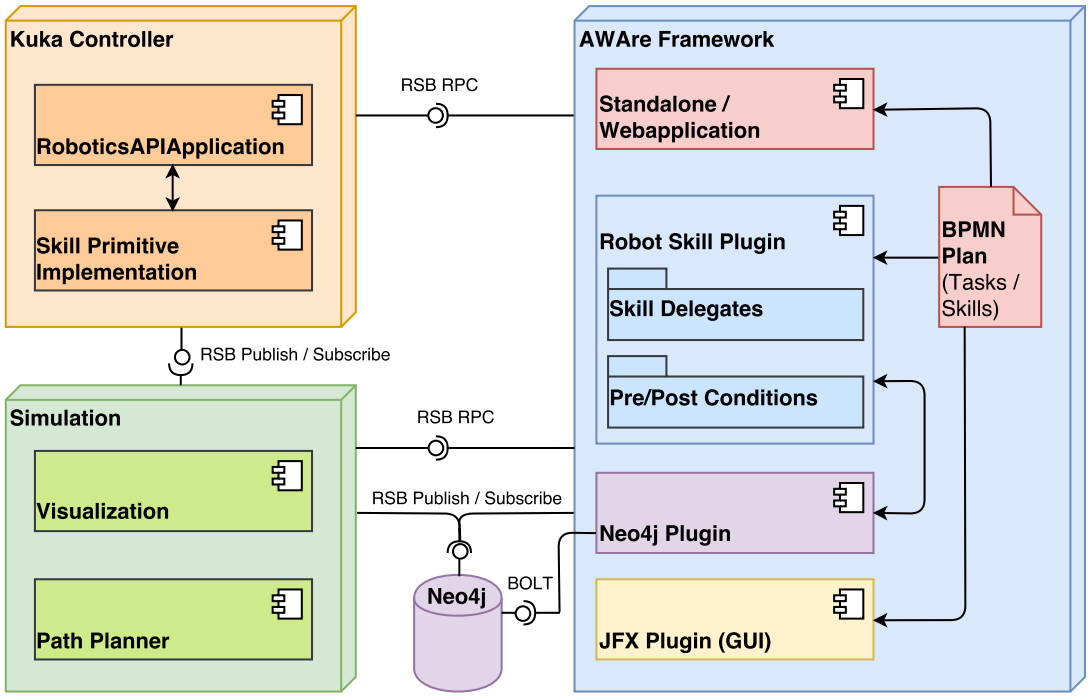


Figure 4.1: Software Architecture

The software architecture mainly consists of two separate frameworks which communicate through a middleware (see Fig. 4.1). On the left side the specific robot implementation is shown (orange box). The *Kuka Sunrise API* de-

termines that there can only be one *RoboticsAPIApplication* running at the same time. The skill primitives are implemented in own classes and their execution is triggered by the application which handles the communication with the other components. On the right side the *AWAre* framework is drawn (blue box) which is an extension of the *Camunda Framework*, a BPMN process engine (see also section 3.5.3). The *Robot Skill Plugin* contains the delegates which are used to parametrize the skill primitives and then communicate with the *Kuka Controller* to trigger the skill execution. Pre- and Postconditions are also defined within the plugin and can also be reused for various skills (through parametrization). *BPMN Service Tasks* then connect the skill delegates and pre- and postconditions and allow the usage inside the BPMN world. Complex Skills and Tasks can be modelled by sequencing *BPMN Tasks* and hierarchically composing them. The execution logic of *Camunda* guarantees a sequential execution of precondition, execution and postcondition and of course of *BPMN Tasks* as well.

Additionally, a database called *Neo4J* is used to store the positions and trajectories which are used for the robot skills. The database is based on a graph structure which means that entities may have attributes and are connected to other entities through relationships. This allows a hierarchical storing of the positions which will be explained more specific in section 4.5.

Another component in the architecture is the simulation (green box) which is based on *Visual components*, a 3D simulation framework. The simulation offers functionalities for collision avoidance and path planning which are crucially important as soon as the environment for the robot gets more complex. It even allows a better usability because the user does not have to teach every trajectory for the process any more, as they can automatically be calculated by the simulation. Although it is part of the framework and has been integrated, it was not focus of this thesis and in this context is regarded as an external component.

The detailed software architecture is shown in figure 4.2. As in the other figure (4.1), the left side shows the robot specific part and the right side shows the *AWAre* part. More detailed figures with all methods and fields can be found in the appendix (C.1 and C.2).

For the robot specific implementation, the *SkillProvider* class is the center which invokes the execution and initializes the communication infrastructure. Therefore it has instances of all skill callbacks which are registered for remote procedure calls at the startup of the application. It also has an instance of the *DoubleGripper* class which gives access to a number of defined task frames below the flange and also defines the load data of the gripper construction which is needed for controller calculations of the *Kuka API*. The Callbacks themselves extend the *DataCallback* class provided by the *RSB* middleware. They have instances of the skills they function as a callback for. For the parametrization process some utility classes have been implemented (*MotionFrameMaster, SkillCon-*

figurator, ControlModeFactory). The skills extend from the *SkillPrimitive* class which centralizes the parameters all skills have in common and defines some abstract methods the specific skills have to implement.

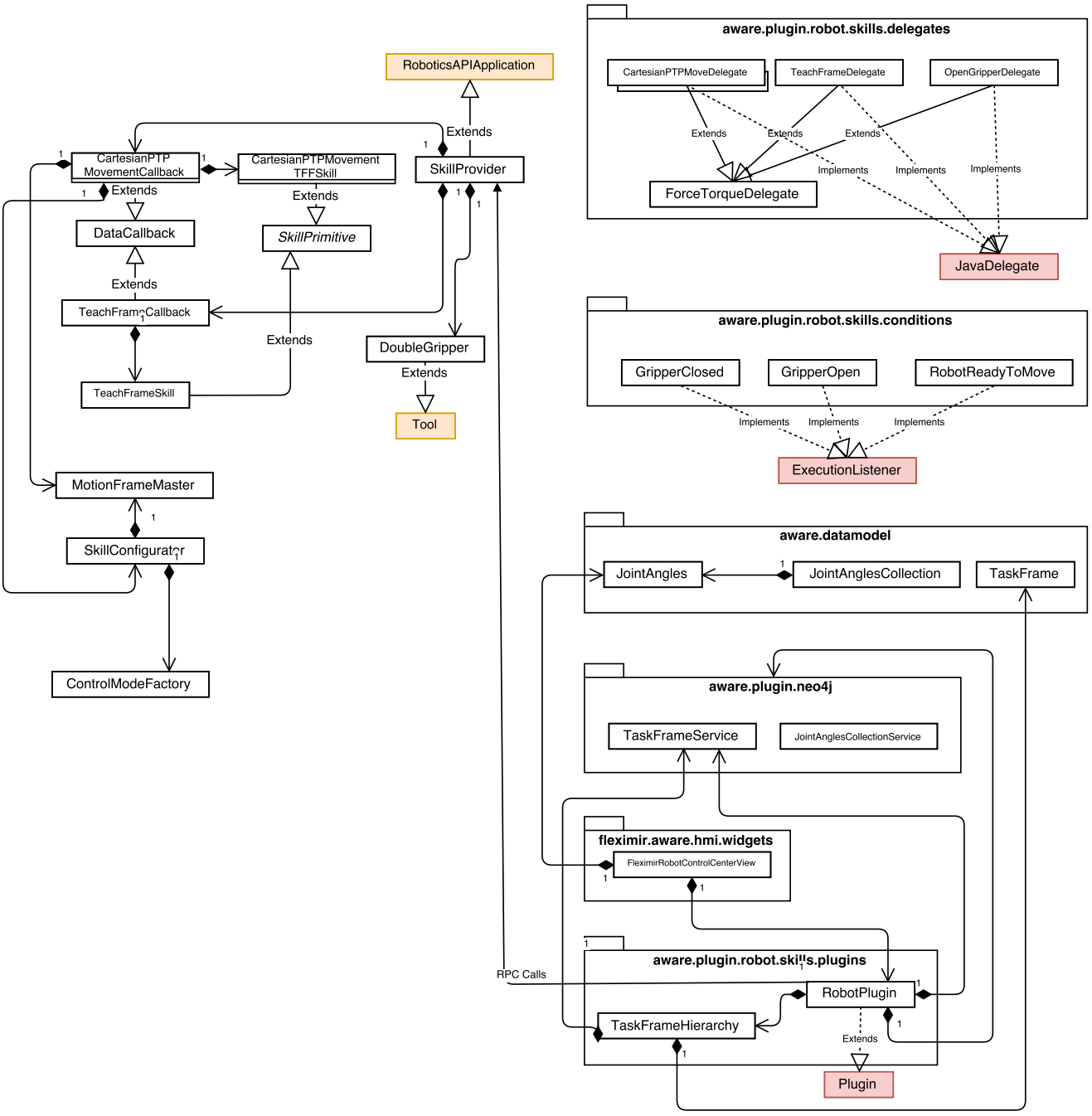


Figure 4.2: UML Class Diagram

The results of the *AWAre* development are shown at the right side of the figure. The distributed software structure is clearly visible through the packet

structure and the packets also belong to different software packages which are all developed independently. The *aware.datamodel* package provides a centrally defined structure for the data types that can be used throughout the various parts of the framework and especially for persistent storage in a database. The *aware.plugin.neo4j* is an *AWAre* plugin which provides a simplified database interface. Storage, retrieval and updating of the data types in the specific *Neo4J* database is possible through this. The *fleximir.aware.hmi.widgets* package contains a controller class for a *JavaFX* view which can be used for teaching and parametrization of the robot. It is application specific and therefore not part of the *AWAre framework* itself but it is based on it and uses the interfaces and GUI capabilities.

The main part which is directly implementing the skill framework, is embedded in the subpackages underneath the *aware.plugin.robot.skills* package. The plugin package itself contains the *AWAre* specifc realization of the plugin capabilities and the *TaskFrameHierarchy* which simplifies the usage of relative task frames. The delegates which allow the integration of skill primitives into the *BPMN* world are organized in the same-named package and all extend from the *ForceTorqueDelegate* to centralize parameters. They also implement the *JavaDelegate* interface which is provided by the *Camunda Framework*. The conditions by contrast implement the *ExecutionListener* interface which is used in *Camunda* to attach listeners for start and end events of a task.

4.2 Skill primitives - Kuka specific

The before mentioned separation of the skill implementation into a robot specific part and the higher level part, complies with the three layer model presented by Pedersen et al.[1] They propose to have a hardware abstraction between skill primitives and skills which is realized through the independent robot implementation that is used.

[1] cf. Pedersen et al. (2013, p. 2)

4.2.1 SkillProvider - RoboticsAPIApplication

The *Kuka Sunrise* Framework only allows one application to be active and loaded at the same time. Background tasks can be implemented as well, but they do not have access to the robot movement commands and though they should only be used for logging, monitoring and other background tasks. Every *RoboticsAPIApplication* should implement three methods: *initialize()*, *run()* and *dispose()*.

Since all the skills are implemented in distinct classes and their parametrization and execution is triggered by callbacks, the *initialize()* method is used to set up the whole middleware communication. A remote procedure call (RPC) server

is initialized and methods with callbacks for every skill are registered. Additionally some informers are instantiated which publish current robot sensor values (like joint angle configuration, forces, etc.) on specific scopes, following the publish-subscribe pattern. The whole middleware is based on RSB[2], which is developed and maintained by the CoR-Lab institute from the University of Bielefeld. Furthermore the gripper is attached to the robot to be considered in the robot controller and some input output mappings are initialized which allow controlling of *Beckhoff IO modules* over *EtherCAT* through the *Kuka Sunrise* framework.

[2] Abbr. for Robotics Service Bus

The *run()* method is kept quite short and only triggers the publishing of the sensor data and keeps the communication channels open, to receive remote procedure calls from the outside.

In the *dispose()* method all communication end points are deactivated and the application is shut down. To ensure the functionality of other programs, the ESM state is reset as well. More information on ESM states and safety can be found in section 5.5.

To have access to the *Kuka Sunrise* framework classes, dependency injection is used. This way the class instances do not have to be passed to each class through the constructors or getter and setter methods. This is also shown in lines 20-29 of the example code in the appendix section C.2.

4.2.2 Skill Callbacks

As shown in the example code in the appendix section C.2, callbacks for each skill primitive have been implemented. Their *invoke()* method is called when a RPC is received and the payload is delivered as a parameter. The payload contains the parameter set for the skill primitive and is encapsulated into a RST[3] data type.

[3] Abbr. for Robotics Service Types

Since the skill instance is also injected (see l. 24-25 in C.2) at first, a method is called (l. 39) which resets all parameters of the skill to ensure that no parameters from an old execution remain set. Afterwards the control mode is set for the skill (l. 41) and the force conditions are added (l. 44). Next in case of the cartesian movement the defined goal position for the skill is retrieved from the payload, set and all the other necessary parameters accordingly.

At the end of the *invoke()* method, the execution is triggered (l. 62) and its result is obtained and returned by the RPC. This is similar for all skill callbacks: The skill specific payload is analysed and the values are set for the skill instance. Afterwards the skill execution is triggered and its result is returned.

4.2.3 Skill Implementation

All skills extend from the abstract *SkillPrimitive* class. This class defines the abstract methods *clear()* and *prepareMotion()* which have to be implemented for each concrete skill. The class also defines a set of parameters which is similar for all skills like force/torque thresholds which may be used for each movement to make it compliant. The *execute()* method is implemented in the *SkillPrimitive* class directly and calls the *prepareMotion()* method at first, to configure the motion that should be executed by the concrete instance. Afterwards the motion itself is executed on the robot in a synchronous manner and the result is returned. Generic error handling methods and break conditions for movements are also defined in the *SkillProvider* class.

The specific skill implementations are held rather simple. Basically, they define additional fields which are necessary to parametrize the skill and provide getter and setter methods to access those fields. Besides and as mentioned before they have to implement the *clear()* method which just resets the fields and the *prepareMotion()* method. This methods takes all parameters from the fields of the class and configures the desired motion with them. Therefore the *Kuka API* specific motions are instantiated and setter methods are called for configuration. If force thresholds have been configured for the movement, additionally break conditions are added to the motion and the ESM state is set accordingly.

4.3 Skill primitives - Generic implementation (AWAre Plugin)

The idea of the skill framework is that the user does not have to deal with code any more but is able to specify the whole robot behaviour graphically through sequencing the skills in a BPMN diagram. This can be realized through sequencing *BPMN Service Tasks* which have a skill specific delegate attached and which define the parameters for the skill through field injection (also a *Camunda* BPMN feature).

4.3.1 Skill Delegate

Service Tasks can have *Java* classes attached which execute code and can read and write variables from / to BPMN processes. In listing C.3 an exemplary delegate is shown which is used to trigger a cartesian PTP movement. The delegates also extend from a generic class (*ForceTorqueDelegate*) that implements methods for encapsulation of force/torque thresholds. They also implement the interface *JavaDelegate* which is part of the *Camunda* Framework. The *Expression* variables

(l. 23) define fields which can be used for the field injection and though allow parametrization of the skills from the outside. So at first all the parameters from the local variables are encapsulated into the RST datatype (l. 42-118). In case of the cartesian PTP movement two possible ways of parametrizing the goal-position have been realized: Either a *framePath* can be provided, which has to be a unique identifier for the task frame stored in the database (l. 54-70). Or the other option is to specify the goal-position manually by passing transformation, rotation and redundancy parameters to the delegate (l. 74-87). When the RST datatype is correctly filled and configured, a remote procedure call is made to trigger the movement on the robot (l. 124). Finally the variables are reset (l. 129-130) since *Camunda* instantiates the delegates only once and not for every service task individually.

4.3.2 Pre- and Postconditions

Since some conditions might be used as pre- and as postconditions, they were defined in a general conditions package inside the robot skills plugin. An example for such a condition can be found in section C.4. The condition classes all implement the *ExecutionListener* interface which is also provided by *Camunda*. This also allows to access process variables, to use field injection for parametrization and it makes it possible to throw BPMN errors which can then be explicitly modelled and handled by the BPMN logic. The provided example (see appendix section C.4) shows the checking of a gripper state. Theoretically this could be used to check whether a gripper is already closed and therefore avoid to trigger closing it again or it can also be used to check the outcome of a close gripper skill as an postcondition.

Preconditions should therefore be used to check whether it is possible, safe and/or necessary to execute a certain skill. Postconditions can be used to check whether the skill execution led to the intended outcome or if it might have failed.

Camunda allows to add *ExecutionListeners* to each Service Task. During process design the user can define whether the listener has to be executed at the start of the task (precondition) or at the end (postcondition). Since *Camunda* guarantees that the listeners are called sequentially after each other, this allows to use the conditions and skill delegates as explained before.

4.3.3 Robot Skill Plugin

The delegates for the skill primitives and the conditions are all packaged within the robot skill plugin. As described in section 3.5.3 the *AWAre Framework* allows

to implement plugins which extend the *Plugin* class and are annotated with the *@AwarePlugin* Annotation. The plugins have to override the *initialize()*, the *activate()* and the *deactivate()* method. In case of robot skill plugin, the RSB communication is set up in the *initialize()* and *activate()* methods and torn down in the *deactivate()* method.

Furthermore the plugins can be used to implement functionality that is required at various other parts in the code. The goal positions for the movements are stored persistently in a database but during runtime they are loaded into a *SceneGraph* which is provided by the *Kuka API*. This allows to handle relative transformations between positions in a comfortable way (see also section 3.7). Such functionality is implemented within the plugin so it can be accessed from the skill delegates and other parts of the framework.
As indicated in the paragraph above, the RSB communication for the skill delegates is also centralized here. Computational overhead for instantiation of communication endpoints is avoided this way.

The plugins can always be retrieved through the *PluginLoader*, a singleton class within the *AWAre Framework* which manages all plugin instances during runtime. This makes them accessible for other plugins, delegates and the graphical user interface.

4.4 Skill and Task Composition

In the appendix in figure B.1 and B.2 skill compositions are shown which solely are based on skill primitives. The notation is BPMN and the annotations at the tasks show the most important parameters. The error handling is also shown exemplary at the “Open Gripper” and “Close Gripper” tasks in both figures. The flash symbols represent the catching of a thrown BPMN error within that task and the attached sequence flow defines where to continue if the error happened. The gear wheel symbols in the upper left corner define that the tasks are *Service Tasks*. *Service Tasks* are used in BPMN to execute custom code within a process. In case of the skill framework, this code invokes robot movement or it triggers actions of the attached tools of the robot.

The figures B.3 and B.4 in the appendix show more complex processes which use skills to realize a desired outcome. The process shown in figure B.4 is encapsulated in the second box of figure B.3. This is an example to show hierarchical composition of tasks to model complex processes. In figure B.4 the bold outlined boxes represent robot skills, the second box encapsulates the skill shown in figure B.1 and the right bold outlined box encapsulates the skill shown in detail in figure B.1.

Because *Camunda* also allows scripting for parameter definitions, the parameters can be created dynamically considering process variables or loop counters. Generally variables can always be passed from one process to another, so variables which are available in a top level task can be passed down to a low level skill primitive. For results, the other way around is also possible.

4.5 Task Frames

There are two main requirements regarding the task frames:
1. The task frames should be stored persistently. They must be accessible for different processes and parts of the framework and most important, they should not be lost, if the application is restarted.
2. They should be stored in a relative manner, related to a world coordinate reference system.

Unfortunately the *Kuka Sunrise API* does not give free access to the frame hierarchy that can be used from *RoboticsAPIApplications* and updated through the *Sunrise Workbench* or interactively through teaching with help of the *Kuka SmartPAD*. Updating positions from an application or adding and removing task frames programmatically is currently not supported. To work around this issue, a combination of persistent storage and relative transformation retrieval has been implemented. The *Kuka API* at least gives access to a *SceneGraphObject* class which automatically calculates the transformations given the relative dependencies (see section 4.5.1). The same relations of task frames can also be stored in the *Neo4J Database* which is perfectly suited for this through its underlying graph structure (see section 4.5.2).

4.5.1 Kuka Scene Graph

The storage inside the database only helps to guarantee a persistent storage. During runtime, the task frames are loaded from the database into a local scene graph which is provided by the *Kuka Sunrise API*. This has the nice benefit that the relative transformations are automatically resolved and complete chains of transformations do not have to be calculated manually. It is even possible to store task frames relative to the world if the world has been measured as a base frame before.

The Scene Graph is instantiated, filled and maintained through the robot skill plugin. All methods for interaction have been implemented inside the *TaskFrameHierarchy* class. Supported actions are similar to the *CRUD*[4] schema. The imple-

[4] Abbr. for Create, Read, Update and Delete

mentation always keeps the temporary scene graph and the database synchronous and consistent. This way it is ensured that changes are always updated in the database as well and information can be read from both sources similarly.

4.5.2 Neo4J - Persistent Storage

As a part of the *AWAre* framework, a datamodel has been implemented which defines datatypes centrally so different components can work with them in the same way. Instances of those datamodel classes can be stored and retrieved from the database.

The instances of the datamodel *TaskFrame* class are the entities of the graph and they are connected through directed relationships. Except of the root node which corresponds to the world task frame, every task frame should have an incoming relationship called “PARENT_OF” that comes from the parent and is directed towards the child. Since the root does not have a parent itself, it does not have any relationships like that. Similarly, except of the root node, every entity should have an outgoing relationship called “CHILD_OF”. Those relationships allow to traverse the graph in both directions, depending on the action that should be realized.

Every entity can also be retrieved by an unique identifier. For the task frames this is the path concatenating all task frame names separated by a "/", beginning at the root frame. For example: "/Root/ComponentX/PositionA".

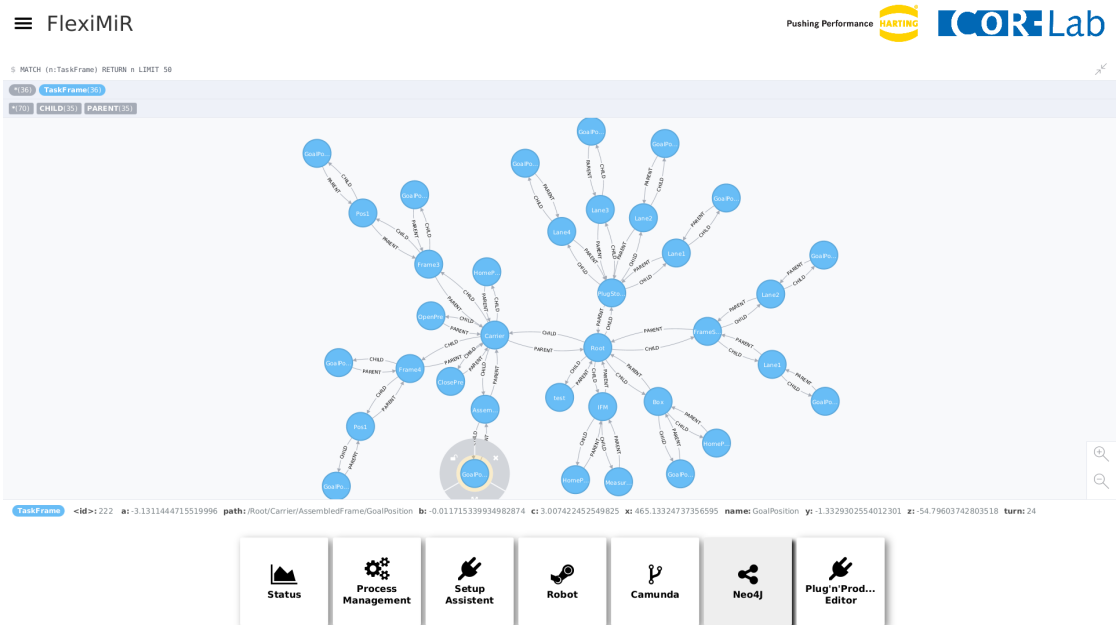


Figure 4.3: Task Frame Hierarchy in *Neo4j* View of *AWAre GUI*

Since the robot positions are changed dynamically, they are not stored inside the database. The current position of one of the mounted tools, or the flange can always be requested through a RPC from the robot directly.

Additionally the database is also used as a storage for trajectories. Comparable to the task frames, classes were also added to the *AWAre* datamodel to model the fields that are necessary for that purpose. Since the trajectories are stored as joint configurations, each trajectory element though consists of the seven joint values and the collection which forms the trajectory has an unique identifier to make it obtainable from the outside (see also figure 4.4).

4.6 Interactive Parametrization

In a first approach the implementation was realized according to the two phase concept mentioned before. A robot program was modelled by creating a BPMN process in a graphical editor, the *Camunda Modeler* (see figure A.1 and A.2). The skills can be partly parametrized inside the editor. For example velocities, movement modes and others can be added in the *Field Injection* tab of the editor when the BPMN task is currently selected which represents the skill. For each movement skill which requires a goal-position, also the unique identifier has to provided. This is used for a database lookup of the task frame for the goal position.

4.6.1 Teach Skill

In listing C.5 the *TeachFrameSkill* class is shown that allows the hand guiding of the robot. It implements the same methods as every other *SkillPrimitive* even though there is nothing to be cleared (l. 16) or prepared (l. 19) every time. The *execute()* method is also *overwritten* which is not done for the other skills. Normally the hand guiding should be possible by only calling the command seen in line 39. But because we do not use the official hand guiding flange from *Kuka*, a little more implementation was necessary.

The first thing to notice is the setting in the *StationConfig* of each *Kuka Sunrise Application*. Two options exist called “Hand guiding allowed in test mode”(1) and “Hand guiding allowed in automatic mode”(2). Option 1 defines the behaviour if execution mode T1 or T2 is chosen on the *Kuka SmartPAD*. Option 2 is relevant for all applications that should be executed in AUT mode. This option has to be configured to the value “false” which may at first be a little irritating. This has the effect that it does not start the internal hand guiding mode which can always be started if the robot is in an idle state and the hand guide switch is ac-

tivated. Instead it only allows the hand guiding to be activated, if the command *lbr.move(handGuiding())* is called by a program.

Since the official solution is not used to activate the hand guiding, the following steps were necessary to integrate the interactivity into the skill framework:
1. The *ESM state* is set to 2. This enables the activation of the handguide through the switch in the safety configuration (l. 27, see also section 5.5).
2. A *while-loop* is started which is only left if the hand guide switch has at least been activated once (l. 30-36).
3. As soon as the user presses one of the buttons at the robot flange which approve the hand guide, the code after the *while-loop* is executed and the execution is resumed (l.38). This is necessary through the unofficial implementation.
4. The hand guiding is officially started when the motion in line 39 is executed which sets the robot to a gravity compensation mode. The robot reacts to all external forces compliantly which makes it possible to move the robot to the desired location and also change its joint configuration as needed.
5. Once the user releases the button, the interaction is automatically finished and the robot gets stiff again. No further movements are possible, if the skill is not restarted.
6. To allow normal movements again which have special force limits for human robot interaction defined, the *ESM state* is set to 1 again (l. 41).

The teaching skill can be repeated as often as needed directly. Currently after the teaching, the *RoboticsAPIApplication* has to be resumed with the *Kuka SmartPAD*. The command from line 38 could also do this automatically, but this would pose a possible thread since the user might not be fully aware, that a motion might be triggered directly afterwards. This might be the case if the teaching is fully integrated into a process and maybe only some positions have to be taught and other movements can be executed directly because they are already fully parametrized.

4.6.2 Teaching Task Frames

As soon as the process is started, which includes skills, a graphical user interface is started as well which is directly connected to the BPMN process engine and is part of the *AWAre Framework* (see figure A.3). To teach the positions that are needed for the skills, the user has to select a special tab which allows interaction with the robot. To teach a task frame, the user first has to click the button "Activate Handguiding". This triggers a remote call to the robot which then activates the handguide skill. Afterwards the user is able to interact with the *Kuka iiwa* by pressing one of the buttons in the flange (see figure A.6). This can be repeated as often as needed until the user is satisfied with the pose of the robot. Then the user has to select the task frame for which he wants to store the current

position from the list on the left (see A.3). If the task frame is selected, the user can press the "Save frame to database" button. The values are automatically updated in the database, the scene graph (see section 4.5.1 and 4.5.2) and the GUI. By pressing the floating button in the lower right corner of the list view a new task frame can be created on the same level in the hierarchy or by pressing the "+" button next to a task frame in the list, a TF is created as a child of the frame directly. A name for the new task frame always has to be specified in a pop up window which is automatically opened.

4.6.3 Teaching Trajectories

To teach a trajectory the same view of the *AWAre GUI* can be used, the necessary control elements can be found in the column on the right. Similar to the teaching of a task frame, the handguide always has to be activated through the click of the related button. If the position is satisfying, the button "Teach Joint Configuration" can be used to store the joint configuration in a temporary ordered list. The storing of joint positions instead of task frames for a trajectory is caused by the redundancy of the 7 DOF robot. Since the robot can reach the same position with different joint configurations, it would not be sufficient to store the cartesian positions of the flange or end effectors. If the user has taught all intermediate steps of the trajectory (including start and goal position), he can enter a unique identifier in the related text field and press the button to finish teaching and store the trajectory afterwards.

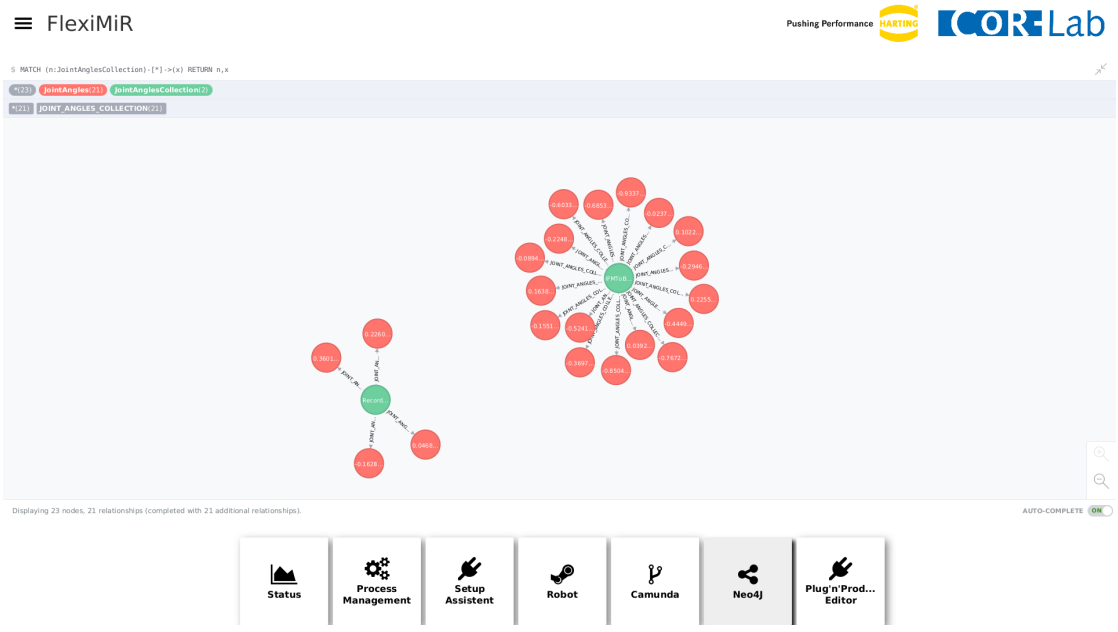


Figure 4.4: Joint Angle Collections in *Neo4j* View of *AWAre GUI*

This procedure allows the user to execute a movement along the taught trajectory afterwards, which might be helpful if a direct PTP movement would cause a collision with another component. Therefore simply the skill *JointPTPBatchMovement* has to be added to the process and parametrized with the assigned unique identifier. The number of intermediate trajectory points is only limited through the maximum number of PTP movements that can be added to *Motion Batch* which is part of the *Kuka API*. Currently the limit is 500 (*Kuka Sunrise Version 1.11*[5]).

[5](KUKA, 2016)

5 Application

The skill framework has been applied to an assembly process which was chosen as a reference because currently the assembly is done manually by hand. The process is characterized by a high variance in the product itself (can be assembled with different specifications) and the process has the potential for dynamic extensions and adaptation. These requirements make the process quite suitable for the skill framework because the focus is not only on efficiency and speed but rather on flexibility of the process and simplified reconfiguration.

5.1 Production Environment and Components

HARTING provided a production cell (see figure 5.1) which was constructed to allow modularized production processes and the combination of different cells with varying functionality. The cell itself provides the skeleton to install other components and to connect them with each other. The following list gives an overview of the components that are installed and what function they contribute to the overall production system:

Conveyor belt: Transports the carriers along the processing stations.

Carrier: Holds the product to allow processing in a fixed posture. Is moved along the conveyor belt. Each carrier has a RFID tag installed which may carry information about the product or at least allows identifying it precisely.

Intelligent Stop stations: Are installed at the conveyor belt to stop the carriers for various purposes. Based on a Raspberry PI they implement an own intelligence which allows automatic collaboration of all connected stop stations.

Frame storage: A specially constructed storage component which maintains supplies of Han-hinged frames. Pneumatic appliances separate the frames so they can be picked by the robot.

Plug storage: A specially constructed storage component which maintains supplies of Han plugs.

IFM component: A visual inspection component that allows quality checks based on image processing and checking for defined features.

RFID Reader: HARTING RFID reader to identify the RFID Tag that is installed on each carrier.

HARTING Mica: Small industry suitable computer that allows installation of linux containers for various applications. Used containers are: RFID identification, Discovery and Plug and Produce editor.

SICK Laser Scanners: Two laser scanners are installed at the diagonally opposite side. Each one has a range of 270 degrees and serves for detection of humans inside the workspace of the robot.

Kuka iiwa: The 7 DOF robot that is used to execute the assembly and which allows human robot interaction.

Gimatic gripper: Gripper installed at a Y-mount underneath the robot flange. The gripper jaws are specially designed to pick Han plugs.

Afag gripper: Gripper installed at a Y-mount underneath the robot flange. The gripper jaws are specially designed to pick Han hinged frames and allow opening and closing the frames inside the carrier.

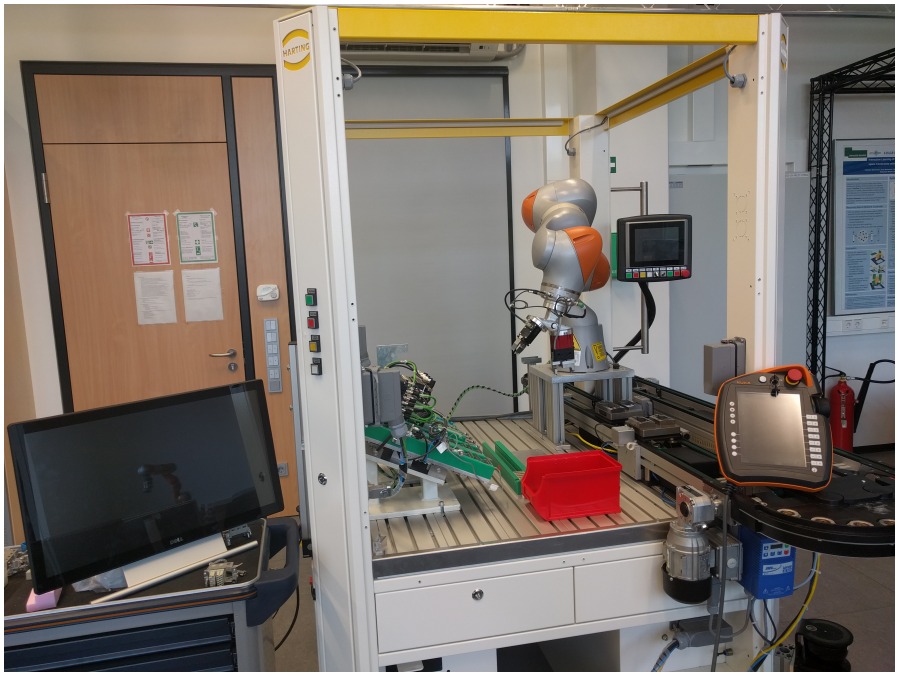


Figure 5.1: *HARTING* Production Cell

5.2 Modelled Assembly Process

The task for the robot is to assemble a Han-modular hinged frame with different plugs. Figure 5.2 shows four exemplary frame sizes with various plug configurations. The abstract process is shown in figure 5.3 in the appendix. Its detailed modelling happened in sub-processes in separate files (e.g. figure B.4).

To assemble the frame the robot first has to pick up the frame from a storage and place it on a carrier which is fixed at a stop station on a conveyor belt. Since the frame is closed, when taken from the storage, the next step is to open the frame inside the carrier. This is done through a force based movement where the *Afag* gripper pushes one of the edges downwards which releases a clamp. Afterwards the frame is opened inside the mount of the carrier. Next the robot uses the *Gimatic* gripper to pick a plug from from a plug storage that should be installed at the first position. Since the plugs are different in shape and length and some have barbs, the force that is needed to push them into their position is always different and the position must be very precise. The pick and place of the plugs has to be repeated x-times, dependent on the size of the frame. To minimize the number of task frames that have to be taught, only the first position is taught and the other positions are calculated through an offset. If all plugs have been installed, the hinged frame has to be closed with the *Afag Gripper*. Therefore again a force based movement was used together with some compliance to lower the needed forces to a minimum and to avoid damages from the gripper against the frame. Afterwards the assembled frame can be picked from the carrier and either be put to a storage box directly or it might be moved to a quality inspection component first and afterwards to the box.

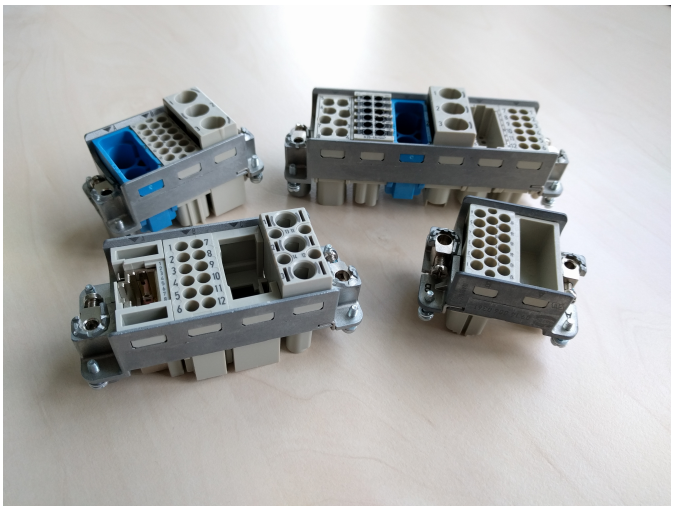


Figure 5.2: *HARTING Han variants*

5.2.1 Exemplary Task composition

The skill framework has been used to model the complete assembly process (including all variances: e.g. different plugs for different orders, different frame sizes, etc.). As an example how the skill framework was used and integrated, figure 5.3 shows how the assemble frame task is composed and hierarchically structured. The rectangles visualize the skill primitives, the rounded rectangles show the skills, the ellipse illustrates a task and the octagon represents another component which is part of the assembly process but not controlled by or through the robot.

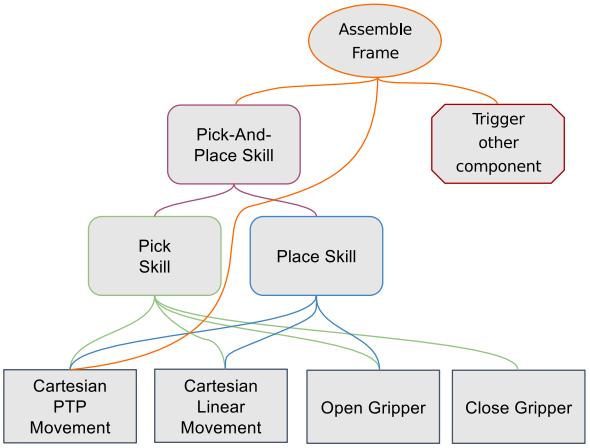


Figure 5.3: Assembly Task

5.3 Plug and Produce

The flexibility of the assembly process should be enhanced through a *Plug and Produce* architecture. Pfrommer et al. already drew the connection between skill based modelling and reconfigurable production in their article from 2015[1]. Since the *Plug'n'Produce* concept was also one focus of the FlexiMiR project it was obvious that the developed skill framework could also be used to support the concept.

[1] cf. Pfrommer et al. (2015)

The basic idea of the *Plug and Produce* concept can be described in the following way: The user wants to integrate a new component into his production process. Therefore he simply connects the component to the production cell. The production cell recognizes and registers the new component with help of the *Discovery* container, installed on the *HARTING Mica*. Afterwards the user will be asked to integrate the component which will be supported through a wizard that is shown in the *AWAre GUI*. The wizard will contain an important step that

involves the robot: The localization of the new component within the working environment (see section 5.3.1 for details). Additionally if the component is actively involved in the production process, it may be necessary to teach positions that are related to the component. This has to be supported through the self description, that each component has to provide. The self description may only contain a defined collection of task frames that have to be taught to interact with it or it may even provide a number of task frames which are relative to its base. This would lead to a better usability because the process possesses a higher automation and less user interaction. Another part of the wizard should be the motion planing of the robot because the placement of the component inside the workspace of the robot might lead to collision in various movements included in the process (see section 5.4 for details).

5.3.1 Localization of Objects

To determine the position of a component inside the workspace of the robot, it can be approached at three different points which lay in the same plane. Because the robot knows its own position inside the world (see section 3.7), it is able to determine the precise location of its mounted tool. To enhance the measurement precision, a measurement tip is inserted into one of the grippers and then the three points have to be approached interactively by the user. The integration wizard (see section 5.3) will lead the user through this process (see figure 5.5) and allows to activate the hand guide mode and to store the approached positions. The measurement order is important because an algorithm that calculates the orientation declares the orientation of the X-Axis from the first to the third measured point (see figure 5.4).

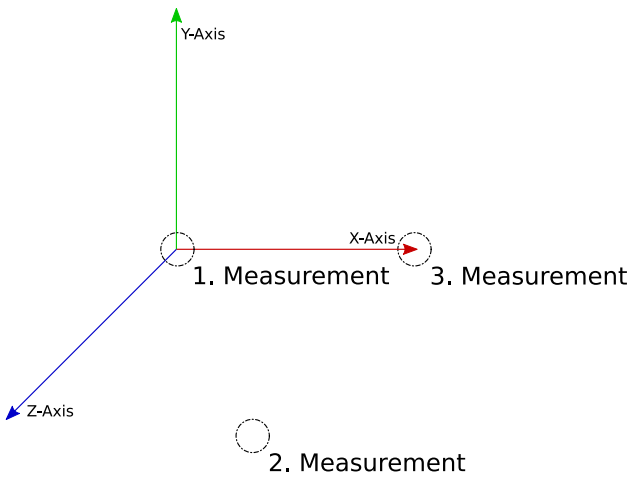
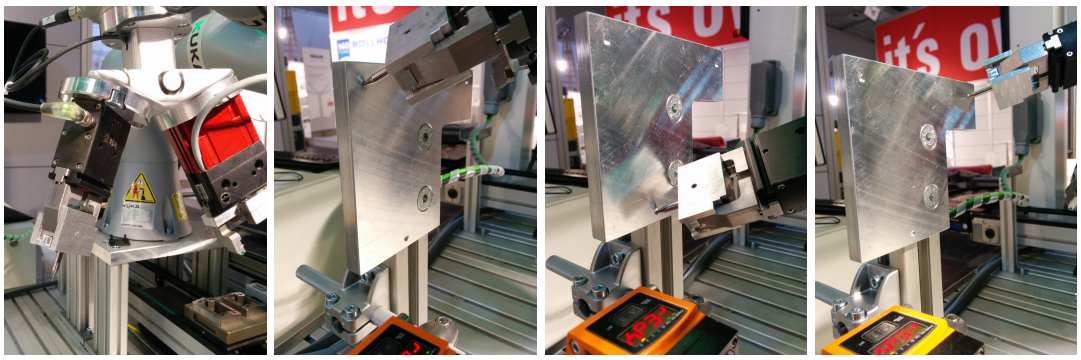


Figure 5.4: Measurement Order and Resulting Orientation

After the three points have been determined and stored, the algorithm implemented in the robot skills plugin of the *AWAre framework* is able to calculate the orientation of the component. The position will be defined through the first approached point. This is the base frame of each component. Other positions can be provided in the self description of the component relative to this task frame.



(a) Measurement Tip **(b)** Measure TF 1 **(c)** Measure TF 2 **(d)** Measure TF 3

Figure 5.5: Measurements for Localization

The necessary steps to calculate the orientation of a plane in 3D space is shown in Algorithm 1. Equation 5.1 shows the definition of the three points that define the plane. They are constructed from the input of the three measurements which result in three task frames. In equation 5.2 the two vectors are shown which span up the plane. The orientation of the plane is defined in equation 5.3 where the three axes are constructed. The x-axis is simply defined to face into direction of point $\vec{P_3}$. The z-axis can be defined by the cross product of the vectors $\vec{v_1}$ and $\vec{v_2}$ because since they span up the plane, the resulting vector of the cross product has to be orthogonally aligned. The same mathematical law is applied for the construction of the y-axis which is the result of the cross product of vector $\vec{x}$ and $\vec{z}$. For the algorithm it is important whether the second measurement point is located above or below the vector between $\vec{P_1}$ and $\vec{P_3}$. Currently the implementation specifies that the y-axis is facing upwards, away from the second measurement point. The z-axis is facing in the direction from where the measurements have been done so normally out of the component (see figures 5.4 and 5.6b).

Combined together the three vectors form the rotation matrix shown in equation 5.4 from which the angles can be calculated through the three equations shown in 5.5. This calculation assumes that the angles are based on the Tait-

Bryan Euler Angles definition. This defines that alpha $\hat{=}$ yaw $\hat{=}$ rotation around z, beta $\hat{=}$ pitch $\hat{=}$ rotation around y and gamma $\hat{=}$ roll $\hat{=}$ rotation around x.

$$\vec{P_1} = \begin{pmatrix} x_{TF1} \\ y_{TF1} \\ z_{TF1} \end{pmatrix}, \quad \vec{P_2} = \begin{pmatrix} x_{TF2} \\ y_{TF2} \\ z_{TF2} \end{pmatrix}, \quad \vec{P_3} = \begin{pmatrix} x_{TF3} \\ y_{TF3} \\ z_{TF3} \end{pmatrix} \tag{5.1}$$

$$\begin{aligned} \vec{v_1} &= \vec{P_2} - \vec{P_1} \\ \vec{v_2} &= \vec{P_3} - \vec{P_1} \end{aligned} \tag{5.2}$$

$$\begin{aligned} \vec{x} &= \|\vec{P_3} - \vec{P_1}\| \\ \vec{z} &= \|\vec{v_1} \times \vec{v_2}\| \\ \vec{y} &= \|\vec{x} \times \vec{z}\| \end{aligned} \tag{5.3}$$

$$\vec{R} = \begin{Bmatrix} x_1, y_1, z_1 \\ x_2, y_2, z_2 \\ x_3, y_3, z_3 \end{Bmatrix} \tag{5.4}$$

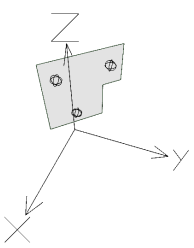
$$\begin{aligned} \alpha &= \arctan 2(x_2, x_1) \\ \beta &= \arcsin(-x_3) \\ \gamma &= \arctan 2(y_3, z_3) \end{aligned} \tag{5.5}$$

Algorithm 1: Determine Orientation of a Plane in $\mathbb{R}^3$

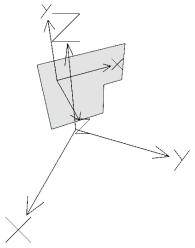
To verify the functionality 7 Measurements of the IFM and a storage component have been made which results are documented in table 5.1. Figure 5.6 first visualizes the three measurements points of the IFM component which is placed in a 45 degree angle in the corner of the production cell (subfigure 5.6a). The calculated coordinate system for the component is then shown in figure 5.6b. The goal of the localization is to find the transformation between the world coordinate system and the local coordinate system. Figure 5.6c shows the four different test positions for the IFM component. The translation is based on the first measurement point and the rotation has been calculated correctly. The last figure (5.6d) additionally shows the three storage components that have been measured. It is important to note that the rotation of the coordinate systems is a little bit different since the frame storage components have the second measurement point located above the x-axis. This results in the y-axis facing downwards and the z-axis facing towards the component. This is a little drawback which can be solved by offering different measurement methods with according algorithms or determining a fixed alignment of the measurement points.

IFM 45 Degrees Corner	**X**	**Y**	**Z**
Point 1	178.93	68.99	340.75
Point 2	155.23	90.12	251.21
Point 3	107.12	140.08	341.74
Correct Rotation	91.05	-0.56	135.28
IFM 45 Degrees Robot	**X**	**Y**	**Z**
Point 1	138.79	384.49	342.89
Point 2	157.68	406.71	251.71
Point 3	207.78	456.62	340.05
Correct Rotation	91.08	1.63	46.27
IFM 90 Degrees Window	**X**	**Y**	**Z**
Point 1	2.82	181.69	341.09
Point 2	0.10	213.32	253.48
Point 3	1.46	284.45	341.94
Correct Rotation	91.50	-0.47	90.76
IFM 90 Degrees Hanning	**X**	**Y**	**Z**
Point 1	216.80	0.26	336.48
Point 2	185.51	-0.67	248.20
Point 3	117.12	-1.83	339.68
Correct Rotation	90.18	-1.84	-178.80
Frame Storage Mounted	**X**	**Y**	**Z**
Point 1	857.34	339.89	99.05
Point 2	753.42	257.80	138.55
Point 3	668.42	340.71	101.46
Correct Rotation	-155.17	-0.73	179.75
Frame Storage Operator	**X**	**Y**	**Z**
Point 1	751.87	666.96	116.09
Point 2	812.80	570.85	142.98
Point 3	752.17	476.29	114.93
Correct Rotation	-155.68	0.35	-89.91
Frame Storage Carrier	**X**	**Y**	**Z**
Point 1	866.08	433.62	112.57
Point 2	962.03	495.93	142.71
Point 3	1057.12	435.50	112.14
Correct Rotation	-153.68	0.13	0.56

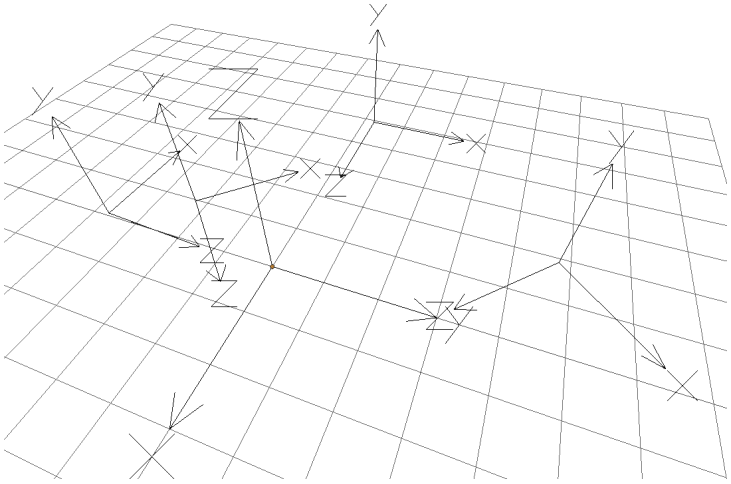
Table 5.1: Measurements Localization



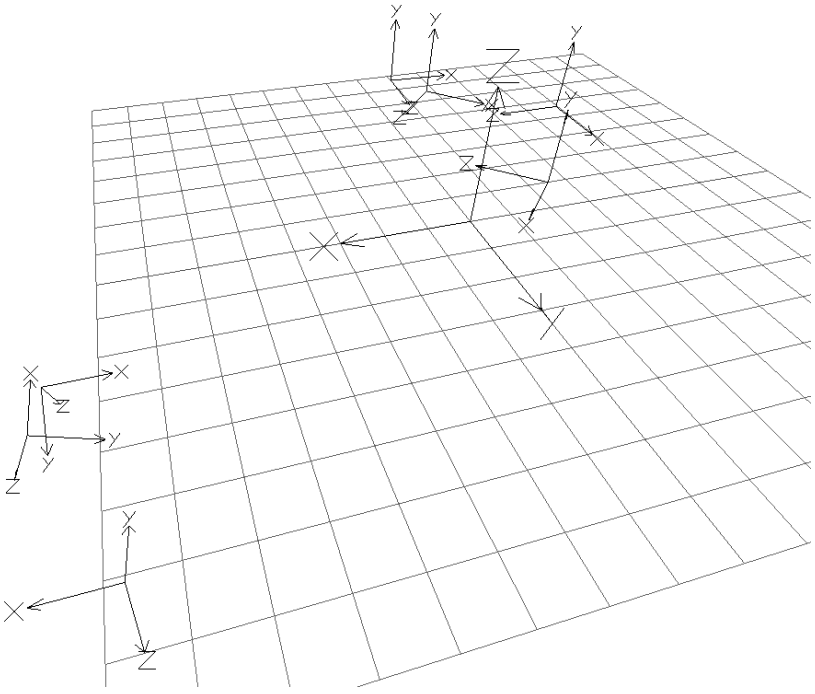
(a) Measurement Points IFM Comonent 45 Degrees



(b) Calculated Local Coordinate Orientation



(c) Different Placements of IFM Component



(d) All Placements of IFM and Storages

Figure 5.6: Translations and Rotations

5.4 Simulation and Motion Planning

Section 4.6.3 treated the topic of teaching trajectories to manually define motions between two task frames. Soon, if the number of movements exceeds a certain threshold, the manual teaching of motions can become quite expensive. It makes sense to automate the motion planing between task frames not least because each motion of the production process would have to be checked for collisions if a new component is integrated into the workspace of the robot.

The motion planning is not part of the skill framework itself and was developed by other collaborators. It is provided by the simulation and can be triggered through RPC calls based on the middleware RSB. The simulation is based on *Visual components* and contains models of all components as well as a model of the production cell itself. The idea behind the self descriptions of the components is that they would also provide 3D models of themselves which can be integrated in the simulation after the component has been localized.

The simulation offers interfaces to check paths for collisions and to plan new paths based on a start and end position. The path planing was integrated into the wizard and is supported by the skill framework. Therefore the task frames from the start and end position are send from the robot skills plugin to the robot controller. With help of its inverse kinematic, the robot is able to determine the joint configuration for the cartesian position and sends it back to the plugin. The plugin then sends the two joint configurations encapsulated in a RST datatype to the simulation which receives them and starts a path planing between them (see figure 5.7 for one possible result). Theoretically the simulation could use different path planning algorithms but in an first approach, OMPL[2] was used. Based on a given time threshold it tries to find a solution and either returns the the trajectory in form of a joint configuration collection or it returns null if no solution could be found in the given time.

[2] Abbr. for Open Motion Planning Library

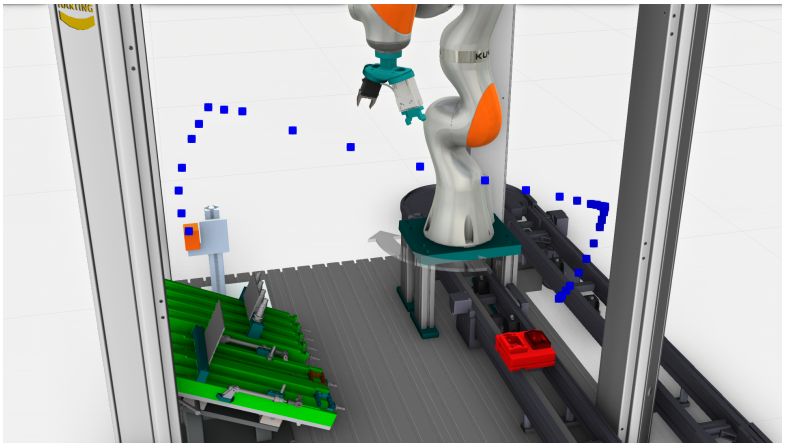


Figure 5.7: Calculated Trajectory from Carrier to IFM Component

5.5 Safety

By the time skill frameworks should be deployed to real industrial scenarios and not only be used for research purposes, lots of standards, regulations and laws have to be considered. Kagermann et al.[3] claim that safety also plays an important role in implementing *Industrie 4.0*. The operational safety has to ensure that a system itself is safe to use and at the same time it has to implement robustness and low fault rates (which stop the production for a longer period of time). Safety is also a criterion which is crucial for acceptance of a production system. So besides a good usability it is important for a user to be confident when interacting with the system.

[3] cf. Kagermann et al. (2013, p. 46-50)

5.5.1 Risk Assessment

To determine the safety of a robot or even a whole production system, a risk assessment should be done by a trained expert. The risk assessment should be based on the *Machinery Directive*[4] and ISO 10218, part 1 and 2[5]. Requirements for the risk assessment itself can be found in ISO 12100. Furthermore ISO / TS 15066 has been released which is not yet a standard but should also be considered in designing the safety of collaborative systems[6]. The ISO standard 13849-1 defines 5 safety performance levels (a-e which reach from a: negligible to e: very high)[7].

[4] Europäsches Paralament / Europäischer Rat (2006)

[5] cf. Bélanger-Barrette (2016, p. 7)

[6] cf. Robotiq (2016)

[7] cf. Matthias (2017, p. 28,36)

There are two basic hazard types that can be distinguished for collaborative processes:
1. Free and transient contact. (Bumping into a human who can escape the situation.) Appropriate risk reduction can be resolved by reducing the speed of the robot and using a collision detection based on external force monitoring in all axes.
2. Quasi-static Contact (Clamping of a human body part between the robot and another object.) In this case appropriate risk reduction can be resolved only through limiting the speed to rather small values because the stopping distance is short. Monitoring functions can be collision detection and force monitoring.
A list of maximum forces allowed for different regions of the body (biomechanical thresholds) can be found in a paper from the german statutory accident insurance[8].

[8] cf. Fachbereich Holz und Metall der DGUV (2017, p. 7-8)

The risk assessment should identify, describe and list all hazards, that can occur in the desired manufacturing process. Afterwards each hazard can be classified and a risk estimation and evaluation can be done. If the result shows, that the risk can not be tolerated, measures have to be taken to reduce the risk. Afterwards the risk sources have to be identified, estimated and evaluated again[5].

Furthermore four different types and possible risk reductions have been identified and defined in ISO 10218-1 (see. table 5.2).

Collaborative Operation:	Explanation:
Safety Monitored Stop:	As soon as a human enters the workspace of the robot (the collaborative workspace), the robot stops its motions. This does not mean that it went into an emergency stop and needs manual recovery.
Hand Guiding:	Based on a sensed input force from a human collaborator, the robot gets compliant and can be directed to any position in its workspace.
Speed and Separation Monitoring:	Through the definition of different safety zones and adaptive behaviour can be realized: The robot moves at full speed if any human is outside of any safety zone, it may slow down if a person enters a first safety zone and it may stop completely if the human enters a second safety zone.
Power and Force Limiting:	If the robot hits something in its environment it will either be stopped, be compliant or automatically move in the opposite direction. The forces are also limited if a human is involved in the process and they would pose a risk otherwise.

Table 5.2: Collaborative Operations

5.5.2 Kuka Safety

[9] cf. KUKA (2016, p. 26)

The *Kuka iiwa* is certified for Category 3 and Performance Level d following EN ISO 13849-1[9]. This means that the robot poses a high risk and must be extended by further safety mechanisms to allow a collaborative work.
Category 3 also means that the safety architecture of the robot controller implements dual-channel, monitoring & cross monitoring and uses proven components.

Regarding the collaborative operations listed in table 5.2, the last three operations were implemented. The first operation seemed to be too strict and disruptive for an industrial assembly process. A full risk assessment would still be necessary to investigate whether all possible hazards have been found and are covered by the implementation.

On the one hand the *Kuka Sunrise API* already offers a lot of functionality to implement safety mechanisms and to activate and deactivate specific parts as needed. On the other hand additional safety mechanisms were introduced: The production cell itself has a safety system that monitors the functionality of all the other automation components. It also has an emergency shutdown button which stops all components immediately, including the robot. Two laser scanners have been installed as well to be able to monitor the complete surrounding of the cell (see screenshot of *SICK Safety Designer* in appendix A.4 and A.5). With them it is possible to define safety zones which can trigger different behaviours of the robot. They are integrated through the IO mappings of the *Kuka Controller* so their state can directly be monitored in the *RoboticsAPIApplication*. *Kuka Sunrise* supports the cartesian monitoring of workspaces and protected spaces. The workspace monitoring can be used to constrain the space where the robot is allowed to move. For the assembly task, the workspace has been defined through the production cell which means that the robot should not be allowed to leave the cell with any part (neither mounted tools nor single joints). The protected spaces have to be defined in the safety configuration (see table 5.3) which can only be changed through the *Kuka Sunrise Workbench*. This makes it rather unflexible for changes in production environment since each change in the safety configuration results in a restart and explicit reactivation of the configuration through the operator with help of the *Kuka SmartPAD*. The protected spaces have not been used in the implementation for that reason.

Bänziger et al. propose to extend skill frameworks with an behaviour concept to integrate safety functions and different movement modes[10]. Although the idea seems quite attractive, for example to even change the execution mode from position to impedance during a skill execution, this is not easy to realize and is currently not supported by the *Sunrise API*. At the moment, specifications like e.g. control mode (position/impedance) are implemented as parameters for each skill and will be valid till the skill execution has been finished. To allow *Speed and Seperation Monitoring* even though, the possibility of the *Kuka API* to override the commanded speeds has been used (see listing 5.1).

[10] cf. Bänziger, Kunz and Wegener (2017, p. 78)

The *ek11group* allows to read the current values of the analogue inputs. The port called "Eing_0" is connected to the safety system which sets the port to low (≙false) if a person is detected in the range of the laser scanners or to high (≙true) if no person is detected. So the *BooleanIOCondition* will be enabled if the signal at port "Eing_0" is changed to *false* (see l. 34). The *ConditionObserver* is configured to trigger the *IAnyEdgeListener* (l. 9) if the condition is enabled (l. 35). Inside the *onAnyEdge()* method it is checked whether the application is not in reduced speed mode (which would mean the program is executed in T1[11] mode) and whether the condition was enabled. If this is the case, something is detected by the safety system and the *for-loop* (l. 15-19) reduces the execution speed of

[11] T1 mode is meant for manual movements and testing not for production

all movements by stepwise lowering the application override to 0.2. In case the condition was not enabled the other *for-loop* is triggered, which increases the override stepwise back to 1 which means the full speeds defined for the executed movements will be used. Currently this is all done in the *initialize()* method of the *SkillProvider* class.

```
1  [...]
2  private EK1100IOGroup ek11group;
3  private ConditionObserver overrideReduction;
4  [...]
5  @Override
6  public void initialize() {
7    [...]
8    ek11group = new EK1100IOGroup(getController("KUKA_Sunrise_Cabinet_1"));
9    IAnyEdgeListener overrideListener = new IAnyEdgeListener() {
10
11     @Override
12     public void onAnyEdge(ConditionObserver conditionObserver, Date time,
13     int missedEvents, boolean conditionValue) {
14       if (conditionValue && !lbr.getOperationMode().isReducedSpeedTestMode()){
15         for (double override = 1; override >= 0.2; override = override - 0.1)
16         {
17           ThreadUtil.milliSleep(2);
18           getApplicationControl().setApplicationOverride(override);
19         }
20
21       }
22       else if (!conditionValue)
23       {
24         for (double override =0.2; override <= 1; override = override + 0.1)
25         {
26           ThreadUtil.milliSleep(2);
27           getApplicationControl().setApplicationOverride(override);
28         }
29
30       }
31     }
32   };
33
34   BooleanIOCondition input = new BooleanIOCondition(ek11group.getIO("Eing_0",
         false), false);
35   overrideReduction = getObserverManager().createConditionObserver(input,
         NotificationType.OnEnable, overrideListener);
36   overrideReduction.enable();
37 }
38 [...]
```

Listing 5.1: Safety Override

5.5.2.1 PSM and ESM Configuration

Besides the override solution to adjust the speed of the robot movements, the safety configuration shown in table 5.3 gives more information about the taken efforts to allow a safe user interaction. The configuration works the following way: The *Kuka PSM* and the *User PSM* are always active and monitor the configured *AMFs*[12]. The *ESMs*[13] can be activated from the programming code of an application. This means, only one at a time will be active.

[12] Abbr. for Atomic Monitoring Function

[13] Abbr. for Event-Driven Safety Monitoring

The *ESM* state 1 is used for normal movements during the process. It monitors the external force to detect collisions and has a maximum cartesian speed defined. If one of the conditions is violated, the robot stops on its calculated path and the application can be resumed manually, if the violation has been eliminated.

ESM state 2 is necessary if hand guiding should be used. It has to be activated before the *handGuiding()* command in the code is triggered. If the hand guiding switch is released, the approval for the hand guiding is no longer given and a path maintaining stop is initiated.

For movements where higher forces are needed, e.g. force-based skills, *ESM* state 3 has been configured. It defines that a maximum force of 150 Newton is allowed to act upon the tool center point (TCP). This state should only be activated if the safety for the user is ensured. At best this means that no person is in range of the robot, in other cases it might be a solution to only activate the state if a person can not physically enter the space between the robots end effector and the object on which the force should act upon. The skill architecture ensures, that the *ESM* state is always reset to 1 after the movement has been finished.

State Descriptor:		**Safety Functions:**			
	Category	AMF1	AMF2	AMF3	Reaction
Kuka PSM	Emergency Stop Local	Emergency Stop SmartPAD			Stop 1 (path-maintaining)
	Approval	Approval Hand Guiding inactive	Operation Mode Test	Approval smartPAD inactive	Stop 1 (path-maintaining)
	Speed Monitoring	Approval Hand Guiding active	Cartesian Speed Monitoring, First Kinematic = 1000 mm/s		Stop 1 (path-maintaining)
User PSM	Emergency Stop External	Input Signal(1)			Stop 1 (path-maintaining)
	Operator Safety	Input Signal (2)	Time Delay (1) = 100ms	Cartesian Velocity Monitoring = 2500mm/s	Stop 1 (path-maintaining)
	Space Monitoring	Cartesian Workspace Monitoring First Kinematic, Robot and Tool, X = -750mm, Y = -280mm, Z = -245mm, A = 315°, B = 0°, C = 0°, Length=1400mm, Width=1400mm, Height=1850mm			Stop 1 (path-maintaining)
ESM 1		Collision Detection, First Kinematic, Max. external Moment: 25Nm			Stop 1 (path-maintaining)
		Cartesian Velocity Monitoring = 1000mm/s			Stop 1 (path-maintaining)
ESM 2		Approval Hand Guiding inactive			Stop 1 (path-maintaining)
ESM 3		TCP Force Monitoring, First Kinematic, Max. TCP Force = 150N			Stop 1

Table 5.3: Safety Configuration Kuka

6 Discussion

The goal of this thesis was to design and implement a robot skill framework based on BPMN which meets all of the following requirements: Modularization, Parametrization, Sequencing, Hierarchies, World Model and Safety. The possibility to interactively parametrize the skills through human-robot interaction and manual configuration was an important focus. A concept for the skill framework has been developed, implemented and applied to a real industrial production scenario. This has been demonstrated at the *it's owl* booth at *Hannover Messe* 2017.

6.1 Parametrization

Through the dependence on BPMN, parametrization of skill primitives, skills and tasks is possible and mechanisms for parameter forwarding and processing are part of the *Camunda Framework*. Especially the scripting possibility for parameters allows to develop flexible parametrization and complex sequences with little effort. The only drawback regarding this feature is that basic programming knowledge is required.

During the process design phase, the tasks and skills were modelled with the tool *Camunda Modeler*. The tool offers a good support and nearly covers the complete BPMN scope, but it is still too complex for the normal shop floor worker which should be enabled to design and adapt production without much effort and previous knowledge. A customized user interface for skill parametrization and sequencing could be developed based on the BPMN capabilities but abstracting from them in the interface. The user would be guided through the process of modelling a production scenario where he would start from scratch. The interface would transform the input into BPMN models, so the user could focus on the modelling and would not have to take care of BPMN logic or other distractions.

The requirement for interactive parametrization has been realized through a two phase approach. Some parameters are added during the design process (specification phase), interactively taught positions are added afterwards in a teaching phase. If all positions are available, the BPMN process and all contained skills can be executed. The concept for interactive paramterization (see

figure 3.4a) has to be further examined and approaches have to be developed how this process can be generalized for easier integration into all kinds of skills.

Another advancement would be to include a pre execution step which parses the plan to check whether all necessary parameters are available. Currently the plan would execute until a skill is reached which is missing some required parameter. Since some parameters are dynamic and depend on the configured process, the checking for all possible parameter inputs gets complex quite fast. Strategies have to be developed and tested in context of the available framework.

6.2 Robustness

One of the main requirements for industrial applications is their robustness. Related to a skill framework this means that skills have to be executed a number of times without producing errors which would stop the process and require manual fixing. Also other demands like precision and efficiency would have to be fulfilled in a successful integration.

One advantage of BPMN is the integrated error modelling capability. Within the scope of this thesis error handling has been used exemplary for preconditions and in the application phase error handling was also introduced at higher task levels. Future work could include investigation in error handling strategies to improve robustness and provide information where the process failed to support the operator in fixing the error. The implementation based on BPMN supports this quite well because different types of errors can be defined, their consequences can be explicitly modelled and error handling can also be separated for different layers to allow decent reactions.

The implementation based on position related skills rather then object-centred skills proofed good results in the investigated assembly task. A high precision is needed for the assembly which would have been harder to achieve through object-centred skills which rely on vision and image processing. Even though the skill framework itself would also allow to develop skills which are based on visual feedback. Since the demonstrated assembly task required precision in sub millimetre range, the interactive teaching of positions was also brought to its limits. Skills including strategies to avoid the necessity of precision by using force sensing and other mechanism might be integrated in the framework as well and offer a solution to improve the robustness even further.

6.3 Composition

Composition condenses the requirements of sequencing, modularization and hierarchies. The chosen modelling framework BPMN supports all of these demands quite well. Complex logic flows exceed the basic requirements for skill sequencing and offer even more possibilities.

One drawback of the implementation using the *Camunda Framework* is the parallel execution of skills and tasks. By default the framework uses a concept called *Optimistic-locking* and therefore only allows *Exclusive jobs* which means that only one *BPMN Task* will be executed at a time. This is done for consistency reasons and might be configured differently, but this is a edge case of the framework and would have to be examined in depth.

Parallel executions can be an interesting topic if multiple robot arms should be used, but the current framework would not support easy to implement synchronization. Especially real time monitoring and collision avoidance during execution of two skill primitive movements were not in the scope of this thesis.

Since the skill primitives have been implemented as *Java Delegates* only depended on the middleware, in the broader sense also a hardware abstraction layer was established. The application of the framework onto another robot would though be possible, assuming that the required callbacks can be implemented on the robot architecture and would need the same set of parameters. The effort would depend on the programming framework that the robot can be controlled with.

The fluent borders between the modelling of tasks, skills and primitives can be seen as a drawback because they might lack a little formalization. Nevertheless the possibilities to combine the different hierarchy levels in processes without additional extensions offer many advantages and flexibility.

Another important aspect to mention is the unified modelling with BPMN. It allows the composition of robot skills and other automation components in the same environment which is a huge advantage compared to other skill frameworks. The usage of a industrial business standard allows integration into real world scenarios and enhances acceptance of such an approach in the field.

6.4 World Model

The world model is currently stored in the *Neo4j* database. The graph structure of this database offers a lot of potential for future development. Following the *Plug and Produce* concept, components can be integrated easily into a process. Therefore standardized information can be added to the database if a component is plugged into the system. For example components can provide task

frames which can be added to the hierarchy through simply adding relations and those task frames can then be used to parametrize skills which are added to integrate the component in the production process.

Future research would have to investigate into the standardized self descriptions of the components, storing BPMN processes inside the world model and then using relationships to connect BPMN processes and parameters stored in the same database. This would enhance the skill architecture integration into the whole automation process and the *Plug and Produce* concept.

6.5 Safety

To allow human robot interaction at least a base level of safety should be guaranteed. Since the robot itself is categorized as a high risk, safety enhancements have been implemented and integrated on different levels. Most of it is located on the robot specific implementation. This includes the integration of the laser scanners, dynamic changing of safety states and hand guiding rules. One could argue that this is a responsibility of the skill primitive programmer, meaning that he has to ensure that the skill primitives can be used in a safely manner. It might be interesting to evaluate whether an approach similar to the one presented by Bänzinger et. al[1] can be realized meaning the extension of the skill framework with behaviours. Those behaviours can enhance the safety and provide a more flexible way of controlling execution modes, but it might be necessary to program the skills on a lower level which would be more controller focused.

[1] cf. Bänziger et al. (2017)

If force based movements are used, the user designing the process must be aware of the possible thread that will arise through their usage. The user must be specially trained to have a fundamental understanding of the skills and their behaviour. If an interface will be realized like proposed above, it should also indicate the risks to improve the safety.

6.6 Potentials and Limitations

Generally the assembly task that the skill framework has been applied to involves only a small number of skills. Four skills have been implemented: The pick and the place skill, the skill to open and the skill to close the frame (both basically force-based movements with pre and post positions). To test the applicability, the framework has to be used in more scenarios. This will show which further skills are missing and how much effort it will cost to integrate these.

Currently the measurements for the localization of a component are controlled by the user interface. This might be generalized to a skill which can be used for all components and in different contexts. Furthermore skills can be developed which implement strategies for solving problems like the peg-in-hole problem. They can then be integrated into the production process as error handling procedures in case the robot could not install one of the plugs at the designated position. Again, also for this approach it can be evaluated how much effort this would take and if it is practical to model such strategies in the skill framework itself.

The approach of this skill framework still considers the human as an important part of the production and design process. While other research aims on automatic task decomposition and task planning, this was not considered for the developed framework of this thesis. It must be investigated how much potential the modelling in BPMN offers to optimize and automate planning. Furthermore parametrization can maybe be improved through machine learning algorithms with different focusses like efficiency, speed, quality and so on.

6.7 Summary

The developed skill framework builds a good foundation for further research and extensions. It is based on task-level programming and can be interactively parametrized through teaching positions and even trajectories. Based on BPMN it implements a hierarchy of three layers with increasing abstraction from skill primitives over skills to tasks. The integration of other components is easy and the framework scales from small processes to complex automation environments.

7 Conclusion and Outlook

7.1 Conclusion

The goal of this thesis was to design and implement a robot skill framework. At first, the beginnings of robot skill programming were shortly summarized and current approaches from research and business were mentioned. Then the concept of the skill framework was presented which showed that the realization is based on the graphical modelling language BPMN and the implementation allows to model skills on three abstraction layers: The skill primitives, basic and complex skills and tasks. The parametrization of the skills was realized in two phases: Some parameters are provided during the process modelling and others are taught interactively. Afterwards the implementation was described which included the chosen architecture and a detailed specification of the robot and BPMN related development results. Next, the application of the skill framework onto an assembly task was presented and finally a discussion pointed out the strengths and weaknesses of the current results.

More than nine skill primitives have been implemented and were tested in a real world industrial assembly task. The combination of the skills and a customized user interface was used to parametrize a process. A reconfiguration scenario with an additional component and integrated path planning could also be demonstrated. Various safety mechanisms were integrated to establish a collaborative behaviour of the robot.

The presented approach for a skill framework is perfectly suitable for industrial scenarios. The foundation on BPMN also allows the integration of other components which in turn supports a unified process model and avoids isolated applications. The scalability of the framework allows its usage from small tasks to complex scenarios.

7.2 Outlook

The results of this thesis offer a lot of potential for extensions and further research. First steps can be the development of an assisted process to design a robot behaviour from scratch and of course the application of the framework to

other industrial tasks. The creation of a skill library would support the usability of the framework because less development would be necessary for every new scenario. In the ideal case a simple combination, sequencing and parametrization of available skills would be sufficient to realize most scenarios.

Other areas of future investigations can include tests of parallelization, application to different robot types, integration of machine learning and other optimization strategies.

User studies which examine the usability and practicability of the concept have to be conducted. Benchmarks could also be interesting to measure the economical value of such a framework. Socially it has to be investigated how the employee roles change through the new concept.

A Appendix

A.1 Camunda Modeler

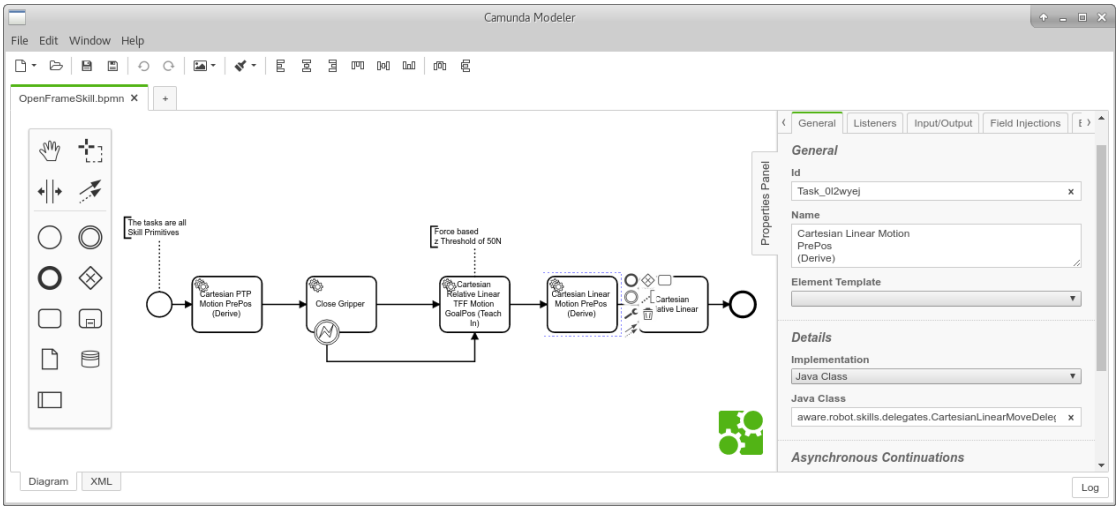


Figure A.1: Screenshot of the *Camunda Modeler* - General Tab

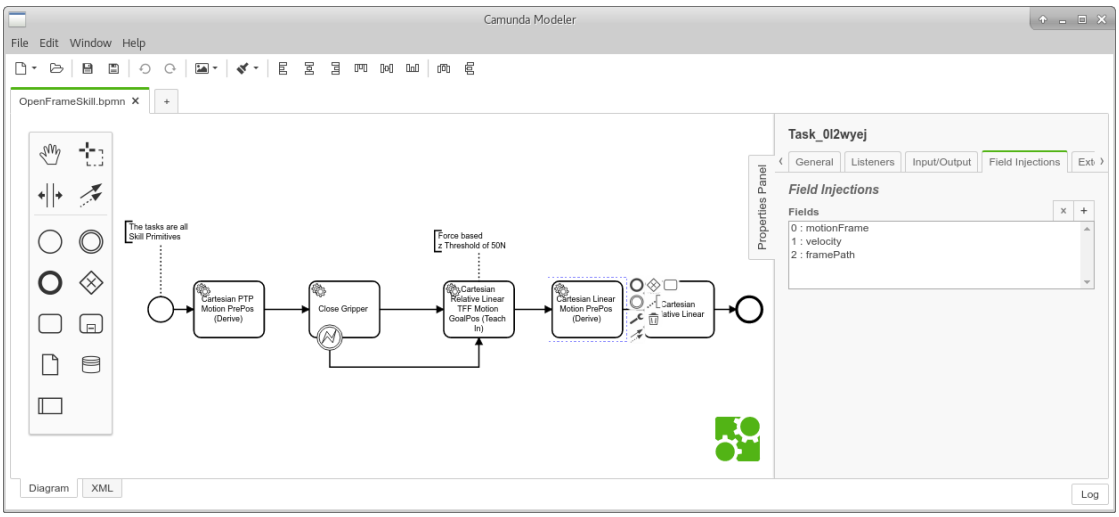


Figure A.2: Screenshot of the *Camunda Modeler* - Field Injections Tab

A.2 AWAre GUI - Robot Control Center

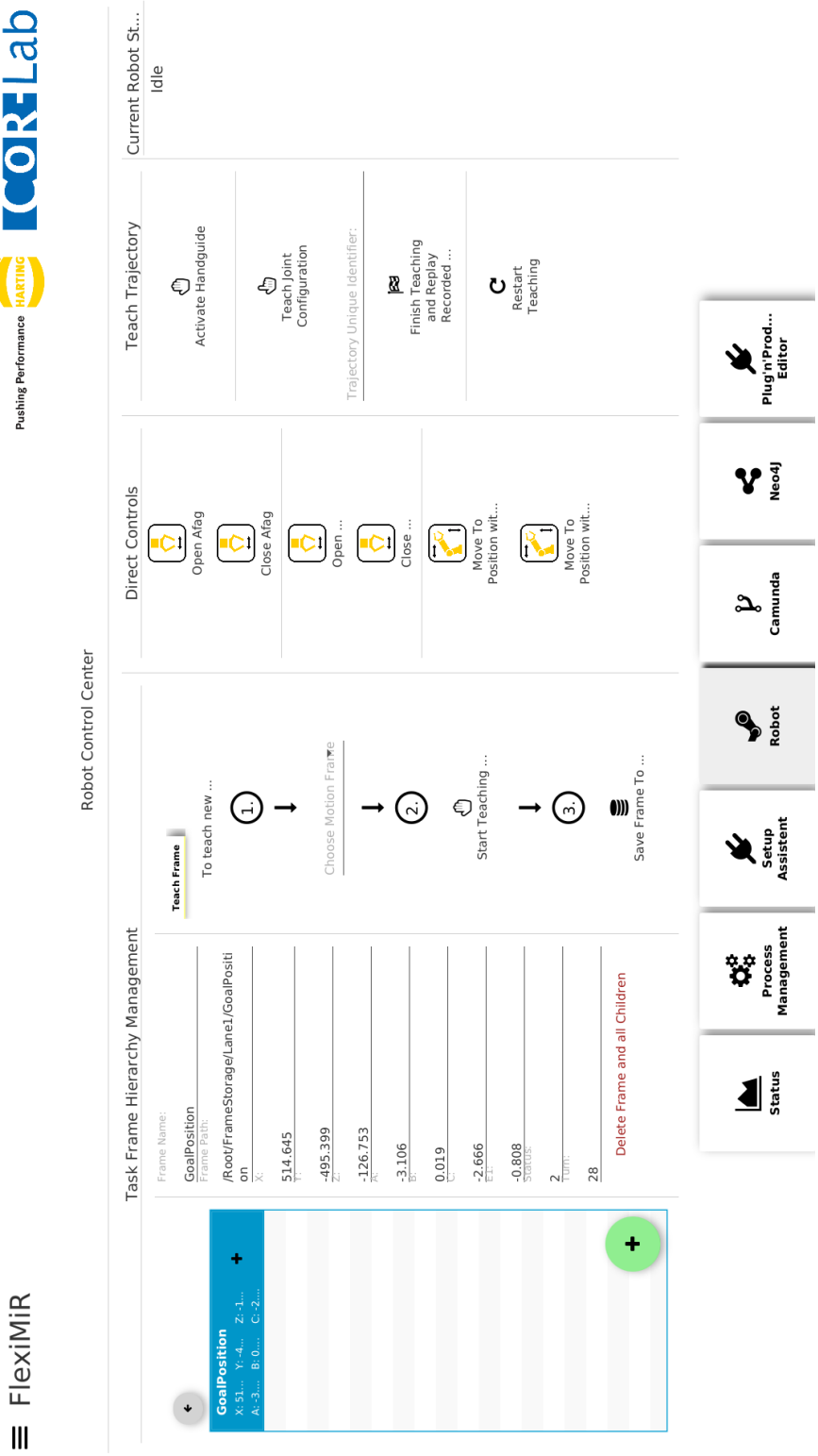


Figure A.3: Screenshot of the *AWAre GUI*

A.3 SICK Safety Designer

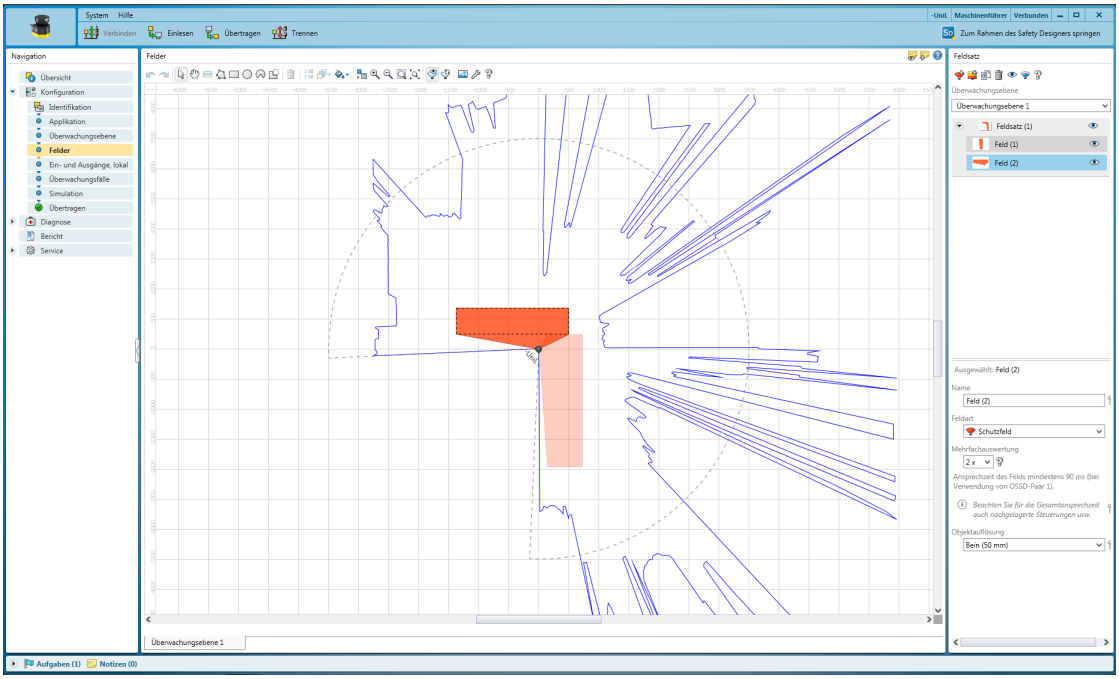


Figure A.4: Configured Safety Zone of Laser Scanner 1

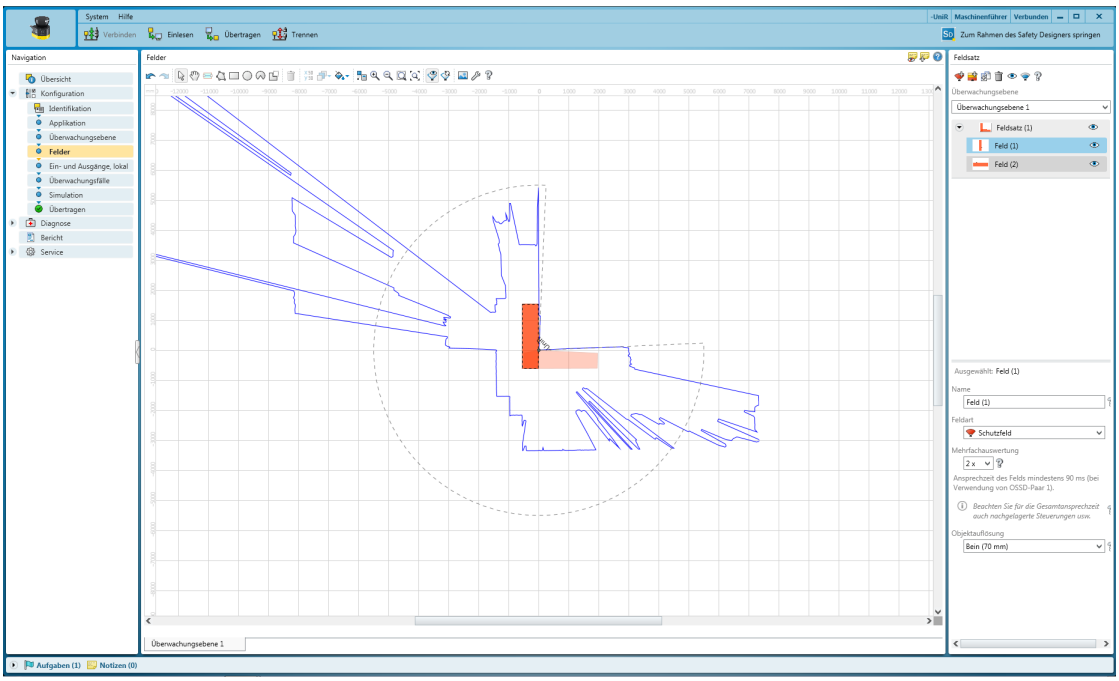


Figure A.5: Configured Safety Zone of Laser Scanner 2

A.4 Kuka iiwa

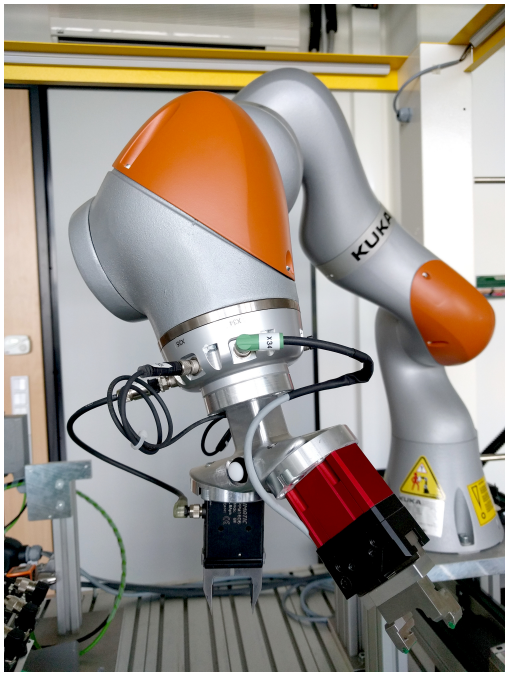


Figure A.6: *Kuka iiwa* with Mounted Tools

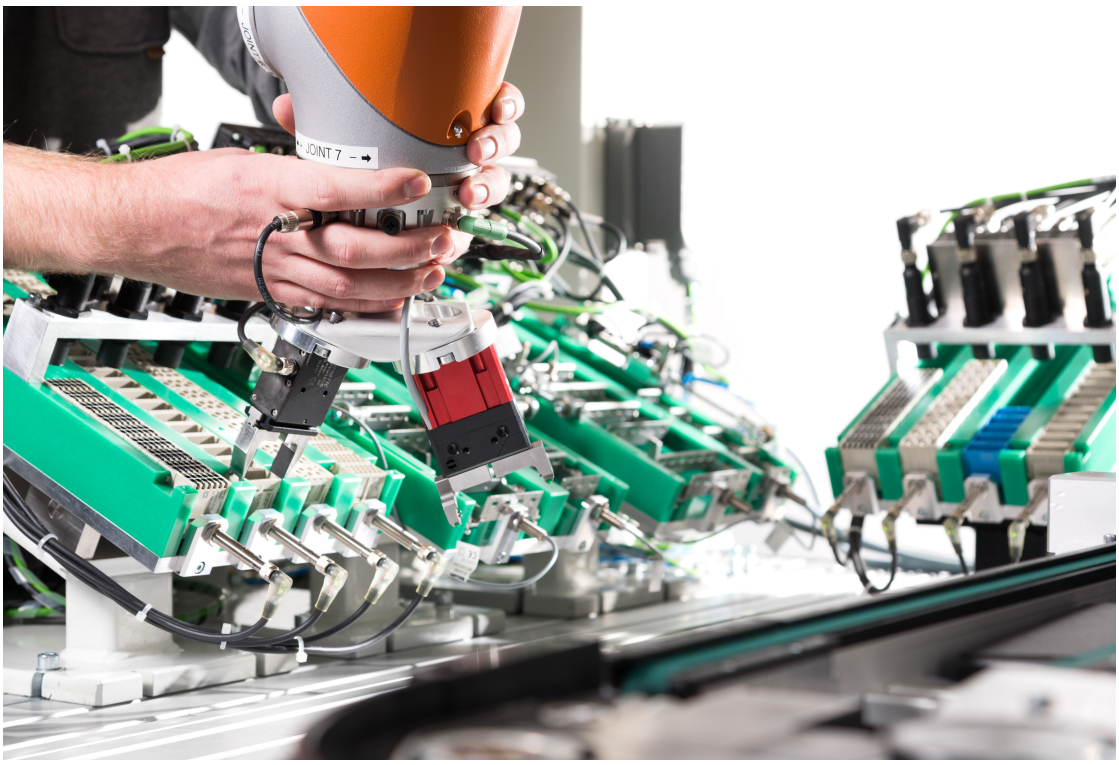


Figure A.7: *Kuka iiwa* Interactive Teaching

B BPMN Diagrams

B.1 Skill example 1

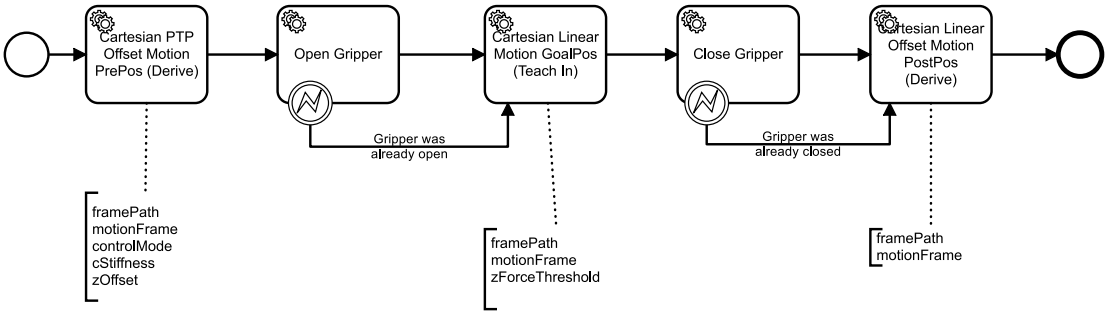


Figure B.1: Pick Skill

B.2 Skill example 2

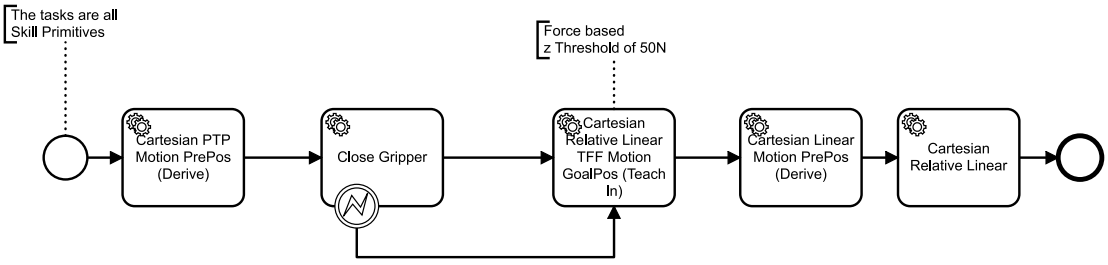


Figure B.2: Open Frame Skill

B.3 Task example 1

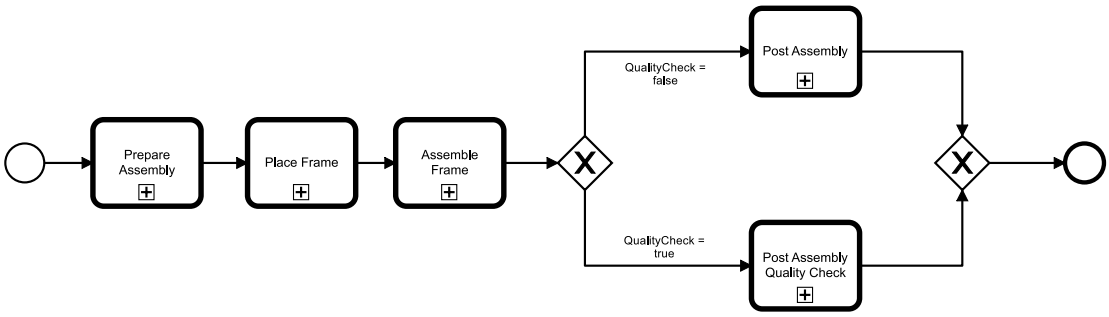


Figure B.3: Assembly Process

B.4 Task example 2

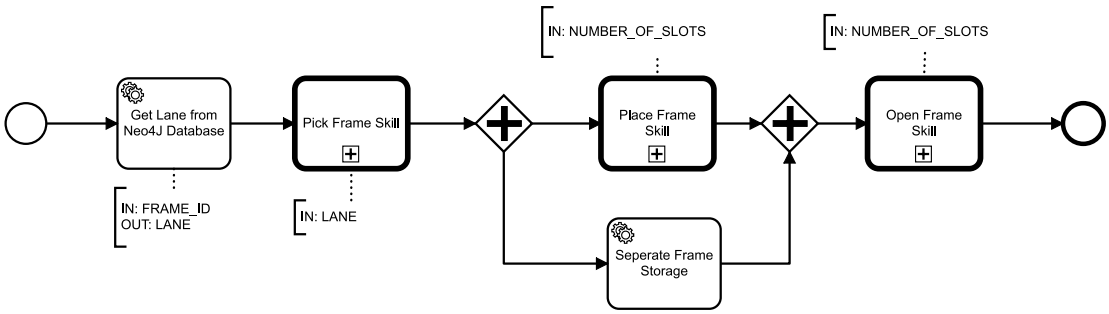


Figure B.4: Pick, Place and Open Frame

C Code

C.1 Repositories

C.1.1 AWAre Skill Implementation

Branch: master
Tag: masterthesisHO
URL:
`https://fleximon.cor-lab.de/git/fleximir.aware-plugin-robot-skills.git`

C.1.2 Kuka Skill Implementation

Branch: master
Tag: masterthesisHO
URL: `https://fleximon.cor-lab.de/git/fleximir.fleximir-kuka.git`

C.1.3 FlexiMiR Application Project

Branch: master
Tag: masterthesisHO
URL: `https://fleximon.cor-lab.de/git/fleximir.git`

C.2 Skill Callback Example

```
1  package skillCallbacks;
2
3  import javax.inject.Inject;
4
5  import rsb.patterns.DataCallback;
6  import rst.skillprimitives.CartesianMovementType.CartesianMovement;
7  import skillFramework.CartesianPTPMovementTFFSkill;
8  import skillFramework.MotionFrameMaster;
9  import skillFramework.OperationStateInformer;
10 import skillFramework.utilities.SkillConfigurator;
11
12 import com.kuka.roboticsAPI.deviceModel.LBR;
13 import com.kuka.roboticsAPI.deviceModel.LBRE1Redundancy;
14 import com.kuka.roboticsAPI.executionModel.ExecutionState;
15 import com.kuka.roboticsAPI.geometricModel.Frame;
16 import com.kuka.task.ITaskLogger;
17
18 public class CartesianPTPMovementCallback extends DataCallback<String,
       CartesianMovement> {
19
20   @Inject
21   private LBR lbr;
22   @Inject
23   private ITaskLogger logger;
24   @Inject
25   private CartesianPTPMovementTFFSkill cartesianPTPMovementTFFSkill;
26   @Inject
27   private MotionFrameMaster motionFrameMaster;
28   @Inject
29   private SkillConfigurator skillConfigurator;
30
31
32   @Inject
33   private  CartesianPTPMovementCallback(){}
34
35   @Override
36   public String invoke(CartesianMovement parameters) throws Exception {
37
38     logger.info("CartesianPTPMovementCallback␣was␣invoked.");
39     cartesianPTPMovementTFFSkill.clear();
40
41     cartesianPTPMovementTFFSkill = (CartesianPTPMovementTFFSkill)
           skillConfigurator.configurateControlMode(cartesianPTPMovementTFFSkill,
            parameters.getControlMode(),
42     parameters.getCartesianImpedianceControlMode(), parameters.
           getPositionControlMode());
43
44     cartesianPTPMovementTFFSkill = (CartesianPTPMovementTFFSkill)
           skillConfigurator.configurateForceTorqueConditions(
           cartesianPTPMovementTFFSkill, parameters.getForceTorqueMovement());
45
46     Frame frame = new Frame(parameters.getFrame().getX(), parameters.getFrame
           ().getY(), parameters.getFrame().getZ(),
47     parameters.getFrame().getA(), parameters.getFrame().getB(), parameters.
           getFrame().getC());
48
49     LBRE1Redundancy newRedundancyInformation = new LBRE1Redundancy();
```

```
50      newRedundancyInformation.setStatus(parameters.getFrame().
            getStatusParameter());
51      newRedundancyInformation.setTurn(parameters.getFrame().getTurnParameter())
            ;
52      newRedundancyInformation.setE1(parameters.getFrame().getE1Parameter());
53
54      frame.setRedundancyInformation(lbr, newRedundancyInformation);
55
56      cartesianPTPMovementTFFSkill.setRelAcceleration(parameters.getAcceleration
            ());
57      cartesianPTPMovementTFFSkill.setRelVelocity(parameters.getVelocity());
58      cartesianPTPMovementTFFSkill.setFrame(frame);
59      cartesianPTPMovementTFFSkill.setMotionFrame(motionFrameMaster.
            getMotionFrame(parameters.getMotionFrame()));
60
61      logger.info("CartesianPTPMovementCallback: CartesianPTPMovementSkill was
            built.");
62      final ExecutionState result = cartesianPTPMovementTFFSkill.execute();
63      logger.info("Skill exectuion state: " + result);
64      return result.toString();
65    }
66
67    public void setOperationStateInformer(OperationStateInformer
          operationStateInformer) {
68      cartesianPTPMovementTFFSkill.setOperationStateInformer(
            operationStateInformer);
69    }
70
71  }
```

Listing C.1: Callback for a Cartesian PTP Movement

C.3 Skill Delegate Example

```
1  package aware.robot.skills.delegates;
2
3  import org.camunda.bpm.engine.delegate.BpmnError;
4  import org.camunda.bpm.engine.delegate.DelegateExecution;
5  import org.camunda.bpm.engine.delegate.ExecutionListener;
6  import org.camunda.bpm.engine.delegate.Expression;
7  import org.camunda.bpm.engine.delegate.JavaDelegate;
8  import org.slf4j.Logger;
9  import org.slf4j.LoggerFactory;
10
11 import aware.core.plugin.PluginLoader;
12 import aware.datamodel.robotics.kuka.iiwa.TaskFrame;
13 import aware.robot.skills.plugins.RobotPlugin;
14 import rst.skillprimitives.CartesianMovementType.CartesianMovement;
15 import rst.skillprimitives.ControlModeType.ControlMode;
16 import rst.skillprimitives.ForceTorqueMovementType.ForceTorqueMovement;
17 import rst.skillprimitives.FrameType.Frame;
18 import rst.skillprimitives.PositionControlModeType.PositionControlMode;
19
20
21 public class CartesianPTPMoveDelegate extends ForceTorqueDelegate implements
       JavaDelegate {
22   private static final Logger LOGGER = LoggerFactory.getLogger(
         CartesianPTPMoveDelegate.class);
23   private Expression x, y, z, a, b, c, status, turn, e1, framePath,
         relVelocity, relAcceleration;
24
25   private double relVelocityValue, relAccelerationValue;
26
27   @Override
28   public void execute(DelegateExecution execution) throws Exception {
29
30     RobotPlugin robotPlugin = (RobotPlugin) PluginLoader.getInstance().
         getPlugin("robotPlugin");
31     if(robotPlugin == null){
32       LOGGER.info("Plugin␣Instance␣could␣not␣be␣retrieved␣in␣Delegate!");
33       return;
34     }
35
36     LOGGER.info("\n\n␣␣...␣CartesianPTPMoveDelegate␣invoked␣by␣" + "
         processDefinitionId="
37     + execution.getProcessDefinitionId() + ",␣activtyId="
38     + execution.getCurrentActivityId() + ",␣activtyName='"
39     + execution.getCurrentActivityName() + "'␣\n\n");
40
41
42     ForceTorqueMovement forceTorqueMove = createForceTorqueMovement(execution)
         ;
43
44     if(relVelocity != null){
45       relVelocityValue = Double.parseDouble((String) relVelocity.getValue(
           execution));
46     } else { relVelocityValue = 0.25; }
47     if(relAcceleration != null){
48       relAccelerationValue = Double.parseDouble((String) relAcceleration.
           getValue(execution));
49     } else { relAccelerationValue = 0.25; }
50
```

```
51      CartesianMovement cartMove = null;
52
53      if(framePath != null){
54        String framePathString = framePath.getValue(execution).toString();
55        TaskFrame taskFrame = robotPlugin.getTaskFrameHierarchy().getTaskFrame(
              framePathString);
56
57        cartMove = CartesianMovement.newBuilder()
58        .setFrame(
59        Frame.newBuilder()
60        .setX(taskFrame.getX())
61        .setY(taskFrame.getY())
62        .setZ(taskFrame.getZ())
63        .setA(taskFrame.getA())
64        .setB(taskFrame.getB())
65        .setC(taskFrame.getC())
66        .setStatusParameter(taskFrame.getStatus())
67        .setTurnParameter(taskFrame.getTurn())
68        .setE1Parameter(taskFrame.getE1())
69        .build())
70        .buildPartial();
71
72      } else {
73
74        cartMove = CartesianMovement.newBuilder()
75        .setFrame(
76        Frame.newBuilder()
77        .setX(Double.parseDouble((String) x.getValue(execution)))
78        .setY(Double.parseDouble((String) y.getValue(execution)))
79        .setZ(Double.parseDouble((String) z.getValue(execution)))
80        .setA(Double.parseDouble((String) a.getValue(execution)))
81        .setB(Double.parseDouble((String) b.getValue(execution)))
82        .setC(Double.parseDouble((String) c.getValue(execution)))
83        .setStatusParameter(Integer.parseInt((String) status.getValue(execution)
              ))
84        .setTurnParameter(Integer.parseInt((String) turn.getValue(execution)))
85        .setE1Parameter(Double.parseDouble((String) e1.getValue(execution)))
86        .build())
87        .buildPartial();
88      }
89
90      if(controlMode != null){
91        String controlModeString = (String) controlMode.getValue(execution);
92        if(controlModeString.equals("Impedance")){
93          cartMove = cartMove.toBuilder()
94          .setCartesianImpedianceControlMode(createCartesianImpedanceControlMode
                (execution))
95          .setControlMode(ControlMode.newBuilder()
96          .setControlModeOption(ControlMode.ControlModeOption.
                CARTESIANIMPEDIANCECONTROLMODE).build())
97          .buildPartial();
98        } else if(controlModeString.equals("Position")){
99          cartMove = cartMove.toBuilder()
100         .setControlMode(ControlMode.newBuilder()
101         .setControlModeOption(ControlMode.ControlModeOption.
                POSITIONCONTROLMODE).build())
102         .setPositionControlMode(PositionControlMode.newBuilder().build())
103         .buildPartial();
104       }
105     } else {
106       cartMove = cartMove.toBuilder()
```

```
107         .setControlMode(ControlMode.newBuilder()
108         .setControlModeOption(ControlMode.ControlModeOption.POSITIONCONTROLMODE)
                .build())
109         .setPositionControlMode(PositionControlMode.newBuilder().build())
110         .buildPartial();
111       }
112
113       cartMove = cartMove.toBuilder()
114       .setAcceleration(relAccelerationValue)
115       .setVelocity(relVelocityValue)
116       .setMotionFrame(getMotionFrame(execution))
117       .setForceTorqueMovement(forceTorqueMove)
118       .build();
119
120       LOGGER.info("CartesianMovement␣Type␣built");
121
122       if(cartMove != null) {
123         LOGGER.info("CartesianMovement␣Type␣built,␣sending␣command␣to␣robot");
124         robotPlugin.makeRemoteCall("cartesianPTPMove", cartMove, 120);
125       } else {
126         throw new BpmnError("CartesianMovement␣was␣not␣built␣correctly.");
127       }
128
129       clearLocalVariables();
130       clearGlobalVariables();
131     }
132
133     private void clearLocalVariables() {
134       x = null;
135       y = null;
136       z = null;
137       a = null;
138       b = null;
139       c = null;
140       status = null;
141       turn = null;
142       e1 = null;
143       framePath  = null;
144       relVelocity = null;
145       relAcceleration  = null;
146       relVelocityValue = 0.0;
147       relAccelerationValue = 0.0;
148     }
149
150   }
```

Listing C.2: Delegate for a Cartesian PTP Movement

C.4 Skill Condition Example

```java
package aware.robot.skills.conditions;

import org.camunda.bpm.engine.delegate.BpmnError;
import org.camunda.bpm.engine.delegate.DelegateExecution;
import org.camunda.bpm.engine.delegate.ExecutionListener;
import org.camunda.bpm.engine.delegate.Expression;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import aware.core.plugin.PluginLoader;
import aware.robot.skills.plugins.RobotPlugin;

public class GripperClosed implements ExecutionListener {

private static final Logger LOGGER = LoggerFactory.getLogger(GripperOpen.class
    );
private Expression gripperName;

@Override
public void notify(DelegateExecution execution) throws Exception {
    RobotPlugin robotPlugin = (RobotPlugin) PluginLoader.getInstance().
        getPlugin("robotPlugin");
    if(robotPlugin == null){
      LOGGER.info("Plugin Instance could not be retrieved in Delegate!");
      throw new BpmnError("Plugin Instance could not be retrieved");
    }

    boolean gripperClosed = robotPlugin.makeRemoteCallBoolean("gripperClosed",
         (String) gripperName.getValue(execution));

    if(gripperClosed){
      LOGGER.info("[GripperClosed Condition]: Gripper was closed - Condition
          fulfilled!");
      return;
    }

    throw new BpmnError("GripperWasNotClosed");
  }

}
```

Listing C.3: Gripper Closed Condition

C.5 Teach Skill

```
1  package skillFramework;
2
3  import static com.kuka.roboticsAPI.motionModel.HRCMotions.handGuiding;
4  import javax.inject.Inject;
5  import skillFramework.SkillExecutionState.SkillExecutionStateValue;
6  import com.kuka.roboticsAPI.controllerModel.sunrise.ResumeMode;
7  import com.kuka.roboticsAPI.controllerModel.sunrise.SunriseSafetyState;
8  import com.kuka.roboticsAPI.executionModel.ExecutionState;
9
10 public class TeachFrameSkill extends SkillPrimitive{
11
12   @Inject
13   private TeachFrameSkill() {}
14
15   @Override
16   public void clear() {}
17
18   @Override
19   public void prepareMotion() {}
20
21   @Override
22   public ExecutionState execute() throws Exception {
23     appControl.registerMoveAsyncErrorHandler(errorHandler);
24     appControl.setApplicationOverride(0.1);
25     logger.info("Starting␣TeachFrameSkill␣execution...");
26     SkillExecutionState.getInstance().setState(SkillExecutionStateValue.ACTIVE
           );
27     lbr.setESMState("2");
28     logger.info("ESM␣State␣set␣to␣2.");
29     logger.info("Waiting␣for␣the␣handguiding␣switch␣to␣be␣activated...");
30     while(lbr.getSafetyState().getEnablingDeviceState() != SunriseSafetyState.
           EnablingDeviceState.HANDGUIDING){
31       try {
32         Thread.sleep(100);
33       } catch (InterruptedException e) {
34         e.printStackTrace();
35       }
36     }
37     logger.info("Handguidswitch␣enabled␣trying␣to␣resume␣execution␣
           automatically.");
38     this.controller.getExecutionService().resumeExecution(ResumeMode.OnPath);
39     lbr.move(handGuiding());
40     this.logger.info("Handguiding␣finished.");
41     this.lbr.setESMState("1");
42     SkillExecutionState.getInstance().setState(SkillExecutionStateValue.IDLE);
43     operationStateInformer.publishDone();
44     operationStateInformer.publishIdle();
45
46     return ExecutionState.Finished;
47   }
48 }
```

Listing C.4: Teach Skill

C.6 RST Examples

```
1  syntax = "proto2";
2
3  package rst.skillprimitives;
4
5  import "rst/skillprimitives/Frame.proto";
6  import "rst/skillprimitives/CartesianImpedanceControlMode.proto"↩
       ;
7  import "rst/skillprimitives/PositionControlMode.proto";
8  import "rst/skillprimitives/ForceTorqueMovement.proto";
9  import "rst/skillprimitives/ControlMode.proto";
10
11 option java_outer_classname = "CartesianMovementType";
12
13 /**
14 * Contains the all information to specify a cartesian movement.
15 * Can also be used for relative movements.
16 *
17 * @author Hendrik Oestreich <hoestreich@techfak.uni-bielefeld.de↩
       >
18 */
19 // @create_collection
20 message CartesianMovement {
21
22   /**
23   * Goal Position of the movement.
24   */
25   required Frame frame = 1;
26
27   /**
28   * Motion Frame identifier
29   */
30   required string motion_Frame = 2;
31
32   /**
33   * Maximum velocity for the movement.
34   */
35   // @unit(millimeter/sec)
36   required double velocity = 3;
37
38   /**
39   * Maximum acceleration for the movement.
40   */
41   // @unit(millimeter/sec^2)
42   required double acceleration = 4;
43
44   /**
45   * Desired ControlMode for the Movement.
46   */
```

```
47    optional ControlMode control_mode = 5;
48
49    /**
50    * PositionControlMode
51    */
52    optional PositionControlMode position_control_mode = 6;
53
54    /**
55    * CartesianImpedanceControlMode
56    */
57    optional CartesianImpedanceControlMode ↩
          cartesian_impediance_control_mode = 7;
58
59    /**
60    * Reference frame of the movement (will only be used for ↩
          relative linear movements).
61    */
62    optional Frame reference_frame = 8;
63
64    /**
65    * Force / Torque Thresholds to allow Task-Frame-Formalism like↩
           movements
66    */
67    optional ForceTorqueMovement force_torque_movement = 9;
68
69 }
```

Listing C.5: Cartesian Movement Datatype

C.7 Class Diagram

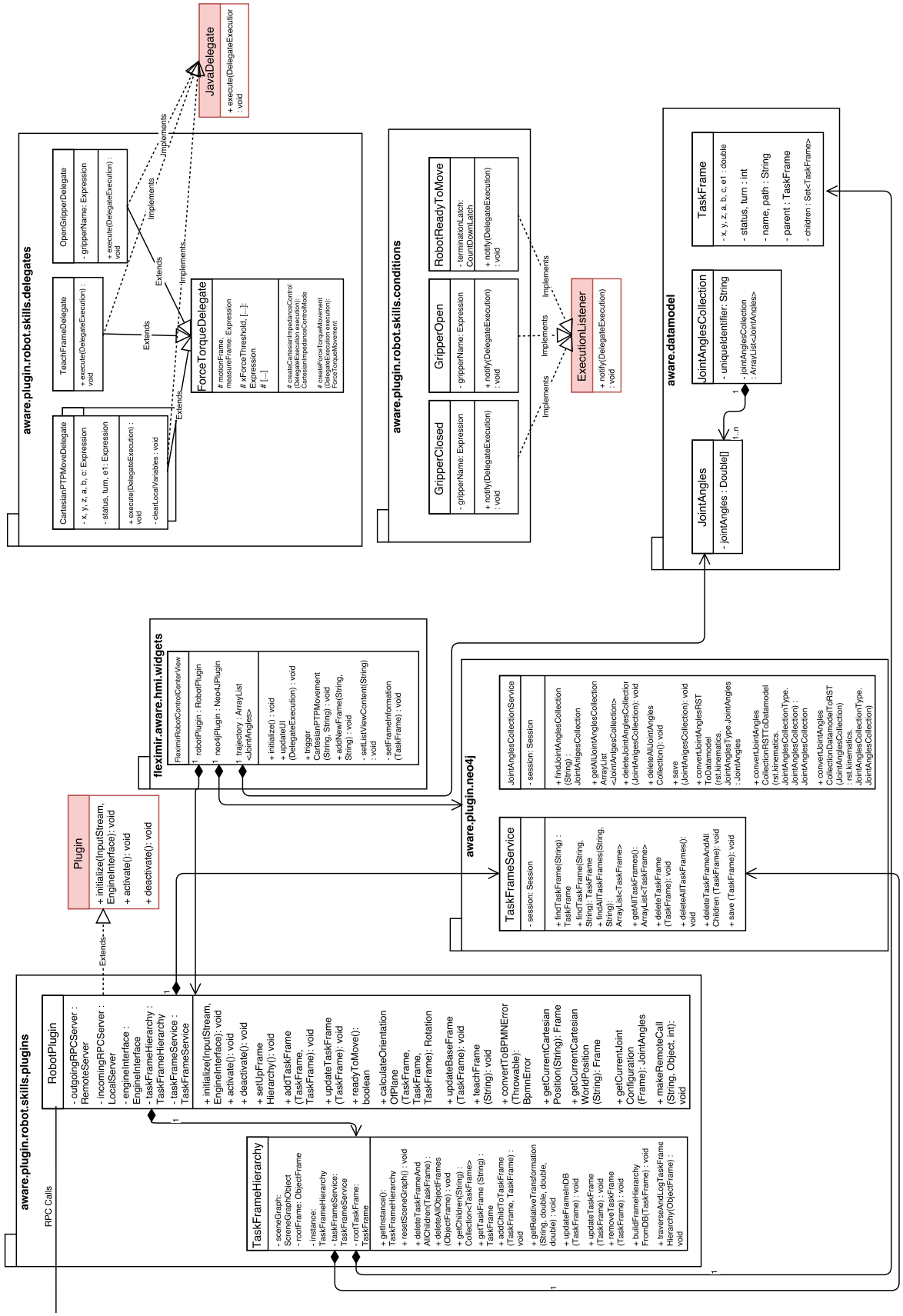


Figure C.1: Genric Part of the Implementation

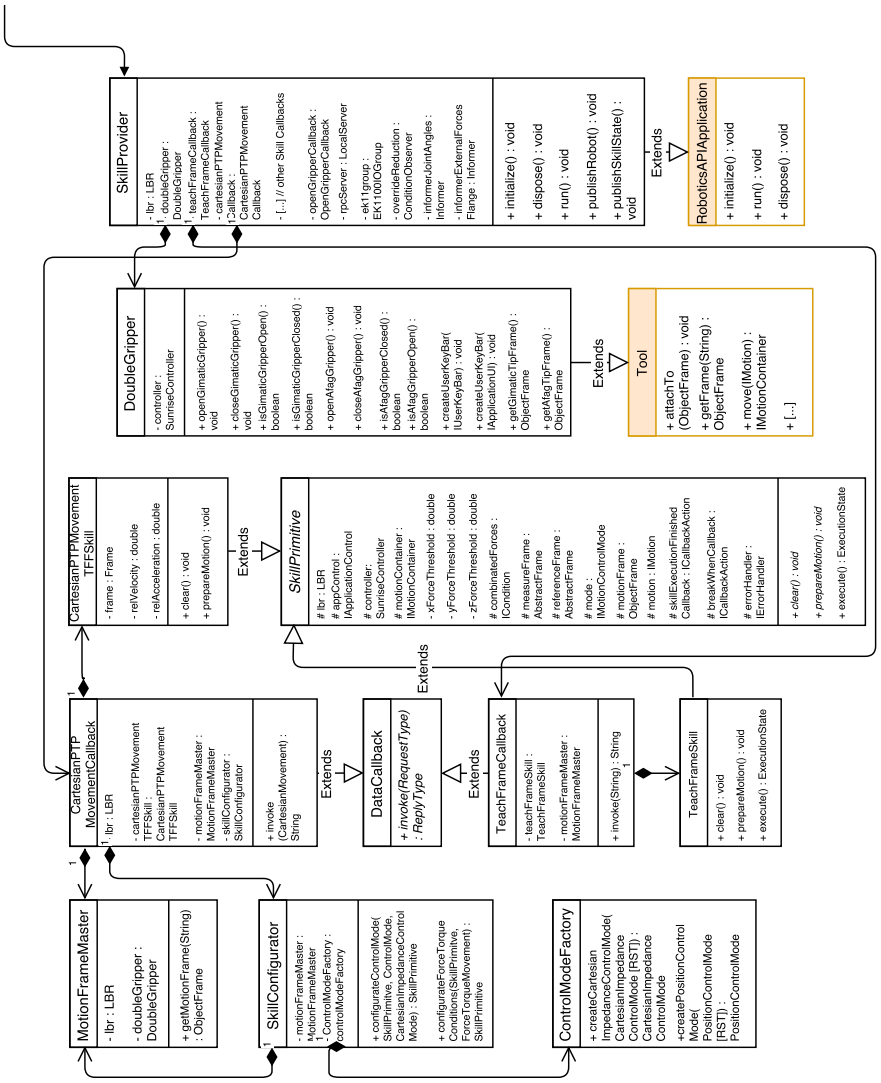


Figure C.2: *Kuka*-specific part of the Implementation

References

Anderl, P. D.-I. R., Picard, A., Wang, Y., Dosch, S., Klee, B., Bauer, J. & Metten, D. B. (2015). *Guideline Industrie 4.0* (Tech. Rep.). Frankfurt am Main: VDMA Verlag GmbH.

Archibald, C. C. (1995). *A computational model for skills-oriented robot programming* (Unpublished doctoral dissertation). University of Ottawa.

Archibald, C. C. & Petriu, E. (1993). Skills-Oriented Robot Programming. In *Ias 3 - intelligent autonomous systems* (pp. 104–113). Pittsburgh, Pennsylvania: IOS Press.

Bänziger, T., Kunz, A. & Wegener, K. (2017). A Library of Skills and Behaviors for Smart Mobile Assistant Robots in Automotive Assembly Lines. In *Hri '17* (pp. 77–78). Vienna, Austria.

Bélanger-Barrette, M. (2016). *COLLABORATIVE ROBOTS RISK ASSESSMENT , AN INTRODUCTION UPDATED* (1.1 ed.). Quebec City, Canada: Robotiq. Retrieved from `http://robotiq.com/wp-content/uploads/2015/08/eBook-V2-VF-Risk-Assessment-Collaborative-Robots.pdf` (Accessed: 2017-05-19 20:33)

Björkelund, A., Edström, L., Haage, M., Malec, J., Nilsson, K., Nugues, P., … Bruyninckx, H. (2011). On the integration of skilled robot motions for productivity in manufacturing. In *Proceedings - 2011 ieee international symposium on assembly and manufacturing, isam 2011.*

Blumenthal, S., Bruyninckx, H., Nowak, W. & Prassler, E. (2013). A scene graph based shared 3D world model for robotic applications. In *Icra - proceedings - ieee international conference on robotics and automation* (pp. 453–460). Karlsruhe, Germany.

Bøgh, S., Hvilshoøj, M., Kristiansen, M. & Madsen, O. (2012, jul). Identifying and evaluating suitable tasks for autonomous industrial mobile manipulators (AIMM). *International Journal of Advanced Manufacturing Technology*, *61*(5-8), 713–726.

Bøgh, S., Nielsen, O. S., Pedersen, M. R., Krüger, V. & Madsen, O. (2012). Does your Robot have Skills? *Proceedings of the 43rd International Symposium on Robotics*, 6.

Bruyninckx, H. & De Schutter, J. (1996). Specification of force-controlled actions in the "Task frame formalism" - A synthesis. *IEEE Transactions on Robotics and Automation*, *12*(4), 581–589.

Butting, A., Rumpe, B., Schulze, C., Thomas, U. & Wortmann, A. (2015). Modeling Reusable, Platform-Independent Robot Assembly Processes. In *Workshop on domain specific languages (dslrob15).*

Ekvall, S., Aarno, D. & Kragic, D. (2006). Task learning using graphical programming and human demonstrations. In *Proceedings - ieee international workshop on robot and human interactive communication* (pp. 398–403).

Europäsches Paralament / Europäischer Rat. (2006). *Maschinenrichtlinie.* Retrieved from `www.maschinenrichtlinie.de` (Accessed: 2017-05-19 20:33)

Fachbereich Holz und Metall der DGUV. (2017). *Kollaborierende Robotersysteme* (Tech. Rep.). Mainz, Germany: Deutsche Gesetzliche Unfallversicherung - Fachbereich Holz und Metall. Retrieved from `http://www.dguv.de/medien/fb-holzundmetall/publikationen-dokumente/infoblaetter/infobl{_}deutsch/080{_}roboter.pdf` (Accessed: 2017-05-19 20:33)

Hasegawa, T., Suehiro, T. & Takase, K. (1992). A Model-Based Manipulation System with Skill-Based Execution. *IEEE Transactions on Robotics and Automation*, *8*(5), 535–544.

Haun, M. (2007). *Handbuch Robotik*. Berlin, Heidelberg, New York: Springer.

Kagermann, H., Wahlster, W. & Helbig, J. (2013). *Recommendations for implementing the strategic initiative INDUSTRIE 4.0* (Tech. Rep.). Frankfurt/Main.

Kröger, T., Finkemeyer, B., Thomas, U. & Wahl, F. M. (2004). Compliant Motion Programming: The Task Frame Formalism Revisited. *Mechatronics and Robotics*, 1029–1034.

Kröger, T., Finkemeyer, B. & Wahl, F. M. (2004). A task frame formalism for practical implementations. In *Icra - ieee international conference on robotics and automation 2004* (Vol. 5, pp. 5218–5223). New Orleans, LA.

KUKA. (2016). *KUKA Sunrise.OS 1.11, KUKA Sunrise.Workbench 1.11, Bedien- und Programmieranleitung für Systemintegratoren* (KUKA Sunrise.OS 1.11 SI V1 ed.; Tech. Rep.). KUKA Roboter GmbH.

Lozano-Perez, T. (1983). Robot programming. In *Proceedings of the ieee* (Vol. 71, pp. 821–841).

Mason, M. T. (1981). Compliance and Force Control for Computer Controlled Manipulators. *IEEE Transactions on Systems, Man and Cybernetics*, *11*(6), 418–432.

Matthias, B. (2017). Human-robot collaboration - Industrial Applications and Open Challenges. In *Dagstuhl seminar on "computer-assisted engineering for robotics and autonomous systems".* Dagstuhl, Germany: ABB.

Morrow, J. (1997). *Sensorimotor primitives for robotic assembly skills* (Doctor of Philosophy, Carnegie Mellon University).

Morrow, J. & Khosla, P. (1997). Manipulation task primitives for composing robot skills. *Proceedings of International Conference on Robotics and Automation*, *4*(April), 3354–3359.

Naumann, M., Wegener, K. & Schraft, R. D. (2007). Control Architecture for Robot Cells to Enable Plug ' n ' Produce. In Fraunhofer IPA (Ed.), *Ieee international conference on robotics and automation* (pp. 287–292). Roma, Italy.

Pedersen, M. R., Nalpantidis, L., Andersen, R. S., Schou, C., Bøgh, S., Kröger, V. & Madsen, O. (2016). Robot skills for manufacturing: From concept to industrial deployment. *Robotics and Computer-Integrated Manufacturing*, *37*, 282–291.

Pedersen, M. R., Nalpantidis, L., Bobick, A. & Krüger, V. (2013). On the Integration of Hardware-Abstracted Robot Skills for use in Industrial Scenarios. *2nd International IROS Workshop on Cognitive Robotics Systems: Replicating Human Actions and Activities*.

Pfrommer, J., Stogl, D., Aleksandrov, K., Escaida Navarro, S., Hein, B. & Beyerer, J. (2015). Plug & produce by modelling skills and service-oriented orchestration of reconfigurable manufacturing systems. *Automatisierungstechnik*, *63*(10), 790–800.

Robotiq. (2016). *ISO / TS 15066 Explained* (Tech. Rep.). Quebec City, Canada. Retrieved from `http://robotiq.com/wp-content/uploads/2016/05/ebook-ISOTS15066-Explained.pdf` (Accessed: 2017-05-19 20:33)

Schou, C., Andersen, R. S., Chrysostomou, D., Bøgh, S. & Madsen, O. (2016). Skill Based Instruction of Collaborative Robots in Industrial Settings. *Robotics and Computer-Integrated Manufacturing*.

Schou, C., Damgaard, J. S., Bogh, S. & Madsen, O. (2013). Human-robot interface for instructing industrial tasks using kinesthetic teaching. In *Isr - 44th international symposium on robotics.*

Schutter, J. D. & Brussel, H. V. (1988). Compliant Robot Motion I. A Formalism for Specifying Compliant Motion Tasks. *The International Journal of Robotics Research*, *7*(4), 3–17.

Skoglund, A. (2009). *Programming by Demonstration of Robot Manipulators* (Tech. Rep.). Örebro: Örebro University.

Steinmetz, F. & Weitschat, R. (2016). Skill Parametrization Approaches and Skill Architecture for Human-Robot Interaction. In *Case - ieee international conference on automation science and engineering* (pp. 280–285). Fort Worth, TX, USA.

Weidauer, I., Kubus, D. & Wahl, F. M. (2014). A hierarchical extension of manipulation primitives and its integration into a robot control architecture. In *Icra - ieee international conference on robotics and automation* (pp. 5401–5407). Hong Kong, China.

Zeiß, S. (2014). *Manipulation Skill for Robotic Assembly* (Master Thesis). Technische Universität Darmstadt.