Controlling *Kuka* Industrial Robots: Flexible Communication Interface *JOpenShowVar*

F. Sanfilippo, L. I. Hatledal and H. Zhang
Department of Maritime Technology and Operations, Aalesund University College
Postboks 1517, 6025 Aalesund, Norway
{fisa, laht, hozh}@hials.no

M. Fago
IMTS S.r.L. Company
Taranto, Italy
massimiliano.fago@gmail.com

K. Y. Pettersen
Department of Engineering Cybernetics, Norwegian University of Science and Technology
7491 Trondheim, Norway
kristin.y.pettersen@itk.ntnu.no

***Abstract*— *JOpenShowVar* is a Java open-source cross-platform communication interface to *Kuka* industrial robots. This novel interface allows for read-write use of the controlled manipulator variables and data structures. *JOpenShowVar*, which is compatible with all *Kuka* industrial robots that use *Kuka* Robot Controller version 4 (*KR C*4) and *Kuka* Robot Controller version 2 (*KR C*2), runs as a client on a remote computer connected with the *Kuka* controller via Transmission Control Protocol/Internet Protocol (*TCP/IP*). Even though only soft real-time applications can be implemented, *JOpenShowVar* opens up to a variety of possible applications, making both the use of various input devices and sensors as well as the development of alternative control methods possible.**

Four case studies are presented to demonstrate the potential of *JOpenShowVar*. The first two case studies are open-loop applications, while the last two case studies describe the possibility of implementing closed-loop applications. In the first case study, the proposed interface is used to make it possible for an *Android* mobile device to control a *Kuka KR 6 R900 SIXX (KR AGILUS)* manipulator. In the second case study, the same *Kuka* robot is used to perform a two-dimensional line-following task that can be used for applications like advanced welding operations and similar. In the third case study, a closed-loop application is developed to control the same manipulator with a *Leap Motion Controller* that supports hand and finger motions as input without requiring contact or touching. In the fourth case study, a bidirectional closed-loop coupling is established between a *Force Dimension omega.7* haptic device and the same *Kuka* manipulator. Related experiments are carried out to validate the efficiency and flexibility of the proposed communication interface.

***Index Terms*— Robot interface, *Kuka* industrial robots, input device.**

I. Introduction

Industries that employ robots in a wide variety of applications are the main customers for robot manufacturers. The manipulator market for research applications, on the other hand, is simply too small for the robot manufacturing industry to develop models specifically for such use. While the hardware and mechanical requirements of developed robots are often similar for both industry and research, scientific software requirements are quite different and even contradictory in many aspects [1], [2]. The goal of scientists is to try to gain as much control over the robot as possible, whereas industries seek safe and easy operational interfaces. In particular, although software interfaces that are appropriate for industrial use are available, it is difficult to find interfaces that are applicable for research purposes. The disclosure of the internal control architecture is also very hard to come by. Many manufacturers are unwilling to publish internal details regarding system architecture due to the high levels of competition in the robot market. Consequently, it is not possible to fully exploit many robotic platforms in a scientific context.

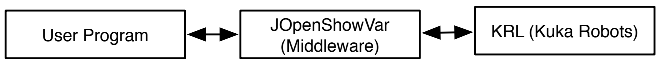


Fig. 1: The idea of realising a communication interface for *Kuka* industrial robots that works as a *middleware* between the user program and the *Kuka Robot Language* (KRL).

Only a small number of industrial manipulators with an open control interface has been released as far as robotics is concerned. Focusing exclusively on *Kuka* industrial robots [3], the *Kuka Robot Language* (KRL) is the standard programming language [4]. It is a text based language that offers data type declaration, specification of simple motions, and interaction with tools and sensors via Input/Output (I/O) operations. It is only possible to run KRL programs on the *Kuka Robot Controller* (KRC), where program execution is done in accordance with real-time constraints. While the KRL offers an interface that is easy to use in industrial applications, it is quite limited for research purposes. In particular, the KRL is tailored to the underlying controller and consequently, only a fixed, controller-specific set of instructions is offered [5]. Advanced mathematical tools such as matrix operations, optimisation or filtering methods are not supported, thus making the implementation of novel control approaches very difficult. There is no native way to include third party libraries and as such, extending the KRL to include new instructions and functionalities is an arduous task. Moreover, it is not possible to directly use external input devices. The standard workaround for partially expanding the robot's capabilities is to use supplementary software packages provided by *Kuka*. Some examples of such packages are the *Kuka.RobotSensorInterface* [6], which allows the manipulator motion or program execution to be influenced by sensor data, and the *Kuka.Ethernet KRL XML* [6], a module that allows the connection of the robot controller with up to nine external systems (e.g. sensors). However, several drawbacks accompany these supplementary software packages: I/O is limited, a narrow set of functions is present and major capital investments are often required to actually purchase these packages from *Kuka*.

To overcome these problems, *JOpenShowVar* was presented by our research group in [7]. *JOpenShowVar* is a Java open-source cross-platform communication interface that allows for reading and writing all the controlled manipulator variables. Even though only soft real-time applications can be implemented, this interface allows researchers to use different input devices, sensors and to develop alternative control methods. *JOpenShowVar* library is compatible with all *Kuka* industrial robots that use *KR C*4 or *KR C*2. The basic concept is shown in Fig. 1: *JOpenShowVar* works as a *middleware* between the user program and the KRL. In this work, more details about *JOpenShowVar* architecture are provided. Several new, more flexible and efficient, procedures are introduced in the latest release of the library to replace the old fundamental reading and writing method that is now marked as deprecated. In addition to these new methods, some other high-level functions are also provided to enable angles and torques readings of the controlled manipulator. This feedback signal is very important to improve the manipulator dexterity and to achieve crucial functions like sensitive collision detection and compliant control actions. Some guidelines for allowing the user implementing new high-level procedures are discussed. *JOpenShowVar* is an open-source project and it is available on the Internet at `https://github.com/aauc-mechlab/jopenshowvar`, along with several detailed class diagrams, documentation and demo videos.

The paper is organised as follows. A review of the related research work is given in Section II. In Section III, we focus on the description of *JOpenShowVar* architecture, analysing the communication protocol, possible control approaches and some high-level methods. In Section IV, four case studies are presented. The first two case studies are open-loop applications, while the last two case study describe the possibility of implementing closed-loop applications. In the first case study, *JOpenShowVar* is used to control a *Kuka KR 6 R900 SIXX (KR AGILUS)* robot with an *Android* [8] mobile application. In the second case study, the same *Kuka* robot is used to perform a two-dimensional line-following task that can be used for applications like advanced welding operations and similar. In the third case study, the same manipulator is controlled in a closed-loop mode with a *Leap Motion Controller* [9] that supports hand and finger motions as input without requiring contact or touching. Finally, in the fourth case study, a bidirectional closed-loop coupling between a *Force Dimension omega.7* haptic device and the same *Kuka* robot is established. A force feedback proportional to the force that the robot's end-effector is supporting is returned by the haptic interface. Related experiments and results are shown in Section V. In Section VI, conclusions and future works are outlined.

TABLE I: Currently available interfaces for *Kuka* robots

Interface	Support to Kuka LBR	Support to Kuka industrial robots	External packages required
OpenKC	Yes	No	Yes
FRI	Yes	No	No
KCT	No	Yes (only small and low-payload)	Yes
Robotics APIs	Yes	Yes (safety limitations)	No
ROS	Yes	No	No
KUKASunrise.Connectivity	Yes	No	Yes

II. Related Research Work

The possibility of creating a software interface to *Kuka* robots has been investigated by several research groups. An open-source real-time control software for the *Kuka lightweight* robot, *OpenKC*, was presented in [1]. This software makes it possible to externally trigger and control all of the features of the *Kuka lightweight* (LBR) manipulator. This is done by using a simple set of routines that can easily be integrated into existing software. As a result, developers of robot applications have an edge in finding solutions for a variety of different software scenarios. In particular, force and torque readings as well as different modes of operation can be remotely read and parametrised. However, this software interface is restricted to a specific model of *Kuka* robots, the *Kuka lightweight* manipulator, and use of the *Kuka.RobotSensorInterface* package is required. Another interface that is currently available for the *Kuka lightweight* robots is the *Fast Research Interface* (FRI) [2]. The FRI provides direct low-level real-time access to the KRC at high rates of up to 1 kHz. On the other hand, all features, like teaching, motion script features, fieldbus I/O and safety are provided. The FRI is based on the *KR C*2. Without much installation efforts, access to different controller interfaces of the *Kuka* system is provided including joint position control, cartesian impedance control, and joint position control. However, also this software interface is restricted to a specific model of *Kuka* robots, the *Kuka lightweight* manipulator. No support for the standard *Kuka* industrial robots is provided.

Later on, the *Kuka Control Toolbox* (KCT), a collection of *MATLAB* functions for motion control of *Kuka* industrial robots, was introduced in [10] to offer an intuitive and high-level programming interface for the user. This toolbox is compatible with all small and low-payload *Kuka* robots that have six degrees of freedom (DOFs). The KCT runs on a remote computer connected to the KRC via TCP/IP. A multi-thread server runs on the KRC and communicates via *Kuka.Ethernet KRL XML* with a client whose job is to manage the information exchange with the manipulator. High transmission rates are guaranteed by this communication set-up, thus enabling real-time control applications. Nonetheless, as in the previous work, this approach is still tailored to the underlying controller and requires the use of the *Kuka.Ethernet KRL XML* package.

A different approach has been tried by other researchers, aimed at the disclosure of the *Kuka* industrial manipulator internal control architecture. For instance, the reverse engineering of the KRL was investigated in [5] and a set of Java-based *Robotics APIs* was presented for programming industrial robots on top of a general-purpose language. The *Robotics APIs* implement robot commands like motions and access to I/O calls. It was shown that KRL can be bridged by batch-executing motions, under the assumption that executing control flow and calculation statements takes only a small amount of time compared to the time it takes the robot to complete a motion command. However, some safety limitations are inherently present in the *Robotics APIs* set because it is the result of a reverse engineering approach and therefore does not include a way of specifying complex triggers in contrast to the KRL.

In the last few years, the Robot Operating System (ROS) [11], an open-source software toolbox for robotic development, has become more and more popular among the research community. The primary goal of ROS is to provide a common platform to make the construction of capable robotic applications quicker and easier. Some of the features it provides include hardware abstraction, device drivers, message-passing and package management. ROS provides support for different industrial robots including vendors like ABB, Adept, Fanuc, Motoman and Universal Robots. Extensive research work has also gone into creating ROS packages for communicating with the *Kuka lightweight* robots but no support is provided for the *Kuka* standard industrial robots yet. One of the main reasons for this lack is the non-disclosure of the KUKA Robot Controller (KRC) internal architecture which currently makes it impossible to directly interact with the robot to be controlled.

Recently, *Kuka* has shown more interest in the research and education market. In particular, the *KUKA Sunrise.Connectivity* has been recently developed for *Kuka lightweight* robots. This software provides a collection of interfaces for influencing robot motion at various process control levels. Third-party software can be easily integrated into the user-specific application using the popular standard programming language Java. Along with the quick update of the target position directly from the robot application, it is also possible to access the robot controller from external computers in hard real-time mode. However, even in this last case, the main limitation is that this software is restricted to the *Kuka lightweight* manipulators.

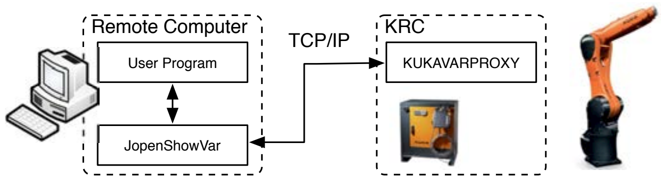


Fig. 2: The proposed architecture for *JOpenShowVar*: a client-server model is adopted.

To provide a more clear overview of the currently available interfaces for *Kuka* robots, a table of comparison of all the reviewed related works is shown in Table I.

To the best of our knowledge, a cross-platform communication interface that works with all *Kuka* industrial robots without requiring any external packages has not been released yet.

III. *JOpenShowVar* ARCHITECTURE

In this section, the authors initially describe the design choices that characterise the proposed architecture. Successively, the architectural concept is presented, analysing the communication protocol, possible control approaches and some high-level methods.

The design of *JOpenShowVar* is based on the following design choices:

- Low-cost: the developed architecture does not requires any supplementary software packages provided by *Kuka* such as the *Kuka.RobotSensorInterface* [6] or the *Kuka.Ethernet KRL XML* [6]. Therefore, no major capital investments are required to actually purchase these packages from *Kuka*. This fact makes the proposed solution very inexpensive;
- Flexibility: the system offers a virtually unlimited I/O and the possibility of including third party libraries. This allows for adding support for advanced mathematical tools such as matrix operations, optimisation or filtering methods, thus making it very simple to implement novel control approaches;
- Reliability: the system is easy to maintain, modify and expand by adding new components and features. In addition the proposed interface is also open-source and cross-platform;
- Integrability: the proposed system interface presents a modular structure that can facilitate the integration with ROS. Even though this integration is outside the scope of this journal paper, it is considered as an important future work which will surely improve the usefulness of the proposed interface. The community of developers at ROS is looking forward to the integration of JOpenShowVar. The developers have confirmed the usefulness

of the proposed interface especially because there are currently no other alternative offering similar features.

Hereafter, several specific functions, variables and configurations related to the KRL and the KRC are referred to in order to introduce the architectural concept. For a more detailed introduction to the KRL, the reader can refer to [4]. The proposed control system architecture is shown in Fig. 2. It is a client-server architecture with *JOpenShowVar* running as a client on a remote computer and *KUKAVARPROXY* acting as a server on the KRC. *JOpenShowVar* locally interacts with the user program and remotely communicates with the *KUKAVARPROXY* server via *TCP/IP*.

In particular, *KUKAVARPROXY* is a multi-client server that is written in Visual Basic 6.0 and can serve up to 10 clients simultaneously. *KUKAVARPROXY* implements the *Kuka CrossComm* interface. 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, errors and faults detection, renaming program files, saving programs, resetting I/O drivers, reading variables and writing variables. *KUKAVARPROXY* implements the reading and writing methods. All the variables that need to be accessed by these methods have to be declared as global variables in the predefined global system data list $CONFIG.DAT. All kinds of variables can be declared in this file from basic types such as INT, BOOL and REAL to more complex structures like E6POS and E6AXIS that allow for storing the robot configuration. Moreover, several system variables can be accessed provided there are no restrictions due to the type of data such as for $PRO_IP, $POS_ACT, $AXIS_ACT or $AXIS_INC. For example, the current robot position, $POS_ACT, cannot be written but only read. Restrictions of this nature are checked by the controller.

As already mentioned, the interface of the *Kuka CrossComm* class allows for the interaction with the real-time control process of the robot to be controlled. However, it should be noted that the *Kuka CrossComm* class can only be remotely accessed via *TCP/IP*. Unfortunately, the *TCP/IP* communication introduces inevitable delays, therefore *JOpenShowVar* cannot provide a real-time access to the robot's data. Only soft real-time applications can be realised. In fact, it takes a non-deterministic time to access a specific variable. Since *Kuka* does not offer any kind of documentation on this topic, several experimental tests were performed at our laboratory to asses this time interval. According to our experiments, reported in Section V, the average access time is about 5 ms. Moreover, this time interval is not affected by the kind of access to be performed (whether it is a reading or a writing operation) or by the length of the message. For these reasons, it is advantageous to aggregate several variables in logical structures when reading or writing data. By using data structures it is possible to

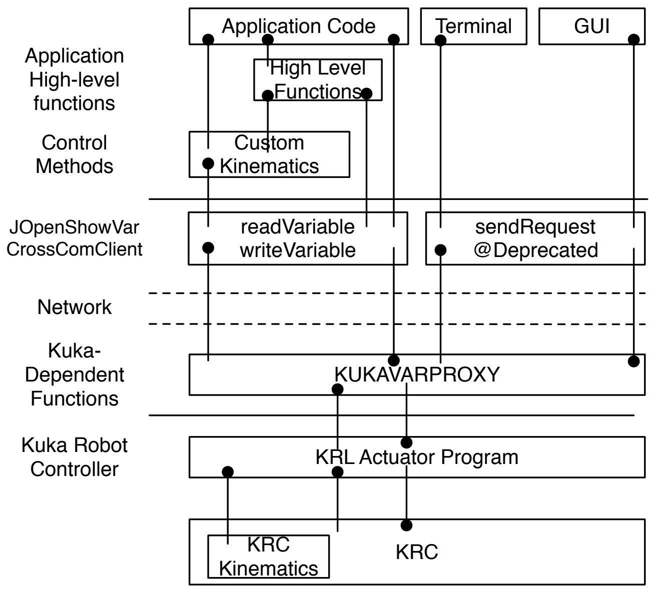


Fig. 3: The architectural levels of *JOpenShowVar*.

simultaneously access several variables, thereby minimising the access time. The only limitation to this approach is on the length of the logical structures that cannot exceed 255 bytes.

JOpenShowVar provides a client, *CrossComClient*, which is written in Java, thus making cross-platform support possible. The architectural details of the *JOpenShowVar* library are shown in Fig. 3. As presented in our previous work [7], the client initially provided only one low level method, *sendRequest*. This method allows for both reading and writing variables. The *sendRequest* method returns a *Callback* instance containing the updated value. However, in the latest release of *JOpenShowVar*, starting from version *v0.2*, the *sendRequest* is marked as a deprecated method, since two new more flexible and efficient methods are introduced: *readVariable* and *writeVariable*.

On top of the *JOpenShowVar*'s methods that implement the low level communication protocol, another logic layer can be added by the user developer allowing for the possibility of implementing alternative control methods (custom kinematics) as well as some higher level functions. The application code can run on top of this hierarchical architecture. In addition a graphical user interface (GUI) and a terminal are provided with *JOpenShowVar* to allow the user for monitoring the robot's state, visualising and manually setting all the desired variables. It should be noted that the GUI still uses the *sendRequest* method of *JOpenShowVar* for a practical reason, since this old method does not require any knowledge on the internal structure of the variables to be accessed compared to the new methods. The low-level communication protocol, a detailed reference explanation of the newly released methods, the possibility of implementing custom control functions, as well as some guidelines to develop high-level procedures will

TABLE II: Reading variables

Field	Description
00	message ID
09	length of the next segment
0	type of desired function
07	length of the next segment
$OV_PRO	Variable to be read

TABLE III: Writing variables

Field	Description
00	message ID
0b	length of the next segment
1	type of desired function
09	length of the next segment
$OV_PRO	Variable to be written
50	value to be written

be discussed later in this section.

A. Communication protocol

The communication protocol relies on the TCP/IP protocol. In particular, on top of the TCP/IP layer, specially formatted text strings are used for message exchanges. *KUKAVARPROXY* actively listens on TCP port 7000. Once the connection is established, the server is ready to receive any reading or writing request from the client.

Reading variables: To access a variable, the client must specify two parameters in the message: the desired type of function and the variable name. To read a specific variable, the type of function must be identified by the character "0". For instance, if the variable to be read is the system variable $OV_PRO, which is used to override the speed of the robot, the message that the client has to send to the server will have the format shown in Table II. In detail, the first two characters of this string specify the message identifier (ID) with a progressive integer number between 00 and 99. The answer from the server will contain the same ID so that it is always possible to associate the corresponding answer to each request even if the feedback from the server is delayed. The next two characters in the string specify the length of the next segment in hexadecimal units. In this specific case, 09 accounts for one character specifying the function type, two characters indicating the length of the next segment and seven characters for the variable length. The fifth character 0 in the message represents the type of the desired function, which in this case is reading. Subsequently, there are two more characters indicating the variable length (in hexadecimal units) and finally the variable itself is contained in the last section of the message.

Writing variables: To write a specific variable, three parameters must be specified: the type of function, the name of the desired variable and the value to be assigned. The writing function is specified by the character "1". For instance, if the variable to be written is the system variable $OV_PRO with a value of 50 (50% override speed), the message that the client has to send to the server will have the format shown in Table III.

B. Variables, structures and methods

From the release version *v0.2* of *JOpenShowVar*, several new classes have been added to the library to improve the usability. In particular, two abstract classes, *KRLVariable* and *KRLStruct* (which extends *KRLVariable*), are provided to allow the user to implement any KRL variable or structure, respectively. In this way, it is possible to create and maintain a local instance of all the desired variables and structures on the client side. Based on these two abstract classes, the most commonly used KRL variables and structures have been implemented. Any other KRL variable or structure that is not included in *JOpenShowVar* library yet can be easily implemented by the user.

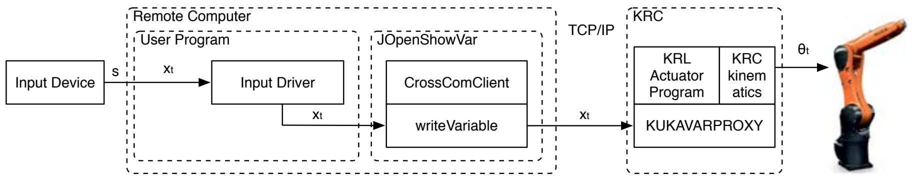
Since the release version *v0.2* of *JOpenShowVar*, the *sendRequest* is marked as a deprecated method. To replace this old method, two new, more reliable methods, are added to the *CrossComClient*:

- the *readVariable* method allows for reading any desired remote variable or structure from the controlled robot and store it locally. An exception is thrown if an error in the communication protocol occurs;
- the *writeVariable* method allows for updating any desired remote variable or structure of the controlled robot with the value of the corresponding local variable or structure, respectively. An exception is thrown if an error in the communication protocol occurs.

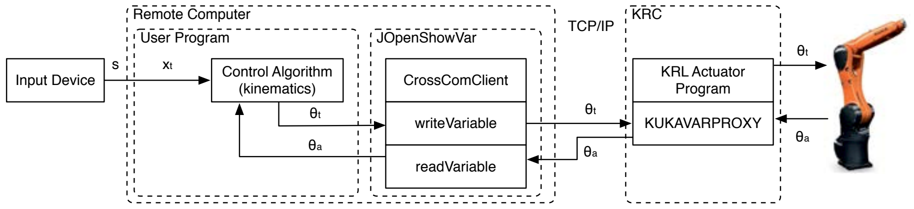
The deprecated *sendRequest* method is being kept as part of the *JOpenShowVar* library simply because the GUI still uses it for a practical reason. In fact, this old method does not require any knowledge on the internal structure of the variables to be accessed compared to the newly introduced methods. Moreover, it should be noted that the new method *writeVariable* cannot handle arrays; this can only be done by using the old *sendRequest* method. In the Algorithm 1 sketch box, a possible use-case example is shown to highlight the differences between the new methods and the deprecated *sendRequest* method.

C. Control methods

JOpenShowVar opens up to a variety of possible applications making it possible to use different input devices and to develop alternative control methods. In particular, the proposed interface provides the possibility of implementing either a position or a velocity control approach. The user experience is substantially different in each case. When using the position control mode, the operator simply controls the position of the robot's end-effector with constant velocity; when operating in velocity control mode, the operator also sets the velocity of the robot tool. In the first case, when the



(a)



(b)

Fig. 4: (a) The user program utilises *JOpenShowVar* to set the desired end-effector position and then the robot joints are calculated by the KRC using the standard kinematic model, (b) a custom control algorithm can be implemented by the user to calculate the joint values for the robot and then send these angles to the KRC to be actuated.

```
try (CrossComClient client = new CrossComClient("
    robotIPaddress", 7000)) {
  //JOpenShowVar v0.1 reading
  Callback readRequest = client.sendRequest(new
      Request(0, "$OV_JOG"));
  System.out.println(readRequest);
  //JOpenShowVar v0.1 writing
  Callback writeRequest = client.sendRequest(new
      Request(1, "$OV_JOG", "100"));
  System.out.println(writeRequest);
  //JOpenShowVar v0.2 reading
  KRLReal jog = KRLVariable.OV_JOG();
  client.readVariable(jog);
  System.out.println(jog);
  //JOpenShowVar v0.2 writing
  jog.setValue(10);
  client.writeVariable(jog);
  System.out.println(jog);}
```

Algorithm 1: A use-case example that highlights the differences between the new methods and the deprecated *sendRequest* method.

operator releases the input device, the end-effector moves back to its starting point, while in the second scenario, the arm just stops moving but it keeps the last given position.

To control the robot motion according to the desired operational scenario, *JOpenShowVar* allows researchers to use the standard kinematics provided with the KRC. However, it is also possible to implement alternative control algorithms according to current needs. This is illustrated in Fig. 4-a and in Fig. 4-b, respectively. It should be noted that the KRL does not provide a native way to obtain velocity control. When using the KRC kinematics, this limitation can be overcome by expressing the target position as:

$$\mathbf{x}_t = \mathbf{x}_d, \tag{1}$$

if operating in position control mode, or by:

$$\mathbf{x}_t = \mathbf{x}_a + \dot{\mathbf{x}}_d \Delta t, \tag{2}$$

if operating in velocity control mode, where Δt is the estimated time interval between two successive iterations. As already mentioned, *JOpenShowVar* cannot provide a real-time access to the robot's data. Only soft real-time applications can be realised. It takes a non-deterministic time to access a specific variable. According to our experiments, reported in Section V, the average access time is about 5 ms. Therefore Δt can be approximated to a slightly bigger factor of the the average access time. To achieve better performance, the average access time should be continuously monitored and updated. Perhaps, this may be a price to high to pay for some applications with real-time requirements but *JOpenShowVar* still provides great advantages in terms of flexibility.

Alternatively, when a custom control algorithm is needed, the target joint configuration is given by:

$$\theta_t = \theta_d, \tag{3}$$

if operating in position control mode, or by:

$$\theta_t = \theta_a + \dot{\theta}_d \Delta t, \tag{4}$$

if operating in velocity control mode.

When the operator manoeuvres the manipulator, a vector signal with no semantic, $\mathbf{s}$, is sent from the input device to the user program. Here, according to the operational scenario, the vector signal is interpreted as the target position $\mathbf{x}_t$. If the intent is to use the standard kinematics provided with the KRC, the user program simply works as a driver for the input device and uses the *writeVariable* method of *JOpenShowVar* to forward $\mathbf{x}_t$ to a KRL program where the standard KRC kinematics is used to calculate the joint angles θ_d. Alternatively, a custom control algorithm can be implemented within the user program to calculate the joint values for the robot according to $\mathbf{x}_t$. Essentially, the custom control method has to implement classic inverse kinematic functions that can be generalised as follows:

$$\theta_d = f_p^{-1}(\mathbf{x}_d), \tag{5}$$

concerning position control, and

$$\dot{\theta}_d = f_v^{-1}(\theta_a, \dot{\mathbf{x}}_d), \tag{6}$$

for velocity control, where θ_a is the the actual joint angles vector that can be retrieved by using the *readVariable* method of *JOpenShowVar*. These values are then forwarded to a KRL program where the standard KRC functions are used to actuate the robot.

Note that the possibility of implementing certain control features does not influence the design for the presented interface. Instead, *JOpenShowVar* extends the functionalities of the KRL language.

D. Additional functions

To simplify the low level communication protocol and improve reliability, some additional methods are provided with the *CrossComClient* class:

- the *simpleRead* and the *simpleWrite* methods are simpler versions of the *sendRequest* function. In particular, these methods do not execute any data parsing as opposed to the *sendRequest* method. They allow for an easier and faster access, as shown in the Algorithm 2 sketch box. The two new methods return a raw string without parsing the information. The aim of these two new methods is to provide an easy way to monitor the status of the robot making it possible to print the raw information returned from the KRC;
- the *readJointAngles* method uses the *readVariable* method retrieve the actual joint angles vector, θ_a, of the controlled robot, all at once;
- the *readJointTorques* method allows for monitoring the load of each joint actuator by retrieving the current torque of each axis of the arm, all at once. In particular, *readJointTorques* retrieves the global KRL array variable \$TORQUE_AXIS_ACT and returns the current torque of each axis. This feedback signal is very important in order to improve manipulator dexterity and to achieve crucial functions like sensitive collision detection and compliant control actions. In the Algorithm 3 sketch box, a possible use-case is shown.

In addition to these methods, some other high-level functions can be implemented by the user on top of the *JOpenShowVar* communication protocol. The implementation of some other possible high-level applications is included as a technical document in the public repository of *JOpenShowVar*.

```
try (CrossComClient client = new CrossComClient("
    robotIPaddress", 7000)) {
  System.out.println(client.simpleRead("$OV_JOG"));
  System.out.println(client.simpleWrite("$OV_JOG",
      "90"));}
```

Algorithm 2: Use-case for the new *simpleRead* and *simpleWrite* methods.

```
try (CrossComClient client = new CrossComClient("
    robotIPaddress", 7000)) {
  double[] torques = client.readJointTorques();}
```

Algorithm 3: Reading the actual torque for each axis, Java side.

E. Terminal and Graphical user interface

Another useful tool that comes with *JOpenShowVar* is a console terminal that provides read-write text-based access to the robot's data. It is particularly useful for system administration and debugging purposes. To read a variable, it is sufficient to type the name of the desired variable and press enter. From an implementation point of view, it uses the new *simpleRead* and *simpleWrite* methods. Fig. 5-a shows a simple use-case.

Besides, *JOpenShowVar* also offers a useful GUI that can be used to monitor the robot's state, visualise and manually set variables. A screen shot of this convenient tool is shown in Fig. 5-b. This interface is very intuitive for the user.

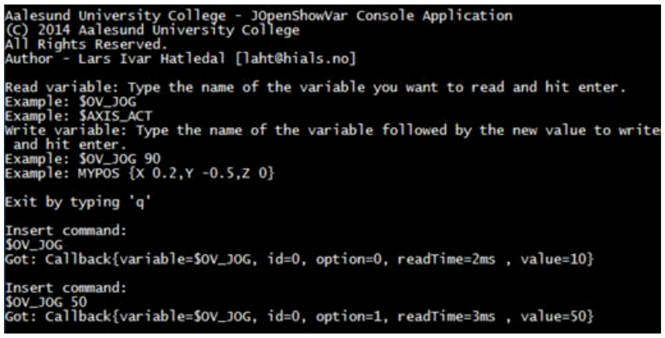
IV. Case studies

In this section, four case studies are presented to demonstrate the potential of *JOpenShowVar*. The first two case studies are open-loop applications, while the last two case studies describe the possibility of implementing closed-loop applications.

A. Case study 1: controlling the Kuka KR 6 R900 SIXX manipulator with an Android mobile device

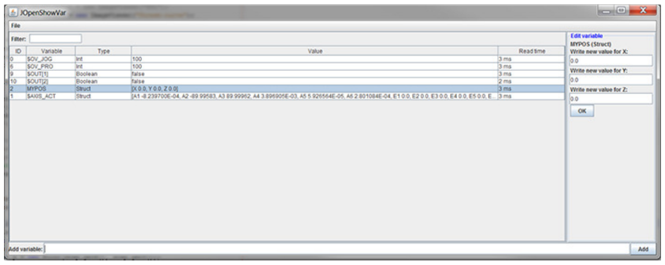
To show the potential of the presented interface in controlling a *Kuka* industrial robot from an alternative input device, as a first case study, *JOpenShowVar* is used to control a *Kuka KR 6 R900 SIXX* manipulator with an *Android* mobile device. In this case, an open-loop application is implemented by using the standard kinematics provided with the KRC. The *Kuka KR 6 R900 SIXX*, shown in Fig. 6-a, is a 6 DOFs robotic arm with a slim design and a small footprint.

According to the operational scenario, an *Android* mobile application whose Graphic User Interface (GUI) is shown in Fig. 6-b, is used to set the target position $\mathbf{x}_t$. By using the *writeVariable* method of *JOpenShowVar* this vector is forwarded to the *KUKAVARPROXY* and stored as a global value in a data structure. Finally, a KRL actuator program iteratively retrieves the new global data and uses the KRC kinematics to actuate the robot. The code of the KRL actuator program is shown in the Algorithm 4 sketch box. For *Kuka* industrial robots, the idle time between motions



```
Aalesund University College - JOpenShowVar Console Application
(C) 2014 Aalesund University College
All Rights Reserved.
Author - Lars Ivar Hatledal [laht@hials.no]

Read variable: Type the name of the variable you want to read and hit enter.
Example: $OV_JOG
Example: $AXIS_ACT
Write variable: Type the name of the variable followed by the new value to write
 and hit enter.
Example: $OV_JOG 90
Example: MYPOS {X 0.2,Y -0.5,Z 0}

Exit by typing 'q'

Insert command:
$OV_JOG
Got: Callback{variable=$OV_JOG, id=0, option=0, readTime=2ms , value=10}

Insert command:
$OV_JOG 50
Got: Callback{variable=$OV_JOG, id=0, option=1, readTime=3ms , value=50}
```

(a)



JOpenShowVar

File

Filter:

ID	Variable	Type	Value	Read time
0	$OV_JOG	Int	100	3 ms
6	$OV_PRO	Int	100	3 ms
9	$OUT[1]	Boolean	false	3 ms
10	$OUT[2]	Boolean	false	2 ms
2	MYPOS	Struct	[X 0.0, Y 0.0, Z 0.0]	3 ms
1	$AXIS_ACT	Struct	[A1 -8.239700E-04, A2 -89.99583, A3 89.99962, A4 3.896905E-03, A5 5.926564E-05, A6 2.801084E-04, E1 0.0, E2 0.0, E3 0.0, E4 0.0, E5 0.0, E...	3 ms

Edit variable
MYPOS (Struct)
Write new value for X: 0.0
Write new value for Y: 0.0
Write new value for Z: 0.0
OK

Add variable: Add

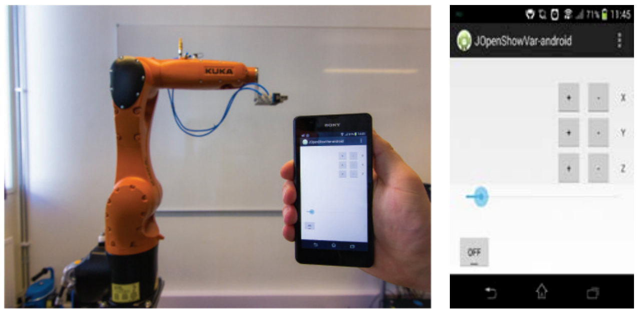
(b)

Fig. 5: (a) *JOpenShowVar* terminal can be used for debugging purposes, (b) *JOpenShowVar* GUI can be used for monitoring the robot's state, visualise and manually set variables and structures.

```
DEF ACTUATOR()
  INI
  PTP HOME Vel = 100 % DEFAULT
  $ADVANCE=1
  LOOP
    PTP_REL MYPOS C_PTP
  ENDLOOP
  PTP HOME Vel = 100 % DEFAULT
END
```

Algorithm 4: KRL actuator program for the case study 1.

can be shortened by executing the time-consuming arithmetic and logic instructions between motion commands while the robot is moving, i.e. processing them during the advance run (the instructions "run" in "advance" of the motion). Using the system variable $ADVANCE, it is possible to define the maximum number of motion blocks that the advance run can process ahead of the main run (the motion block currently being executed). Since the main loop of the Server program consists of only one instruction, the system variable $ADVANCE is initially set to 1 to avoid the unwanted execution of the same line of code. Inside the main loop, a relative movement is iteratively executed to the global variable MYPOS, which is the one that stores the target position. The key word C_PTP is used to approximate the movement. The approximate positioning instruction is executed in a time-optimised manner: there is always at least one axis moving with the programmed acceleration or velocity limits. The



(a) (b)

Fig. 6: Case study 1: (a) the *Kuka KR 6 R900 SIXX* manipulator, (b) the GUI of the Android mobile application used to control the arm.

system simultaneously ensures that the permissible gear and motor torques for each axis are not exceeded. Furthermore, the higher motion profile, set by default, ensures motion that is optimised in terms of velocity and acceleration.

B. Case study 2: a two-dimensional line-following task with the Kuka KR 6 R900 SIXX manipulator

In this case study, *JOpenShowVar* is adopted to perform a two-dimensional line-following task with the same *Kuka* robot used in the previous example. In this case, an open-loop off-line application is implemented by using the standard kinematics provided with the KRC. The considered task can be used for applications like advanced welding operations and similar. In this experiment, a camera is mounted on the robot's end-effector and can capture a photo of the desired line to be followed on a plane. This vision feedback is used for the off-line detection of the path before starting the movement. In particular, once a photo of the line to be followed is taken, the operator manually selects the desired initial and final points. Then, the *A* search algorithm* [12] is used to efficiently find a traversable path between these two points within the region covered by the desired line. The detected traversable path is sampled with a predefined resolution and the resulting samples are stored in an array variable. The same array is used as an off-line input for the robot's end-effector to be actuated point by point. The experiment setup is shown in Fig. 7.

C. Case study 3: controlling the Kuka KR 6 R900 SIXX manipulator with a Leap Motion Controller

In this case study, *JOpenShowVar* is used to control the same robot (from the previous case studies) in a closed-loop and with a custom control algorithm. This is done to highlight the potential of the presented interface in developing alternative control methods that do not use the standard kinematic model provided by *Kuka*. Moreover, a *Leap Motion Controller* [9], shown in Fig. 8, is used as alternative input device to control the robot. The *Leap Motion*

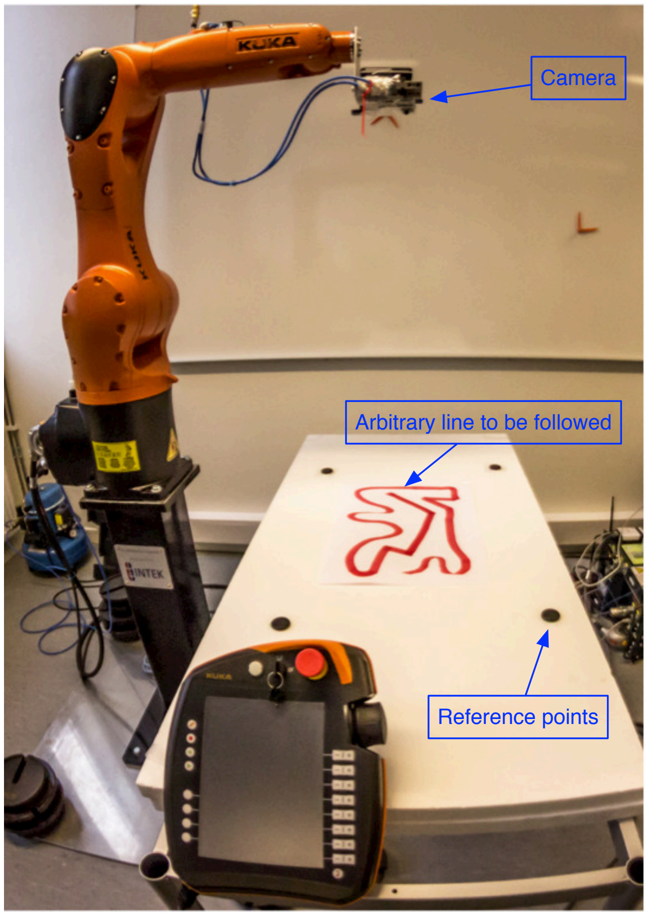


Fig. 7: Case study 2: the experiment setup for a two-dimensional line-following task with the *Kuka KR 6 R900 SIXX* manipulator.

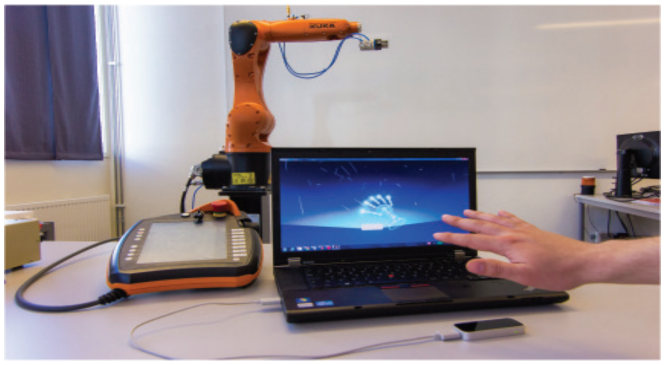


Fig. 8: Case study 3: the *Leap Motion Controller* used to operate the *Kuka KR 6 R900 SIXX* manipulator.

Controller is a small USB input device that supports hand and finger motions as input without requiring contact or touching. This controller is designed to be placed on a physical desktop, facing upwards. Using two monochromatic infra-red (IR) cameras and three IR light-emitting diodes (LEDs), the device observes a roughly hemispherical area, to a distance of about 1 meter. The LEDs generate a 3D pattern of dots of IR light and the cameras generate almost 300 frames per second of reflected data, which is then sent through a USB cable to the host computer, where it is analysed by the *Leap Motion Controller* software and can be retrieved using the *Leap Motion APIs*. While the *Leap Motion Controller* makes it possible to control all the joints of human hands, in this specific case study, only the DOFs of the wrist are used as an input signal to control the robot's end-effector. Each DOF of the wrist corresponds to a translational axis in the workspace of the robot to be controlled. When operating in position control mode, the input device works as a position proportional replica so that the wrist motion maps exactly to the motion of the robot's end-effector with constant speed, while, when operating in velocity control mode, a movement of the wrist in a particular direction will produce a translational motion in the same direction at a velocity proportional to the wrist displacement. In order for small vibrations not to affect the motion of the robot's end-effector, a small spherical imaginary volume with a diameter of about 8 cm is defined in the centre of the controller monitoring space. As long as the operator's wrist is located within this volume, the robot's end-effector does not move. The operator's hand has to be moved more than 4cm from the center of the monitoring space in order to generate a motion. Thanks to the modularity of the architecture, any other joystick or input device can be used without influencing the effectiveness of the proposed interface.

The user program runs on a remote computer and uses the *Leap Motion APIs* to retrieve the target position $\mathbf{x}_t$ according to the operational scenario. By using the *readVariable* method of *JOpenShowVar*, the actual joint angles θ_a are received. This data is used as input for the custom control algorithm. In this specific case study, the classical kinematic functions and the Jacobian method [13] are used to implement (5) and (6). Then, by using the *writeVariable* method of *JOpenShowVar* the target joint configuration θ_t is forwarded to the *KUKAVARPROXY* and stored as a global value in a structure. Finally, a KRL actuator program iteratively retrieves the new global data and actuates the robot.

The code of the KRL actuator program is shown in the Algorithm 5 sketch box. It should be noted that the variable MYAXIS is initialised to default values inside the initialisation (INI) fold. The system variable $ADVANCE is initially set to 1. Then the current joint values are assigned to a local structure variable named LOCAL. Inside the main loop, the desired joint angles are iteratively assigned to LOCAL, axis by axis. Finally, a PTP movement with C_PTP approximation is executed.

D. Case study 4: controlling the Kuka KR 6 R900 SIXX manipulator with a omega.7 haptic device from Force Dimension

The aim of this fourth case study is to show the possibility of operating the robot and transferring the corre-

```
DEF EXT_MOVE_AXIS()
  DECL AXIS LOCAL
  INI
  PTP HOME Vel = 100 % DEFAULT
  $ADVANCE=1
  LOCAL.A1 = $AXIS_ACT.A1
  ...
  LOCAL.A6 = $AXIS_ACT.A6
  LOOP
    LOCAL.A1 = LOCAL.A1 + MYAXIS.A1
    ...
    LOCAL.A6 = LOCAL.A6 + MYAXIS.A6
    PTP LOCAL C_PTP
  ENDLOOP
  PTP HOME Vel = 100 % DEFAULT
END
```

Algorithm 5: KRL actuator program for the case study 3.

sponding force feedback to the operator. For this purpose, a bidirectional coupling between a *Force Dimension omega.7* [14] haptic device and the same *Kuka* robot used in the previous sections is established. In this case, an closed-loop application is implemented by using the standard kinematics provided with the KRC. The *omega.7* is a 7 DOF haptic interface with high precision active grasping capabilities and orientation sensing. Finely tuned to display perfect gravity compensation, the force-feedback gripper offers extraordinary haptic capabilities, enabling instinctive interaction with complex haptic applications. In this case study, the principle of virtual works [13] is applied. According to this principle, the following equation is valid:

$$J^T\mathbf{F} = \tau, \tag{7}$$

where J is the Jacobian matrix of the arm, $\mathbf{F}$ is the vector of forces exerted from the robot's end-effector to the environment and τ is the vector of torques at the joints that can be retrieved by using the *readJointTorques* method. By applying this principle it is possible to simulate on the haptic device a force feedback proportional to the force that the robot's end-effector is supporting. The experiment setup is shown in Fig. 9.

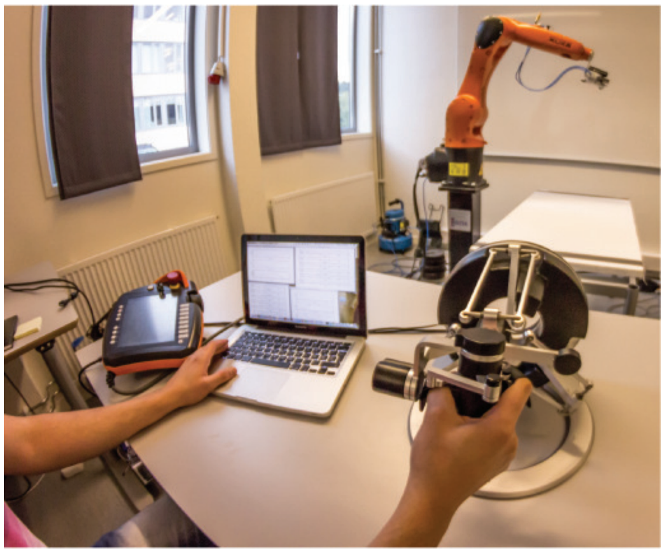


Fig. 9: Case study 4: controlling the *Kuka KR 6 R900 SIXX* manipulator with a *omega.7* haptic device from *Force Dimension*.

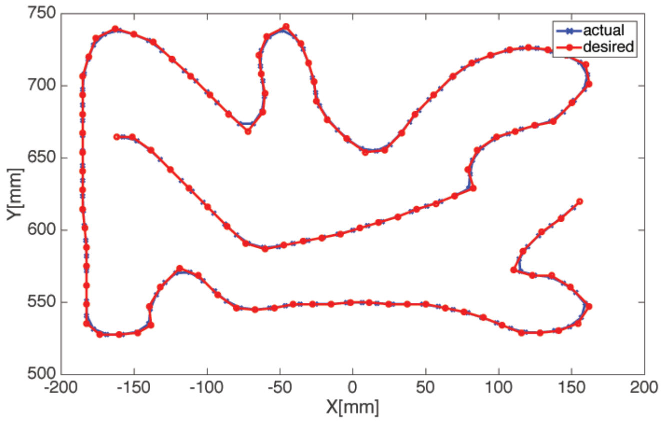


Fig. 10: Case study 2: the detected line and the actual path followed by the robot's end-effector respectively.

V. Experimental Results

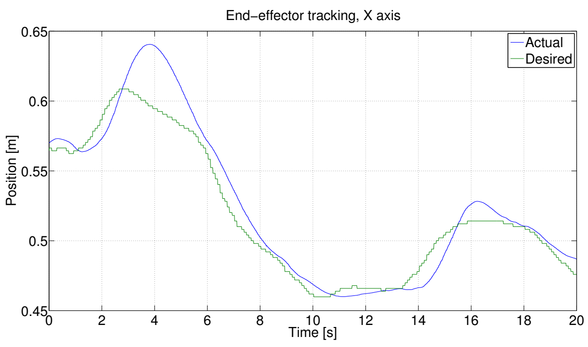
Experiments related to the proposed case studies are carried out to test the proposed communication interface in terms of accuracy, performances and effectiveness.

Concerning the first and the third case studies, a demo video is available on-line at:
`https://youtu.be/fC8jb9MKgGw`.

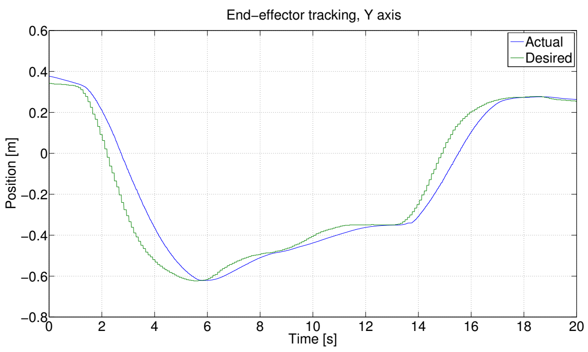
The first case study highlights the potential of *JOpenShowVar* in remotely controlling a *Kuka* industrial robot from an *Android* mobile device. This possibility opens up to a variety of useful purposes including human interface applications and teleoperations. Nowadays, smartphones and tablets are becoming computationally more and more powerful. In this perspective, they are a perfect match with robots for developing alternative control systems. The use of smartphones and tablets in research and development is also found in other areas, as they represent a significant business opportunity for manufacturers, who need to consistently develop better hardware and operating systems. For these reasons, these applications are very interesting and appealing in the forthcoming industrial applications.

A. Accuracy

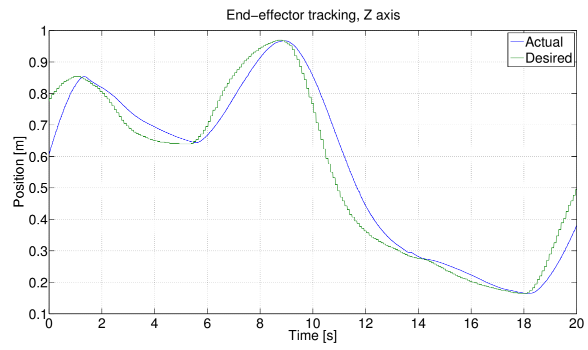
Accuracy refers to the possibility of positioning the robot's end-effector at a desired target point within the workspace. Concerning the second case study, the line-following experiment is performed on a randomly generated line, drawn on a table. Fig. 10 shows the detected line and the actual path followed by the robot's end-effector respectively. Once the line is detected, the robot executes the movement off-line in about 10s with a maximum position error less than 5 mm.



(a)



(b)



(c)

Fig. 11: Case study 3: path tracking for (a) the **X** coordinate, (b) the **Y** coordinate and (c) the **Z** coordinate.

This position error could be reduced even more by increasing the sampling resolution of the detected traversable path.

B. Performances

Within the particular case study of the *Leap Motion Controller* (case study 3), a real-time path tracking analysis of the Cartesian paths for X, Y and Z coordinates is performed, measuring the difference between the desired and actual position of the robot's end-effector. The results are shown in Fig. 11.

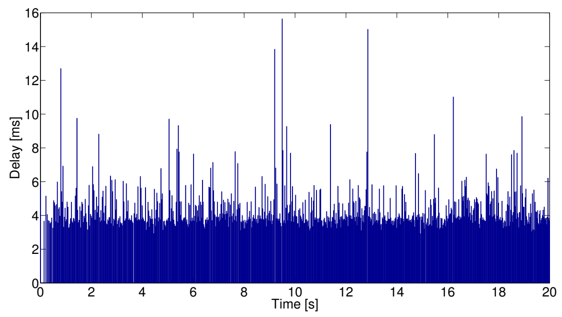


Fig. 12: Case study 3: time-delay analysis for the corresponding Cartesian paths shown in Fig. 11.

Moreover, to assess the communication delay of *JOpenShowVar*, a time-delay analysis is carried out.

The considered time-delay represents the time for each message to be received, performed and notified to the client by the KRC. Particularly, this time-delay is obtained by considering the exact instant in which the request is dispatched from the client and the exact instant in which the information is received back from the client. It is not possible to exactly determine the time-delay mainly because *Kuka* has not released any information about it. During our experiments, a deterioration of the time-delay has usually been noticed when making the selection of a program and when the robot is engaged in some movements or there are several active interrupts. When considering the causes of the delay, it is possible to distinguish two main components that affect the access time for a variable to be either read or written from the client side:

- the time interval that is required for the *TCP/IP* protocol to transfer the information from the client to the server and then back to the client. This time component is non-deterministic;
- the time interval that is required for the *Kuka* controller to acquire the information from the robot. Also this time component is non-deterministic.

As already mentioned in Section III, the time-delay is not affected by the kind of access to be performed (whether it is a reading or a writing operation) or by the length of the message. Therefore, it is beneficial to aggregate several variables in logical structures when reading or writing data. By using data structures it is possible to simultaneously access several variables, thereby minimising the access time.

Considering the third case study, a time-delay analysis is carried out for the same Cartesian paths as shown in Fig. 12. Even though there are a few spikes with a larger time interval, an average access time of 4.27 ms is obtained in this case. It should be noted that all the considered case studies are equally affected by similar communication delays except for the second case study which is performed off-line and therefore not presenting any run-time delays.

To further assess the performances of the proposed inter-

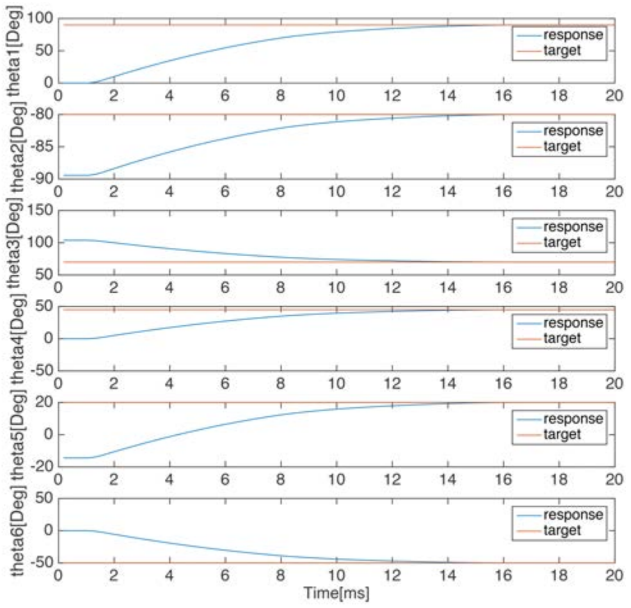


Fig. 13: Target and response of the adopted PID controller for all the joints of the robot.

face with regard to the communication delay, an additional experiment is performed. In particular, the possibility of developing alternative control methods is considered (as presented in case study 3). Any custom control algorithm that does not use the standard KRC kinematics must calculate the corresponding sampling point configurations for the desired end-effectors positions. In other words, each control algorithm works as a motion planner. To ensure smooth movements for the manipulators it is necessary to generate trajectories out of these given sampling points. A well-suited trajectory is the basic prerequisite for the design of a high-performance tracking controller and ensures that no kinematic nor dynamic limits are exceeded. Such a controller guarantees that the controlled robot will follow its specified path without drifting away. Therefore, feedback control has to be applied to be able to compensate external disturbances as well as disturbances from communication time delays. Note that time data is a free parameter because, as already mentioned, the sampling time of the mapping algorithm is not constant. A possible solution for generating well-suited trajectories consists of using a Proportional Integral Derivative (PID) controller for each joint. To tune the PID parameters, different methods can be used, such as the one proposed in [15]. The response of the adopted PID controller is shown in Fig. 13 for all the joints of the robot. The interface provided by *JOpenShowVar* demonstrates a relatively fast reaction to the inputs and reasonable output error for research purposes, considering the dimension of the controlled model.

C. Effectiveness

Concerning the fourth case study, the aim is to show the possibility of operating the robot and transferring the corresponding force feedback to the operator. The plots in Fig. 14-a and Fig. 14-b show the actual position for the X, Y and Z axes as a result of the haptic input device's movements, operated by the user, and the corresponding joint angles, respectively. In this particular case, the operator manoeuvres the robot to lift the end effector up at first, then down again with a displacement also in the X and Y axes. In this case study, the input signal is not scaled to the robot's workspace since the haptic device is only used to set the direction of movement for the robot and to transfer the corresponding force feedback to the operator. Even though there is a delay between the input signal and the actual position, the results show that the system is quite responsive to the user's inputs. Fig. 14-c and Fig. 14-d show the torques applied to the robot's joints and the corresponding forces applied to the robot's end-effector, respectively. The operator also perceives a force feedback that is proportional to forces applied to the robot's end-effector.

VI. Conclusions and Future Work

This paper highlights the features of *JOpenShowVar* as a cross-platform communication interface to *Kuka* industrial robots. Even though *JOpenShowVar* only provides a soft real-time access to the manipulator to be controlled, this *middleware* package opens up to a variety of possible applications making it feasible to use different input devices, sensors and to develop alternative control methods. Special care has been devoted to keep *JOpenShowVar* methods intuitive and easy to use. The versatility and effectiveness of the interface have been demonstrated through four case studies. The first two case studies are open-loop applications, while the last two case studies describe the possibility of implementing closed-loop applications. Recently, our research group employed *JOpenShowVar* to realise a framework that makes it possible to reproduce the challenging operational scenario of controlling offshore cranes via a safe laboratory setup [16].

In the future, different control algorithms such as the ones implemented in [17], [18] and [19] may be tested as alternatives to the standard KRC. Finally, some effort should be put in the standardisation process of *JOpenShowVar* to make it even more reliable for both the industrial and the academic practice. It is the opinion of the authors that the key to maximising the long-term, macroeconomic benefits for the robotics industry and for academic robotics research relies on the closely integrated development of open content, open standards and open source. In this perspective, the integration of the proposed interface with ROS is of crucial importance as a necessary future work. The community of developers at ROS is looking forward to the integration of *JOpenShowVar*.

VII. ACKNOWLEDGMENTS

This work is partly supported by the Research Council of Norway through the Centres of Excellence funding scheme, project number 223254 and the Innovation Programme for Maritime Activities and Offshore Operations, project number 217768.

REFERENCES

[1] M. Schopfer, F. Schmidt, M. Pardowitz, and H. Ritter, "Open source real-time control software for the kuka light weight robot," in *Proc. of the 8th IEEE World Congress on Intelligent Control and Automation (WCICA), Jinan, China*, 2010, pp. 444–449.

[2] G. Schreiber, A. Stemmer, and R. Bischoff, "The fast research interface for the kuka lightweight robot," in *Proc. of the IEEE ICRA Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications How to Modify and Enhance Commercial Controllers*, 2010, pp. 15–21.

[3] KUKA Robotics Corporation. (2014, March) "KUKA". [Online]. Available: http://www.kuka-robotics.com/

[4] KUKA, *Expert Programming*. KUKA Robotics Corporation, 2003.

[5] H. Mühe, A. Angerer, A. Hoffmann, and W. Reif, "On reverse-engineering the kuka robot language," in *1st IEEE/RSJ International Workshop on Domain-Specific Languages and models for Robotic systems DSLRob'10, International Conference on Intelligent Robots and Systems (IROS)*, 2010, pp. 16, 217, 223, 224.

[6] KUKA Robotics Corporation. (2014, March) "KUKA, Hub technologies". [Online]. Available: http://www.kuka-robotics.com/en/products/software/hub_technologies/print/start.htm

[7] F. Sanfilippo, L. I. Hatledal, H. Zhang, M. Fago, and K. Y. Pettersen, "JOpenShowVar: an open-source cross-platform communication interface to kuka robots," in *Proc. of the IEEE International Conference on Information and Automation (ICIA), Hailar, China*, 2014, pp. 1154–1159.

[8] Google Inc. (2014, March) "Android". [Online]. Available: http://www.android.com/

[9] Leap Motion Inc. (2014, March) "The Leap Motion Controller". [Online]. Available: http://www.leapmotion.com/

[10] F. Chinello, S. Scheggi, F. Morbidi, and D. Prattichizzo, "Kuka control toolbox," *IEEE Robotics & Automation Magazine*, vol. 18, no. 4, pp. 69–79, 2011.

[11] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[12] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering route planning algorithms," in *Algorithmics of large and complex networks*. Springer, 2009, pp. 117–139.

[13] B. Siciliano and O. Khatib, *Springer handbook of robotics*. Springer, 2008.

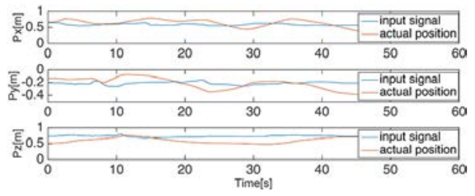
[14] Force dimension. (2014, March) "omega.7". [Online]. Available: http://www.forcedimension.com/

[15] I. Pan, S. Das, and A. Gupta, "Tuning of an optimal fuzzy PID controller with stochastic algorithms for networked control systems with random time delay," *ISA transactions*, vol. 50, no. 1, pp. 28–36, 2011.

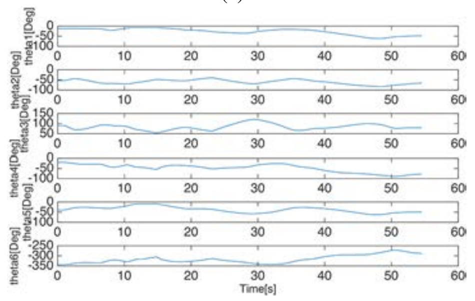
[16] F. Sanfilippo, L. I. Hatledal, H. Zhang, W. Rekdalsbakken, and K. Y. Pettersen, "A wave simulator and active heave compensation framework for demanding offshore crane operations," in *Proc. of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Halifax, Nova Scotia, Canada*, 2015, pp. 1588–1593.

[17] F. Sanfilippo, L. I. Hatledal, H. G. Schaathun, K. Y. Pettersen, and H. Zhang, "A universal control architecture for maritime cranes and robots using genetic algorithms as a possible mapping approach," in *Proc. of the IEEE International Conference on Robotics and Biomimetics (ROBIO), Shenzhen, China*, 2013, pp. 322–327.

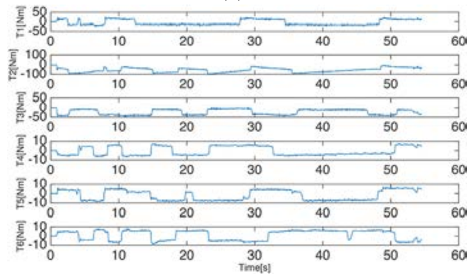
[18] F. Sanfilippo, L. I. Hatledal, H. Zhang, and K. Y. Pettersen, "A mapping approach for controlling different maritime cranes and robots using ANN," in *Proc. of the 2014 IEEE International Conference on Mechatronics and Automation (ICMA), Tianjin, China*, 2014, pp. 594–599.



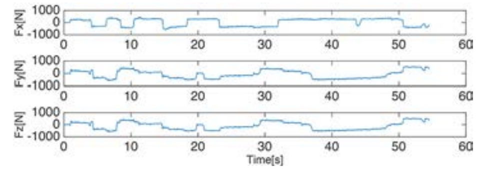
(a)



(b)



(c)



(d)

Fig. 14: Case study 4: (a) actual position as a result of the haptic input device's movements (in this case, the input signal is not scaled to the robot's workspace since the haptic device is only used to set the direction of movement for the robot and to transfer the force feedback to the operator), (b) joint angles, (c) joint torques and (d) forces applied to the robot's end-effector.

[19] L. I. Hatledal, F. Sanfilippo, and H. Zhang, "JIOP: a java intelligent optimisation and machine learning framework," in *Proc. of the 28th European Conference on Modelling and Simulation (ECMS), Brescia, Italy*, 2014, pp. 101–107.