Implementation of a Bridge for KUKA Robots Control using a REST API

Bachelor Thesis

Bachelor of Science in Computer Science

Submitted by
Marco B. Gabriel
Matriculation Number: 20-209-656
Fribourg, 2023

Thesis supervisor:
Prof. Dr. Hans-Georg Fill

Digitalization and Information Systems Group
Department of Informatics
University of Fribourg (Switzerland)

Abstract

Robots are an important reason for increased productivity in many industrial sectors. This development is part of Industry 4.0, which aims to combine information systems such as Artificial Intelligence with robots to increase performance. However, as robot manufacturers like KUKA often use proprietary communication protocols and interfaces, compatibility problems arise between information systems and robots. Although there exist various approaches to control KUKA robots, none of them solve the problem universally or without caveats. Consequently, a REST API for KUKA robots is a valuable addition as it provides a human- and machine-friendly, client-independent interface. In this project, we design and implement such a solution using the Design Science Research Methodology for Information Systems Research (DSRM). We evaluate our implementation using the OrangeApps Education Robot System (ERS). The results show that our product provides similar or greater generalisability than existing approaches, depending on the respective solution, while solely relying on a KUKA Robot Controller (KRC) supporting communication over files. Our implementation provides the KUKA Robot Language's key functions over a REST API. However, it could be improved by introducing error handling or increasing the reliability and number of provided functions.

Keywords

Robots, KUKA Robots, Bridge, REST API, OrangeApps ERS, Education Robot System, OrangeApps Education Robot System

Table of Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Robots are an important reason for an increased productivity in many industrial sectors [Schierl, 2017]. Currently, installations of robots are at an all time high and expected to continue growing in the near future [International Federation of Robotics, 2022].

This development is part of the concept of Industry 4.0, which aims to, among other things, combine information systems such as Artificial Intelligence with collaborating robots [Benotsmane *et al.*, 2018]. Such collaboration allows to distribute the intelligence of robots to achieve the desired performance [Day, 2018].

1.1. Problem Statement

But there is an obstacle to overcome before achieving the integration of various information systems with robots. Most manufacturers of robots use proprietary communication protocols and interfaces, making it difficult for robots from various producers to collaborate [Arnarson *et al.*, 2020]. This translates to compatibility problems between information systems and robots. Thus, the selection options of control systems for robots is often severely limited.

1.2. Existing Solutions

To find existing solutions to the identified problem, we utilise the search engines Google and Google Scholar. In the latter, we search for the term "kuka robot controller krl". Combining the terms "kuka", "rest" and "api" also yields a considerable solution. Furthermore, we consider a standard Google search for the keywords "kuka" and "api" to not restrict ourselves on formally published solutions. Finally, we investigate other solutions mentioned in relevant projects that we found.

Although there are some initiatives to homogenise the use of protocols, languages and interfaces in robotics, the problem is far from solved. Amongst others, the solutions at hand are:

- Fast Research Interface [Schreiber *et al.*, 2010]
- JOpenShowVar [Sanfilippo *et al.*, 2015]
- Robot Operating System (ROS) [Open Source Robotics Foundation, n.d.a]

- OPC Unified Architecture (OPC UA) [OPC Foundation, n.d.]

Regrettably, none of them solve the problem universally and completely. For example, even the popular OPC UA is not supported natively by KUKA controllers. Other solutions, like the Fast Research Interface, are limited to a special kind of robot.

1.3. Research Gap

The programming language used on KUKA robots is the proprietary KUKA Robot Language (KRL) [KUKA, 2022], which does not include all features of modern programming languages, like matrix operations or the usage of external libraries. KRL is tailored for industrial usage and does not natively support creating new functionalities for research purposes [Mühe *et al.*, 2010]. Only certain features are offered at an additional cost, as with the KUKA RSI [KUKA AG, 2023g].

Although there already exist open source alternatives, they often require model-specific drivers, which are not always readily available. Consequently, a model-agnostic solution would provide value to the field of robotics, as the same program could be used on different robots. This opens the possibility to test code on a small scale robot instead of relying on simulations before executing it on a big and expensive robot.

An important complement to the existing solutions can be made in the form of a REST API, as [Alam *et al.*, 2020] believe. A REST API provides a human- and machine-friendly interface and simplifies programming, for example with Swagger [Surwase, 2016]. The API can be used in combination with any programming language supporting HTTP Requests.

As there does not yet exist a straightforward system that allows controlling KUKA robots through a REST API, the goal of this thesis is to develop a REST API for KUKA Robots specifically, with the OrangeApps Education Robot System (ERS) [OrangeApps, n.d.] as a concrete use case. The system should be robot-agnostic, supporting various KUKA robot models. Furthermore, it should support movement commands to be specified in a coordinate system, not just axis movements.

1.4. Research Questions

To guide the development process, we state the following research questions:

Research Question 1: What is the State of the Art?

It is researched, if a similar or same project was already done. So far, we are only aware that there are approaches to use REST APIs to control robots from different brands and approaches to use APIs or indirect REST APIs with KUKA robots. But no REST API with KUKA robots directly.

Research Question 2: What is the best Approach to implement a REST API bridging KUKA Robots?

First, we study how the KUKA Robot Language (KRL) works, to understand which features it offers that we need to bridge. Afterwards, we decide on the best approach to develop a REST API to provide the KRL functionality. This includes, whether we need

additional hardware or only software, and documenting every decision. Finally, we decide on how to test the APIs functionality.

Research Question 3: How is the REST API implemented and how does it perform?

This includes the documentation of the final implementation and the detailed description of its usage. The result of the evaluation, e.g. if the API works, and how extensively and robust it works, and whether it also works for other KUKA robots than the Education Robot System from OrangeApps.

1.5. Methodology

Answering the Research Questions is done using the Design Science Research Methodology for Information Systems Research (DSRM) from [Peffers *et al.*, 2007]. The DSRM consists of multiple phases which are outlined in the following paragraphs.

Problem identification and motivation
"Define the specific research problem and justify the value of a solution. [...] Resources required for this activity include knowledge of the state of the problem and the importance of its solution" [Peffers *et al.*, 2007].

We conduct research on similar previous work. This includes approaches using a REST API to bridge proprietary programming languages or APIs to control robots, but also approaches using non-REST APIs with a focus on KUKA robots. Such an example is [Alam *et al.*, 2020]. Most of the research is done using the Google Scholar search engine.

Define the objectives for a solution
"Infer the objectives of a solution from the problem definition and knowledge of what is possible and feasible" [Peffers *et al.*, 2007].

After studying how the proprietary robot controls, e.g. the KUKA robot controller (KRC) and the KUKA Robot Language (KRL) work, we decide on mainly qualitative requirements. These should roughly resemble the capabilities identified while studying the KRC.

Design and development
"Create the artifact. Such artifacts are potentially constructs, models, methods, or instantiations [...]. This activity includes determining the artifact's desired functionality and its architecture and then creating the actual artifact" [Peffers *et al.*, 2007].

We analyse different approaches for their quality and feasibility in regards to our objectives. Then we compare them to select the best alternative. We plan how to best implement and structure this solution and finally develop it according to the previously defined requirements.

Demonstration
"Demonstrate the use of the artifact to solve one or more instances of the problem" [Peffers *et al.*, 2007].

An ERS [OrangeApps, n.d.] is controlled via Bee-Up [Burzynski & Karagiannis, 2020], an ADOxx-based modelling tool able to exploit the REST API of the artifact. The concrete use-case is drawing a sketch on paper, as the tool-set for the ERS is rather limited.

Evaluation
"Observe and measure how well the artifact supports a solution to the problem. [...] At

the end of this activity the researchers can decide whether to iterate back to [Design and development] to try to improve the effectiveness of the artifact or to continue on to communication and leave further improvement to subsequent projects" [Peffers *et al.*, 2007].

Iteration to improve quality is not possible in the venue of this project. As such, it is just evaluated how well the product performs. We use Bee-Up to create sample tasks for the robot to subsequently evaluate the following metrics:

- Generalisability
- Ease of usage
- Movement functionalities (circles, lines, time, ...) by coordinates
- Error handling

The final implementation is tested with a program for functionality and evaluated according to the result. These findings are discussed and rated for possible future improvements.

Communication
"Communicate the problem and its importance, the artifact, its utility and novelty [...] and its effectiveness to [...] relevant audiences" [Peffers *et al.*, 2007].

This part is not covered in this thesis.

2

State of the Art in Programming Robots

It is researched, how robots are controlled using existing software and investigated, whether a similar or identical project already exists. So far, we are only aware that there are approaches to use REST APIs to control robots from different brands and approaches to use APIs or indirect REST APIs with KUKA robots. But no REST API with KUKA robots directly. The major projects are covered in the following sections.

2.1. KUKA Robot Controller (KRC)

KUKA provides the KUKA Robot Controller (KRC) to control its robots. This is essentially a machine running the Windows operating system with the robot interpreter which interprets the KUKA Robot Language (KRL) source code [KUKA AG, 2023h]. The computer is extended with a smartPAD, which is a hand-held tablet [KUKA AG, 2023a]. It provides the Human Machine Interface (HMI), allowing the user to either control a robot manually, teach movements or even write code [KUKA AG, 2023b]. This control structure is visualised in figure 2.1.

Fig. 2.1.: Architecture for controlling a KUKA Robot using the KRC.

The Education Robot System (ERS) not only represents a complete 6-axis jointed-arm robot, but can also be operated and programmed like a real KUKA robot [OrangeApps, n.d.]. Practically, the only difference between the ERS and a real robot is that the ERS is constructed using LEGO pieces, resulting in less accurate motions. To control it, a similar architecture as for real robots is used, as shown in figure 2.2. A KRL script is interpreted by a KRC which is run inside a KUKA.OfficeLite Virtual Machine. The ERS-Interface, a KUKA Options Package (KOP), handles the communication with the ERS via USB. Internally, the ERS is built on a LEGO SPIKE Prime Hub running MicroPython. This hub controls the motors of the robot.

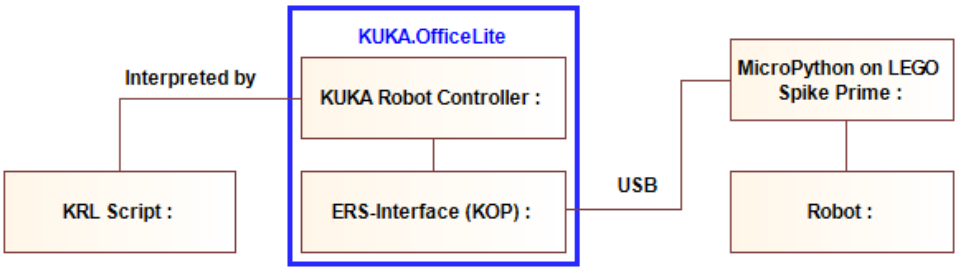

Fig. 2.2.: Controller Architecture of the OrangeApps Education Robot System (ERS). To control the ERS, the KUKA Robot Language (KRL) is used. A KRL Script is interpreted by the KUKA Robot Controller (KRC) in the KUKA.OfficeLite Virtual Machine. A KUKA Options Package (KOP) named ERS-Interface is necessary to communicate with the ERS via USB. Internally, the ERS is built on a LEGO SPIKE Prime Hub running MicroPython.

The KRL is an imperative programming language. In addition to typical programming statements such as variable assignments, conditionals, and loops, KRL provides robotic specific statements such as point-to-point or linear motions [Mühe *et al.*, 2010].

However, it does not include common features in modern programming languages, such as advanced mathematics like matrix operations, optimisation or filtering methods. Additionally, the use of external libraries is not natively supported. Hence, it is only of limited utility for research purposes [Sanfilippo *et al.*, 2015].

2.2. KUKA Robot Sensor Interface (RSI)

The KUKA RSI allows the robot controller to receive and process data from external sensors. The data can be ingested via Ethernet, among other options. The resulting system architecture is modelled in figure 2.3. An example implementation with RSI is shown in [Zuther, 2019]. However, this interface is far from resembling a REST API and comes with a significant financial cost due to its proprietary nature [KUKA AG, 2023g].

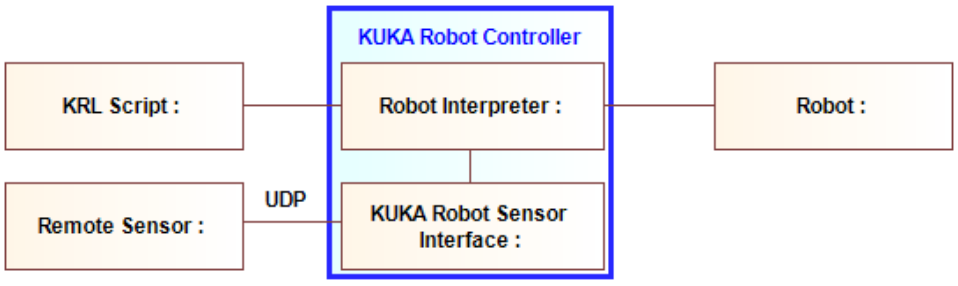

Fig. 2.3.: Architecture for controlling a KUKA Robot using the KUKA Robot Sensor Interface (RSI) to extend the KRCs functionality by allowing it to communicate with external sensors over Ethernet.

2.3. Fast Research Interface (FRI)

The Fast Research Interface (FRI) enables applications controlling a KUKA lightweight robot (LWR) to be hosted on all operating systems. The FRI is a low latency interface, as it allows up to 1000 messages per second. Its design is shown in figure 2.4.

The features of the KRL, such as linear and circular motions, are exposed by the FRI over a UDP connection [Schreiber *et al.*, 2010].

The FRI is provided by KUKA as an add-on software and therefore closely integrated into the KUKA ecosystem [KUKA, 2019b]. Previously, it worked with KSS [KUKA, 2011b], but this version is discontinued [KUKA AG, 2023e]. Currently, it is used in combination with the KUKA Sunrise environment [KUKA AG, 2023c] which has to be purchased from KUKA.

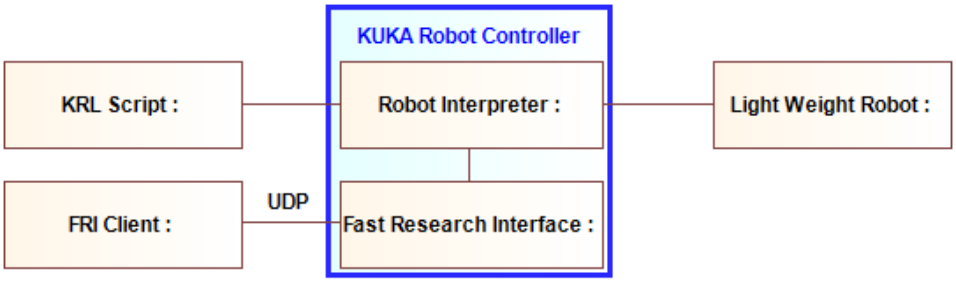

Fig. 2.4.: Architecture for controlling a KUKA Robot using the Fast Research Interface (FRI) to extend the KRCs functionality by allowing it to communicate with remote clients over UDP.

2.4. JOpenShowVar

JOpenShowVar is a Java open-source cross-platform communication interface to KUKA industrial robots.

This interface allows for the interaction with the real-time control process of the robot and makes it possible to perform several operations, such as selection or cancellation of a specific program, error and fault detection, renaming program files, saving programs, resetting I/O drivers, reading variables, and writing variables. To achieve this, it incorporates KUKAVARPROXY, a multiclient server running on the KRC, which in turn relies on the Crosscomm interface to interact with the KRL runtime [Sanfilippo *et al.*, 2015].

To convert these functionalities to actual motions, JOpenShowVar employs an actuator program in KRL. In the simplest use case, this is essentially a loop that keeps executing a motion according to a global variable. Hence, JOpenShowVar requires a custom KRL program depending on the use case. The complete architecture of JOpenShowVar is shown in figure 2.5.

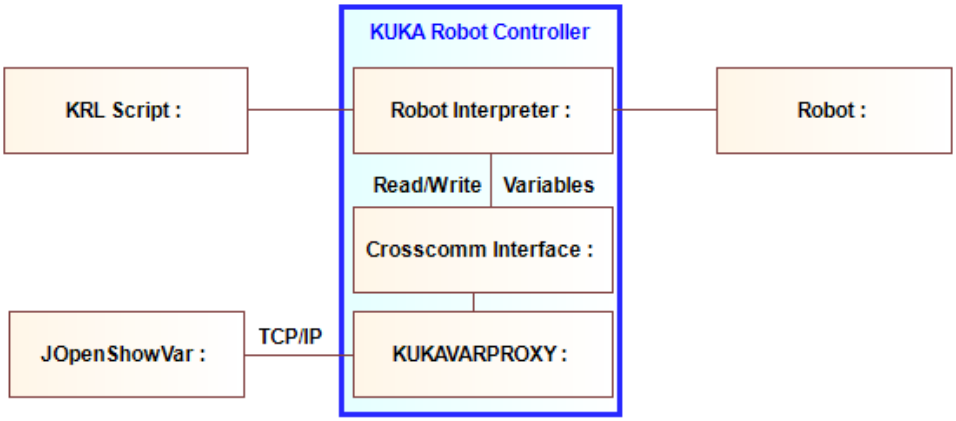

Fig. 2.5.: Architecture for controlling a KUKA Robot using JOpenShowVar to read and write variables of the Robot Interpreter.

2.5. OPC UA

The OPC UA protocol is mainly used for information sharing between systems by reading and writing variables to and from a robot. [KUKA , 2019]

It is used by [Arnarson *et al.*, 2020] to achieve collaboration and communication between robots from different manufacturers. The KUKA robots are connected to the OPC UA server using the KUKA RSI and KUKAVARPROXY-OpenShowVar. The RSI is used for real-time control of the robot, whereas OpenShowVar is used to collect additional data from the robot. This setup is visualised in figure 2.6.

However, there also exists an option package provided by KUKA, but it can not simply be ordered anymore [KUKA AG, 2023f].

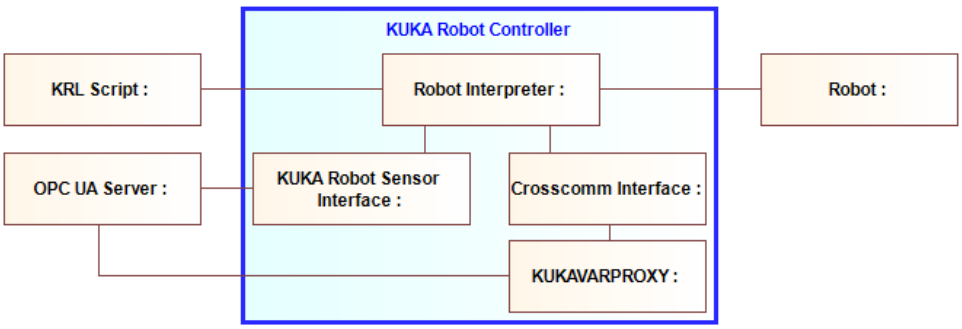

Fig. 2.6.: Architecture for controlling a KUKA Robot using OPC UA.

2.6. Robot Operating System (ROS)

The Robot Operating System (ROS) is a widely used framework for personal and industrial robots. ROS contains software libraries and tools for developing applications involving robots [Open Source Robotics Foundation, n.d.a]. Despite the name, ROS is not actually an operating system. Essentially, ROS provides a message-passing system

called “plumbing”. Since communication is often a key part in software systems that interact with hardware, ROS can be very helpful [Open Source Robotics Foundation, n.d.b].

Concerning the available tools within ROS, there is a package in development for KUKA robots [Hoorn, n.d.]. However, support for actual robot models is still experimental and particularly parameters for inverse kinematics (IK) only exist for a few models [Jülg, n.d.].

Alternatively, the KRC can be used as the robot driver. It can be connected to ROS by various means, such as the RSI and KUKAVARPROXY as shown by [Arbo *et al.*, 2020], the FRI as demonstrated by [Chatzilygeroudis *et al.*, 2019], or by using the KUKA Sunrise environment [Mokaram *et al.*, 2017]. Figure 2.7 depicts how ROS is used with a robot. The internals of ROS such as the "plumbing" are omitted.

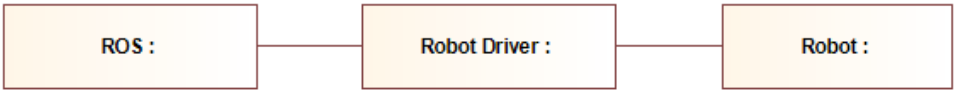

Fig. 2.7.: Architecture for controlling a KUKA Robot using ROS. The Robot Driver can be a native implementation, or a KRC connected to ROS via one of the discussed approaches.

2.7. Robotics API

Robotics API is an object-oriented framework for industrial robots, implemented in Java [Angerer *et al.*, 2013]. It is a front end to a Robot Control Core (RCC) which physically controls a robot. This relation is shown in figure 2.8. While the Robotics API is RCC independent, it comes bundled with an RCC implementation based on the Kinematics and Dynamics Library (KDL) from the Orocos framework developed by Bruyninckx [Bruyninckx, 2001].

The Orocos framework does not provide concrete kinematic models of robots, but does handle the calculation of inverse kinematics for arbitrary kinematic chains. The Robotics API does however provide models for some robots, even multiple of the KUKA KR series, but the OrangeApps ERS and many others are not covered.

There are 2 Versions of the Robotics API, Version 0.9 and Version 2. Unfortunately, Version 2 is only sparsely documented. The API is licensed under the Mozilla Public License (MPL) v2 which allows for use in combination with proprietary code.

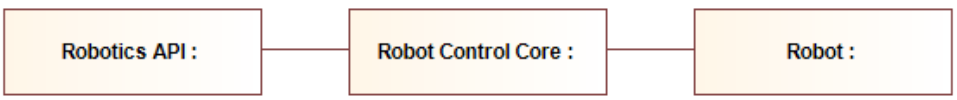

Fig. 2.8.: Architecture for controlling a KUKA Robot using the Robotics API.

2.8. REST API

As an exemplary for a REST API controlling a robot, we consider the Dobot arm from OMiPOB [Karagiannis & Muck, 2017]. In this project, a REST API is used to steer a robot arm through the open-dobot firmware [maxosprojects, n.d.]. In OMiPOB, GET, PUT and POST requests onto various resources (URIs) are used for control. The destination positions for movement commands are specified in Cartesian coordinates since the inverse kinematics are handled by the firmware. This allows the robot arm to be controlled by tools such as Bee-Up [Burzynski & Karagiannis, 2020].

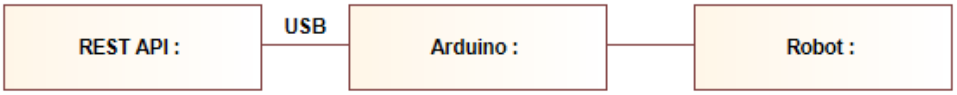

Fig. 2.9.: Architecture for controlling a Robot using a REST API.

3 A REST API for Accessing KUKA Robots

The requirements for our solution define the external structure of the implementation. As visualised in figure 3.1, on the client side, the bridge must expose a REST API. On the back end, we need an interface to control a KUKA robot. In the following sections, we research and decide on how to fill this black box to bridge REST requests to robot movements.

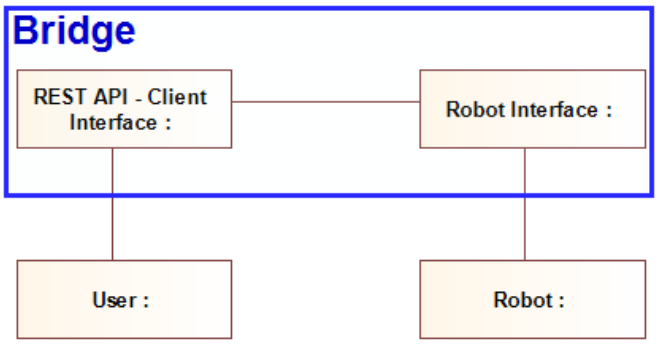

Fig. 3.1.: Communication Diagram of the design skeleton of a Bridge for KUKA robots. The system has a REST interface exposed to the client and an undefined channel to the robot. The bridge might involve internal communication.

3.1. Functional Scope

The most important decision to take is which methods to provide. Choosing the wrong or too few may lead to the implementation being of limited utility, as it may not support the functionality expected from a robot. The inherent purpose of a software bridge is to retain functionality while altering the interface.

The KUKA Robot Language contains three motion commands, PTP, LIN and CIRC, which can be combined into Splines [KUKA, 2022]. Together with logic elements and additional parameters to configure motions, this allows to use a robot to its full capacity.

Hence we suppose, that the REST API needs to provide the basic motion functions PTP, LIN and CIRC. More elaborate elements such as logic, or interaction with inputs can be

outsourced to the client. More complex motions like Splines can be achieved by chaining multiple movement commands on the client-side. For time-critical motions this approach is sub optimal, as the commands are sent in separate requests, the bridge might introduce delays between them. To mitigate this limitation, the REST API shall provide a method to submit multiple commands at a time, which are then executed continuously.

3.2. Possible Approaches for the Robot Interface

An important decision in the design of the bridge is which approach to take for the part communicating with the robot. We call this part the robot interface. The upcoming sections provide explanations for various ideas. They are compared, assessing their strengths and limitations. Some approaches mentioned in chapter 2 that are not available anymore, such as OPC UA, or are not sufficiently documented like the Robotics API, are not considered in this section.

3.2.1. File KRL

It is possible to have an external (in respect to the KRC) REST API communicating with the KRC through a shared file. On the KRC, there has to be a KRL script repeatedly reading the shared file and executing whatever command the file contains.

A caveat is, that opening a file might lock it and prevent other programs from opening it. We would have to make sure, that every program needing to open a file eventually gets access to it.

Because this approach makes use of the KRC, the inverse kinematics are already built in for all compatible robot models. Thus, this approach would lead to the product theoretically being compatible with all robots that work with a compatible KSS version [KUKA, 2022].

3.2.2. Serial KRL

Previous versions (5.4 - 7.0) of the KUKA System Software (KSS) supported communication over a serial port natively [KUKA, 2011a]. That way, an external REST API could communicate with the KRC through a USB cable for example. Unfortunately, KSS 8.7 does not support this. We would have to implement our own external module [KUKA, 2019].

Searching in Google Scholar for "external module" and "CREAD" (the method calling the external module) only yields one relevant result, [Li, 2011] making use of serial communication on an older KSS version.

It is uncertain, how much time and resources the development of such an external module for KSS 8.7 would take up.

3.2.3. Web Interface KUKA RSI

As shown in 2.2, the KUKA RSI could be extended with a REST API. In this case, the same advantages as with the other approaches integrating the KRC would be present.

Nonetheless, the high cost of the KUKA RSI renders this approach infeasible [KUKA AG, 2023g].

3.2.4. KUKA Ethernet KRL

The KUKA.Ethernet KRL option package allows the exchange of arbitrary XML or binary data over TCP from and to the KRC during runtime [KUKA, 2019c].

In addition to a necessary capital investment [KUKA AG, 2023d] for purchasing the package, [Sanfilippo *et al.*, 2015] mention that it only provides a limited set of functions.

3.2.5. Direct Serial Communication

It is also possible to control the OrangeApps ERS via the USB interface directly, skipping the KRC. The ERS uses a LEGO Spike Prime Technic Hub to control its motors. On the hub, a micropython shell is running executing a file provided by OrangeApps that will allow the robot to be controlled by sending JSON data over the serial interface. In the JSON payload, the robot's axis rotation values and therefore its movements can be defined. With this approach, the kinematics still need to be solved. Not only for the ERS, but all future models.

3.2.6. Y200 Interface

The Y200 interface allows an external program to communicate with the robot controller. It enables a client to read the inputs ($IN variables), outputs ($OUT variables), axis and coordinate values of the KRC. It also provides write access to set Boolean values as the digital inputs. By using the Y200 interface to send data to the KRC, it is not necessary to implement the inverse kinematics. However, the interaction with the Y200 interface on the external side, as well as internally converting the binary inputs back to structured data is not trivial. Due to the lack of easily available documentation the expected cost is uncertain.

3.2.7. FastResearchInterface (FRI)

The FRI exposes the functions of KRL over UDP, which would suit our demand, as it allows any software to control the robot while handling the kinematics. As the FRI depends on the software of the KRC, it is unsuited for our use case with the OrangeApps ERS. The ERS is provided with KSS, whereas the FRI is only supported in combination with the KUKA Sunrise environment.

3.2.8. JOpenShowVar

JOpenShowVar uses an actuator program written in KRL to control the robot, essentially allowing to leverage all of KRL's functionalities. To stop the need for a specific actuator script for each use case, it is possible to develop a script that is able to handle many different use cases. To interact with this actuator, JOpenShowVar uses the Crosscomm

interface. There is hardly any documentation on the Crosscomm interface available, neither using the google search engine nor on KUKA websites.

3.2.9. ROS

There already exists a REST API for ROS, called ROStful [Kehoe, 2014]. We are confident, that it is possible to combine ROStful or another REST API through ROS' plumbing with a robot driver. By using these preexisting solutions, we can save time in development and ensure our product is both extendable with more functionality and generalisable for a variety of robot models.

However, ROS being an extensive framework poses a significant entry barrier, since it takes time to read into and get used to the ROS environment. Furthermore, the existing software is not production ready [Hoorn, n.d.]. As a consequence, additional time would need to be accounted for, as unforeseen problems or bugs could occur.

Implementing a robot driver to work with ROS might also take considerable time, as the parameters for inverse kinematics need to be found. In the best case, the manufacturer or other entities already knowing the joint parameters are willing to provide these. Otherwise, there are multiple ways to identify the kinematic parameters, as shown in [Kolyubin *et al.*, 2015]. For each robot model that the system should be used with, this process of finding the IK parameters has to be repeated, unless there already exists a robot driver for said model in ROS.

Alternatively, the KRC can be used as the robot driver. It can be connected to ROS via many of the previously discussed approaches. However, this means that this approach inherits the negative aspects of any solution used to connect the KRC to ROS. This is the case for financial costs as for the RSI or for undocumented solutions. While ROS is a generally useful framework, it might be excessive for our use case, considering its entry barrier and further its dependence on a robot driver.

3.3. Comparison of Approaches for the Robot Interface

Our solution should, as defined in Section 1.3, be usable with not just the OrangeApps ERS, but with other KUKA robots as well. Furthermore, the selected approach must be feasible under the constraints of limited time and financial resources. Consequently, these two factors play a major role in deciding which approach to take.

Approach	generalisability	Feasibility	Score
File KRL	Compatible KSS required	Yes	2
Serial KRL	Compatible KSS required	External module required, cost uncertain	1.5
RSI	Compatible KSS required	No, financial hurdle	1
Ethernet KRL	Compatible KSS required	No, financial hurdle	1
Direct Serial	No, robot driver required	Yes	1
Y200	Compatible KSS required	Uncertain	1.5
FRI	KUKA Sunrise required	No, financial hurdle	1
JOpenShowVar	Compatible KSS required	Uncertain	1.5
ROS	Robot driver or a proxy approach required	Might exceed the available time frame	1.5

Tab. 3.1.: Comparison of different approaches for the robot interface under the metrics of generalisability and feasibility.

The comparison of the different approaches in table 3.1 shows that for the metric of generalisability, there is a dominant case, namely "Compatible KSS required". For approaches utilising the proprietary KRC, the controller must run a compatible KSS Version or the Sunrise environment. Alternatively, for the Direct Serial approach, a driver for each particular robot model is required, rendering the approach not generalisable. Whereas in the case of ROS, one of the other approaches would sensibly be used as a robot driver, in turn relying on a compatible KSS. Relying on open source drivers would only work for a limited number of robots.

Implementing a robot driver is non-trivial, as it is necessary to find the IK parameters, and has to be repeated for every new robot model [Kolyubin *et al.*, 2015]. In contrast, having a KSS version compatible to KSS 8.7 seems to be the looser condition. KSS versions 8.X should be compatible, and for a KRL program to work on other version, it is to be expected that only parts would need to be rewritten. Different KSS versions are relatively compatible to each other [KUKA, 2009], except input/output using CREAD/CWRITE [KUKA, 2019a].

Concerning the feasibility metric, only the two approaches File KRL and Direct Serial are certainly feasible within timely and financial boundaries of this thesis.

Considering this in combination with the generalisability, we assign each alternative a score that can be seen in table 3.1. For each metric, we assign 1 to the best and 0 to the worst value. Decimals between 0 and 1 are used to express uncertainty or medium values. The score for each alternative is the sum over the metrics.

As an example, the approach Serial KRL gets a score of 1.5. This represents the sum of 1 from the generalisability and 0.5 from the feasibility metric, given that requiring a compatible KSS version is the looser and therefore better constraint limiting generalisability. And the uncertainty of the feasibility of the external module valued with 0.5.

We deem the File KRL approach mentioned in 3.2.1 the best alternative, as it has the highest score.

3.4. Client Interface

Not only the robot-side of this software bridge has to be designed, but also the REST API, e.g. the client interface.

3.4.1. Protocol

According to [Bormann *et al.*, 2012] HTTP [Thomson & Benfield, 2022] is the most popular application protocol on the internet. HTTP can fulfil the requirements of REST [Surwase, 2016], and can thus be used to build REST APIs. This makes it the ideal protocol to use for our approach.

HTTP allows the usage of requests of different methods onto resources (URIs). These methods are GET, POST, PUT and more. Example use cases for the methods are: For GET: Retrieving information located at the specified URI, for POST: Send data to a URI for it to be processed by that URI. And for PUT: Upload data for it to be accessible on the server at the given URI [Nielsen *et al.*, 1999].

Obviously, as the name Hypertext Transfer Protocol already suggests, its methods are designed for text, not for robot movements. Nevertheless, it can be used with robots, as for example with the RobotArm in the OMiPOB environment [Karagiannis & Muck, 2017]. It makes use of all of the three mentioned methods. GET is used to collect the current position, PUT will move the robot to the given destination, and POST is used for controlling mounted tools or submitting sequences of movements. Coordinates are sent as JSON in the request body, but tools are controlled via request parameters.

While the previously mentioned approach is in line with the intended use of the request methods, we decide to rely solely on POST requests. Enforcing different methods unnecessarily increases complexity of programming, as the differentiation between functions is already done with URIs. It is simpler and more consistent to use POST for every request. Especially, given that this too is compatible with the definition of POST, as in any case the robot processes the request. Both perspectives to look at the robot's position are reasonable, for example. With PUT, moving the arm updates the URI, which then can be fetched with GET. In contrast, using POST for commands is principally the same as sending the robot a message.

3.4.2. Framework

We implement the REST API with FastAPI [Ramírez, n.d.]. FastAPI is well adopted and works with asynchronous functions [Kornienko *et al.*, 2021]. Additionally, it natively includes Swagger which allows visualising and interacting with the API without needing any implementation [SmartBear Software, n.d.].

3.4.3. Web Server

For the actual implementation of the API we rely on a web server to handle the HTTP requests. We use Hypercorn [Jones, n.d.], as it supports asyncio locks. As a robot can only process commands in sequence, not in parallel, our implementation will need to

incorporate a mutual exclusive zone. This essentially leads to a phase change between the parallelised and asynchronous REST API and the sequential interface to the robot.

3.4.4. Design of the Entire Implementation

Assembling the previously discussed parts, we reach the entire architecture shown in figure 3.2. The part of the bridge that interacts with the robot (called robot interface in figure 3.2) is implemented on the KRC. It is simply a KRL script controlling the robot with the built in commands. We can treat this as a black box as it is already implemented. The robot interface communicates with the other part of the bridge via shared files. The bridges client interface is built using a Hypercorn web server in the FastAPI framework. Clients can communicate to this interface utilising HTTP.

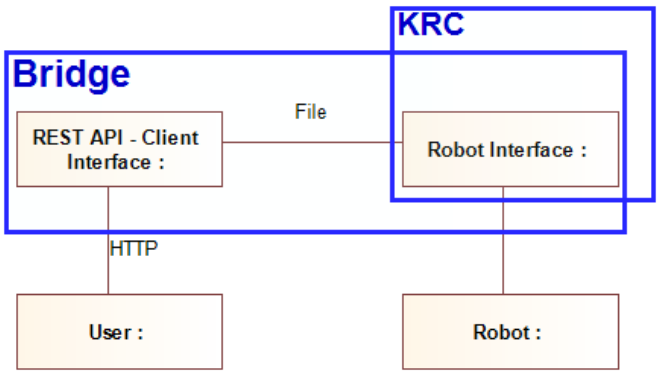

Fig. 3.2.: Concrete architecture of the proposed information system. Components and their communication mediums are depicted. Communication between the client and the bridge (Hypercorn web server) uses the HTTP protocol. Internally, the bridge uses files for inter process communication. The bridge, which is partly built on the robot controller (KRC) uses KUKA's proprietary protocol to control the robot, which we can treat as a black box.

3.5. How to evaluate

To evaluate the system, we utilise a Bee-Up model [Burzynski & Karagiannis, 2020], wherein we use the AdoScript [ADOxx.org, 2019] language to send POST requests to the Bridge. In order to examine the functionalities of the system, we try to execute linear, PTP and circular movements. It must be possible to combine commands to realise complex behaviours. As no tools such as suction cups or grippers are readily available for the OrangeApps ERS, we will limit our testing to drawing with a pen that can be easily fitted to the robot's flange. Subsequently, the implementation is assessed according to the predefined metrics:

- Generalisability
- Functionality
- Error handling
- Ease of usage

4
Implementation and Evaluation

Documentation of the final implementation, including detailed description of its usage. The result of the evaluation, e.g. if the API works, and how extensively and robust it works, and whether it also works for other KUKA robots than the Education Robot from OrangeApps.

4.1. Implementation

Figure 4.1 visualises the complete implementation [Gabriel, 2023]. A client, which can be a user of the Swagger UI [SmartBear Software, n.d.], a Bee-Up instance [Burzynski & Karagiannis, 2020] or any other program, sends a request to the REST API. The Hypercorn webserver [Jones, n.d.] calls an asynchronous function to handle this request. It will then try to enter a critical region. Firstly, in this zone a file is accessed for writing, which by the operating system is limited to one process at a time. This file is the communication channel between the client interface and the robot interface. Secondly, we need to wait for a response from the robot controller to be sure that the command was executed before overwriting it with another command. As the robot can only process commands sequentially, we are forced to somehow transfer from an asynchronous flow to a synchronous one anyways.

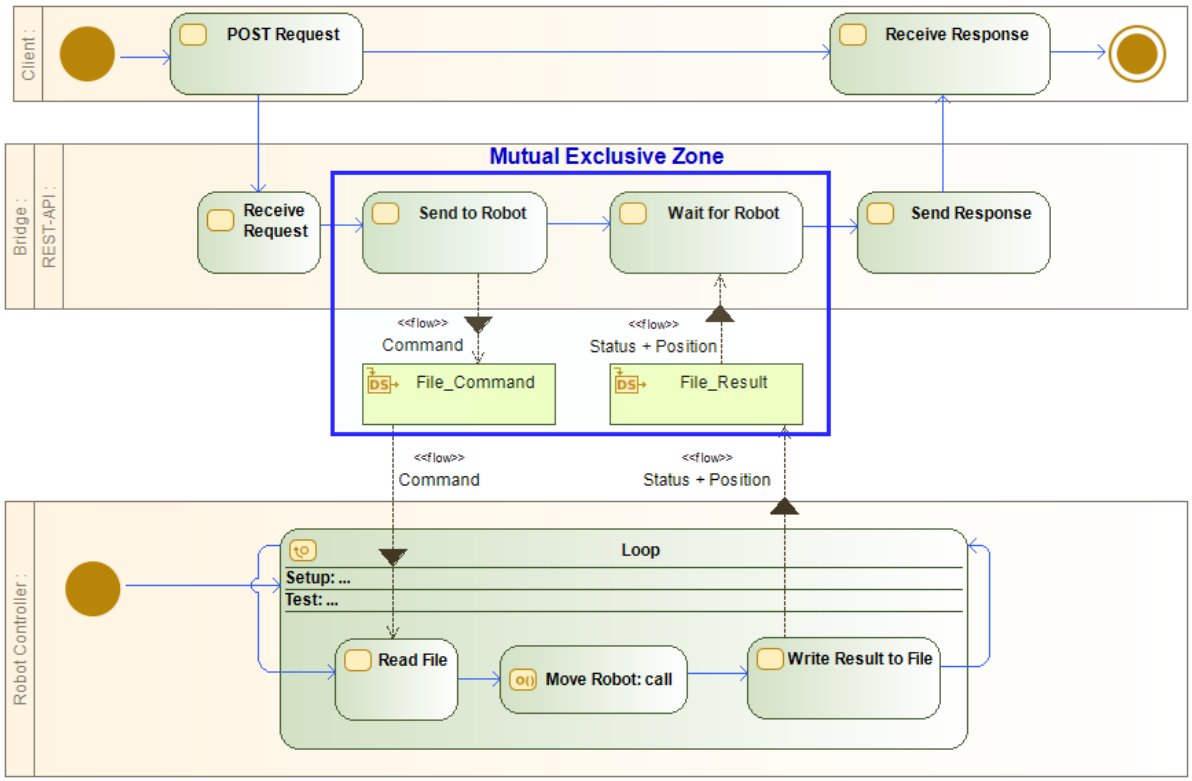

Fig. 4.1.: Activity Diagram showing the process flow when a POST request arrives.

On the robot controller, the KRL interpreter is in a loop and reads the content of the formerly mentioned file. The file contains the next instruction for the robot, which the KRC promptly executes. When finished, the controller writes the result into another file. This file is periodically checked by the client interface. Once the result is found in the file, it is sent back to the client as the POST response.

4.1.1. Exposed Functionalities

To provide the functionalities defined in Section 3.1, we suppose the methods listed in table 4.1. These are implemented in the current version of the KUKA bridge. The IDs are integer representations of the functions that are used to communicate function calls from the client interface to the robot interface. Because of KRL not supporting switch cases with strings [KUKA, 2022], we opted for this approach as we deemed it to be the most readable way of working around this limitation. More details on this topic is given in section 4.1.3.1

These functions provide the ability to move to a point (ptp), do straight lines (linmov) and circular motions (circ). Getting the current position (getpos) is obviously essential. The tool center point (tcp) is a handy function to use when mounting a tool to the robot. It is possible to define the shape of the tool such that when using motion commands, they are applied in respect to the tools tip instead of the robot flange. Essentially, it is like a constant offset applied to target coordinates.

ID	Method	Description
1	ptp	Moves to the given point on an nondeterministic path
2	linmov	Moves to the given point in a straight line
3	getpos	Returns the current position
6	base	Moves to the base position
7	circ	Moves in a circle through the auxillary point to the destination point
8	tcp	Set the tool center point (TCP)
9	gettcp	Returns the current tool center point (TCP)
10	uploadfile	Upload a KRL .src file/function which is then executed

Tab. 4.1.: A list of the implemented functions. The ID is only used internally by the bridge.

The file upload allows the client to upload a custom KRL source file containing a function that will be executed on the robot controller. This allows to use all of KRL's capabilities without the need to implement them separately. This includes splines and time-critical movements, as the time delay introduced by the bridge is removed.

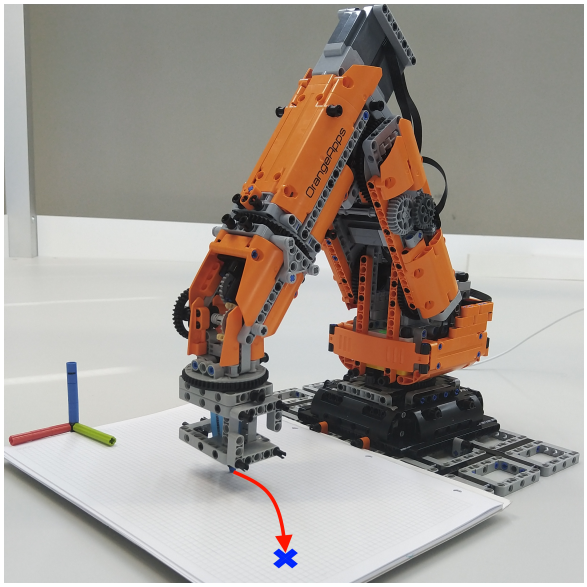

Fig. 4.2.: Illustration of the ptp method. The robot's flange is at (X=200, Y=0, Z=70, A=0, B=180, C=0). The red arrow shows the motion of the robot according to a ptp motion to (x=250, Y=100).

The ptp method is visualised in figure 4.2. It is important to note that the path of the robot is not deterministic [KUKA, 2022]. The path, represented as a red arrow, is essentially a three-dimensional curve.

In Contrast, the linmov method guarantees the path to be a straight line between the start and end point, as shown in figure 4.6.

To receive the information of where the robot currently is, the getpos method is used. The robot's position is specified in the base coordinate system, which originates at the robot's base, see figure 4.4. The same system is used to provide target points for the motion methods. The Cartesian system is six dimensional. The dimensions X, Y and Z determine where a point is in space. Additionally, the orientation of the robot is specified, e.g. the pitch, roll and yaw. These orientations named A, B and C, constitute the other three dimensions. They correspond to rotations around the Z, Y and X axis respectively.

Method 6, base, provides a save "home" position. It is called "canonposition" by OrangeApps and has to be used to transport the ERS. The KRC also requires a "BCO run" at the start of a program [KUKA, 2022]. The BCO run initiated by the bridge also ends at this base position. The base position can also be used as a preprogrammed safe position.

Another motion command is the circ method, allowing smooth circular motions. Unlike the ptp and linmov methods, circ requires two points to be specified. An example for this is depicted in figure 4.5. To define the circular path shown in red, an auxiliary point (blue cross) is used. The path goes through the auxiliary point in a circle and ends at the destination point. There are multiple options for the tool to behave during the motion, more on that in Section 4.1.2.1.

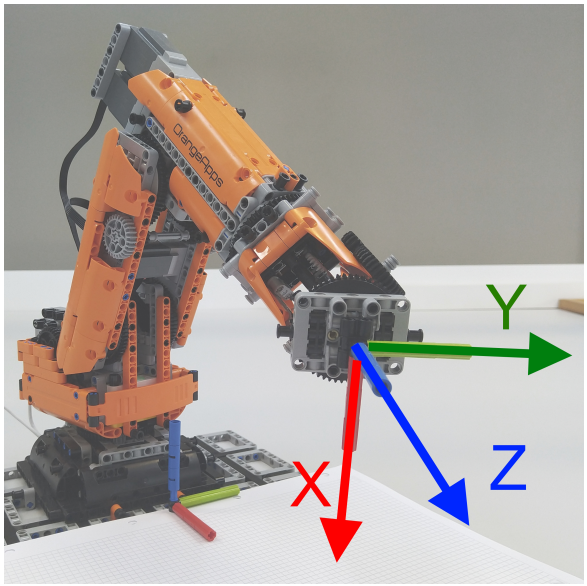

Fig. 4.3.: Illustration of the tool coordinate frame. Its origin is at the robot's flange.

The TCP is a cartesian frame originating at the robot's flange, used to model the mounted tool. It essentially acts as a constant offset to the target coordinates, such that instead of the robot's flange moving to a target point, the top of the tool does. The TCP is illustrated in figure 4.3. The tool frame also includes the orientation, identically to the base coordinate system. Specifying the dimensions of the tool is possible using the tcp method. It sets the tool center point (TCP).

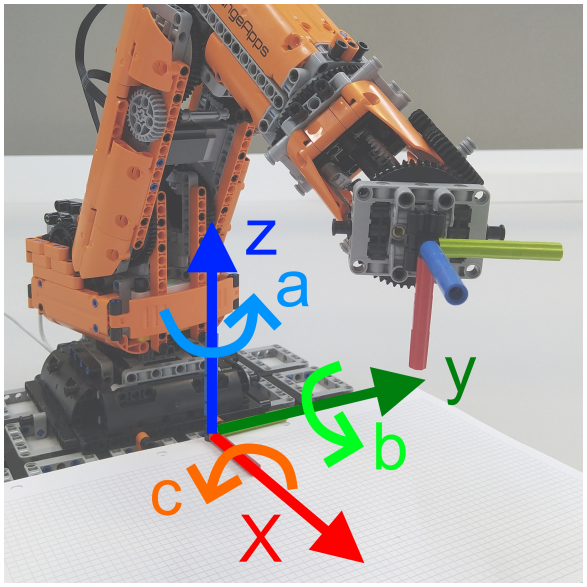

Fig. 4.4.: Illustration of the base coordinate system. The three dimensions X, Y and Z originate at the base of the robot. The rotational orientations A, B and C correspond to rotating around the Z, Y and X axis respectively. Here, the Right-Hand-Rule applies.

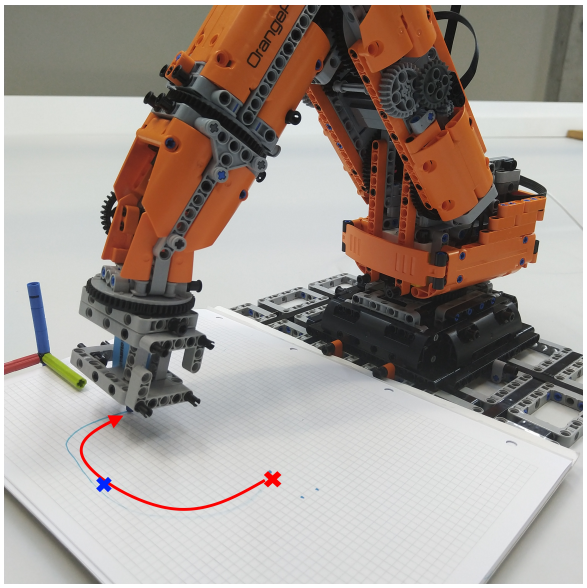

Fig. 4.5.: Illustration of the circ method. The red arrow shows the motion of the robot according to a circ motion. The robot's flange is at the destination point (X=200, Y=-50, Z=80). The red cross (X=200, Y=50) marks the starting point, whereas the blue cross is the auxiliary point (X=250, Y=0).

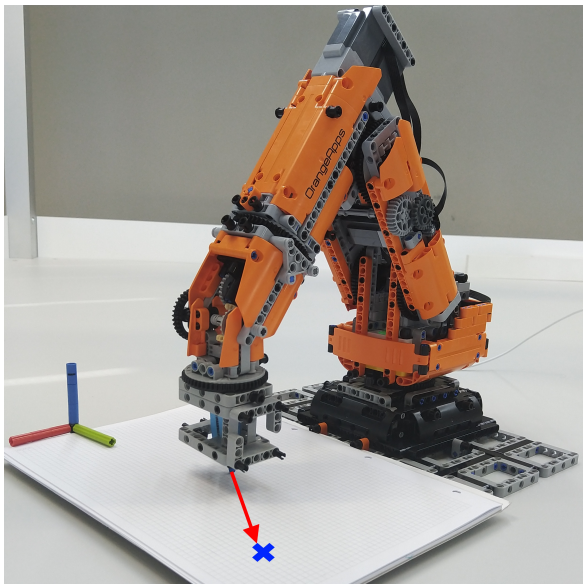

Fig. 4.6.: Illustration of the linmov method. The robot's flange is at (X=200, Y=0, Z=70, A=0, B=180, C=0). The red arrow shows the motion of the robot according to a lin motion to (x=250, Y=100).

4.1.2. Client Interface

This section goes into detail about the client interface. For an overview, this part is highlighted in figure 4.7.

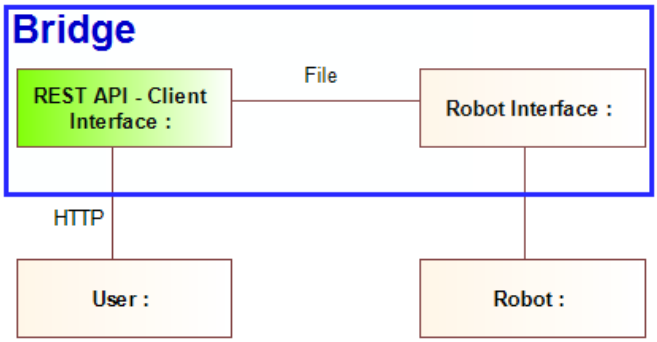

Fig. 4.7.: The architecture of the proposed information system. Highlighted in green is where the client interface is located within the overall architecture.

The base of this part of the bridge is the Hypercorn web server. In listing 4.1, it is shown how it is started. To use mutex locks, we use the asyncio event loop to run the server. In line 6, we create a lock provided by the asyncio library. The mutex is global, so we can access it from within every function. In Line 9, we add the server to the previously created event loop and specify to use the FastAPI framework.

```
1  # create asyncio event loop necessary to use mutexes
2  loop = asyncio.get_event_loop()
3
4  # create a mutex lock, so only one command is sent to the robot at once
5  global mutex_wfile
6  mutex_wfile = asyncio.Lock()
7
8  # create the REST API
9  fapi = loop.create_task(serve(FastAPI(), config))
10 loop.run_forever() # start event loop
```

List. 4.1: Code of the eventloop and mutex.

Within the FastAPI framework, we can define functions that are called upon the server receiving a request. As an example, the Point-to-Point function is given in listing 4.2. The `@app.post` decorator defines upon which HTTP method and on which resource (URI) the function is called and also its response code. For the example, this means using a POST request onto the URI `/krl/ptp` and upon success, it responds with code 200.

The function uses the async prefix, as in the context of webservers it makes sense to have asynchronous function to allow processing requests in parallel. The function parameters are equal to the HTTP parameters. For PTP, these parameters are the coordinates of the target point. Although using 6 separate variables is less readable, it leads to a user-friendly representation in the Swagger UI, which is covered in Section 4.1.2.1.

In Line 5 the method is defined. This number is the internal method ID used to tell the robot interface which function should be executed. Ideally, this would be a string

like "ptp". But because KRL does not support switch cases with strings, we opted for integers. Which ID belongs to which method can be looked up in table 4.1.

```
1  @app.post("/krl/ptp", status_code=200)
2  async def ptp(x : float = None, y : float = None, z : float = None,
3              a : float = None, b : float = None, c : float = None,
4              s : int = None, t : int = None) -> Status:
5      method = "1"
6      async with mutex_wfile:
7          cid = increase_cid()
8          command = await send_to_robot(cid, method, pos=(x,y,z,a,b,c), s=s, t=t)
9          status = await wait_for_robot(cid)
10     return parsed_robot_response(status)
```

List. 4.2: PTP function of the REST API.

The mutex is acquired in Line 6. Within this protected area we essentially move into a sequential realm in contrast to the parallelised REST API. Subsequently, the unique command ID (CID) is calculated. This is necessary for the robot interface to be able to distinguish two identical consecutive commands. The CID is initialised as a random integer as shown in listing 4.3 and increased by one on each function call.

```
1  cid = random.randint(1, 0x7FFFFFFF) # KRL INT max value is 2**32-1
```

List. 4.3: Initialisation of the command ID (CID).

Next on line 8 in listing 4.2, the PTP command is sent to the robot. Then, the program waits for its result. Finally, the status is returned in the HTTP response.

Listing 4.4 contains the code of the *send_to_robot* function. It sends the command, in this example the ID for the PTP method, the CID, and the target coordinates to the robot interface. This is achieved by writing it to the file designated as the communication channel to the robot interface. Said file has the structure shown in listing 4.5. The first line contains the CID, method ID, and the target coordinates, each separated by a space. For methods not using a target point like getpos, zeroes are written. Otherwise, the KRL script would crash since the format would not match the expected format. The coordinates must be saved in a form readable by the StrToE6POS function defined in KRL [KUKA, 2022]. The next lines contain additional data, if the method necessitates it. This can be the auxiliary point and orientation variable for the circ method.

```
1  command = str(cid) + " " + method + " " + pos_str + "\n"
2  while not ok:
3     try:
4         async with aiofiles.open(cmd_file_path, "w") as f:
5             await f.write(command)
6             ok = True
7             return command
8     except: # when failing to open the file
9         await asyncio.sleep(cooldown)
10        print("retry")
```

List. 4.4: Source code of the send_to_robot function. This function writes a command to a file.

```
{CID} {METHOD} {E6POS}\n
[{auxilliary E6POS}\n]
[{ORIENTATION}]
```

List. 4.5: The structure of the file containing the command. Parameter in Brackets [] are optional.

The wait_for_robot function takes the CID as an argument. It keeps reading the receiving file until it contains the robot interfaces response to the given CID. The file is of the structure: `CID STATUSCODE X Y Z A B C` where the single-letter parameters indicate the robot's current position. If the file can not be read, it is retried after a short delay.

```
async def wait_for_robot(cid):
    # Wait for and get the response from the robot controller
    while not ok: # wait for the response of the robot
        try:
            # open the receiving file
            async with aiofiles.open(recv_file_path, "r") as f:
                # read contents and print first line
                lines = await f.readlines()
                elements = lines[0].split()
                # check that the CID of the response matches the CID that we sent and
                    are waiting for
                if elements[0] == str(cid):
                    ok = True

            # wait if it did not yet contain the expected response
            if not ok:
                await asyncio.sleep(cooldown)

        except: # when failing to open the file
            await asyncio.sleep(cooldown)
            print("retrying...")
    return status
```

List. 4.6: Source code of the wait_for_robot function. This function repeatedly reads a file until the robot interface writes its response to the sent command into that file.

The graphical user interface provided by Swagger is shown in figure 4.8. It is accessible by web browser and through it, the methods can be accessed manually.

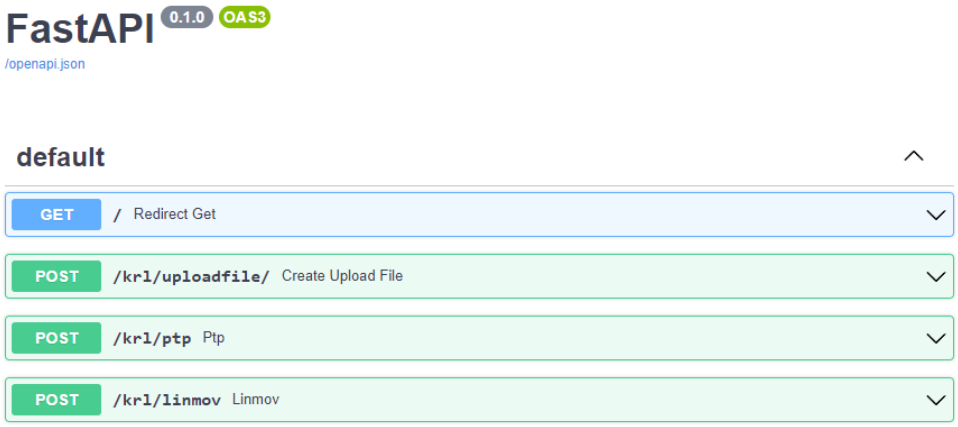

Fig. 4.8.: Overview of the Swagger UI.

4.1.2.1. POST Request Format

For most methods, the necessary POST parameters are coordinates. Each dimension is defined by a separate argument. For the ptp method, the available parameters are `x,y,z,a,b,c,s,t`. The interface for the ptp method in Swagger UI is show in figure 4.9. It allows the user to input each parameter manually and send the request by pressing the *Execute* button. Variables x to c are the points coordinates. At this point, we will not delve into the meaning of variables s and t. They are explained in section 5.2.

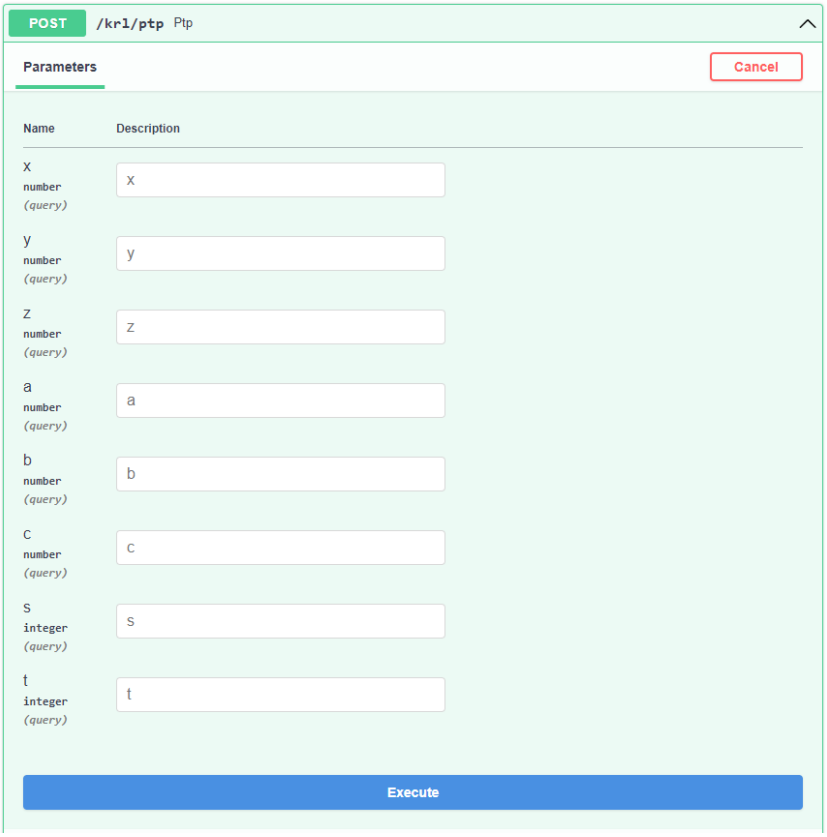

Fig. 4.9.: Parameters for the PTP function in the Swagger UI.

If some parameters are not specified, it depends on the method whether default values are used or the variables are not forwarded at all. For example, if for a PTP motion not all dimensions are specified, the current values of missing dimensions are kept. So if the robot currently is at `x=20, y=0, z=10` and you make the request: `/krl/ptp?x=30` the robot will move to the position `x=30, y=0, z=10`.

4.1.2.2. POST Response Format

The response sent to the client upon executing a command is formatted as JSON, for example `{"statuscode": 1, "position": [200, 0, 200, 0, 90, 0] }`. The responses schema is specified using the pydantic library, shown in listing 4.7. It is a dictionary including the keys satuscode and position. The latter contains the coordinates of the robot's current position, whereas the former indicates the status with which the command was finished.

```
1 class Status(BaseModel):
2     statuscode: int = 1
3     position: List[float] = [200, 0.0, 200.0, 0, 90, 0] #position of robot after the
          command
```

List. 4.7: Schema of the response based on the pydantic library.

4.1.3. Robot Interface

This part describes the robot interface. In figure 4.10 it is highlighted where this part is situated in respect to the entire implementation.

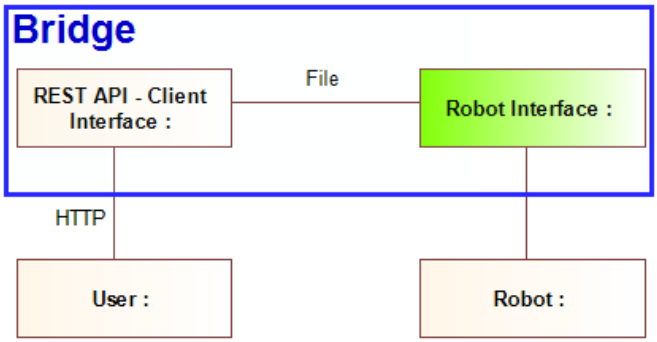

Fig. 4.10.: The architecture of the proposed information system. Highlighted in green is where the robot interface is located within the overall architecture.

The robot interface is implemented using the "File KRL" introduced in Section 3.3. This means, we utilise the KUKA Robot Controller to execute an actuator script written in KRL. This script interacts through files with the client interface. The KRL interpreter handles the communication to and from the robot as well as the kinematics. KSS Version 8.7 is used for development and testing of the software.

In listing 4.8, pseudo code is used to explain the logic of the actuator program. It continuously reads the CID, method ID and coordinates from the command file. If the

CID is new, e.g. not the same as last time, the command is executed, otherwise this procedure is repeated after a short delay. After the command is executed, the result is written to the result file combined with the CID.

```
while true:
    read (cid, method, point) from command file
    if cid is new:
        do method(point)
        write (cid, status, position) to result file
    else:
        wait some time
```

List. 4.8: Pseudo-code of the robot interface.

4.1.3.1. Internal Communication

Listing 4.9 contains the KRL code used to read and parse the command file. The file is opened using the CWRITE statement [KUKA, 2019a], which takes as arguments the channel, in this case `$FCT_CALL` which allows to interact with files. STAT and MODE are variables where the return code is stored and the write-mode is specified respectively, which are irrelevant in our use case. The next argument is the function `krl_fopen` which opens a file, followed by the arguments for that function, the filename, write mode, and the variable where the file handle is stored.

Next, on line 3, the `krl_fgets` function is used to read the files content to the rawchar array. The next arguments are the buffer size of rawchar and the variable where the number of actually read bytes is stored. The raw string is then parsed in Line 5 using the SREAD statement. It reads from rawchar using the specified format `"%d %d %r"`, with the two `%d` indicating the integers representing the CID and the method ID, and the `%r` indicating a string, in this case the string representation of the target position. This position is then converted into the E6POS type on line 7. This type is necessary to pass the target point to any motion commands in KRL.

```
;file needs to be in C: KRC/ROBOTER/USERFILE
CWRITE($FCT_CALL,STAT,MODE,"krl_fopen","numpospyv6_d5.txt","r",HANDLE)
CWRITE($FCT_CALL,STAT,MODE,"krl_fgets",HANDLE,rawchar[],190,read)
OFFSET = 0
SREAD(rawchar[],STAT,OFFSET,"%d %d %r", CID, method, POS_STR[])
STR_RET = StrToE6POS(POS_STR[], TAR6P)
```

List. 4.9: KRL Code for reading the command file.

After executing the received command, its result, which is the current position and the status code, is written to the other file. The code for this is in listing 4.10. We pass the result and the CID, together with how it should be formatted, to the custom function fprintf. The function internally uses the krl_fprintf function in combination with CWRITE, similar to listing 4.9, to write the result to the specified filename.

To actually execute a command, the robot interface relies on various KRL statements. Most methods are directly built into KRL, such as the PTP or LIN commands. Others, such as the getpos or tcp methods, are implemented by reading or writing to variables.

The $POS_ACT variable contains the current position of the robot, whereas the variable $TOOL$ represents the current configured tool.

```
;SECTION WRITE
filename[] = "kapi_result.txt"
format[] = "%d %d %f %f %f %f %f %f"
WSTAT = fprintf(filename[], format[], CID, FRET.STATUSCODE, FRET.RETPOS)
```

List. 4.10: KRL Code for writing to the result file.

For the method of uploading a KRL file, the uploaded file will be called like a normal function. An example for a valid KRL function is given in listing 4.11. The file must include a function named `fuploaded` returning an integer, which will be called by the main program. There are no further restrictions, all of KRL's functionalities can be utilised.

```
DEFFCT INT fuploaded()
    SPLINE
    SPL {X 200, Y 0, Z 200, A 0, B 90, C 0}
    SPL {X 200, Y -50, Z 150, A 0, B 90, C 0}
    SPL {X 200, Y 50, Z 100, A 0, B 90, C 0}
    ENDSPLINE
    RETURN 1
ENDFCT
```

List. 4.11: Example of the code for the fuploaded function. The SPLINE keyword denotes a section describing a Spline motion, and the SPL keyword defines a point on the curve.

4.2. Demonstration

In this section, the usage of the bridge for KUKA robots is demonstrated.

We use Bee-Up to simulate a client. By using AdoScript [ADOxx.org, 2019], we can send POST requests interactively from within a Bee-Up model. The model we use contains a variety of commands, from simple motions to a combination of primitive motions to achieve complex behaviours like drawing a smiley. These are visible in figure 4.11.

Bee-Up is run on a separate device from the KUKA robot controller, but both machines are connected over the local area network. The client interface however is run on the KRC, as well as the robot interface. We start both according to the instructions in the readme file.

In the model, we run example tasks like drawing on paper, as the set of available tools is limited. For example, there is no gripper available for the Education Robot System which we use for the demonstration. Nevertheless, drawing on paper is sufficient to test the complete functionality.

To demonstrate the usage of our product, we implement various procedures containing commands, e.g. POST requests to the robot interface, in a Bee-Up model. They can be executed by clicking the "execute" button at the top right of a procedure's representation

as seen in figure 4.11. Procedures can either be primitive, only executing a single POST request, or more complex, including multiple requests or also calling other procedures.

The procedure "Move to Base" is considered a primitive, as it only contains one POST request to move the robot to its base position. Contrary, the "Draw Smiley" procedure utilises other procedures, such as the "Draw Fixed Horizontal Circle", and combines them in a sequence.

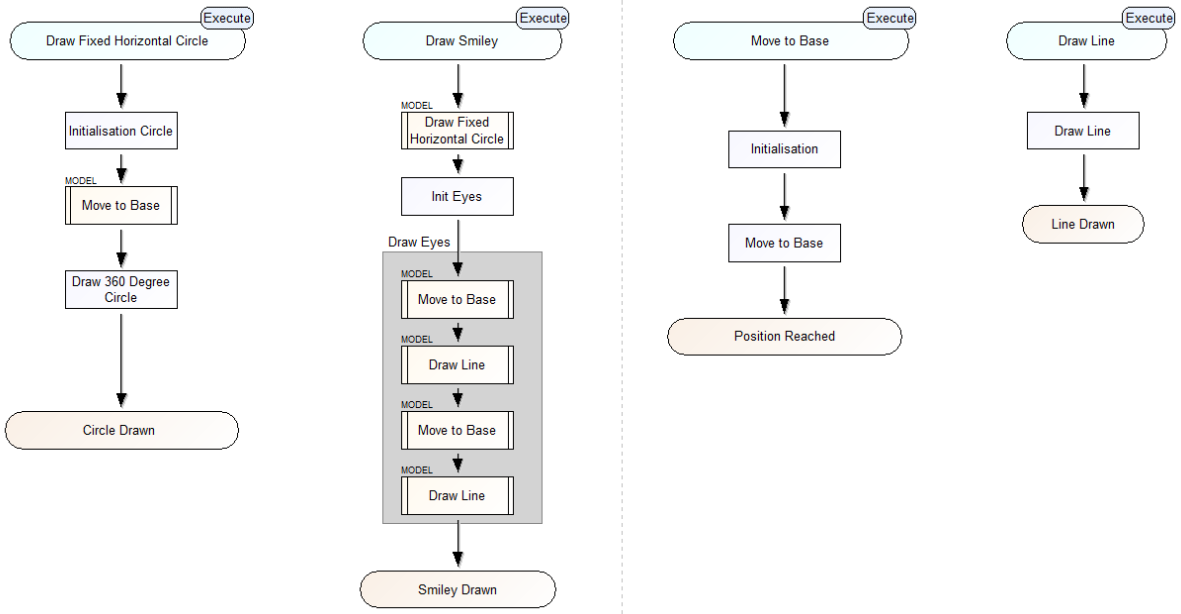

Fig. 4.11.: The Bee-Up model to draw smileys.

4.3. Evaluation

In this section, an evaluation of the implementation is presented in which the bridge is compared against the objectives. We use Bee-Up [Burzynski & Karagiannis, 2020], an ADOxx-based modelling tool to create sample tasks for the robot to subsequently evaluate the following metrics:

- Generalisability
- Movement functionalities (circles, lines, time, ...) by coordinates
- Error handling
- Ease of usage

Generalisability and the functionalities are metrics specified previously. We used Ease of Usage and Error Handling as additional metrics, since during the research and development we deemed these metrics as important for a good solution.

Regarding the generalisability, we are not able to test our implementation on robots other than the OrangeApps ERS. Thus, we can only assume, that in theory the implementation is compatible across various KUKA robot models. As the robot interface is written in KRL, it is supposed to be runnable on any compatible KRC/KSS version. The client

interface is completely independent of the used robot, and the communication between the two interfaces should also be possible on most systems.

Concerning the functionalities, we did succeed in sending commands to the robot via the bridges REST API. All of the implemented primitives behaved as designed and thus the implementation provides the same functionality as the KRL motion statements. The primitives can be combined sequentially, just as in KRL. Complex motions (splines, time critical) are also possible using the file upload functionality. Hence, the functionality requirement is fulfilled. However, the method to upload a custom KRL source file is not fully functional. While at first the method succeeds in meeting the objective, by allowing the user to upload a KRL file containing for example, a spline motion along multiple points, no second file can be uploaded. If this method is used twice or more times, each time the first uploaded code is executed. This severely limits the utility of this functionality.

As discussed, we are able to chain together multiple commands to achieve complex motions, such as drawing non-trivial shapes like smileys. However, there exist some shortcomings to this feature. Firstly, it may happen that after some motions the robot reaches an axis configuration that makes further movements in some directions impossible. Secondly, the bridge offers hardly any error handling. Frequent errors, such as "Software Limit Switches" that occur when an axis reaches its rotational limit, are unhandled, hence require a restart of the entire ridge. This is the case if the procedure for drawing the smiley is extended with drawing a circle as the smileys mouth. Notable is, that this issue also exists in plain KRL, unless one uses the status and turn variables. However, their usage is not trivial. Thus, calculating the intended value for status and turn is time consuming and might be necessary for each point.

As previously mentioned, the occurrence of an error message in the robot interface necessitates a restart of the KRC. Normally, when using KRL, it is sufficient to confirm the error message and continue running other commands. But it seems, that the open file handles used to communicate with the client interface are not closed upon crashing or confirming the error. Thus, leading to locked files until the operating system is rebooted. The client interface also needs to be restarted, otherwise it would indefinitely wait for a response.

Installing the bridge according to the instructions requires approximately the same effort as setting up comparable solutions. To start the bridge, both the client and the robot interface need to be launched. This is fairly direct, albeit not plug and play. Finally, using the bridge is easy, as the methods and parameters presented in a human-readable format in the UI. There were also no impediments for programmatic usage.

5 Conclusion

Robots are a key reason for increased productivity [Schierl, 2017] and their numbers are expected to continue growing [International Federation of Robotics, 2022] This aligns with the Industry 4.0 concept, which seeks to combine information systems and robotics to improve performance [Benotsmane *et al.*, 2018].

However, there is a major challenge impeding this integration. Many robot manufacturers employ proprietary communication protocols and interfaces, resulting in compatibility issues between different robots and information systems [Arnarson *et al.*, 2020].

In this project, we analyse the state of the art in controlling KUKA robots through REST APIs. We identify that there does not yet exist a product providing a REST API for KUKA robots. Existing approaches to control KUKA robots either do not offer a REST API, require model-specific drivers, or incorporate costly, unavailable, or undocumented dependencies. Consequently, we work out the requirements and design for such a product. Ultimately, we implement a solution to bridge KUKA robots to a REST API.

The developed product [Gabriel, 2023] provides various functions like PTP motions over a REST API. It is built on the FastAPI framework and communicates with the KUKA Robot Controller (KRC) through files. On the KRC, a KRL script is run to read the files and execute commands on the robot. The product is supposed to be usable with many KUKA robots, given the KSS version is compatible.

The following section analyses the added value of our solution.

5.1. Discussion

In comparison to the state of the art, the implemented product provides equal or better generalisability, depending on the specific product, without costly, unavailable or undocumented dependencies. Not needing per-model robot drivers makes our solution work with all compatible KSS versions supporting communication over files. Similar to JOpenShowVar [Sanfilippo *et al.*, 2015], our proposed solution utilises an actuator program written in KRL. However, we use a single actuator for all use cases and communicate to the KRC through files instead of the Crosscomm interface.

Regarding functionality, our implementation provides the KRL's key functions over a REST API. The API includes the functionality to upload KRL files, making all of KRL's methods available to the client, although less conveniently than the natively supported

functions. Some previous solutions provide similar functionality, while others are limited to information exchange, requiring a robot driver for execution of a command by the robot.

The error handling of our implementation is limited compared to the state of the art. For example, JOpenShowVar and the Robotics API [Angerer *et al.*, 2013] offer some levels of error handling, whereas our products error handling is limited to the errors KRL is able to catch.

Although the installation process of our implementation requires manual steps, they are not complex. Starting the program is straight forward with only the order of commands being important. The bridge is user-friendly with its parameters and methods clearly presented in the UI and poses no issues for programmatic usage.

5.2. Improvements

On the basis of this work, future research can be conducted in controlling robots. For example, our implementation could be extended with more functionalities. For instance, allowing multiple commands to be sent in the request body might be a valuable simplification for executing motions without delay, rather than necessitating a file upload in a proprietary language.

Another possible improvement is to prevent the robot from reaching unfavourable positions after a motion that could lead to the robot reaching axis limitations. The `KUE_WEG` function provided by KUKA calculates the S and T values for axis 5 for a destination point such that the axis 4 and 6 move as little as possible. We suppose that by adapting that function, a smart program can be developed to intelligently set S and T values making it less likely to reach an axis limit.

Further possible improvements concern the error handling and simplifying the bridges usage. An example of a potential improvement is ensuring the robot interface does not fail when there is no file available to read.

References

[Alam *et al.*, 2020] Alam, Md Shahedul, Atmojo, Udayanto Dwi, Blech, Jan Olaf, & Lastra, Jose L. Martinez. 2020. A REST and HTTP-based Service Architecture for Industrial Facilities. *Pages 398–401 of: 2020 IEEE Conference on Industrial Cyberphysical Systems (ICPS)*, vol. 1. 3, 4

[Angerer *et al.*, 2013] Angerer, Andreas, Hoffmann, Alwin, Schierl, Andreas, Vistein, Michael, & Reif, Wolfgang. 2013. Robotics API: object-oriented software development for industrial robots. 01. 10, 34

[Arbo *et al.*, 2020] Arbo, M.H., Eriksen, I., Sanfilippo, F., & Gravdahl, J.T. 2020. Comparison of KVP and RSI for Controlling KUKA Robots Over ROS. *IFAC-PapersOnLine*, **53**(2), 9841–9846. 21st IFAC World Congress. 10

[Arnarson *et al.*, 2020] Arnarson, Halldor, Solvang, Bjørn, & Shu, Beibei. 2020. The application of open access middleware for cooperation among heterogeneous manufacturing systems. *Pages 1–6 of: 2020 3rd International Symposium on Small-scale Intelligent Manufacturing Systems (SIMS)*. 2, 9, 33

[Benotsmane *et al.*, 2018] Benotsmane, R, Dudás, L, & Kovács, Gy. 2018. Collaborating robots in Industry 4.0 conception. *IOP Conference Series: Materials Science and Engineering*, **448**(1), 012023. 2, 33

[Bormann *et al.*, 2012] Bormann, Carsten, Castellani, Angelo P., & Shelby, Zach. 2012. CoAP: An Application Protocol for Billions of Tiny Internet Nodes. *IEEE Internet Computing*, **16**(2), 62–67. 17

[Bruyninckx, 2001] Bruyninckx, H. 2001. Open robot control software: the OROCOS project. *Pages 2523–2528 vol.3 of: Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 3. 10

[Burzynski & Karagiannis, 2020] Burzynski, Patrik, & Karagiannis, Dimitris. 2020 (February). bee-up - A teaching tool for fundamental conceptual modelling. *Pages 217–221 of:* Michael, Judith, & Bork, Dominik (eds), *Modellierung 2020 Tools & Demos*. Modellierung 2020 Short, Workshop and Tools & Demo Papers. 4, 11, 18, 19, 31

[Day, 2018] Day, Chia-Peng. 2018. Robotics in Industry—Their Role in Intelligent Manufacturing. *Engineering*, **4**(4), 440–445. 2

[Kolyubin *et al.*, 2015] Kolyubin, Sergey, Paramonov, Leonid, & Shiriaev, Anton. 2015. Robot Kinematics Identification: KUKA LWR4+ Redundant Manipulator Example. *Journal of Physics: Conference Series*, **659**(1), 012011. 15, 16

[Kornienko *et al.*, 2021] Kornienko, D V, Mishina, S V, Shcherbatykh, S V, & Melnikov, M O. 2021. Principles of securing RESTful API web services developed with python frameworks. *Journal of Physics: Conference Series*, **2094**(3), 032016. 17

[KUKA , 2019] KUKA , Deutschland GmbH. 2019. *KUKA.OPC UA 2.0. Version: KST OPC UA 2.0 V2* Publication: PB9970 `https://xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-common_PB9970_en` (accessed July 10, 2023). 9

[KUKA, 2019a] KUKA, Deutschland GmbH. 2019a. *CREAD/CWRITE. Version: KSS/VSS 8.2 to 8.7 CREAD CWRITE V1* Publication: PB3031 `https://xpert.kuka.com/ID/PB3031` (accessed May 24, 2023). 16, 29

[KUKA, 2019b] KUKA, Deutschland GmbH. 2019b. *KUKA Sunrise.FRI 2.4. Version: KUKA Sunrise.FRI 2.4 V1* Publication: PB10479 `https://xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-common_PB10479_en` (accessed June 16, 2023). 8

[KUKA, 2019c] KUKA, Deutschland GmbH. 2019c. *KUKA.Ethernet KRL 3.2. Version: KST Ethernet KRL 3.2 V1* Publication: PB12988 `xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-common_PB12988_en` (accessed May 31, 2023). 14

[KUKA, 2022] KUKA, Deutschland GmbH. 2022. *KUKA System Software 8.7. Version: KSS 8.7 SI V6* Publication: PB14656 `https://xpert.kuka.com/ID/PB14656` (accessed May 24, 2023). 3, 12, 13, 20, 21, 25

[KUKA, 2009] KUKA, Roboter GmbH. 2009. *Compatibility from 5.x to 8.x. Version: Kompatibilität 8.x V2* Publication: PB4059 `xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-common_PB4059_en` (accessed May 31, 2023). 16

[KUKA, 2011a] KUKA, Roboter GmbH. 2011a. *CREAD/CWRITE. Version: KSS 5.4, 5.5, 5.6, 7.0 CREAD/CWRITE V1* Publication: PB1053 `https://xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-common_PB1053_en` (accessed May 31, 2023). 13

[KUKA, 2011b] KUKA, Roboter GmbH. 2011b. *KUKA.FastResearchInterface 1.0. Version: KUKA.FRI 1.0 V2 en* Publication: PB4321 `https://xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-common_PB4321_en` (accessed June 16, 2023). 8

[Li, 2011] Li, Yunlai. 2011. *Development of a robot-based magnetic flux leakage inspection system.* Ph.D. thesis. 13

[Mokaram *et al.*, 2017] Mokaram, Saeid, Aitken, Jonathan, Martinez-Hernandez, Uriel, Eimontaite, Iveta, Cameron, Dave, Rolph, Joe, Gwilt, Ian, McAree, Owen, & Law, James. 2017. A ROS-integrated API for the KUKA LBR iiwa collaborative robot. *IFAC-PapersOnLine*, **50**(07), 15859–15864. 10

[Mühe *et al.*, 2010] Mühe, Henrik, Angerer, Andreas, Hoffmann, Alwin, & Reif, Wolfgang. 2010. *On reverse-engineering the KUKA Robot Language.* 3, 7

[Nielsen *et al.*, 1999] Nielsen, Henrik, Mogul, Jeffrey, Masinter, Larry M, Fielding, Roy T., Gettys, Jim, Leach, Paul J., & Berners-Lee, Tim. 1999 (June). *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. 17

[Peffers *et al.*, 2007] Peffers, Ken, Tuunanen, Tuure, Rothenberger, Marcus A., & Chatterjee, Samir. 2007. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, **24**(3), 45–77. 4, 5

[Sanfilippo *et al.*, 2015] Sanfilippo, Filippo, Hatledal, Lars Ivar, Zhang, Houxiang, Fago, Massimiliano, & Pettersen, Kristin Y. 2015. Controlling Kuka Industrial Robots: Flexible Communication Interface JOpenShowVar. *IEEE Robotics Automation Magazine*, **22**(4), 96–109. 2, 7, 8, 14, 33

[Schierl, 2017] Schierl, Andreas. 2017. *Object-Oriented Modeling and Coordination of Mobile Robots*. doctoralthesis, Universität Augsburg. 2, 33

[Schreiber *et al.*, 2010] Schreiber, Günter, Stemmer, Andreas, & Bischoff, Rainer. 2010. The fast research interface for the kuka lightweight robot. *Pages 15–21 of: IEEE workshop on innovative robot control architectures for demanding (Research) applications how to modify and enhance commercial controllers (ICRA 2010)*. Citeseer. 2, 8

[Surwase, 2016] Surwase, Vijay. 2016. REST API modeling languages-a developer's perspective. *Int. J. Sci. Technol. Eng*, **2**(10), 634–637. 3, 17

[Thomson & Benfield, 2022] Thomson, Martin, & Benfield, Cory. 2022 (June). *HTTP/2*. RFC 9113. 17

[Zuther, 2019] Zuther, Patrick. 2019. *Integration einer sensorgeführten Robotersteuerung*. 7

Referenced Web Resources

[ADOxx.org, 2019] ADOxx.org. 2019. *AdoScript Documentation.* https://www.adoxx.org/AdoScriptDoc/index.html (accessed July 30, 2023). 18, 30

[Chatzilygeroudis *et al.*, 2019] Chatzilygeroudis, Konstantinos, Mayr, Matthias, Fichera, Bernardo, & Billard, Aude. 2019. *iiwa_ros: A ROS Stack for KUKA's IIWA robots using the Fast Research Interface.* http://github.com/epfl-lasa/iiwa_ros (accessed July 29, 2023). 10

[Gabriel, 2023] Gabriel, Marco B. 2023 (July). *A Bridge for KUKA Robots using REST API.* doi: 10.5281/zenodo.8156235 url: https://doi.org/10.5281/zenodo.8156235 (accessed July 20, 2023). 19, 33

[Hoorn, n.d.] Hoorn, G.A. vd. *kuka - ROS Wiki.* http://wiki.ros.org/kuka (accessed May 15, 2023). 10, 15

[International Federation of Robotics, 2022] International Federation of Robotics. 2022. *World Robotics 2022.* https://ifr.org/downloads/press2018/2022_WR_extended_version.pdf (accessed February 13, 2023). 2, 33

[Jones, n.d.] Jones, Philip. *Hypercorn.* https://pgjones.gitlab.io/hypercorn/ (accessed June 07, 2023). 17, 19

[Jülg, n.d.] Jülg, Christian. *ros-industrial/kuka_experimental.* https://github.com/ros-industrial/kuka_experimental/tree/melodic-devel (accessed July 18, 2023). 10

[Karagiannis & Muck, 2017] Karagiannis, Dimitris, & Muck, Christian. 2017. *OMiLAB Physical Objects (OMiPOB).* https://www.omilab.org/assets/docs/OMiROB_description_draft.pdf (accessed May 15, 2023). 11, 17

[Kehoe, 2014] Kehoe, Ben. 2014. *Introducing ROStful: ROS over RESTful web services.* https://www.ros.org/news/2014/02/introducing-rostful-ros-over-restful-web-services.html (accessed May 16, 2023). 15

[KUKA AG, 2023a] KUKA AG. 2023a. *KUKA robot controller KR C5.* https://www.kuka.com/en-ch/products/robotics-systems/robot-controllers/kr-c5 (accessed June 16, 2023). 6

[KUKA AG, 2023b] KUKA AG. 2023b. *KUKA smartPAD.* https://www.kuka.com/en-ch/products/robotics-systems/robot-controllers/smartpad (accessed June 16, 2023). 6

[KUKA AG, 2023c] KUKA AG. 2023c. *KUKA Sunrise.FRI 2.5*. https://my.kuka.com/s/product/kuka-sunrisefri-25/01t1i000000tTEpAAM (accessed June 16, 2023). 8

[KUKA AG, 2023d] KUKA AG. 2023d. *KUKA.Ethernet KRL 3.2*. https://my.kuka.com/s/product/kukaethernet-krl-32/01t1i000000sTqVAAU (accessed May 22, 2023). 14

[KUKA AG, 2023e] KUKA AG. 2023e. *KUKA.FastResearchInterface 1.0*. https://my.kuka.com/s/product/kukafastresearchinterface-10/01t58000004kzQ5AAI (accessed June 16, 2023). 8

[KUKA AG, 2023f] KUKA AG. 2023f. *KUKA.OPC UA Premium 2.0*. https://xpert.kuka.com/service-express/portal/project1_p/document/kuka-project1_p-basic_AR36039_en (accessed July 10, 2023). 9

[KUKA AG, 2023g] KUKA AG. 2023g. *KUKA.RobotSensorInterface 5.0*. https://my.kuka.com/s/product/detail/01t1i000000sTqLAAU?language=en_US (accessed May 15, 2023). 3, 7, 14

[KUKA AG, 2023h] KUKA AG. 2023h. *Upgrade packages for updating the KUKA operating system*. https://www.kuka.com/en-de/company/press/news/2020/04/kss-vss-upgrade (accessed May 31, 2023). 6

[maxosprojects, n.d.] maxosprojects. *open-dobot*. https://github.com/maxosprojects/open-dobot/ (accessed May 16, 2023). 11

[OPC Foundation, n.d.] OPC Foundation. *Unified Architecture*. https://opcfoundation.org/about/opc-technologies/opc-ua/ (accessed February 13, 2023). 3

[Open Source Robotics Foundation, n.d.a] Open Source Robotics Foundation. *ROS - Robot Operating System*. https://www.ros.org (accessed February 13, 2023). 2, 9

[Open Source Robotics Foundation, n.d.b] Open Source Robotics Foundation. *ROS - The ROS Ecosystem*. https://www.ros.org/blog/ecosystem/ (accessed May 15, 2023). 10

[OrangeApps, n.d.] OrangeApps. *Education Robot System*. https://www.orangeapps.de/?lng=de&page=apps%2Fers3 (accessed February 13, 2023). 3, 4, 6

[Ramírez, n.d.] Ramírez, Sebastián. *FastAPI*. https://github.com/tiangolo/fastapi (accessed June 07, 2023). 17

[SmartBear Software, n.d.] SmartBear Software. *REST API Documentation Tool | Swagger UI*. https://swagger.io (accessed June 08, 2023). 17, 19

A
Common Acronyms

API Application Programming Interface
BCO Block Coincidence
CIRC Circular Motion
DSRM Design Science Research Methodology
ERS Education Robot System (by OrangeApps)
FRI Fast Research Interface
HTTP Hypertext Transfer Protocol
IK Inverse Kinematics
IoT Internet of Things
JSON JavaScript Object Notation
KRC KUKA Robot Controller
KRL KUKA Robot Language
KSS KUKA System Software
LIN Linear Motion
OMiLAB Open Models Initiative Laboratory
OMiPOB OMiLAB Physical Objects
PTP Point-to-Point (Movement)
REST Representational State Transfer
ROS Robot Operating System
RSI Robot Sensor Interface
TCP Tool Center Point
UI User Datagram Protocol
UI User Interface
URI Uniform Resource Identifier
USB Universal Serial Bus

B
Product

The product created in this work is made available to readers on Zenodo. It can be accessed under the following DOI and URL:

DOI: 10.5281/zenodo.8156235

URL: `https://doi.org/10.5281/zenodo.8156235`