D5.4 – FINAL CPSWARM MODELLING TOOL

Deliverable ID	**D5.4**
Deliverable Title	**Final CPSwarm Modelling Tool**
Work Package	**WP5 – CPSwarm Design Workbench**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**13-01-2020**
Status	**Final**
Lead Editor	**SOFTEAM**
Main Contributors	**Etienne Brosse (SOFTEAM), Gianluca Prato (LINKS), Kais CHAABOUNI (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	2019-02-07	Etienne Brosse (SOFTEAM)	Initial ToC
0.2	2019-11-22	Etienne Brosse (SOFTEAM)	First contribution
0.3	2019-12-20	Gianluca Prato (LINKS)	LINKS contribution
0.4	2019-12-22	Kais CHAABOUNI (SOFTEAM),	SOFTEAM contribution
0.5	2020-01-06	Etienne Brosse (SOFTEAM)	Polishing
1.0	2020-01-13	Etienne Brosse (SOFTEAM)	Final version after updates according to review comments

Internal Review History

Review Date	Reviewer	Summary of Comments
2020-01-10	Andreas Eckel (TTT)	Minor comments
2019-12-24	Angel Soriano (ROBOTNIK)	Minor comments

Table of Contents

1 Executive summary

This deliverable, namely "D5.4 – Final CPSwarm Modelling Tool", presents 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 of 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 D3.3 - Final System Architecture and Design Specification, delivered at M30 - 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.

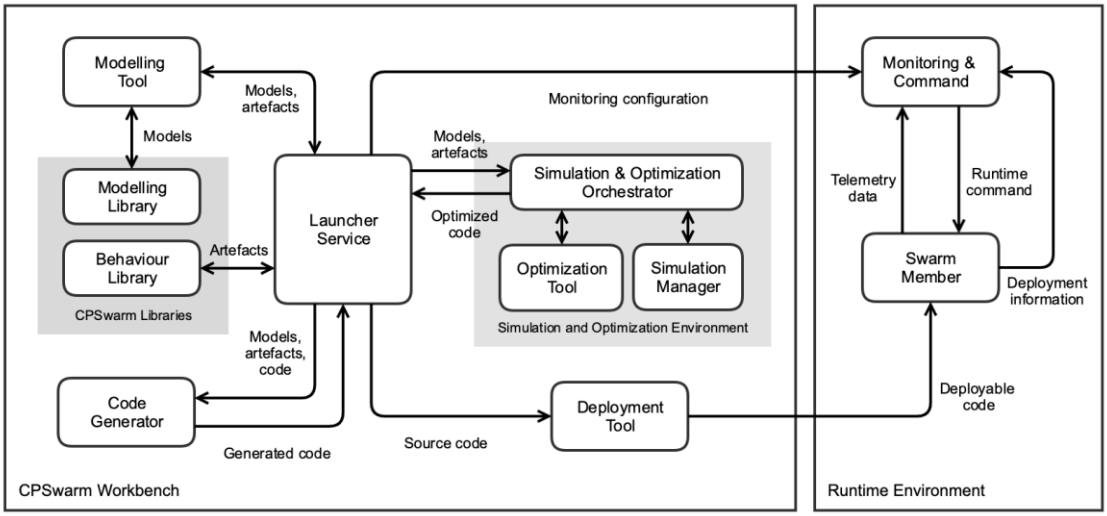

Figure 1: Final architecture design (see D3.3 for more information)

This "D5.4 - CPSwarm Modelling Tool" is a public deliverable focused on the Modelling Tool implementation on CPSwarm M35. It details the M36 status of Modelling Tool component and its implemented interfaces with related components (mainly the code generator and simulation optimization orchestrator).
SOFTEAM, 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 M36 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 Modelling Tool and its updates in state machine design. Section 4 describes the different wizards implemented in Modelling Tool. Finally, Section 5 focuses on the code generation out of the state machines provided by Modelio in Section 3.

2.3 Related documents

ID	Title	Reference	Version	Date
[D3.3]	Final System Architecture and Design Specification	D3.3	1.0	10/07/2019
[D4.3]	Final CPS Modelling Library	D4.3	1.0	31/12/2019
[D4.6]	Final Swarm Modelling Library	D4.6	1.0	30/11/2019
[D7.2]	Final CPSwarm Abstraction Library	D7.2	1.0	31/12/2019

3 CPSwarm Modelling

3.1 Overview

The CPSwarm Modelling Tool is built on top of Modelio open source modelling environment as previously described in Deliverable D5.2. CPSwarm modelling activity can be succinctly described as the creation and population of several diagrams or views. The following sections describe the main modelling concepts.

3.2 Swarm Composition Modelling

A Swarm is composed of one to many Swarm Member type. Each Swarm Member type may be instantiated from 1 to n time. To model this relation, UML[1] composition relation is used from the Swarm block to one or many Swarm Member blocks. The multiplicity at the end of the relation indicates the number of Swarm Member instances. Figure 6 depicts a Swarm composed of one unique Swarm Member.

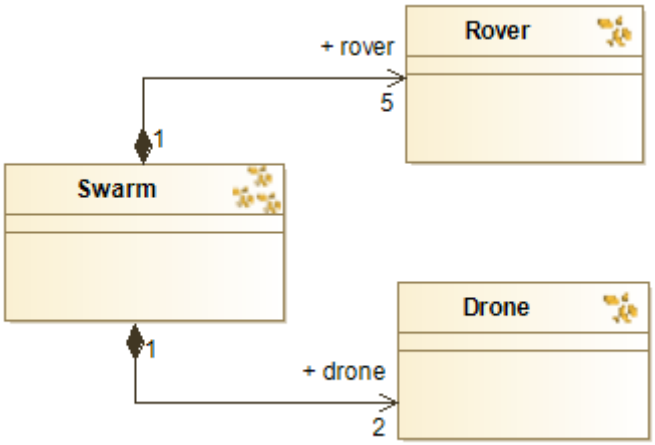

Figure 2: Swarm Composition Modelling Elements

3.3 Swarm Member Architecture Modelling

Another aspect of Swarm Modelling is the specification of each Swarm Member's internal architecture. This specification is made in two steps. In a first step, 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 SysML FlowPorts respectively named *fp1* and *fp2*. *Fp1* FlowPort expresses the fact that the component provides a Boolean value at contrary *fp2* FlowPort expresses the fact that the component requires a Boolean

[1] https://www.omg.org/spec/UML/About-UML/

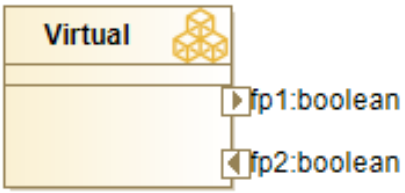

Figure 3: Simple Component example

The second step for modelling the internal architecture of a Swarm Member consists of instantiating each appropriate component and define the connection between them. In Figure 8, the previously predefined Component has been instanced twice and each port has been connected to model the data flow between the internal components.

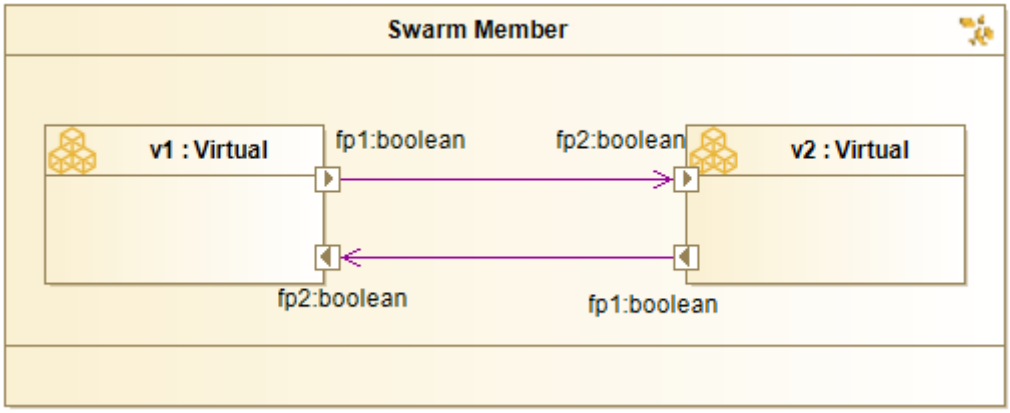

Figure 4: Swarm Member Architecture Example

3.4 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 the 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 Machines. The two transitions respectively connect the Initial state to the "State" state and the "State" state to the Final state.

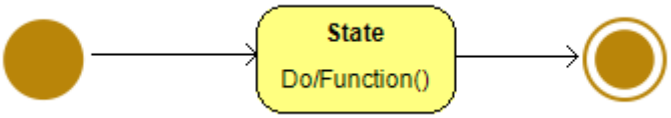

Figure 5: Simple Swarm Member Behavior

Of course, a real behavior will be more complex. Figure 10 for example represents two states – respectively named State1 and State2 – executed in parallel.

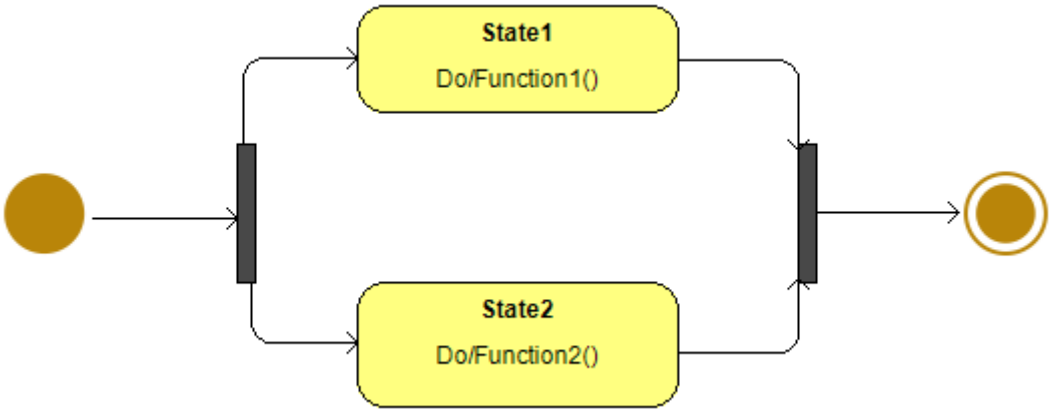

Figure 6: Swarm Member Behavior

To handle the complexity of these state machines, it is possible to extract part of them into 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.

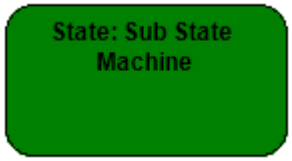

Figure 7: Hierarchical State

3.5 Swarm Modelling Library

As described in deliverable D3.3, the Swarm Modelling library is composed of a set of predefined.

- 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 the model with four possible actions respectively named Send, Pick, Place, and PickAndPlace.

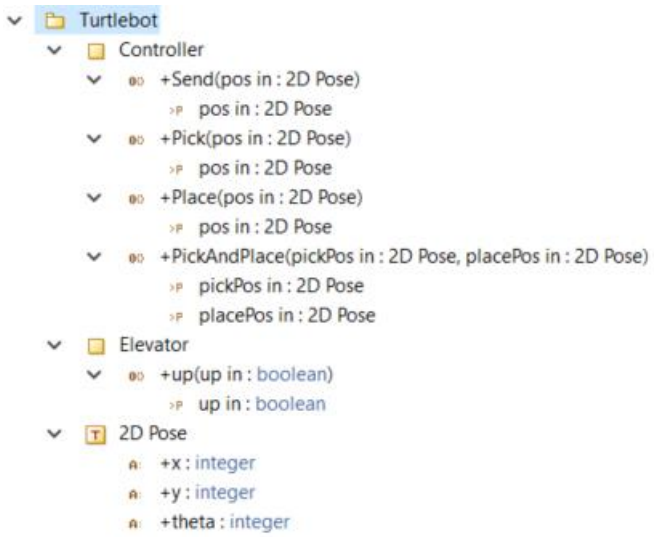

Figure 8: Part of the Modelling Library

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

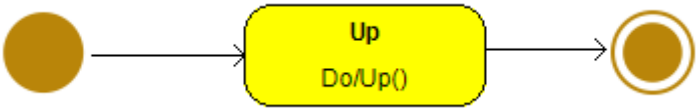

Figure 9: Simple reuse of the Modelling Library

4 CPSwarm Wizards

Modelling might be difficult task to carry out from scratch specially if you are not familiar with modeling tool or the modeling language. In this case, guidance is helpful. The main goal of CPSwarm wizards are to help the Modeler to easily create CPS swarm models.

4.1 Create a new CPSwarm model

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 10).

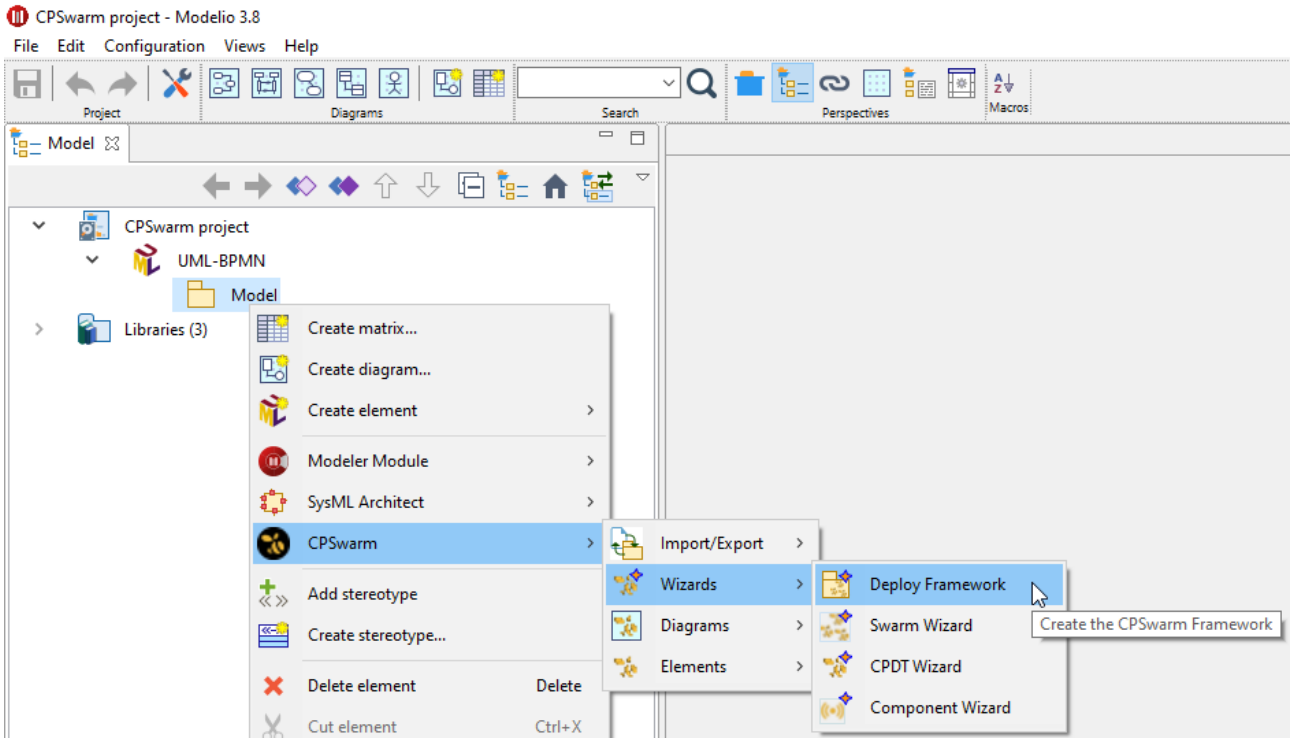

Figure 10: Creating a new swarm modelling

Figure 11 shows the result of the CPS swarm template generation.

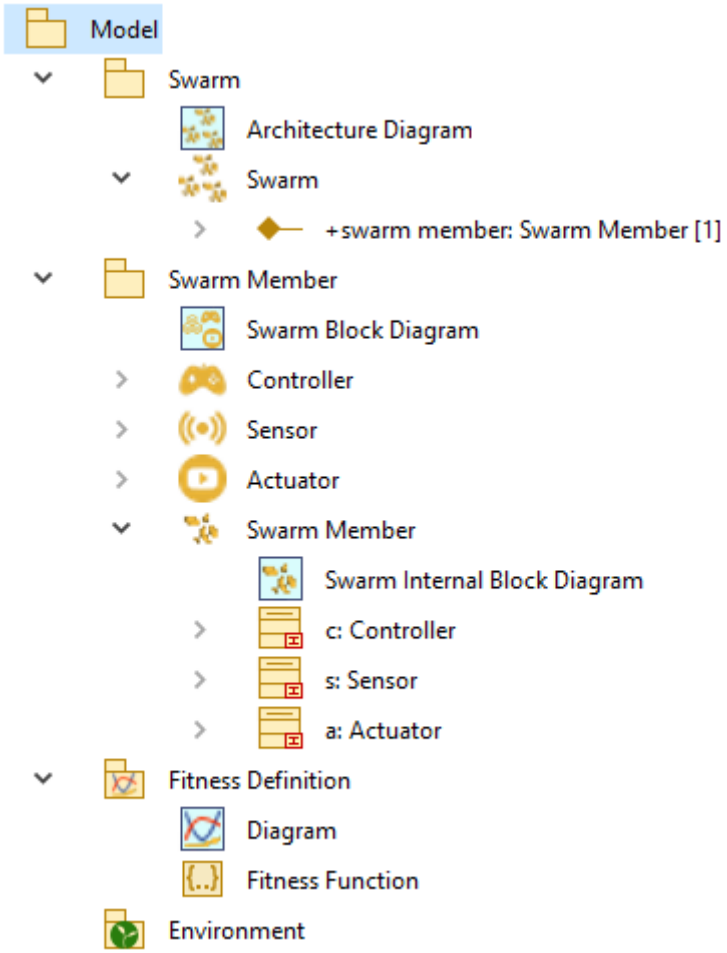

Figure 11: New swarm result

The swarm template generator produces a set of initial diagrams (as shown in Figure 12) that have been identified as necessary to completely model a CPS swarm.

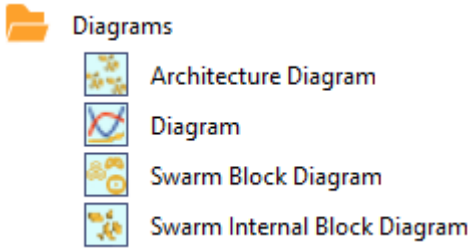

Figure 12: CPSwarm predefined diagrams

The CPSwarm modeler can modify the initial content following the needs of the specific case study he/she is modelling. For each diagram, the CPSwarm modeler will find, as depicted in Figure 13, the predefined selection of the modelling elements he/she can specifically use for that specific diagram context.

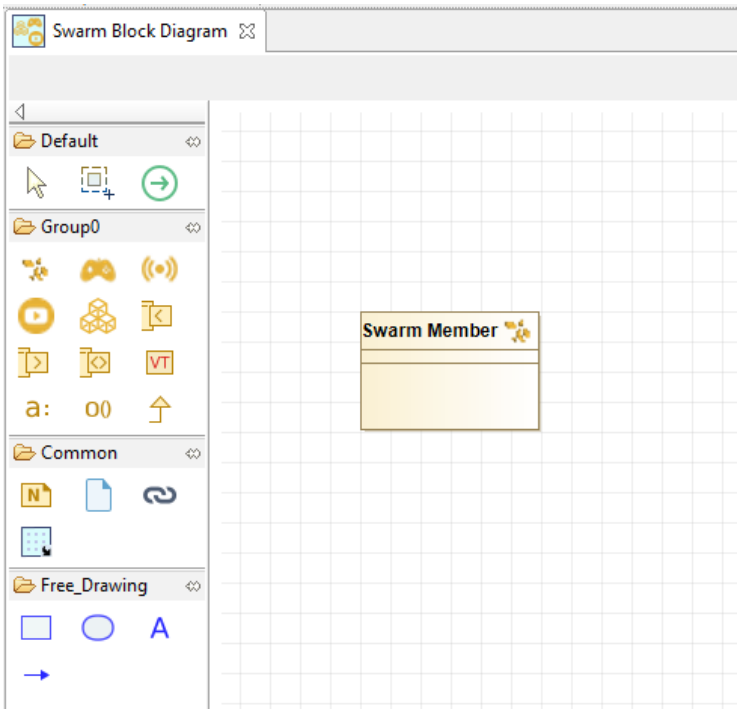

Figure 13: Palette of the CPSwarm Swarm Block Diagram

4.2 CPSwarm wizards

The CPSwarm wizards are used to define specific aspect of swarm modeling. The main idea is to provide a simple entry point, where an aspect of the swarm can be pre-configured by means of a wizard. This might include for example the type and number of CPSs to be included in the swarm.

Figure 14 gives a capture of the component creation wizard within the Modelling tool. The user can design the component by selecting specifying its name, a description, its type (Sensor, Actuator, Virtual or Controller), if it has a behaviour i.e. a FSM, a list of SysML FlowPort.

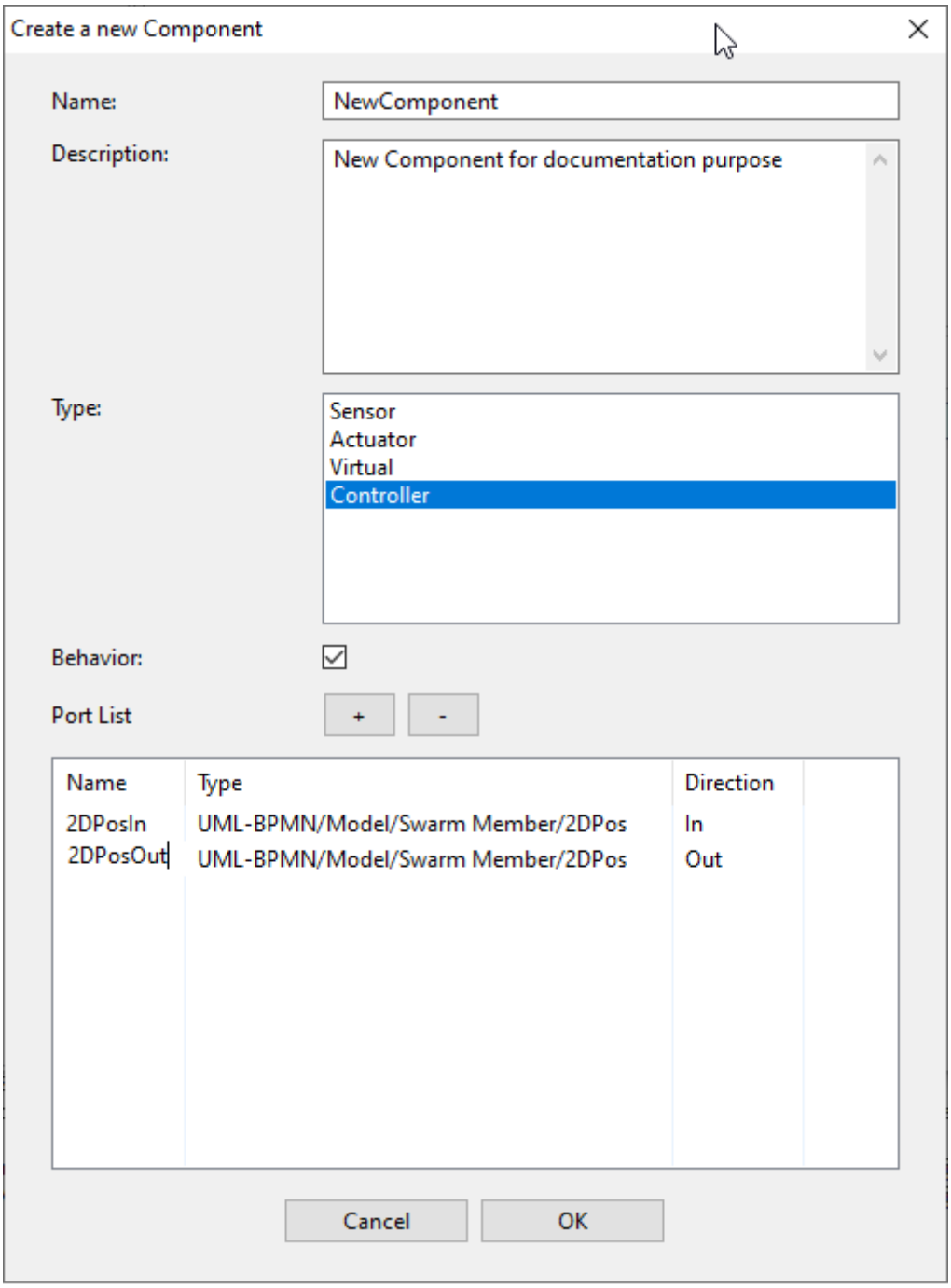

Figure 14: Component creation using dedicated wizard

Figure 15 depicts the result of component creation using the wizards previously shown.

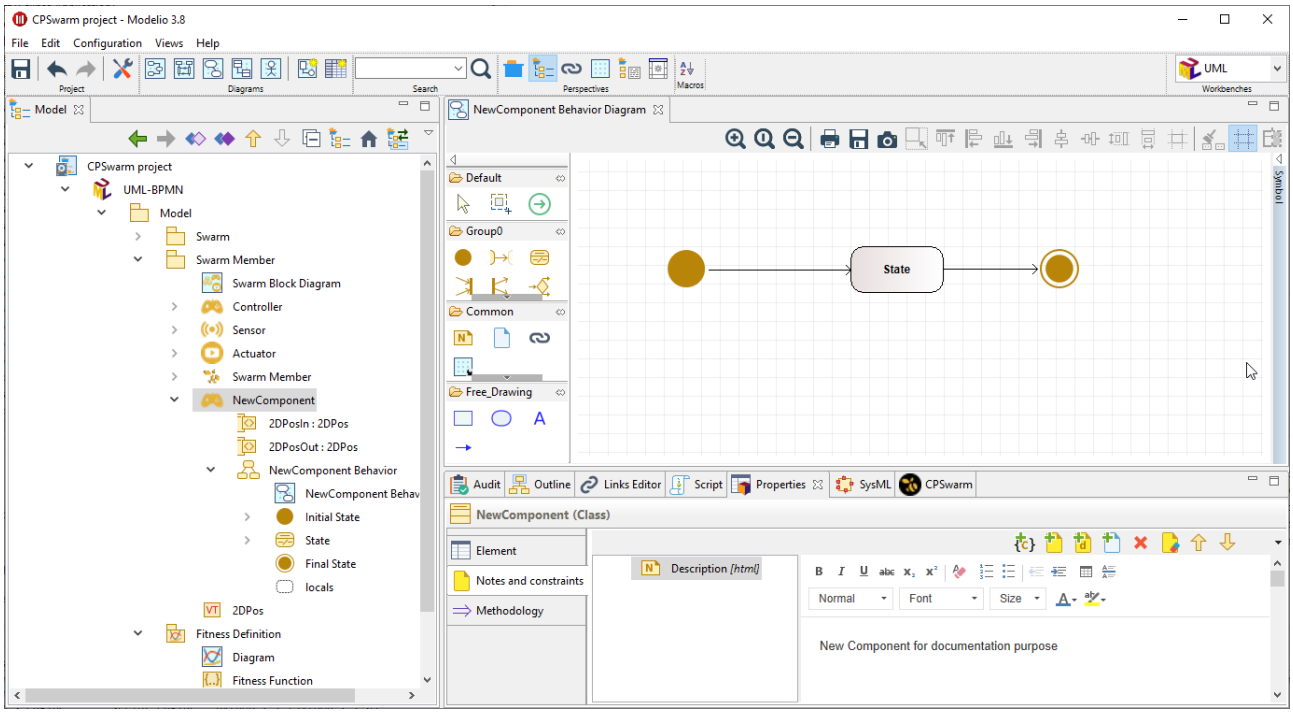

Figure 15: Result of the component creation

5 Code Generation for CPS Systems

One thing that distinguishes the software development for a CPS system from a common desktop software is the interaction with real world systems. An engineer needs to interact with sensors and actuators through the provided APIs, which often makes the implementation process difficult and labor-intensive [1]. To overcome these kinds of limitations, CPSwarm project promotes the exploitation of model-based methodologies for the design of CPS software. Hence, the goal of the CPSwarm workbench is to provide a framework where the definition of CPS swarm applications can be realized reducing the involvement with hardware and system architecture details.

The promise of modelling is to shift the focus from implementation to design. Models can be used as mechanisms to get a better understanding, but they can also be input for code generators [2]. By model-based automated code generation for robots, CPSwarm project means the process of automatic generation of compliable and verifiable code for robotic systems. Automated code generation is a challenging task in software engineering but brings with it some benefits:

- **Productivity**: code generators can be written once and be reused many times. Providing the specific inputs to the generator and invoke it is normally faster than writing the code manually. Code generation can significantly allow to save time.
- **Complexity hiding:** the complexity of application development can be moved to a higher level of abstraction. The input of a code generator is generally a high-level description of the code that is usually easier to analyze and validate.
- **Portability**: the same model can be used to generate code for different target language and platform just using a different generator.
- **Consistency**: a code generator can guarantee that his output will be always consistent with the expected result based on his defined code generation process. Furthermore, the uniformity of the code implementation considerably reduces the error rate.

The advantages of code generation are not for free and can be afflicted by some possible drawbacks:

- **Maintenance**: a code generator must be maintained or it can easily become outdated.
- **Complexity**: code generated automatically tend to be more complex and less optimized than code written by hand. Furthermore, the generated code can be less flexible as the number of use cases that the generator can support is limited.

A code generation process defines how information extracted from the models has: to be transformed into an executable code. This process depends on and is guided by the modelling language with its concepts, semantics and rules. To be effectively useful, the generation process should be as complete as possible, avoiding, whenever possible, the need of manual re-writing by the developer. This objective is often difficult to achieve but can be easier, if the code generator and the related modeling language used to provide his inputs, are designed and limited to fit a set of specific situations. For this reason, the consortium has decided to focus on the implementation of CPS behavior algorithms modelled as Hierarchical Finite State Machines (HFSM). Furthermore, state machines represent an attractive solution for robot behavior modeling due to various properties [3]:

1. HFSM can be used to design program execution in a transparent and reproducible way. This is particularly important for robot experiments, which need to be designed in a manner that allows different experimenters to obtain the same results under similar experimental conditions.
2. Even if the implementation depends on the target software platform, many programming languages have specific libraries for the definition of FSM (e.g. SMACH[2] for Python, SMACC[3] for C++).

[2] SMACH library
[3] SMACC – State Machine Asynchronous C++

3. Finite State Machines are easy and understandable also for non-developers. The abstraction provided by an FSM represents a good compromise (in relation to simplicity and descriptive power) compared to the usual hard-to-read description that a common real computer program usually has.

As already presented in D5.3, the current implementation of the CPSwarm Code Generator accepts as input a formal description of the state machine behavior in the SCXML[4] standard format. SCXML was selected among other possible languages (such as Amazon States Language[5] or RoboChart [4]) for his high flexibility and adaptability to different working context. In fact, in order to capture specific aspects related to the CPS domain, the language has been slightly adapted for code generator purposes. The detailed description of this extension will be provided in the following section.

CPSwarm Code Generator[6] aims at not to substitute developer work, but to give support during the development and ease the process of integrating and re-using external existing algorithm implementations. For this reason, in order to make the code generation process easier, a uniform support framework so called Behavior Libraries was defined as an intermediate level between the code generator output and the platform components on board of the CPSs. The Behavior Libraries are mainly composed by the CPSwarm Swarm Library (see D4.6) and the CPSwarm Abstraction Library (see D7.2) and constitute the basic building block used to compose an FSM algorithm. As can be observed in
Figure 16: Code generator bond
, the Code Generator is a bonding agent between the modeling phase realized into the Modeling Tool and the actual code that will be deployed and run on board of the CPS.

[4] https://www.w3.org/TR/scxml
[5] https://states-language.net/spec.html
[6] Open source code available on GitHub

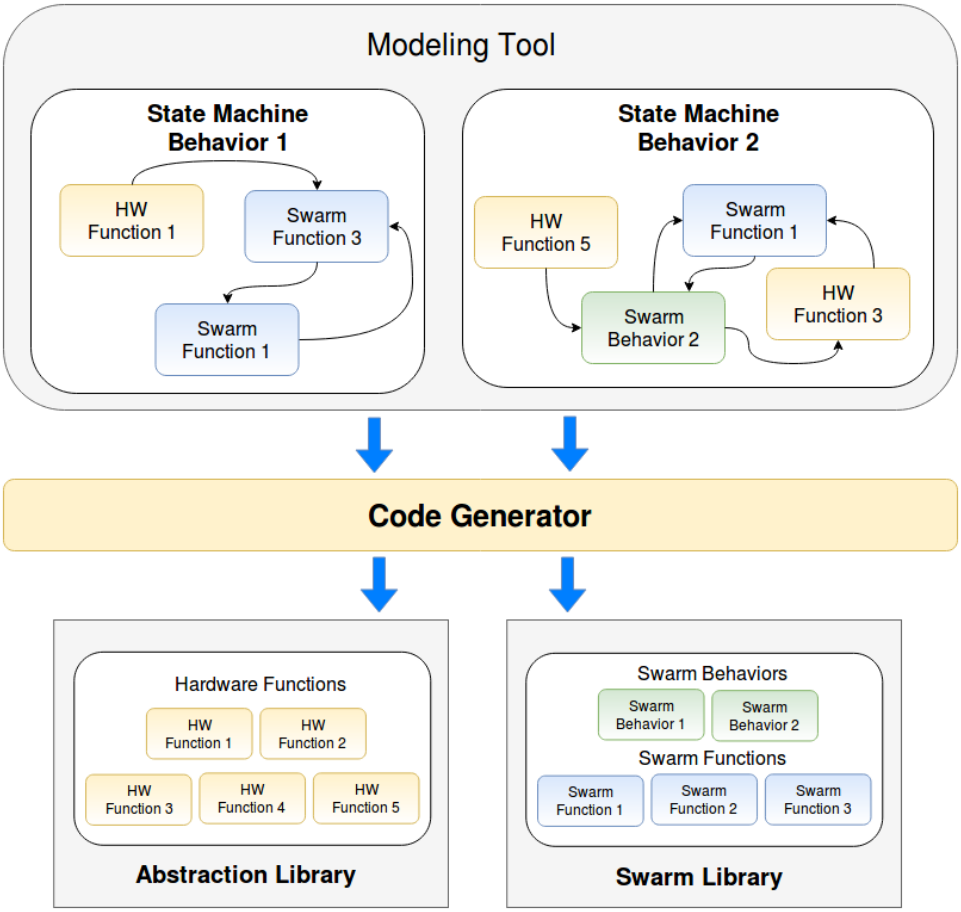

Figure 16: Code generator bond

With the current CPSwarm Code Generator implementation is possible to generate a state machine implementation that rely on the SMACH library[7], a Python-based project that let easily implement and execute state machine-designed algorithm. The choice has fallen to this library not only for his extreme simplicity and scalability, but also for his direct integration with ROS, the runtime environment supported by almost all of the CPS platforms that is used in the final use case scenarios.

In the second part of the CPSwarm project, a relevant amount of time was dedicated in the mapping of the concepts used at the modeling level with the software running on the CPS. The relevant result is exposed in the following sections.

[7] http://wiki.ros.org/smach

5.1 Connection between models and code libraries

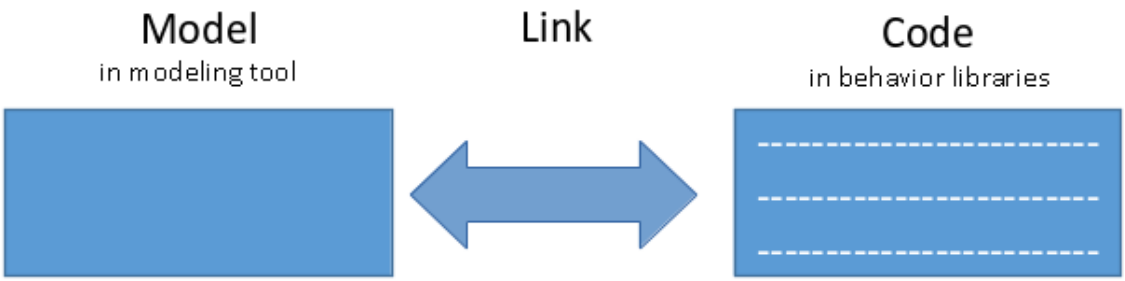

Figure 17: Link between model and code

In collaboration with tasks T4.3 and T7.1, a comprehensive analysis of how the models in the Modeling Tool and the code in the Behavior Libraries was realized.

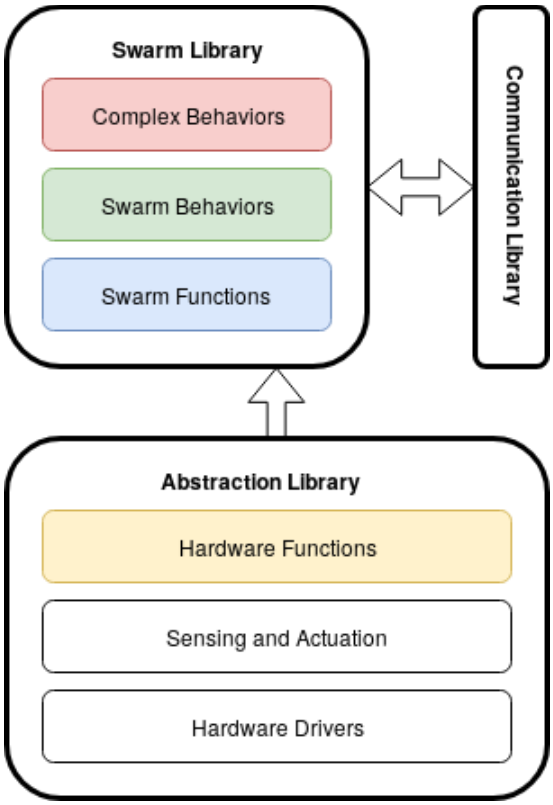

Figure 18: Behavior libraries architecture

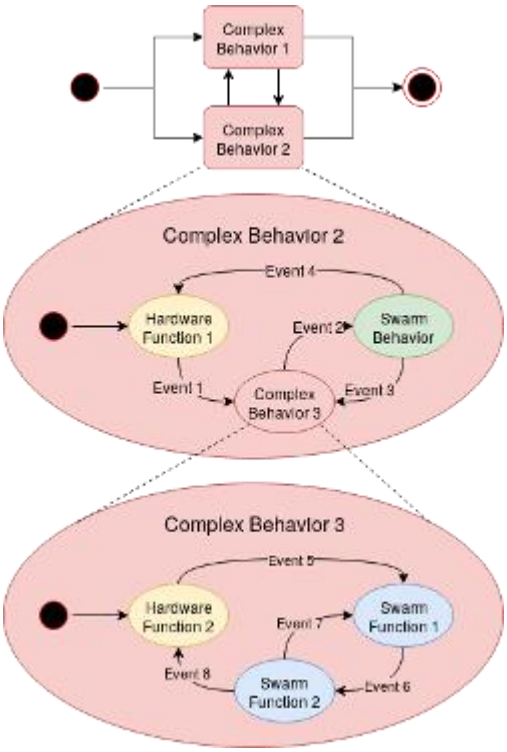

Figure 19: Multi-level behavior

The Behavior Libraries have a structure to differentiate between different types of behaviors:

- Swarm library: Contains swarm behaviors to be used as states by the complex behavior FSMs.
  - Swarm behavior: There can be different types of swarm algorithms. First, they can be handwritten, e.g., using biological inspiration. Second, they can be created automatically, e.g., using artificial evolution.
  - Swarm function: These are simple swarm functions unrelated to the CPS hardware. They can be used, e.g., to perform computations or to coordinate swarm members.
- Abstraction library: Contains hardware related behaviors to be used as states by the complex behavior FSMs.
  - Hardware functions: Routines that involve sensors or actuators.
  - Sensing and actuation: Provide sensor readings and drive actuators.
  - Hardware drivers: Drivers to control the hardware.

Actually, two relevant use cases have been identified.

5.1.1 Use case 1

This use case describes the workflow when the algorithms are already implemented. It requires at least that the states (i.e. swarm behaviors and abstraction/swarm functions) are implemented in the behavior libraries. Possibly, there are also models of the state machine and/or the states in the Modeling Tool. This use case is visualized in the figure below. It requires two interactions between the modeling tool and the behavior libraries:

1. Import the modeling tool into the abstraction functions using the Abstraction Description File (ADF) and the behaviors using the Algorithm Meta File (AMF) (step 1 in Figure 20). In case the states are already in the Modeling Tool, they will be updated.
2. Export the state machine to the behavior library using the SCXML (step 3 in Figure 20).

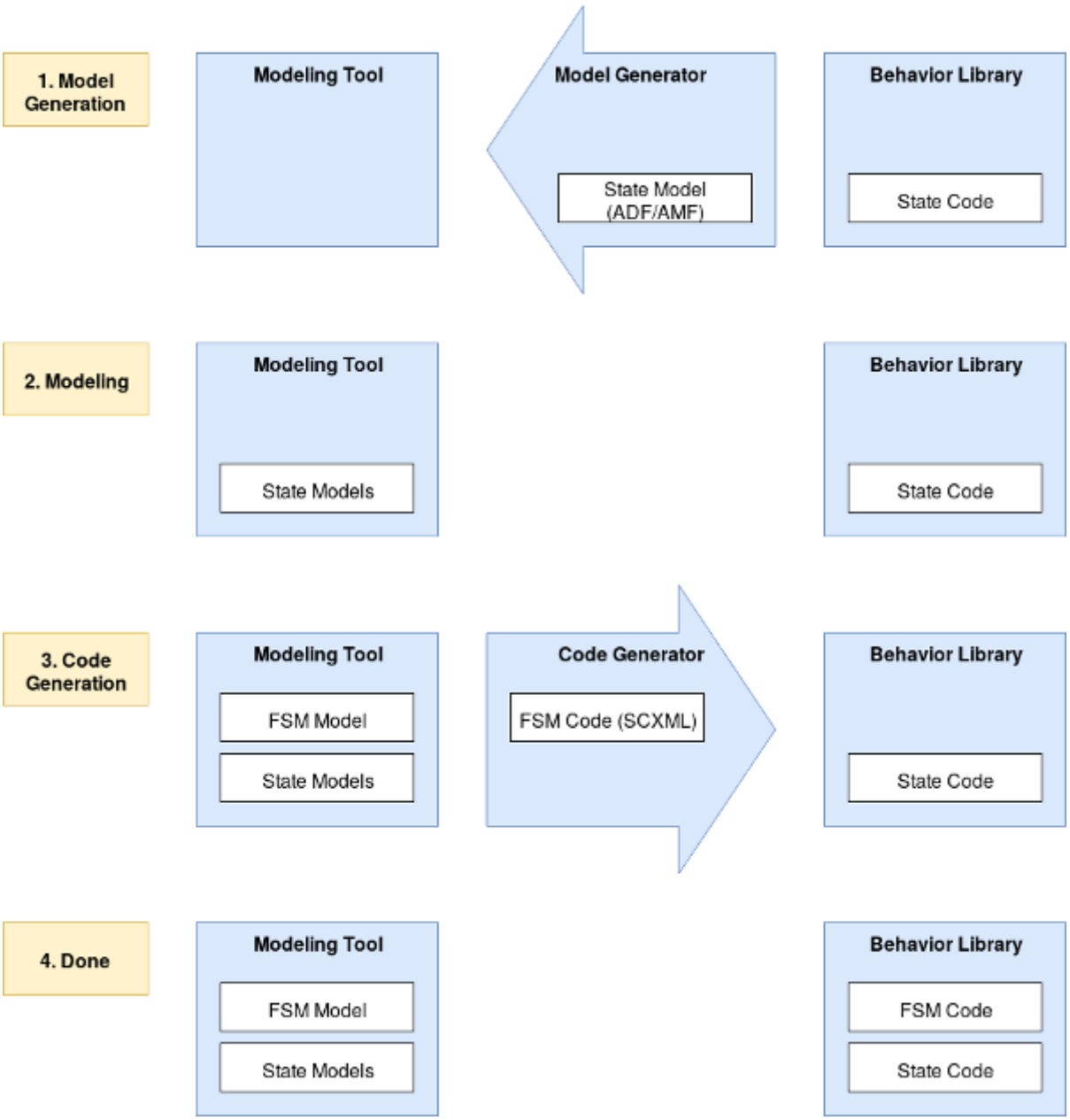

Figure 20: First use case

The state machine can be either created from scratch based on the states or it can exist already in the modeling tool (step 2 in Figure 18). In the latter case, it can be modified in the Modeling Tool.
Therefore, in this use case the consistency among models and code is maintained through specific data file:

- *Abstraction Description File* (ADF) and *Algorithm Meta File* (AMF): these are two json-based formats defined by the Consortium to collect a description of hardware and software on board of a robotic system. In our definition ROS was considered as the selected target environment. Specific details of the format are presented in D7.2.
- SCXML finite state machine description.

5.1.2 Use case 2

This use case describes the workflow for two different scenarios:

1. Manual implementation of the behavior algorithms by software developers.

2. Automatic behavior generation, e.g., using evolutionary optimization.

This use case requires at least that the functionalities (i.e. swarm behaviors and abstraction/swarm functions) are modeled in the Modeling Tool. This mainly consists of a high-level description of the inputs and outputs from such functionalities. This use case is visualized in Figure 19. It requires one interaction between the Modeling Tool and the behavior libraries:

1. Export (step 2 in the figure below) the state machine as SCXML and the "not implemented" states in an ADF containing the APIs of such functionalities. The ADF file, will be used to generate a template file from which starting the implementation of the new function. This template will contain an initial implementation of the new functionality applying common convention used for the Behavior Library implementation. This process will help the developer to easily integrate his code with the already implemented one present in the Behavior Library.

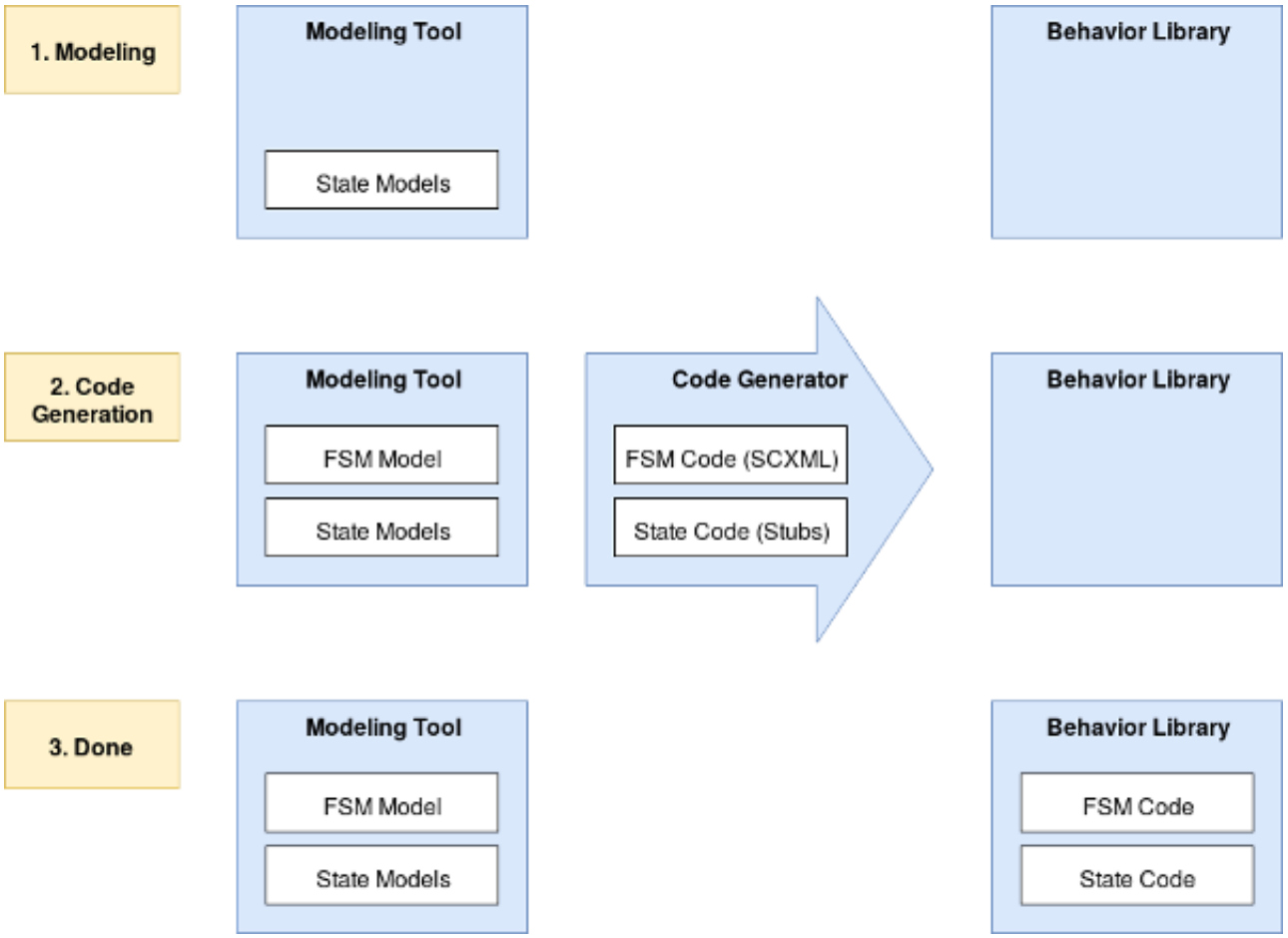

Figure 21: Second use case

The state machine can be either created from scratch based on the available functionalities/algorithms or it can be imported from an SCXML file description in the Modeling Tool (step 1 in the figure). In the latter case, it can be modified in the Modeling Tool.

5.2 SCXML adaptation for CPSwarm project

This section is not meant to be fully explanatory of the SCXML standard (for a complete description of the format, please, refer to the W3C official description[3]).

Instead, the main purpose of the following guide is to clarify how to use the SCXML format to describe Hierarchical State Machines accepted as input by the CPSwarm Code Generator.
The most basic state machine concepts are <*state*> and <*transition*>. Each state contains a set of transitions that define how it reacts to a specific event (further details on FSM modeling can be found in D4.6).
In the Figure 22, the system will transition to *"askForNewTask"* state when the event *"completed"* occurs but will transition to *"errorRoutine"* if event *"failed"* occurs.

```
<state id=doTask">
   <transition event="completed" target="askForNewTask"/>
   <transition event="failed" target="errorRoutine"/>
</state>
```

Figure 22: Simple SCXML transition example

5.2.1 Linking a state with an implemented software functionality

In the CPSwarm context, as already stated in the previous sections, each state of the state machine can be associated with a high-level functionality that can be selected from two different sources:

1. the Abstraction Library
2. the Swarm Library, which contains different types of swarm algorithms

SCXML format was adapted in order to properly describe this link. The Consortium decided to adapt 2 already existing tag used by the standard: <*invoke*> and <*datamodel*>. The former is used to identify the type interface that the associated functionality is exposing. This information is used by the Code Generator to correctly parse the content of the <*datamodel*> tag.

Considering the choose of ROS as target runtime platform, the extension has enriched the SCXML format with concepts related to ROS standard methodologies and the FSM implementation library so called SMACH ROS.

5.2.2 ROS Interfaces Description

At the current state of the CPSwarm project, most of the functionalities offered both by the Abstraction Library and the Swarm Library are exposing common ROS interfaces (also known as "paradigm"):

- ROS Service[8]: short running tasks, such as moving up and down an elevator or taking a photo from a camera, can be activated using services.
- ROS Action[7]: for long running and computational expensive operations (e.g. moving to a specific position or letting a drone to take off), the ROS actionlib package is usually preferred.

The information of which paradigm a selected functionality has implemented can be specified using the "*type*" attribute value inside the <*invoke*> tag:

- ROS_ACTION: if the linked functionality provides ROS action API.
- ROS_SERVICE: if the linked functionality provides ROS service API.

SMACH provides specific support to call services and actions from a State, respectively *ServiceState*[9] and *SimpleActionState*[10].

[8] http://wiki.ros.org/Services
[9] http://wiki.ros.org/smach/Tutorials/ServiceState
[10] http://wiki.ros.org/smach/Tutorials/SimpleActionState

Therefore, the SCXML file has to contain all the information needed to correctly instantiate both these classes using the right parameters. For this purpose, all API related information have been gathered inside the <*datamodel*> tag that can be associated to each specific state.

5.2.3 ROS Service

```
<state id="ServiceExample">
   <datamodel>
      <data id="invoke">
         <rosservice>
            <name>service_example</name>
            <srv>ServiceExample</srv>
            <request type="fixed">
               <param>"param_value"</param>
            </request>
            <response type="userdata">
               <var>variable_a</var>
            </response>
         </rosservice>
         <mappings>
            <x>state_userdata</x>
            <y>global_userdata</y>
         </mappings>
      </data>
   </datamodel>
   <invoke type="ROS_SERVICE" />
   .
   .
</state>
```

Figure 23: ROS Service example

As stated above, SMACH provides a state class that acts as a proxy to a ROS service (an example is depicted in Figure 23).
To correctly instantiate a Service State, the Code Generator needs the following data:

- service name
- service type
- service request generation policy (empty, fixed, userdata, callback)[11]
- service response generation policy (userdata, callback)
- mappings

The "mapping" tag is used to let data pass from one state to the following one[12].

[11] More details related to each policy can be found in the ROS wiki tutorial
[12] Check this page for further details

5.2.4 ROS Action

```
<state id="ActionExample">
   <datamodel>
      <data id="invoke">
         <rosaction>
            <name>action_example</name>
            <action>ActionExample</action>
            <goal type="callback"/>
            <result type="empty"/>
         </rosaction>
         <mappings>
            <x>state_userdata</x>
            <y>global_userdata</y>
         </mappings>
      </data>
   </datamodel>
   <invoke type="ROS_ACTION" />
   .
   .
</state>
```

Figure 24: ROS Action example

SMACH has specific support to call actions and provides a state class that acts as a proxy to an actionlib action as depicted in Figure 24.
To correctly instantiate a SimpleActionState, the Code Generator needs the following data:

- action name,
- action type,
- action goal generation policy (empty, fixed, userdata, callback),
- action result generation policy (userdata, callback),
- mappings.

```
<state id="Takeoff">
   <datamodel>
      <data id="invoke">
        <rosaction>
           <name>cmd/takeoff</name>
           <action>uav_mavros_takeoff/TakeOff</action>
           <goal type="object">
              <param>1.5</param>
           </goal>
           <result type="empty"/>
        </rosaction>
      </data>
   </datamodel>
   <invoke type="ROS_ACTION" />

   <transition event="succeeded" target="Coverage" />
</state>
```

Figure 25: State example from SAR scenario

In the example above (Figure 25), extracted from the Search&Rescue scenario, a specific state called "Takeoff" - part of the FSM used for the drones - is linked with a takeoff function exposing a ROS action as interface.

5.3 Skeleton function generation

In section 5.1 was presented the possibility to automatically generate the initial skeleton of a new functionality. In order to complete this task, the Code Generator should receive the following inputs:

- SCXML file containing the FSM description.
- Abstraction Description File (ADF) containing the new function API description.

In the SCXML, for each state that will have a skeleton to be generated, the <*datamodel*> has to contain a specific tag marked with an *id* value to "adf" containing the reference to the ADF section to be used to generate the skeleton.
For example, in the figure 26 is the description of a Takeoff function, which description is specified into an associated ADF (see appendix section 7.1) referenced by the name "uav_mavros_takeoff".

```
<state id="TakeOff">
   <datamodel>
      <data id="adf">
         <name>uav_mavros_takeoff</name>
      </data>
      <data id="invoke">
         <rosservice>
            <name>cmd/takeoff</name>
            <srv>uav_mavros_takeoff/TakeOff</srv>
            <request type="userdata">
               <var>req1</var>
               <var>req2</var>
            </request>
            <response type="callback" />
            <mappings>
               <map>
                  <x>state_a</x>
                  <y>userdata_a</y>
               </map>
               <map>
                  <x>state_b</x>
                  <y>userdata_b</y>
               </map>
            </mappings>
         </rosservice>
      </data>
   </datamodel>
   <invoke type="ROS_SERVICE" />
   <transition event="succeeded" target="IdleThreads" />
</state>
```

Figure 26: Datamodel example to generate ROS action skeleton

6 Conclusion

This deliverable describes the status of the CPSwarm Modelling Tool at the end of CPSwarm project. It presents the main features (concepts for modelling, diagrams, wizards, code generation) developed and integrated during project lifetime and a complete user manual can be found at http://forge.modelio.org/projects/cpswarm. This result also highlights the existing strong collaboration between tool providers and end user within the project.

Appendix

Abstraction Description File for UAV


```
{
 "runtime-env": "ROS",
 "functions": [
  {
   "name": "uav_mavros_takeoff",
   "description": "Send takeoff command",
   "category": "abstraction-lib",
   "param_list": [
    {
     "class": "number",
     "name": "pos_tolerance",
     "value": 0.1
    },
    {
     "class": "number",
     "name": "frequency",
     "value": 10.0
    },
    {
     "class": "number",
     "name": "stabilize_time",
     "value": 5
    },
    {
     "class": "number",
     "name": "takeoff_steps",
     "value": 1
    },
    {
     "class": "number",
     "name": "initial_yaw",
     "value": 90
    }
   ],
   "api": {
    "inputs": [
     {
      "topic": "mavros/state",
      "msg": {
       "class": "mavros_msgs/State",
       "fields": [
        {
         "class": "stds_msgs/Header",
         "name": "header",
         "description": "ros header"
        },
        {
         "class": "bool",
         "name": "connected"
        },
        {
```

```
          "class": "bool",
          "name": "armed"
         },
         {
          "class": "bool",
          "name": "guided"
         },
         {
          "class": "bool",
          "name": "manual_input"
         },
         {
          "class": "string",
          "name": "mode"
         },
         {
          "class": "uint8",
          "name": "system_status"
         }
        ]
       }
      },
      {
       "topic": "pos_provider",
       "msg": {
        "class": "geometry_msgs/PoseStamped",
        "fields": [
         {
          "class": "stds_msgs/Header",
          "name": "header",
          "description": "ros header"
         },
         {
          "class": "geometry_msgs/Pose",
          "name": "pose"
         }
        ]
       }
      }
     ],
     "outputs": [
      {
       "topic": "pos_controller/goal_position",
       "msg": {
        "class": "geometry_msgs/PoseStamped",
        "fields": [
         {
          "class": "stds_msgs/Header",
          "name": "header",
          "description": "ros header"
         },
         {
          "class": "geometry_msgs/Pose",
          "name": "pose"
```

```
          }
         ]
        }
       }
      ],
      "comm_model": {
       "paradigm": "rosaction",
       "definition": {
        "name": "cmd/takeoff",
        "class": "TakeOff",
        "goal": {
         "fields": [
          {
           "class": "float64",
           "name": "altitude"
          }
         ]
        }
       }
      }
     }
    },
    {
     "name": "uav_mavros_land",
     "description": "Send land command",
     "category": "abstraction-lib",
     "api": {
      "comm_model": {
       "paradigm": "rosservice",
       "definition": {
        "name": "cmd/land",
        "class": "Empty"
       }
      }
     }
    },
    {
     "name": "auction_action",
     "description": "Assign a task in a specific position to another CPS",
     "category": "swarm-lib",
     "api": {
      "inputs": [
       {
        "topic": "bridge/events/cps_selection",
        "msg": {
         "class": "cpswarm_msgs/TaskAllocationEvent",
         "fields": [
          {
           "class": "stds_msgs/Header",
           "name": "header",
           "description": "ros header"
          },
          {
           "class": "swarmros/EventHeader",
```

```
          "name": "swarmio",
          "description": "cpswarm swarmio swarmros header"
         },
         {
          "class": "int32",
          "name": "task_id",
          "description": "id of the task"
         },
         {
          "class": "float64",
          "name": "bid",
          "description": "bid of the cps for the task (inverse of cost)"
         }
        ]
       }
      }
    ],
    "outputs": [
     {
      "topic": "cps_selected",
      "msg": {
       "class": "cpswarm_msgs/TaskAllocatedEvent",
       "fields": [
        {
         "class": "stds_msgs/Header",
         "name": "header",
         "description": "ros header"
        },
        {
         "class": "swarmros/EventHeader",
         "name": "swarmio",
         "description": "cpswarm swarmio swarmros header"
        },
        {
         "class": "int32",
         "name": "task_id",
         "description": "id of the task"
        },
        {
         "class": "string",
         "name": "cps_id",
         "description": "uuid of the cps to which the task has been allocated"
        }
       ]
      }
     }
    ],
    "comm_model": {
     "paradigm": "rosaction",
     "definition": {
      "name": "cmd/task_allocation_auction",
      "class": "cpswarm_msgs/TaskAllocation",
      "goal": {
       "fields": [
```

```
          {
            "class": "string",
            "name": "auctioneer",
            "description": "UUID of the CPS performing the task allocation"
          },
          {
            "class": "uint32",
            "name": "task_id",
            "description": "ID of the task"
          },
          {
            "class": "geometry_msgs/PoseStamped",
            "name": "task_pose",
            "description": "Local position of the task"
          }
        ]
      },
      "result": {
        "fields": [
          {
            "class": "string",
            "name": "winner",
            "description": "UUID of the CPS to which the task is allocated"
          },
          {
            "class": "uint32",
            "name": "task_id",
            "description": "ID of the task"
          },
          {
            "class": "geometry_msgs/PoseStamped",
            "name": "task_pose",
            "description": "Local position of the task"
          }
        ]
      }
    }
  }
 }
}
```

Acronyms

Acronym	Explanation
CG	Code Generator
CPS	Cyber Physical System
(H)FSM	(Hierarchical) Finite State Machine
ROS	Robot Operating System
SCXML	State Chart XML
SysML	System Modeling Language
SOEnvO	Simulation and Optimization Environment Orchestrator
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language

List of Figures

References

[1] Ramtin Raji Kermani, Model-based Design, Simulation and Automatic Code Generation For Embedded Systems and Robotic Applications. Master's thesis, Arizona State University, 2013.
[2] Tolvanen, J.-P, Making model-based code generation work, Embedded Systems, 2004.
[3] Mitrevski A., Plöger P.G. (2019) Reusable Specification of State Machines for Rapid Robot Functionality Prototyping. In: Chalup S., Niemueller T., Suthakorn J., Williams MA. (eds) RoboCup 2019: Robot World Cup XXIII. RoboCup 2019. Lecture Notes in Computer Science, vol 11531. Springer, Cham.
[4] Miyazawa, A., Ribeiro, P., Li, W. et al. Softw Syst Model (2019) 18: 3097. https://doi.org/10.1007/s10270-018-00710-z