Impact-Aware Manipulation by Dexterous Robot Control and Learning in Dynamic Semi-Structured Logistic Environments

I.AM. Software Integration Policy (update D5.1)

Dissemination level	Public (PU)
Work package	WP5 - Integration and Scenario Validations
Deliverable number	D5.8
Version	F-1.0
Submission date	30-12-2022
Due date	31-12-2022

www.i-am-project.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 871899

Authors

Authors in alphabetical order		
Name	**Organisation**	**Email**
Harshit KHURANA	EPFL	harshit.khurana@epfl.ch
Claude LACOURSIÈRE	Algoryx	claude.lacoursiere@algoryx.com
Alessandro MELONE	TUM	alessandro.melone@tum.de
Fredrik NORDFELDTH	Algoryx	fredrik.nordfeldth@algoryx.com

Control sheet

Version history			
Version	**Date**	**Modified by**	**Summary of changes**
0.1	15-11-2022	Fredrik NORDFELDTH	First iteration of Integration policy and plan
0.2	17-11-2022	Fredrik NORDFELDTH	Second iteration of Integration policy and plan
0.3	18-11-2022	Fredrik NORDFELDTH	Started with the Working with GLUE section
0.4	20-11-2022	Claude LACOURSIÈRE	Edits with Fredrik
0.5	21-11-2022	Fredrik NORDFELDTH	Add notes for other authors
0.6	12-12-2022	Harshit KHURANA	Add persona and workflow regarding Dynamical Systems
0.7	12-12-2022	Claude LACOURSIÈRE	Add conclusion and summary
0.8	13-12-2022	Fredrik NORDFELDTH	Add text in architecture subsection + cleanup
0.9	13-12-2022	Alessandro MELONE	Add persona and workflow regarding I.Sense
1.0	26-12-2022	Fredrik NORDFELDTH and Claude LACOURSIÈRE	Fixed comments from reviewers and polished the text.

Peer reviewers		
	Reviewer name	**Date**
Reviewer 1	Teun BOSCH	16-12-2022
Reviewer 2	Jari VAN STEEN	19-12-2022

Legal disclaimer

TABLE OF CONTENTS

EXECUTIVE SUMMARY . . . **6**

1 Introduction . . . **7**
- 1.1 Purpose of the deliverable . . . 7
- 1.2 Intended audience . . . 7
- 1.3 Expected industrial adoption . . . 7

2 Identified personas . . . **8**

3 Identified usage . . . **8**
- 3.1 Identified workflows . . . 8
- 3.1.1 Workflow: Developing DS based motion planning . . . 9
- 3.1.2 Workflow: Impact monitoring . . . 9
- 3.1.3 Workflow: Developing control algorithms . . . 10
- 3.1.4 Workflow: Development and validation of simulation models . . . 10
- 3.1.5 Workflow: Method adaption for industrial applications . . . 11
- 3.2 Required equipment . . . 11
- 3.2.1 Laboratory . . . 11
- 3.2.2 Virtual lab . . . 12
- 3.2.3 Cluster computing . . . 12

4 Requirements on software . . . **12**
- 4.1 Fundamental integration requirements . . . 12
- 4.2 Software feature requirements . . . 13
- 4.3 Requirements on software maintenance . . . 14

5 User experience . . . **14**
- 5.1 Introducing the concept of GLUE . . . 15
- 5.2 Accessability . . . 15
- 5.3 Version control . . . 15
- 5.4 Commercial software . . . 15
- 5.5 Data formats . . . 16

6 Software components of GLUE . . . **16**
- 6.1 Communication: CLICK . . . 16
- 6.2 Simulation model definition . . . 16
- 6.3 A new solution: BRICK . . . 17
- 6.4 Communication protocol . . . 17
- 6.5 Control framework . . . 18
- 6.6 Simulation framework and runtime . . . 18

7 Working with GLUE . . . **18**
- 7.1 I.AM. BRICK files . . . 18
- 7.2 Batch jobs . . . 19

7.3 Dynamical System based Motion Planning 20
7.4 Architecture 21
7.5 Screenshots of runtime 22

Conclusion 23

REFERENCES 24

ABBREVIATIONS

Abbreviation	Definition
EC	European Commission
PU	Public
WP	Work Package

EXECUTIVE SUMMARY

The I.AM. project aims to boost collaboration between industry and academic research. For this to succeed, we need to make software and data easy to use, share and adapt for different applications such as parameter identification and synthetic data generation to name but two. Each of these may use all or only some of the software components developed within I.AM. , or by third party. Of central importance here is a software-in-the-loop (SIL) configuration where a variety of controllers can be coupled with a physics simulation library that faithfully reproduces the motion of a virtual robot.

The components include specific simulation models, libraries of control algorithms and assorted configuration files, experimental datasets, model identification datasets, synthetic data, and finally, virtual experiments configuration and results. This is in addition to libraries or modules implementing controllers, adapter software that makes modules compatible using communication protocols over different transport layers, and tools to manipulate binary data files.

It is anticipated that people will use different software libraries to simulate robot and contact dynamics at different level of details for instance. The same applies to nearly all aspects of simulation and analysis stacks. It is also expected that components will be assembled to produce different applications, and that these will be run in deterministic, synchronous mode, or in asynchronous mode as is the case for real robotics systems.

As argued in D5.1[OUD+21], a monolithic design such as Gazebo[Fou20] is not appropriate. For one, some features such as flexible joints needed for some of the analysis aren't available in the simulation libraries used in Gazebo or other commonly used tools. The same goes for flexible suction cups which are very important in this project. More fundamentally, monolithic tools are not well suited to produce distributed applications or reconfigurable applications.

The policy of maximum reuse, reconfiguration, modularity, and support for multiple applications with different purposes is supported by a set of tools. These include domain specific languages for model declaration, model composition and augmentation, as well as network communication protocols and binary data formats which are easily manipulated. Along with this come policies for continuous integration, release management, distribution with binary packages and portable lightweight containers, and a number of small single purpose application scripts.

The policies are meant to realize the Unix philosophy: "do one thing and do it well". This also comes with the use of signals instead of APIs, and using distributed processes instead of linking modules and libraries. In addition, components should be easily interchangeable.

As a central component is a declarative language to specify physical models which can be composed with each other. For instance a standard Panda robot URDF model can be augmented to have compliant joints or a suction cup, without intruding on the original file. This means that we can easily have models for different levels of details, as needed for the different applications.

1 Introduction

A full project background is found in the D5.1 deliverable report [OUD+21]. Therein we argue for the need of impact-aware manipulators, i.e., manipulators capable of establishing contact at nonzero relative velocity. Report D5.1 details how the I.AM. project is visioned to bring tossing, boxing and grabbing to the repertoire of industrial robots. The present document starts from the identified users and developers of impact-aware technology, and proceeds to describe software and integration policies to satisfy their respective requirements.

1.1 Purpose of the deliverable

This deliverable is an update of the I.AM. software integration policy D5.1, addressing testing of the software integration used for the three validation scenarios (TOSS, BOX, GRAB), and the public release of software used in the project. Once the policies are clarified and agreed upon, the developers of each component should then strive to comply with them.

This document describes the requirements on software design and integration for smooth collaboration between researchers, developers and industry. These are needed for development and deployment of state of the art impact-aware technology. The design principles needed to achieve the requirements are also counted as policies.

1.2 Intended audience

The dissemination level of D5.8 is “public” (PU), i.e., it is meant for members of the Consortium, including Commission Services, as well as the general public.

1.3 Expected industrial adoption

The I.AM. project has one year to go and it is now time to clarify the different uses of the software components and use cases. Policies presented below are pragmatic choices to achieve goals in terms of functionality, dissemination, and support for usage in a variety of contexts. As industrial application are both intended and expected, the policies include software license considerations to make adoption easier, including decisions regarding which component needs to be published as open source to have any chance of adoption in commercial and industrial applications.

2 Identified personas

To understand the software policy and integration plan we first define a set of personas with different aims and goals, and analyze and their respective required workflow.

A persona has a specific set of aims and use cases, as well as requirements on work flow, data storage and even license requirements. The different members of I.AM. generally represent different personas, and the requirement range from specific physical models, viz, a suction cup, or virtual sensor, or specific ways to actuate and control joints.

The personas we describe now are based on the interns, graduate students, postdocs, and PIs who have participated in I.AM.. This covers the gamut from developers to end users of the impact-aware features. They are as follows.

Controller developer: this persona needs simulation to test, validate and tune different control algorithms such as task based QP control or Dynamical Systems (DS).

Model developer: this persona needs to perform parameter identification for both the development of mathematical models for the simulation of robots, contact and impact dynamics, as well as parametric, statistical models usable by the different control algorithms when reacting to impacts.

Commissioning engineer: this persona needs to run simulation to determine optimal configuration of operational installation and assess performance. This also implies parameter identification to match simulation and reality.

3 Identified usage

Given the different goals of the different persona, there is not one single application which can meet all the requirements and workflows. For some cases one needs to perform just a few simulation step and vary parameters or initial conditions, for others one need to run simulations over a very long time to estimate wear and tear. Therefore, software components must be designed to be integrated in variable ways. In other words, we are considering several different applications, all of which use the same components, but each supporting a different usage.

Here we identify the different tasks, and the equipment and workflows required to fulfill them.

Consider for instance the task of mathematical modeling and validation of physics simulations. This requires equipping a lab environment with robots, sensors and objects to interact with and develop control functions that handle non-zero relative velocity contacts, and of course, a way to store all that data for subsequent analysis. In turn, the analysis involves model validation and parameter identification for instance.

This task is required for more than one of our personas since each needs validated models. That would include the **Model developer** and **Commissioning engineer**. The results of this validation are then used by all personas in the context of other tasks.

Note that the same model is simulated for different purposes, which means that there are multiple, reconfigurable applications.

3.1 Identified workflows

The different personas have different aims when using the software components. This implies a workflow, i.e., a set of steps they need to realize in order to produce the results they need.

We have describe workflows for the personas defined above, and these will be used to motivate the software integration policies.

3.1.1 Workflow: Developing DS based motion planning

The persona concerned here is the **Controller developer** who, in this case, uses techniques based on Dynamical Systems (DS)s. These are differential equations which respond to the pose and velocity of the end effector to produce commands that make it follow a desired trajectory. In different scenarios in the I.AM. project, the end effector comes into contact at non-zero relative velocity with the semi-structured and dynamic environment. The goal is to specify a DS that is robust to such changes.

- Designing DSs
    1. The experiment can be setup according to the scenario (TOSS, BOX, GRAB)
    2. Given the initial state of the robot from the simulation and the desired final state, the motion of the robot is controlled using a DS which sends and receives messages
    3. The DS based motion can be directly tested with the simulation
    4. If the response to the DS control depends on the properties of the robot so the simulation helps understand the desired motion visually
- Learning DSs
    1. The experiment can be setup as previously
    2. Data is collected for successful experiments with a variety of initial configurations of the robot
    3. Desired final configuration and the velocity of the end effector to perform the task is learnt using the data collected
    4. The DS is updated with the learned data and is then used for online motion planning from the initial to the desired state of the robot

3.1.2 Workflow: Impact monitoring

The objective here is to compare data from the impact monitoring algorithm running on a real robot using sensors directly with data from the corresponding simulation. Note that this can also be done in the simulation if adequate simulated sensors are available.

The workflow is as follow.

1. Setup the simulation with validated parameters and prescribed initial conditions
2. Run the simulation using the same control algorithm as for the real robot
3. Collect the data from the simulation which corresponds to that available on the real robot such as encoder positions and torque values (this is described in the Franka Emika interface `franka::RobotState` [Gmb20])
4. As needed, augment this with additional information available in the simulation but not directly on the real robot

5. Run the impact monitoring on the simulation

Note that in this context, the simulation can be stripped down by disabling graphics for instance.

The last point allows for running the scenario very many times, adding uncertainty by changing parameters, in order to test the robustness of the impact monitoring algorithm.

The simulation time can be reduced by identifying the approximate time location of impact, and restart just before that point, and then varying the initial conditions.

It should be possible to save simulation data in a data file, for later data analysis. For large datasets HDF5 [Gro19] should be used.

3.1.3 Workflow: Developing control algorithms

There are of course other types of controllers besides DS. For instance, `mc_rtc` contains task based QP control methods. These must be tuned in different ways, one involves large data sets with domain randomization to improve robustness, much as in the case of impact monitoring. In fact, it is expected that impact monitoring would be used in impact-aware controller development.

1. Setup experiment as previously
2. Setup simulated validation experiment
3. Implement control algorithm
4. Generate synthetic data with simulations
5. Analyze synthetic data to improve control algorithm
6. Deploy control algorithm in simulation environment
7. Deploy control algorithm with real robot
8. Publish validated simulations and results

We suggest a workflow which utilize both a physical and virtual lab. Also, large datasets containing synthetic data must be stored in HDF5 with an agreed upon layout. Tools for manipulating the data must be produced.

3.1.4 Workflow: Development and validation of simulation models

The models we are considering here range from compliant joints to electric drivelines to suction cups. For all of these, new simulation modules must be produced. Some can be built from existing components but some require entirely new mathematical model.

1. Setup experiment
2. Record real world behavior using a control algorithm when needed
3. Setup virtual lab to match the physical one
4. Run simulation with the same control algorithm when relevant
5. Perform parameter identification

6. Construct mathematical models
7. Update the simulation library
8. Distribute the validated simulation model

This can be performed repeatedly to produce models of different fidelity, resolution, and computational costs. Not all applications require the most advanced model.

3.1.5 Workflow: Method adaption for industrial applications

The final test is commissioning with real robots, but now with the aim of producing a reliable combination of control and sensor algorithms to perform a task hopefully more reliably and faster than previous installations.

This would through an experimental stage for validation of course.

1. Construct both a real and a virtual lab, both capable of compatible data collection
2. Perform parameter identification to calibrate the simulation as needed
3. Find a suitable control algorithm for the intended application
4. Implement this in tools compatible with company policies
5. Integrate simulation with existing controller framework
6. Compare experimental data with virtual lab results
7. Test the performance of the hardware
8. Convert the real lab to a production station
9. Test final performance

Companies have policies regarding the use of software from third parties and restrictions regarding acceptable licenses. This must be addressed in our policies.

3.2 Required equipment

The **Controller developer** and **Commissioning engineer** need access to a lab for hardware in the loop (HIL) testing and to a virtual twin for simulation in the loop (SIL) testing. For the **Model developer**, at least experimental data from a lab is needed.

3.2.1 Laboratory

A suitable lab must be equipped with the necessary robots, objects, sensors, cameras etc., and the robots must have compatible interfaces to control libraries, preferably the main one used in I.AM., namely, `mc_rtc`. Otherwise, adapters are needed. There must be a way to save data in binary form in a commonly agreed format and layout. HDF5 is to be used for that.

3.2.2 Virtual lab

The virtual lab must contain virtual models of robots, both in terms of 3D graphics but also in terms of physical properties. It must provide interfaces for the control libraries, but also for the formatted data produced in the lab as well as tools to manipulate and analyze it.

The virtual lab can operate in interactive mode with 3D graphics, allow for different types of visualization by communicating kinematic data to other applications such as RViz for instance, and support dynamic plotting. Likewise, the virtual lab must provide for offline execution as needed for parameter searches, domain randomization, generation of synthetic data, and learning experiments.

The robot models must provide for different level of details so that simulations can be sped up when high fidelity is not needed, or when testing whether or not flexible joints are relevant for a particular application.

The meaning of composition here is that though joints are considered to be rigid in, say, a URDF model, flexibility can be added to this without any interference with he basic model. The same goes with suction cups.

The configuration mechanism must make it easy to define the signals to be exchanged between the simulation and the controller. Whether a robot is velocity, position or torque controlled is independent of the robot model itself. The composition strategy separates the configuration of components that are orthogonal to each other, so one can choose the relevant signals for a given controller.

Composition avoids “copy-paste disasters” where an original model becomes modified in different ways in different applications. Also, this strategy aims at using native formats instead of conversion to a common format. If one needs to modify a URDF model, one uses standard tools for that. The configuration loader performs necessary conversion and mapping to given simulation libraries during initialization.

3.2.3 Cluster computing

Generation of synthetic data via millions of virtual experiments. This too requires techniques for the storage and manipulation of data, as well as special scripts to launch jobs.

4 Requirements on software

This section describes the functionality of the software components required for the different personas to realize their respective workflows, but also to easily incorporate on-going development from others into their work. The aim is to allow everyone to focus on their work knowing that other components are validated.

4.1 Fundamental integration requirements

For our personas to be able to use the software on their workstation, rely on the results and collaborate with each other we define a list of fundamental requirements.

For each workflow above, several software components are required and these must be assembled in some sort of application. The components must be integrated in some way to send and receive data to and from each other. Integration can be done at the software library level, using different APIs to construct one main application. But we need several applications each using different subsets of the components, each wiring them differently. This can lead to difficulties.

However, integration can be done in data-centric way. For this case, each separate library uses one common API for a communication library and a small client application is written. Such a client is generally very easy to write. For cases such as controller libraries which already support several communication protocols and transport layers, an adapter can be written if none of the existing protocols is suitable.

In the SIL context, it is important to have both synchronous and loss-free communication since determinism can be lost otherwise. In the HIL context however, communication is usually asynchronous and lossy. This is the case of CAN busses for instance. This hardly matters generally because the communication rates are very high, at 1KHz or 2KHz. For complicated systems, simulations cannot always achieve such speed.

A signals based application needs some sort of master and as we discussed, this must be able to run in different modes for different use cases. It is much easier to do that when using communication protocols than with API bindings. One can have several masters, one for each application, hopefully reusing modules. Also, one can replace a module with a different one having the same functionality by simply writing a compatible client. The master is entirely oblivious of the particulars of any and all modules.

A client-server architecture like the one described, based on signal, makes it easier to store simulation data since only the server needs to link to a library that can write formatted data. Though HDF5 is favored because of how well the data can be organized and annotated, writing CSV files is possible as well.

Of course, this comes with some degree of complexity. Given N software components we want to use in a given application, we need N client processes and one server one. These processes have to be launched in the right sequence, made to wait until everyone is ready, and the server must perform checks to see if the modules have compatible messages. A reliable and comprehensive launcher for such distributed applications is difficult to produce in the general case, but all the examples we have covered so far require only simple scripts.

One can also do integration by producing bindings to a scripting languages such as Python for each of the different modules. This is usually easier than working directly with the different APIs in `C++` for instance, but more difficult than the message based integration, as one would need to write wrapper APIs if different libraries were used for the same tasks, viz, physics simulation.

Central to this design is the specification of which signals must be sent or received by each module, and this must of course be kept in sync with the model definition. With the composition model described previously, this means that the list of signals is stored along the models they belong to, and can be augmented as needed.

The simulation models must be standalone and usable without any control framework if desired, and must be declared using open format, or combined from existing components available in open format. For instance, we have made extensive use of URDF in conjunction with BRICK. See Sec. 7 for more information.

4.2 Software feature requirements

A number of features are required to support SIL, parameter identification, validation of the simulations and hypotheses behind the controller designs, parameter search for controllers, and generation of synthetic data. Some of these features actually correspond to different applications, or at least, a polymorphic application.

– Easy to use application launcher

- Easy access to suitable control libraries
- Synchronous mode simulations for SIL
- Support for multiple communication protocols
- Support for URDF files
- Different control modes: position, velocity, torque
- Validation experiments
- Parameter identification
- Parameter optimization
- Generation of synthetic, randomized data
- Real-time multibody simulation supporting impacts and frictional contacts
- Open communication framework for integration
- Offline and online simulation mode
- Offline massively parallel simulations
- Saving simulation data in HDF5
- Supporting scripts for HDF5 data

4.3 Requirements on software maintenance

Any component with a `C++` code base or Python bindings should be able to integrate and communicate with the others in the framework. Of course, any component which can communicate over an internet protocol can be integrated as well.

Compatible components are required to have stable releases which then can be documented for future reproduction of the results using them. They should also comply with standard release management and continuous integration practices, including revisioning, unit, and integration testing.

5 User experience

Even though several software packages are required for any one of our personas, users should be able to install everything with just one operation. They should also expect rigorous adherence by the developers to standard development practices with respect to release management, testing, documentation and bug reporting.

In particular, the user should expect the following.

- Release management and continuous integration of all components
- Integration testing for full system releases
- Standard binary packages for installation on a set of supported platforms

- Docker containers for other platforms
- Open source modeling framework
- Seamless switch between SIL and HIL
- Python as runtime environment for access to data handling and processing tools
- A generic data format based on HDF5 for storing experimental and synthetic data

Releasing binaries compatible with native package managers on Linux distributions provides the best user experience. The same can be done on Mac OS X. Ideally, there should be very little if any hand intervention when installing the required packages. Because several non-standard packages might be involved, release via self-contained Docker images is also supported. This requires nearly no configuration by the user.

This being a GitLab centric project, users can of course clone the repository and compile themselves.

5.1 Introducing the concept of GLUE

To meet the user descriptions, requirements and identified workflows a development of a Python framework which implements the requirements was started. We decided to call the set of compatible software components for impact-aware manipulation "GLUE". The type of glue we have in mind here is the prototyping kind which is used when building experimental mock-ups as components can be replaced dynamically. Since the objective is to support multiple applications from parameter identification to synthetic data, and reconfigure from SIL to HIL, GLUE is not a framework, nor is it an application.

We are considering renaming this to contain RACK in the name. The RACK we think of is that used for either modular synthesizers or complicated lab setups. This is because using patch cables, one can re-route signals in different ways and transform the function of the connected devices.

5.2 Accessability

Users of GLUE can clone the Git repository, and then choose between a full installation if they are on a clean Ubuntu 20.04 system, or pull the latest Docker container, which is built by the GitLab runners. At the moment the Git repository is being maintained by Algoryx. Users of GLUE should be able to exchange their work and repeat each others' experiments.

5.3 Version control

Maintainers of software modules used in GLUE should try to support backward compatibility. Models should include the version number against which they were developed and the range of version number against which they can run successfully.

5.4 Commercial software

We cannot make an assumption that all involved software packages are open source. For example AGX Dynamics is distributed in binary form and requires a yearly license fee. So we have to assume users of GLUE can access personal licenses of all commercial software required. What is done to minimize friction is that a license to AGX Dynamics is built into the Docker Container, which has a builtin valid, time limited license. This will allow users connected to the project to use the library without making an immediate purchase.

5.5 Data formats

Instead of having one framework that can do everything, GLUE enables any software component to be part of a specific solution. By using open formats and protocols with automatically generated messages from model definitions, controller, simulation libraries, visualization frameworks and other components can be replaced easily. Of course the detailed behavior will change depending on which simulation library is used, which numerical time integration method and which solver is chosen within same. Details of friction models and solvers, the availability of certain types of models such as joint friction and flexible joints can limit functionality. Nevertheless, this kind of polymorphism is useful.

6 Software components of GLUE

GLUE is meant to connect a number of components together with thin interfaces. We follow the Unix philosophy here: “do one thing but do it well”. Here we list the current choices of components, formats and protocols.

6.1 Communication: CLICK

Communication protocols used on hardware work at high frequency, in the kilohertz range, and are generally lossy. Control modules are designed for such protocols. To communicate with a simulation and to guarantee determinism however, it is necessary to have a client which buffers messages and guarantee that no frame boundary is missed. The frame boundary here is connected to the fixed integration step used in the simulation.

The need for an intermediate library to establish synchronous communication became clear as Algoryx worked on projects with different robot manufacturers. The result is CLICK, that will soon (before mid 2023) be released as open source. CLICK is communication layer built on top of ZeroMQ[1], a widely used open-source messaging library. Like anything ZeroMQ, CLICK is lightweight and fast, and works on all operating systems. CLICK has been used by Algoryx' customers by writing a small adapter layer to bridge with their own proprietary communication transport layer and protocol. It also has a Python interface, namely, pClick.

There are plans to implement alternative transport layers for CLICK, replacing ZeroMQ with the Robot Operating System[2] (ROS). ROS includes two communication protocols, ROS and ROS2. Both implement “nodes” and follow a publisher-subscriber model. They differ in the transport layer. This is not synchronous per se which is why an adapter is needed.

6.2 Simulation model definition

Models of robots defined with URDF [Rob20] files are widely and freely available. They can be used directly in the GLUE simulation framework. The list of messages going to and from the control module are read directly from the robot definition thanks in part to comprehensive support for ROS and URDF in the QP robot control framework `mc_rtc`, developed by the partner CRNS.

However, the URDF format has limitations. It only supports ideal joints, not flexible ones, it does not have any notion of flexible or soft elements (e.g., a suction cup), and it cannot easily handle closed

[1]https://zeromq.org/

[2]https://www.ros.org/

kinematic loops. And of course, definition of material properties for contacts are not included. That is before we start discussing simulated sensors.

We need to look further to fulfill the modeling requirements.

In D5.1, Universal Scene Description (USD)[3] was mentioned as a good candidate declarative physics modeling. This has been investigated at Algoryx and the conclusion is that it is inadequate. Despite USD providing great features for 3D authoring and animation, it does not support simulation in a native way. NVidia did manage to introduce physics simulation but that effort is far from complete and applies only to their own physics simulation library, namely, PhysX. Mapping the AGX API to this would be a monumental effort and is a big commercial risk. Therefore, Algoryx decided to not invest in supporting USD in the foreseeable future.

6.3 A new solution: BRICK

During the I.AM. project Algoryx has been developing a new open declarative format for physics modeling. This is called BRICK and is a schema based on a subset of YAML[4] which makes the configuration files human readable and editable. The semantics of the declarative statements is not directly connected to the AGX Dynamics API and it is possible to write backends to support other nowadays popular simulation toolkits such as Bullet Physics or MuJoCo, for instance.

The BRICK semantics is polymorphic and split into specific bundles, e.g., Physics, Mechanics, Robotics. These are developed for generic modeling of simulation environments. Inside each bundle, the terminology is adapted to the specific domain and the backend then maps these to the specific API elements of, e.g., the AGX Dynamics library. For instance, in the robotics community, it is more common to say “link” instead of “rigid body”.

The bundles come with a data driven runtime module, currently written in C#, which can be integrated to any C# or Python simulation framework[5]. BRICK.Robotics can serve as a layer on top of the URDF definition and augment the physics with essential models of IMU sensors, flexible suction grippers, along with respective command and signal definitions to be used by CLICK explained below in Sec. 6.1. By integrating BRICK, GLUE benefits from a modeling format with reusable components, which can be used to compose simulation environments with a hierarchial methodology. Users can reuse what others have declared, and then extend parts of the definition, like augmenting the suction gripper to be flexible, or joint motors to have internal friction. BRICK also comes with a runtime called AGXBrick — the backend — which instantiates the simulation models in AGX Dynamics. This means that users of GLUE need not code anything to get started with their work.

The design of BRICK includes data typing and inheritance by composition. A BRICK file can contain a reference to a URDF file. The latter is parsed and converted to the BRICK semantic, and this can then be augmented with features available in AGX Dynamics which cannot be defined in the URDF schema.

6.4 Communication protocol

The list of commands to send and signals to read are defined in the simulation model files. A handshake at startup tests if the controller and model have compatible signals.

By using Python as a runtime, also the functionality in the AGXBrick Python module can be utilized, including resetting the scene in variable configurations enabling parameter search and domain random-

[3] `https://graphics.pixar.com/usd/docs/index.html`

[4] `yaml.org`

[5] Algoryx is currently working on a rewrite of Brick in C++ to enable a wider range of possible frameworks for integration.

ization. When launching the simulation models using AGXBrick instead of the CLICK runtime, the robots can also be controlled asynchronously over ROS or ROS2. The BRICK simulation models then need to be extended with ROS topics for signals that are not part of the URDF definition. However, the asynchronous messaging will not be deterministic in general.

6.5 Control framework

The mc_rtc control framework developed by CNRS have implemented a CLICK client, called mc_click. The control sequences written with mc_rtc can then be launched either to control the real robot or simulation, meaning that it supports both SIL and HIL. Similarly, other (open-source) QP control robot libraries could be employed in the future by implementing a suitable CLICK client.

6.6 Simulation framework and runtime

For impact-aware controllers to receive accurate sensor feedback from the simulation, comparable with the real robot sensors, the simulation framework need to support non smooth real time simulation with accurate force calculations. This is exactly what AGX Dynamics does. However, AGX Dynamics comes with a C++ API accessible from Python, and no GUI for authoring. Therefore the choice for the simulation runtime to depend on AGXBrick and AGXClick enables single instruction execution of simulation environments, with synchronous communication and no coding needed since all choices, configurations and functionality are built into the model files.

Note that a comprehensive application launcher is beyond the scope of this project. Indeed, a GLUE-centric simulation generally involves multiple processes distributed over multiple computers communicating over agreed-upon ports. Each process must be started on a designated machine and in the right sequence. We wrote special purposes launchers but this problem must be addressed in more generality.

7 Working with GLUE

A Git repository has been set up to fulfill the requirements listed above. In this section we give a taste of a few snapshots from the current implementation and usage.

7.1 I.AM. BRICK files

The GLUE framework supports at least the following use cases: (i) real time simulations with a QP controller or (ii) offline simulations for parameter identification or (iii) generation of synthetic data with batch jobs. The simulation models need to be multipurpose so that they can be used for both cases. Same model, different applications.

There is a model catalogue including the available robots, objects for manipulation, conveyor belts and the pairwise interaction definitions. A simulation environment can be composed using any combination of these models, and used for either of the above mentioned cases. Both the models and the simulation environment, which is also a BRICK model, are based upon I.AM. specific template models.

```
I_AM_SIMULATION:
  .extends: Physics.Component

  exit_threshold_linear:
    .doc: >
```

```
      A threshold for linear velocities used as an end condition.
      [m/s]
    .type: Real
    .value: 0.001

  exit_threshold_angular:
    .doc: >
      A threshold for angular velocities used as an end condition.
      [rad/sec]
    .type: Real
    .value: 0.01

  early_exit_conditions:
    .doc: >
      List of functions to test if the simulation should
      do an early exit. Examples:
      'fallen_below_floor',
      'zero_velocity',
      'zero_relative_conveyor_velocity'.
      Note: All batch_objects will be tested.
    .type: List<String>
    .value: []

  time_step:
    .type: Real

  simulation_time:
    .type: Real
```

7.2 Batch jobs

The example above shows BRICK declarations extending a `Physics.Component`, the most basic `Brick.Physics` scene, with thresholds and exit conditions for batch simulations. Snapshot of the lowest level template for a simulation environment in I.AM.The real time simulation application does of course not need to use the attributes that does not make sense, and vice versa. Each attribute can be overridden in any model extending the I_AM_SIMULATION model.

A case specific simulation environment template is as follows

```
I_AM_TOSS_BOX_CONVEYOR:
  .extends: I_AM_SIMULATION
  box:
    .type: CardboardBox

  conveyor:
    .type: LabConveyor
  early_exit_conditions:
    - fallen_below_floor
    - zero_velocity  # if fallen on ground
    - zero_relative_conveyor_velocity # if fallen on
                                      # conveyor successfully
```

which extends I_AM_SIMULATION and defines a box and a conveyor in the scene. Then there are several examples on how this scene is extended further depending on which box to toss, and which conveyor to land on.

```
TossScene:
  .extends: I_AM_TOSS_BOX_CONVEYOR
  conveyor:
    .value: LabConveyor
    speed: 1
    dir: Vec3(1.5,0,0)
    lengths: [2.55,1.0,0.84]
    localTransform:
      position: Vec3(-0.33,-1.9,0.42)
      rotation: Math.EulerAngles(0, 0, -Math.PI*0.5)

TossSceneBox4:
  .extends: TossScene
  box:
    .value: Box4
    localTransform:
      position: Math.Vec3(-0.3,0.2,0.7)

TossSceneBox5:
  .extends: TossScene
  box:
    .value: Box5
    localTransform:
      position: Math.Vec3(-0.2074,0.2040,0.6967)
```

The batch functionality of GLUE allows for loading one of the TossScenes with a set of initial conditions collected from real world recordings, simulate for a chosen amount of time, and then reset the scene at the next initial state. The result of the toss batch simulations is a large set of recordings, which are used as input to a postprocessing algorithm currently implemented in MATLAB.

A future feature is to load the batch job with a controller in the loop, also with online parameter search, instead of the current offline parameter search done with MATLAB.

7.3 Dynamical System based Motion Planning

As explained in Section 3.1, DS based motion planning can be extended to be impact-aware. A dynamical systems provides the rate of change of state variables and hence its output can be sent to the controllers.

Currently the DSs are implemented as standalone `C++` libraries with `CMake` support. Some interfacing was implemented so the libraries can be called by mc_rtc through its components. For now, the dynamical systems are independent of the robot. Different DSs can be used to generate the desired motion and the controller takes care of the joint limits, velocity limits, collisions etc. Creating DSs as a stand alone library makes the system modular and easy to use. The DS framework can also interact with the controller framework and the simulation framework through CLICK communication, separate from `mc_rtc`, though this will not be implemented in this project. The data that a DS would need either to learn the motion, or the desired final state of the robot can be easily generated through the simulation framework. For designing DS based motion which are robot dependent, for e.g., motions that are dependent on robot's inertia, or other metrics, the DS framework independently has access to the robot's URDF and will exist as a stand alone library. This modularity allows for adding different dynamical systems, systematically understanding their behavior with and without the control system, and understand how they lead the robot to interact with the environment through the simulation framework as they implement non-smooth interaction models.

7.4 Architecture

Figure 1 illustrates the content of the Docker container including all components needed for running the integrated software components we call GLUE.

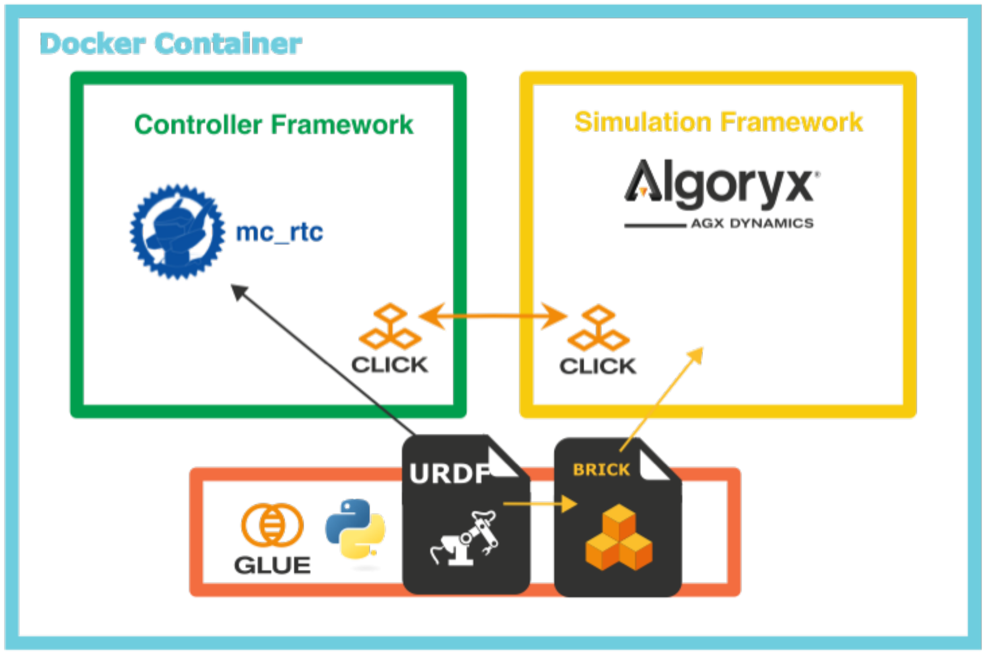


Figure 1: I.AM. integration architecture.

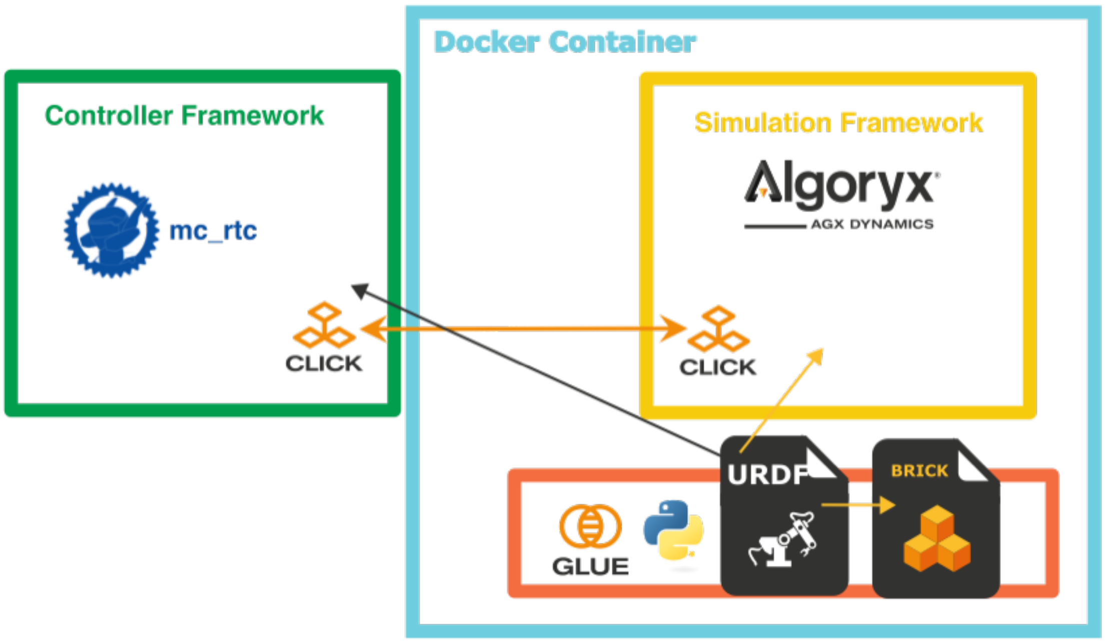


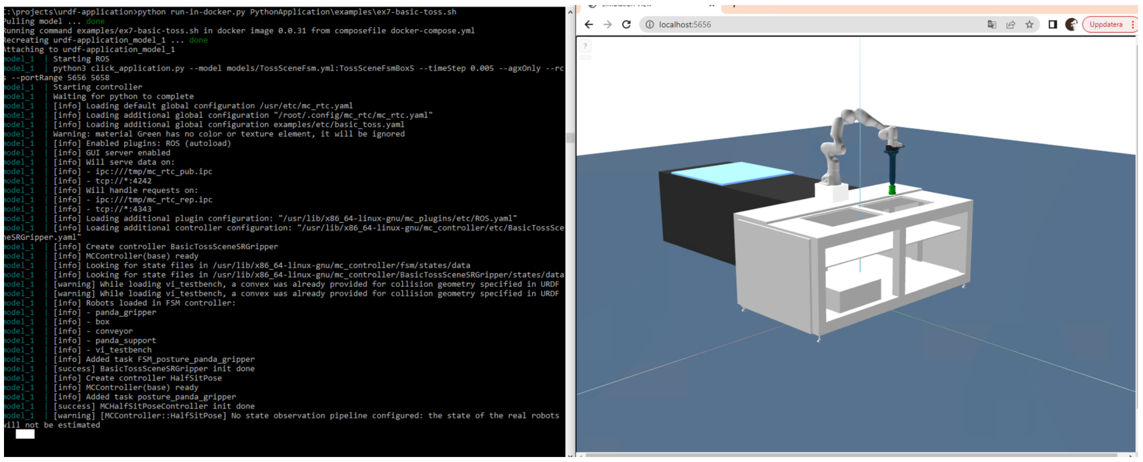
Figure 2: I.AM. integration architecture with controller framework outside docker.

The simulation model files and the Python applications that implement the logic for impact-aware manipulation within the I.AM. project are placed in a Docker Volume[6] which allow for updating the models and applications. To introduce something like a new controller, we suggest that the controller framework is built and running native on the machine of the developer, see Figure 2. The other parts of GLUE are still maintained in the container. The developer may distribute the controller as an executable, within the GLUE repository, for others to be able to reuse the work. As long as a software component communicates using CLICK, by implements a client as done for the mc_rtc controller framework, components outside of GLUE can be integrated in the runtime.

7.5 Screenshots of runtime

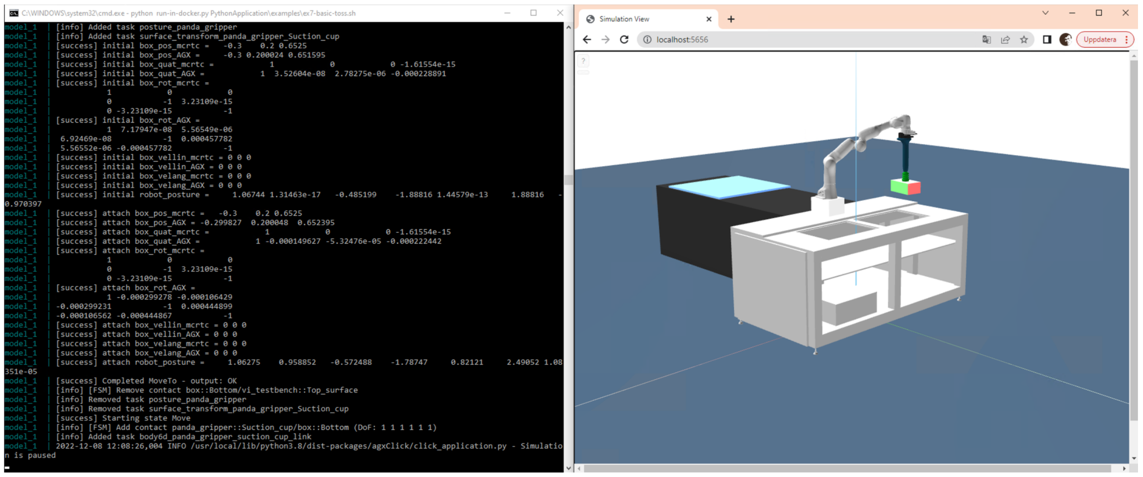
Figure 3 shows the instruction to load one of the examples of GLUE with Python next to the web viewer used for visualizing when the simulation runs in the Docker container. Figure 4 shows the simulation in a later state when the robot has picked up the box before it is being tossed. Some details about the communication between the simulation and the controller is visible in the terminal next to the 3D visual.

[6]`https://docs.docker.com/storage/volumes/`



```
:\projects\urdf-application>python run-in-docker.py PythonApplication\examples\ex7-basic-toss.sh
ulling model ... done
unning command examples/ex7-basic-toss.sh in docker image 0.0.31 from composefile docker-compose.yml
ecreating urdf-application_model_1 ... done
ttaching to urdf-application_model_1
odel_1  | Starting ROS
odel_1  | python3 click_application.py --model models/TossSceneFsm.yml:TossSceneFsmBox5 --timeStep 0.005 --agxOnly --rc
 --portRange 5656 5658
odel_1  | Starting controller
odel_1  | Waiting for python to complete
odel_1  | [info] Loading default global configuration /usr/etc/mc_rtc.yaml
odel_1  | [info] Loading additional global configuration "/root/.config/mc_rtc/mc_rtc.yaml"
odel_1  | [info] Loading additional global configuration examples/etc/basic_toss.yaml
odel_1  | Warning: material Green has no color or texture element, it will be ignored
odel_1  | [info] Enabled plugins: ROS (autoload)
odel_1  | [info] GUI server enabled
odel_1  | [info] Will serve data on:
odel_1  | [info] - ipc:///tmp/mc_rtc_pub.ipc
odel_1  | [info] - tcp://*:4242
odel_1  | [info] Will handle requests on:
odel_1  | [info] - ipc:///tmp/mc_rtc_rep.ipc
odel_1  | [info] - tcp://*:4343
odel_1  | [info] Loading additional plugin configuration: "/usr/lib/x86_64-linux-gnu/mc_plugins/etc/ROS.yaml"
odel_1  | [info] Loading additional controller configuration: "/usr/lib/x86_64-linux-gnu/mc_controller/etc/BasicTossSce
eSRGripper.yaml"
odel_1  | [info] Create controller BasicTossSceneSRGripper
odel_1  | [info] MCController(base) ready
odel_1  | [info] Looking for state files in /usr/lib/x86_64-linux-gnu/mc_controller/fsm/states/data
odel_1  | [info] Looking for state files in /usr/lib/x86_64-linux-gnu/mc_controller/BasicTossSceneSRGripper/states/data
odel_1  | [warning] While loading vi_testbench, a convex was already provided for collision geometry specified in URDF
odel_1  | [warning] While loading vi_testbench, a convex was already provided for collision geometry specified in URDF
odel_1  | [info] Robots loaded in FSM controller:
odel_1  | [info] - panda_gripper
odel_1  | [info] - box
odel_1  | [info] - conveyor
odel_1  | [info] - panda_support
odel_1  | [info] - vi_testbench
odel_1  | [info] Added task FSM_posture_panda_gripper
odel_1  | [success] BasicTossSceneSRGripper init done
odel_1  | [info] Create controller HalfSitPose
odel_1  | [info] MCController(base) ready
odel_1  | [info] Added task posture_panda_gripper
odel_1  | [success] MCHalfSitPoseController init done
odel_1  | [warning] [MCController::HalfSitPose] No state observation pipeline configured: the state of the real robots
ill not be estimated
```

Figure 3: Example of simulated Panda tossing a box started in the docker environment with a web browser viewer.



```
C:\WINDOWS\system32\cmd.exe - python  run-in-docker.py PythonApplication\examples\ex7-basic-toss.sh
model_1  | [info] Added task posture_panda_gripper
model_1  | [info] Added task surface_transform_panda_gripper_Suction_cup
model_1  | [success] initial box_pos_mcrtc =   -0.3    0.2 0.6525
model_1  | [success] initial box_pos_AGX =     -0.3 0.200024 0.651595
model_1  | [success] initial box_quat_mcrtc =           1            0            0 -1.61554e-15
model_1  | [success] initial box_quat_AGX =           1  3.52604e-08  2.78275e-06 -0.000228891
model_1  | [success] initial box_rot_mcrtc =
model_1  |            1            0            0
model_1  |            0           -1  3.23109e-15
model_1  |            0 -3.23109e-15           -1
model_1  | [success] initial box_rot_AGX =
model_1  |            1  7.17947e-08  5.56549e-06
model_1  |  6.92469e-08           -1  0.000457782
model_1  |  5.56552e-06 -0.000457782           -1
model_1  | [success] initial box_vellin_mcrtc = 0 0 0
model_1  | [success] initial box_vellin_AGX = 0 0 0
model_1  | [success] initial box_velang_mcrtc = 0 0 0
model_1  | [success] initial box_velang_AGX = 0 0 0
model_1  | [success] initial robot_posture =     1.06744 1.31463e-17   -0.485199    -1.88816 1.44579e-13     1.88816   -
0.970397
model_1  | [success] attach box_pos_mcrtc =   -0.3    0.2 0.6525
model_1  | [success] attach box_pos_AGX = -0.299827  0.200048  0.652395
model_1  | [success] attach box_quat_mcrtc =           1            0            0 -1.61554e-15
model_1  | [success] attach box_quat_AGX =           1 -0.000149627 -5.32476e-05 -0.000222442
model_1  | [success] attach box_rot_mcrtc =
model_1  |            1            0            0
model_1  |            0           -1  3.23109e-15
model_1  |            0 -3.23109e-15           -1
model_1  | [success] attach box_rot_AGX =
model_1  |            1 -0.000299278 -0.000106429
model_1  | -0.000299231           -1  0.000444899
model_1  | -0.000106562 -0.000444867           -1
model_1  | [success] attach box_vellin_mcrtc = 0 0 0
model_1  | [success] attach box_vellin_AGX = 0 0 0
model_1  | [success] attach box_velang_mcrtc = 0 0 0
model_1  | [success] attach box_velang_AGX = 0 0 0
model_1  | [success] attach robot_posture =     1.06275    0.958852   -0.572488    -1.78747     0.82121     2.49052 1.08
351e-05
model_1  | [success] Completed MoveTo - output: OK
model_1  | [info] [FSM] Remove contact box::Bottom/vi_testbench::Top_surface
model_1  | [info] Removed task posture_panda_gripper
model_1  | [info] Removed task surface_transform_panda_gripper_Suction_cup
model_1  | [success] Starting state Move
model_1  | [info] [FSM] Add contact panda_gripper::Suction_cup/box::Bottom (DoF: 1 1 1 1 1 1)
model_1  | [info] Added task body6d_panda_gripper_suction_cup_link
model_1  | 2022-12-08 12:08:26,004 INFO /usr/local/lib/python3.8/dist-packages/agxClick/click_application.py - Simulatio
n is paused
```

Figure 4: Example of simulated Panda tossing a box running in the docker environment with a web browser viewer.

Conclusion

This deliverable D5.8 provides a review of the architecture of the common software tools used by the different groups in I.AM. By reviewing the different persona along with their use case and workflow we conclude that the design is sound and can, in principle, handle all the cases. Experience already shows the usefulness and versatility of the different components.

This kind of architecture clearly leaves each of the components independent and free to evolve according to the respective developers. This is necessary to lower the threshold to collaboration and continuation of the independent development efforts beyond the end of the project.

There is more to do to prove the flexibility of the framework. For instance, we are planning to provide support for different physics simulation libraries following an “export to all” model, instead of the standard lock-in “import all” one. An application launcher is another area which needs development, or the selection of an existing tool for that.

There is a strong future for these policies as they are in use at Algoryx in other research and commercial projects in the fields of robotics.

REFERENCES

[Fou20] Open Source Robotics Foundation. *Gazebo*. `http://gazebosim.org`. June 2020.

[Gmb20] Franka Emika GmbH. *libfranka*. `https://frankaemika.github.io/docs/libfranka.html`. June 2020.

[Gro19] The HDF5 Group. *HDF5 version1.10.15*. `https://www.hdfgroup.org`. May 2019.

[OUD+21] Jos DEN OUDEN et al. *I.AM. Sotware Integration Policy*. Tech. rep. I.AM. Horizon2020, 2021. URL: `https://i-am-project.eu/images/delivarables_public/IAM_-_D51_-_Software_Integration_Policy_-_v10.pdf`.

[Rob20] Open Robotics. *Unified Robot Description Format (URDF)*. `wiki.ros.org/urdf`. June 2020.