



D3.1 – INITIAL SYSTEM ARCHITECTURE & DESIGN SPECIFICATION

Deliverable ID	D3.1
Deliverable Title	Initial System Architecture & Design Specification
Work Package	WP3 – Architecture design and Component Integration
Dissemination Level	PUBLIC
Version	1.0
Date	2017-08-18
Status	Final
Lead Editor	Junhong Liang (FRAUNHOFER)
Main Contributors	Junhong Liang (FRAUNHOFER), Dario Bonino (ISMB), Etienne Brosse (SOFTEAM), Melanie Schranz (LAKE), Wilfried Elmenreich (UNI-KLU), Regina Bíró (SLAB), Edin Arnautovic (TTTECH), Rafa Lopez (ROBOTNIK), Omar Morando (DGSKY)

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
0.01	2017-05-2	Junhong Liang (FIT)	First Draft with TOC
0.02	2017-05-31	Dario Bonino (ISMB)	Minor revisions, integration of missing Internal Review History, specification of section 2 TOC
0.03	2017-06-18	Junhong Liang (FIT)	Rework of TOC for tool analysis and architecture design.
0.04	2017-06-19	Junhong Liang (FIT) Dario Bonino (ISMB)	Added partial contribution from FIT and ISMB
0.05	2017-06-21	Junhong Liang (FIT)	Added executive summary, introduction and conclusion
0.10	2017-06-28	Junhong Liang (FIT)	Added contributions from LAKE, SOFTEAM, SLAB, TTTECH, ROB, ISMB and FIT
0.11	2017-06-29	Junhong Liang (FIT)	Fixed formats, grammars and references
0.20	2017-06-30	Junhong Liang (FIT)	Released for review
0.30	2017-07-19	Junhong Liang (FIT)	Added modifications from partner addressing reviewers' comments, released for second review
0.31	2017-07-20	Junhong Liang (FIT)	Added contribution from UNI-KLU
0.32	2017-08-17	Claudio Pastrone (ISMB)	A few minor modifications inserted after plenary discussion
1.0	2017-08-18	Junhong Liang (FIT)	Final version to be submitted to EC

Internal Review History

Review Date	Reviewer	Summary of Comments
2017-07-05	Alessandra Bagnato (SOFT)	<ul style="list-style-type: none"> Spelling and various comments throughout the document.
2017-07-10	Dario Bonino, Claudio Pastrone (ISMB)	<ul style="list-style-type: none"> Modifications inserted throughout the document Comments have been provided requiring updated contributions.
2017-07-20	Alessandra Bagnato (SOFT)	<ul style="list-style-type: none"> 0.31 document version approved with minor comments.
2017-08-17	Claudio Pastrone (ISMB)	<ul style="list-style-type: none"> 0.31 document version approved with minor comments.

Table of Contents

Document History	2
Internal Review History	2
Table of Contents	3
1 Executive Summary.....	4
2 Introduction.....	5
2.1 Related documents.....	5
3 Analysis of Relevant Engineering Methods, Tools, Technologies and Standards	6
3.1 Methodology	6
3.2 Survey of Tools and Frameworks currently adopted for CPS design / development.....	6
3.3 Survey of existing modelling standards / patterns.....	13
3.4 Preliminary Analysis of design methodologies applicable to the CPS domain.....	16
4 Architecture Design.....	20
4.1 Methodology	20
4.2 Stakeholders and Requirements	23
4.3 Context View	28
4.4 Functional View.....	29
4.5 Information View.....	47
4.6 Deployment View.....	57
5 Security Perspective	59
5.1 Security threat analysis	59
5.2 Countermeasures	60
5.3 Security aspects in CPSwarm architecture design.....	63
6 Scalability Perspective	64
6.1 CPSwarm Workbench	64
6.2 CPSwarm Deployment Toolchain	65
6.3 Summary and discussion	66
7 Future Steps.....	67
7.1 Future activities and timeline	67
7.2 Alternative architecture for future exploration	67
8 Conclusions.....	68
Acronyms	69
List of figures.....	69
List of tables.....	70

1 Executive Summary

This document is a deliverable of the CPSwarm project, funded by the European Commission's Directorate-General for Research and Innovation (DG RTD), under the Horizon 2020 Research and innovation Program (H2020). It is a public report organized along 2 main parts respectively introducing an analysis of relevant engineering methods, tools and standards currently available for swarm design and the initial design of the CPSwarm system architecture.

The part *Analysis of relevant engineering methods, tools and standards* introduces the technical background of the CPSwarm project. It provides an up-to-date snapshot of available technologies that can be readily exploited in the project. All partners participated in this analysis by surveying different relevant domains of swarm development, from CPS modeling to robotics programming. The technologies described in this part serve as technical starting point for the architecture design presented in this report.

The part *Architecture Design* presents the initial architecture design of the CPSwarm system. The architecture design process is documented by complying with the ISO/IEC/IEEE 42010 "System and software engineering – Architecture description" [1] standard. Relevant viewpoints of the system are presented as documentation for different architectural aspects of the CPSwarm system.

Besides the main functional and component-based descriptions, cross-cutting concerns such as security, scalability are also addressed. It is worth mentioning, that this is the outcome of the initial iteration of architecture design, a process running throughout the whole project lifetime. Since the overall CPSwarm methodology is built upon an iterative approach, the architecture design is subject to possible modifications in future iterations. Those changes will be documented in a later Deliverable "Updated System Architecture and Design Specification", due by the end of June, 2018.

2 Introduction

This deliverable documents the results of activities taken in Task 3.1 *Analysis of relevant engineering methods and tools, technologies and standards* and Task 3.2 *CPSwarm System Architecture*. The general purpose of this document is to provide a common technical background and an initial architecture design as starting point for future developments.

In the first part of the deliverable, available technologies to develop swarms of CPS (Cyber Physical Systems) are introduced. Since the goal of CPSwarm is to establish new methodologies and tools to tackle challenges in swarm development, the interest is mainly in the currently used tools and frameworks for swarm development, as well as possible tools and methodologies for improvement. Different surveys were conducted and the results are documented in section 3. They include the following surveys:

- Survey of Tools and Frameworks currently adopted for CPS design / development;
- Survey of existing modelling standards / patterns;
- Preliminary Analysis of design methodologies applicable to the CPS domain.

In the second part of this document, an initial version of architecture design is described. To build a solid foundation for future developments, the focus is on defining functionalities of different components and interfaces between them. The design process follows the standard ISO/IEC/IEEE 42010 "System and software engineering – Architecture description" [1]. The following viewpoints are chosen to represent the architecture design, which is documented in section 4:

- Context view
- Functional view
- Information view
- Deployment view

Besides the aforementioned actions, to ensure high-quality development, important cross-cutting concerns such as security and scalability have already been addressed during this initial design effort in section 5 and section 6.

At the end of the document, a plan for future steps and possible exploration directions are presented in section 7. It is worth noting that the CPSwarm architecture is expected to be revised, extended and refined in the future, accounting for evidences and lessons learned while accomplishing project activities.

2.1 Related documents

ID	Title	Reference	Version	Date
D2.1	Initial Vision Scenarios and Use Case Definition	D2.1		M4
D2.3	Initial Requirements Report	D2.3		M6
D3.2	Updated System Architecture Analysis & Design Specification	D3.2		M18
D3.7	Test and Integration Plan	D3.7		M9
D4.1	Initial CPS modeling library	D4.1		M9
D4.4	Initial Swarm Modelling Library	D4.4		M10
D5.1	CPSwarm Modelling Language Specification	D5.1		M12
D5.2	Initial CPSwarm Modelling Tool	D5.2		M9
D6.1	Initial Simulation Environment	D6.1		M9

3 Analysis of Relevant Engineering Methods, Tools, Technologies and Standards

Designing and deploying swarms of CPS is still an open research domain with many challenges to be solved and issues to be addressed. Several interesting approaches are emerging from literature and some de-facto modelling approach is slowly appearing from the intense scientific activity being carried on the topic. Nevertheless, defining the CPSwarm architecture requires to gather an up-to-date view on currently adopted design methodologies, engineering tools and emerging standards to ensure that even the most recent findings are considered in the overall CPSwarm workbench design.

On such premises, an intense technology survey activity was carried in the first months of the project and was exploited as foundation for defining the actual CPSwarm workbench architecture. In order to provide a solid ground for design decisions and architectural choices described in subsequent chapters, the most relevant findings of such survey activities have been summarized and organized in this chapter. Rather than aiming at providing an extensive overview of the current state of the art in CPS design, this chapter describes a snapshot of current efforts, which are particularly related to CPSwarm, also considering possible technologies and solutions upon which the project outcomes can be built.

The remainder of this section is organized as follows. Section 3.1 describes the methodology adopted for surveying the current solutions and approaches, including CPS design techniques and tools, while Section 3.2 provides details on tools and frameworks currently adopted for day-to-day development of CPS, with a focus on rovers and drones. Section 3.3 provides an up-to-date summary of models and model-based technologies adopted in the CPS design domain. Finally, Section 3.4 surveys methodologies adopted for tackling design in such a specific domain with a particular focus on swarm and self-organizing behaviors and on human-robotics interaction.

3.1 Methodology

Many methodologies may be followed for surveying the current state-of-the-art in a given knowledge domain. Systematic literature review, for example, allows providing objective, systematic, transparent and replicable results. It involves a systematic search process to locate studies, which address a research question, as well as a systematic presentation of the characteristics and findings of the results of such a search.

In the first phase of CPSwarm, a less structured approach is adopted to gather the current state-of-the-art in CPS design. The main motivation for such a lower replicability approach was two-folded. On one hand, a first set of results is desirable quite early in the project, to drive the initial architecture design. On the other hand, since the CPS design domain is very actively researched, there was a not negligible risk that adopting a longer surveying methodology would have implied exclusion of relevant works that might still be under review or unpublished.

For these reasons, involved partners are engaged to report their specific knowledge and their own findings on engineering tools, methods and standards relevant for the project. Such a process has been implemented both offline, through task assignments and on-line with synchronization meetings and dedicated, remote, interviews.

Throughout the project lifespan, the expectation is to corroborate this initial analysis with evidence coming from surveys performed by adopting sound and systematic methodologies, thus offering third party stakeholders the ability to verify, exploit and re-use the outcomes of such CPSwarm activity.

3.2 Survey of Tools and Frameworks currently adopted for CPS design / development

CPSwarm involves industrial partners coming from different branches including drones, rovers and car system manufacturing. This section describes the tools and frameworks that are currently used for designing, simulating and building runtime environments for CPS in these branches.

3.2.1 Design

CPS design can be roughly defined as the process of setting up CPS hardware and software for a specific purpose. Technologies and methods used for such a purpose are clearly dependent on the actual application domain and on specific targeted results, thus generating a high variability of viable solutions and approaches that should be considered in bootstrapping the CPSwarm design. Nevertheless, some common platforms and approaches emerge, such as the adoption of the ROS – Robotic Operating System environment in both drone and rover design.

ROS is defined as *“an open-source, meta-operating system for robots. It provides the services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers”*¹. ROS, therefore, enables easier development of robotic solutions thanks to the ability of supporting easier solutions for issues that are typical of software development in the robotics domain. In particular, it helps dealing with:

- *Distributed communication.* Some robots, in fact carry multiple computers on board, each controlling a subset of the robot sensors and/or actuators. When multiple robots attempt to cooperate on a shared task, communication is needed to coordinate the single efforts, etc.
- *Software reuse.* ROS standard packages provide stable implementations of many important robotic algorithms. Moreover, the ROS message passing interface is becoming the de-facto standard for robot software interoperability.
- *Rapid testing.* ROS systems separate the low-level direct control of hardware from the high-level processing and decision-making, using separate “programs”. Such separation permits to easily switch low-level control with simulators, to enable testing the higher levels of a robotic system. Moreover, ROS supports sensor logging and playback, thus enabling experimentation with data captured in real-world, without requiring to perform actual field-test campaigns. The change between simulated and real-hardware is almost seamless.

Subsequent sections, try to provide a short overview of most relevant technologies adopted in the flying drones, rover and autonomous vehicles domains, highlighting commonalities and overlap, where possible.

3.2.1.1 Drones

Building a drone requires more than just flight controls — it requires at least: vision and/or GPS based navigation, obstacle avoidance and path planning. For this reason, in CPSwarm it has been selected the PX4 Project², one of the leading flight control platforms, as the base platform upon which building the complex coordination and swarm behaviors targeted by the project. The PX4 architecture is designed to be ready for complex environments and can empower any vehicle from racing and cargo drones to ground vehicles.

PX4 is a complete autopilot solution composed by several parts:

- PX4 Flight Stack³ (the flight control system / autopilot)
- MAVLink⁴: a highly efficient, lightweight and blazing-fast robotics communication toolkit
- QGroundControl⁵: a modern, mobile and desktop user interface to configure the system and execute flights
- HITL (hardware in the loop) /SITL (software in the loop) simulation based on jMAVSim⁶ and ROS⁷ + Gazebo⁸.

¹ <http://wiki.ros.org/ROS/Introduction>

² <http://px4.io/>

³ https://dev.px4.io/en/concept/flight_stack.html

⁴ <http://qgroundcontrol.org/mavlink/start>

⁵ <http://qgroundcontrol.com/>

⁶ <https://pixhawk.org/dev/hil/jmavsim>

⁷ <http://www.ros.org/>

Simulation and modelling of software functions (e.g. control algorithms, AHRS - Attitude and Heading Reference System, collision avoidance) and modelling tools is typically accomplished by exploiting Simulink/MATLAB⁹, a block diagram environment for multidomain simulation and Model-Based Design. These tools are also used for generating production-level code, which is subsequently tested using HITL simulation, based on jMAVSim, or SITL simulation, based on Gazebo and ROS.

To simulate a drone behavior and to create the corresponding model the CPSwarm drone manufacturer (DGSKY) typically exploits SDF¹⁰: a special purpose XML dialect that allows describing objects and environments for robot simulation, visualization, and control. SDF, in particular, permits to represent all “physical” aspects of a robot, be it a simple chassis with wheels or a humanoid. In addition to kinematic and dynamic attributes, sensors, surface properties, textures, joint friction, and many more properties can be defined for a robot. These features allow exploiting SDF for simulation, visualization, motion planning, and robot control.

3.2.1.2 Rovers

As stated above, ROS – Robotic Operating System – can be considered as the de-facto standard for the design of ground robots. In ROS all the robot parts are defined in a so-called URDF description file. This plain text file describes two types of components: links and joints. On one hand, links define the fixed parts of a robot, with their weight and inertia. They also include the 3D model of each robot part described in the STL format. This, for example, allows computing possible collisions and showing the complete 3D model of the robot in visual simulators. Joints, on the other hand, represent how links are connected, in a hierarchic model. In other words, each joint defines a connection from a parent link to a child link. For example, a wheel is modelled as a revolution joint that connects the “wheel link” that contains the 3d model of wheel, with the “base link” that defines the frame of the robot to which the wheel is attached. It is easy to understand that this hierarchy-based modeling can iteratively be applied to describe any kind of robot, of whatever complexity. The overall URDF file describing a given robotic platform, e.g., a rover, is loaded as a parameter of the ROS system and made available to any node (computational unit) inside ROS.

ROS provides a complete toolchain that greatly facilitates interaction (control) and monitoring of robots, e.g., through comfortable and editable graphic interfaces such as: Rviz¹¹ and RQT¹². Both interfaces consist of a 3D environment visualizer that allows viewing how the robots perceives, measures and interacts with the environment. Based on a pluggable modules architecture, ROS, together with its companion software, provides the possibility of not only developing custom packages but also reusing third party tools, very easily.

3.2.1.3 Automotive

Software systems in a modern vehicle are among the most complex software systems existing today. Software in complex Electronic Control Units can contain millions of lines of code (modern vehicles contain up to 100 million lines of code) with a complex structure of real time components, acting on thousands of attributes which are adjusted to refine the car’s character, fulfil the regulations, etc.

For representing the requirements driving the design of such complex systems and to manage and orient the corresponding software design processes, tools such as IBM Rational Doors¹³ are widely exploited. For high-level software design, general UML tools such as IBM Rational Rhapsody¹⁴ or Modelio¹⁵ are used to represent

⁸ <http://gazebo-sim.org/>

⁹ <https://www.mathworks.com/products/simulink.html>

¹⁰ <http://sdformat.org/spec>

¹¹ <http://wiki.ros.org/rviz>

¹² <http://wiki.ros.org/rqt>

¹³ <http://www-03.ibm.com/software/products/en/ratidoor>

¹⁴ <http://www-03.ibm.com/software/products/en/ratirhpfami>

software structures (e.g., UML component of class diagrams) and/or behaviors. Since typical automotive software architectures are mainly based on the AUTOSAR (AUTomotive Open System ARchitecture)¹⁶ standard, some specific tools are used for automotive software development. For example, for model-based specification of electronic vehicle systems and their components, as well as for the design vehicular network architectures, the Vector's PREvision¹⁷ tool is frequently used. For designing AUTOSAR-compliant (low-level) software structures, including port interfaces, data types, composition of components, or scheduling, tools such as, e.g., DaVinci Developer¹⁸ tool from Vector are used. Whereas, simulation and modelling of software functionality (e.g., control algorithms), is typically accomplished with tools such as Ascet¹⁹ from ETAS or Simulink/MATLAB from MathWorks. These tools are also used for the generation of real-time, production code. In CPSwarm, the plan is, therefore, to integrate Simulink/MATLAB for the development of the exemplary CPSwarm automotive application.

3.2.2 Simulation

CPS simulation can be described as a complex domain where a plethora of solutions and approaches to simulation are available. Simulation tools by themselves are strongly related to the purpose of the simulation process. Depending on the aspects under consideration several different types of simulations can be performed, from finite state simulation for structural analysis to particle filtering analysis for motion in fluids, to physics simulation for evaluating action-reaction chains in the envisioned CPS-World interactions. Providing an extensive analysis of currently adopted simulation engines and techniques, is an overwhelming task and can be considered slightly out of topic for this document. Instead, a more focused analysis on most used simulation tools and technologies in the domain lying at the intersection between robotics and swarm algorithm research is exactly the kind of information needed to bootstrap the CPSwarm architecture design.

Based on this assumption, an initial survey of simulation platforms dealing with robotic systems and able to be exploited in swarm algorithms testing, and design, has been performed and it is currently being finalized (the corresponding task T6.1, just started at M6). In this analysis, the CPSwarm partners aimed at identifying the major solutions for CPS simulation, with a focus on the drone / rover domain and at evaluating them with respect to a set of initial, qualitative, requirements drawn from each partner expertise.

Identified requirements for surveyed simulators are the following:

1. *Ease of use.* 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 modelled CPS. This is particularly important for physics simulators where specific robot customization shall be avoided;
4. *Scalability.* One of the approaches to swarm design envisioned in CPSwarm is based upon evolutionary optimization of the swarm parameters. This implies a relevant number of multi-robot simulations, equal to the number of solution candidates evaluated at each generation multiplied by the number of generations needed to reach a viable solution. For example, assuming a very simple set-up with 10 candidates evaluated at each generation and a mean number of generation of 100, targeted simulators will be required to execute 1000 different multi-cps simulations. It is easy to

¹⁵ <https://www.modelio.org/>

¹⁶ <https://www.autosar.org/>

¹⁷ https://vector.com/vi_preevision_en.html

¹⁸ https://vector.com/vi_davinci_developer_en.html

¹⁹ https://www.etas.com/en/products/ascet_software_products.php

understand that in such conditions the ability to run in parallel, on multiple machines is one of the crucial aspects that must be evaluated.

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.

As can be easily noticed, such requirements are not yet refined enough to drive a selection of a precise simulation tool or technique (most likely co-simulation will be needed due to the different aspects being considered in a CPS design). However, they can already be leveraged to pre-screen the huge amount of currently available simulation solutions and to prepare the right software infrastructure needed for integrating them in the CPSwarm workbench being currently designed.

In the following a preliminary list of candidate simulation tools, partly derived from literature research, e.g., exploiting the analysis reported in [2] and in [3], and partly obtained by direct investigation carried by the CPSwarm partners, is reported. It must be noticed that all these tools are mainly aimed at evaluating motion inside a certain environment while few or none of them allows simulation of other relevant aspects, e.g., inter-robot communication, etc.

Motion simulators may be divided in 2 main categories: bi-dimensional and tri-dimensional simulators, they are respectively listed in Table 1 and in Table 2.

Table 1. Initial list of simulation tools analysed to direct the architecture design.

Simulation Engine	License	Language formats	Supported Robotic Platforms	Possibility to extend	Fidelity (functional, physical)	OS	Active development
Stage ²⁰	GPL v2.0	C++, Configurations in plain text	Pioneer, Chatterbox, iRobot Roomba, UMASS UBot	yes	Low, low	Linux, Windows	yes
TeamBots ²¹	Free for education and research	Java, configuration in source code or plain text files	Probotic Cyb, Nomad 150	yes	?, low	Linux, Windows, MacOS	no
Swarm ²²	GPL v2.0	Java – Objective-C	-	-		Linux, Windows, MacOS, Solaris	no
MRSim ²³	All rights reserved	Matlab	-	-			no
STDR ²⁴	GPL v3.0	C++, configuration in XML and YAML	Khepera, Pandora	yes	Low, low	Linux	yes
Rossum	GPL v2.0 /	Java	-	-			no

²⁰ <http://playerstage.sourceforge.net/doc/stage-svn/>

²¹ <https://www.cs.cmu.edu/~trb/TeamBots/>

²² http://www.swarm.org/wiki/Main_Page

²³ <https://de.mathworks.com/matlabcentral/fileexchange/38409-mrsim-multi-robot-simulator--v1-0-requestedDomain=www.mathworks.com>

²⁴ http://wiki.ros.org/std_r_simulator

Playhouse ²⁵	MIT						
MobotSim ²⁶	All rights reserved	Visual Basic	-	-		Windows	no

Table 2. . Initial list of 3D Simulation tools analysed to direct the architecture design.

Simulation Engine	License	Language formats /	Supported Robotic Platforms	Possibility to extend	Fidelity (functional, physical)	OS	Active development
Gazebo	Apache License v2.0	C++, configuration files in SDF	Many including: PR2, Pioneer2 DX, IRobot Create, TurtleBot, etc.	yes	High,high	Linux, Windows, MacOS	yes
ARGoS	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
Webots	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
Swarmbot3D	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
MuRoSimF	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
DPRSim	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
Mission Lab	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
MORSE	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
SimSpark	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
V-REP	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
Breve	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
Simbad	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
Marilou	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
jMAVSim	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.
peekabot	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.

Clearly, the analysis is not yet complete (see n.d. values in Table 2) and actual completion of the survey is expected by M9, when the deliverable D6.1 Initial Simulation Environment will be published, summarizing the outcomes of such survey and describing a very first implementation of the CPSwarm Simulation Environment. Nevertheless, it appears quite evident that a single-simulator solution is not viable, thus requiring architectural solutions able to deal with several different simulators. Moreover, as already stressed before, analyzed simulators only consider robot movement in a selected test environment, Other relevant parameters such as network connectivity, inter-robot communication, and human interference are not considered and would probably require dedicated simulation tools.

In summary, this very first analysis of existing simulation tools provided the two following architectural requirements:

1. Support to different simulators shall be provided
2. Co-simulation shall be included in the workbench design

²⁵ <http://rosum.sourceforge.net/sim.html>

²⁶ http://www.mobotsoft.com/?page_id=9

3.2.3 Runtime

3.2.3.1 ROS and PX4

The CPSwarm consortium acknowledges that ROS is currently being adopted as the de-facto standard for professional-level robot programming. For such a reason, ROS is considered as the main target runtime for CPSwarm developed CPS, although many other runtimes are targeted, also outside of the robotics domain.

According to this choice, flying platforms targeted in CPSwarm are based on ROS compatible software and hardware stacks. In CPSwarm, the drones employed in the project pilots exploit the PX4 Project flight control, one of the leading flight control platforms available on the market. Its architecture (Figure 1) encompasses the following main components:

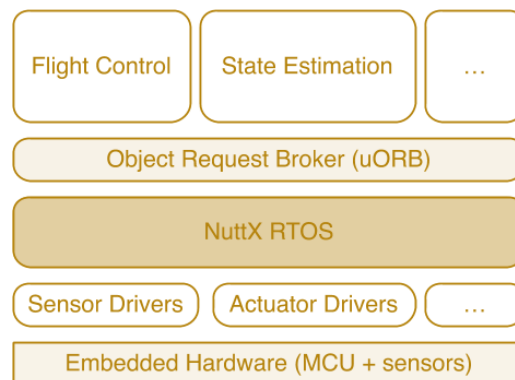


Figure 1 - The PX4 software architecture.

μORB Middleware

The uORB is an asynchronous publish()/subscribe() messaging API used for inter-thread/inter-process communication. The object request broker provides a data structure for data distribution. It follows the one-to-many publish-subscribe design pattern: a publisher wanting to share information advertises a topic. A topic is defined as a semantic message channel, such as 'attitude' or 'position'. A subscriber can subscribe to a topic, and after the subscription is established ask at his own pace for new data (polling), or be woken from the thread sleep state at the instant new data is available.

MAVLink

MAVLink²⁷ is a very lightweight, header-only message marshalling library for micro air vehicles / drones. MAVLink follows a modern hybrid publish-subscribe and point-to-point design pattern: Data streams are sent / published as topics while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with retransmission. Because MAVLink doesn't require any additional framing it is very well suited for applications with very limited communication bandwidth. Its reference implementation in C/C++ is highly optimized for resource-constrained systems with limited RAM and flash memory.

MAVROS

The mavros ROS package enables MAVLink extendable communication between companion computers running ROS, MAVLink autopilots and MAVLink enabled GCS.

3.2.3.2 AUTOSAR

In the automotive industry, runtime environments used on electronic control units depend on the available resources and safety or performance requirements. In resource-constrained electronic control units for

²⁷ <http://qgroundcontrol.org/mavlink/start>

safety-relevant controls (e.g., for engine control, stability or braking), AUTOSAR is used. AUTOSAR is a standardized automotive software architecture and it provides hard real-time, deterministic environment with static scheduling, defined low-level services (e.g., for memory management or communication) and interfaces between software components. On control units with more computational resources, VxWorks or real-time Linux derivatives are used. TTTech's platform solutions can run different operating systems (AUTOSAR, VxWorks, Linux, etc.).

3.3 Survey of existing modelling standards / patterns

3.3.1 Existing modeling standards

Depending of the methodology used, the company size, the company history, etc., CPS design may be done across many domains adopting many different tools. This variety of domains and tools implies the co-existence of several modelling standards or patterns which might be relevant in the context of CPS swarm modelling.

The following table lists the existing standards identified as potentially relevant for CPSwarm project. For each standard, a quick summary is provided, to explain its purpose, as well as the organisation responsible of its definition.

Table 3. Modelling standards for CPS swarm design.

Standard	Summary	Organisation
AADL	The SAE AADL is an extensible architecture analysis and design language for embedded and real-time systems. The core language provides a precise semantic specification for modelling task and communication architectures and their mapping onto distributed execution platforms.	SAE www.sae.org
ARINC 653	ARINC 653 (Avionics Application Standard Software Interface) is a software specification for space and time partitioning in Safety-critical avionics Real-time operating systems. It allows hosting multiple applications of different software levels on the same hardware in the context of a Integrated Modular Avionics architecture.	AEEC www.aviation-ia.com
AUTOSAR	AUTOSAR (AUTomotive Open System ARchitecture) defines a set of specifications for describing modules (basic, hardware and software), defining application interface and building a common development methodology.	AUTOSAR Consortium http://www.autosar.org/
BPEL	Business Process Execution Language is an OASIS standard executable language for specifying interactions with Web Services. Processes in Business Process Execution Language export and import information by using Web Service interfaces exclusively. BPEL is an orchestration language that specifies an executable process that involves message exchanges with other systems, such that the message exchange sequences are controlled by the orchestration designer.	OASIS www.oasis-open.org
CORBA CCM	CORBA Component Model (CCM) is an addition to the family of CORBA definitions. It was introduced with CORBA 3 and it describes a standard application framework for CORBA components.	OMG www.omg.org
FMI	FMI (Functional Mock-up Interface) defines a standardized interface to be used for model exchange and cyber physical system simulation.	Modelica Association http://fmi-standard.org
IP-XACT IEEE 1685-2009	IP-XACT is an XML format that defines and describes electronic components and their designs. The standard ensures delivery of compatible component descriptions from multiple component vendors, enables exchanging complex component libraries between electronic design automation (EDA) tools for SoC design (design environments), describes configurable components using metadata, and enables the provision of EDA vendor neutral scripts for component creation and	SPIRIT Consortium www.spiritconsortium.org

	configuration (generators, configurators).	
MARTE	MARTE (Modelling and Analysis of Real-Time and Embedded systems) is a specification of a UML profile that aims to replace UML capabilities for model-driven development of Real-Time and Embedded Systems (RTES), and for analysing schedulability and performance of UML specifications. It provides capabilities such as the support for specification, design, and verification/validation stages, the definition of non-functional properties, time and time related concepts and analysis frameworks.	OMG www.omg.org
ROS	ROS (Robot operating System) is a set of software frameworks for robot software development. ROS provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. ROS uses the Unified Robot Description Format (URDF), which is an XML format for representing a robot model as well as COLLADA (COLLABorative Design Activity), which is an interchange file format for interactive 3D applications (ISO/PAS 17506).	ROS community http://www.ros.org/
SCXML	SCXML (State Chart XML) is an XML notation for describing complex finite state machine in a generic way.	W3C https://www.w3.org/TR/scxml/
SysML	SysML is a UML Profile for System Engineering intended to support modelling of a broad range of systems, which may include hardware, software, data, personnel, procedures, and facilities. Its main purpose is to assist in the system requirements engineering and design processes, having as background the reference system processes and principles defined in the ISO System Engineering – System Life Cycle Processes [ISO15288], as well as industrial practices.	OMG www.omg.org
SystemC	SystemC is a set of C++ classes and macros which provide an event driven simulation kernel in C++. These facilities enable a designer to simulate concurrent processes, each described using plain C++ syntax. SystemC processes can communicate in a simulated real-time environment, using signals of all the data types offered by C++, some additional ones offered by the SystemC library, as well as user defined. In certain respects, SystemC deliberately mimics the hardware description languages VHDL and Verilog, but is more aptly described as a system-level modelling language.	OSCI www.systemc.org
UML	Unified Modelling Language is a general-purpose modelling language used to specify, visualize, modify, construct and document the artefacts of an object-oriented software intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as actors, business processes, components, activities, programming language statements, database schemas, and reusable software components.	OMG www.omg.org
Verilog	Verilog is a hardware description language used to model electronic systems. Verilog is most commonly used in the design, verification, and implementation of digital logic chips at the register transfer level (RTL) of abstraction. It is also used in the verification of analogue and mixed signal circuits.	IEEE www.ieee.org
VHDL	VHDL is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field programmable gate arrays and integrated circuits, and has constructs to handle the parallelism inherent in hardware designs.	IEEE www.ieee.org

Some open source examples using these standards are detailed in D4.1 deliverable.

The CPSwarm modelling environment and language described in deliverable D5.1 will be built on top of these standards (and possibly other relevant ones, currently emerging) adding new concepts related to the swarm design domain, wherever needed.

3.3.2 Existing Swarm Intelligence Models

In literature, swarm intelligence models are described as the process of adopting models found in nature in swarm behavior of, e.g., insects [4]. Swarm intelligence models are computational models to undertake distributed (optimization) problems in a swarm of, e.g., CPSs. The state-of-the-art process follows the steps described in Figure 2. Nature inspired and still inspires in imitating its behavior to solve complex real world problems. A closer look is taken on the actions and an analysis of observations enables the creation of a model. The simulation gives then an assessment of how well the intended result can be achieved with a given behavior. This assessment is usually given through a fitness value. Finally, the algorithm is extracted to design a nature or bio-inspired swarm intelligence algorithm.

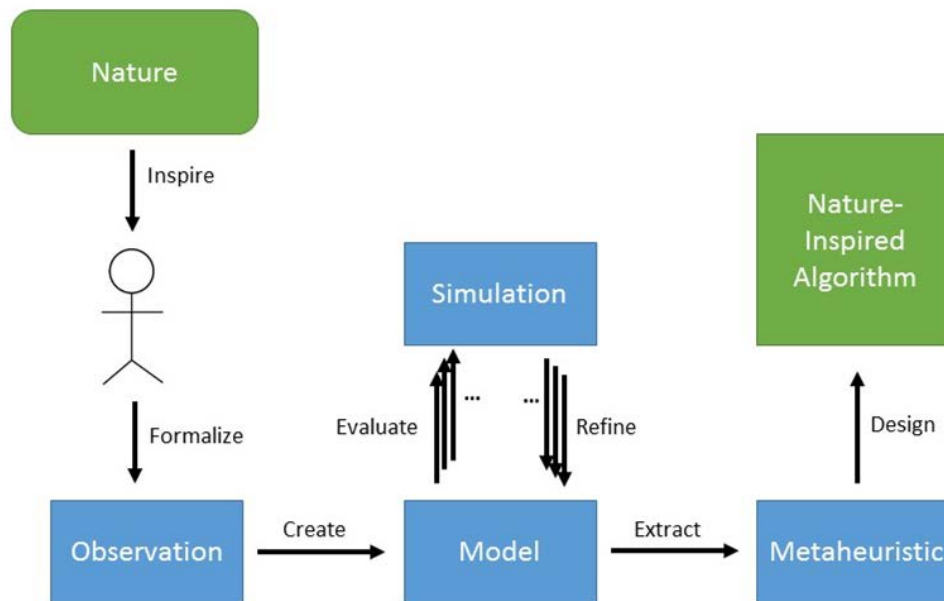


Figure 2 - Process of designing a swarm intelligence model and the corresponding algorithm (adapted from [5]).

Examples of swarm intelligence models include e.g.:

- Insect swarms (ant/bee colonies, etc.)
- Bacteria swarms
- Fish schools
- Bird flocks
- Quadruped herds

A more detailed overview of swarm intelligence models can be found in D4.4.

In all swarm intelligence models, the swarm is made of individual, simple agents. Through communication concepts they cooperate without a central control. Only through their interactions a collective behavior emerges, that can solve complex tasks. This makes the model easy to scale.

Five basic principles set the basis for swarm intelligence models [4]:

- 1) *Proximity*: ability to perform simple computation of time and space, respond to environmental stimuli

- 2) *Quality*: react to quality (fitness) factors
- 3) *Diverse Response*: distribute tasks
- 4) *Stability*: maintain the group behavior in case of environmental changes
- 5) *Adaptability*: change the group behavior in case of environmental changes

Beside biological models, a very popular boids model was introduced by Reynolds [6] to describe the flocking behavior of birds. The name "boid" is a shortened version of "bird-oid object". As in other swarm algorithms, the complexity arises from three simple rules:

- Rule 1 - Separation: Avoid Collision with neighboring birds;
- Rule 2 - Alignment: Match the velocity of neighboring birds;
- Rule 3 - Cohesion: Stay near neighboring birds.

Many subsequent models use these rules, vary them, add variables and extend the model's functionality.

In summary, each bio-inspired swarm intelligence model stand on its own. There is no ability to model swarm algorithms with a common modelling approach for swarm intelligence. Therefore, a future step for the CPSwarm project is to define a common approach with a common modeling language for modeling swarm intelligence.

3.4 Preliminary Analysis of design methodologies applicable to the CPS domain

3.4.1 Methodologies for Swarm & Self-Organizing Behavior Design

Swarm behavior results from individual actions of the agents in the swarm. The actions that the agents should take can be designed using two classes of approaches, behavior-based design or automatic design [7].

3.4.1.1 Behavior-Based Design Methods

Behavior-based design is an iterative bottom-up process where the agents' behavior is designed in a trial and error process. The behavior of each agent is designed and the resulting swarm behavior is observed. This is repeated until the desired swarm behavior is reached. In the following, the most commonly used behavior-based design methods are introduced.

Probabilistic Finite State Machine Design

Probabilistic finite state machines (PFSMs) are special types of state machines where the transitions between states happen non-deterministic with a given probability. The agent is described as a state machine where its actions are defined by the current state and the sensor inputs. The state is defined by the internal memory of the agent, the state transitions result from the sensor inputs. The transitions probabilities between states can be fixed or changing over time according to a mathematical function, e.g. the response threshold function [8]. PFMS design has been applied to aggregation [9], chain formation [10], and task allocation [11].

Virtual Physics-Based Design

Virtual physics-based design is inspired by classical physics. Each agent is considered as a particle exerting forces on other agents and experiencing forces by other agents and the environment. To compute the forces acting on an agent, this agent needs to be able to sense other agents and distinguish them from the environment. Different potential functions can be used to calculate the total force acting on an agent. This design method is mainly used for creating spatial formations such as pattern formation, collective exploration, or coordinated motion.

3.4.1.2 Automatic design methods

Automatic design methods allow to automatically generate the behaviors of individual agents from a high-level swarm behavior description. The required agent behavior that results in the desired swarm behavior is achieved without the need for a developer to explicitly define it.

Reinforcement Learning

Reinforcement Learning (RL) agents learn a behavior by receiving positive and negative feedback for the cumulative result over all agents. This is a trial-and-error process where an agent behavior is developed that maximizes the rewards received. Several challenges still exist when applying RL to swarm robotics: (i) It is difficult to decompose the global reward into individual rewards for each agent. (ii) The state space of RL problems is huge as the agent interactions introduce high complexity and make the environment non-stationary from the agents' perspective. (iii) The agents perceive only parts of the environment making it difficult to find the optimal behavior.

Evolutionary Robotics

Evolutionary robotics (ER) applies the Darwinian principle of natural selection, as described in biology, to robotics. Through multiple iterations an agent behavior is evolved that exhibits the desired global swarm behavior. The process begins with a population of random individual agent behaviors. Every behavior is evaluated experimentally and ranked according to a fitness function that evaluates the swarm-level behavior. The highest scoring behaviors are modified using genetic operators like cross-over or mutation forming the next generation of agent behaviors. It is therefore necessary that the agent behavior is implemented in a generic structure that allows application of genetic operators. Typically, artificial neural networks (ANN) are chosen which can be further distinguished into feed-forward ANN where no memory is required and recurrent ANN which take previous actions and observations into account. Most commonly, homogeneous swarms of agents are considered but also heterogeneous swarms can be evolved if the fitness function is designed properly. The challenges in ER are: (i) Evolution is a computationally expensive process since each agent behavior needs to be evaluated experimentally. (ii) Convergence of the evolutionary process is not guaranteed. (iii) ANN are difficult to understand and hence considered as black-box.

Other Learning and Automatic Design Methods

The ALLICANCE multi-robot architecture framework proposed by [12] features on-line learning that focuses on robust and adaptive task-allocation. [13] proposed the learning momentum framework which combines on-line learning with virtual physics-based design. [14] introduced an algorithm for on-line learning of certain parameters of agent behaviors to achieve diversity and specialization. [15] combined virtual physics-based design with ER in an off-line learning approach to achieve navigation with obstacle avoidance. [16] proposed a learning algorithm for multi-agent coordination that estimates the coordination cost to choose the best coordination method. [17] concluded that particle swarm optimization can achieve a higher degree of diversity in a swarm of robots compared to a genetic algorithm for on-line learning of parameters.

3.4.2 Methodologies for user-centered design applied to Human-CPS interaction

Last few years witnessed a sensible rise in the actual spread of drones, rovers and autonomous robotic platforms. Thanks to dramatic cost reductions, and to a much wider target community including householders, teenagers, amateurs, photographers, etc., robotic platforms, drones and rovers are becoming part of the everyday life of many people around the world. With this increasing co-existence of humans and robotic platforms, interaction between human and robots is becoming of critical importance, and several researchers are now targeting their efforts at finding better ways to cooperate and live with such cybernetic entities. Stemming from the well-established field of Human-Computer-Interaction (HCI), the Human-Robot-Interaction (HRI) community is seeking new and innovative ways to support natural cooperation between people and robots.

According to literature, several relevant cases might require effective cooperation between humans and drones (or rovers). Flying buddy [18], for example, envisions several scenarios in which humans might exploit drones to extend their physical abilities, e.g., by flying over the people field of view, reporting situation from above in a search and rescue task, or by helping people shopping. Drone-based flying displays have been

proposed as personal companions and to actively support people in emergency situations (e.g. search and rescue) or in tour guides [19]. In such scenarios, co-located interaction is the main pattern to be addressed, and can usually be mediated (e.g. using a remote or a phone) or direct e.g., using voice or gestural control.

In past years, HRI focused on anthropomorphic robots exploiting posture, and artificial facial expressions to establish a certain degree of engagement and empathy between the human and the robotic platform. Emotions, in fact, have been shown to have a vital role in human interaction and support processes such as perception, decision-making, empathy, memory, as well as social interactions [20]. This affective dimension, for example, can aid in intelligent interaction and decision-making [20], as well as in gaining social acceptance for robots in domestic environments [21]. Drones and rovers, however, present different physical characteristics that preclude designers to exploit facial features or gait to represent emotional states. Other characteristics, e.g., flight patterns and/or behaviour shall therefore be exploited to convey the “emotional” state of such platforms, thus enabling a much more natural interaction between them and the humans.

While one-to-one interaction between drones and humans requires a certain degree of mutual empathy and understanding, interaction between multiple rovers (or drones) and their supervisory team are even more challenging, as in such cases natural interaction encounters issues involving: remote control (in many cases robotic platforms are not in “sight” of their supervisors), interaction target (e.g., a single drone in a swarm or the swarm), etc. Conventional solutions address these interaction tasks by using some kind of physical device when communicating with the systems, e.g., keyboard or mouse, thus limiting the scope and dimension of such an interaction. However, more innovative and effective user interfaces, namely Natural User Interfaces (NUI), might be exploited to allow, e.g., non-expert users, who have a little knowledge on how to operate a robot (or a group of robots) to interact using natural gestures [22].

Given the CPSwarm application scenarios, depicted in Section 4.2 and better detailed in deliverable D2.3 – Initial Requirements Reports (public), NUI and, more in general, all methodologies and techniques aimed at defining a more effective interaction between humans and robots are among the best candidates for addressing Human-to-CPS interactions. Particularly, CPSwarm project is interested in direct interaction between humans and swarms of CPS to achieve coordinated operation in Search and Rescue and in Logistics, while the focus is on mediated interaction in the automotive use case, where humans shall be able to acknowledge (or not) platooning and other swarming behaviour while being inside the CPS.

For this purpose, CPSwarm is planning, on one side, to leverage on existing approaches at the state of the art regarding emotion encoding and direct, e.g., gesture based, interactions with swarms of CPS. For example, the work by [23] on emotion encoding would enable the definition of drone/rover personality models to be included in the CPSwarm modelling library. On the other hand, works like the one of [24] on NUI for interactions between drones/rovers and humans might define a viable set of interaction patterns, e.g., gesture based, to be included as supported interaction models in the CPSwarm library.

On the other side, within the CPSwarm context, the aim is also to implement and apply methodologies for assessing the effectiveness and intuitiveness of different, possible, HRI designs. Among the different solutions already published in literature, Adams [25] used Goal Directed Task Analysis to determine the interaction needs of officers from the Nashville Metro Police Bomb Squad. Sholtz et al. [26] used the Endsley’s Situation Awareness Global Assessments Method to determine robotic vehicle supervisors’ awareness of when vehicles were in trouble and thus required closed monitoring or interventions. Yanco and Drury [27] employed usability testing to determine among other things how well a search and rescue interface supported use by first responders. Drury et al. [28], also provided a first adaptation of the well-known, formal assessment method named Goals, Operations Methods and Selection Rules (GOMS) to the Human-Robot Interaction domain.

In summary, in the upcoming months the CPSwarm consortium is planning to concentrate on two main aspects regarding Human-CPS-Interaction. The first involves natural ways of collaboration between humans and drones/rovers, while the second is focused on suitable, possibly formal, methodologies to assess the

effectiveness of tentative designs, with the aim of supporting the overall swarm design activities carried within the project framework.

4 Architecture Design

4.1 Methodology

This section presents the key concepts of the methodology for software architecture design adopted in CPSwarm. In particular, the basic design references and principles adopted in the architecture definition process are presented, thus enabling the occasional reader to better understand the design choices taken by the project consortium. Experienced readers, aware of the ISO/IEC/IEEE 42010:2011 standard, might want to skip this section.

4.1.1 Software Architecture Design Standards

The process used for software architecture design is based on ISO/IEC/IEEE 42010:2011 "Systems and software engineering - Architecture description" [1]. Such a standard establishes a methodology for the Architectural Description (AD) of software-intensive systems. It implies a workflow, which includes the following steps:

- Identify and record the stakeholders for the architecture and the system of interest;
- Identify the architecture-related concerns of those stakeholders;
- Select and document a set of architecture viewpoints which can address the stakeholder concerns;
- Create architecture views (one view for each viewpoint) which contain the architectural models;
- Analyze consistency of the views;
- Record rationales for architectural choices taken.

The IEEE 42010:2011 standard extensively uses viewpoints and views to document different aspects of a software system allowing to focus on specific concerns and issues, while at the same time ensuring an overall consistency of the architecture design. This approach has been validated in several success cases, showing a higher quality of produced artifacts and outcome specification with respect to less-structured approaches trying to tackle all issues in a single pass. Viewpoints are collections of patterns, templates and conventions for constructing one type of view. One example is the functional viewpoint (and therefore the functional view) which contains all functions that the system should perform, the responsibilities and interfaces of the functional elements and the relationship between them.

4.1.2 Definitions

The following definitions are derived from [1] and from the extensions provided by Rozanski and Woods in [29].

- **Architecture:** fundamental concepts or properties of a system in its environment, embodied in its elements, relationships, and in the principles of its design and evolution. In other words, it's the concept of a system's structure, properties, interaction with its environment, etc.
- **Architectural Description:** work product used to express an architecture, such as component diagram, data flow diagram, etc.
- **Stakeholder:** an individual, team, organization, or classes thereof, having an interest in the realization of the system
- **Concern:** an interest in a system, which is relevant to one or more stakeholders. It might be a requirement (functional or non-functional) or an objective that a stakeholder has regarding the system.
- **View:** a set of models and descriptions representing a system or part of a system from the perspective of a related set of concerns
- **Viewpoint:** collection of patterns, templates and conventions for constructing one type of view
- **Model:** a simplified representation of an aspect of the architecture, could be in form of a UML diagram
- **System-of-Interest:** the system whose architecture is under consideration

The relationships between these concepts and the corresponding system-of-interests are shown in Figure 3.

The core of the modeling approach formalized within the IEEE 42010:2011 standard is composed of the architecture view and the architecture viewpoint concepts. According to [1], they are defined as follows.

- **Architecture viewpoint:** "Work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns"
- **Architecture view:** "A representation of a whole system from the perspective of a related set of concerns."

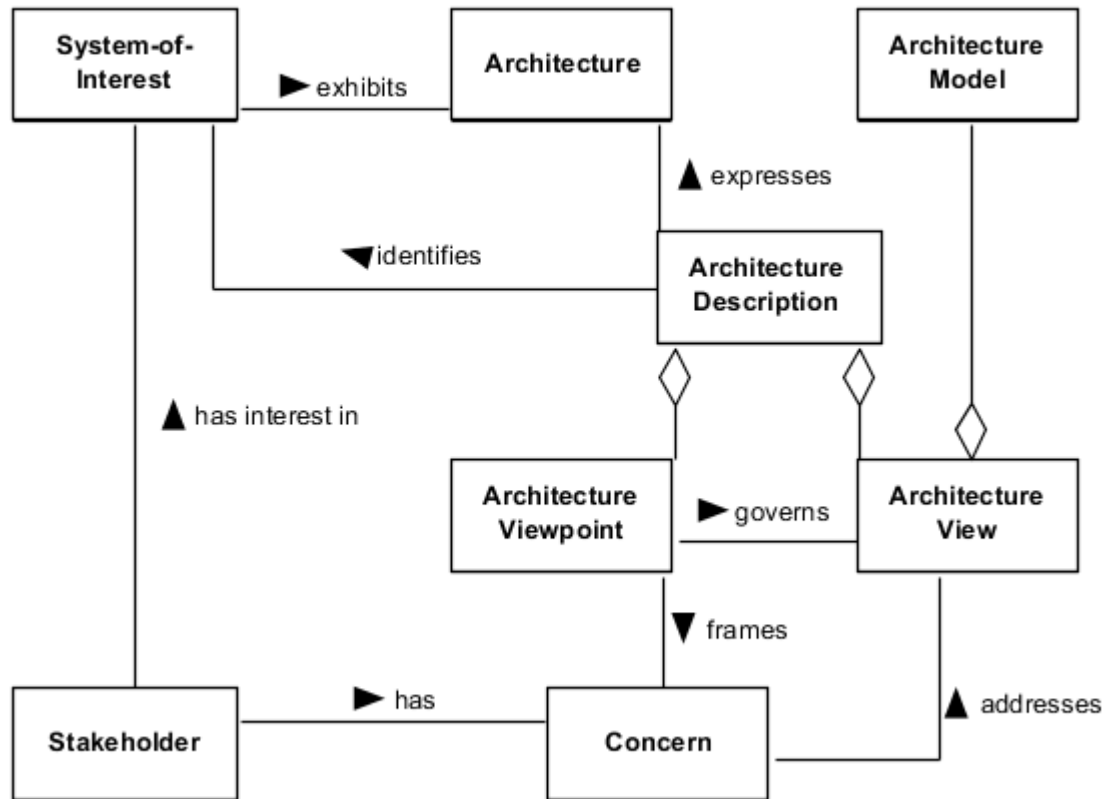


Figure 3 - Architecture description concepts (Adapted from [1])

A viewpoint defines, in other words, the aims, the intended audience, and the content of a class of views and defines the concerns that such views will address e.g. Functional Viewpoint or Deployment Viewpoint. A view conforms to a viewpoint and communicates the resolution of a number of concerns (and a resolution of a concern may be communicated in a number of views).

According to [29] using vision (view) and point of view (viewpoint) to describe the system architecture can bring many benefits such as:

- **Separation of concerns:** Separating different models of a system into distinct (but related) descriptions helps the design, analysis and communication processes by allowing designers to focus on each aspect separately.
- **Communication with stakeholder groups:** Different stakeholder groups can be guided quickly to different parts of the AD based on their particular concerns, and each view can be represented using language and notation appropriated to the knowledge, expertise, and concerns of the intended readership.
- **Managements of complexity:** By treating each significant aspect of the system separately, the architecture can focus on each in turn and so help conquer the complexity resulting from their combination.
- **Improved developer focus:** Separating into different views those aspects of the system that are particularly important to the development team, helps ensuring that the right system is built.

4.1.3 Software Architecture Design Process

In a software architecture design process there are several principles that should be followed to ensure a high quality design. The different stakeholders should be engaged in the system design and their concerns taken into account. There might be conflicting or incompatible concerns from different stakeholders which must be dealt with. Besides, an effective way to communicate decisions and solutions should be implemented and the whole architecture design process should be flexible and pragmatic to be able to deal with changing requirements. The entire process should be technology-neutral.

4.1.3.1 Architecture Design Process

Rozanski and Woods have based the architectural design process on the following definition [29]:

"Architecture Definition is a process by which stakeholder needs and concerns are captured, an architecture to meet these needs is designed, and the architecture is clearly and unambiguously described via an architectural description."

The foundation for this process is the ISO/IEC/IEEE 42010:2011 standard and the CPSwarm project used the process proposed by [29], which is aligned to such standard (see Figure 4).

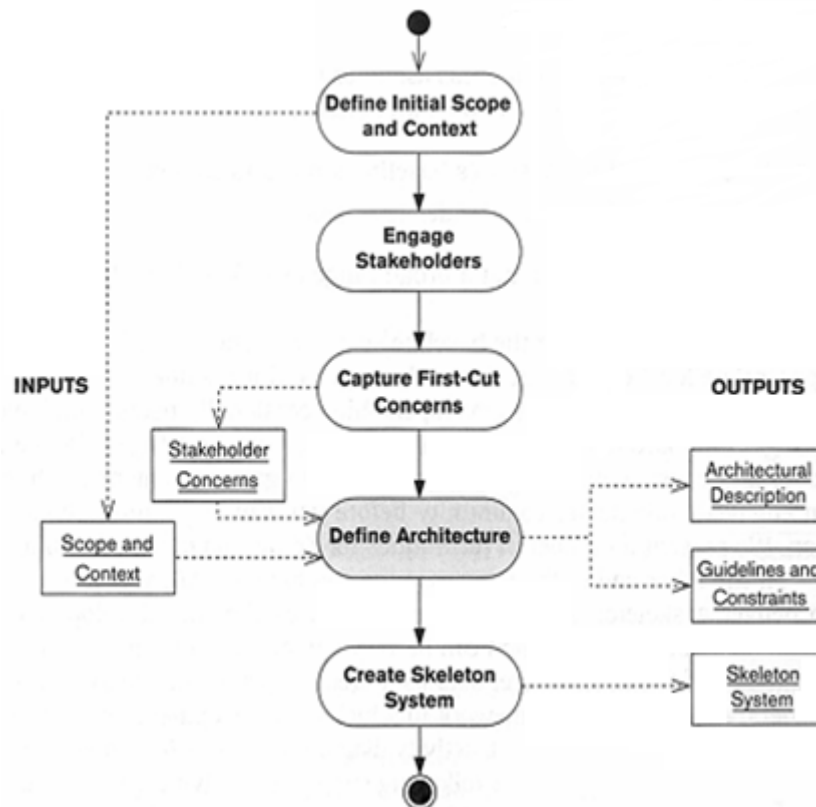


Figure 4 - Activities supporting architecture definition [29]

4.1.3.2 Architecture Viewpoints

Rozanski and Woods defined seven core viewpoints to document a software architecture. They are:

- **Context viewpoint:** The context viewpoint describes interactions, relationships as well as dependencies between the system-of-interest and its environment. The environment includes those external entities with which the system interacts, such as other systems, users, or developers.

- **Functional viewpoint:** Describes the system's functional elements, their responsibilities, interfaces, and primary interactions. A Functional view is the cornerstone of most AD. It drives the shape of other system structures such as the information structure, concurrency structure, deployment structure, and so on.
- **Information viewpoint:** The information viewpoint describes the data models and the data flow as well as the distribution of data along the system components. This viewpoint develops a complete but high-level view of static data structures and information flows within the system being designed.
- **Concurrency viewpoint:** Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how they are coordinated and controlled. This entails the creation of models that show the process and thread structures that the system will use and the inter-process communication mechanisms used to coordinate their operation.
- **Development viewpoint:** This is the viewpoint which addresses concerns from the developers' point of view. It describes how the software development process is supported, e.g. what conventions should be followed and how the artefact management will look like.
- **Deployment viewpoint:** Describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment. This view captures the hardware environment that a system needs (primarily the processing nodes, network interconnections, and disk storage facilities required), the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them.
- **Operational viewpoint:** Describes how the system will be operated, administered, and supported when it is running in its production environment. The aim of the Operational viewpoint is to identify system-wide strategies for addressing the operational concerns of the system's stakeholders and to identify solutions that address these.

As CPSwarm is in the initial phase of the relevant system design, it has been chosen to focus on the **context**, **functional**, **information** and **deployment viewpoints** to document the initial CPSwarm architecture design.

4.2 Stakeholders and Requirements

While CPS have been adopted more and more widely, in large-scale domains, the development methodologies and the typical design procedures adopted for the CPS domain are not yet mature enough. As result, the development of CPS tends to be complex, error-prone, and often requires a collection of tools to be actually achieved.

CPSwarm tries to solve this problem by proposing a new science of system integration and tools to support engineering of CPS swarms. CPSwarm tools will ease the development and integration of complex herds of heterogeneous CPS that collaborate based on local policies and that exhibit a collective behavior capable of solving complex, industrial-driven, real-world problems.

The CPSwarm project aims at defining a complete toolchain that enables the designer to: (a) set-up collaborative autonomous CPS; (b) test the swarm performance with respect to the design goal; and (c) massively deploy solutions towards "reconfigurable" CPS devices. Model-centric design and predictive engineering are the pillars of the project, enabling definition, composition, verification, and simulation of collaborative, autonomous CPS while accounting for various dynamics, constraints, and for safety, performance, and cost efficiency issues. Figure 5 shows the original concept of such a system from the Document of Action:

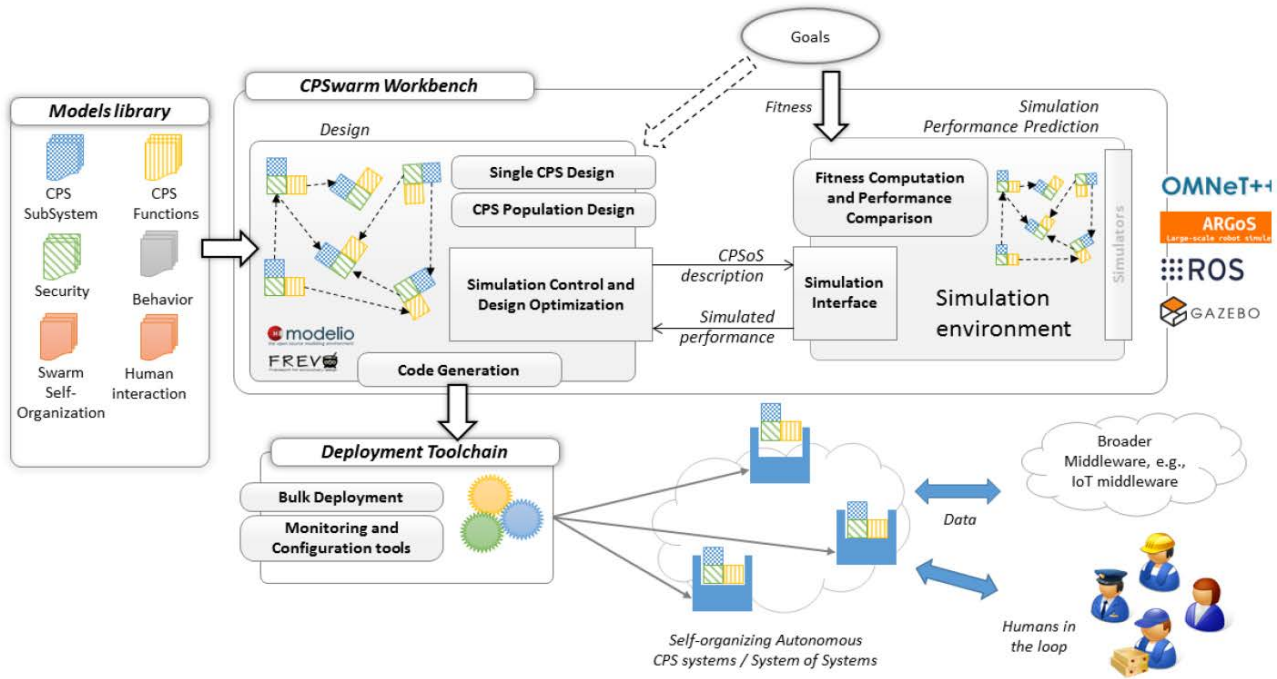


Figure 5 - CPSwarm conceptual architecture diagram

To guide the design and the implementation of the CPSwarm system, stakeholders, scenario and requirements have been identified in D2.3 based on the conceptual diagram above. This section provides a brief summary of the D2.3 outcomes, which are fundamental to the description of the architectural choices depicted in subsequent sections.

4.2.1 Stakeholders

As part of the CPSwarm WP2 activities, multiple stakeholders of the CPSwarm system were identified. These users include several categories of people having interests in such a system and that are expected to interact both directly or indirectly with the CPSwarm toolset. The most important roles identified in this activity, and the relative responsibilities, are described in Table 4. A more detailed description of these roles can be found in D2.1. In particular, the interactions between stakeholders are described in D2.1, section 6.2.1 (Communication Flow between Stakeholders).

Table 4. Stakeholders of the CPSwarm System (extracted from D2.1)

Stakeholder	Description
Workbench Engineer	A person, group or an organization responsible for the development and maintenance of the workbench
Mission Planner	A person responsible for planning the mission. The mission includes: <ul style="list-style-type: none"> • Problem definition • Approach to solve the problem • Environment description • Mission parameters • Mission success condition
Swarm Designer	A person responsible for designing the swarm based on the mission defined by the mission planner. The

	swarm designer analyses the given problem and designs the structure and behavior of the swarm accordingly.
Domain Expert	A person, group or an organization who is an expert of the problem domain. He is responsible for providing expert advice about the domain e.g. rules, regulations, limitations etc.
Security Expert	A person, group or an organization responsible for providing expertise on safety and security of the swarm.
Swarm Modeler	A person who constructs the model of the swarm. This model is the visual representation of the structure and behavior of the swarm specified by the swarm designer.
Algorithm Optimization and Simulation Expert	A person or group who provides the expertise regarding the swarm algorithm. He decides the aptness of a certain algorithm given a specific swarm problem.
Swarm Developer	A person or a group responsible for adding logic to the generated code. This code is later on deployed on each component of the swarm.
Deployer	A person or group responsible for deploying the code of the swarm.
Swarm Commander/Operator	A person with the command control in his hand. He is responsible for directly manipulating the components of the swarm.

4.2.2 Requirements

According to the requirements emerging from discussions and interviews with the different stakeholders, multiple technical constraints were gathered and analyzed in D2.3. As first step in the architectural design, such technical constraints guided the definition and identification of the main CPSwarm components. They encompass:

Model Library: A library collecting reusable CPS models, swarm behavior algorithms, security guidelines etc. It enables high reusability and interoperability of core functions adopted in swarm development.

Modelling Tool: The modelling tool is a graphical interface offering functions to model the swarm structure, behavior, environment and other necessary parameters. The modelling tool provides an easy way for swarm experts to design a swarm without having profound expertise in programming and/or hardware specific knowledge.

Optimization Tool: Due to the complexity of swarm behaviors, in many cases it is very difficult, if not impossible, to define the exact algorithm to be adopted for each individual member of a swarm. For this reason, an optimization tool is envisioned to exploit methods based on Darwinian evolution to optimize the algorithm automatically, according to the configuration given by users.

Optimization Simulator: In order to evaluate an algorithm, the Optimization Tool needs an Optimization Simulator to evaluate the performance a swarm population within a “controlled” environment. Thanks to the availability of the Optimization Simulator, different generations of algorithms proposed by the Optimization Tool are ranked and optimized across multiple simulations, on the basis of achieved performances.

Code Generator: A swarm typically consists of multiple heterogeneous devices. To ease the process of deploying to devices based on different robotics platform, a code generator is envisioned to generate platform specific code from the optimized algorithm prepared by the Optimization Tool.

Deployment Tool: To minimize repetitive effort in deploying to multiple targets, a Deployment Tool is envisioned to automate the deployment process.

Hardware Abstraction Layer (HAL): In order to enhance the reusability of generated algorithms, it is necessary to have an abstraction layer which hides the hardware specific details of target devices and provides an interface for code deployment.

Monitoring Tool: After the deployment phase, a monitoring and configuration framework is necessary to monitor the current status of the swarm, as well as to send reconfiguration commands to modify the swarm behaviour, e.g., for re-purposing part of the swarm individuals.

Translator: It is the component responsible for translating the input/output of Optimization Tool into a format understandable by Optimization Simulator.

The respective technical requirements were specified in D2.3, which for the sake of simplicity are not repeated here. The high-level Data flows between these components were also identified in D2.3. It is illustrated in Figure 6. The architecture of the CPSwarm system is based on the above defined components and the specified data flow.

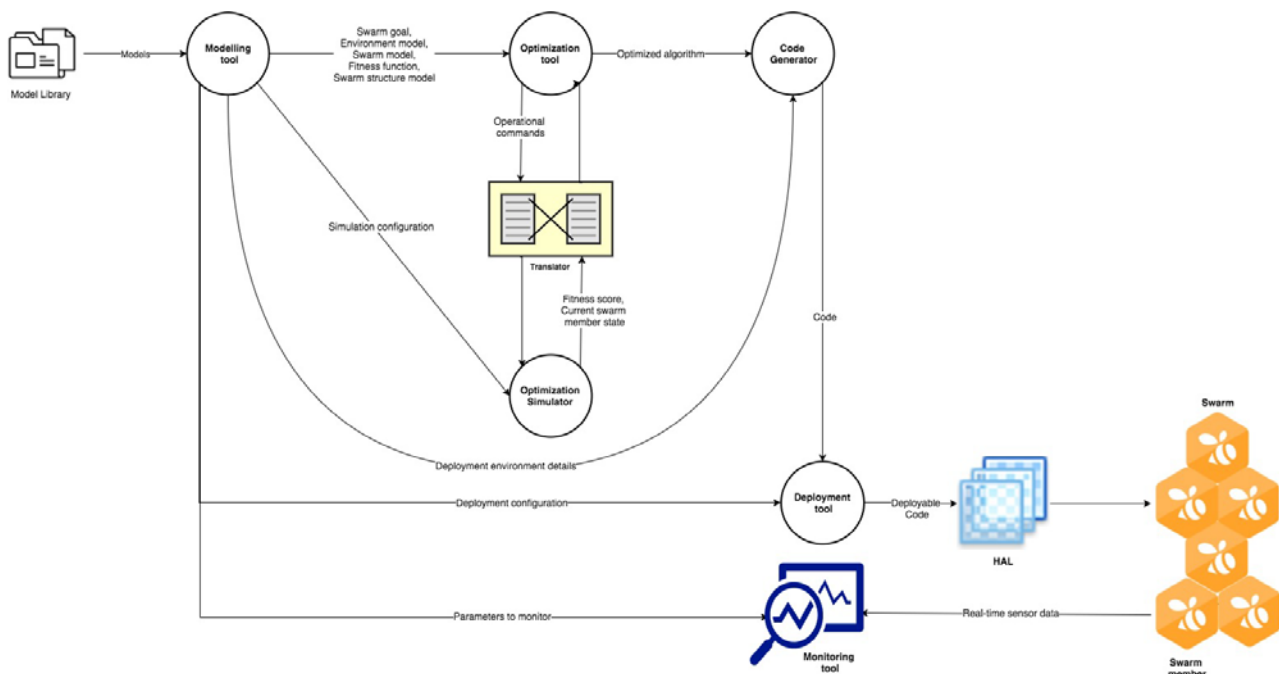


Figure 6 - Data flow between workbench components (extracted from D2.3)

It is worth mentioning that the above specified components and data flow represent a high-level logical structure of the system. During architecture design, naming adjustments were made to components in order to better describe them in technical and functional perspective. The table below summarizes these changes and the reasoning behind.

Table 5 - Mapping between components specified in D2.3 and those in architecture design

Component name in D2.3	Component name in architecture design	Reasons for change
Monitoring Tool	Monitoring and Configuration Framework	To emphasize that the component consists of sub-components to fulfill both monitoring and configuring functionalities
Hardware Abstraction Layer (HAL)	Runtime Environment	To emphasize that it consists of a stack of sub-components which fulfill multiple functionalities, such as program update, real-time data communication and hardware abstraction
Translator	Optimization Simulator Wrapper	To emphasize that the component acts as an intermediary layer between external simulators and CPSwarm components, and enables high modularity and extensibility.

4.3 Context View

As a first step of the architecture design, a context view is developed to highlight the interaction between the CPSwarm system, and the external actors, and systems. Given the stakeholders and requirements mentioned in the previous section, the context diagram is depicted in the following Figure 7.

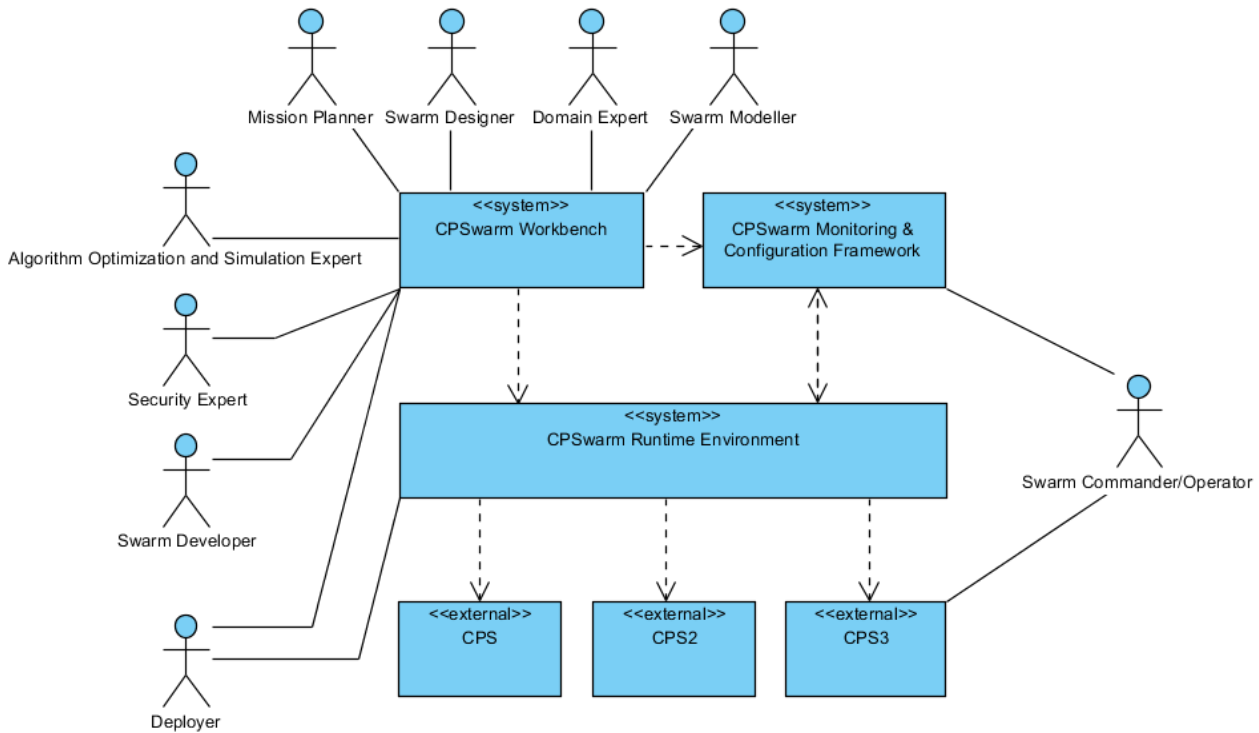


Figure 7 - CPSwarm system context diagram

The single software components of the CPSwarm system are grouped into three logical sub-systems:

The CPSwarm Workbench: a toolset including the Modelling Library, the Modelling Tool, the Optimization Tool, the Optimization Simulator, the Code Generator, as well as the Deployment Tool. It provides an integrated solution for swarm development from the modelling/design to the deployment phase. As shown in Figure 7, the CPSwarm Workbench is the major interaction point between external users and the CPSwarm system.

The CPSwarm Runtime Environment is based on the CPSwarm abstraction layer which provides a uniform API for algorithms generated from the CPSwarm Workbench. It enables reusability of algorithms as well as interoperability of heterogeneous devices. This is the interaction point between the external CPS devices and the CPSwarm system.

The CPSwarm Monitoring and Configuration Framework solves the problem of real-time monitoring and configuration of swarms during operation. It interacts mainly with Swarm Commander or Swarm Operator.

4.4 Functional View

4.4.1 High-level functional view

The CPSwarm architecture specification adopts a component-based architectural style where the system functions are provided by a set of well-defined, self-contained, modules named “components”. Components talk to each other through well-defined interfaces, which make them decoupled from each other. With such architecture style, maximum flexibility and extensibility could be achieved, since the system is not bound by certain component implementations. Instead, components could be easily replaced with new ones, as long as they share the same interface. For example, different simulators may be required to optimize the algorithm in different aspects. In this case, there is no need to change the whole CPSwarm system. Instead, only the component Optimization Simulator needs to be replaced with the desired simulator. In summary, a plug-in oriented architecture is being promoted within CPSwarm project and will be further developed within the updated versions of the architecture specification.

From a functional standpoint, the components participating in the overall system architecture are logically grouped into three main blocks: the CPSwarm Workbench, the CPSwarm Deployment Toolchain and the CPSwarm Runtime Environment (see Figure 8):

- The CPSwarm Workbench supports the core functions for modelling, optimizing, simulating and deploying swarms of CPS;
- The CPSwarm Runtime environment supports easier deployment on real-world platforms, providing functions and APIs designed to decouple the CPS business logic (the algorithms) from the CPS infrastructure (hardware and operating-system, for example);
- The CPSwarm Monitoring and Configuration framework, which groups components, located at the CPS side, that enable remote monitoring of the CPS operations and remote configuration of tunable parameters of designed algorithms.

It is worth mentioning that while the design process aimed to define a generic architecture based on the requirements collected in WP2, in this initial version, the focus has been on the adoption of “Evolutionary Optimization” approaches (see section 3.4.1.2) to generate swarm algorithms. This means that swarm algorithms candidates are developed using evolutionary principle. Each swarm algorithm candidate is tested in simulator and evaluated according to its performance. Candidates with good performance will be chosen to evolve until an optimized solution is found.

The following paragraphs better detail the three CPSwarm main blocks, providing insights on their inner organization, on the foreseen sub-components and on the interactions occurring both at component and at package level. The functionality of each component will be further introduced in the following sections.

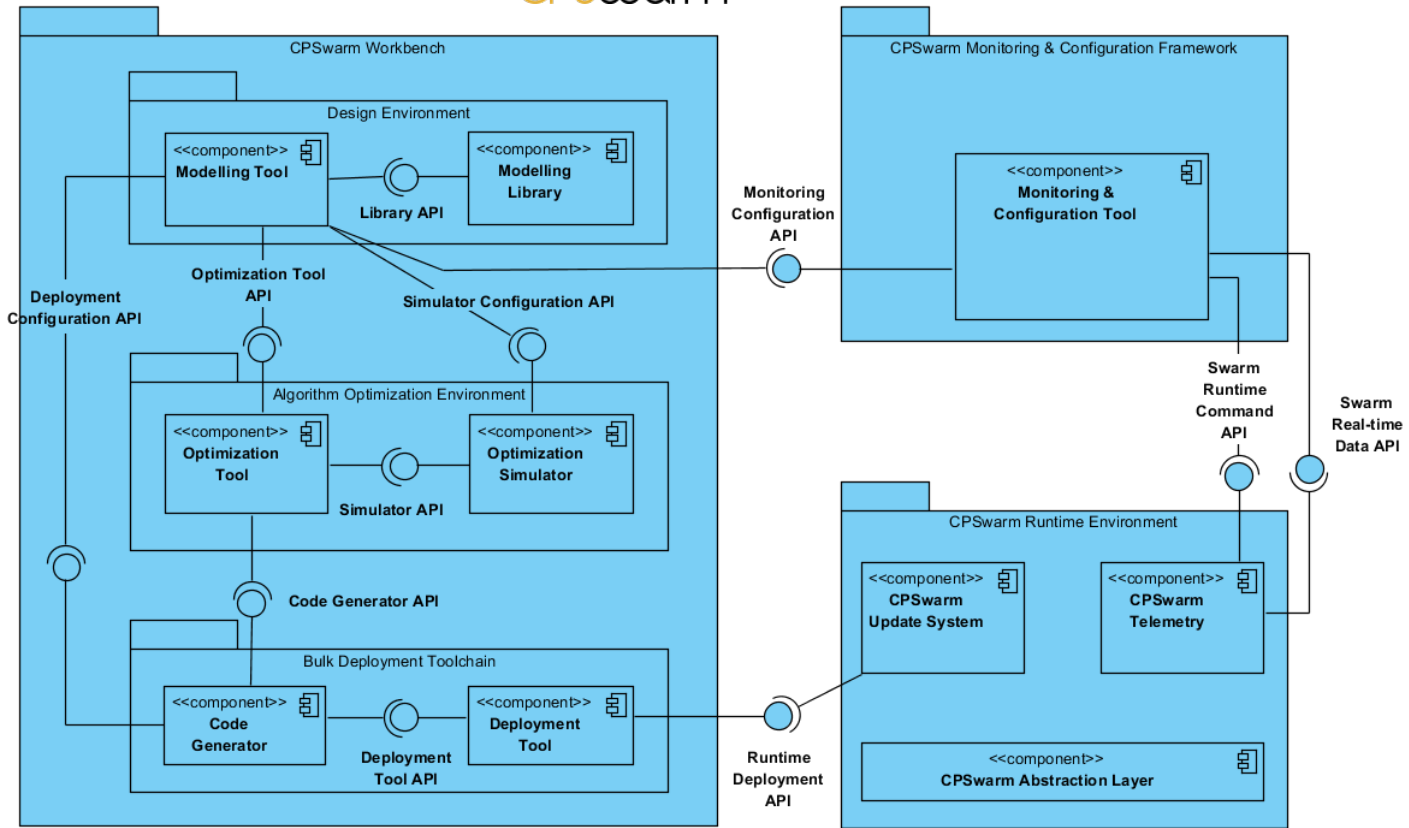


Figure 8 - Overview of components in CPSwarm system

4.4.2 Design Environment

4.4.2.1 Modelling Tool

The Modelling tool provides a graphical editor (GUI) for the CPSwarm models. This GUI allows users to edit the models with a set of languages specific for the CPS domain, as defined in deliverable D5.1.

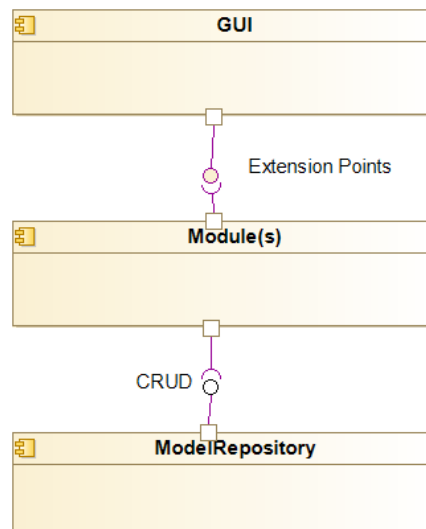


Figure 9 - Functional structure model of Modelling tool

The Modelling tools architecture, depicted in Figure 9, may be decomposed into three components following the Model-View-Controller design pattern. The *representation layer* is depicted as the GUI component. The

data layer is represented by the ModelRepository component for the model. The *control layer* incorporates the Module(s) component(s) deployed inside each CPSwarm project. Each type of component is described in the following sections.

Graphical User Interface (GUI) component

The GUI is the front-end which enables an intuitive and friendly ergonomics for CPSwarm model editing.

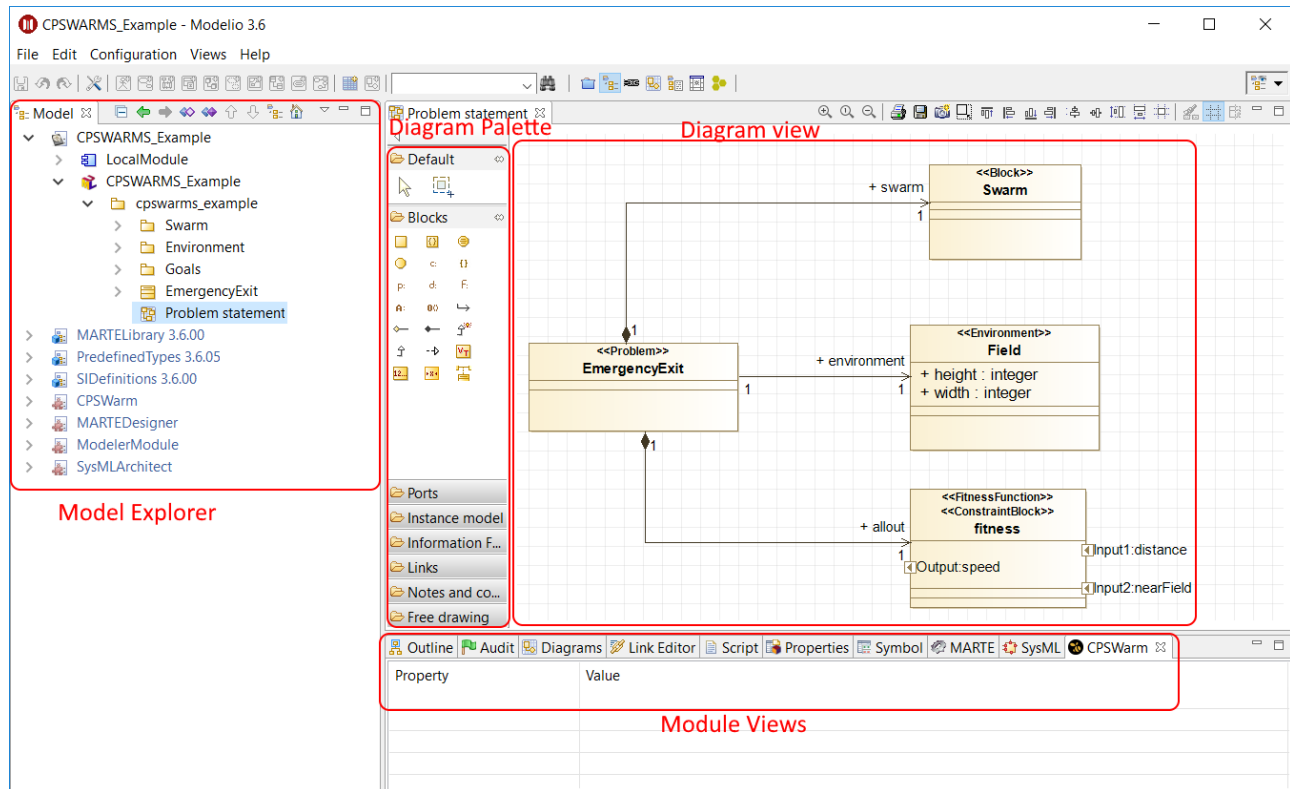


Figure 10 - Overview of Modelling tool view

Figure 10 shows an example of the CPSwarm Modelling tool GUI where the following elements are highlighted:

- The Model Explorer is used to browse the model, to create/delete model elements and diagrams and to select model elements for editing.
- The Diagram View is used to edit diagrams.
- The Diagram Palette contains buttons which provide access to model element creation commands. Modules can configure and extend this palette to create specific environment in which only relevant element (and not all) can be created.
- The Module Views (one per Module component deployed in the project) are used to view and to edit domain specific properties of an element selected in the Model Explorer or Diagram Views.

Module(s) component(s)

These components act on both Layout component and model repository component. They are notified of user interactions with the GUI and make corresponding calls to the GUI and to the ModelRepository components to reflect operations on the model. Modules have access to the GUI. They can create specific properties views, element and diagram creation contextual menus, property view tabs, can customize diagram palettes and implement constraints for specific UML Profiles. In addition, the modules can access the model repository at runtime via to query or modify the model programmatically. In addition, modules may reuse different services exposed by other modules

ModelRepository component

The local model repository layer is a runtime data model layer that manages the model elements and also notifies the GUI and modules about the changes made. Modules have an easy access to the model repository that allows querying and modification of the model.

4.4.2.2 Modelling Library

The Modelling Library's aim is storing and providing reusable parts of models. These reusable parts are sorted by category inside the library to facilitate their usage. Three kinds of artefacts have currently been identified as part of the CPSwarm modelling library:

- Agent;
- Environment;
- Goal.

Agent

These artefacts represent one or many individuals involved in a given swarm. Two kinds of agents have been identified:

- **Regular Agents:** These individuals act to fulfil a goal modeled by a cost-function, inside a given environment. Their communication can be centralized or peer to peer.
- **Malicious Agents:** These parts of models will subvert the operation of the swarm by acting against its interests. Possible malicious behaviors that can be modelled include:
 - Not performing the assigned task;
 - Reporting false data;
 - Causing physical damage to other members of the swarm;
 - Forwarding information to third parties.

Modelling the fact that the malicious behavior is spread between members of the swarm will also be considered.

Environment

This part of a CPSwarm model represents the field or environment in which the swarm will be involved. By simulating the swarm behavior in different fields (with different kind of constraint as size, obstacle or grounds for example), it will allow to test the swarm robustness against possible changes of the external conditions.

Goal

To evaluate the resulting behavior of a modelled swarm, one or many criteria have to be defined. Such criteria, called goal or cost-function, are computed exploiting the results from one or multiple simulation(s). A lot of criteria are possible, including:

- Time spent for achieving a given goal;
- Accuracy;
- Security;
- Robustness;
- Power consumption.

Each simulation of a swarm configuration will provide a result for each criterion. Then a ranking method, such as Pareto front, will be use to highlight the best swarm configuration.

4.4.3 Algorithm Optimization Environment

4.4.3.1 Optimization Tool

The Optimization Tool optimizes a control algorithm for an agent. It starts with a generic representation of a control algorithm and searches for a viable solution using a heuristic search algorithm. The resulting control algorithm is then deployed in a CPS. The control algorithm needs to be evolvable, i.e. modifiable by mutation and recombination. An example for such a representation is an Artificial Neural Network (ANN). To apply the iterative heuristic search to find an optimized configuration of the controller for a CPS, an optimization measure, called fitness needs to be defined for a given problem scenario. A fourth component takes care of evaluating a pool of possible algorithm candidates and provides a ranking for the evolutionary algorithm.

The result is a controller that implements local interaction rules that lead to the desired global behavior of the system. The controller can be evaluated with the Optimization Simulator by testing it in a reference scenario or by performing a statistically significant number of simulations on a given scale of parameters under predefined conditions.

The Optimization Tool uses a modular approach, where the distinct steps of evolutionary design are split into different components (see Figure 11).

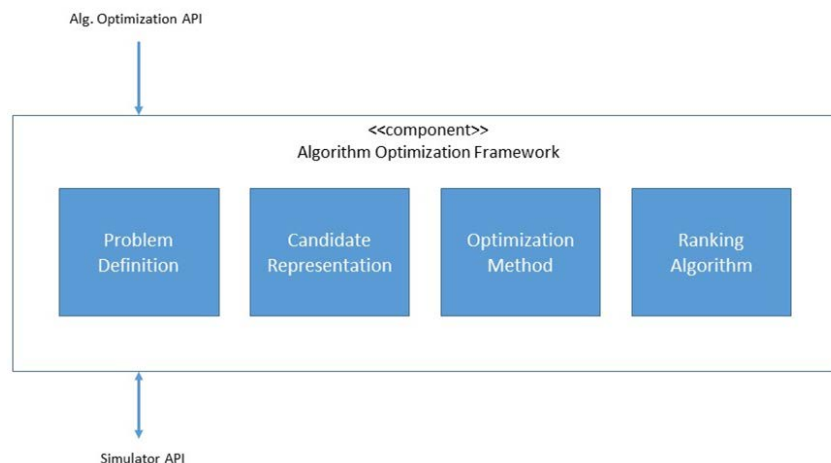


Figure 11 - Optimization Tool

Problem

The problem is defined by agents, environment, and the goal to be achieved. The structure and description of these three components are provided by the Modelling Tool. The problem is responsible for reading the optimization parameters and providing them to the optimization simulator, setting up and running the simulation via the simulator API, and finally retrieving the fitness score of every candidate solution. It uses an interface to connect with the optimization simulator.

- Reading the optimization parameters: Certain parameters are predefined by the modeling tool whereas others are left open to be fine-tuned during the optimization phase. The parameters are read from the configuration file to set up the simulation. The open parameters have a default value defined by the modeling tool. They are required as the swarm of CPSs might act in ways unforeseen by the modeler.
- Setting up the simulation: A simple problem can contain also its simulation code. However, for most cases involving a physical environment, there will be an external simulation used, which is interfaced by the Optimization Tool.

- Running the simulation: Each representation that has been evolved with evolutionary operators is evaluated through simulation regarding the fitness function. This is achieved with the optimization simulator. For introspection, the simulation can also be run using a GUI.
- Calculating the fitness score of every candidate solution: Using the fitness function defined by the modeling tool, the fitness score is calculated based on the data returned from the optimization simulator.

Candidate Representation

The candidate representation models the algorithm governing the behavior of an agent. Typically, there are multiple instances created, which are then treated as different candidates for the best solution, hence the name. The candidate representation is a generic structure which is evolvable, i.e. supporting mutation and recombination. Since the representation does not depend on the problem, it can be chosen from a library of pre-defined representations. The Optimization Tool supports several representations, for example a feed-forward ANN with a hidden layer, a fully meshed ANN with several hidden neurons or a fully meshed ANN implementing Hebbian learning.

Optimization Method

The optimization method performs a search algorithm that looks for the candidate representation that yields the highest fitness as defined in the problem description. It uses the genetic operators defined in the candidate representation to create new candidates in each generation to replace the worst performing candidates of the population. Through iterative heuristic search, it gradually obtains candidates with better performance. The optimization method can be applied independently of Problem and Candidate Representation, hence it can be chosen from a library of pre-defined methods. For this, the Optimization Tool provides different evolutionary algorithm and a random search approach.

Ranking Algorithm

The ranking algorithm evaluates the candidate representations and provides a ranking based on their fitness values. This functionality is required by every optimization method. Ranking algorithms differ in the way how they compare solutions. The particular algorithm can be chosen from a library of pre-defined ranking algorithms.

The Optimization Tool's GUI simplifies the design process and offers statistics and graph generation for easy evaluation of the chosen design. The main purpose of the Optimization Tool is to support the optimization process by guiding the end user through the individual steps of the evolutionary design. Such steps might require coding work by the software developer to implement the modeled details.

As illustrated in Figure 8, the Optimization Tool provides three APIs:

Optimization Tool API

The Optimization Tool API supports the generic implementation of new Problems, Candidate Representation, Optimization Methods or Ranking algorithms. When developing an algorithm for a CPS, the API for implementing a new Problem are used.

Simulator API

The Optimization Tool communicates with the Optimization Simulator through a Simulator API. There are two data flows between them:

- **Initialization and start of a simulation:** this is controlled from the Optimization Tool.
- **Continuous exchange of sensor inputs and actuator outputs:** Here control goes back and forth between Optimization Tool and simulator. This is referred to as the real-time data exchange in Section 4.5. If the initialization also transfers the algorithm to be evaluated the real-time exchange can be avoided at the cost of a longer initialization.

- **Fitness feedback from the simulation:** The fitness is calculated by the Optimization Simulator and submitted to Optimization Tool. A possible implementation could be to derive the fitness by analyzing a comprehensive log file of the simulation.

Algorithm Deployment API

The Algorithm Deployment API allows providing the output of the Optimization Tool for the Deployment Toolchain. An evolved algorithm will be provided as source code (possibly in a less complex subset of C) that is independent of the used representation. The deployment tool will then further process the algorithm.

4.4.3.2 Optimization Simulator

The Optimization Simulator is used to evaluate the performance of a generated controller algorithm/module. At each generation of the evolutionary optimization, the Optimization Simulator executes the current controller in a predefined environment. The result of the simulation is exploited to compute a fitness score, which allows the Optimization Tool to further optimize the controller. Figure 12 shows the structure of the Optimization Simulator.

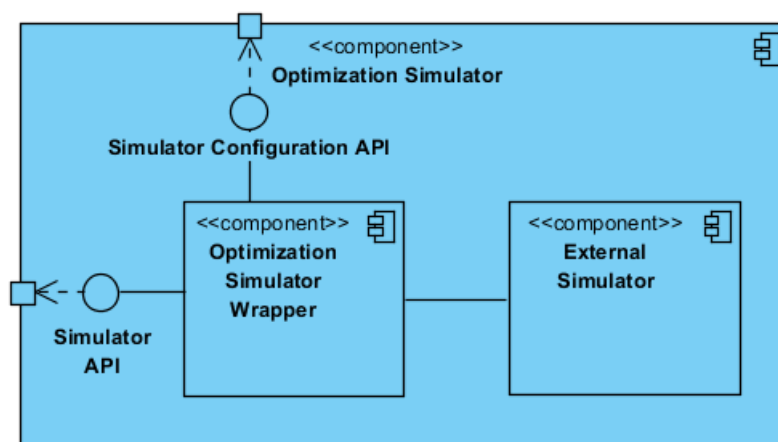


Figure 12 - Optimization Simulator

Depending on the problem to be solved, different external simulators can be exploited, such as ROS Stage for 2D problem and Gazebo for high fidelity 3D problem. To achieve high modularity and extensibility, a wrapper for the external simulator is necessary to serve as an intermediary communication layer between the external simulator and other components in CPSwarm system. This wrapper on one hand conforms to the Simulator API and Simulator, on the other hand communicate with the external simulator in use. In case a new simulator is required, only the wrapper component needs to be modified. As illustrated in Figure 8, the Optimization Simulator is surrounded by two APIs: Simulator API (described in section 4.4.3.1) and Simulator Configuration API:

Simulator Configuration API

The Modelling Tool provides different kinds of information from the models to the Optimization Simulator via the Simulator Configuration API. This includes setting up the environment and the agents. The environment setup requires a description of the surroundings in sufficient detail. E.g., for ground robots, a floorplan and some configuration parameters would be sufficient. The agent setup consists of the agents' physical properties such as size, position, and sensors/actuators but also of its behavior. The Optimization Simulator needs to be fed also details on the fitness function so that the required performance measures are recorded during a simulation.

4.4.4 Bulk Deployment Toolchain

The Deployment Toolchain is responsible for two tasks: the generation of platform-specific source code and the deployment of code onto designated runtime environments.

The CPSwarm project aims at creating a workbench that integrates with common CPS platforms with a varying degree of flexibility. Typically, swarms of CPS are based on different open-source or proprietary platforms. These include the Robot Operating System (ROS) frameworks, pure Linux systems, and closed educational or commercial platforms with limited functionalities. The component belonging to the bulk deployment toolchains has been designed with extensibility in mind. This will enable easier integration of other CPS platforms in future iterations of the project and/or will support extensions by third party developers.

4.4.4.1 Code Generator

Algorithms designed and optimized through the CPSwarm components located at the higher logic-levels of the CPSwarm Workbench will finally be deployed on real-world CPS systems, e.g., robotic platforms. Optimized algorithms cannot be directly deployed on a target CPS as, on one hand, they are developed and optimized to be portable across platforms, and on the other hand, they are typically evolved in a behavior / swarm-centric manner, with less focus on platform-related details such as event delivery subsystems, sensor communication interfaces, etc. The glue layer between such high-level algorithm definitions (e.g., provided in a C-like syntax) and the actual code running on real CPS is partly provided by the abstraction libraries available on the target CPS platform (see the CPSwarm Runtime Environment) and partly generated by the CPSwarm Code Generator. More specifically, the CPSwarm Code Generator is responsible to generate executable code for the target CPS platform selected through the Deployment Toolchain Configuration APIs. Such executable code leverages on the functions and libraries part of the CPSwarm Abstraction Layer and performs the needed generation, translation and code-wiring tasks. These activities may radically vary depending on the target language; however, exploited Abstraction Layer APIs are equivalent between the different supported platforms, thus easing the overall code generation process.

Several code generation and code-wiring patterns are applied by the CPSwarm Code Generator depending on the target CPS platform and the target programming language (e.g., C++ vs Python). Currently two main generation patterns are foreseen, respectively named "template-based" and "programmatic".

The **template-based generation pattern** applies to all the cases in which simple translation between the high-level algorithm specification and the target runtime environment is possible. Among these cases, it is for example possible citing the case of simple solutions where the needed glue code can be simply parametrized with respect to the initial algorithm definition. In other words, template-based generation applies whenever it is possible to define a simple set of target-templates to be filled with data extracted from the algorithm specification. Finite State Machine controllers (see Figure 13), for example, are easily handled through such a pattern (see Figure 14 for an applicable generation template).

```

<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0" initial="forward" name="Wandering">
  <state id="forward">
    <transition event="leftBumper" cond="leftBumper==1" target="turnRight"/>
    <transition event="rightBumper" cond="rightBumper==1" target="turnLeft"/>
  </state>
  <state id="turnRight">
    <transition cond="leftBumper==1" target="turnRight"/>
    <transition cond="leftBumper==0 || rightBumper==0" target="forward"/>
  </state>
  <state id="turnLeft">
    <transition cond="rightBumper==1" target="turnLeft"/>
    <transition cond="leftBumper==0 || rightBumper==0" target="forward"/>
  </state>
</scxml>

```

Figure 13 - Sample Finite State Machine specification.

```

#include <iostream>
#include <decision_making/SynchCout.h>
#include <decision_making/BT.h>
#include <decision_making/FSM.h>
#include <decision_making/ROSTask.h>
#include <decision_making/DecisionMaking.h>

#include <ros/ros.h>
#include <std_msgs/Bool.h>
#include <sensor_msgs/Range.h>

using namespace std;
using namespace decision_making;

volatile bool leftBumper, rightBumper;

ros::Subscriber leftBumperSub;
ros::Subscriber rightBumperSub;

EventQueue* mainEventQueue;
struct MainEventQueue{
  MainEventQueue(){ mainEventQueue = new RosEventQueue(); }
  ~MainEventQueue(){ delete mainEventQueue; }
};

/*****
*   FSM
*   *****/

FSM($scxml.name)
{
  enum STATES
  {
    #foreach($target in $scxml.targets.keySet())
    $target#if( $foreach.hasNext ),#end
    #end
    FSM_START($scxml.initial);
    FSM_BGN
    {
    #foreach($target in $scxml.targets.keySet())
    FSM_STATE($target)
    {
      {
        FSM_CALL_TASK($target)
        FSM_TRANSITIONS
        {
          #foreach($transition in $scxml.targets.get($target).transitionsList)
          FSM_ON_CONDITION($transition.cond, FSM_NEXT($transition.next));
          #end
        }
      }
    }
    #end
    FSM_END
  }
}

```

Figure 14 - Excerpt of template for template-based generation pattern of a state machine in ROS.

On the other hand, more complex algorithms specifications are sometimes difficult to handle with template-based generation. Cases involving many data structures, utility functions and specific implementations that need to be tailored for the input algorithm and for the given configuration data are seldom addressable by a "fixed", parametrized template.

In these cases, **programmatic generation** is much more suited, although it implies a stricter binding to the algorithm specification syntax and semantics. Such a stricter binding is traded-off by the flexibility of the generation pattern, which can in principle deal with all possible algorithms described using a given formal syntax.

To better understand the subtle difference between template-based generation and programmatic generation the following comparison might be useful. While template-based generation works in way similar to XSLT stylesheets, which permit to convert a given XML syntax to another one, using a “fixed” stylesheet; programmatic generation is like DSL interpretation: given a well-known domain-specific language, the code generation tool “compiles” an executable (or interpretable) set of code units. Every valid combination of the DSL directives will generate a valid (runnable or interpretable) set of code units.

From a functional standpoint, the CPSwarm Code generator takes an algorithm specification (through the Algorithm APIs) and a target platform specification as input (through the Deployment Toolchain Configuration APIs) and generates a software bundle ready to be deployed (and compiled / interpreted) on the target CPS. Its main components (not yet fully specified) include:

- A library of code-templates
- A library of DSL compilers
- The code generator core, which applies either a compiler or a template to generate code to be deployed
- The code generator bundler which “bundles” generated code units into a deployable entity (e.g., a python module with the corresponding requirements specification).

Figure 15 provides a preliminary architecture for the CPSwarm Code Generator while Figure 16 provides a very high level UML activity diagram describing the code generator behavior.

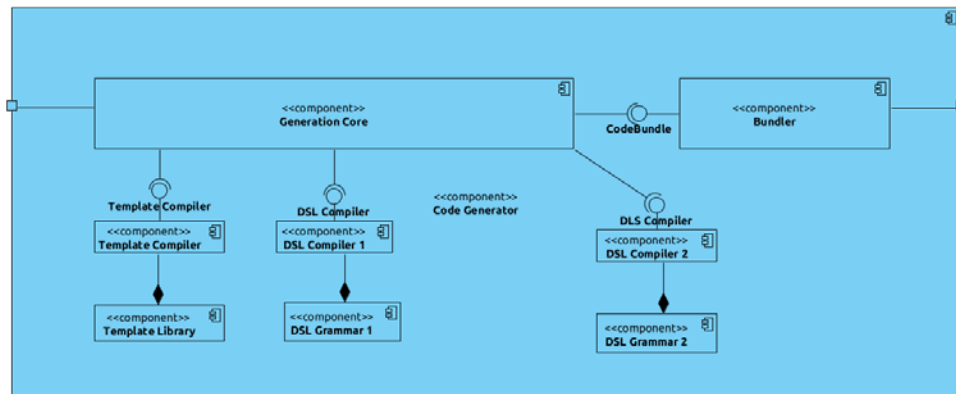


Figure 15 - Code Generator: preliminary architecture.

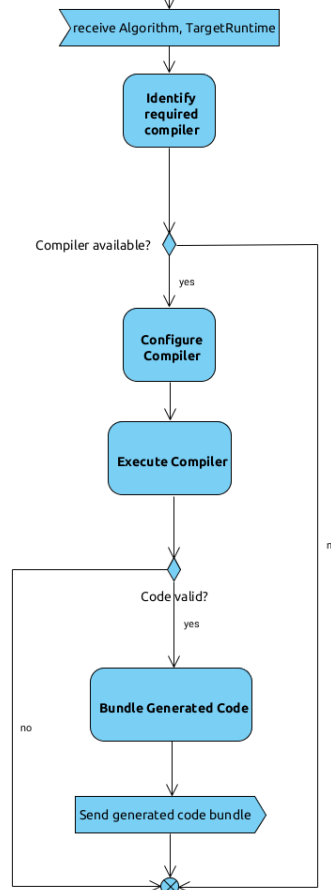


Figure 16 - High-level activity diagram describing the CPSwarm Code Generator behavior.

4.4.4.2 Deployment Tool

After the code is successfully generated, it must be deployed on different targets. To ease the efforts to execute and manage the deployment to a group of heterogeneous devices, the Deployment Tool automates the process according to the configuration provided by the system users. The initial design of the Deployment Tool offers an Over-the-air (OTA) update mechanism to deliver software to swarm members on-the-go and at large scale. How the CPSwarm system benefits from the OTA approach to overcome deployment scalability concerns is discussed in Section 6.2.

- **Over-the-air (OTA) update:** A publish/subscribe update strategy in which the new update (deployable code) is published to an "update-channel". All swarm members subscribing to this channel will be notified about the update and receive necessary instructions on how and when to execute it. While the publish update, strategy is preferred for scalability reason, it requires the swarm member to have the ability to subscribe to OTA events and perform updates during the runtime. More detailed description can be found in section 4.4.6.2.

Figure 17 shows subcomponents of the Deployment Tool:

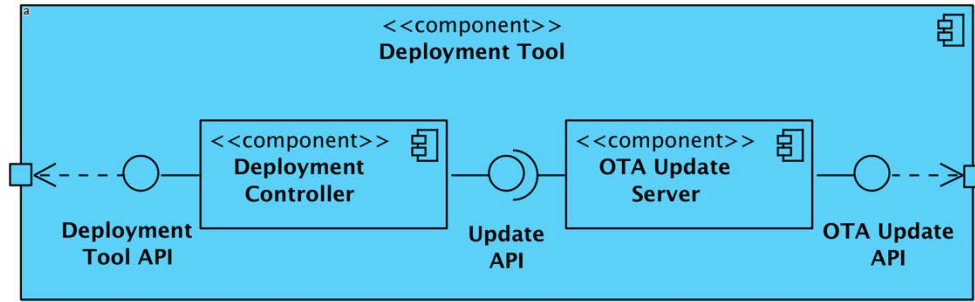


Figure 17 - Deployment Tool: preliminary architecture.

The Deployment Controller is responsible for (1) receiving deployable codes from the exposed interface and (2) responding to service update requests. This component ensures a request for updates to a target machine is responded with correct software distribution and version.

The OTA Update Server is in charge of (1) informing target machines when there is an update and (2) scheduling when the update should be downloaded by the target. Target machines periodically perform polling on the channel exposed by the OTA Update Server. After successful authentication and authorization, the server negotiates with Deployment Controller to check whether an update is available for this particular target. If an update is available, the server schedules a suitable time for this target, depending on the priority of the update and the load on the server. The target is then responded with the suitable download time. Below is an example of how an OTA Update request is performed by a target machine (For simplicity, Authorization headers are omitted):

GET /status

Response:

Location: /update?token=YWJjZGVmMTIzNDU2

```

{
  "status" : "security",
  "scheduled_time": "2017-07-18T08:56:35Z"
}

```

GET /update?token=YWJjZGVmMTIzNDU2

Response:

```

{
  "compressed_size": 743,
  "method": "gzip",
  "uncompressed_size": 5410,
  "data": << array of of bytes >>
}

```

Depending on the kind of update (i.e. batch or individual), the OTA Update Server may cache requests so that the load on the Deployment Controller is reduced. The OTA Update Server itself can be replicated and distributed for load balancing and scalability.

Exposed Interfaces

The Deployment Tool requires the generated code as well as deployment configuration as input. As output, it provides an API for update requests from swarm members. As a result, the Deployment Tool provides the following APIs:

- **Deployment Tool API:** The *provided* interface allowing submission of generated code from Code Generator as well as deployment configuration from the CPSwarm Modelling Tool.
- **OTA Update API:** The *provided* update API, with one endpoint for getting status of updates and another endpoint for getting the update when it is available.

4.4.5 Monitoring and Configuration Framework

The Monitoring and Configuration Framework is responsible for the runtime configuration and reconfiguration of single CPS and multiple CPSs (CPS swarms), as well as for monitoring the critical system and mission parameters. The framework takes advantage of existing protocols for system configuration and monitoring (such as e.g., NETCONF from the field of network management). In addition to protocols for communication between the Monitoring and Configuration Framework and the runtime system, appropriate modelling of the data to be configured and monitored is an important part. More details on related data modelling is given in Section 4.5.1.

The high level architecture of the Monitoring and Configuration Framework is shown in Figure 18. It consists of the *Configuration and Monitoring UI*, *Configuration and Monitoring Controller* and the *clients* for monitoring and configuration. Clients are responsible for low-level delivery of configuration and monitoring data and they communicate with the corresponding servers integrated with the runtime environment. For example, Monitoring Client *subscribes* to the Monitoring Server for the particular data to be monitored and Monitoring Server publishes the current values of this data (periodically or event triggered). Monitoring server will contain the specification of the data that can be published (including data structures, publishing frequency, etc.). Configuration and Monitoring Controller manages monitoring and configuration based on specifications (e.g., exchanged with the CPSwarm Workbench). User Interface components enable the user to input the monitoring and configuration specifications and to view the monitored data.

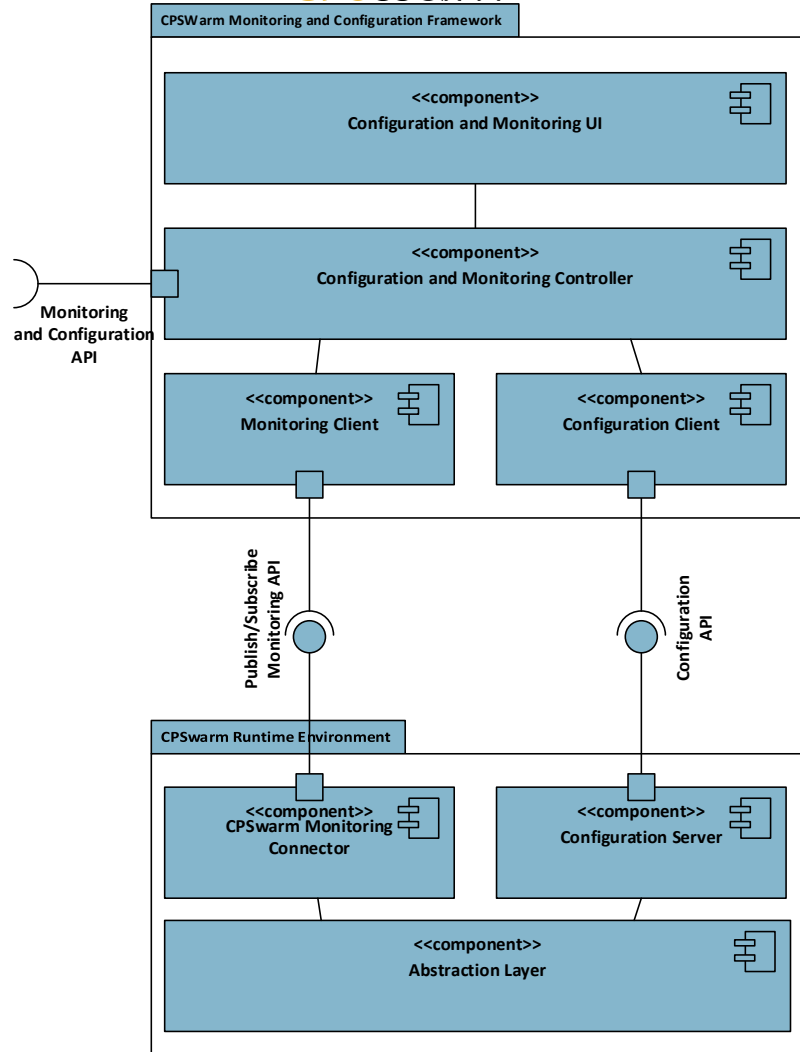


Figure 18 - Functional structure model of Monitoring and Configuration Framework

4.4.6 Runtime Environment

CPS designed and programmed with CPSwarm exploit the so-called CPSwarm Runtime Environment (CPSwarm RTE, see Figure 19) to support execution of generated swarm algorithms, to provide on-line update and re-programming capabilities and to support remote telemetry and data monitoring. These main functions are respectively provided by: the (a) CPSwarm Abstraction Layer, (b) the CPSwarm Update System and (c) the CPSwarm Telemetry components.

Following subsections better detail the architecture and the design choices / assumptions underlying each of these components.

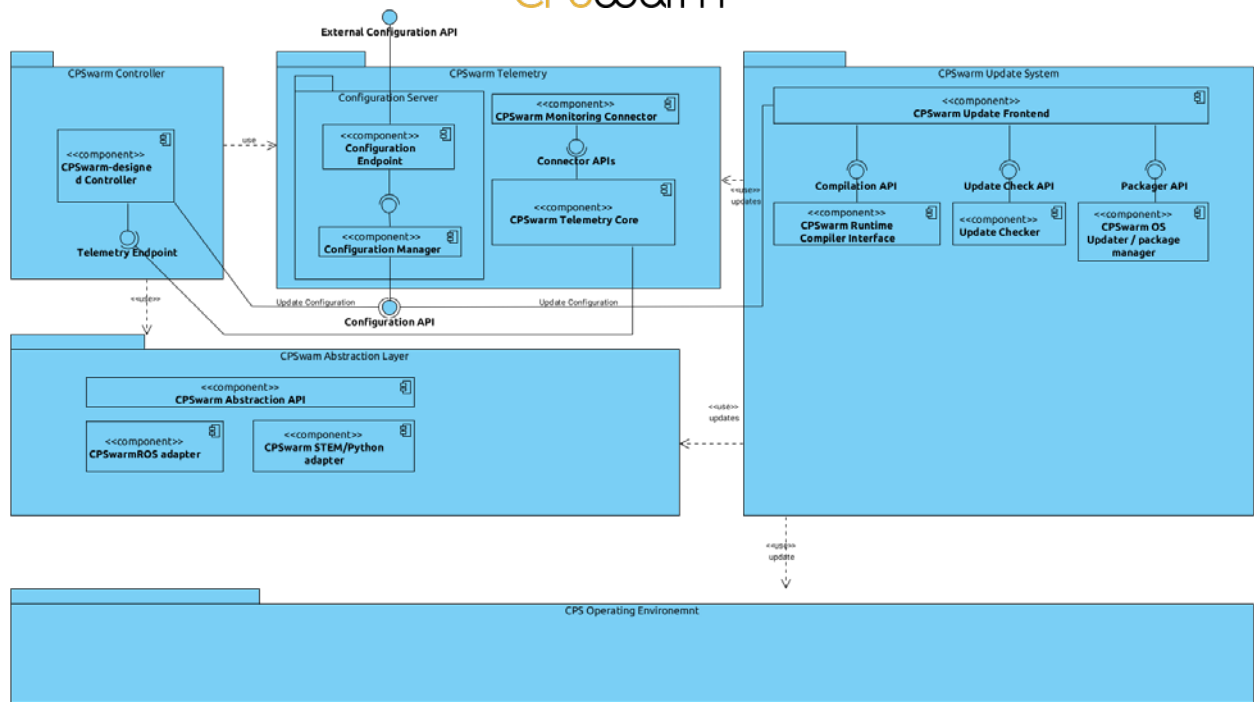


Figure 19 - The CPSwarm Runtime Environment.

4.4.6.1 Abstraction Layer

To ease the process of generating code to be deployed on target CPS, the CPSwarm project defines a so-called CPS abstraction layer whose purpose is to decouple the implementation of swarm algorithms from platform / system-specific function calls and primitives. The CPSwarm abstraction layer is composed by a set of platform-specific libraries that provide a common, high-level API that enables generated programs to uniformly interact with concrete CPS functions and subsystems. Depending on the CPS nature and operating environment the abstraction layer might be implemented as shared library, as adaptation middleware and so on. Several different implementations are foreseen mainly including the actual platforms considered by the project: STEM educational robots, ROS-powered drones and rovers, and automotive fog-nodes.

It is important to notice that depending on the application use case the abstraction layer might be realized as an “active” component of the architecture (i.e., as a service) or as a “passive” library to be included and exploited by generated programs. The overall internal architecture of the abstraction layer is depicted in Figure 20 and the actual “thickness” of the relative implementations depends on the target runtime.

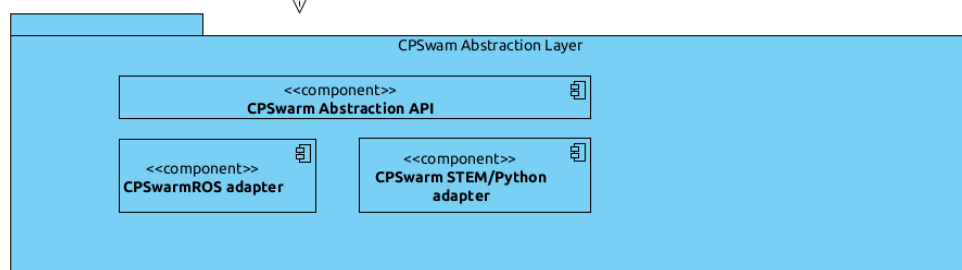


Figure 20 - The CPSwarm abstraction layer.

As an example, for ROS platforms, the CPSwarm abstraction layer will likely be very thin, by exploiting the abstraction and modelling efforts already carried by the ROS community. For STEM robots, on the other hand, the layer implementation is foreseen to be more complex and articulated.

4.4.6.2 CPSwarm Update System

The CPSwarm Update System is designed to support the deployment process on the CPS side. It might work in different modes depending on configuration provided through the initial deployment or through subsequent updates.

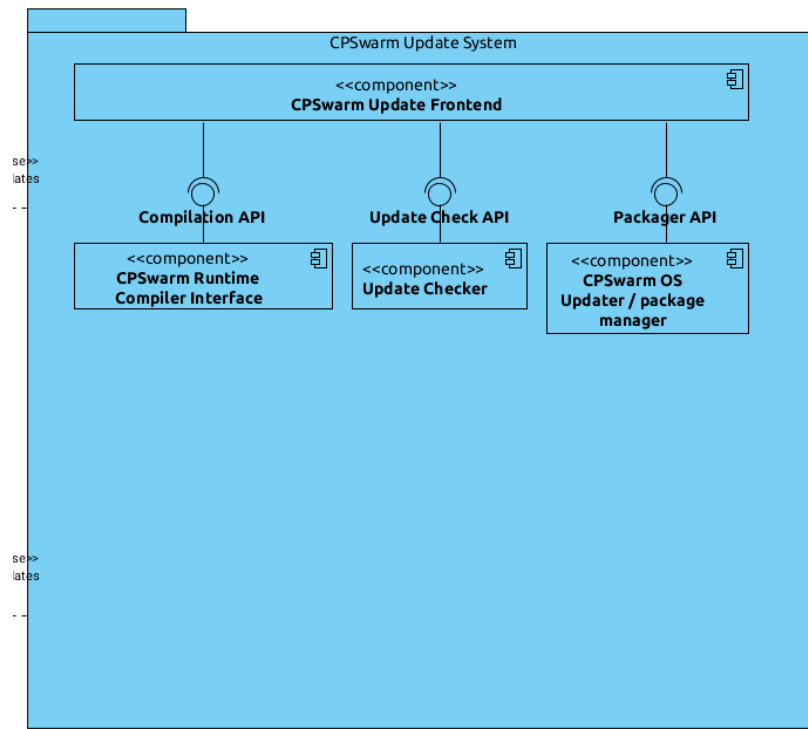


Figure 21 - The CPSwarm Update System.

Among the functions provided by this component, it is worth citing: (a) the ability to compile any source-code that was not compiled by the Deployment Toolchain (e.g., because of different hardware architectures), (b) the ability to update the CPS software components by using the native platform tools, e.g., package managers in Linux-based platforms, (c) the ability to autonomously check for the availability of updates and (d) the capability to securely retrieve and check such updates before actually applying required modifications. Moreover, the CPSwarm Update System supports a transaction-based behavior where any failure during the update process is automatically reverted to the last working set-up. This, for example, enables to retain fully-functional CPS even when update failures occur. Self-updating of the CPSwarm Update System is also supported, thus enabling the CPS to be completely upgraded from remote.

While direct update initiated by the Deployment Toolchain is supported, it is nevertheless not advisable due to scalability concerns. Whenever an entire population of CPS needs to be programmed, it is extremely inconvenient to perform the process one-by-one. Instead, in CPSwarm, the suggested methodology requires the Deployment Toolchain to “publish” a new update on the “update” channel of the population. Such an update will be autonomously fetched by the swarm components thus enabling parallel deployment over all platforms with no additional load on the Deployment Tool, which can therefore be exploited for other tasks (see Figure 22).

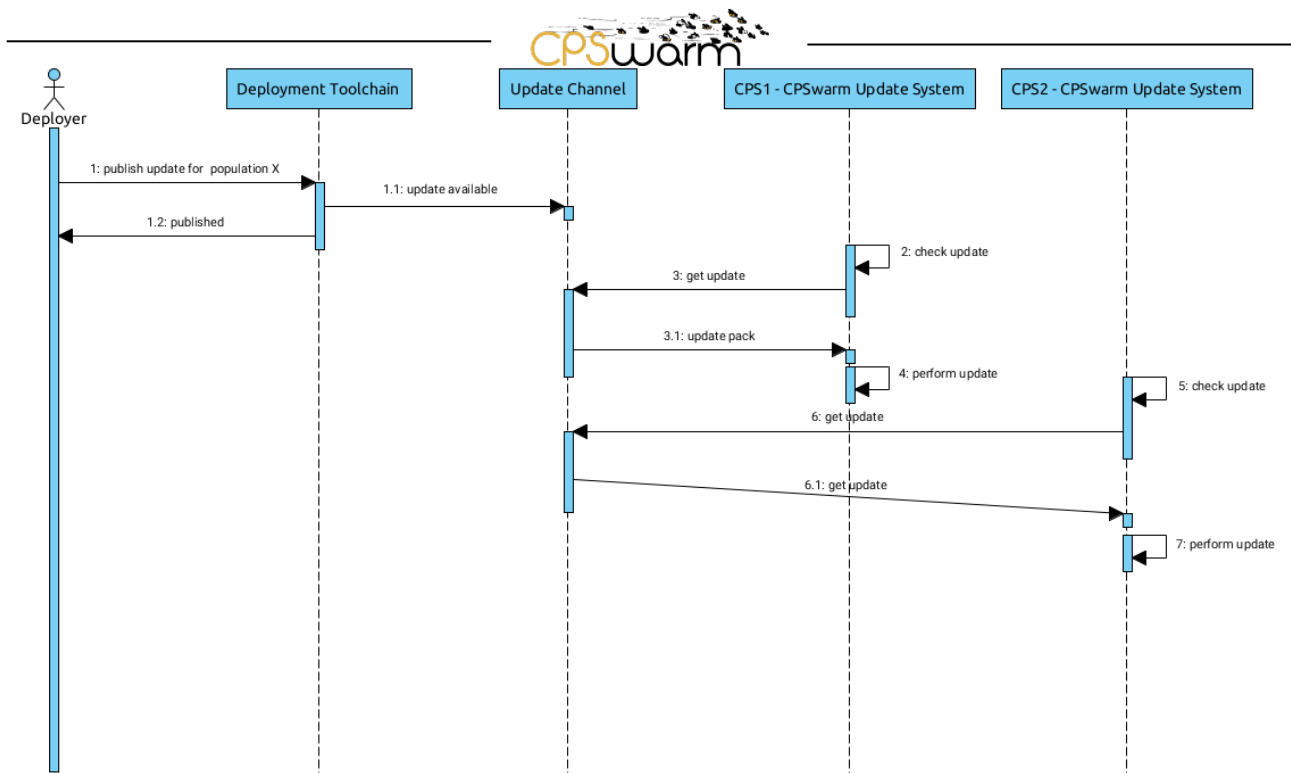


Figure 22 - The CPSwarm update process.

Clearly all single-CPS updates must be finished before the swarm can start operating, therefore suitable communication mechanisms should be provided to ensure that all members of a swarm are running the right version of software.

4.4.6.3 CPSwarm Telemetry

The CPSwarm Telemetry components, part of the CPSwarm RTE, enable delivery of sensor and telemetry data to remote platforms, be they dedicated mission management platforms or generic IoT solutions.

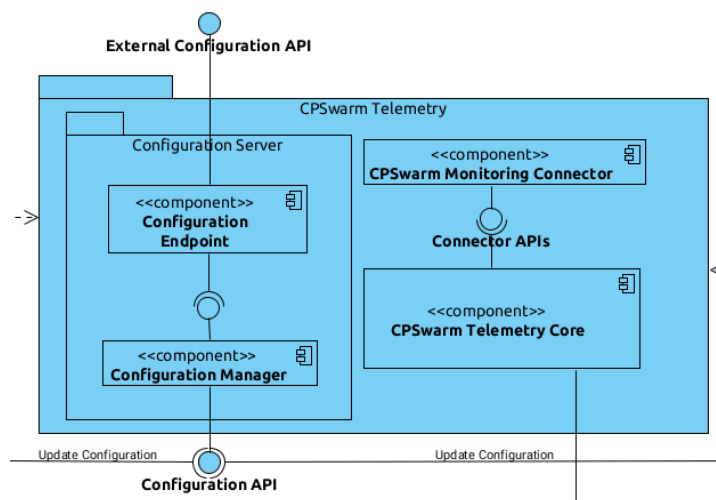


Figure 23 - The CPSwarm Telemetry components.

The CPSwarm Telemetry is composed by 4 main components:

- A Telemetry Core able to interface the running algorithms and to hook into the CPS messaging systems (when existing) to harvest all telemetry and/or sensory data treated at the CPS level;

- A CPSwarm Monitoring connector able to deliver data to, e.g., the CPSwarm Monitoring and Configuration tool.
- A Configuration Endpoint exploited by the CPSwarm Monitoring and Configuration tool to deliver CPS-specific configuration data
- A Configuration Manager, handling the delivery of received configuration data to the destination components within the CPSwarm Runtime Environment

Data harvested by the Telemetry Core can be transferred to different backend platforms, at the same time, and filtering policies may be applied to select the subset of information to deliver to each of the selected platforms. By default, a CPSwarm RTE natively supports the CPSwarm monitoring framework, however other connectors may be installed for enabling connectivity towards other platform, e.g., generic IoT platforms such as Kaa²⁸.

²⁸ <https://www.kaaproject.org>

4.5 Information View

The information view describes the way that the system manipulates, manages, and distributes information. Figure 24 shows the data flow between components in the CPSwarm system. Detailed documentation of each flow's data content, functionality and structure is given in section 4.5.1. A high-level overview of how these flows interact with each other is presented in section 4.5.1

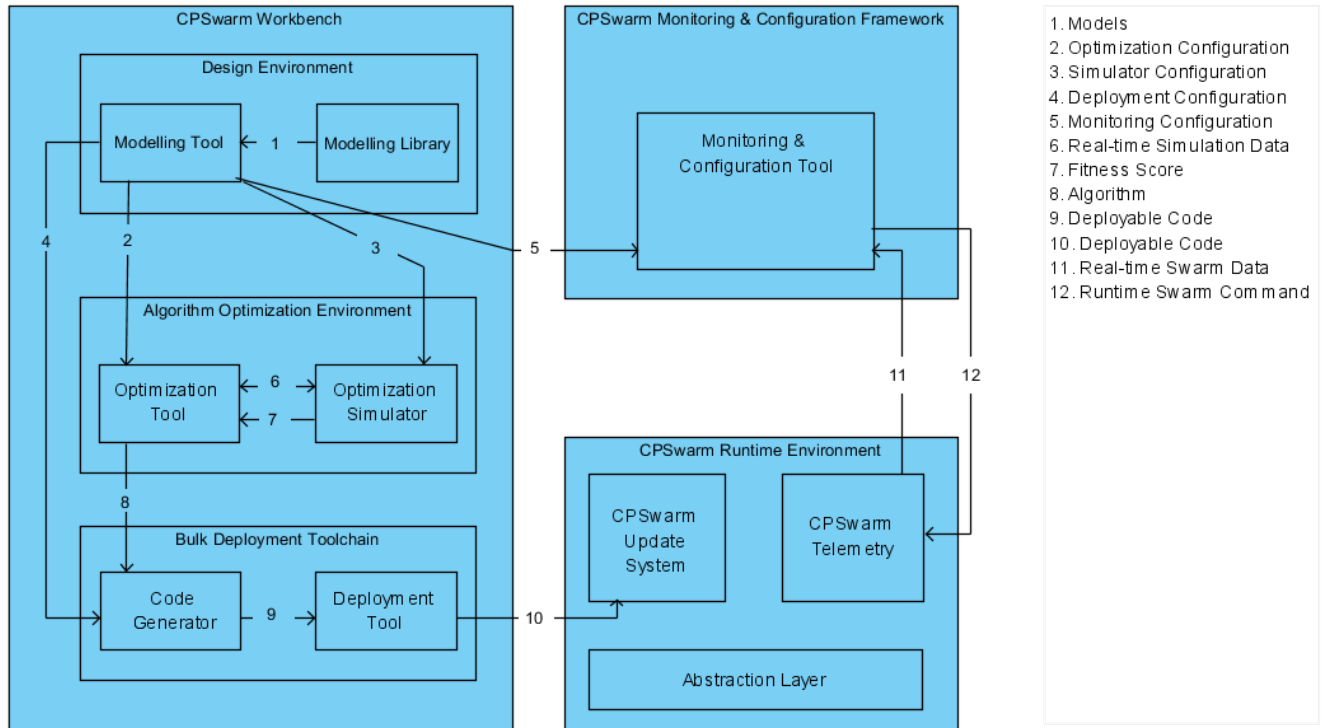


Figure 24 - Data flow of CPSwarm system

4.5.1 Data flow in CPSwarm System

In this section, the data flows within the CPSwarm system will be described to give readers a high-level overview of the workflow of the system. Figure 24 shows the data flows between components in the CPSwarm system, which is derived from the data flow identified in D2.3 (Figure 6). The whole workflow could be divided into different phases:

Modelling phase

During this phase, the swarm modeler/designer interacts with the system through the Modelling Tool to model a swarm. The modeler/designer may create new models, or choose to import reusable models of CPS from the Modelling Library. The imported data is represented by Flow 1 (Models) in Figure 24.

In a swarm model, the following configurations are specified by the modeler/designer:

- The Optimization Configuration, which includes the goal of the swarm, the fitness function as well as necessary parameters for the Optimization Tool. These configurations will be passed to the Optimization Tool to prepare for the upcoming algorithm optimization phase. This is represented by the Data flow 2 in Figure 24.
- The Simulation Configuration, which provides necessary configuration to setup the Optimization Simulator to be used by the Optimization Tool (Flow 3, Simulation Configuration).
- The Deployment Configuration (Flow 4, Deployment Configuration). This configuration specifies how the optimized algorithm should be processed by the Code Generator.

- The monitoring configuration, which provides necessary configuration setup for the monitoring and configuration tool (Flow 5, Monitoring Configuration).

Algorithm Optimization Phase

After the modelling phase, the algorithm optimization phase will be carried out, in which algorithms will be optimized using e.g. evolutionary method across multiple iterations. In each iteration, algorithm agents residing in Optimization Tool will be tested in the Optimization Simulator. During the simulation, the algorithm agents act as the brain and gives command to the simulated devices in the Optimization Simulator. In return, it gets simulated sensor signals from the simulator. The agents then react to these feedback and gives new command accordingly. Flow 6 (Real-time Simulation Data) represents such real-time simulated data exchange between the two components.

After one simulation is finished, the result is evaluated and a score representing how good the agents performed is calculated according to the fitness function. This fitness score is then returned to the Optimization Tool to help to rate the agents (Flow 7, Fitness Score). The returned fitness score marks the end of one iteration. Multiple iterations will be carried out during the algorithm optimization phase to find the optimized algorithm.

Deployment Phase

Following the algorithm optimization phase is the deployment phase, in which the generated algorithm is deployed on target devices. The generated algorithm is first passed to the Code Generator inside the Bulk Deployment Toolchain (Flow 8, Algorithm). The task of the Code Generator is to generate platform-specific code according to deployment configuration from the algorithm. The generated code will then be passed to the Deployment Tool for later deployment (Flow 9, Deployable Code).

The task of Deployment Tool is to deploy the generated code to target devices by publishing it to an update channel (Flow 10, Deployable Code), just as described in section 4.4.6.2.

Operation Phase

After successful deployment, target devices can be started to run in operation mode. On one hand, target devices will stream real-time data to the Monitoring and Configuration Tool, which reflects the current status of the devices. Flow 11 (Real-time Swarm Data) represents this stream of data. On the other hand, the Monitoring and Configuration Tool can also send commands to the swarm, reconfiguring its behavior. Flow 12 (Runtime Swarm Command) represents this flow of commands.

4.5.2 Data model in CPSwarm System

After the high-level introduction of data flow in the CPSwarm system, readers should now have an overall idea of the information exchange between components. Based on this assumption, the detailed data model of each data flow will be presented in this section to provide readers a deeper understanding of the information flow within the system.

In this section, the detailed data model of each data flow will be presented.

4.5.2.1 Flow 1 (Models):

Swarm models depict several aspects of a swarm including its behavior, its environment, its structure in terms of involved drones, the structure and the behavior of each drone, the communication between the drones, etc. Modelling all these aspects will be made on top of existing standards mentioned in section 3.3. The most relevant standards will be used and enriched if necessary.

Until now two kinds of models are expected to be stored in the Modeling Library and retrieved by the Modeling Tool (Flow 1):

- **Structural models:** To show the composition and classification of swarm structural element (drone, sensor, etc.). The interface of each structural element and static connections between them will also be modeled by using the structural modelling.
- **Behavioral modelling:** To highlight the inputs, outputs, aka data flow and conditions aka control flow of the CPSwarm behavior.

Details of these models will be found in D5.2 Deliverable.

4.5.2.2 Flow 2 (Optimization Configuration)

This flow transports the models of the Modelling Tool describing environment, agents, and the goal of the optimization in terms of a fitness function. They are used to set up the optimization tool. This flow contains two parts:

- Problem (including descriptions of goal, environment, and agent)
- Parameters (for configuring the optimization tool)

Problem: The problem contains everything that is necessary to test and evaluate a possible solution. This functionality is used by the Optimization Tool to perform an automated search for a viable solution.

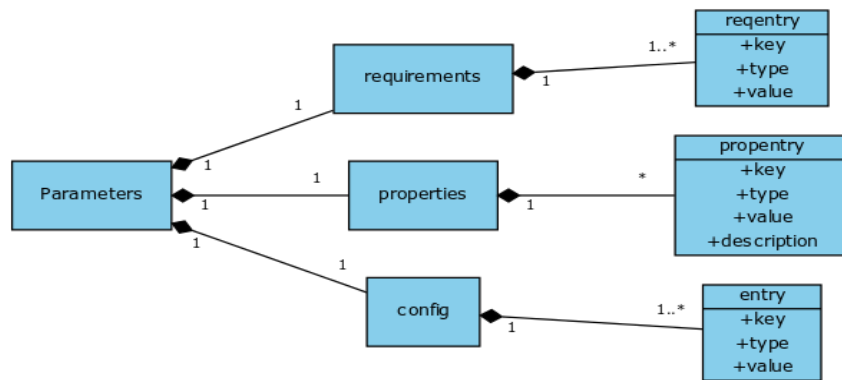


Figure 25 - Parameters for the Algorithm Optimization Framework

Parameters: This is a set of parameters that describe the setup of the optimization tool. They are passed from the modeling tool to the optimization tool. These parameters include *configuration*, *properties*, and *requirements*, see Figure 25.

The *configuration parameters* describe the problem that is modeled with a set of strings.

The *properties* are a set of parameters that are not fixed during the modeling phase but left open for the algorithm optimization phase. However, these parameters have a default value set by the modeling tool. These parameters are used to evaluate the algorithm with different setups. This is useful to achieve a robust algorithm that works in different environments. There are no mandatory parameters. Exemplary parameters in the properties section are:

- description of agents, e.g. cardinality, positioning, size
- description of environment, e.g. size, obstacles, POI

The *requirements* define important parameters for the evolutionary optimization process. These parameters are:

- *inputnumber*: the number of inputs for the controller, i.e. the number of sensed inputs, each input has the type of a single scalar value, but depending on the problem, the input signal could also use only a part of the possible values

- *outputnumber*: the number of outputs of the controller, i.e. the number of actuator controls, each output corresponds to a single scalar value, which might be interpreted further
- *minimumCandidates*: the minimum number of agents necessary to address the problem
- *maximumCandidates*: the maximum number of agents that can be used in the problem

4.5.2.3 Flow 3 (Simulator Configuration)

The simulator configuration is used to set up the optimization simulator. It contains three parts:

- Environment
- Agent
- Fitness function

Environment: The environment description contains all details of the environment that are necessary for the simulation to run. This includes the following parts:

- Floor plan of the underlying map: This floor plan is loaded by the simulator to display the environment and to provide the environment to the agent's sensors. This could be for example a bitmap graphics for a 2D simulation.
- Environment description for the simulator: This is a configuration file for the simulator which includes necessary parameters. These parameters are for example the path to the floor plan file, the resolution of the floor plan, or the simulation speed.
- Environment description for the agent: The agent might require some a-priori information about the environment. This could include the floor plan bitmap file or the location of certain points of interest.

Agent: Each agent is described by its physical properties such as size, starting position, sensors, and actuators as well as by its behavior. Such properties are encoded into a simulator independent format, e.g., YAML, and delivered to the CPSwarm Optimization Simulator tool. As an example, the following lines show a possible definition of a range sensor derived from a generic sensor description:

```
define lidar ranger
(
  sensor
  (
    range [0.0 30.0]
    fov 270.25
    samples 1081
  )
  color "black"
  size [0.05 0.05 0.1]
)
```

The agent behavior is described with functional code. This code is the core of the optimization simulator as it defines how the sensors are read, how the actuators are used, and how the robot decides on its actions. The behavior changes every time the simulator is executed with a different controller candidate by the optimization tool.

Fitness function: The fitness function is used by the optimization tool to compute the fitness of a controller candidate. In order to calculate the fitness, relevant simulation parameters need to be analyzed, either during the simulation or by processing a simulation log after the simulation has finished.

4.5.2.4 Flow 4 (Deployment Configuration)

Data flowing from the CPSwarm Design Tool to the Deployment Toolchain (flow 4 in Figure 24) belongs to 3 different categories:

- code-generation directives
- bundling directives
- deployment configurations

Code generation directives are couples of algorithm descriptions and target runtime identifications. They provide the complete specification of a control algorithm using either a well-defined formalism or a DSL, and the information needed to identify the target runtime for which the algorithm shall be “compiled”.

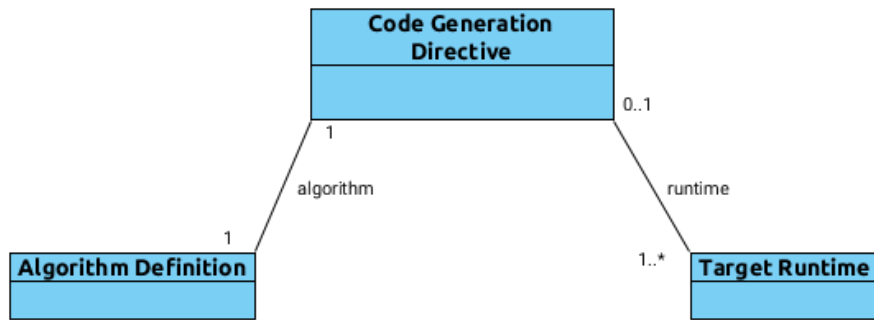


Figure 26 - Preliminary data-model of Code Generation directives.

Figure 26 reports a preliminary data model of code-generation directives. Each directive refers to a single algorithm definition (e.g., expressed through a DSL) and identifies the set of runtimes for which runnable code should be generated. This multiplicity allows code generation for multiple runtime platforms in a “single” pass.

Bundling directives, identify the set of dependencies and functional blocks that need to be bundled together with the generated algorithm to provide a fully working CPS code (see Figure 27). In a ROS-based platform, for example, this information includes the set of ROS nodes that need to be installed for deploying the generated code.

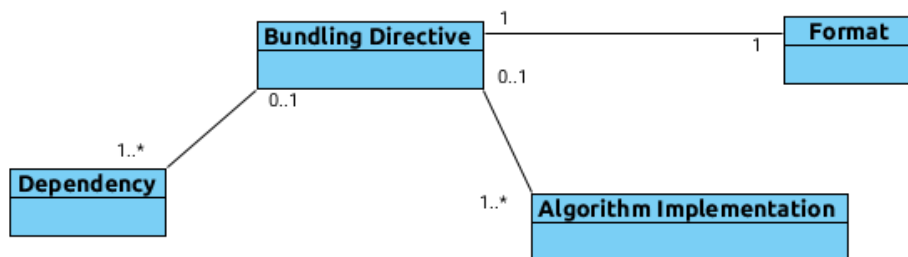


Figure 27 - Preliminary data-model of Bundling Directives.

The currently available specification²⁹ defines a bundling directive as composed of a list of dependencies (similarly to, for example, Linux package dependencies), and algorithm implementation and a bundling format, e.g., Debian package, ROS module, etc.

Deployment configuration specifies the target platforms and/or the update channel to which the bundles to be deployed shall be published. Such information is in principle independent from the generated code, which does not need to be generated at deployment time.

²⁹ Under refinement.

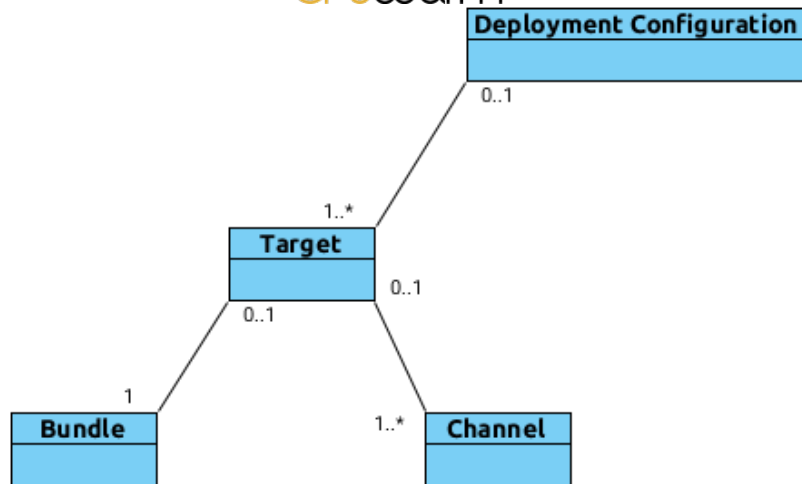


Figure 28 - Preliminary data-model of Deployment configuration.

It includes the set of target platforms (single CPS instances) to which deploy the bundles. For each target two main elements shall be specified:

- a bundle to deploy;
- a deployment channel to leverage on.

This flexibility in the specification allows, for example, the deployment of bundles to different, heterogeneous CPS, in one step.

4.5.2.5 Flow 5 (Monitoring Configuration)

The data model for Monitoring and Configuration will include the structure, syntax, and semantics of data to be monitored and configured, as well as primitives to view and manipulate the data (e.g., in a form of defined operations). With this data model, different configurations, e.g., for system start-up, shut-off, running, etc., are specified. The reuse of existing data models and data exchange notations such as YANG³⁰ or JSON will be promoted. Some of the necessary commands that the Configuration and Modeling Tool could send to the single CPS or CPSwarm are: *get-config* to get the currently deployed configuration, *edit-config* to modify some of the parameters of current configuration, *get* to read the concrete configuration values and status, and *hello* to get the information about the *capabilities* of the CPS. Capabilities define what monitoring and configuration features or mechanisms can be used.

Transactions for modifying or exchanging configurations must have ACID properties (also for the complete Swarm): Atomicity (Transactions are indivisible, all-or-nothing); Consistency (Transactions are all-at-once and there is no internal order inside a transaction); Independence (Parallel transactions do not interfere with each other); Durability (Committed data remains in the system even in the case of a fail-over, power failure, restart, etc.).

4.5.2.6 Flow 6 (Simulation Real-time Exchange)

The simulation real-time exchange enables the Optimization Tool to use an external optimization simulator for evaluating each controller candidate. Figure 29 depicts a data flow between Optimization Tool and Optimization Simulator. The exchange is performed by a network interface that also allows to control the optimization simulator. Each simulator requires a specific interface that considers the characteristics of the corresponding simulator. The following steps describe a typical interaction between Optimization Tool and Optimization Simulator:

1. Setting up the simulator: The simulator needs to be configured as the parameters have been chosen by the optimization tool. These parameters include e.g. the number and placement of agents or

³⁰ YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)
<https://tools.ietf.org/html/rfc6020>

environment details. Also the network address of the optimization tool needs to be passed to the simulator so it can return required information.

2. Starting the simulation: Once the setup has completed and the optimization tool is ready to evaluate a candidate controller, the optimization tool issues a command to the simulator to start the simulation.
3. Reading the sensor data: The simulator sends the sensor readings to the optimization tool.
4. Sending actuator commands: The optimization tool processes the sensor readings using its evolved controller candidate. The resulting actuator commands are returned to the optimization simulator which then executes these commands.

Steps 2 and 3 of this process are repeated until the optimization tool finishes the optimization process. Further commands that need to be passed in this flow are:

- Stopping/pausing the simulation: To ensure that the simulator is completely controllable by the optimization tool the stop and pause commands are required to interrupt the simulation process.
- Replay: For introspection purposes the optimization tool can be used to analyze a specific solution that has been evolved. Therefore, an additional command is required that enables a visual mode where the simulator displays a GUI showing the simulation progress.

4.5.2.7 Flow 7 (Fitness Score)

The fitness score measures the performance of a controller in a single simulation run. The data for calculating the fitness score could be based on a continuous monitoring function in the simulator or by processing log files after the simulation run has finished. The flow of fitness score from optimization simulator is depicted in Figure 29.

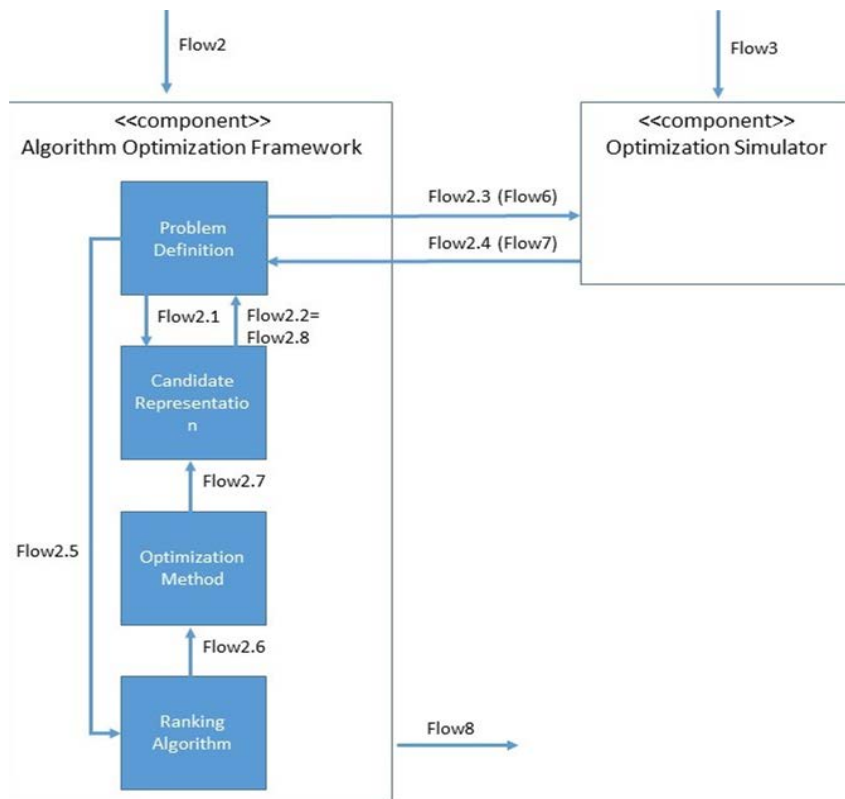


Figure 29 - Data flow within the optimization tool

4.5.2.8 Flow 8 (Algorithm)

When the optimization tool finishes, the result is an algorithm that has a general format and can be passed to the deployment toolchain for generating deployable code. The algorithm is expressed in source code to be further processed by the deployment toolchain.

4.5.2.9 Flow 9 (Deployable Code)

Code generated by the CPSwarm Code generator is typically compiled and bundled with required libraries and made available to the deployment tool via a dedicated interface. Two main kinds of deployable code data are currently envisioned depending on the target deployment platforms. When the target platform is compatible with compilation accomplished at the code generator side (i.e., it has the same or a compatible hardware architecture, or cross-compilers are available), the outcome of the code generation process is a compiled code unit, e.g., an OS package for Linux-based CPS, which is transferred to the deployment tool to be “delivered” over the right update channel. Instead, when compilation cannot occur beforehand, the data flowing across the link between the code-generator and the deployment tool is composed of the generated source code plus all the required dependencies, either packaged as a library bundle or as a requirements file. Such data is complemented by a dedicated compilation script that can be exploited by the deployment tool to trigger compilation once the generated code is uploaded to the target CPS.

4.5.2.10 Flow 10 (Deployable Code)

Flow 10 (Deployable Code) represent the code that is transferred to target devices by the Deployment Tool. The Deployment Tool does not change the generated code, but rather transfers the code in a form suitable for the specific deployment strategy (i.e. OTA / Direct Deployment).

4.5.2.11 Flow 11 (Swarm Real-time Data)

Once the code designed, generated and deployed through the CPSwarm Workbench has been deployed on the individuals of a swarm of CPS, telemetry data, as well as data generated by CPS payloads (e.g., thermal imaging, etc.) can be monitored through the CPSwarm Monitoring and Configuration Tool. CPSwarm-enabled CPS have the possibility to either leverage the CPSwarm native monitoring infrastructure or they can connect to generic IoT platforms, selected at deployment time.

Data exchanged between the swarm individuals and the Monitoring tool, natively exploits a Publish/Subscribe interaction pattern to account the fact that:

1. Multiple listeners might need to receive telemetry or sensory data, on a dynamic subscription bases. Publish/Subscribe natively support this requirement by decoupling event sources (i.e., the CPS) from event consumers.
2. Data may be transferred opportunistically, depending on the actual connectivity and network conditions. This prevents the adoption of any client-server-like interaction paradigm where the CPS acts as server. Cases in which the CPS system plays the client role are possible, however they might not be suited for high-frequency / high-cardinality data streams.

Publication of selected subsets of telemetry/sensory information to on-line IoT platforms through REST API calls is supported, at least for low/medium frequency data streams. While depending on selected IoT platforms, required data-formats might change, CPSwarm CPS natively provide sensory data encoded according to the OGC Sensor Things API formalism.

In such a standard, an *Observation* is modelled as an act that produces a result whose value is an estimation of a property of the observation target or *FeatureOfInterest*. An *Observation* instance is classified by its event time (e.g., *resultTime* and *phenomenonTime*), *FeatureOfInterest*, *ObservedProperty*, and the procedure used (often corresponding to a Sensor). *Things* are also modelled in the SensorThings API, together with the

historical set of their geographical positions. More specifically, in the Sensing profile, a *Thing* has *Locations* and *HistoricalLocations*. It can also have multiple *Datastreams* associated. A *Datastream* is a collection of *Observations* grouped by the same *ObservedProperty* and *Sensor*. An *Observation* is an event performed by a *Sensor* that produces a result whose value is an estimate of an *ObservedProperty* of the *FeatureOfInterest*. Following subsections better detail the single data model entries.

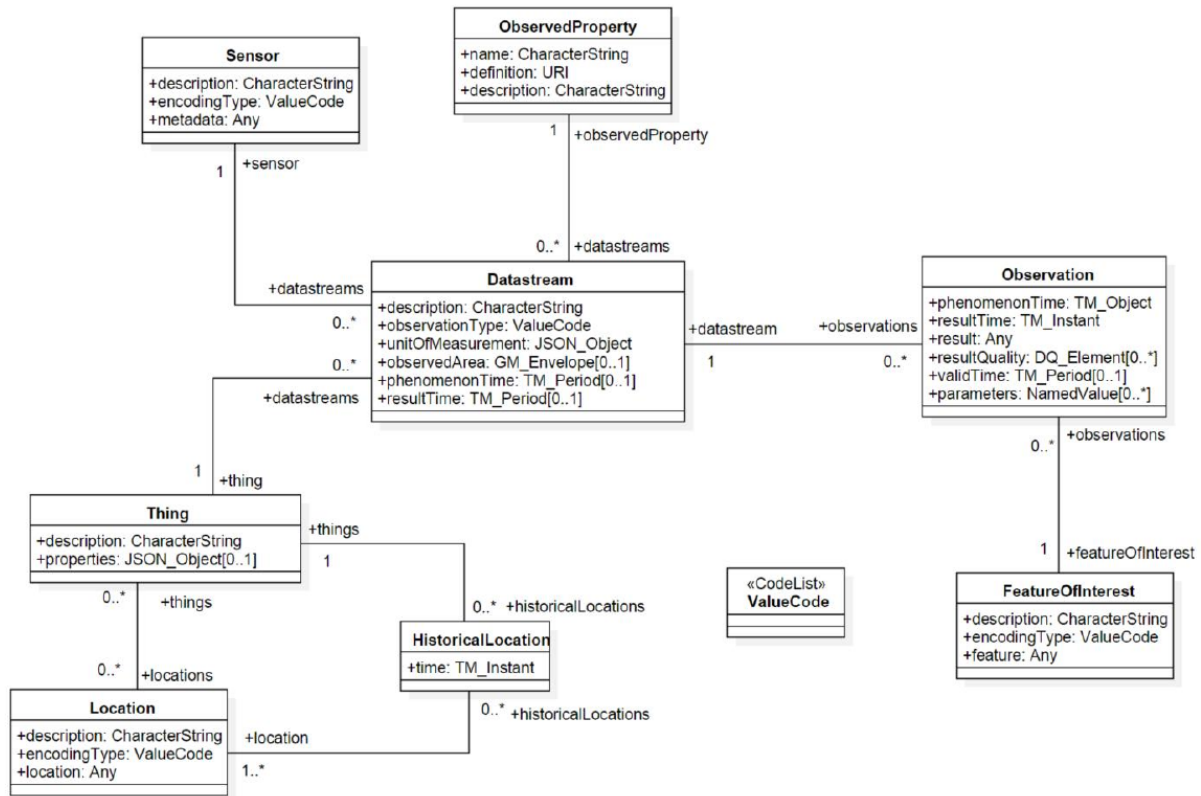


Figure 30 - The OGC Sensor Things API data model.

Thing

The OGC SensorThings API follows the ITU-T definition, i.e., regarding the Internet of Things, a *Thing* is an object of the physical world (physical things) or the information world (virtual things) that is capable of being identified and integrated into communication networks [30].

Location

The *Location* entity locates the *Thing* or the Things it is associated with. A *Thing's* *Location* entity is defined as the last known location of the *Thing*.

HistoricalLocation

A *Thing's* *HistoricalLocation* entity set provides the current (i.e. last known) and previous locations of the Thing with their time.

Datastream

A *Datastream* groups a collection of *Observations* and the *Observations* in a *Datastream* measure the same *ObservedProperty* and are produced by the same *Sensor*.

Sensor

A *Sensor* is an instrument that observes a property or phenomenon with the goal of producing an estimate of the value of the property.

ObservedProperty

An *ObservedProperty* specifies the phenomenon of an *Observation*.

Observation

An *Observation* is an act of measuring or otherwise determining the value of a property (OGC and ISO 19156:2011).

FeatureOfInterest

An *Observation* results in a value being assigned to a phenomenon. The phenomenon is a property of a feature, the latter being the *FeatureOfInterest* of the *Observation* (OGC and ISO 19156:2001). In the context of the Internet of Things, many *Observations'* *FeatureOfInterest* can be the Location of the Thing. For example, the *FeatureOfInterest* of a wifi-connected thermostat can be the Location of the thermostat (i.e. the living room where the thermostat is located in). In the case of remote sensing, the *FeatureOfInterest* can be the geographical area or volume that is being sensed.

4.5.2.12 Flow 12 (Swarm Runtime Command)

CPSwarm-programmed CPS can receive commands, e.g., to switch between pre-programmed behaviors, and/or configuration parameters through the channel established by the monitoring tool, exploiting the telemetry core of the runtime environment. Through this back-channel, two main kinds of data are delivered: configuration data, typically encoded as a set of key-value pairs, e.g., encoded in YAML or JSON, and runtime commands, typically triggered through dedicated calls, be they REST-based or exploiting a message-based interaction paradigm that can leverage the telemetry publish/subscribe channel as transmission mean. Currently, the set of allowed commands as well as the set of envisioned configuration parameters has still to be refined, therefore a more precise specification of data exchanged at this level is not yet available. Nevertheless, at the architectural level, the need for such a backward communication channel is duly accounted: the next releases of the architecture specification will include a more detailed description of data models, and APIs involved in the information flow between the monitoring tool and the CPSwarm RTE.

4.6 Deployment View

The CPSwarm workbench consists of components with a variety of dependencies. Figure 31 shows the network diagram of these components. This section describes the environment required for each component.

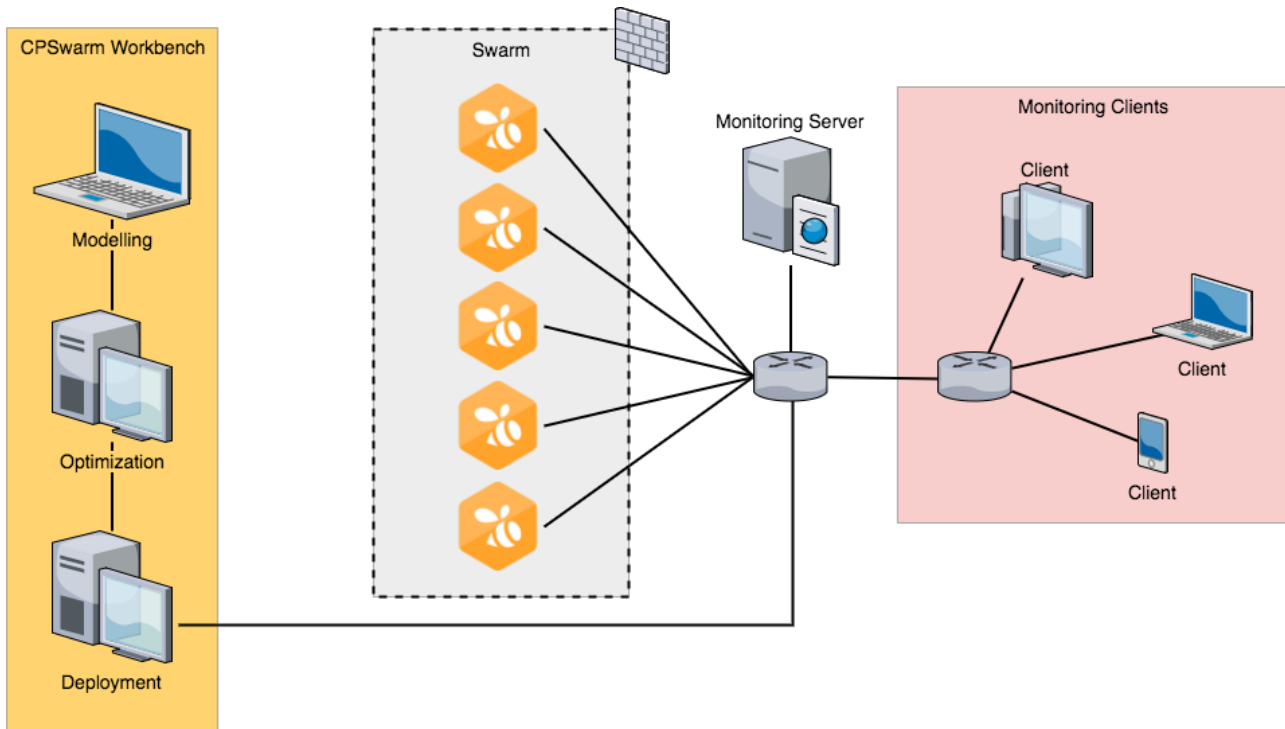


Figure 31 - Network diagram of CPSwarm component during the deployment.

The modelling tool is one main interaction point between the users and the CPSwarm system. It should be a desktop application with graphical user interface to provide easy access for users. It is designed to be a local application that runs on typical personal computers using common operating systems, such as Microsoft Windows, Linux and/or macOS.

For the optimization, external simulators may be used to evaluate the performance of algorithms. Third-party open-source simulation environments such as Gazebo³¹ or Stage³² could be used for such purpose. To address the scalability issues mentioned in Section 6, the CPSwarm system provides the possibility to run multiple optimization simulators across a cluster of machines. For easy deployment and management of a cluster of simulators, Docker could be used. Docker is a container technology that allows a developer to package up an application with all the dependencies it needs into containers to isolate it from the host environment³³. In our case, each simulator will be packaged together with its software dependencies within a Docker container. The containerized simulator then can be easily deployed and run on local machine, remote server, or across a cluster of machines with the help of Docker Swarm application regardless of software dependencies, as long as the machine supports Docker.

The monitoring and configuration framework is the center for swarm real-time data and runtime configurations exchange. Thus, it must be exposed to both swarm members and monitoring clients. CPSwarm allows the monitoring server to be deployed on either local machine or remote server, as long as

³¹ <http://gazebo-sim.org/>

³² <http://playerstage.sourceforge.net/doc/stage-svn/>

³³ <https://www.docker.com/>

the machine is accessible by swarm members and monitoring clients. For easy deployment and management, Docker container could be utilized to isolate the monitoring server from the running environment.

The CPS Runtime Environment (CPSwarm RTE) is an environment which is built upon the operation environment of swarm members to support execution of generated swarm algorithm, to provide on-line update and to support remote telemetry and data monitoring. The CPSwarm project aims at creating design specifications and guidelines for such RTE so that it is extensible to fit on common robotic platform, such as Robot Operating System (ROS), STEM-robots, etc. As a proof of concept, the focus will be on the development of a RTE based on ROS. ROS is chosen because devices from DIGISKY and ROBOTNIK rely on ROS to operate. Besides, ROS is also one of the most popular and widely used platforms for developing robotic applications world-wide. A successful implementation of such RTE will very likely bring great impact to the community of swarm robot developers.

5 Security Perspective

The current state-of-the-art literature on the security of cyber-physical systems addresses a variety of issues, such as the security of CPSs as industrial control systems [31], threat analysis based on physical-, network- and application layer vulnerabilities in CPSs [32] or describing security challenges regarding swarm robotics [33], etc. The CPSwarm project aims to contribute to this field of science by providing a general threat and risk analysis of CPS security, followed by use-case specific threat and risk analyses. In the following sub-sections an initial threat analysis is provided regarding the confidentiality, integrity and availability of the CPSwarm system, followed by security best practices and countermeasure recommendations to address these threats.

5.1 Security threat analysis

As in common threat analysis practice, this sub-section describes threats in the context of cyber-physical systems through the main security objectives of **Confidentiality, Integrity and Availability (CIA)** in short). First, let's define what these terms mean in the special case of a swarm and its agents:

- **Confidentiality** means preventing the disclosure of sensitive data collected or stored by the CPS to unauthorized parties. Only authorized agents and operators should be able to see mission targets, objectives and progress reports, as well as maps, camera recordings and other sensor data.
- **Integrity** refers to data or system information that cannot be modified without authorization. In case of a swarm of robots, integrity means that such data should originate from authorized entities and unauthorized third parties should not be able to tamper with it.
- **Availability** means that the system must be able to operate when needed. On the swarm level, it focuses on the ability to complete the mission, while on the level of individual agents, it mostly relates to the ability of the agent to act as a member of the swarm.

In Figure 32, a part of the threats identified are shown grouped into 3 categories based on whether they compromise the Confidentiality, Integrity or Availability of the swarm.

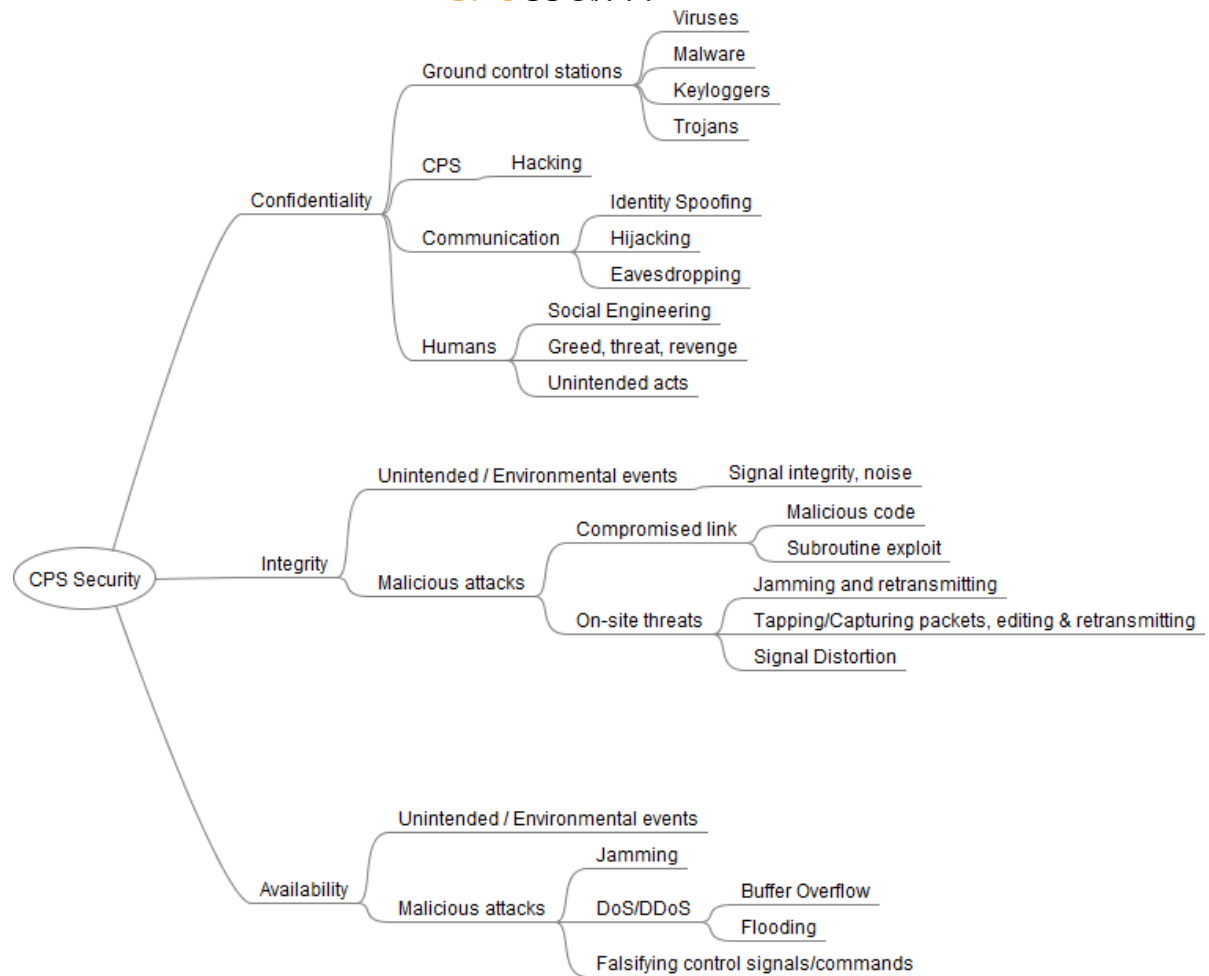


Figure 32 - Threats categorized by CIA

Concerning Confidentiality, one can possibly aim attacks on different components of a CPS. Ground control stations are the computers/devices from which the operator gives commands to the CPS and these devices can be compromised by viruses, malware, keyloggers or trojans. It is also possible to obtain confidential information by hacking the CPS directly or compromising the communication link with the operator by identity spoofing, hijacking or eavesdropping. In some cases, it is much easier to hack the human in the loop (e.g. the Swarm Operator) by social engineering, bribes, or by using threats and violence; or a compromise of confidentiality can simply happen by accident due to human error.

Integrity can be compromised by mistakes made by personnel or environmental effects resulting in noise in signals and sensor recordings. Malicious attacks aimed against integrity can compromise the link using malicious code or subroutine exploits, or the attacker can go on-site to jam signals, or to capture packets, then edit and retransmit them.

Absence of availability can be a result of environmental effects - for instance extreme weather conditions - or of malicious attacks, such as signal/sensor jamming, denial of service or distributed denial of service attacks, either by exploiting vulnerabilities like buffer overflows, or by flooding, or by falsifying control signals or control commands.

5.2 Countermeasures

Since agents in a swarm consist of many components with different possible vulnerabilities, it is important to emphasize that a system is as secure as its weakest component. For purely economic reasons, an attacker will

always target the vulnerable component that can be exploited with the least effort [34]. Therefore, to ensure the security of agents and swarms, a developer should make sure that there are no weak points.

Some of the threats mentioned above are unrelated to the swarm itself – such as social engineering, malware and viruses. Risks associated with these threats could be reduced by training personnel and ensuring that all operating system and third party software components are up to date on the agents' and operators' systems, and that the software security and secure software development methodologies described by Figure 33 have been adopted. It is also important to ensure the integrity of the software running on the agents by digitally signing all components, ensuring that software updates can only come from trusted parties and have not been tampered with.

Feeding false information to software components, either through the sensors or through communication channels between software and hardware components could endanger the learning ability and decision making pertinence of the agents. It is a complex threat which could arise from man in the middle attacks, malware, node impersonation, or by subverting sensors as depicted in Figure 32. Strong encryption and authentication mechanisms would protect against man in the middle attacks and some of the consequences of having false input fed to the systems. On the sensor side, sanity checks should be applied. The effect of such an attack can be mitigated on the swarm level by monitoring the consistency of the data shared between agents and by taking advantage of the different redundancies in case noise is detected in the system.

Resource exhaustion can be a consequence of diverting the route of the agent or physically harming it, to increase the power, fuel or other consumable usage of the agent until the resource eventually runs out. Thinking about a drone with at most 30 minutes of battery time, the severity of this issue can be clearly seen. Another possible denial of service attack is overloading the processing, memory or storage capacity of individual members. To handle hardware failures of any kind, fault detection is a key feature and the first step to corrective actions.

Fault tolerance is also a crucial feature to handle attacks against sensors, thus preventing miscommunication, bad decisions and collisions in the swarm. Jamming could pose a serious security threat in swarm robotics since it could disrupt multiple agents at the same time, even if they possess very sophisticated high-level security mechanisms. A possible source of inspiration to look at is wireless sensor networks – several possible countermeasures are proposed to defend against jamming attacks targeting such systems, like transmission power regulation, DSSS modulation, frequency hopping, ultra-wideband signaling, etc. [35]

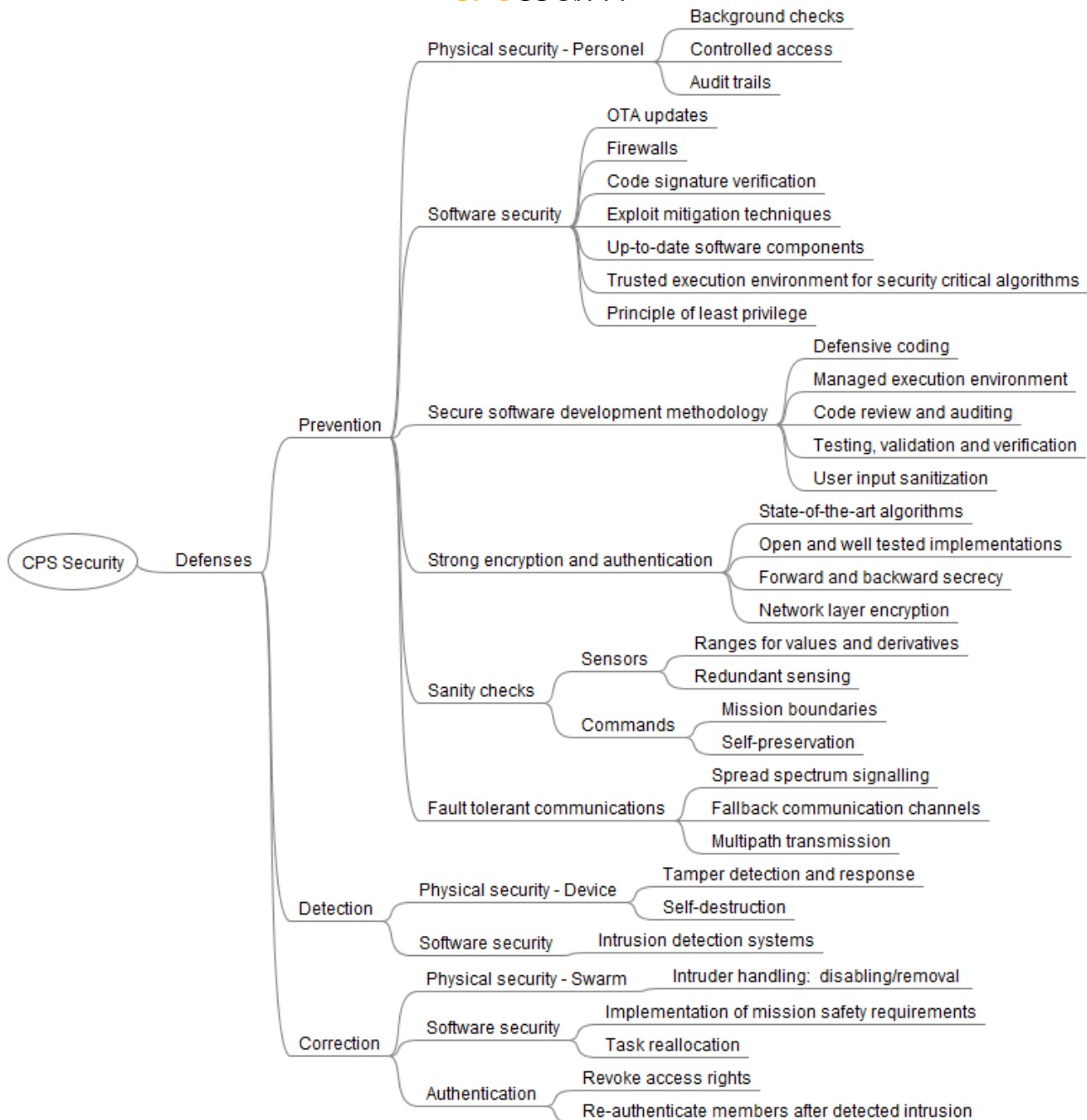


Figure 33 - Countermeasures to the CPSwarm system

To defend against attacks on communication channels, it is required for the swarm to have proper cryptography and authentication mechanisms. The security of individual agents can be ensured by following the best practices concerning system security. One straightforward defence is to encrypt all data streams between any two agents or nodes, making eavesdropping and man-in-the middle attacks effectively impossible. Cryptography depends on key-management, which is a difficult to implement correctly when dealing with the mobile agents of a swarm. In a case study by Hansson [36] on the security of mobile ad hoc networks, the authors mention that currently the only widely deployed solution for key management in this scenario is the manual creation and distribution of keys. However, this method increases traceability and the chance of misusing the keys, and implies that the keys are used for a relatively long time, which gives the attacker more opportunities to crack the keys. Fast, on-site revocation of cryptographic keys would be crucial in swarm robotics.

The main difference between the countermeasures applicable to individual agents and swarms is that in a system of multiple robots, it is possible to implement corrective measures depicted on Figure 33. For instance, after an intruder has been detected, the swarm could collectively decide whether to avoid, disable or remove it from the swarm, taking the pre-defined action depending on the mission safety requirements. If an agent fails, either due to an attack or an accident, the swarm could reallocate the workload between the remaining agents and continue the mission. To prevent an intruder from spying on the mission and from obtaining sensitive information, the swarm could decide to revoke the rights assigned to compromised members and re-authenticate between non-compromised members.

Even if all preventive controls presented on Figure 33 concerning personnel, software security, secure software development, encryption, authentication, sanity checks and fault tolerant communications would be implemented, the security aspects of the distinctive collective behaviour and autonomy of swarms would still pose significant challenges. Protecting individual agents might prevent most cyber-attacks against the swarm as a whole; however, the possible corrective actions in case of a successful attack still need to be thoroughly investigated. New, different kinds of threats can arise depending on the many scenarios and environments a swarm could encounter. The biggest challenge of security research in the field of swarm robotics is developing intrusion detection and post-intrusion corrective techniques.

5.3 Security aspects in CPSwarm architecture design

Many of the countermeasures described in the previous section can and should be integrated into the CPSwarm system in a way that it provides security by default to all swarms designed with the CPSwarm system. While the overall security objectives of the swarm depend on the use case and the mission, all effort must be undertaken to ensure that the system in general provides a solid foundation on which secure applications can be built on. Listed below are some potential points to be noticed in designing the CPSwarm system:

The Bulk Deployment Toolchain should help users securely push updates to swarm members. Infrastructure to sign code (and on the target system, validate signatures) could make secure software updates work without further user effort. Great care must be taken to ensure that the code generated here is not the source of additional vulnerabilities, as the user is unlikely to audit the results of the automatic code generation tools.

The Monitoring and Configuration Framework could collect data on the integrity of swarm members and on the integrity of the swarm itself for auditing and early threat detection.

The Runtime Environment should provide a secure communication channel for swarm members, and ensure that integrity requirements (like code signature verification) are enforced. The base platform on which the runtime environment operates should also be configured to defend against common attacks – this includes the correct configuration of system features like firewalls and privileges.

The Design Environment and the Algorithm Optimization Environment can also contribute to the overall security of the system: by aiding the design and simulation of different threat vectors and environmental disturbances, the behaviour of the swarm can be tested in and optimized for these edge cases.

6 Scalability Perspective

Scalability and performance are of most importance in many ICT systems, no matter if they are used in monolithic applications or in applications for fully-distributed systems. While the CPSwarm architecture might appear as a relatively vertical solution with few or no scalability issues, there are several points in the architecture which require a “design for scalability” approach. More specifically, the CPSwarm architecture can be divided into two main subsystems, each having particular requirements in terms of performance and scalability.

6.1 CPSwarm Workbench

The CPSwarm Workbench designed in this first version of the architecture specification is mainly a “single” application used to model, simulate and optimize a population of CPS. While modeling can be considered as a task that has typically low scalability requirements (and issues), optimization and simulation can be defined as the two most demanding phases in terms of performance and parallel execution.

The CPSwarm Workbench relies on evolutionary optimization algorithms, which are designed to exploit “natural evolution” mechanisms to explore the space of possible solutions for a given problem, with the intent of finding the global optimum (or a nice approximation of it). This exploration is usually carried by generating a set of candidate solutions and by evaluating such solutions against a performance metric (fitness function). The best performing solutions gain the right to “survive” and are “propagated” to subsequent generations through operators that mimic natural evolution (e.g., crossover and mutation in genetic algorithms). At every iteration of the evolutionary algorithm, a new population is generated and its candidates evaluated. Typically, the process will converge to a “good enough” solution, where the degree of optimization depends on the “fitness function definition” and on the ability of the algorithm to avoid local optima. It is easy to notice that the whole process requires several iterations (the number of generations required by the algorithm to converge) and involves several candidate solutions at each generation.

In the CPSwarm case candidates are the controllers for each individual agent of a swarm of CPSs. Evaluating their fitness requires simulation with a certain fidelity related to the design stage. For example, in early design phases simple 2D simulation might be sufficient while in final stages, full physics simulation can be exploited. This requirement may potentially lead to serious performance and scalability issues that shall therefore be addressed from the very early architecture design iterations.

To better clarify these issues let the reader consider a simple example. A CPSwarm user needs to design a swarm of 5 drones for a “search and rescue” task. After the initial definition of swarm components and subsystems, the swarm developer defines the family of algorithms to be used for solving the problem and starts the CPSwarm Optimization Tool. For each candidate solution, composed of 5 CPS controllers, a simple 2D simulation of a mission is exploited to compute the candidate fitness. The mission has an average duration of 1 minute in simulated time³⁴. If at each generation 50 different candidates are evaluated, the optimization components need to run 50 different simulations of 1 minute each, per generation. Given the simplicity of this sample problem, the optimization converges quite fast and it can reach a satisfying fitness within only 100 iterations. In a pure sequential setting, this leads to a total optimization time of roughly 83 hours, see (1).

$$50 \cdot 60s \cdot 100 = 300000s = 83,3h \quad (1)$$

While for some tasks such a time frame might be acceptable, for search and rescue the above optimization time would probably be too long. The situation might easily get worse, when swarms are complex, when

³⁴ This is a naive assumption as the simulation time depends on the “quality” of the solution and, in principle, might be quite long, and e.g., reach a maximum, pre-set time limit.

simulations are computationally intensive and when the nature of the addressed problem implies slower convergence rates.

Therefore, the main workbench bottleneck can be identified in the evolutionary optimization step and suitable countermeasures shall be designed from the very beginning of the CPSwarm architecture design. In particular, in this first release of the architecture design it is already envisioned the possibility to exploit cloud-based and/or containerized solutions for simulations, thus enabling to simulate an entire population of solutions in one step. This is expected to sensibly reduce the amount of time needed to complete the optimization phase, thus improving the overall performance of the CPSwarm workbench.

More formally, let n be the number of candidates per generation and m the number of generations required to reach a valid solution. Assuming that for each individual evaluation a maximum simulation time (t_m) is allowed, in the sequential case the total optimization time (in the worst case) would be:

$$t_{opt} = n \cdot t_m \cdot m \quad (2)$$

In the parallel simulation case, instead, if k is the number of simulations that can be run in parallel, the total simulation time (in the worst case) is reduced as follows:

$$t_{sim} = \lceil \frac{n}{k} \rceil \cdot t_m \cdot m \quad (3)$$

If $k \geq n$ this reduces to

$$t_{sim} = t_m \cdot m \quad (4)$$

This shows an improvement in terms of performance and scalability.

6.2 CPSwarm Deployment Toolchain

Typically, single CPS deployment workflows are based on direct connection between the programming/deployment tool and the target CPS (e.g., through direct cable connection or over the air). While this is perfectly acceptable for programming single CPS – considering that typical deployments tasks are quite infrequent – it may quickly become a serious bottleneck when multiple systems need to be configured and deployed at the same time, with relatively high frequency. In this last scenario, direct deployment quickly fails and leads to unacceptable high times for deploying new solutions, with a consequent impact on the ability to re-act to changing contexts.

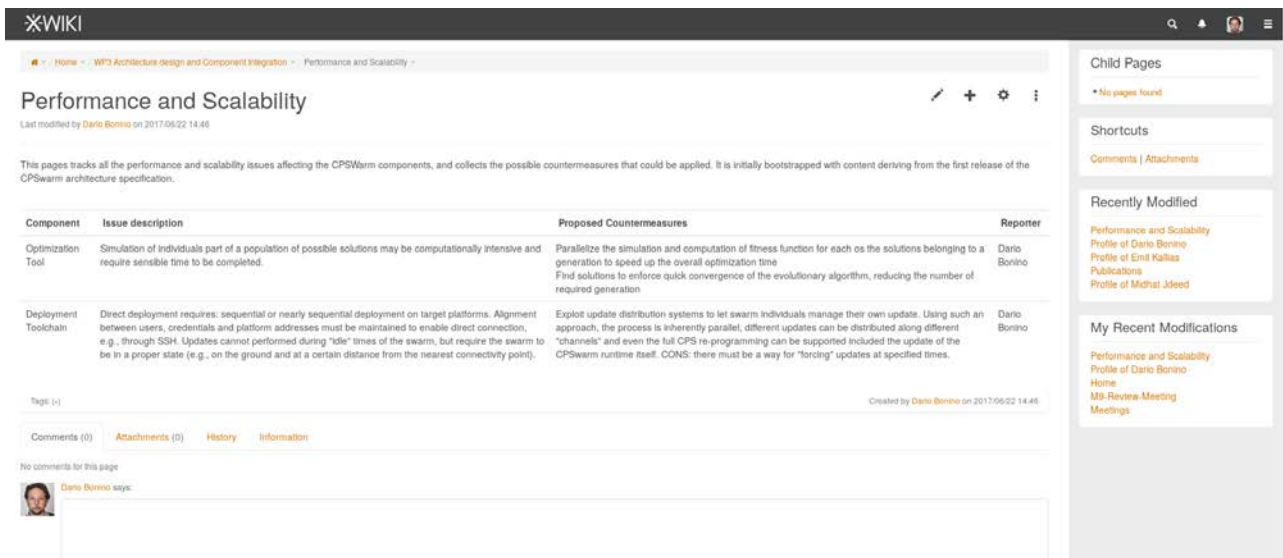
To address this scalability issue, several solutions can be exploited which were developed in the context of automatic update systems, and thoroughly tested in the last 20 years. Packaging systems and distribution systems are designed to dispatch updates to high numbers of platforms, typically not a-priori known. In these systems, new software releases are published on channels reachable over the Internet and target systems autonomously decide when download and installation of updates shall be performed, depending on their internal schedule and conditions. In contrast with direct deployment, deployment based on updates scales gracefully and does not add additional requirements on the deployment system, which can be fairly simple (a web application would be sufficient in many cases). On the other hand, it does not guarantee a precise point in time in which updates are installed on the target platforms. In other words, with the update-based approach scalability is traded-off with potentially high latency in code deployment.

In this first release of the CPSwarm architecture specification, the Deployment Toolchain has been designed by taking inspiration from update systems, thus enabling easier scaling to swarms composed by dozens of platforms. Nevertheless, dedicated mechanisms for “forcing” updates are under study, in the context of this

project, to trigger update checking at the CPS side in response to certain events generated by the Deployment Toolchain. To this purpose, the project is currently investigating the possibility of leveraging the CPS Telemetry components to host an inbound message queue (e.g., through MQTT subscriptions) for forcing updates at a specified time.

6.3 Summary and discussion

To better summarize the initial considerations on scalability and performance issues, reported in previous paragraphs, a dedicated performance and scalability page on the project wiki has been set up. Such a page is exploited by partners to timely trace detected bottlenecks and scalability issues and to propose possible counter measures.



Component	Issue description	Proposed Countermeasures	Reporter
Optimization Tool	Simulation of individuals part of a population of possible solutions may be computationally intensive and require sensible time to be completed.	Parallelize the simulation and computation of fitness function for each of the solutions belonging to a generation to speed up the overall optimization time Find solutions to enforce quick convergence of the evolutionary algorithm, reducing the number of required generation	Dario Bonino
Deployment Toolchain	Direct deployment requires: sequential or nearly sequential deployment on target platforms. Alignment between users, credentials and platform addresses must be maintained to enable direct connection, e.g., through SSH. Updates cannot be performed during "idle" times of the swarm, but require the swarm to be in a proper state (e.g., on the ground and at a certain distance from the nearest connectivity point).	Exploit update distribution systems to let swarm individuals manage their own update. Using such an approach, the process is inherently parallel, different updates can be distributed along different "channels" and even the full CPS re-programming can be supported included the update of the CPSwarm runtime itself. CONS: there must be a way for "forcing" updates at specified times.	Dario Bonino

Figure 34 - The Performance and Scalability tracking page on the project wiki.

Each of the issues listed on such a page will be strictly monitored during the implementation and integration phases and corresponding test protocols will be established (and documented in D3.7, D2.8 and D8.7). For each issue, quantitative metrics will be defined, e.g., the number of CPS systems deployable in one pass, to provide a sound and replicable evaluation of achieved results and to drive subsequent design refinements.

7 Future Steps

7.1 Future activities and timeline

This deliverable specified an initial architecture design for the CPSwarm system. As stated earlier, the architecture is subjected to modifications as the project evolves. Future changes and improvements will be documented in the later deliverable 3.2, *Updated System Architecture Analysis and Design Specification*, which is due on month 18.

According to the architecture diagram, a test and integration plan should be established very shortly. This plan should serve as the guideline for future development activities and it will be documented in deliverable 3.7, *Test and Integration Plan*. The deliverable is due on month 9.

7.2 Alternative architecture for future exploration

According to different use cases, two architectural approaches for the Optimization Environment are possible. In the first approach, the agent controller that is optimized resides only within the Optimization Tool. In the second approach, the Optimization Tool is running the optimization process by passing the code of the agent controller to the optimization simulator before a simulation.

In this deliverable, the focus is on presenting the software architecture with the first approach (Figure 8). The optimization simulator is used to evaluate each single step of the simulation providing the required sensor readings to the Optimization Tool and executing the actuator commands coming back from the Optimization Tool. This approach requires frequent communication between the Optimization Tool and the simulator thus a fast communication channel between them is crucial. This is the case if both the Optimization Tool and the Simulator run on the same system or are connected via a fast network. The actual influence on the performance depends on the ration between the communication overhead and the time to perform one time step in the simulator. If the computations of a simulator step take significantly longer in comparison to the communication of sensor and actuator values, this approach is feasible. Another case are simple problems, where the overall time to evolve a solution is short enough so that the waiting time is insignificant. The advantage of this approach is that a generic interface can be defined that receives and sends the interface data, thus allowing an easy integration of the Optimization Tool with several different simulators.

In the second approach, a code generation unit converts the controller algorithm into target code that can be executed by the simulator. The simulator would then execute a simulation run without further interaction with the Optimization Tool and communicates only the resulting fitness evaluation when the simulation ends. This approach requires to pass code from the Optimization Tool to the simulator and might need a compilation run with the exported code with the simulation. While the overall execution time of the simulation time will be shorter than in the first approach, the starting and set-up time for the simulation might increase, for example if a recompilation of the simulation is required. The advantage of this approach is that the same component can be used to generate the code for deployment in the simulation or on the target hardware, thus the generated code is already evaluated in the simulation.

Although both approaches have their advantages and disadvantages in different use cases, the conclusion is that approach one represents the most common use cases and should be prioritized. To establish a clear goal in the CPSwarm project, the effort will focus on the implementation of the first approach. The second approach will only be explored, if the implementation of the first approach is finished and resources are still available for the exploration.

8 Conclusions

This deliverable presents the available methodologies, tools and standards for swarm development as well as an initial version of architecture design for the CPSwarm system, which is derived from the system requirements gathered in WP2.

One objective of this document is to establish a technical common ground among partners for future developments. On one hand, tools that are available to be utilized in CPSwarm were presented as later implementation options. On the other hand, components and interfaces were defined in the architecture design, serving as a blueprint for future development of WP4, WP5, WP6 and WP7.

Now that components and interfaces are defined, this deliverable also paves the way for the next activity of WP3, namely Task 3.3, *Continuous System Integration*. The result in this deliverable will be the foundation of establishing the test and integration plan for the CPSwarm system.

As next steps, the architecture design will be further revisited, revised and refined as the project evolves. The new modification will be documented in later deliverable D3.2, *Updated System Architecture Analysis and Design Specification*.

Conclusively, this document presents the first step in the iterative process of the overall CPSwarm architecture design. This is a first significant result that will be used as input to subsequent activities of the project.

Acronyms

Acronym	Explanation
CPS	Cyber physical system
SITL	Software in the loop
HITL	Hardware in the loop
PFSM	Probabilistic finite state machines
ANN	Artificial neural network
DSL	Domain specific language
RTE	Runtime environment
IoT	Internet of Things
ROS	Robot Operating System

List of figures

Figure 1 - The PX4 software architecture.	12
Figure 2 - Process of designing a swarm intelligence model and the corresponding algorithm (adapted from [5]).....	15
Figure 3 - Architecture description concepts (Adapted from [1]).....	21
Figure 4 - Activities supporting architecture definition [29].....	22
Figure 5 - CPSwarm conceptual architecture diagram.....	24
Figure 6 - Data flow between workbench components (extracted from D2.3).....	26
Figure 7 - CPSwarm system context diagram.....	28
Figure 8 - Overview of components in CPSwarm system.....	30
Figure 9 - Functional structure model of Modelling tool.....	30
Figure 10 - Overview of Modelling tool view.....	31
Figure 11 - Optimization Tool.....	33
Figure 12 - Optimization Simulator.....	35
Figure 13 - Sample Finite State Machine specification.....	37
Figure 14 - Excerpt of template for template-based generation pattern of a state machine in ROS.....	37
Figure 15 - Code Generator: preliminary architecture.....	38
Figure 16 - High-level activity diagram describing the CPSwarm Code Generator behavior.....	39
Figure 17 - Deployment Tool: preliminary architecture.....	40
Figure 18 - Functional structure model of Monitoring and Configuration Framework.....	42
Figure 19 - The CPSwarm Runtime Environment.....	43
Figure 20 - The CPSwarm abstraction layer.....	43
Figure 21 - The CPSwarm Update System.....	44
Figure 22 - The CPSwarm update process.....	45
Figure 23 - The CPSwarm Telemetry components.....	45
Figure 24 - Data flow of CPSwarm system.....	47
Figure 25 - Parameters for the Algorithm Optimization Framework.....	49
Figure 26 - Preliminary data-model of Code Generation directives.....	51
Figure 27 - Preliminary data-model of Bundling Directives.....	51
Figure 28 - Preliminary data-model of Deployment configuration.....	52
Figure 29 - Data flow within the optimization tool.....	53
Figure 30 - The OGC Sensor Things API data model.....	55

Figure 31 - Network diagram of CPSwarm component during the deployment.....	57
Figure 32 - Threats categorized by CIA.....	60
Figure 33 - Countermeasures to the CPSwarm system.....	62
Figure 34 - The Performance and Scalability tracking page on the project wiki.....	66

List of tables

Table 1. Initial list of simulation tools analysed to direct the architecture design.....	10
Table 2. . Initial list of 3D Simulation tools analysed to direct the architecture design.....	11
Table 3. Modelling standards for CPS swarm design.....	13
Table 4. Stakeholders of the CPSwarm System (extracted from D2.1).....	24
Table 5 - Mapping between components specified in D2.3 and those in architecture design.....	27

References

- [1] IEEE, *ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description*, 2011.
- [2] J. Craighead, R. Murphy, J. Burke and B. Goldiez, "A Survey of Commercial and Open Source Unmanned Vehicle Simulators.," in *Proceedings of ICRA 2007*, 2007.
- [3] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, p. 132, 2007.
- [4] L. Chee Peng and S. Dehuri, *Innovations in Swarm Intelligence*, Springer-Verlag Berlin Heidelberg, 2009.
- [5] H. R. Ahmed and J. I. Glasgow, "Swarm Intelligence: Concepts, Models and Applications," Kinston, Canada, Queen's University, School of Computing Technical Reports, 2012.
- [6] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," in *SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987.
- [7] M. Brambilla, E. Ferrante, M. Birattari and M. Dorigo, "Swarm robotics: a review from the swarm engineering perspective," *Swarm intelligence*, pp. 1-14, 2013.
- [8] M. Granovetter, "Threshold Models of Collective Behavior," *American Journal of Sociology*, pp. 1420-1443, May 1978.
- [9] O. Soysal and E. Sahin, "Probabilistic aggregation strategies in swarm robotic systems," in *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*.
- [10] S. Nouyan, A. Campo and M. Dorigo, "Path formation in a robot swarm," *Swarm Intelligence*, vol. 2, no. 1, pp. 1-23, 2008.

- [11] T. H. Labella, M. Dorigo and J.-L. Deneubourg, "Division of labor in a group of robots inspired by ants' foraging behavior," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 1, no. 1, pp. 4-25, September 2006.
- [12] L. E. Parker, "L-ALLIANCE: task-oriented multi-robot learning in behavior-based systems," *Advanced Robotics*, vol. 11, no. 4, pp. 305-322, 1996.
- [13] J. Lee and R. Arkin, "Adaptive multi-robot behavior via learning momentum," in *Intelligent Robots and Systems, 2003. (IROS 2003)*, 2003.
- [14] L. Li, A. Martinoli and Y. S. Abu-Mostafa, "Systems, Learning and Measuring Specialization in Collaborative Swarm," *Adaptive Behavior*, vol. 12, no. 3-4, pp. 199-212, 2004.
- [15] S. Hettiarachchi, *Distributed Online Evolution for Swarm Robotics*, University of Wyoming, Laramie, WY., 2007.
- [16] A. Rosenfeld, G. A. Kaminka, S. Kraus and O. Shehory, "A study of mechanisms for improving robotic group performance," *Artificial Intelligence*, vol. 172, no. 6-7, pp. 633-655, 2008.
- [17] J. Pugh and A. Martinoli, "Parallel learning in heterogeneous multi-robot swarms," in *Evolutionary Computation, 2007. CEC 2007*, 2007.
- [18] D. He, H. Ren, W. Hua, G. Pan, S. Li and Z. Wu, "FlyingBuddy: augment human mobility and perceptibility," in *13th International Conference on Ubiquitous Computing*, 2011.
- [19] S. Schneegass, F. Alt, J. Scheible and S. A., "Midair displays: concept and first experiences with free-floating pervasive displays," in *International Symposium on Pervasive Displays*, 2014.
- [20] R. W. Picard, "Affective computing: challenge," *International Journal of Human-Computer Studies*, pp. 55-64, 2003.
- [21] C. Breazeal, "Emotion and sociable humanoid robots," *International Journal of Human-Computer Studies*, pp. 119-155, 2003.
- [22] H. Chao, M. M.Q., L. P.X. and W. Xiang, "visual gesture recognition for human-machine interface of robot teleoperation," in *Intelligent Robots and Systems, 2003. (IROS 2003)*, 2003.
- [23] J. R. Cauchard, K. Y. Zhai, M. Spadafora and J. A. Landay, "Emotion Encoding in Human-Drone Interaction," in *The Eleventh ACM/IEEE International Conference on Human Robot Interaction*, Christchurch, 2016.
- [24] R. A. S. Fernández, J. L. Sanchez-Lopez, C. Sampedro, H. Bavle, M. Molina and P. Campoy, "Natural user interfaces for human-drone multi-modal interaction," in *International Conference on Unmanned Aircraft Systems (ICUAS)*, Arlington, 2016.
- [25] A. Adams, "Human-Robot Interaction Design: Understanding User Needs and Requirements," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 2016.

- [26] J. Scholtz, B. Antonishek and J. Young, "Evaluation of a Human Robot Interface: Development of a Situational Awareness Methodology.," in *Hawaii International Conference on System Sciences*, 2004.
- [27] H. A. Yanco, J. L. Drury and J. Scholtz, "Beyond usability evaluation: analysis of human-robot interaction at a major robotics competition," *Human Computer Interaction*, vol. 19, no. 1, pp. 117-149, 2004.
- [28] J. L. Drury, J. Scholtz and D. Kieras, "Adapting GOMS to model human-robot interaction,," in *nd ACM/IEEE International Conference on Human-Robot Interaction (HRI)*,,, Arlington, 2007.
- [29] N. Rozanski and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, Addison-Wesley Professional, 2011.
- [30] ITU-T-Y.2060, *Overview of the Internet of Things*, 2012.
- [31] A. e. a. Cardenas, "Challenges for securing cyber physical systems.,," *Workshop on future directions in cyber-physical systems security*, vol. 5, 2009.
- [32] Y. e. a. Gao, "Analysis of security threats and vulnerability for cyber-physical systems," in *Computer Science and Network Technology (ICCSNT), 2013 3rd International Conference on IEEE*, 2013.
- [33] F. A. T. a. K. M. M. Higgins, "Survey on security challenges for swarm roboticsSurvey on security challenges for swarm robotics," in *Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on. IEEE*, 2009.
- [34] I. Grigg, *Pareto-Secure, A definition of security using the theory of Pareto Efficiency*, 2005.
- [35] A. Mpitiopoulos, D. Gavalas, C. Konstantopoulos and G. Pantziou, "A Survey on Jamming Attacks and," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 42-56, 2009.
- [36] A. B. A. V. Elisabeth Hansson, "Security in mobile ad hoc networks," 2005.
- [37] Z. Huang and Y. Chen, "Log-Linear Model Based Behavior Selection Method for Artificial Fish Swarm Algorithm," *Computational Intelligence and Neuroscience*, p. 10, 2015.
- [38] A. Cardenas, S. Amin and B. Sinopoli, "Challenges for securing cyber physical systems.,," *Workshop on future directions in cyber-physical systems security*, vol. 5, 2009.
- [39] Y. Gao, Y. Peng and F. Xie, "Analysis of security threats and vulnerability for cyber-physical systems," in *Computer Science and Network Technology (ICCSNT), 2013 3rd International Conference on IEEE*, 2013.
- [40] F. Higgins, A. Tomlinson and K. M. Martin, "Survey on security challenges for swarm roboticsSurvey on security challenges for swarm robotics," in *Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on. IEEE*, 2009.

