D5.3 – UPDATED CPSWARM MODELLING TOOL

Deliverable ID	**D5.3**
Deliverable Title	**Updated CPSwarm Modelling Tool**
Work Package	**WP5 – CPSwarm Design Workbench**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**30-06-2018**
Status	**Final**
Lead Editor	**Melanie Schranz (LAKE)**
Main Contributors	**Gianluca Prato (ISMB), Etienne Brosse (SOFTEAM)**

Published by the CPSwarm Consortium

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

Document History

Version	Date	Author(s)	Description
0.1	2018-02-07	Melanie Schranz (LAKE)	Initial TOC
0.2	2018-05-07	Melanie Schranz (LAKE)	Update initial TOC
0.3	2018-06-04	Melanie Schranz (LAKE)	4.4 Modelling dynamic environment
0.4	2018-06-13	Melanie Schranz (LAKE)	Include content for code generation
0.5	2018-06-25	Melanie Schranz (LAKE)	Include content for modelling
1.0	2018-06-30	Melanie Schranz (LAKE)	Integrated comments from internal review (DIGISKY, ROBOTNIK) and updates from SOFTEAM

Internal Review History

Review Date	Reviewer	Summary of Comments
2018-06-28	Etienne Brosse (SOFTEAM)	Minor comments.
2018-06-25	Ángel Soriano (ROBOTNIK)	Few modifications and comments.

Contents

1 Executive summary

This deliverable, namely "**D5.3 – Updated CPSwarm Modelling Tool**", introduces three parts of implementation of the CPSwarm workbench related to modelling. This includes the CPS population design tool that will be implemented as entry point for the modelling tool; the updates to the modelling tool itself together with the design of state machines; and the generation of code for the deployment process using the modelled state machines.

2 Introduction

As described in CPSwarm deliverable D3.2 - Updated System Architecture & Design Specification, delivered at M18 - the CPSwarm architecture adopts a launcher-based definition, where each component of the system is connected to a central launcher able to provide a set of well-defined functionalities as shown in Figure 1.

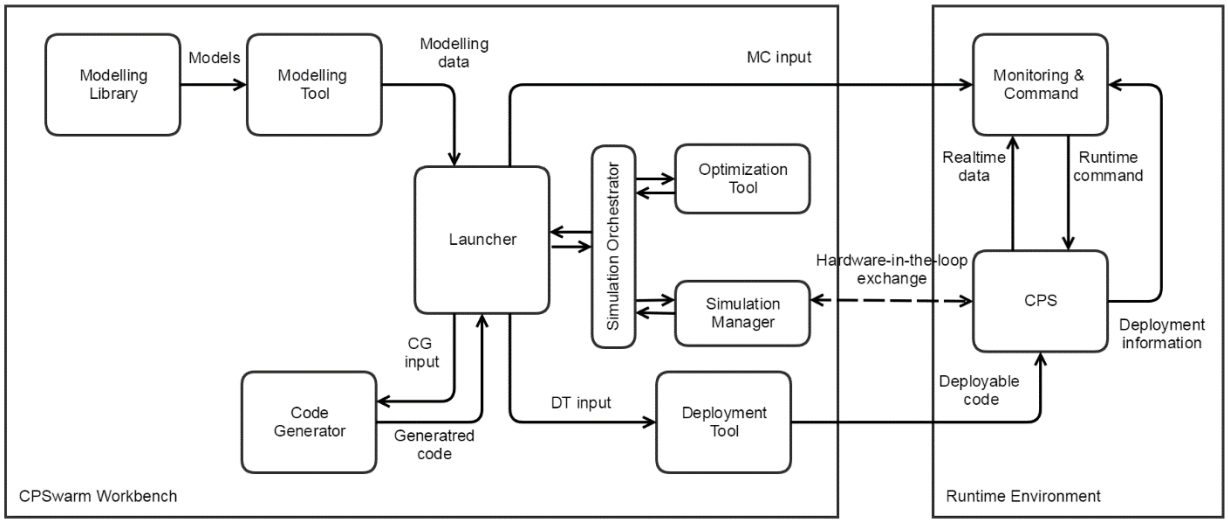

Figure 1 Overview of the Software Components in CPSwarm (see D3.2 for more information)

This "**D5.3 - CPSwarm Modelling Tool**" is a public deliverable focused on the Modelling Tool implementation on CPSwarm M18. It details the M18 status of Modelling Tool component and its implemented interfaces with related components (including the CPS population design tool and the related code generation).
LAKE, as deliverable leader, initially drafted the document, which has subsequently been enriched by all partners' contributions describing their developments.

2.1 Scope

This deliverable describes the M18 implementation of CPSwarm modelling tool and its connections to other CPSwarm components. For each component we provide a short description, but focus on concepts and implementations.

2.2 Document organization

The remainder of this deliverable is organized as follows:
Section 3 describes the concept of the CPS population design tool that will be implemented in Modelio. Section 4 describes the Modelling Tool and its updates in state machine design. Finally, Section 5 focuses on the code generation out of the state machines provided by Modelio in Section 4.

2.3 Related documents

ID	Title	Reference	Version	Date
[D3.2]	Updated System Architecture & Design Specification	D3.1	1.0	30/06/2018
[D4.1]	Initial CPS Modelling Library	D4.1	1.0	30/09/2017
[D5.2]	Initial CPswarm Modelling Tool	D5.2	1.0	30/09/2017
[D4.4]	Initial Swarm Modeling Library	D4.4	1.0	30/10/2017
[D5.1]	CPSwarm Modelling Language Specification	D5.1	1.0	31/12/2017

3 CPS Population Design Tool

The CPS population design tool (CPDT) is used to define the swarm composition. The main idea is to provide a simple entry point, where a swarm can be pre-configured through a wizard. This includes the type and number of CPSs to be included in the swarm. First, we proposed that this tool resides within the launcher (see 2). But, as it needs a direct connection with the modelling library (that only has a connection with the modelling tool), we decided to implement the CPDT as entry point to the modelling tool in form of a wizard.

3.1 Design

Figure 2 gives a first design example of the CPDT within the launcher. The user can compose the swarm by selecting existing CPS that is extracted from the modelling library. The user interface could also be drag and drop based. The modelling tool only needs to be launched in order to adapt the selected models or to add new ones. When the modelling is done, the changes are saved to the modelling library. If the CPS is saved under a different name, the CPDT should automatically update the selected swarm composition.

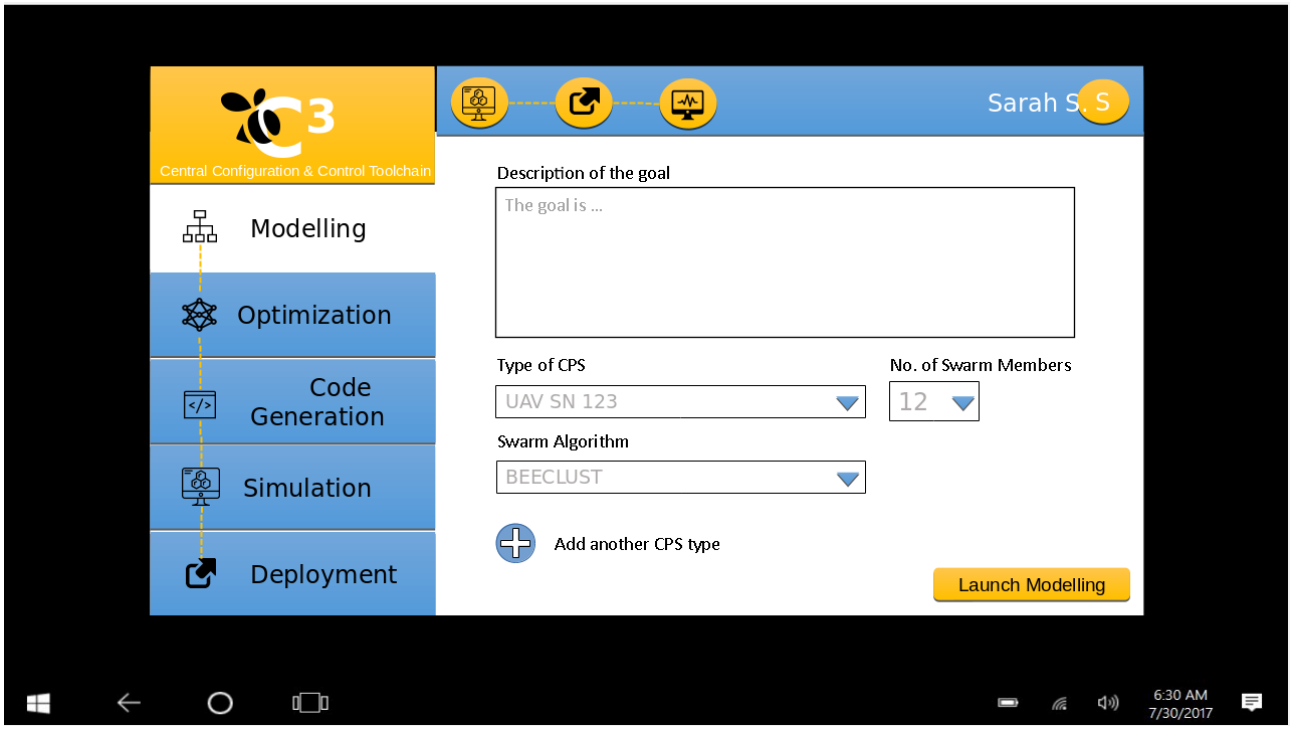

Figure 2 CPDT sketched as part of the launcher

3.2 Parameters

The parameters should allow to define a swarm of CPS. We propose to stay on a high level by only selecting from existing swarm member structures and behaviors. Alternatively, the swarm member structure could be refined here by allowing parameters such as the communication interface.

Description of the goal
This is a textual description of the goal of the swarm application. This information is especially useful if there are more than one person working on the project, e.g., a modeler and a programmer.

Type of CPS
This parameter selects one of the existing CPSs from the modelling library. It defines the hardware, i.e., the swarm member structure. If none of the existing CPSs is appropriate, a new one can be modeled with the

modelling tool and saved to the modelling library. Also, a very abstract agent could be selected here if only simulations are to be carried out.
Exemplary options are:

- DIGISKY UAV,
- ROBOTNIK rover,
- Spiderino,
- Add new.

Add another CPS type
This option allows creating heterogeneous swarm populations by simultaneously selecting different types of CPSs.

Swarm Algorithm
This parameter selects one of the existing swarm algorithms from the modelling library. It defines the software, i.e., the swarm member behavior. If the user wants to create a custom swarm algorithm, he can launch the modelling tool to create a state machine from simple behaviors or he can continue to the optimization pane for evolving an algorithm.
Exemplary options are:

- BEECLUST,
- Random Walk,
- Flocking,
- Evolutionary optimization,
- Add new.

3.3 Future work

In a next step, the CPDT will be implemented in Modelio as Wizard. Together with WP8 and WP4 activities the hardware description will be collected by the individual industrial partners, and modeled in Modelio. As soon as they are part of the modelling library, they can be connected with the CPDT and will be available for selection there.
During the first test phases of the CPDT with the industry partners, we are still open to enlarge the wizard. For example, there is still the possibility to add more parameters to select in advance, including:

- Swarm member structure
    - Sensors
    - Actuators
    - Communication device
- Swarm member behavior
    - Add bio-inspired swarm algorithm
    - Add state machine based swarm algorithm
- Environment
- Optimization
    - Fitness function
    - Objective: What needs to be minimized? What needs to be maximized?

4 Modelling Tool

4.1 Overview

The CPSwarm Modelling Tool is built on top of Modelio open source modelling environment as previously described in Deliverable D5.2. Swarm modelling activity can be succinctly described as the creation and population of several diagrams or views. The following sections describe how to model a simple but complete swarm.

4.2 Swarm Modelling

4.2.1 Create a new CPSwarm model

Modelling is always a difficult task to carry out from scratch. In this case, guidance is helpful. The main goal of this swarm template generation command is to help the Modeler to create a simple CPS swarm model with all minimum concepts. The CPS swarm generation can be done by right clicking on any package, then selecting CPSwarm > CPS swarm creation entry as depicted in the following figure (Figure 3).

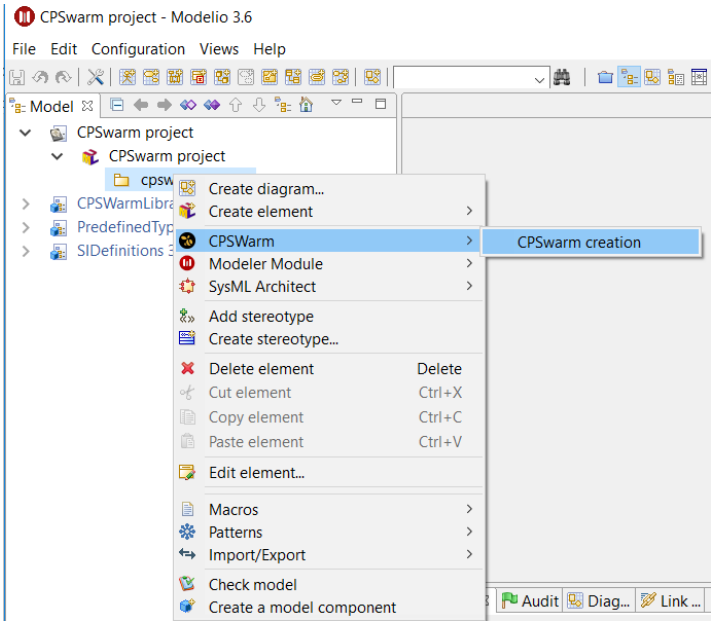

Figure 3 Creating a new swarm modelling

Figure 4 shows the result of the CPS swarm template generation.

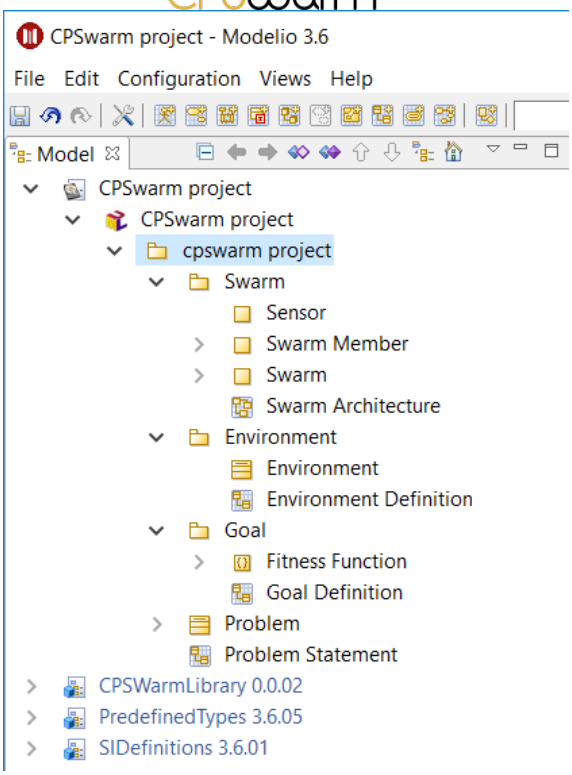

Figure 4 New swarm result

The swarm template generator produces a set of initial diagrams (as shown in Figure 5) that have been identified as necessary to completely model a CPS swarm.

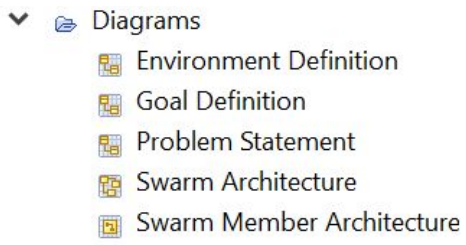

Figure 5 CPSwarm predefined diagrams

The CPSwarm modeler can modify the initial content following the needs of the specific case study he/she is modelling. The CPSwarm modeler will find on the right part of each of the diagrams (namely Environment Definition, Goal Definition, Problem Statement, Swarm Architecture, Swarm member architecture) the predefined selection of the modelling elements he/she can specifically use for that specific diagram context.

4.2.2 Swarm Composition Modelling

A swarm is composed of one to many Swarm Member. To model this relation, you must use the UML composition relation from the Swarm block to one or many Swarm Member block. The multiplicity at the end of the relation indicates the number of Swarm Member instance. Figure 6 depicts a Swarm composed of one unique Swarm Member.

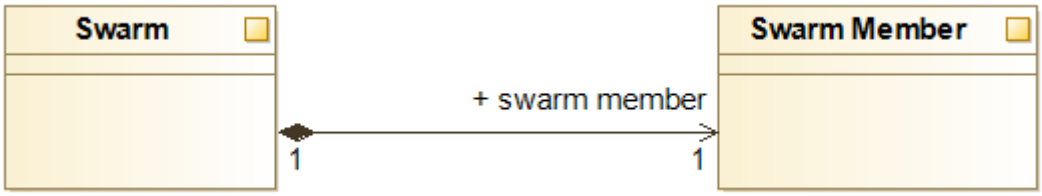

Figure 6 Swarm Architecture Modelling Elements

4.2.3 Swarm Member Architecture Modelling

Another aspect of Swarm Modelling is the specification of each Swarm Member internal architecture. This specification is made in two times. First, the list of internal components (which can be a controller, a sensor, or an actuator component) must be defined. Each of this internal component must expose the data it provides or requires. Figure 7 represents a simple component having two ports respectively named FlowPort and FlowPort1. FlowPort expresses the fact that the component provides a Boolean value at contrary FlowPort1 expresses the fact that the component requires a Boolean

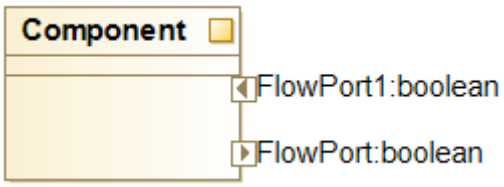

Figure 7 Simple sub Component example

The second steps in modelling the internal architecture of a Swarm Member consists in instantiate each appropriate component and connect them between each other. In Figure 8, the Component predefined previously has been instanced twice and each port has been connected to model the data flow between the internal component

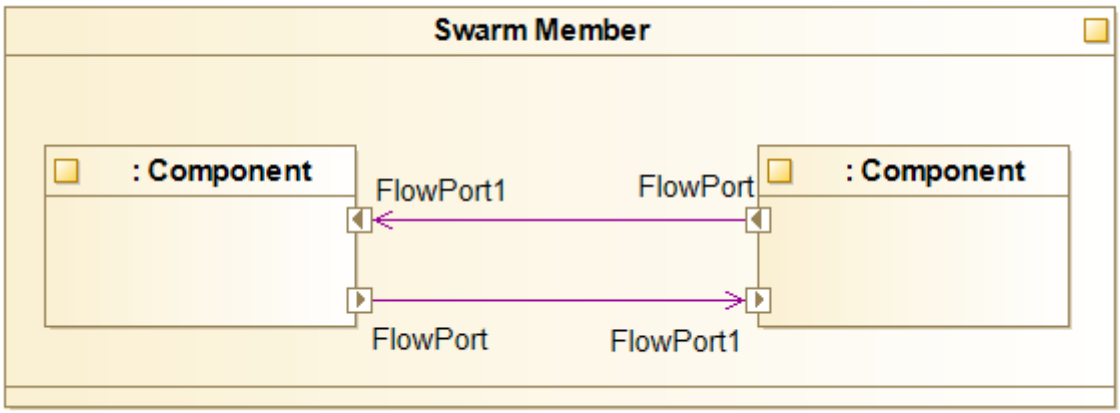

Figure 8 Swarm Member Architecture Example

4.2.4 Swarm Member Behaviour Modelling

The internal architecture of a Swarm Member is a key aspect of its definition. The second key aspect is its internal behavior. As defined in Deliverable D5.1, UML state machines are used to model Swarm Member behavior. Figure 9 depicts the simplest possible Swarm Member behavior. This latter is simply composed of a State named State. Both Initial and Final state are mandatory to all State Machine. The two transitions respectively connect the initial State to the State state and the State state to the Final state

Figure 9 Simple Swarm Member Behaviour

Of course, a real behavior will be more complex. Figure 10 for example represents two states - aka State1 and State 2 – executed in parallel.

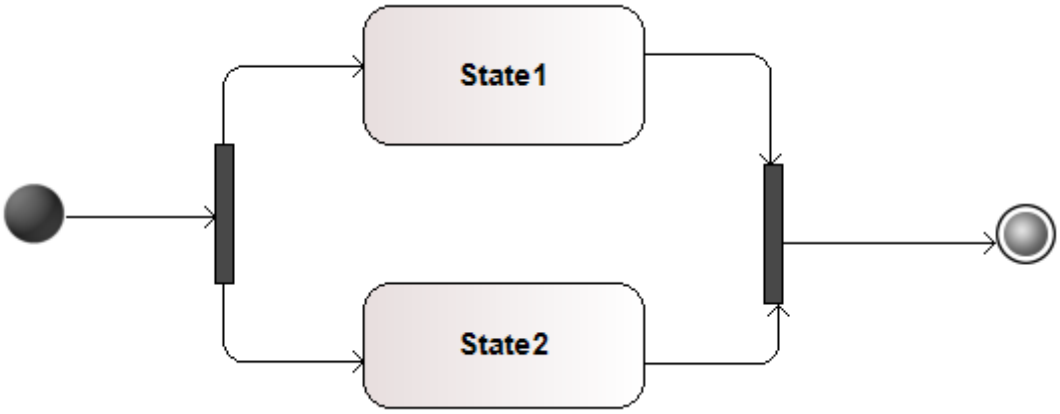

Figure 10 Swarm Member Behaviour

To handle the complexity of these state machines it is possible to extract part of them inside another state machine and then refer this extracted content as a sub state machine. Figure 11 shows the call of a sub state machine by a particular State.

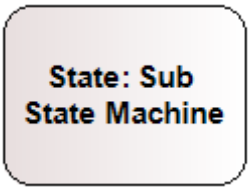

Figure 11 Hierarchical State

4.3 Swarm Modelling Library

As described in deliverable D3.2, the Swarm Modelling library is composed of a set of predefined.

- Environment;
- Cost function;
- Swarm Member;
- Hardware Component;
- Behavior;

This predefined set of elements can be reused, for example Figure 12 shows extract of this modelling library. In this extract, a component named Controller is model with four possible actions respectively named Send, Pick, Place, and PickAndPlace.

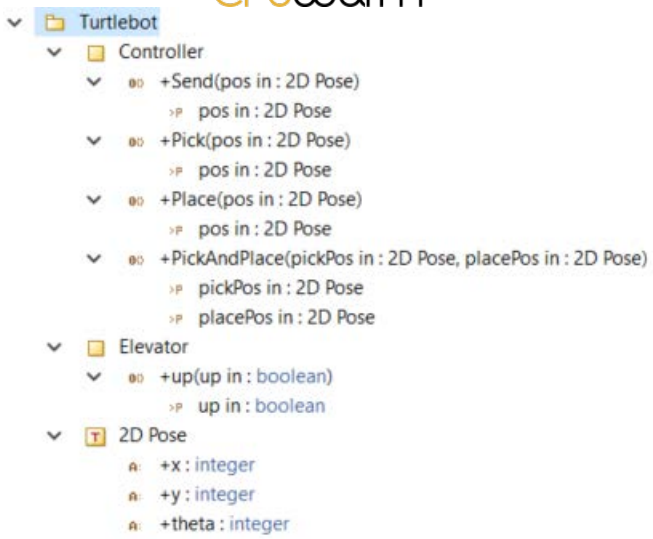

Figure 12 Part of the Modelling Library

The following illustration shows through a simple behavior modelling, the reuse of the Up action inside another Swarm Member behavior:

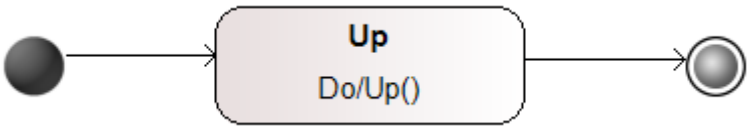

Figure 13 Simple reuse of the Modelling Library

4.4 Dynamism in CPS swarms

In the CPSwarm project cyber-physical systems (CPS) operate in real environments. Real environments are inherently dynamic. Thus, swarms of CPSs should have the ability to cope with multiple, dynamically varying critical and non-critical constraints coming from the dynamics beyond the CPS control including i) environment, ii) scenario, and iii) interactions with other systems in the environment. As soon as the CPSs are able to get along with the run-time dynamics and the corresponding situation-specific control actions, trustworthy behavior can be ensured in critical CPS solutions.

The key innovative elements in CPSwarm are to enable a population of interacting CPSs that jointly deliver a task without a central point of coordination. Furthermore, they are able to cope with dynamic reconfiguration and emergent properties from the environment, scenario, and interaction with other systems.
CPSwarm-designed systems will provide a previously unreachable flexibility, adaptability and capability to deal with complex, dynamic and heterogeneous problem landscapes. The solutions are self-organized algorithms, more specifically swarm algorithms. These algorithms lead to a behavior that overcomes many unforeseeable obstacles and events in real-time, thus reaching the final goal of reliable, scalable, and robust systems by simultaneously improving trust to humans involved in the loop. Swarms of CPSs operate as a self-organizing mixed team where particular tasks for each CPS are not predefined at mission start, but negotiated during mission execution. Such a swarm is highly adaptive to changes in the environment and can act dynamically.

In this project we won't deal with the modelling of dynamic environments, as this field belongs to the simulation of swarms of CPS rather than to engineering of CPS swarms. Therein, explicit concepts and constructs that relate to the real world are modelled. These include the modelling of all possible environmental changes expressed in state variables, equations, time and state events focusing on the

constituents involved, the time interval of occurrence, the error rate and the evolution strategy of all parts of the environment (Helleboogh, et al., 2007), (Dykes, et al., 2015).

5 Code Generation for CPS Systems

In order to ease the development of new CPSs' applications the CPSwarm workbench supports the use of model-driven paradigm. The main advantage of this approach consists in the possibility to design CPSs' behaviors without being concerned about the underling hardware details. Moreover, the focus of the user is raised from a platform-dependent level to a more application-centric perspective, giving also to not domain-expert programmers the opportunity to develop new algorithms in the robotic context.
Inside the CPSwarm workbench, the Code Generator (CG) perform two different tasks:

- Generating the whole CPS's behavior starting from a formal description. In this case, the produced output will be passed to the Deployment Tool to be installed on the actual CPSs.
- Wrapping an algorithm generated by the Optimization Tool in order to be simulated inside one of the integrated Simulator Engines.

In the next section this two functionalities will be described in more details.

5.1 CPS's behaviour generation

In the first half of the project particular attention has been given to the modelling of new CPS behaviors using Finite State Machines (FSM), as described in Section 4.2.4. In consequence of that, the first release of the Code Generator supports the generation of code starting from the formal description of a FSM. The SCMXL[1] data format has been recognized as a proper language to describe finite state machines, with a CPSwarm dedicated extension that will be described in the following part of this paragraph. While this format allows the description of very complex state machines with sub-states (nested state machines inside a single high-level state) and concurrent states, the first implementation of the Code Generator only supports the generation of simple state machines with these characteristics:

- The state machine has just one level, so no nested state machines can be generated.
- Each state of the FSM corresponds to one of the functionalities accessible through the CPSwarm Abstraction Library.

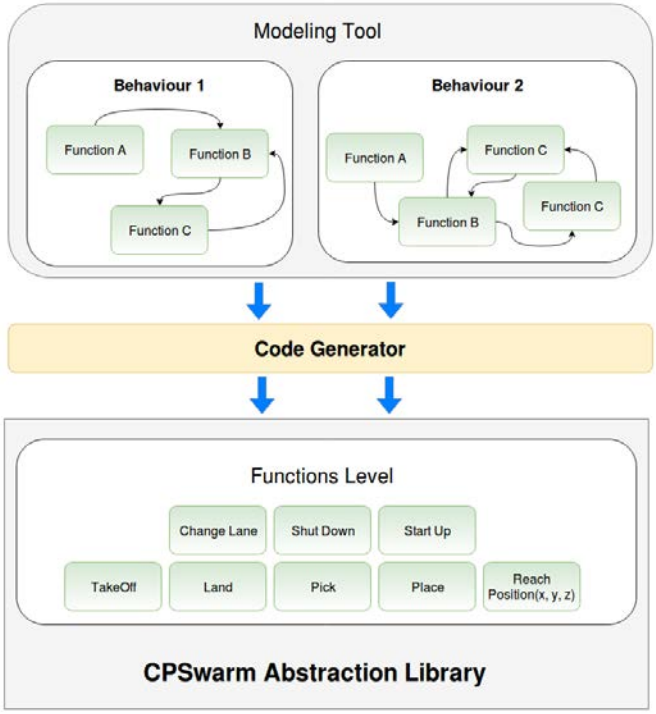

Figure 14 The role of Code Generator in the CPSwarm workbench

[1] https://www.w3.org/TR/scxml/

In Figure 15 can be observed the role of the CG as a "glue" between the modelling level (realized inside the Monitoring Tool) and the deployment of the code on the actual CPS. To achieve this task, the CG relies on the Abstraction Library that provides a set of platform-independent functionalities that can be executed by the robot.

Due to the very schematic and repeatable structure that all algorithms defined using a FSM have, the template-based technique has been selected as the most proper approach to complete the code generation task. As already depicted in D3.1, a template-based code generator can be described by his 3 main components (summarized in Figure 15):

- The **data**: this part corresponds to the set of information needed to produce the output. In order to be correctly processed by the Code Generator, the data have to conform to a specific data format (e.g. SCXML).
- The **templates**: a template is text file and it is usually composed by a *static* part that appears in the output "as it is" and a dynamic part written with a template meta-code. This code contains all the directives that are processed at runtime (together with the input data) by a *template engine* to produce the final source code, also called output. A set of templates is prepared for each target runtime platform (ROS, Python, …). To select the correct template, the CG must be configured with the target runtime environment.
- The **output**: the source code that is produced as result of the generation task.

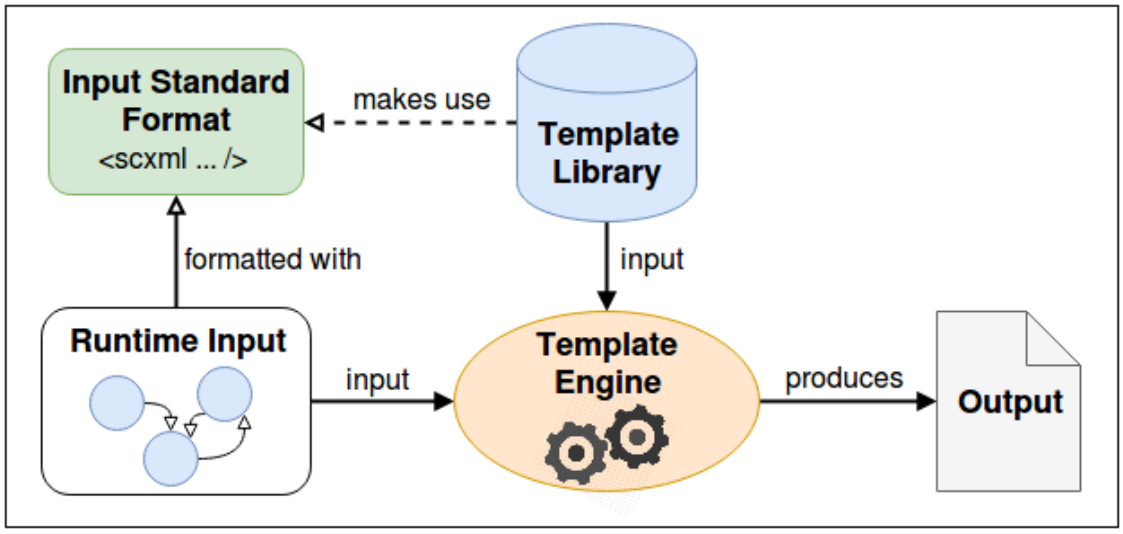

Figure 15 Code Generator main components

In this implementation of the CPSwarm GC the code generation process is driven by a Java-based template engine called Velocity[2]. This tool was chosen because of his powerful and simple template language called Velocity Template Language (VLT). In addition to this template engine, the Apache Commons SCXML library[3] has been used to parse the SCXML input files.

With the first set of templates it is possible to generate a code implementation of a state machine that relies on the SMACH[4] library, a Python-based project that let easily implement and execute State Machine-designed algorithm. The choice has fallen to this library not only for its extreme simplicity and scalability, but also for its direct integration with ROS, the runtime environment supported by 3 out of 4 of the CPS platforms that will be used in our use case scenarios.

To better clarify all the aspects presented up to this point, a simple example of a code generation process extract from the Search and Rescue scenario is now presented.

[2] http://velocity.apache.org/
[3] Apache Commons SCXML
[4] http://wiki.ros.org/smach

5.1.1 Code generation example

In Figure 16 a simple state machine for a flying drone is presented. It is composed by 2 states:

- **Idle**: in this state the drone just waits for a fixed amount of time without executing any specific action. After this wait, the "*missionStart*" event is triggered and the TakeOff state is selected as next state.
- **TakeOff**: when this state is reached the procedure to take-off is triggered and the drone reach a predefined height.

Figure 16 Simple FSM

During the design phase, each state of the FSM is associated with a specific functionality exposed by the Abstraction Library implementation aboard the drone. For instance, as shown in Figure 17, the TakeOff state is linked with the MavROS Takeoff functionality.

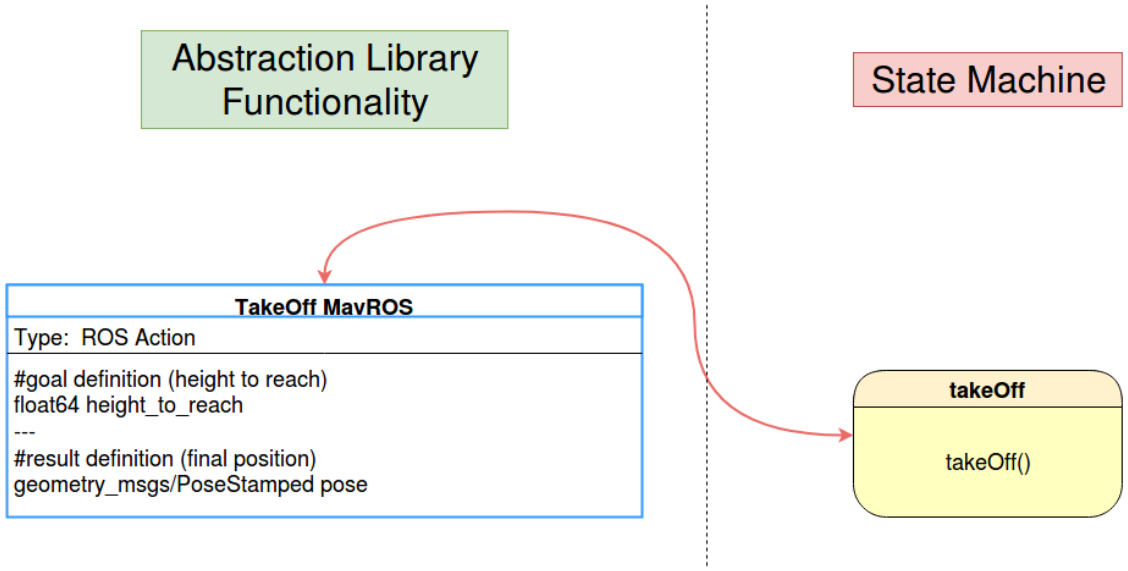

Figure 17 Linking a State with an Abstraction Library functionality

When the modelling phase is completed, the resulting FSM is translated by the modelling tool in an SCXML file as can be seen in Figure 18. The SCXML format has been extended with specific *CPSwarm* tags to describe how the Abstraction Library functionality should be called inside the state. In particular, The CG receive two information in order to correctly call the function **takeOff()**:

- The type of the functionality. In this case the value is "ROS_ACTION". This information is used to select the correct templates during the generation process.
- A description of inputs and outputs of the function. This information is used to provide all the needed inputs to the function and to properly manage its outputs.

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml xmlns="http://www.w3.org/2005/07/scxml"
       xmlns:cpswarm="http://my.custom-actions.domain/cpswarm/CUSTOM"
       version="1.0" initial="Idle" name="Sar">

       <state id="Idle">
              <transition event="missionStart" target="TakeOff" />
       </state>

       <state id="TakeOff">
              <invoke id="takeoff_action" type="ROS_ACTION">
                <cpswarm:input type="TakeOffAction" paramlist="altitude"  values="5"/>
                <cpswarm:output paramlist="finalpose" vartypes="PoseStamped"/>
              </invoke>
       </state>
</scxml>
```

Figure 18 SCXML description of the example FSM

Using the SCXML file and the selected set of templates ready for the desired runtime environment, the code is
finally generated. In Figure 19, there is the final output of the whole process as a Python ROS node implementing the designed FSM using the SMACH library.

```
#!/usr/bin/env python

import rospy
import smach
import smach_ros
import actionlib
import mavros_msgs

from state_machine_demo.msg import *
from mavros_msgs.srv import *

# define state Idle
class Idle(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=['missionStart'])

    def execute(self, userdata):
        rospy.loginfo('Executing state Idle')
        rospy.sleep(10.0)
        return 'missionStart'

def main():
    rospy.init_node('behaviour')

    # Create a SMACH state machine
    sm = smach.StateMachine(['succeeded','aborted','preempted'])

    # Open the container
    with sm:
        # Add states to the container
        smach.StateMachine.add('Idle', Idle(),
                               transitions={'missionStart':'TakeOff'})

        smach.StateMachine.add('TakeOff',
                               smach_ros.SimpleActionState('takeoff_action', TakeOffAction),
                               transitions={'succeeded':'succeeded'})

    # Create and start the introspection server
    sis = smach_ros.IntrospectionServer('my_smach_introspection_server', sm, '/SM_ROOT')
    sis.start()
    rospy.sleep(1.0)

    # Execute SMACH plan
    outcome = sm.execute()

    rospy.signal_shutdown('All done.')

if __name__ == '__main__':
    main()
```

Figure 19 Final Code Generator output

5.2 Candidate's wrapper generation

In this paragraph the second task that the Code Generator must fulfill is described. Even if this functionality is not implemented yet, a quick overview is given for the sake of completeness and to present the current understanding related to this process.

Other than generating a behavior to be deployed on an actual CPS, the CG is also involved during the optimization process of an algorithm evolved using the Optimization Tool. In this case, it provides a service to the Simulation and Optimization Environment Orchestrator (also known as SOEnvO). Since the candidate is a C code produced by the Optimization Tool, in order to be executed inside one of the Simulation Engine supported by the CPSwarm workbench (e.g. Gazebo) a wrapper code is needed. The code generator will provide this wrapper to let the SOEnvO simulate the candidate algorithm at each step of the optimization process.

The candidate represents the decision-making unit in charge of elaborating the input data and deciding the next action to be executed by the CPS. As "next action" we mean a call to one (or more than one) of the services exposed by the Abstraction Library. The wrapper serves as an intermediate layer between the decision-making unit and the simulated CPS. Therefore, the wrapper defines how sensors are read and provide their data to the candidate. Moreover, it interprets the outputs of the candidate linking the selected action with its real implementation inside the Abstraction Library. A more detailed description of the candidate and the wrapper is presented in D6.5 (Sections 6.1.3 and 6.1.4).

While still not completely defined, a first list of inputs needed by the CG to generate the wrapper is foreseen:

- The target runtime environment that is executed inside the simulation engine (in the first release ROS will be supported).
- A description of inputs and outputs needed by the *candidate* algorithm. At the moment a possible format that is still under evaluation is SDF[5] with a specific CPSwarm extension, if needed.

[5] http://sdformat.org/

6 Conclusion

This deliverable describes the status of the CPSwarm Modelling Tool due to M18. Even if the presented features (CPS population design tool, concepts for modelling, code generation) have not yet been fully developed or integrated, they already show good collaboration and result among them and project's partners.

Acronyms

Acronym	Explanation
CG	Code Generator
CPDT	CPS population design tool
CPS	Cyber Physical System
FSM	Finite State Machine
ROS	Robot Operating System
SCXML	State Chart XML
SOEnvO	Simulation and Optimization Environment Orchestrator
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language

List of figures