Functional Requirements for integrating a controller in drag&bot

This is a list of all requirements that the robot controller should satisfy for being used by drag&bot. If some of them are not meet, then further discussions are required.

a) Hardware / Communication requirements

drag&bot can provide the IPC or a IPC provided by the robot maker can be used if it accomplish the following requirments:

FRA001	There is a standard PC / Industrial PC available with Ubuntu 16.04 LTS installed as operating system. The PC can be 64 or 32 bits. 64 is desired. **Alternative possible solutions:** - Other Linux operating systems can be used - ARM architecture
FRA002	ROS (Robot Operating System) is running on the PC defined in FRA001. **Additional information:** - d&b system uses ROS as middleware.

This requirements have to be satisfied by the robot controller:

FRA003	There is a data communication channel between robot controller and PC. **Additional information:** - Ideally it should be a TCP/IP connection through Ethernet.
FRA004	It is possible to send and receive data between the robot controller and the PC.
FRA005	Messages can be created / read through a Python / C++ API usable as part of a ROS program. The messages must be as simple and understandable as possible. **Additional information:** - E.g. Socket library in Python / C++. - E.g. IIWA Messages Protocol (see example in Appendix B)
FRA006	Each message triggers an operation in the robot controller such as moving the robot.
FRA007	Controller can communicate status information back to drag&bot

If messages are non-blocking: polling. The robot controller responds so fast as possible to each message without needing to finish the execution of the message. E.g. if the controller receives a move command, then responds “ok” and starts to move. drag&bot knows when the move is finished through the status information of the robot (target position, is it moving or target reached).

FRA101	The communication between the robot controller and the PC is non-blocking. **Additional information:** - Acknowledge of message received is immediately received - A new message is immediately processed without needing to wait that the previous message finished. No queue of messages. - A move of the robot doesn’t block the communication to the driver
FRA102	If a new command arrives to the controller, the controller must respond in milliseconds-range time to the message.
FRA103	If a new command arrives to the controller, the controller will immediately trigger the requested action. The action should begin as fast as possible.
FRA104	Controller must be able to process a polling rate high enough to ensure the user experience and the safety (typically 50 Hz).

If messages are blocking then drag&bot needs a second communication channel in order to stop the robot during a move or continue receiving status information. This is not desired because it makes blending more difficult to implement.

FRA201	There is a separate channel which broadcast robot status information (position, status, etc.)
FRA202	A separate channel is needed for stopping the robot.

Also a combination of both of them can be used (non-blocking + status information streaming channel).

b) Controller functionality

Notice:

drag&bot shouldn’t be the responsible for calculating the inverse kinematics (given a Cartesian position, calculating which joint position corresponds to this Cartesian position). The robot controller should be able to move the robot to a given Cartesian position.

FRC001	Each message triggers an action in the robot controller, such as move the robot or stop the robot, or triggers information retrieval. **Additional information:** - A message can be called command or operation.
FRC002	Move command must exist. The robot shall move to a given position after triggering this command.
FRC003	Stop current move command must exist. The robot shall stop after triggering this command. The stop must be fast enough to ensure the security and a nice user experience.
FRC004	Controller can use a Joint Position (rotation in degrees/radians) as a target position in a move command. **Additional information:** - The position is defined as a list of rotation degrees / distance (for lineal axis) corresponding with the DoF of the robot.
FRC005	Controller can use a Cartesian position as target position in a move command. **Additional information:** - A Cartesian position can be defined as X,Y,Z distances with a rotation which can be specified either in quaternions, intrinsic Euler or extrinsic Euler coordinates. Ideally quaternion or intrinsic Euler ZYX. - Conversion between formats is also possible (e.g. ZYZ -> ZYX)
FRC006	The robot controller can move the robot from current robot position to any target position as long as it is inside the workspace. **Additional information:** - A target position can be as far as needed from the current position. The controller will be able to move the robot to the requested position.
FRC007	Target positions are preferable defined as absolute positions although drag&bot can work with incremental positions by calculating the increments from current robot position and target position.

	Additional information: - In case of incremental positions drag&bot needs a high frequency (at least 50 Hz) robot status retrieval.
FRC008	Controller can PTP moves with Euler position as target in a move command.
FRC009	Controller can LIN moves with Euler position as target in a move command.
FRC010	Controller can PTP moves with Joint position as target in a move command.
FRC011	Optional: Controller can LIN moves with Joint position as target in a move command
FRC012	Optional: Controller can do blending between consecutive move commands.
FRC013	Controller can cancel the current executed move command.
FRC014	Controller can queue several move commands and execute them one after another.
FRC015	Desired: Controller can do blending between queued move commands.
FRC016	Controller can discard all queued move commands.
FRC017	Controller can send the current values of all joints to drag&bot.
FRC018	Controller can send the current position and orientation of the flange to drag&bot.
FRC019	Controller can send the the current movement goal position and orientation to drag&bot. If a trajectory (queued commands) is being executed then the goal position is the current waypoint, not the last one.
FRC020	Optional: Controller can know if a position is physically reachable and send this information to drag&bot.
FRC021	Controller can send information about occurred errors to drag&bot (e.g. emergency stop)
FRC022	Optional: Controller can activate each digital IO separately
FRC023	Optional: Controller can send the status of each digital IO separately to drag&bot.
FRC024	Optional: Controller can send the status of all digital IOs together to drag&bot.
FRC025	Optional: Controller can change the speed during the movement.

FRC026	Optional: Controller can recover from an error by triggering a command.

c) Other requirements

FRO001	There is URDF model of the robot. **Alternative possible solutions:** - There are CAD models of the robot - There are specifications including length of links and rotation / translation limits.

Appendix A1 – ROS Topics provided by d&b drivers

command_list (robot_movement_interface::CommandList.msg)

This topic communicates the robot movement commands to the robot driver.

Header `header`: Timestamp information (see: std_msgs::Header.msg)
Command[] `commands`: List of movement commands (see below)
bool `replace_previous_commands`

- true: Replaces any planned command with current new commands.
- false: Adds the new commands to the command queue.

robot_movement_interface::Command.msg:

This message contains a flexible movement command for robotic drivers.

Header `header`: Timestamp information (see: std_msgs::Header.msg)
uint32 `command_id`: ID of the movement command
string `command_type`: the movement type (LIN, PTP)

string `pose_reference`: Name of the used base frame for the relative positions
EulerFrame `pose_reference_frame`: (optional) The values of the reference frame. If not set, the command will be executed in '/base' frame.

string `pose_type`: type of the movement pose (JOINTS, QUATERNION, EULER_INTRINSIC_ZYX)
float32[] `pose`: the values of the target frame relative to the refrence frame.

string `velocity_type`: type of the velocity values (M/S, RAD/S, %, ...)
float32[] `velocity`: the velocity values

string `acceleration_type`: type of the acceleration values (M/S^2, RAD/S^2, %, ...)
float32[] `acceleration`: the acceleration values

string `effort_type`: type of the effort values (N, NM, ...)
float32[] `effort`: the effort values

string `blending_type`: type of the vlending values (M, RAD, %, ...)
float32[] `blending`: the blending values

string[] `additional_parameters`: list of additional parameter names
float32[] `additional_values`: list of the corresponding values to the additional parameter names

command_result (robot_movement_interface::Result.msg)

This topic communicates the result of the last movement from the robot driver.

Header `header`: Timestamp information (see: std_msgs::Header.msg)
uint32 `command_id`: ID of the corresponding movement command
int32 `result_code`: Result code of the corresponding movement

- 0: no error occured during movement
- !0: error occured during movement

string additional_information: Additional information

robot_status (industrial_msgs::RobotStatus.msg)

This topic communicates the current robot status from the driver.
See industrial_msgs::RobotStatus.msg for further information.

tool_frame (robot_movement_interface::EulerFrame.msg)

This topic communicates the current coordinates of the robot tool frame from the driver. See below for EulerFrame definition.

robot_movement_interface::EulerFrame.msg

Coordinates message of an arbitrary frame in Euler Intrinsic ZYX convention. For orientation alpha is rotated at first in the z-axis (yaw), then beta is rotated in the new y-axis (pitch) and finally gamma is rotated in the new x-axis (roll).

float32 `x`, `y`, `z`: translational coordiantes of the frame in meters
float32 `alpha`, `beta`, `gamma`: rotational coordiantes of the frame in rad

joint_states (sensor_msgs::JointState.msg)

This topic communicates all current joint states for the robot from the driver.
See sensor_msgs::JointState for further information.

current_speed_scale (std_msgs::Float32.msg)

This topic communicates the current speed factor for the robot from the driver.

float32 `data`: The current speed factor from 0.0 to 1.0.

io_states (robot_movement_interface::IOStates.msg)

This topic communicates the current states of inputs and outputs from the driver.

IOState[] `inputs`: Array of all digital input pins and their current value. See below for IOState definition.
IOState[] `outputs`: Array of all digital output pins and their current value. See below for IOState definition.

robot_movement_interface::IOState.msg

Status message for an arbitrary digital input or output.

int32 `pin_number`: The number of the digital I/O pin.
bool `pin_value`: The status of the digital I/O pin.

Appendix A2 – ROS Services provided by d&b drivers

scale_speed (robot_movement_interface::setFloat.srv)

This service scales the maximum speed of the robot in percentage. For example this function can be called to reduce the overall speed to 50%.

```
rosservice call /scale_speed "value: 0.5"
```

Parameters:
float32 `value`: The value to scale the speed with. It can be set to a number between **0.0 and 1.0**. A value smaller than 0.0 will be cropped to 0.01 and a value greater than 1.0 will be cropped to 1.0. When the value is set to **0.0** the robot will stop and cancel the movement commands.

Returns:
nothing

Note: Some robots might stop their current movement to standstill, before accelerating again and continuing the movement with the new speed.

get_io (robot_movement_interface::getIO.srv)

This service reads the value of a digital i/o port and returns it to the user. For example this function can be called to get the value of port 5.

```
rosservice call /get_io "number: 5"
```

Parameters:
int32 `number` is the digital i/o port to get the value for. Valid port numbers depend on the number of ports provided by the driver.

Returns:
bool `value`: The value of the respective port, if no error occured.
int32 `error`: Error number:

- 0 no error occured
- 1 invalid port number

Note: Calling this service during robot movement might stop the movement and cancel the movement commands.

set_io (robot_movement_interface::setIO.srv)

This service sets the value of a digital i/o port. For example this function can be called to set the value of port 5 to true.

```
rosservice call /set_io "number: 5
value: true"
```

Parameters:
int32 `number` is the digital i/o port to set the value for. Valid port numbers depend on the number and type of ports provided by the driver.
bool `value`: The value to set the port to.

Returns:
int32 `error`: Error number:

- 0 no error occured
- 1 invalid port number
- 2 attempted to set an input port

Note: Calling this service during robot movement might stop the movement and cancel the movement commands.

stop_robot_right_now (std_srvs::Trigger.srv)

This service lets the robot stop moving immediately and cancels the following movement commands.

```
rosservice call /stop_robot_right_now
```

Parameters:
none

Returns:
bool `success`: true on successful stopping of the robot movement, false otherwise
string `message`: informational, e.g. for error messages

Note:
see also std_srvs::Trigger.srv

Appendix B - IIWA Message Protocols Example

```
===========================================================================
Structure and functionality
=========================================================================

Basically ROS_driver program is a Telnet-based command-driven TCP server.
A command is a string line (terminated with /r/n, or /n). Each commands
produces a fast response.
Command rate is stable until several thousands of commands per second with
direct ethernet connection.
Produced responses (replies) are also string lines.
Exceptions are catched in order to allow reconnecting in case of error or
collision.

Server logic:
    -   wait until a new command is received through the socket (new line
ending character)
    -   parse the command and parameters
    -   execute the corresponding operation (non-blocking)
    -   send back the reply with the produced arguments or message

Command types:
        -       Status: robot won't move. Command reply include information
such as position.
        -       Move: robot will move

Robot has four different modes:
    -   Normal: moves cannot be cancelled without stopping.
        Allowed rate is insufficient for reactive operations (~100 ms
stopping). In this mode moves are
        internall queued in the controller if move commands are received
faster than processing time.
    -   Impedance: spring-like behaviour.
    -   (Connectivity) Smart: accepts PTP and LIN moves in the whole work
space, also joints. ~50 Hz
    -   (Connectivity) Direct: suitable for reactive operations, but target
point must be very near
        (approx. degree range) otherwise it produces error. ~100 Hz
Changing between modes will stop the robot for about 100 ms.

A detailed available command list is provided in section Protocol. It is
also possible to directly speak with the driver through telnet (telnet
172.31.1.147 30000).


==========================================================================
Protocol
===========================================================================

-       Format

        Each command is represented as a string line (string ended in \n,
\r or \n\r)
        which contains a command id, a : and parameters as space separated
values:
        -       first value: command
        -       separator: :
        -       rest: parameters
        Spaces are trimed and multiple spaces (and also tabs) ignored.
Separator and parameters
        are not necessary for commands which don't require input
parameters.

        Each sent command will produce a space separated values string line
(string ended in \n, \r or \n\r)
        as reply containing the reply parameters.

-       Command list

        All joints (j) are in rad, velocities (v) in percentage (0..1) if
not specified,
        forces and thresholds in Newton, time in seconds, stiffness
constant in N*m, blending in mm.
        Kuka uses Z,Y,X Euler Intrinsic in Rad -> Z * Y * X (alpha = RotZ,
beta = RotY, gamma = RotX).
        4000 Nm is a good stiffness constant.

    hello -> hello // Ack
    bye -> bye // Driver Shutdown
    smart joint move : j0 j1 j2 j3 j4 j5 j6 v0 v1 v2 v3 v4 v5 v6 -> done //
Smart joint move to joints
    direct joint move : j0 j1 j2 j3 j4 j5 j6 v0 v1 v2 v3 v4 v5 v6 -> done
// Direct joint move to joints, must be to target near current joints.
    smart cartesian move : x y z alpha beta gamma speed -> done // Smart
cartesian move at % speed
    direct cartesian move : x y z alpha beta gamma speed -> done // Direct
cartesian move at % speed
    joint move : j0 j1 j2 j3 j4 j5 j6 v0 v1 v2 v3 v4 v5 v6 -> done //
Normal ptp joint move, velocities are in %
    lin move : x y z alpha beta gamma speed blending -> done // Normal
cartesian LIN move to given frame. Speed in %
    ptp move : x y z alpha beta gamma speed blending -> done // Normal
cartesian PTP move to given frame. Speed in %
    linr move : x y z alpha beta gamma speed blending r -> done // Normal
cartesian LIN redundancy move to given frame. Redundancy r is an angle
value in rad.
    ptpr move : x y z alpha beta gamma speed blending r -> done // Normal
cartesian PTP redundancy move to given frame. Redundancy r is an angle
value in rad.
    linforcez move : x y z alpha beta gamma speed blending thresholdZ ->
done // Normal cartesian LIN move with force collision checking in Z (stops
if force exceeds threshold absolute value)
    ptpforcez move : x y z alpha beta gamma speed blending thresholdZ ->
done // Normal cartesian PTP move with force collision checking in Z (stops
if force exceeds threshold absolute value)
    linforce move : x y z alpha beta gamma speed blending thresholdX
trhesholdY thresholdZ -> done // Normal cartesian LIN move with force
collision checking in X Y Z
    ptpforce move : x y z alpha beta gamma speed blending thresholdX
trhesholdY thresholdZ -> done // Normal cartesian PTP move with force
collision checking in X Y Z
    linstiff move : x y z alpha beta gamma speed stiffness fx fy fz tx ty
tz -> done // LIN stifness move to position, f and t are required offset
force and torque. Usually current force is sent after smoothing.
    ptpstiff move : x y z alpha beta gamma speed stiffness fx fy fz tx ty
tz -> done // PTP stifness move to position, f and t are required offset
force and torque. Usually current force is sent after smoothing.
    start gravity : stiffness -> done // Starts a position holder,
experimental
    get tool frame -> x y z alpha beta gamma // Current TCP (tool)
    get flange frame -> x y z alpha beta gamma // Current flange frame
(without tool)
    get status -> moving / stopped // Current robot state
    get cartesian force -> fx fy fz tx ty tz // Force at TCP
    get joint position -> j0 j1 j2 j3 j4 j5 j6 // Joints values request
    get joint torque -> t0 t1 t2 t3 t4 t5 t6 // Joint torque in N*m
    stop -> done // Stops current move (hard stop, 100 ms or some seconds
if impedance was active)
```

Commands return "error" in case of error or "unknown command" if not recognized.

Appendix C – Kuka KRC4 Message Protocols Example

Commands details

I0 is always the msgid
I1 is always the command id

ID tag	ID number	Inputs (Int 0-9, Real 0-9)	Outputs (Int 0-9, Real 0-9)	Description
STO	1	None	None	Stop robot right now
LIN	10	R[0..5] = X,Y,Z,A,B,C in mm and degrees	None	Cartesian LIN move
LIA	15	R[0..5] = A1..A6 in degrees	None	Joint LIN move
PTP	20	R[0..5] = X,Y,Z,A,B,C in mm and degrees	None	Cartesian PTP move
PTA	25	R[0..5] = A1..A6 in degrees	None	Joint PTP move
FIN	100	None	I[2] --> 0/1	Is robot iddle?
SIO	200	I[2] = IO number, I[3] = IO value 1/0	None	Set IO
GIO	210	I[2] = IO number	I[3] --> IO value 1/0	Get IO
DIS	300	None	D[0] --> distance in mm	Get distance to next cartesian target
GOP	310	None	R[0...5] --> X,Y,Z,A,B,C in mm and degrees	Get next cartesian target pose
GOA	320	None	R[0..5] --> A1..A6 in degrees	Get next joint target