D6.1 – INITIAL SIMULATION ENVIRONMENT

Deliverable ID	**D6.1**
Deliverable Title	**Initial Simulation Environment**
Work Package	**WP6 – Simulation and Performance prediction**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**05/10/2017**
Status	**Final**
Lead Editor	**Micha Rappaport (LAKE)**
Main Contributors	**Davide Conzon (ISMB), Enrico Ferrera (ISMB)**

Published by the CPSwarm Consortium

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731946.

Document History

Version	Date	Author(s)	Description
0.1	2017-07-26	Micha Rappaport (LAKE)	First Draft with TOC
0.11	2017-08-11	Micha Rappaport (LAKE)	Assignment of partners to sections
0.2	2017-08-17	Micha Rappaport (LAKE)	Specification of the APIs
0.21	2017-08-23	Micha Rappaport (LAKE)	Specification of the messages
0.22	2017-08-24	Micha Rappaport (LAKE)	Adding simulator details
0.23	2017-09-04	Micha Rappaport (LAKE)	Incorporating feedback from ISMB
0.30	2017-09-04	Micha Rappaport (LAKE)	Defining content of implementation section
0.4	2017-09-06	Micha Rappaport (LAKE)	Removed benchmarking section
0.41	2017-09-06	Micha Rappaport (LAKE)	Updated message definitions
0.42	2017-09-06	Micha Rappaport (LAKE)	Removed simulator configuration API
0.43	2017-09-13	Micha Rappaport (LAKE)	Integration of partner contributions
0.5	2017-09-14	Micha Rappaport (LAKE)	Restructuring implementation section
0.51	2017-09-17	Micha Rappaport (LAKE)	Providing content for implementation section
0.52	2017-09-18	Micha Rappaport (LAKE)	Integrating standards section by ISMB
0.53	2017-09-18	Micha Rappaport (LAKE)	Final version for reviewing
0.54	2017-09-25	Micha Rappaport (LAKE)	Integrating review feedback from ISMB
0.55	2017-09-26	Micha Rappaport (LAKE)	Integrating review feedback from ROBOTNIK
0.56	2017-09-26	Micha Rappaport (LAKE)	Harmonization with D4.1
0.57	2017-09-29	Micha Rappaport (LAKE)	Fixed JSON data formats
0.58	2017-10-02	Enrico Ferrera (ISMB), Davide Conzon (ISMB)	Harmonization with D5.2
1.0	2017-10-05	Melanie Schranz (LAKE)	Final Version to be submitted to EC

Table of Contents

1 Executive Summary

This deliverable, namely "D6.1 Initial Simulation Environment", gives a detailed description of the simulation environment and its integration into the algorithm optimization environment. Firstly, it describes the analysis done on available simulation solutions. This analysis has led to the choice of a set of solutions, which have been tested with a first version of the architecture. After the presentation of this first architecture, the deliverable describes the reasons that have led to extend it, with a broker-based approach and the API to be used. Finally, a first prototype that implements this architecture is described.
This deliverable reports the results of Task 6.1 activities.

2 Introduction

The document D3.1 - Initial System Architecture & Design Specification, delivered at M6, describes the initial architecture of the CPSwarm system, see Figure 1.

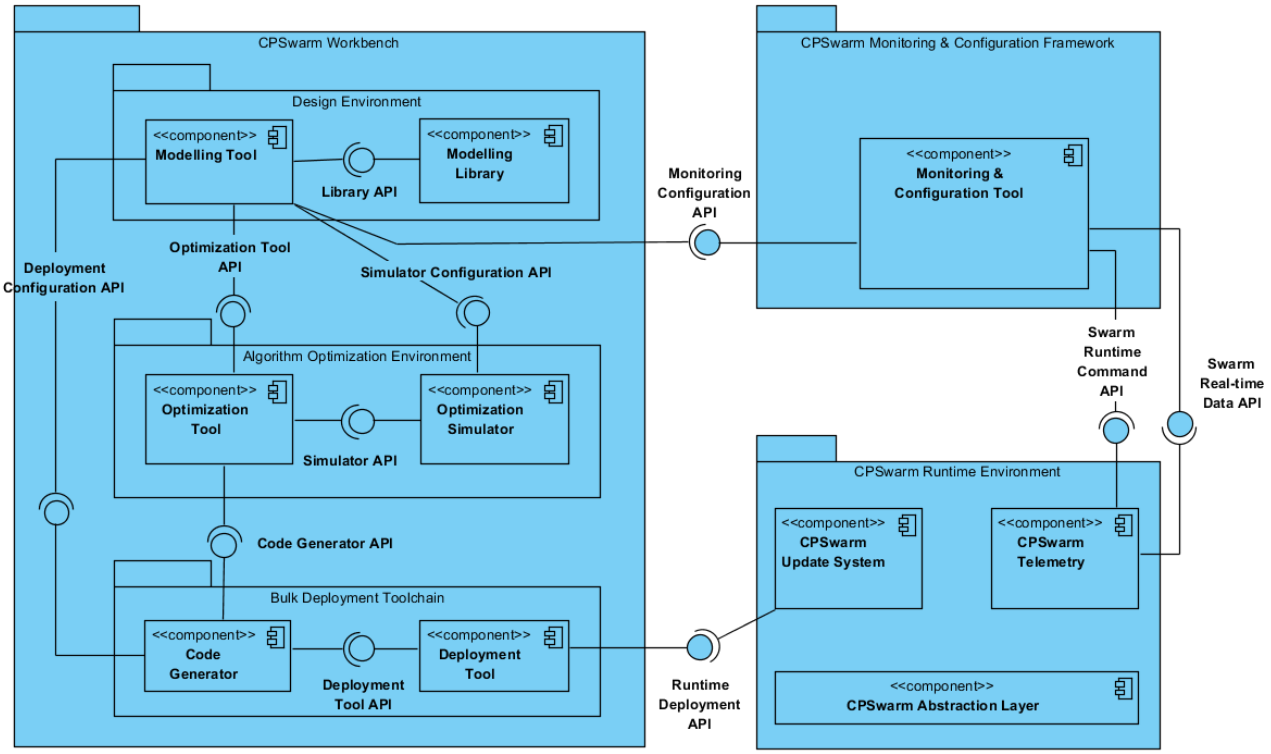

Figure 1 - Overview of components in CPSwarm system

Based on D3.1, this deliverable aims at better designing and providing a first prototype of the simulation environment, specifically considering the interaction with the modeling and optimization tools. The results of this work will be useful to update the Algorithm Optimization Environment component.
Together with the Design Environment components, it composes the part of the CPSwarm workbench responsible for the realization of an optimized controller that implements local interaction rules which lead to the desired global behavior of the system. The intended approach is to develop customizable environments, which allow using the most suitable implementations of the tools according to specific use cases. This means that different types of modelling, optimization and simulation tools can be used. This customization is allowed thanks to the definition of generic APIs which decouple tools among each other. The optimization phase requires frequent communication between the optimization tool and one or more optimization simulators.
Firstly, the deliverable analyzes a list of simulation solutions available on the market. Then, a filesystem based implementation, which has allowed to analyze issues related to the interoperation between the optimization tool and the simulation environment. An improvement of this approach, based on a network communication protocol and a broker architecture is subsequently addressed. In such architecture, the communication is handled by a broker that decouples the optimization tool from the optimization simulator. This enables a pluggability, allowing different simulators to be used as well as multiple instances of the same simulator to run in parallel.
The design of the architecture has included the definition of a set of interfaces, which have been implemented by the optimization tool and the optimization simulator to be able to communicate with the broker.

To test the architecture, has been used the Minisim simulator (see Section 7.2.1), a simple Java based simulator that has allowed to evaluate the connection of optimization tool and simulation environment in the CPSwarm workbench.
The work done has been used to support the design of a generic simulation environment and to define a communication infrastructure, needed to enable effective exchange of data between generic simulators and the optimization tool.
In the most general case, it would be possible to run a simulation without the requirement of pursuing the optimal solution. In this case, the optimization tool could in principle be removed from the toolchain to allow direct simulation without optimization.

2.1 Scope

This deliverable is limited to simulation environments that simulate robotics behavior with a focus on rovers and drones. Other types of simulators such as network communication simulators are not considered in this deliverable. Furthermore, only the simulator API is completely covered as it is part of the algorithm optimization environment. The interfaces connecting to the modeling environment are only briefly explained, because they are the main focus of D5.2 – CPSwarm Modelling Tool.

2.2 Document Organization

The rest of this deliverable is structured as follows. Firstly, Section 3 analyses simulation environments, identifying requirements in order to select best candidates to be used in CPSwarm. Workbech. Section 4 describes the architecture that is used within the algorithm optimization environment. Section 5 addresses the architecture and implementation of a broker based version of the algorithm optimization environment. Section 0 describes the required interfaces and defines how the information is exchanged between the different tools. Section 7 gives the implementation details of the algorithm optimization environment. Finally, Section 8 concludes this deliverable.

2.3 Related documents

ID	Title	Reference	Version	Date
D3.1	Initial System Architecture & Design Specification	D3.1	1.0	2017-08-18
D4.1	Initial CPS modeling library	D4.1	1.0	M9
D5.2	CPSwarm Modelling Tool	D5.2	1.0	M9

3 Simulation Environment Analysis

This section provides an initial state-of-the-art survey of simulation environments, which aims to identify the major solutions for Cyber Physical System (CPS) simulation. Such solutions shall be able to simulate the behavior of swarms of CPSs where the focus is on the drone / rover domain. Therefore, the simulation tools and technologies from the domains of robotics and swarm algorithm research are considered. The survey aims at identifying suitable candidates for the optimization simulator and to evaluate their suitability for the CPS swarm design methodologies, developed in the context of this project.

3.1 Requirements

1. *Simplicity*. Simulators shall be easy to use and integrate. Ideally to be successfully integrated in the CPSwarm workbench a simulator shall require minimal or no adaptation and should not force core-level development.
2. *Flexibility*. Integration with other tools, e.g., for remote control / set-up of simulation parameters shall be possible.
3. *Extensibility*. Investigated simulators shall be in principle independent from the kind of modeled CPS. This is particularly important for physics simulators, where specific robot customization shall be avoided.
4. *Scalability*. The simulation of large swarms of CPSs requires that the simulators performance shall scale well with the number of agents being simulated. The performance is measured in terms of required real time to perform a simulation. Ideally, the real time shall be smaller than the simulated time and scale at most linearly with the number of agents being simulated. This is especially important since the number of simulations carried out during the optimization process is very high. This is because the evolutionary approaches require to continuously evaluating the current candidate solution for guiding the evolution into the right direction.
5. *Abstraction*. Ideally, simulation granularity shall be tunable to the kind of feature / problem being evaluated. While simulation of the general behavior of a swarm might not benefit of a detailed, 3D, physical simulation, evaluation of the behavior of a swarm individual in response to external influences might require fine simulation of forces, accelerations and moments involved in the analyzed interaction.

3.2 Evaluation Criteria

All simulator candidates are evaluated using a set of functional features that guide the decision process for selecting the most appropriate simulator(s). An initial set of features is given below.

- License, cost
- Availability
  - high: binaries, documentation, and source code
  - medium: only two of the above
  - low: only one of the above
- Code / markup language of
  - source code
  - configuration files
  - log files
- Supported hardware models
  - many: the most common robotic platforms are modeled
  - some: some robotic platforms are modeled
  - none: no robotic platforms are modeled
- Possibility to extend
- Robot Operating System (ROS) interface
- Portability to hardware (using the same code)

- Fidelity
  - functional
    - high: all forces modeled on individual components (gravity, drag, motor acceleration, collisions, ...)
    - medium: all forces modeled only on vehicle as a whole
    - low: no forces, only position and velocities are modeled
  - physical (physics engine, dimensions, ...)
    - high: 3D, high resolution textures, reflections, ...
    - medium: 3D, no object detail
    - low: 2D
- Requirements
  - hardware (CPU, memory, ...)
  - software (operating system, libraries, physics engine, ...)
- Active development

3.3 List of Simulation Environments

An extensive number of simulation environments have been reviewed. Not all of them fulfill the aforementioned requirements and thus have not been selected for further evaluation. The ones that fulfill the requirements have been investigated further according to the evaluation criteria. The results are shown in the following subsections.

3.3.1 2D Simulation Environments

The simulation environments featuring only two dimensions are listed in Table 1.

Table 1 - Overview of two dimensional simulation environments.

Simulation Environment	License, Cost	Availa-bility	Language / Format	Hardware Models	ROS Inter-face	Hardware Portability	Fidelity (functional, physical)	OS	Active develop ment
MobotSim	All rights reserved, $30	low	Visual Basic					Win.	no
MRSim	All rights reserved	low	Matlab						no
Rossum Playhouse	GPLv2 / MIT, free	high	Java						no
Stage	GPLv2, free	high	C++, config: plain text	some	yes	yes, using ROS or Player	low, low	Linux, Win.	yes
STDR	GPLv3, free	high	C++, config: XML, YAML	some	yes	yes, using ROS	low, low	Linux	yes
Swarm	GPLv2, free	high	Java – Objective-C					Linux Win. MacOS Solaris	no
TeamBots	Free for education / research		Java, config: in source, plain text	some	no	yes	?, low	Linux Win. MacOS	no

3.3.2 3D Simulation Environments

The simulation environments featuring up to three dimensions are listed in Table 2.

Table 2 - Overview of three dimensional simulation environments

Simulation Environment	License, Cost	Availa-bility	Language / Format	Hardware Models	ROS Inter-face	Hardware Portability	Fidelity (functio-nal, physical)	OS	Active develop ment
ARGoS	MIT, free	high	C++, config: XML	some	yes	no	depending on physics engine	Linux MacOS	yes
Breve									no
DPRSim									no
Gazebo	ALv2, free	high	C++, config: SDF (XML)	some	yes	yes, using ROS or Player	high, high	Linux Win. MacOS	yes
jMAVSim	BSDv3, free	medium	Java, config: Java, shell script	some	yes	yes		Linux Win. MacOS	yes
Marilou	All rights reserved, €499							Win.	yes
Mission Lab									no
MORSE	BSDv3, free	high	Python, config: Python	some	yes	yes, using ROS	high, high	Linux	yes
MuRoSimF									no
peekabot									no
Simbad									no
SimSpark									no
Swarmbot3D									no
USARSim									no
v-rep	GPL / commercial not free	high	Lua, C++, config: binary	many	yes	yes, using ROS	high, high	Linux Win. MacOS	yes
Webots	All rights reserved, €3450	medium	C/C++, Java, Python, MATLAB	some	yes	yes		Linux Win. MacOS	yes

4 Algorithm Optimization and Simulation Environment

This section describes the inner architecture of the Algorithm Optimization Environment where the simulation environment is actually considered. This includes the modeling tool, the optimization tool, the optimization simulator and the interfaces. Figure 2 gives an overview of the different tools and the interfaces between them.

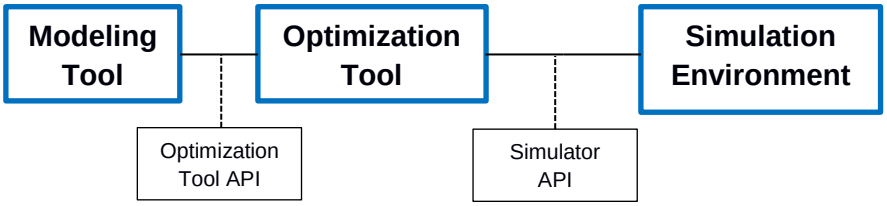

Figure 2 - Architecture of the algorithm optimization environment.

As a first approach, the communication between the optimization tool and the simulation environment has been based on a filesystem inter-process communication technique and ROS, which is a middleware that can control robots in simulation and on physical hardware. The ROS simulations can be launched from optimization tool by executing a script that first compiles the ROS package that implements the simulation and then executes this package. The simulator API, used for this first test, are detailed in the following section

4.1 Filesystem-based Simulator API

The optimization tool employs the optimization simulator to evaluate evolved candidate representations of a controller through simulation. This communication is enabled by the simulator API. This API passes the candidate representation and problem specific parameters from the optimization tool to the simulator and the performance of the candidate back from the simulator to the optimization tool. The API is implemented in a helper class of the optimization tool, providing all required functionality to the problem components. The class provides the following functions:

- `exportRepresentation`: exports the candidate representation to C code
- `getLogs`: returns the log file contents
- `readLogs`: reads the log files produced by ROS
- `run`: runs the simulation without GUI
- `runVisual`: runs the simulation with GUI
- `setEnvironment`: sets the environment to use
- `setParameters`: writes problem parameters to YAML file for ROS

4.1.1 Representation

The candidate controller is represented as an artificial neural network (ANN). It is exported by Framework for EVOlutionary design (FREVO) the optimization tool into a C source code file, as defined by Arduino[1]. The simulator includes this file, into its own agent source code. Once this file is exported to the simulator, a recompilation of the simulator becomes necessary. The ROS implementations include this source code file to enable the agents in the simulation making decisions by translating the sensor readings to actuator commands.

[1] https://www.arduino.cc/

4.1.2 Parameters

Parameters that need to be transferred from the optimization tool are written into parameter files in the YAML[2] format that is used by ROS. The parameters include some property parameters that the optimization tool forwards from the modelling tool.

4.1.3 Log

The performance of a candidate is measured by performance metrics defined in the modelling tool. The optimization simulator measures the metrics and writes them to log files in text format. The optimization tool reads these log files and applies the fitness function to calculate the fitness score of a candidate controller.

4.2 Integration tests

The architecture described in the previous sections is implemented using different, existing tools. The set of candidates that have been selected for this first implementation are Modelio as modeling tool, FREVO as optimization tool, and ROS-based simulators such as Stage and Gazebo, as simulation environment.
Modelio is an open source modeling environment that supports many standards. It is used in this context to define the models that are then used during the optimization process by the optimization tool and the optimization simulator. Modelio exports the modeled details as files readable by the optimization tool. A screenshot of the Modelio Graphical User Interface (GUI) is shown in Figure 3.

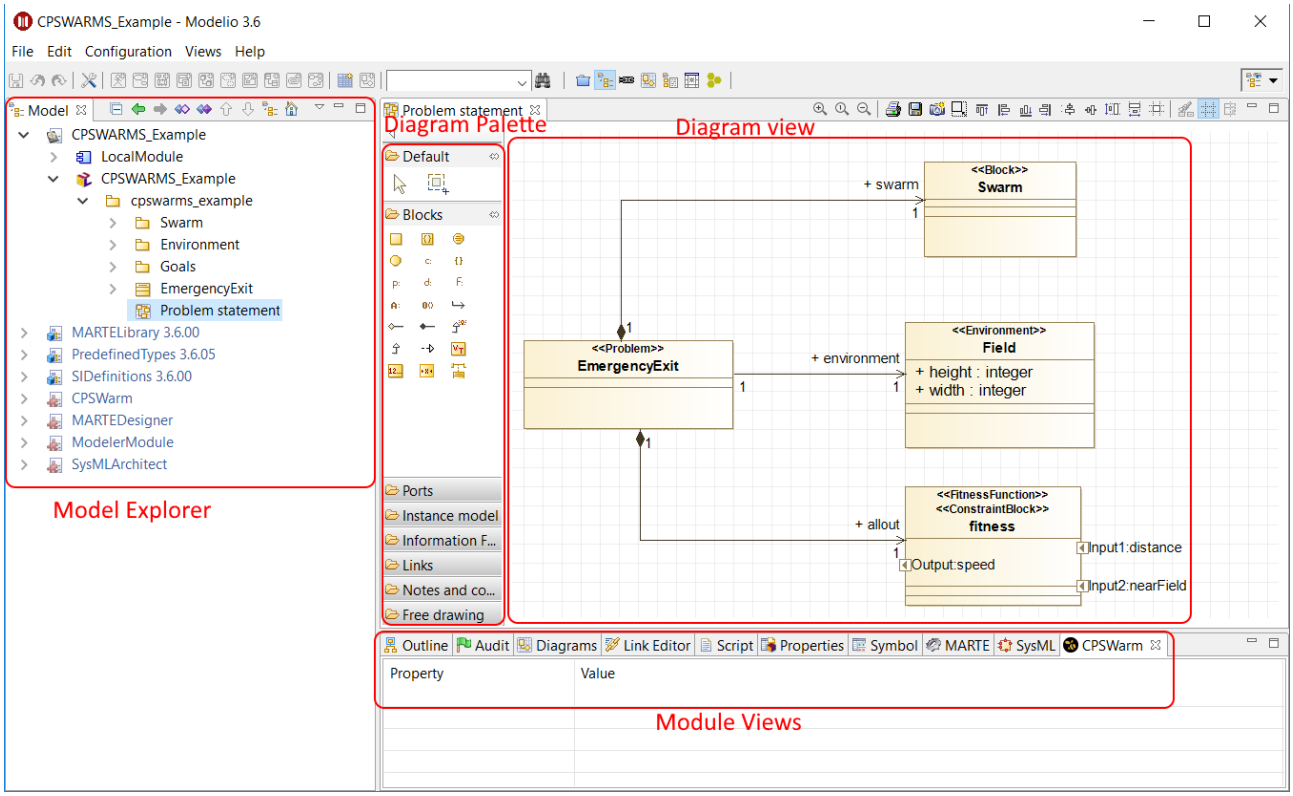

Figure 3 - Modelio graphical user interface.

FREVO is a tool for evolving and evaluating self-organizing systems by use of evolutionary algorithms. FREVO needs an input consisting of several components illustrated in Figure 4. First, it is necessary to define the problem where the evaluation context of the agent has to be implemented. Second, a controller representation should be selected that describes the structure of a possible solution. Third, the optimization

[2] http://yaml.org

method must be selected to optimize the chosen controller representation to maximize the fitness returned from the problem definition. Finally, the ranking module is configured to evaluate all agents in a problem. It returns a ranking of the candidates based on their fitness.

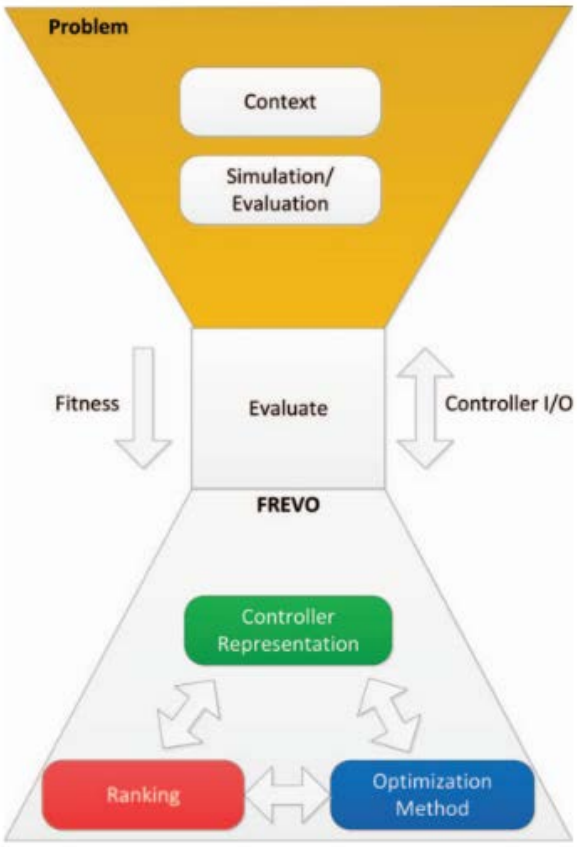

Figure 4 - FREVO architecture.

FREVO is written in Java and it requires Java environment version 1.6 as a minimum. Each component described above can be developed and tested separately for reuse in new projects. FREVO comes with a GUI, illustrated in Figure 5, to allow a user to make the selections described above from a list of predefined components. For instance, in the controller representation, a user can choose between different types of neural networks or a finite state machine. By running both configurations, the user can then decide which one is working better for solving a defined problem. FREVO can also be run from the command line without the GUI which facilitates the use on dedicated simulation servers.

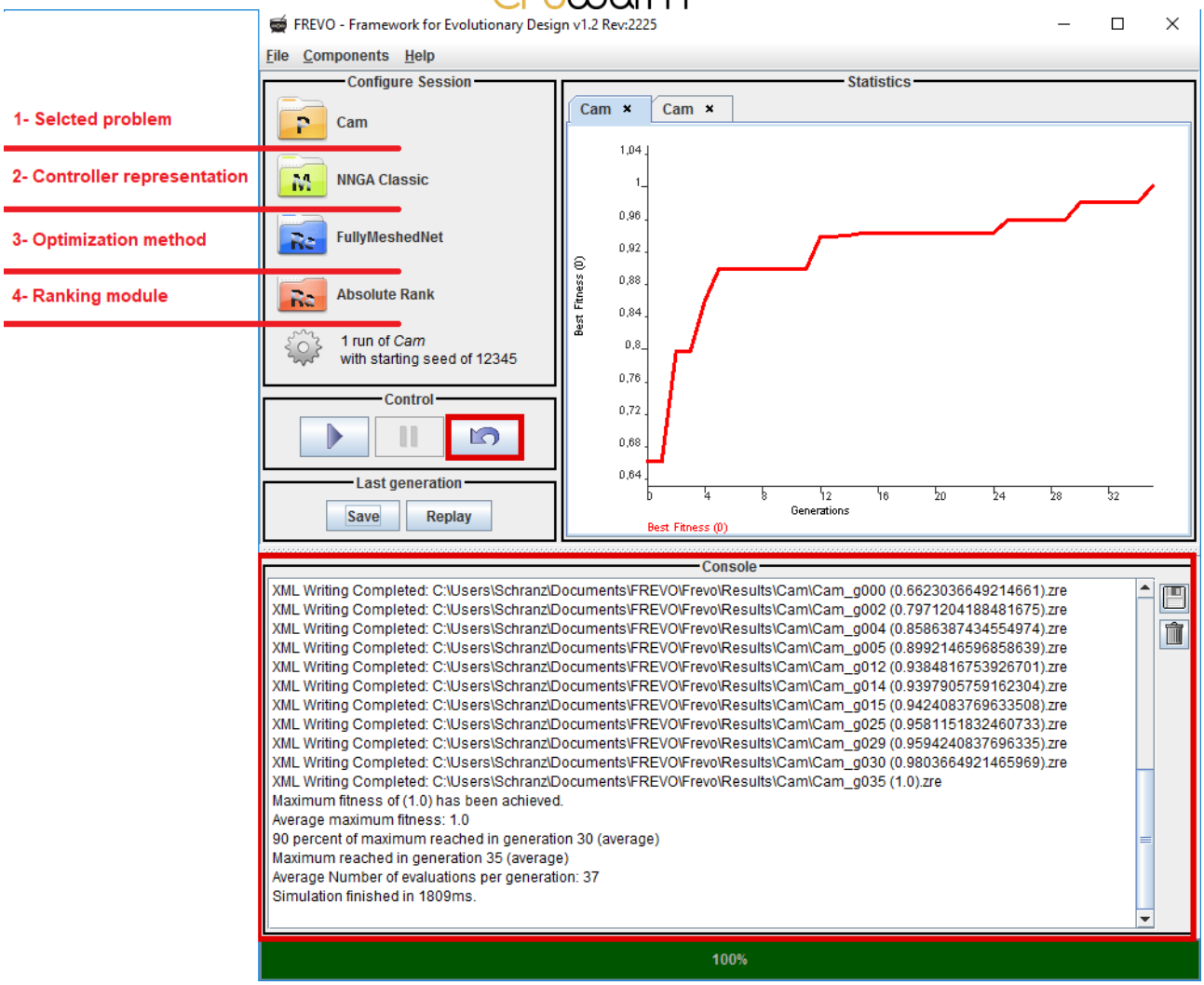

Figure 5 - FREVO graphical user interface.

This implementation serves two purposes. First, this implementation shows how to visually replay a candidate controller through simulation. Secondly, this implementation demonstrates how to use an optimized controller in simulation environments and on robotic hardware.

The simulation implements the initial model library use case described in deliverable 4.1 called *EmergencyExit*. This is a simple multi-agent problem where the agents have to escape from the environment. It runs in discrete time and space. In every time step the agents can move to one of the adjacent fields. Every agent tries to reach one of the exits of the environment while avoiding the fields occupied by obstacles or other agents. The performance of the simulation is measured as the distance of every agent from the nearest exit. Every agent logs this distance into a log file that is used by FREVO to assess the overall fitness of the candidate controller used in that simulation.

The *EmergencyExit* simulation is a good example to show how the initial catalogue of CPS models can be implemented and how problems can be added to an Optimization Tool like FREVO.

The ROS implementation includes configuration scenarios for two simulators: The Stage simulator[3] and the Gazebo simulator[4].

Stage is a low-fidelity two-dimensional robot simulator cf. Figure 6. The Stage scenario models the agents as simple squares in a two-dimensional environment. A screenshot of such a simulation can be seen in Figure 7.

[3] http://playerstage.sourceforge.net/index.php?src=stage
[4] http://gazebosim.org/

The Gazebo scenario is three dimensional and provides more detailed and realistic models. The agents are modeled as TurtleBots[5] defined in xacro files following the Unified Robot Description Format (URDF) specifications. Each robot runs in a distinct namespace but executes the same ROS nodes. This allows differentiating nodes, topics, services, and actions between the robots. The environment from which the robots try to escape is designed as a world in the Gazebo simulation. Figure 8 shows an exemplary simulation.

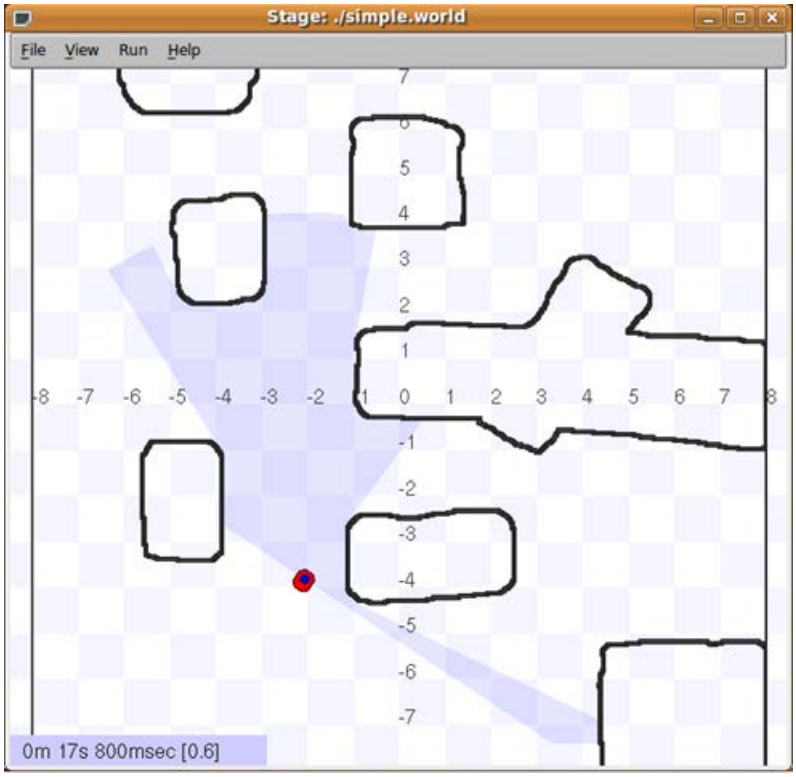

Figure 6 - The Stage simulator

[5] http://www.turtlebot.com/

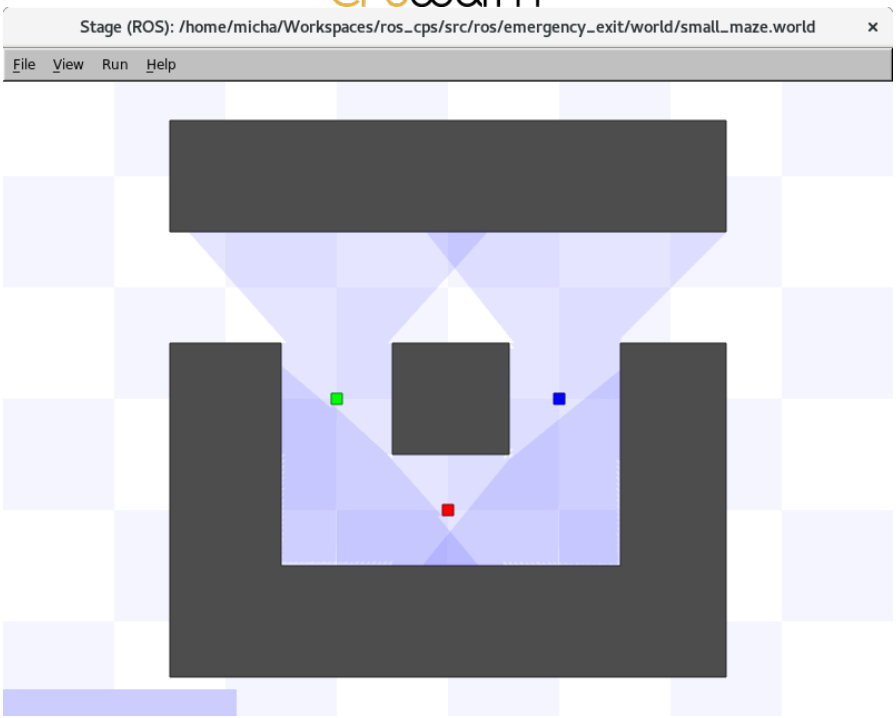

Figure 7 - Simulation in Stage.

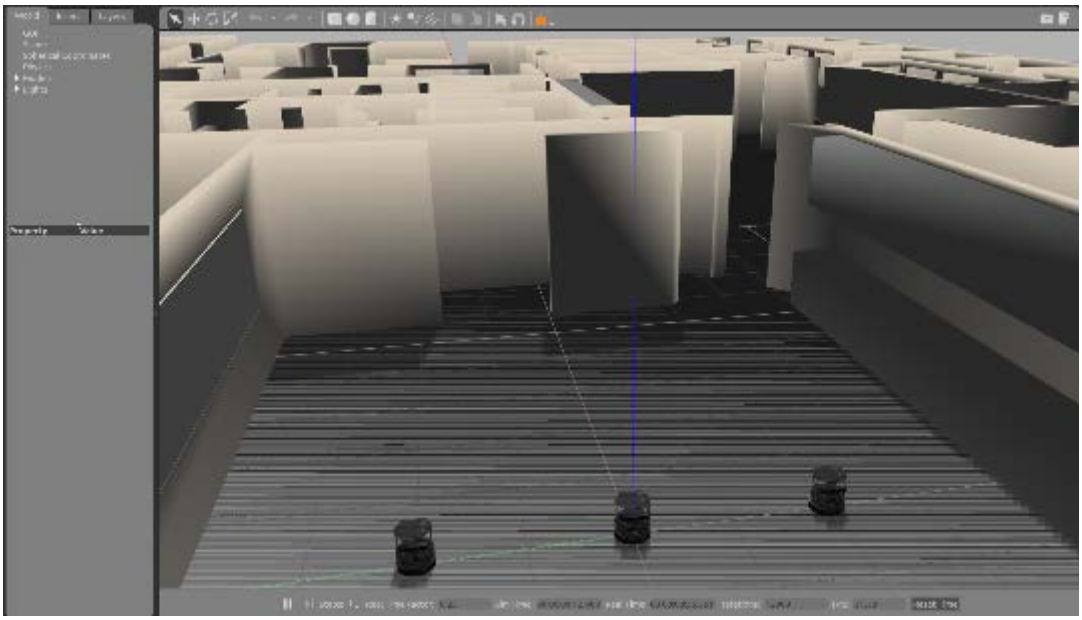

Figure 8 - Simulation in Gazebo.

5 Broker-based Algorithm Optimization and Simulation Environment

The previous implementation does not easily allow to perform simulation on multiple parallel simulation environments running remotely. In order to make this possible, it has been designed a broker based Algorithm Optimization Environment. Figure 9 gives an overview of the functional architecture of this broker-based approach.

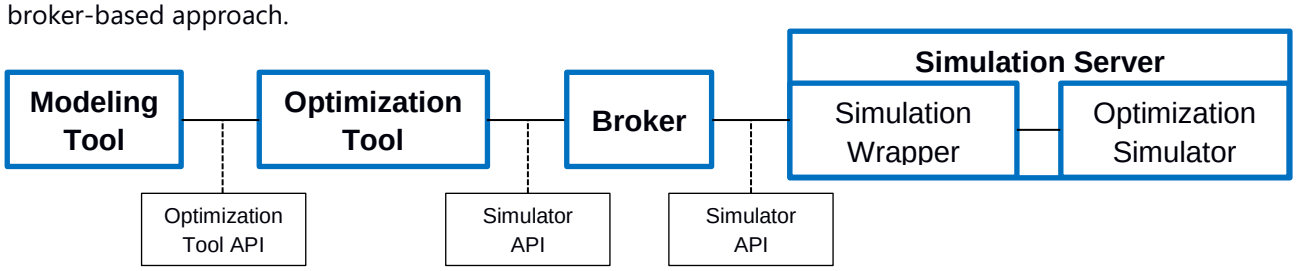

Figure 9 - Architecture of the broker-based algorithm optimization environment.

5.1 Requirements

First, the requirements that need to be fulfilled by this implementation are defined:

- Multiple simulation servers, even remotely located, offer simulation capabilities to the optimization tool through a broker.
- Each simulation server offers one or more simulation environments.
- A simulation environment exhibits certain characteristics but is also configurable to some extent by the optimization tool.
- Each simulation environment instance evaluates one candidate robotic controller (A. Sobe, 2012).
- Candidate controllers of one generation can be evaluated in parallel.
- A simulator simulates a homogeneous population of agents.
- The fitness of a candidate controller results from the final state of all agents in the simulation.
- The optimization tool can respond to requests from the optimization simulator at any point in time.
- Sensor and actuator values are uniformly represented as a vector of floating point values. Actuator values have an output range between -1 and +1.

5.2 Specification

5.2.1 Broker

The implementation of the proposed architecture is based on the use of the Message Queue Telemetry Transport (MQTT)[6] protocol. In recent times, this solution has been recognized as the de-facto standard for event-driven architecture in the IoT domain, based on the publish/subscribe paradigm. MQTT has been chosen because of its extreme simplicity and lightness. Its design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. The protocol has been originally designed at IBM with the goal to build robust communication channels over unreliable networks by constrained devices with strong latency requirements, working on top of Transmission Control Protocol (TCP). It is particularly suitable to be used in scenarios with resource-constrained devices because it is a binary, payload-agnostic protocol with minimal overhead. The protocol has been standardized by OASIS and recently version 5[7] has been released. This version introduces many new features to the protocol like the introduction of the request / response pattern. Indeed, MQTT has been originally designed as a purely publish / subscribe protocol. It provides many features like topic wildcards, different level of quality of service, retained messages, last will and testament, and persistence sessions. The architecture of a solution based on MQTT is composed by a message broker and several clients,

[6] http://mqtt.org
[7] https://www.oasis-open.org/committees/download.php/60716/mqtt-v5.0-wd13.pdf

which connects itself to the broker to send and receive events on specific topics. Several message broker implementations as well as client libraries are available under free and open-source licenses.
In the proposed architecture, the MQTT protocol is used for the exchange of messages between the optimization tool and the simulation servers. A MQTT client will be integrated in those tools to connect to a broker installed in a server or deployed using a cloud service.
Every client will be able to publish messages and subscribe to topics, in order to implement the communication schema defined in Section 0.

5.2.2 Simulation Wrapper

The simulation wrapper is a software layer installed on every simulation server that implements the simulator API as defined in Section 6.1. The simulation wrapper includes a MQTT client that automatically subscribes itself to the topic where the optimization tool publishes its messages for the simulation server. Furthermore, it provides a set of API functions to be used by the simulation server to handle the messages received and to send to the optimization tool the sensor values and the fitness score.
Thanks to this software layer, the optimization tool can communicate with the simulation server without knowing what type of simulation environment is actually used. Several simulators can be used in the same way also working in parallel, to reduce the simulation times. Indeed, after the optimization tool has created one generation of controller candidates, it can send different candidates to different simulation servers, which perform the simulation and calculate the fitness score. When all the fitness scores are returned, the optimization tool can perform the evolutionary steps to create the candidates for the next generation.

5.2.3 Simulation Server

The simulation server contains the actual simulation environment (described in detail in Section 3) used to perform the simulation. Thanks to the nature of the architecture proposed, the simulation server is decoupled from the optimization tool. In this way, every simulation server can be on a dedicated machine with the hardware requirements needed to execute the simulation in time.
The simulation server leverages the API provided by the simulation wrapper to communicate with the optimization tool. Every simulation server receives the discovery messages from the API (and through MQTT) and (if the server fulfills the requirements of the request and it has enough resources available), it answers to this request sending its information. It receives the configuration parameters to be used to configure the environment, the commands to start/stop the simulation and the actuation commands. In the same time, the server uses the API to send the values measured by the sensors and finally the fitness score obtained with the simulation. The messages exchanged are detailed in Section 0.
As already said in 5.2.2 several simulation servers can work in parallel to reduce the time required to complete a simulation, also if the simulation environments are heterogeneous among each other. Furthermore, it is important to say that a simulation server can be used by several optimization tools, but only one at a time.

6 Interfaces

As shown in Figure 1, the architecture envisions three types of API: the optimization tool API, the simulator configuration API and the simulator API, which are described in detail in Section 6.2. The optimization tool API are described in D5.2 – CPSwarm Modelling Tool, regarding the interaction between modeling tool and optimization tool. Instead, the simulation configuration API have not been defined in this first version of the architecture, because they are used to pass the models directly from the modelling tool to the optimization simulator and, in this first prototype this task is done by the optimization tool through the simulator API.

6.1 Standards

First, a review of standards for tool independent exchange of simulation models and co-simulation is done. In this deliverable, the messages used by the API are described using a custom JavaScript Object Notation (JSON) data format (Bray, 2014). Such messages are leveraged to implement the described prototype. The reason of this initial design choice is the will to realize a first prototype version as fast as possible, according to the agile development, testing, and evaluation approach. At the same time, more standardized approaches are being considered, in order to increase the compatibility with different existing solutions. This state-of-the art analysis is still progressing and other standards not described within this document could be considered in the next phases of this project and addressed in the following deliverables.
The approach addressed in this deliverable is mainly about integration of different applications, each aiming to achieve a different goal. Due to different applications, models of a system often should be developed using different programs (modeling and simulation environments).
To simulate the system, the different programs must interact with each other. The system integrator must cope with simulation environments from many suppliers. This makes the model exchange a necessity.
At the moment, not many standardized interfaces exist; lots of proprietary interfaces are usually used by tools:

- Simulink: S-function
- Modelica: External function, external object interface
- QTronic Silver: Silver-module API
- SimulationX: External model Interface
- NI LabVIEW: External model interface, simulation interface toolkit
- Simpack: Uforce routines
- ADAMS: User routines

This section introduces a Functional Mockup Interface (FMI) as a good candidate standard approach for API definition and message exchange in use-cases where several simulation tools need to be controlled by external entities, like co-simulation.

6.1.1 Functional Mockup Interface (FMI)

FMI is a standard approach used to enable both model exchange and co-simulation of dynamic models using XML files and executable C code.
One of the main entities defined by FMI is the Functional Mockup Unit (FMU). An FMU is the executable that implements the FMI interface.
An FMU contains the following information:

- A model description XML file that contains information about the CPS model and also general model information such as model name and FMI version. This allows avoiding model execution overhead and enables tools independency, because tools can read this information with their preferred programming language.
- Model equations (differential, algebraic or discrete equations) that can be used to describe a CPS model. These equations are represented by a small set of C functions. The FMU contains the source code and/or binary code for one or more platforms.
- Optional resource files that might be useful for the model, like (HTML) documentation files, model icon (bitmaps), maps and tables, and eventually libraries which can be useful in the model.

The FMI originally was subdivided in two standards, one for model exchange and one for co-simulation. Currently, the FMI specification 2.0 includes both standards.
The FMI model exchange defines how the subsystem model is exported from a simulation tool in the form of an FMU archive and how the subsystem model is imported into the simulation system for system simulation.
For the CPSwarm project the important protocol is the co-simulation one, since the project requires connecting the optimization tool with simulation servers which is a co-simulation application. The details of this protocol are described in the next sub-section.

6.1.1.1 FMI for Co-simulation

This subsection describes FMI protocol for the coupling of two or more simulation models in a co-simulation environment (FMI for co-simulation). The co-simulation technique is a general approach to the simulation of coupled technical systems. FMI for co-simulation is designed both for using subsystem models, exported by their simulators together with its solvers as runnable code and for coupling of simulation tools. In the latter case, the FMU implementation wraps the FMI function calls to API calls, provided by the simulation tool (e.g. CORBA API). In its most general form, the co-simulation scenario is implemented on distributed hardware with subsystems handled by computers with differences in hardware and software. FMI can be used to implement such distributed scenarios, taking in consideration that the communication channel to be used for the data exchange between the subsystems is not part of the FMI standard. An example architecture for this type of solutions is shown in Figure 10.

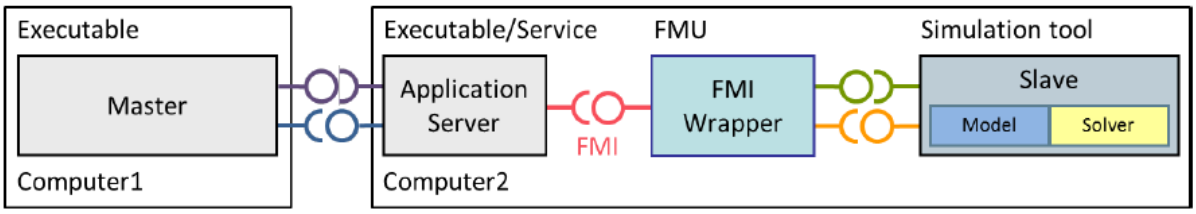

Figure 10 - Distributed co-simulation infrastructure.

This architecture is the one that can be used in the CPSwarm scenario. In this case, the Optimization Tool will be the master and the simulation server will be the slave. To integrate the FMI specification in CPSwarm, the set of API defined by the standard will be mapped with the one defined in this deliverable. Then a FMU will be implemented for each simulation server and finally it will be used to connect the simulation servers with the optimization tools.

6.2 Simulator API

The simulator API passes information between the algorithm optimization tool and a suitable simulation server during the optimization process. This interface shall first instantiate the communication channel. Then, it shall start the simulation and regularly pass real-time information between both tools. This includes the sensor readings and actuator commands. When the simulation finishes, the simulation server shall provide the fitness score to the optimization tool. The simulator API uses the broker architecture described in Section 0. The communication is handled by a publish / subscribe mechanism, where messages are published to a topic and all subscribers of that topic receive the messages. A list of all required topics is given in Table 3. The topic / message column states the name of the topic and the name of the messages that are transmitted over this topic. The publisher column states which component publishes on that topic and the content column states the message content. The messages are explained in detail below.

Table 3 - Communication topics managed by the broker.

Topic / Message	Publisher	Content
discovery	Optimization tool	Query of available simulation servers including requirements
server	Simulation server	Server ID and its capabilities
parameters	Optimization tool	Parameters for configuring the simulation
control	Optimization tool	Control messages for the simulation server
sensor	Simulation server	Sensor values of the agents
actuator	Optimization tool	Actuator commands for the agents
fitness	Simulation server	Fitness score of a candidate controller

Discovery message

The discovery message is published by the optimization tool when it connects to the MQTT broker. It allows querying for available simulation servers and includes a unique title of the simulation, a hash identifying the instance of the simulation, and a list of requirements on the simulation server. Every simulation server that is online and fulfils these requirements shall respond with a corresponding server message where it advertises its ID and capabilities. The hash reflects the server setup consisting of simulation and corresponding parameters. The hash is used for uniquely identifying the simulation in later messages. This is required to support multiple simulations in parallel. The JSON schema for discovery messages is given in Schema 1.

```
{
   "$schema": "http://json-schema.org/schema",
   "title": "discovery",
   "type": "object",
   "required": ["simulation", "simulation_hash", "requirements"],
   "properties": {
      "simulation": {
         "type": "string",
         "description": "Unique title of the simulation"
      },
      "simulation_hash": {
         "type": "string",
         "description": "Hash of the simulation"
      },
      "requirements": {
         "type": "object",
         "required": ["dimensions"],
         "description": "Required capabilities of the simulator",
         "properties": {
            "dimensions": {
               "type": "number",
               "description": "Number of spatial dimensions in the simulator"
            },
            "max_agents": {
               "type": "number",
               "description": "Maximum number of agents supported by the simulator"
            }
         }
      }
   }
}
```

Schema 1 - Discovery message definition.

Server message

The server message is used by the simulation server to advertise its capabilities. This message is published as response to a discovery message if the requirements of the optimization tool are fulfilled by the simulation server. This message informs the optimization tool about the simulation server presence, the simulations it can perform, and its capabilities. It includes a server ID and the simulation hash to enable the optimization tool addressing the server for executing simulation jobs. The JSON schema for server messages is given in Schema 2. The given list of capabilities is not exhaustive and can be extended in future.

```
{
   "$schema": "http://json-schema.org/schema",
   "title": "server",
   "type": "object",
   "required": ["server", "simulation_hash", "simulations", "capabilities"],
   "properties": {
      "server": {
         "type": "integer",
         "description": "ID of the simulation server"
      },
      "simulation_hash": {
         "type": "string",
         "description": "Hash of the simulation"
      },
      "simulations": {
         "type": "array",
         "description": "A list of all the simulations that can be performed at this server",
         "minItems": 1,
         "items": {
            "type": "string"
         }
      },
      "capabilities": {
         "type": "object",
         "required": ["dimensions"],
         "description": "Capabilities of the simulator",
         "properties": {
            "dimensions": {
               "type": "number",
               "description": "Number of spatial dimensions in the simulator"
            },
            "max_agents": {
               "type": "number",
               "description": "Maximum number of agents supported by the simulator"
            }
         }
      }
   }
}
```

Schema 2 - Server message definition.

Parameters message

The parameters message contains the parameters that describe the models as well as necessary configuration parameters for the simulation environment. It is provided by the optimization tool and contains parameters that have been introduced in the modeling tool during the modeling phase. The parameters have a default value but are changeable by the optimization tool for performing the optimization process under varying conditions. These parameters are problem specific but typically include the number of agents or the type of environment. This message is published by the optimization tool every time before starting a simulation. The minimal JSON schema for parameters messages is given in Schema 3. This message must include at least the server ID and the hash of the simulation. Further parameters are added on demand.

```
{
   "$schema": "http://json-schema.org/schema",
   "title": "parameters",
   "type": "object",
   "required": ["server", "simulation_hash"],
   "properties": {
      "server": {
         "type": "integer",
         "description": "ID of the simulation server"
      },
      "simulation_hash": {
         "type": "string",
         "description": "Hash of the simulation"
      }
}
```

Schema 3 - Minimal parameters message definition.

Control message

The control message is used by the optimization tool to control a specific simulation server. The control message includes at least the server ID and the simulation hash. Furthermore, it can include flags to indicate whether to run or terminate a simulation and to indicate whether the simulation shall be run with a GUI. The optimization tool publishes this message every time during the optimization process a new candidate controller needs to be evaluated through simulation. As a direct reaction, the addressed simulation server starts the simulation with the model parameters received earlier and publishes the sensor messages of the first simulation time step. The JSON schema for control messages is given in Schema 4.

```
{
   "$schema": "http://json-schema.org/schema",
   "title": "control",
   "type": "object",
   "required": ["server", "simulation"],
   "properties": {
      "server": {
         "type": "integer",
         "description": "ID of the simulation server"
      },
      "simulation_hash": {
         "type": "string",
         "description": "Hash of the simulation"
      },
      "run": {
         "type": "boolean",
         "description": "Whether to run or terminate a simulation",
         "default": true
      },
      "visual": {
         "type": "boolean",
         "description": "Whether to run the simulation headless or using a GUI",
         "default": false
      }

   }
}
```

Schema 4 - Control message definition.

Sensor message

The sensor message provided by the simulation server transmits the sensor readings of one agent to the optimization tool. The optimization tool uses the sensor readings for computing the next actuator commands for this agent. Furthermore, this message includes the simulation hash as well as an agent ID for correctly returning the actuator commands to that agent. This message is published by the simulation server at every

new simulation time step. The sensor readings are generic and consist only of an array of numbers. The way these numbers are interpreted by the optimization tool depends on the problem that is simulated. More particular sensor messages e.g. for range sensors can be added in future. The JSON schema for sensor messages is given in Schema 5.

```
{
   "$schema": "http://json-schema.org/schema",
   "title": "sensor",
   "type": "object",
   "required": ["simulation", "agent", "sensor"],
   "properties": {
      "simulation_hash": {
         "type": "string",
         "description": "Hash of the simulation"
      },
      "agent": {
         "type": "integer",
         "description": "ID of the agent"
      },
      "sensor": {
         "type": "array",
         "description": "Sensor reading as numeric values",
         "minItems": 1,
         "items": {
            "type": "number"
         }
      }
   }
}
```

Schema 5 - Sensor message definition.

Actuator message

The actuator message provided by the optimization tool transmits the actuator commands for one agent to the simulation server according to a previously received sensor message. This message furthermore includes the simulation hash as well as an ID of a specific agent for which the actuator commands are computed. The simulation server executes the actuator commands for the agent addressed in this message. The actuator commands are generic and consist only of an array of numbers. The way these numbers are interpreted by the simulation server depends on the problem that is simulated. More particular actuator messages e.g. for waypoint navigation can be added in future. The JSON schema for actuator messages is given in Schema 6.

```
{
   "$schema": "http://json-schema.org/schema",
   "title": "actuator",
   "type": "object",
   "required": ["simulation", "agent", "actuator"],
   "properties": {
      "simulation_hash": {
         "type": "string",
         "description": "Hash of the simulation"
      },
      "agent": {
         "type": "integer",
         "description": "ID of the agent"
      },
      "actuator": {
         "type": "array",
         "description": "Actuator command as numeric values",
         "minItems": 1,
         "items": {
            "type": "number"
         }
      }
   }
}
```

Schema 6 - Actuator message definition.

Fitness message

The fitness message is the final message of a simulation run. When the maximum number of simulation steps has been reached, the simulation server computes the fitness of that simulation and publishes it with the fitness message. Furthermore, the fitness message includes the hash of that simulation for correctly assigning the fitness to the correct candidate controller. The JSON schema for fitness messages is given in Schema 7.

```
{
   "$schema": "http://json-schema.org/schema",
   "title": "fitness",
   "type": "object",
   "required": ["simulation", "fitness"],
   "properties": {
      "simulation_hash": {
         "type": "string",
         "description": "Hash of the simulation"
      },
      "fitness": {
         "type": "number",
         "description": "Fitness score of the candidate controller used in the simulation"
      }
   }
}
```

Schema 7 - Fitness message definition.

6.3 Optimization Process Sequence

During the optimization process the messages mentioned above are passed through the simulator API multiple times. A typical sequence of an optimization process is shown in Figure 11.

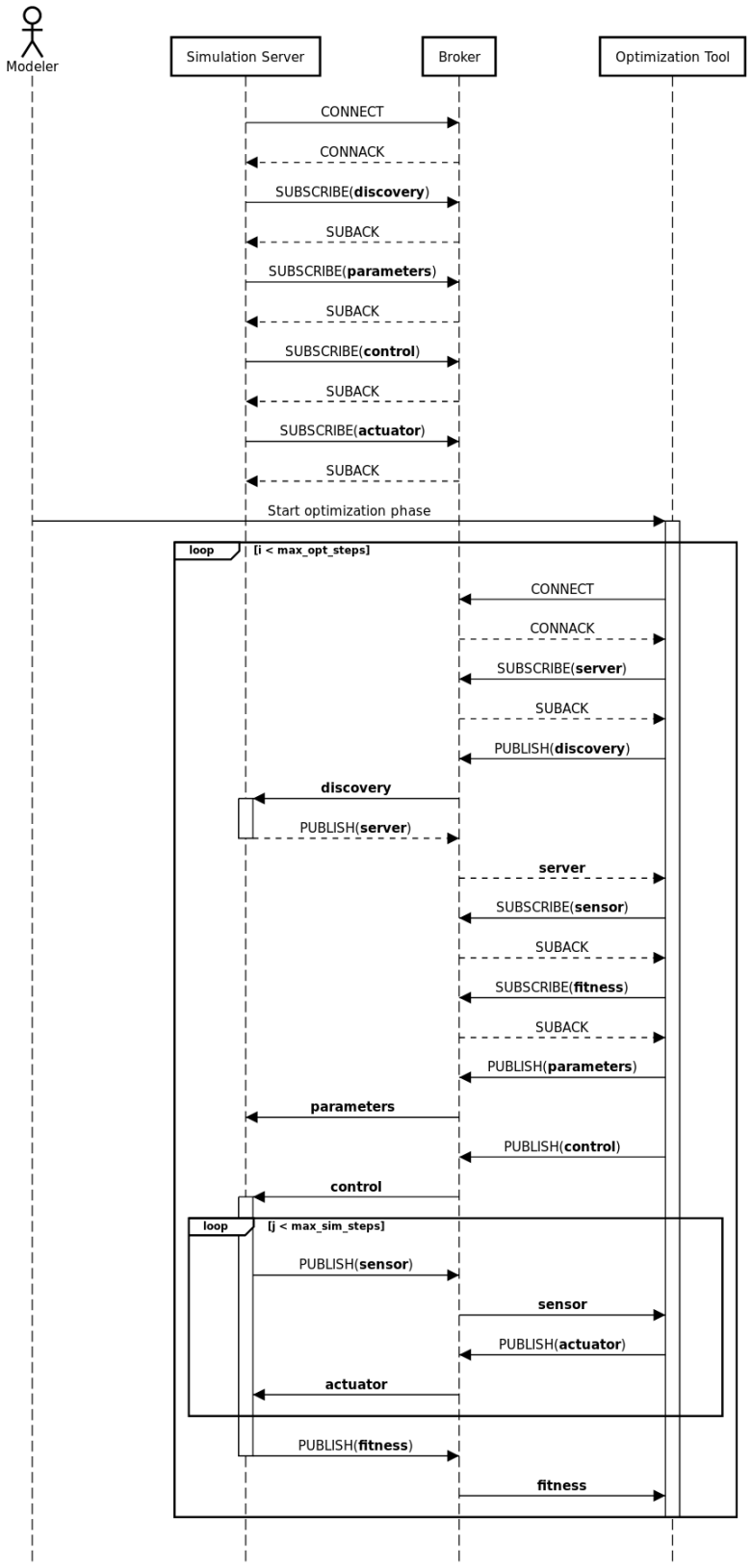

Figure 11 - Exemplary optimization process sequence.

7 Broker-based Simulation Environment Prototype

The architecture described in the previous sections is implemented using different, existing tools. The tools are extended to conform to the specification of the interfaces described above. Even in this case Modelio and FREVO have been considered as modeling and optimization tools. Considering the complexity of the simulator wrapper implementation for complex simulation environments such as Stage or Gazebo, it has been decided to use, as a first version, a simpler simulation environment such as the Minisim. This choice meets the requirements of the agile development approach, which allows implementing and evaluating design choices more quickly and build solutions in an iterative and incremental manner.

7.1 FREVO MQTT Client

To interconnect FREVO with the simulation server, an MQTT client is added to FREVO. The client is implemented as a helper class called *simMQTT* that implements the MQTT callbacks for receiving messages from the broker. The class is instantiated by the problem component in FREVO that evaluates a candidate controller through simulation. The *simMQTT* class handles all the communication with the broker. This includes establishing of the connection, subscribing to the relevant topics for message reception, and functionality for publishing messages to the broker. The *simMQTT* class is executed in a separate thread that waits for the message callbacks. FREVO is blocked from further execution by a mutex when the *simMQTT* class is instantiated. This mutex is released once the fitness of the candidate controller is received and passed back to FREVO.

7.2 Simulation Server

The simulation server consists of a simulation environment and a simulation wrapper. The simulation environment is used as optimization simulator that evaluates a specific candidate controller in the optimization process. The simulation wrapper serves as client that handles the connection to the MQTT broker.

7.2.1 Simulation Environment: Minisim

Minisim is a simple Java based simulation environment that runs without GUI. It implements a multi-agent simulation where one or more agents have to reach a goal before being caught by one or more defenders.
The *Minisim* example has been used to show how a simulation can be invoked by the optimization tool FREVO via a network connection. While the *EmergencyExit* problem is part of an implementation example showing how problems can be added to FREVO, *Minisim* allows evaluating the connection of optimization tool and simulation server in the CPSwarm workbench. The *Minisim* example was chosen as initial implementation showcasing the network connection between the optimization tool and the simulation server since it has a lower complexity level.
The defenders are placed between the agents and the goal but move slower than the agents. The agents move according to the commands given by the optimization tool. The defenders always move towards the closest agent. Modeling parameters for this simulation are:

- *Integer mapWidth*: Width of the map.
- *Integer mapHeight*: Height of the map.
- *Integer numAgents*: Number of agents.
- *Integer numDefenders*: Number of defending agents.
- *double speedAgents*: Distance that an agent travels in one time step.
- *double speedDefenders*: Distance that a defending agent travels in one time step.
- *double[][] agents*: Position of each agent.
- *double[][] defenders*: Position of each defending agent.
- *double[] goal*: Position of goal.

The simulation environment can be configured by following parameters:

- *String helloMessage*: A custom welcome message.

- *Integer stepSize*: Milliseconds between two visualizations.
- *Integer maxSteps*: Maximum number of steps that are simulated.

The simulation environment does not support a GUI but the output of the simulation can be visualized on the command line.

7.2.2 Simulation Wrapper

The simulation wrapper is implemented as a Java library. The library embeds the MQTT Paho client[8] for MQTT communication. This library exports an abstract class called *SimulationWrapper*. This class implements the behavior that is common to all the simulators. When the implementing class is instantiated the *SimulationWrapper* connects the client to the MQTT broker configured and subscribes to the topics of interest: parameters, control, actuator and discovery. The *SimulationWrapper* implements the *MqttCallback* interface. When a new MQTT event is received from the client, it receives the notification of this message, through the *messageArrived(String topic, MqttMessage message)* callback. This callback examines the topic where the event has been published and accordingly calls one of these methods:

- *parseParameters(MqttMessage message);*
- *parseControl(MqttMessage message);*
- *parseActuator(MqttMessage message);*
- *parseDiscovery(MqttMessage message);*

The first three methods, listed above work in the same way. They parse the received message in a Plain Java Object (POJO), generated from the JSON schemas defined in Section 0. All the JSON functionalities are implemented using the GSon library[9]. Once the POJO is parsed, it is passed to one of these abstract methods:

- *handleParametersContent(Parameters parameters);*
- *handleControlContent(Control control);*
- *handleActuatorContent(Actuator actuator);*

The implementation of these methods is left to the implementing simulation server, which will implement them to handle the messages received in the proper way.

The discovery message is handled in a different way compared with the behavior described in the previous point. For this scope, the *SimulationWrapper* provides two APIs:

- *setServerInfo(Server serverInfo);* which is called by the implementing simulation server to set its information. The *serverInfo* object passed as parameter is a POJO obtained from the schema defined in Section 6.1.
- *setAvailalble(boolean availalble);* which is used by the simulation server to indicate if it is available to do a simulation or not.

When a new discovery message is received, the *SimulationWrapper* checks the *serverInfo* object to control if the server fulfills the requirements indicated in the discovery message (see Section 6.1. for details about the message) and control if the simulation server has indicated it as available. If the two conditions are satisfied, the simulation server publishes a message with the *serverInfo*.

7.2.3 Simulation Wrapprer Integration in Minisim

For this deliverable, a first implementation of the *SimulationWrapper* has been done for the Minisim. The *SimulationWrapper* library has been added to the Minisim project and the class *MinisimSimulationWrapper* implementing the *SimulationWrapper* abstract class has been added. This class contains the implementation of the three abstract methods

- *parseParameters(MqttMessage message);*
- *parseControl(MqttMessage message);*
- *parseActuator(MqttMessage message);*

They are used to set up the simulation environment and to perform simulations using Minisim.

[8] http://www.eclipse.org/paho/
[9] https://github.com/google/gson

7.2.4 Simulator API

The simulator API allows interconnecting the optimization tool FREVO to one or more simulation servers running the Minisim simulator. The connections are managed by a broker running the MQTT protocol. FREVO and Minisim both use the MQTT client described above to connect to the broker. Before the optimization process can be started by FREVO, the broker server as well as the simulation server needs to be running. The simulation wrapper of Minisim connects to the broker and waits for incoming requests. Once FREVO starts the optimization process it needs to perform multiple simulations. When a simulation starts it connects to the broker and sends a discovery message with the requirements on the simulation server. In the following an exemplary communication sequence using a single agent simulation is shown. Schema 8 shows such a discovery message.

```
{
   "simulation": "minisim",
   "simulation_hash": "21a57f2fe765e1ae4a8bf15d73fc1bf2a533f547f2343d12a499d9c0592044d4",
   "requirements": {
      "dimensions": 2
   }
}
```

Schema 8 - Exemplary discovery message.

The simulation server replies with the server message given in Schema 9.

```
{
   "server": 1,
   "simulation_hash": "21a57f2fe765e1ae4a8bf15d73fc1bf2a533f547f2343d12a499d9c0592044d4",
   "simulations": ["minisim"],
   "capabilities": {
      "dimensions": 2
   }
}
```

Schema 9 - Exemplary server message.

To transfer the modelling details and the simulation parameters to the simulator, FREVO sends the parameters message described in Schema 10.

```
{
   "server": 1,
   "simulation_hash": "21a57f2fe765e1ae4a8bf15d73fc1bf2a533f547f2343d12a499d9c0592044d4",
   "mapWidth": 13,
   "mapHeight": 7,
   "numAgents": 1,
   "numDefenders": 1,
   "speedAgents": 1,
   "speedDefenders": 0.33,
   "agents": [[0.5, 3.5]],
   "defenders": [[8, 3.5]],
   "goal": [12.5, 3.5],
   "helloMessage": "Welcome to minisim",
   "stepSize": 250,
   "maxSteps": 15
}
```

Schema 10 - Exemplary parameters message.

Then it sends the control message described in Schema 11 to start the simulation.

```
{
   "server": 1,
   "simulation_hash": "21a57f2fe765e1ae4a8bf15d73fc1bf2a533f547f2343d12a499d9c0592044d4",
   "run": true
}
```


Schema 11 - Exemplary control message.

The Minisim simulator replies with the initial sensor readings of the agent shown in Schema 12.


```
{
   "simulation_hash": "21a57f2fe765e1ae4a8bf15d73fc1bf2a533f547f2343d12a499d9c0592044d4",
   "agent": 0,
   "sensor": [0, 0, 0, 0]
}
```


Schema 12 - Exemplary sensor message.

Schema 13 shows an exemplary actuator message sent by FREVO as answer to the previous sensor message.


```
{
   "simulation_hash": "21a57f2fe765e1ae4a8bf15d73fc1bf2a533f547f2343d12a499d9c0592044d4",
   "agent": 0,
   "actuator": [0.25]
}
```


Schema 13 - Exemplary actuator message.

The simulation ends with the fitness message shown in Schema 14.


```
{
   "simulation_hash": "21a57f2fe765e1ae4a8bf15d73fc1bf2a533f547f2343d12a499d9c0592044d4",
   "fitness": -2.85
}
```


Schema 14 - Exemplary fitness message.

8 Conclusions

This deliverable describes the initial implementation of the simulation environment and its integration into the algorithm optimization environment. The document outlines the architecture and interfaces as well as the implementation using different tools.
In future work, CPSwarm will investigate different types of simulation environments that focus on models in domains additional to the robotics one e.g., considering network simulators like OMNeT++. Furthermore, the architecture proposed in Section 5 will be applied to other simulation environments, developing wrappers for other simulators (i.e. the ROS-based simulators Stage and Gazebo). Finally, the code generator API will be implemented allowing to export the optimized controller to the bulk deployment toolchain. Additional next steps will focus on the integration of heterogeneous CPS swarms and on the support for additional actuator and sensor types.

Acronyms

Acronym	Explanation
API	Application Programming Interface
CPS	Cyber Physical System
ROS	Robot Operating System
CPU	Central Processing Unit
JSON	JavaScript Object Notation
ANN	Artificial Neural Network
GUI	Graphical User Interface
FREVO	Framework for EVOlutionary design
URDF	Unified Robot Description Format
FMI	Functional Mockup Interface
FMU	Functional Mockup Unit

List of figures

List of tables

References

A. Sobe, I. F. (2012). FREVO: A tool for evolving and evaluating self-organizing systems. *IEEE Self-adaptive and Self-organizing Systems Workshop Eval4SASO'12.* Lyon, France.

Bray, T. (2014). The javascript object notation (json) data interchange format. Internet Engineering Task Force.

Hofmann-Wellenhof, B., Legat, K., & Wiesner, M. (2003). *Navigation: principles of positioning and guidance.* Wien: Springer.

Jarvis, R. A. (1983, March). A Perspective on Range Finding Techniques for Computer Vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 122-139.

Vaughan, R. (2008). Massively Multi-Robot Simulation in Stage. *Swarm Intelligence, 2*, pp. 189-208.