D4.6 – FINAL SWARM MODELING LIBRARY

Deliverable ID	**D4.6**
Deliverable Title	**Final swarm modeling library**
Work Package	**WP4 – Models and algorithms for CPS Swarms**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**03-12-2019**
Status	**Final**
Lead Editor	**Melanie Schranz (LAKE)**
Main Contributors	**Melanie Schranz, Micha Sende (LAKE), Gianluca Prato (LINKS), Angel Soriano (ROBOTNIK), Arthur Pitman (UNI-KLU), Etienne Brosse (SOFTEAM)**

Published by the CSPwarm 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	2019-07-03	Melanie Schranz (LAKE)	First Draft of TOC
0.2	2019-09-10	Micha Rappaport (LAKE)	Add Concept of Behavior Library
0.3	2019-09-10	Melanie Schranz (LAKE)	Add Automotive Use Case,
0.4	2019-11-11	Melanie Schranz (LAKE), Arthur Pitman (UNI-KLU), Angel Soriano (ROBOTNIK), Gianluca Prato (LINKS)	Update Section 4
1.0	2019-12-03	Melanie Schranz (LAKE)	Integrate reviews and finalize document

Internal Review History

Review Date	Reviewer	Summary of Comments
2019-11-12	Ákos Milánkovich (SLAB)	Minor corrections and improvements
2019-12-02	Wilfried Elmenreich (UNI-KLU)	Suggestions and minor corrections

1 Executive summary

This document, namely "D4.6 Final swarm modeling library", is a deliverable of the CPSwarm project, funded by the European Comission's Directorate-General for Research and Innovation (DG RTD), under its Horizon 2020 Research and innovation program (H2020), reporting the final results of the activities carried out by WP4 – Models and algorithms for CPS Swarms, Task 4.3. The main objective of the CPSwarm project is to develop a workbench that aims to fully design, develop, validate and deploy engineered swarm solutions. More specifically, the project focuses on modeling of swarms of CPSs, implementing and optimizing the corresponding swarm intelligence algorithms, driven by WP4.

Deliverable D4.6 is the final document in the WP4 deliverables and thus, builds on previous ones in this series (D4.4 – Initial swarm modeling library and D4.5 – Updated swarm modeling library). In this version we describe the final structure of the behavior library that consists of a hierarchical formalization including Swarm Functions, Swarm Behaviors and Complex Behaviors. Furthermore, we describe updates for the use cases "Search and Rescue" and "Logistics" in terms of state-machine design. The use case "Automotive" will be described in more detail: the state-machine design for the use case and the algorithmic selection for individual states.

Table of Contents

2 Introduction

D4.6 – "Final swarm modeling library" is a public document defining the publicly available swarm models and swarm algorithms found till CPSwarm M34.

LAKE, as deliverable leader, initially drafted the document, which has subsequently been extended by all partners' contributions. Deliverable D4.6 is the final document in the WP4 deliverables and thus, builds on previous ones in this series (D4.4 – Initial swarm modeling library and D4.5 – Updated swarm modeling library). In this version we describe the final structure of the behavior library and all updates for the individual use cases in terms of state machine design and algorithms.

2.1 Document organization

The remainder of this deliverable is organized as follows:
Section 3 describes the final behavior library structure, state types and events. Section 4 describes the individual use cases (logistics, automotive, search & rescue) with the final state machines. Section 5 draws the conclusion.

2.2 Related documents

ID	Title	Reference	Version	Date
[D4.4]	Initial Swarm Modeling Library	D4.4	1.0	M10
[D4.5]	Updated Swarm Modeling Library	D4.5	1.0	M22
[D2.2]	Final Vision Scenarios and Use Case Definition	D2.2	1.0	M16
[D7.1]	Initial CPSwarm Abstraction Library	D7.1	1.0	M18
[D7.2]	Final CPSwarm Abstraction Library	D7.2	1.0	M32
[D8.4]	Final Swarm Logistics Demonstration	D8.4	1.0	M36
[D8.6]	Final Automotive Demonstration	D8.6	1.0	M36

3 Behavior Library

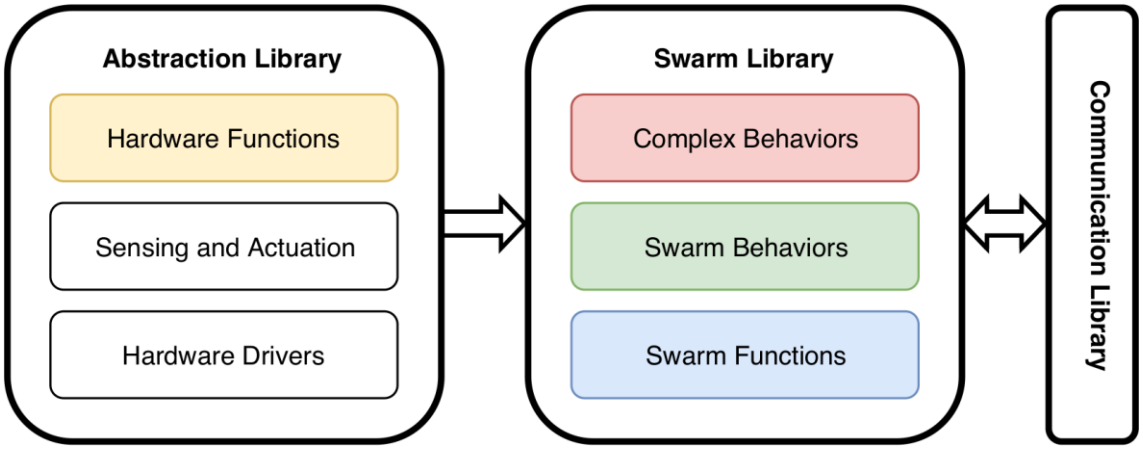


Figure 1: The behaviour library structure.

The behavior library contains all software models of the CPSs that allows them to perform complex swarm behaviors. It is a library of modular, reusable behaviors that are hierarchically structured and assembled in finite state machines (FSMs). Layering the behaviors in a hierarchy with different degrees of abstraction allows a separation of concerns. It has the benefit that often recurring behaviors and functionalities need to be defined only once. This avoids redundancies in design and implementation. These behaviors are therefore defined and implemented only once but can be used in different FSMs. They can be adapted to serve as starting point when designing new behaviors. This significantly reduces the effort to develop new CPS applications by letting developers reuse existing solutions and focus on application-specific problems. We propose the library structure shown in Figure 1 where the colors represent the different behaviors types that will be explained in more detail in the following.

These libraries are structured according to the level of hardware abstraction. The Swarm Library works independent of the underlying hardware. It provides the complex behaviors modeled as FSMs together with the swarm behaviors and swarm functions. It makes use of a communication library that provides an interface for communication between CPSs. The Abstraction Library abstracts away the hardware specifics. It provides functions that are related to the hardware based on hardware specific drivers and functionalities to access the sensors and actuators. More details on the Abstraction Library can be found in Deliverable D7.1 and D7.2.

3.1 State Machines

A mission for a swarm of CPSs requires typically many different behaviors to be executed by the CPSs to complete the different tasks of the mission. These individual behaviors can be simpler to describe and implement, and are regarded to be atomic during modeling. Combining these simple behaviors into more complex behaviors allows to achieve complex missions. A commonly used approach for this are FSMs where the states correspond to simple behaviors and the FSM describes a complex behavior. FSMs are well suited for behavior modeling because (i) they allow a visual representation that is easily understood by the human modeler, (ii) they can be formally described to allow automatic code generation, and (iii) allow a modular design which enables the reuse of behaviors. Each CPS can thus execute a FSM while always being in a defined state which can vary between CPSs. Hence, even in a homogeneous swarm of CPSs, complex swarm configurations can emerge where CPSs take on different roles based on the interactions between them.

3.1.1 State Types

The behavior state machine model is based on the UML behavior state machines. Specifically, the simple behaviors are modeled by simple states and the complex behaviors are modeled using composite or submachine states. Composite states allow a state to be modeled by another state machine whereas submachine states allow to encapsulate generic state machines that can be reused within more than one state.

In the context of CPS swarm behaviors, we propose four different behaviors types (the color codes refer to the ones used in Figure 1):

- Complex behaviors (red): Behaviors that are defined by a state machine of simple behaviors, e.g., SAR.
- Swarm behaviors (green): Simple behaviors that execute a specific swarm algorithm exhibiting an emergent swarm behavior, e.g., aggregation.
- Swarm functions (blue): Simple behaviors that execute a single function including the interaction between CPSs, e.g., task allocation.
- Hardware functions (yellow): Simple behaviors that execute a single function including hardware interaction, e.g., moving to a given location.

The behaviors are formalized by a unique name, a short description of the behavior, the behavior type, and the inputs and outputs of the behavior.

3.1.2 Hierarchy

To allow a clean and structured design of the behavior state machines, we propose to use hierarchically nested states defined in the UML standard. The organization as a hierarchy H using a set of levels $L_i \in H$allows to consider different levels of detail at different hierarchy levels. An example with two levels $H = \{L_1, L_2\}$is shown in Figure 2 where each color represents a different behavior type. In this example, level L_1 represents the highest level of the hierarchy, and considers parallel processing. Some behaviors need to be executed in parallel such as navigating while performing collision avoidance. Each process performs a complex behavior in parallel that is necessary for accomplishing the mission. Next, each complex behavior is assembled by using FSMs in the second level L_2 of the hierarchy. Therein, each state corresponds to a simple behavior such as swarm behaviors, swarm functions, or hardware functions, where transitions are triggered by dedicated events. The number of levels is dependent on the design strategy of the modeler and allows to model different levels of abstraction.

3.1.3 Events

The transitions between behavior states are triggered by events that can either originate locally, e.g., from sensor readings or behavior rules or remotely, e.g., from communication between CPSs, or from a command and control station. Exchanging events between CPSs enables the coordination of the swarm behaviors. Events are processed locally and autonomously by each CPS as defined in the behavior FSMs. This allows heterogeneous CPSs to influence each other's behavior changes by exchanging events. In complex behaviors, events can either trigger a state change of the sub FSM or force the complex behavior itself to terminate and thereby also terminate the currently running sub behavior. Events are formalized by a unique ID, a timestamp,

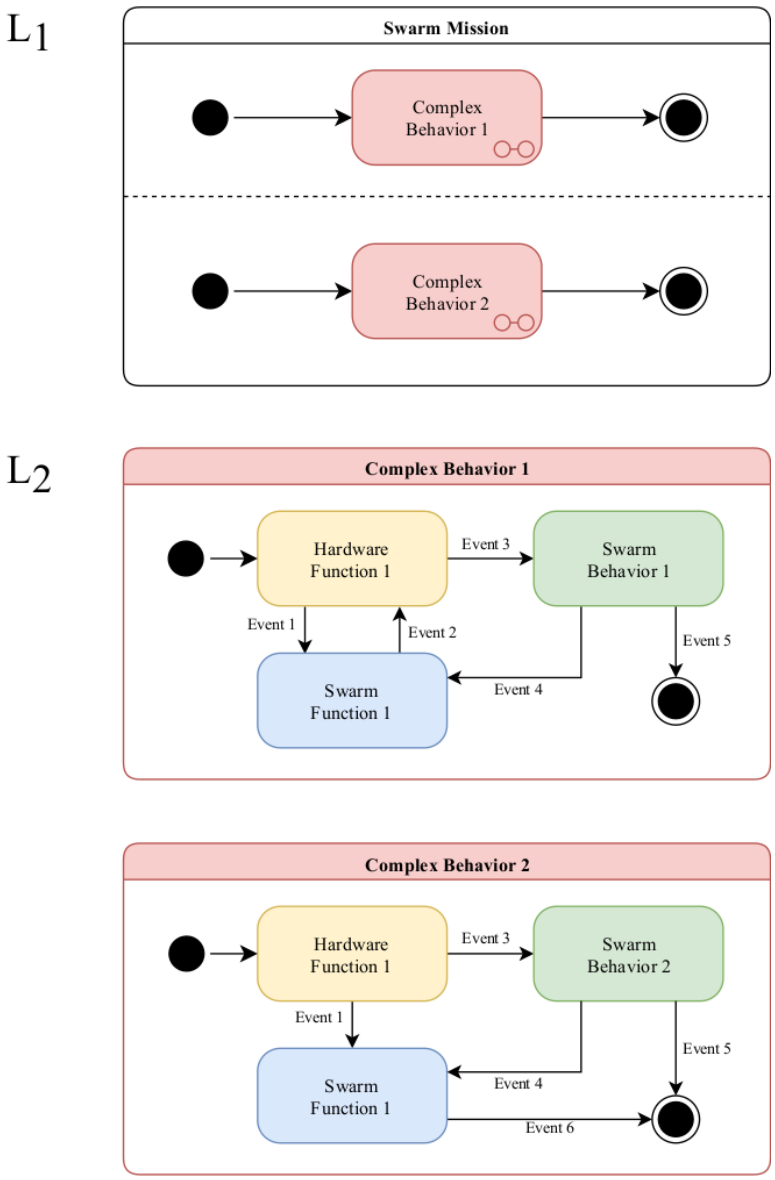


Figure 2: The behavior model hierarchy.

and a unique ID of the sender. Furthermore, events can have data associated with them in form of typed key-value pairs. This data is passed to the behavior using special inputs that are bound to the data in the event in order to supply constant input values to the behavior while the state is active. This binding is done on a per-transition basis, where an association between the data of the incoming event and the special inputs of the behavior can be made. Default values for the data can be defined in order to supply values for data missing in the event. When an event is triggered, the data of the outgoing event is handled similarly to how incoming

events are treated. The outputs taken from the behavior are bound to the data of the outgoing event. This binding is also done on a per-transition basis. Constant values can also be supplied for any of the data.

3.2 Swarm Library

The Swarm Library contains the swarm behaviors executed by the individual CPSs leading to the global swarm behavior. They are platform independent and thus can be reused among different types of CPSs. The swarm library is structured into three sub libraries in accordance with the previously introduced swarm behaviors. First, the Complex Behaviors library contains the FSM that model the complex, high-level mission behaviors such as SAR. They are defined as UML composite or submachine states. Second, the Swarm Behaviors library contains individual swarm behaviors that exhibit an emergent behavior. Typically, such swarm behaviors are hand crafted based on biological inspiration or generated automatically, e.g., using evolutionary optimization. Examples are flocking, phototaxis, or collective transport. They are defined as UML simple states to be used in the complex behavior FSMs. Third, the Swarm Functions library contain simple swarm related tasks. These are tasks that do not lead to an emergent behavior but rather are used to enable the functioning of the swarm behaviors. Examples are exchange of position information, task allocation, or computing the average velocity of the swarm. They are defined as UML simple states to be used in the complex behavior FSMs.

4 Use Cases

This section describes the updates on the related use cases of their state machines.

4.1 Logistics

The Logistics use case envisages a scenario where two classes of robots, scouts and workers, assist in moving boxes in a warehouse. The scout robots, equipped with a QR-code reading camera, rove around the warehouse space searching for boxes. Once a box is located, the scout notifies all workers robots of its location. Idle workers robots bid for the job of transporting the box to a specified location based on their current location, i.e. the distance to the box, and their remaining battery level. The selected workers moves to the box, autonomously navigating around obstacles, lifts it using its elevator mechanism, moves to the destination, sets the box down and returns to an idle state. The entire scenario in its final version is described in "D8.4 – Final Swarm Logistics Demonstration" (M36).

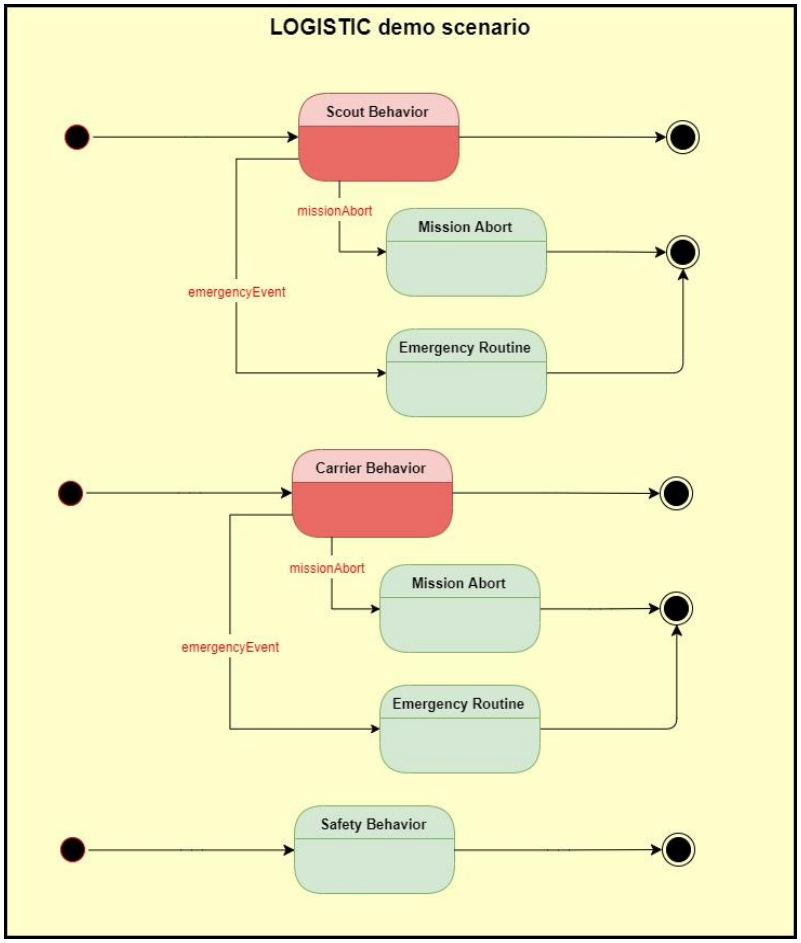


Figure 3 1st Level State Machine

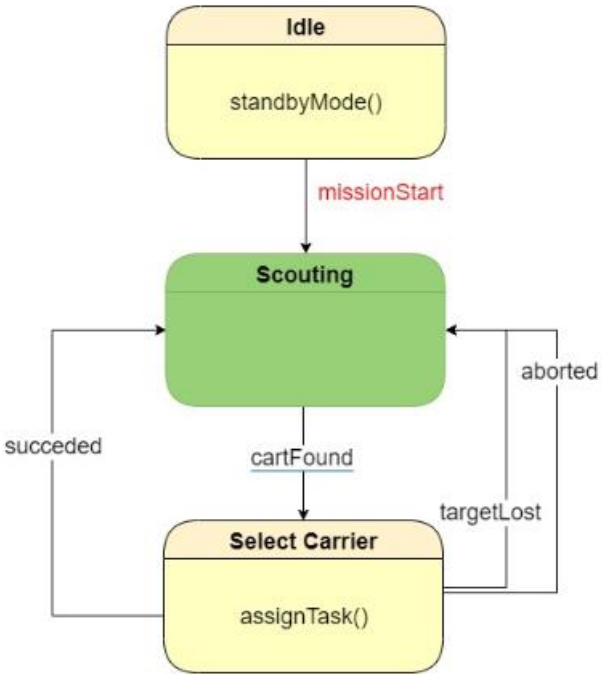


Figure 5 Scouts

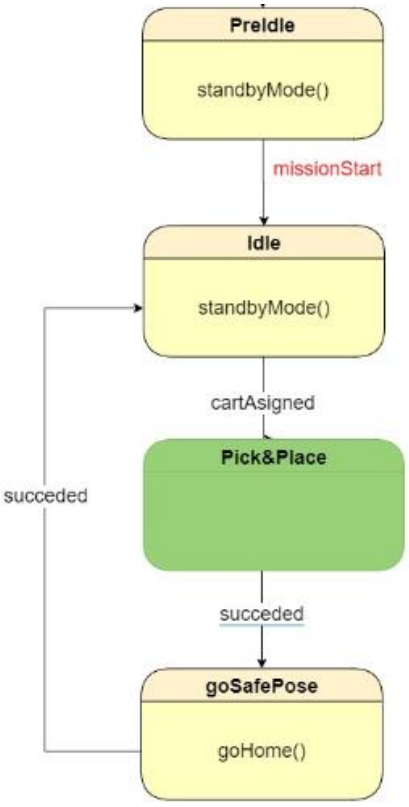
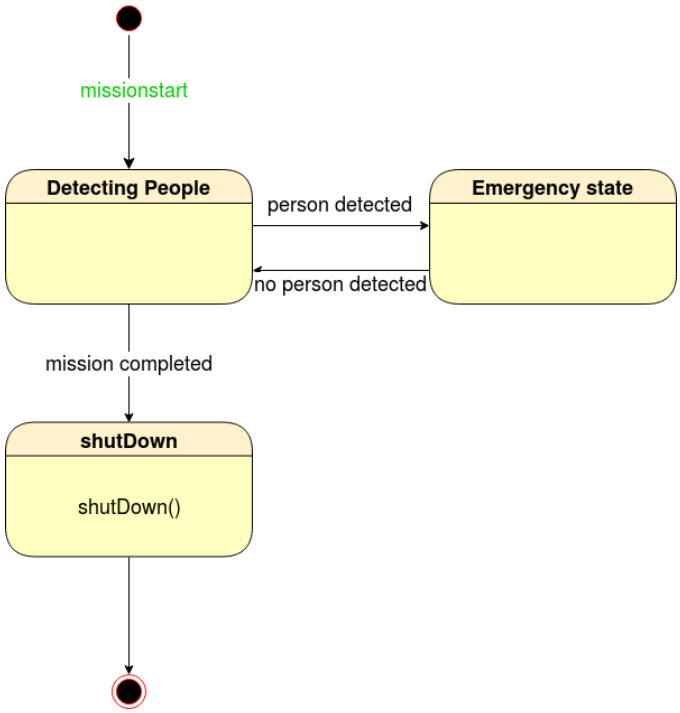


Figure 4 Workers

Related to this use case it is important to highlight the addition of the human in-the-loop feature by the implementation of a human presence alarm that will be given through the image processing of an external camera. The details of the hardware are also present in D8.4.



4.1.1 Optimization of Scouting Behaviour

In Figure 4, several different behaviors may be used as implementations for scouting during the discovery of boxes. To demonstrate the optimization components in the CPSwarm Workbench, a random walk algorithm is provided. While following this behaviour, the Scout robot picks a random direction and "walks" until it encounters a box, the edge of the operating space or a distance threshold has been exceeded. It then picks a new random direction and continues walking as before. As the distance parameter affects the effectiveness of the Scout coverage of operating space, it may be optimized for the specifics of the scenario. Essentially the optimization tool varies the parameter and assesses its performance by applying an objective function to a large number of simulation runs thus estimating the parameter's fitness. Within the CPSwarm Workbench this is implemented by a series of interactions between the Simulation and Optimisation Orchestrator (SOO), the Optimisation Tool (OT) and several Simulation Managers (SMs). As illustrated in Figure 6, Following an initial setup phase where the components are configured correctly, the SOO instructs the OT to start an optimisation. Using simulations run by individual SMs, the OT tests different parameter values and attempts to pick an optimum value, which it then returns to the SOO and in turn, to the user. This optimised behaviour may then be deployed to real hardware. The individual components are loosely coupled using a central XMPP server.

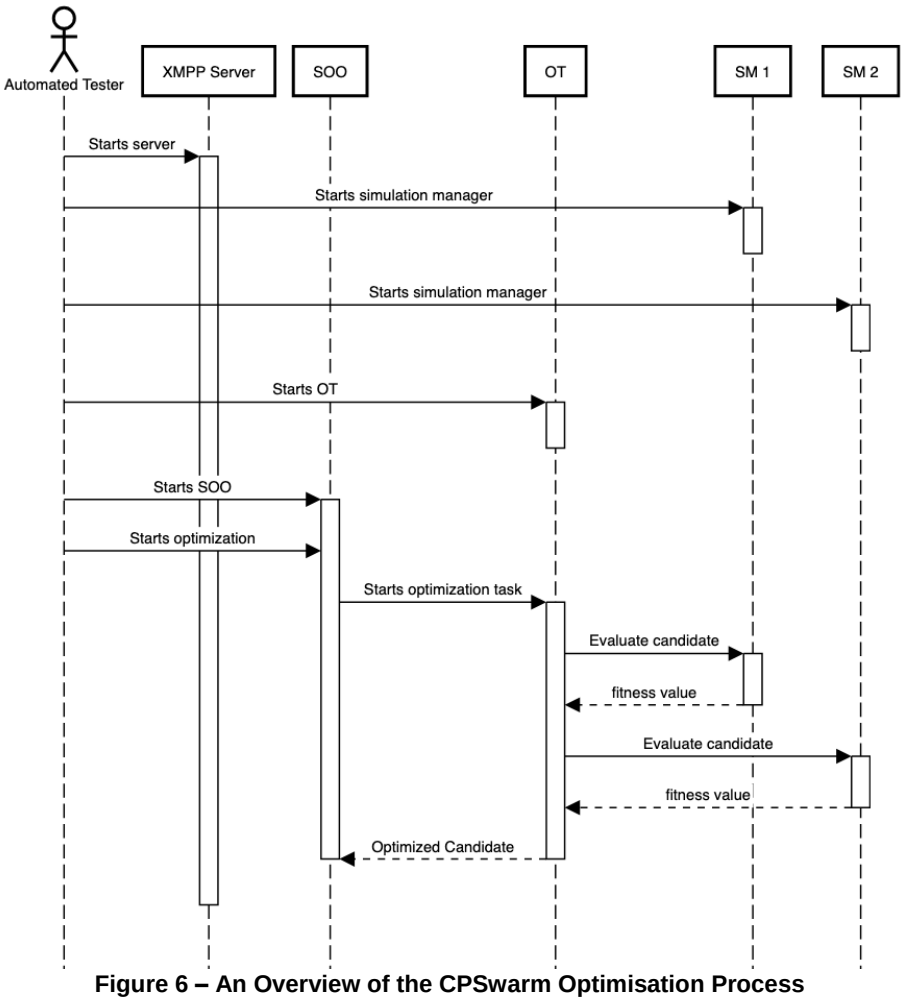


Figure 6 – An Overview of the CPSwarm Optimisation Process

4.2 Automotive

In summary, the automotive use case can be described as follows (find all details in D8.6 - Final Automotive Demonstration): The aim is to form platoons of connected freight vehicles. The leading vehicle prescribes the actions and decisions (e.g., navigation, decision on take-over maneuvers, sequencing maneuvers, lane change) for the follow-up vehicle(s) that will make use of the leading vehicle's actions. The follow-up vehicle will need full autonomous driving capability and environmental awareness. However, they will follow the leading vehicle in a preset distance even when they have to make decisions, e.g., lane change maneuver on their own due to an emergency situation, or leaving the platoon if the route to the destination deviates). The follow-up vehicles will also take over full control in case the lane change needs to be interrupted for the complete swarm due to other traffic prohibiting to change lanes.

For the platooning algorithms in the automotive use case, we defined following state machine (Figure 7).

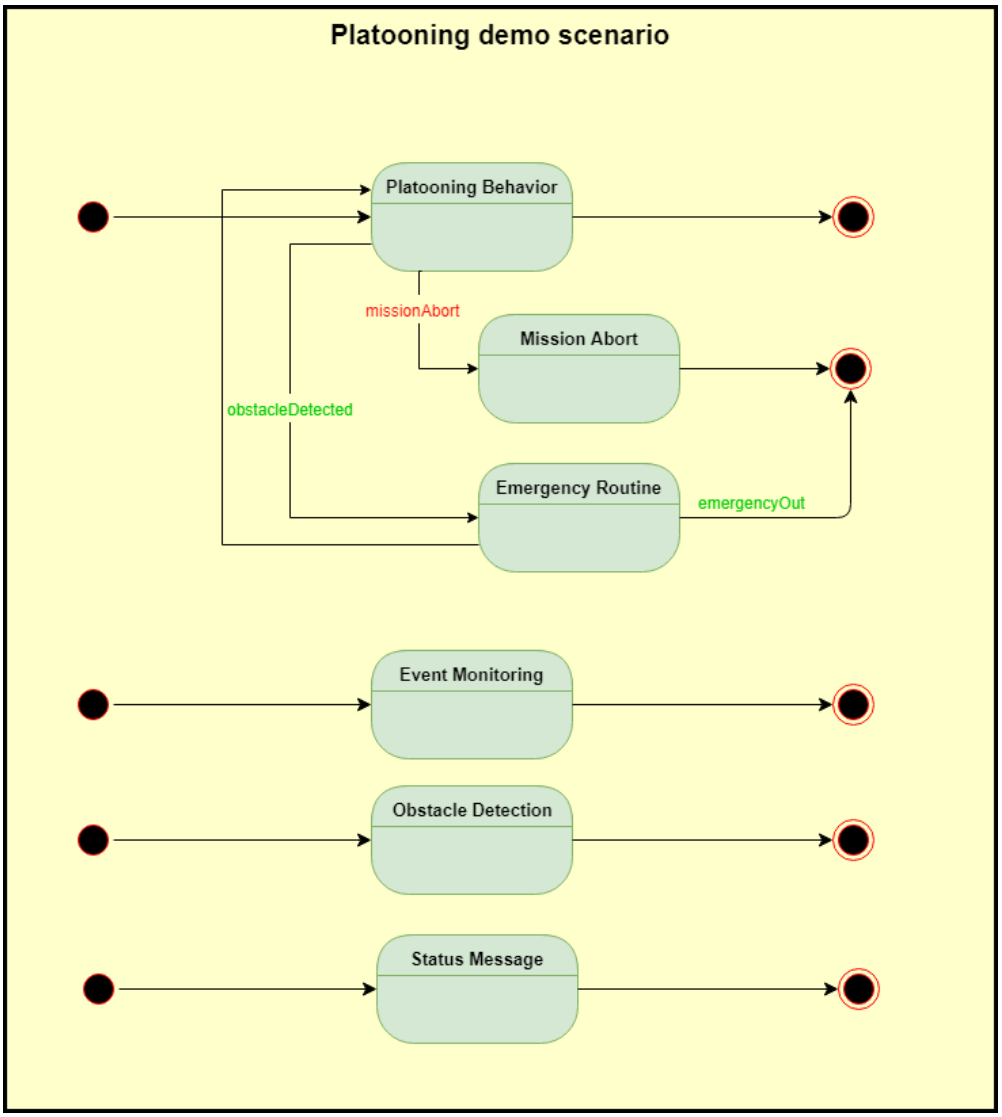


Figure 7 Automotive Use Case Swarm Mission Level 1

The second level of the state machine can be seen in Figure 8.

For the algorithmic part we consider directed graphs only and denote them with

- set of edges E
- set of vertices V
- For each $e \in E$: corresponding edge weight is c_e

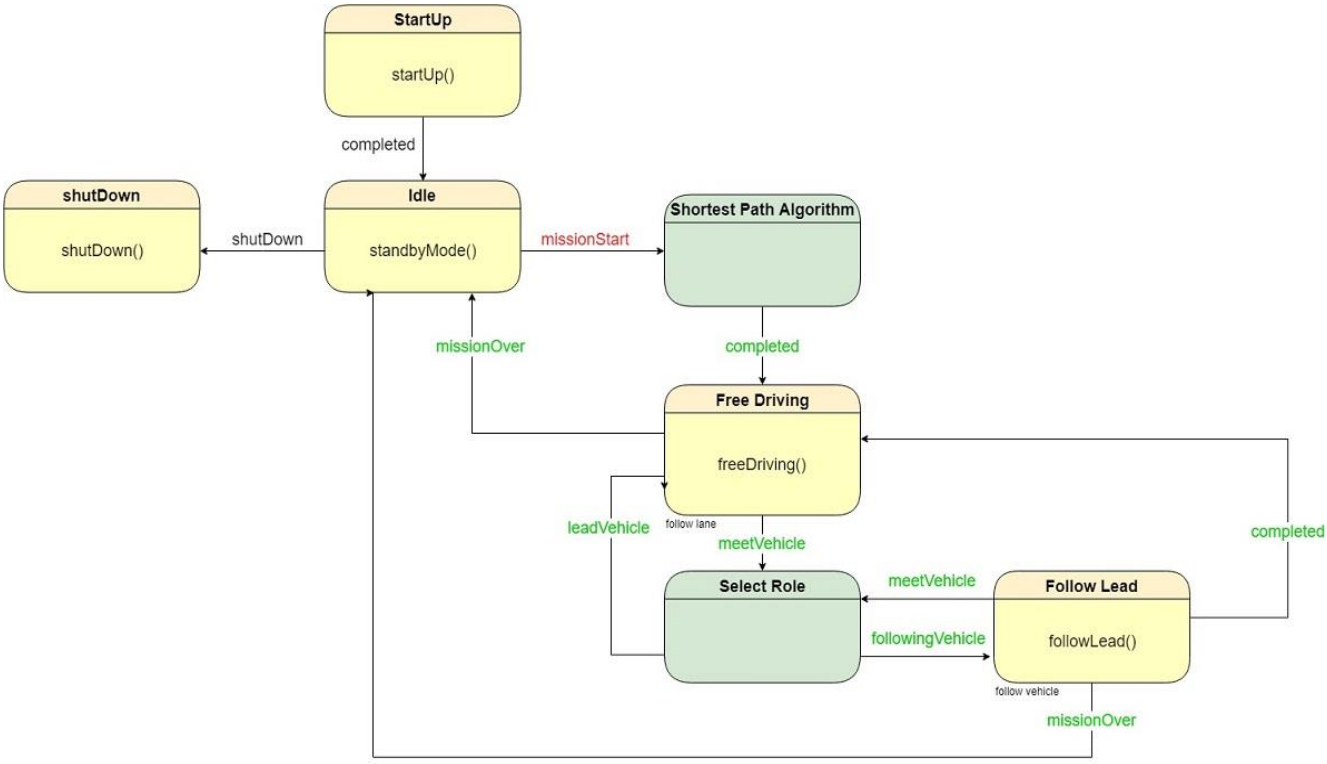


Figure 8 Automotive Use Cases Complex Behavior 1

4.2.1 Shortest Path Algorithm

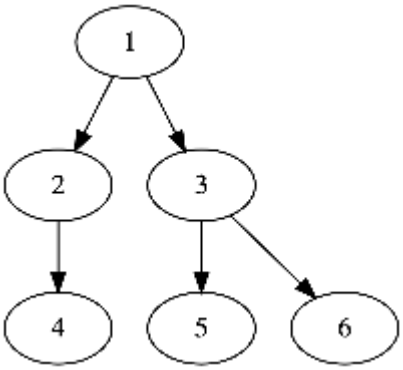
The shortest path algorithm is an algorithm to find a path between two vertices to minimize the sum of the weights of the edges. This algorithm is a crucial and time-relevant part of the behavior implementation. Therefore different candidate algorithms are to be investigated, together with their advandages, disadvantages and computational complexity O.

4.2.1.1 Breadth-First Search

Breadth first[1] traversal visits all shallower nodes before visiting deeper nodes.

[1] Skiena, S. S. (1998). The algorithm design manual: Text (Vol. 1). Springer Science & Business Media.

Example - Consider the following tree



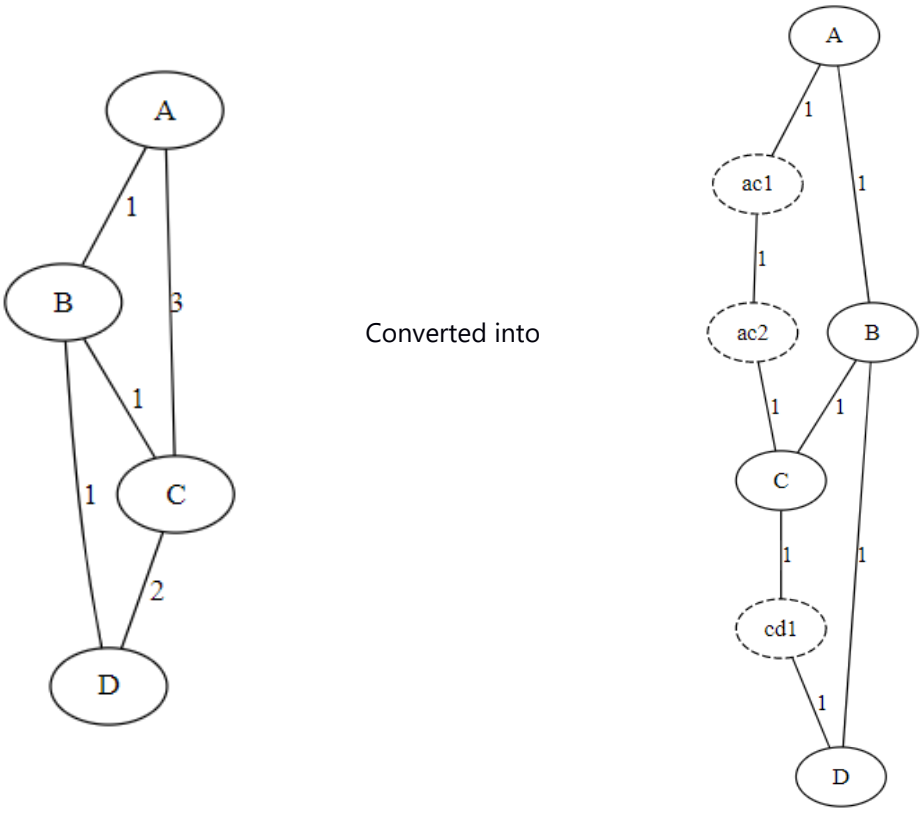
Apply breadth-first traversal to it would visit nodes in the order: 1, 2, 3, 4, 5, 6 using following pseudo code

```
1   function BreadthFirstTree(headNode) {
2     Q := empty queue of nodes
3     add headNode onto Q
4     while Q is not empty:
5       n := node on front of queue
6       remove node on front of queue
7       visit n
8       for each child c of n:
9         add c to Q
10      end for
11    end while
12  }
```

In terms of unweighted graphs, the algorithm may be applied to graphs by maintaining a table of nodes that have already been visited.

```
1   function BreadthFirstGraph(headNode) {
2     Q := empty queue of nodes
3     V := empty set of nodes
4     add headNode onto Q
5     while Q is not empty:
6       n := node on front of queue
7       remove node on front of queue
8       visit n
9       for each linked node l of n:
10        if l in not in V:
11          add l to Q
12          add l to V
13        end if
14      end for
15    end while
16  }
```

Weighted graphs may be supported by adding dummy nodes as demonstrated in the following example:



Complexity

- Time complexity

  O(|V|+|E|), where V is the number of vertices, E the number of edges. Note: each vertex is enqueued and dequeued once, each edge is scanned once.

- Space complexity

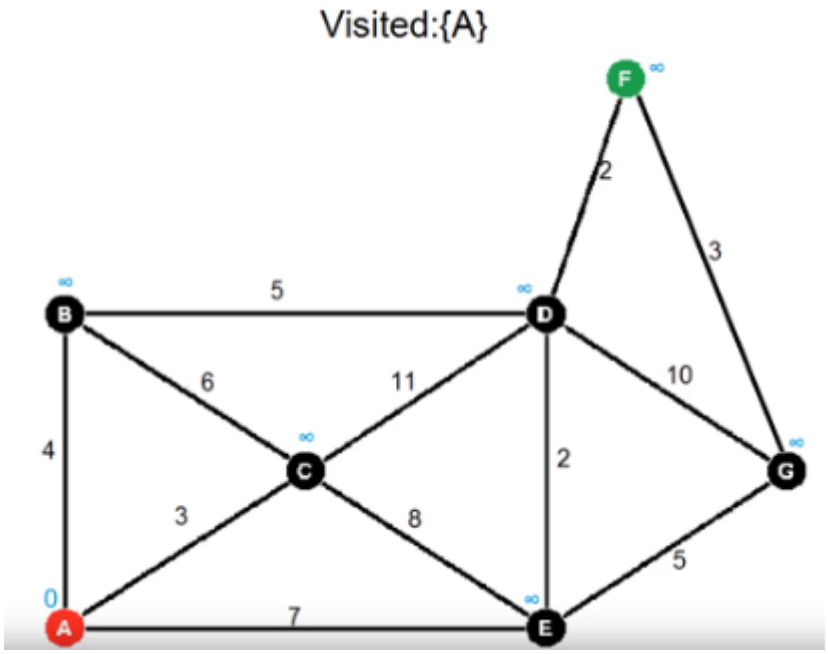
  O(|V|)
  Up to |V| vertices may have to be enqueued

4.2.1.2 Dijkstra

The Dijkstra's Algorithm (also called Uniform Cost Search algorithm) has the goal to find the least cost path from the initial node to a target node. Instead of exploring all possible paths equally, it favors lower cost paths. It runs on a weighted graphs, but can be used either for directed or undirected ones. At the beginning, it starts with the initial node and the target node [1, 2, 3, 4].

Example (extracted from [5]):

At the very beginning we initialize the initial node V with zero, all others with infinity. We have a list to track all the nodes we visited.



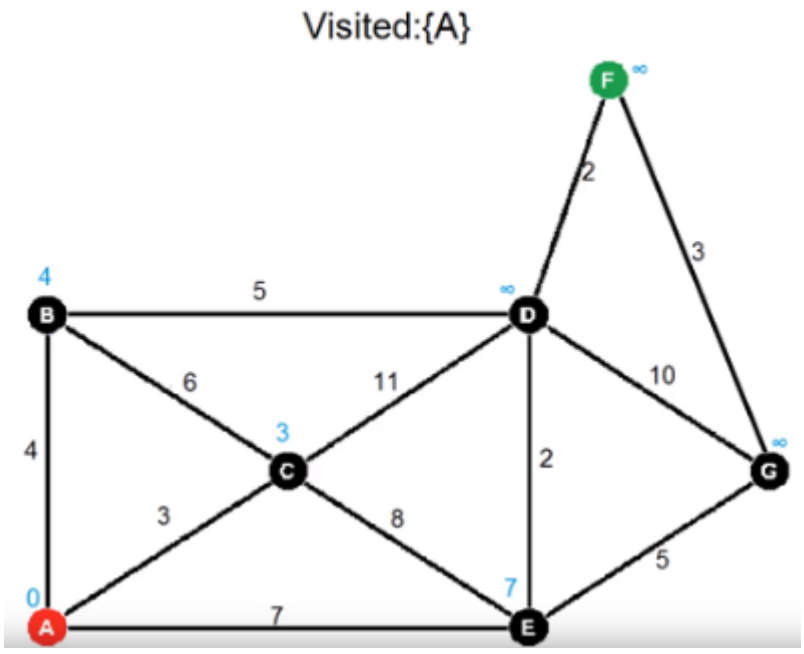
As we have only visited A, we check the neighboring nodes B, C and E. For each vertex we calculate: distance to the current node + distance from current node to neighbor. In this example for the nodes

B: 0+4=4

C: 0+3=3

E: 0+7=7

If this value is less than the current tentative distance, then we replace it with this newly calculated value.



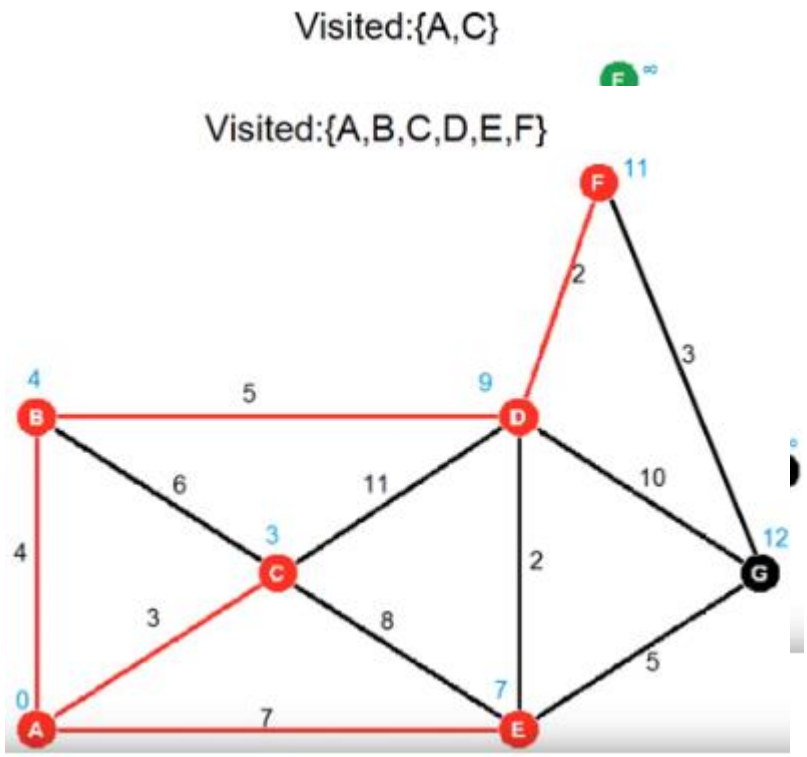
In a next step we visit the vertex having the lowest cost value, in our example this is C (it gets part of the list of the visited vertices). Again, we update the values for the neigboring vertices:

B: 3+6=9

D: 3+11=14

E: 3+8=11

As only vertex D shows a lower distance value (14<infinity) we update only this vertex.



The same procedure is repeated with all the vertexes that have a low weight cost. After C we visit B, from B we go to D and from D to the target node F.

The Dijkstra algorithm's pseudo code is the following:

```
function Dijkstra(Graph, source):
  for each vertex v in Graph: //initialization
    dist[v] := infinity //initial distance from source to vertex v is set to infinite
    previous[v] := undefined //previous node in optimal path from source
  end for
  dist[source] := 0 //distance from source to source
  Q := the set of all nodes in Graph //all nodes in the graph are unoptimized → are in Q
  while Q is not empty: //main loop
    u := node in Q with smallest dist[ ]
    remove u from Q
     for each neighbor v of u: //where v has not yet been removed from Q
      alt := dist[u] + dist_between(u, v)
      if alt < dist[v] //relax (u,v)
        dist[v] := alt
        previous[v] := u
      end if
     end for
```

```
  end while
  return previous[ ]
```

Complexity

- Time complexity
  - Using a binary heap: O((|E|+|V|)log(|V|))
- Space complexity
  - Using a binary heap: O(|V|)
  - Up to |V| vertices may have to be stored

Advantages

1. Uniformed Algorithm: Dijkstra is an uninformed algorithm. This means that it does not need to know the target node beforehand. For this reason it's optimal in cases where you don't have any prior knowledge of the graph when you cannot estimate the distance between each node and the target.

2. For multiple target nodes: Since Dijkstra picks edges with the smallest cost at each step it usually covers a large area of the graph. This is especially useful when you have multiple target nodes but you don't know which one is the closest.

Disadvantages

1. Fails if weights are negative: If we take for example 3 nodes (A, B and C) where they form an undirected graph with edges: AB = 3, AC = 4, BC=-2, the optimal path from A to C costs 1 and the optimal path from A to B costs 2. If we apply Dijkstra's algorithm: starting from A it will first examine B because it is the closest node. and will assign a cost of 3 to it and therefore mark it closed which means that its cost will never be reevaluated. This means that Dijkstra's cannot evaluate negative edge weights.

*4.2.1.3 A**

A* is a modification of Dijkstra's Algorithm that is optimized for a single destination. Dijkstra's Algorithm can find paths to all locations; A* finds paths to one location, or the closest of several locations (in contrast to Dijkstra, A* uses an additional heuristic that tells how far we have to go, e.g., Euclidean distance + the way from the initial node (as in Dijkstra)). It prioritizes paths that seem to be leading closer to a goal. This useful especially in environments with obstacles [6, 7, 8].

g(n)…exact cost of the path from the starting point to any vertex n

h(n)…heuristic estimated cost from vertex n to the target

In every step it examines the vertex n that has the lowest f(n)=g(n)+h(n)

```
Initialize the open list
Initialize the closed list
put the starting node on the open
list (you can leave its f at zero)
while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"
    b) pop q off the open list
    c) generate q's 8 successors and set their
       parents to q
    d) for each successor
        i)   if successor is the goal, stop search
             successor.g = q.g + distance between successor and q
             successor.h = distance from goal to successor (This can be done using many way
             we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)
             successor.f = successor.g + successor.h
```

```
        ii)  if a node with the same position as
             successor is in the OPEN list which has a lower f than successor, skip this
             successor
        iii) if a node with the same position as successor  is in the CLOSED list which has
             a lower f than successor, skip this successorotherwise, add  the node to the
             open list
       end for
    e) push q on the closed list
end while
```

Advantages

1. Heuristic: A* expands on a node only if it seems promising. It's only focus is to reach the goal node as quickly as possible from the current node, not to try and reach every other node

2. Complete: A* is complete, which means that it will always find a solution if it exists.

3. Can be morphed into other algorithms: A* can be morphed into another path-finding algorithm by simply playing with the heuristics it uses and how it evaluates each node. This can be done to simulate Dijkstra, Best First Search, Breadth First Search and Depth First Search.

Disadvantages

1. Not useful if you have many target nodes: If you have many target nodes and you don't know which one is closest to the main one, A* is not very optimal. This is because it needs to be run several times (once per target node) in order to get to all of them.

Complexity

- Time complexity
  - Depends on the heuristic, e.g., if the search space is a tree O(log h*(x))
- Space complexity
  - Roughly the same as all other graph search algorithms

4.2.1.4 Bellman-Ford

Starting at a particular node in a directed graph, the Bellman-Ford algorithm computes the shortest paths to all other nodes by progressively improving approximations [9, 10, 11].

```
function BellmanFord(vertices, edges, source)
   // set up distance table
   distance := array
   for each vertex v in vertices:
       distance[v] := inf
   end for
   distance[source] := 0
   // iteratively improve results
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
           end if
       end for
   end for
   return distance
```

Complexity

- Time Complexity
  O(|V| * |E|) where |V| is the number of vertices and |E| the number of edges in the graph
- Space Complexity
  O(|V|) where |V| is the number of vertices in the graph

4.2.2 Select Role

1. Decide on the role (leading or following vehicle)

- "Physical" first
  - Auctioning
  - Priorities
  - Platoon size
- Consensus:
  For applying such an algorithm, the main goal of platooning should be clear.
  For example: in [1], Leader-Following consensus algorithm has been used in Vehicle platoons:
  The goal is to maintain a fixed distance of the so-called ''String Stability''
  Each vehicle can share information with the neighbours, and with the Leader as well if it is a neighbour of the leader (in case of having a direct path, the leader information could be passed onto followers then a cost function is applied to maintain a fixed distance
  Generally, to decide if a vehicle should act a new leader, or just to follow the other one, or to do nothing (keep leading on its own) is highly related to the exchanged information in the model and the corresponding cost function as well. another example of using a consensus algorithm can be found in [2] ' consensus-based approach for platooning with inter-vehicular communications'. Also, the main goal was to regulate speed and relative distance of each vehicle with respect to its predecessor and the leading vehicle.
  References
  - [1] Ruan, Y. and Jayaweera, S.K., 2014, October. Leader-following consensus in vehicle platoons with an inter-vehicle communication network. In 2014 8th International Conference on Telecommunication Systems Services and Applications (TSSA) (pp. 1-6). IEEE.
  - [2] Santini, S., Salvi, A., Valente, A.S., Pescapè, A., Segata, M. and Cigno, R.L., 2015, April. A consensus-based approach for platooning with inter-vehicular communications. In 2015 IEEE Conference on Computer Communications (INFOCOM) (pp. 1158-1166). IEEE.

2. Cooperative Path Finding

2a. Exchange costs (individual algorithms)

2b. Recalculate shortest path algorithm (individual algorithms)

4.2.3 Follow Lead

Follow the lead, until the recalculated shortest path defined to leave the platoon

4.3 Search & Rescue

The first design of the state machines for CPSs involved in the SAR scenario was described in D4.5. In the following period the focus was directed towards the enrichment of the modeling facilities to realize more complex state machines. In particular, the attention was focused in improving aspects related to the management of events received by a specific CPS.

Before explaining the result of this activity, taking a step back to review some concepts related to the event management in the CPSwarm context is useful.

4.3.1 Events management

An event is anything that happens or take place during the execution of a mission. The main purpose of an event is to trigger a reaction from one or more members of the swarm. Generally, this reaction consists in a change of the current behavior of the CPS.
During the first period of the project threee different source of events have been identified:

- Swarm member: an event can be received from other swarm members.
- Monitoring & Command Tool: this case can be considered an extension of the previous one as, in the current implementation, the tool works as one of the members of the swarm.
- The CPS itself: event can be arisen internally to the single swarm member. In this case, the source can be identified among one of the sensors and actuators (through the Abstraction Library) or one of the high-level functionalities and behaviors that are simultaneously running on board of the CPS.

Independently from the source, the reaction to a specific event inside the CPS can be managed at different level of our software architecture:

- Abstraction Library level: the event can be processed by one of the components that are part of the Abstraction Library. For instance, if the event corresponds to a specific request to one of the actuators on board of the CPS, the related ROS component interfacing with the hardware will manage that request.
- Swarm Library level: in this case, the event is managed by one of the functionalities provided by the Swarm Library. The specific function will be able to read the data that are part of the event and execute any computations based on those data. Furthermore, this event can cause the drop-down generation of other events inside the CPS.

In both cases, the reaction to a specific event was expected by the developer that was implementing that particular hardware driver or swarm functionality. To overcome the limitations of such approach, the Finite State Machine Modeling was extended with the event monitoring at the state machine level. In fact, each state machine corresponds to a single process running such as any other ROS component executing on the CPS.
In the previous version of this deliverable, this capability was modeled using a dedicated Event Monitoring state that was responsible to process all kind of events (as can be seen in Fig.). Subsequently, this concept has been better detailed to satisfy the requirement needed to allow the automatic generation of the state machine through the CPSwarm Code Generator.
This improved technique consists in having a new dedicated Monitoring State running in parallel with a specified executing state. This new state will be in charge of listening to a single specific event and after the reception, will stop the executing task allowing the transition to another state. Associated to each specific state there may be one or many Monitoring States. One for each event of interest.
The Monitoring State has been used into the updated version of the state machines for both drones and rovers. More details on the new FSM will be presented in the following section.

4.3.2 Updated State Machines

In this section the main updates applied to the FSM for SAR scenario will be detailed. The mission that is actually considered is just an extension of the previous one. Many parts of these improvements were possible using the previously described Monitoring State.

4.3.2.1 1st Level – General State Machine

The main change from the last version (D4.5 – Updated Swarm Modeling Library) of this FSM is the deletion of the Event Monitoring event.

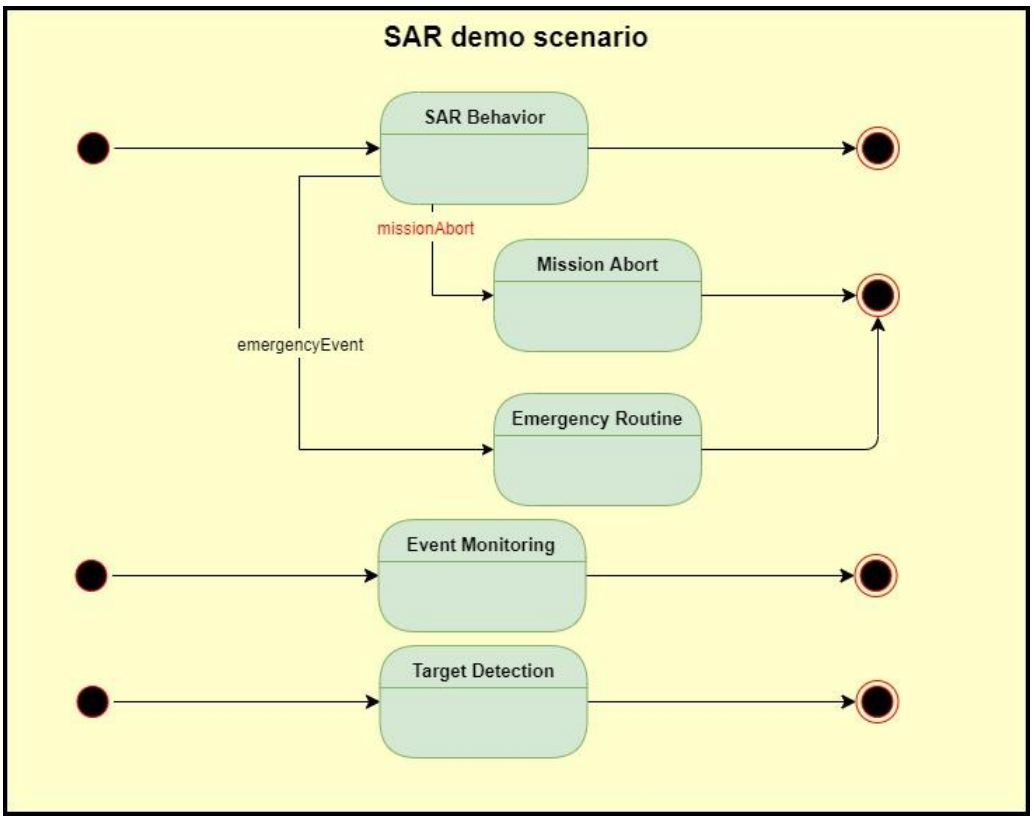


Figure 9 1st Level State Machine (old version)

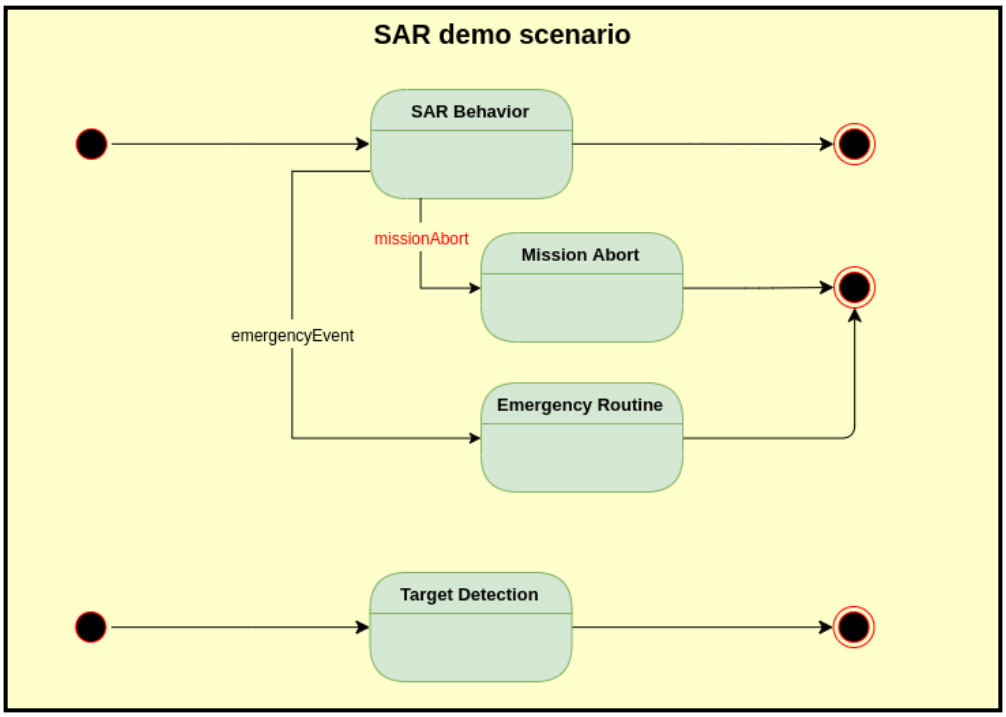


Figure 10 1st Level State Machine (final version)

4.3.2.2 2nd Level – Drone State Machine

This level of the drone state machine has been strongly modified. As can be observed in Figure 11, 3 Monitoring States have been added to the FSM:

1. The first one is the Idle Monitoring state. While the drone is in Idle phase, the new state will wait for the mission start event coming from the Monitoring Tool.
2. The others 2 addition were used to extend the previous version of the mission: a drone tracking a target needs to come back home because of battery low issues. In order not to lose the target, another drone, doing coverage, has to substitute for it. The new drone will be selected autonomously by the swarm. Indeed, when a tracking drone received a *battery low* event through the new Tracking Monitoring state, a new drone has to be selected ("assignTask()" function). In the meanwhile, a drone performing coverage will be now able to react to the substitution request coming from another drone thanks to the Coverage Monitoring State.

4.3.2.3 2nd Level – Rover State Machine

The rover state machine (see Figure 12) has just been updated with the correction related to the reception of the *target found* event coming from one of the drones involved in the mission.

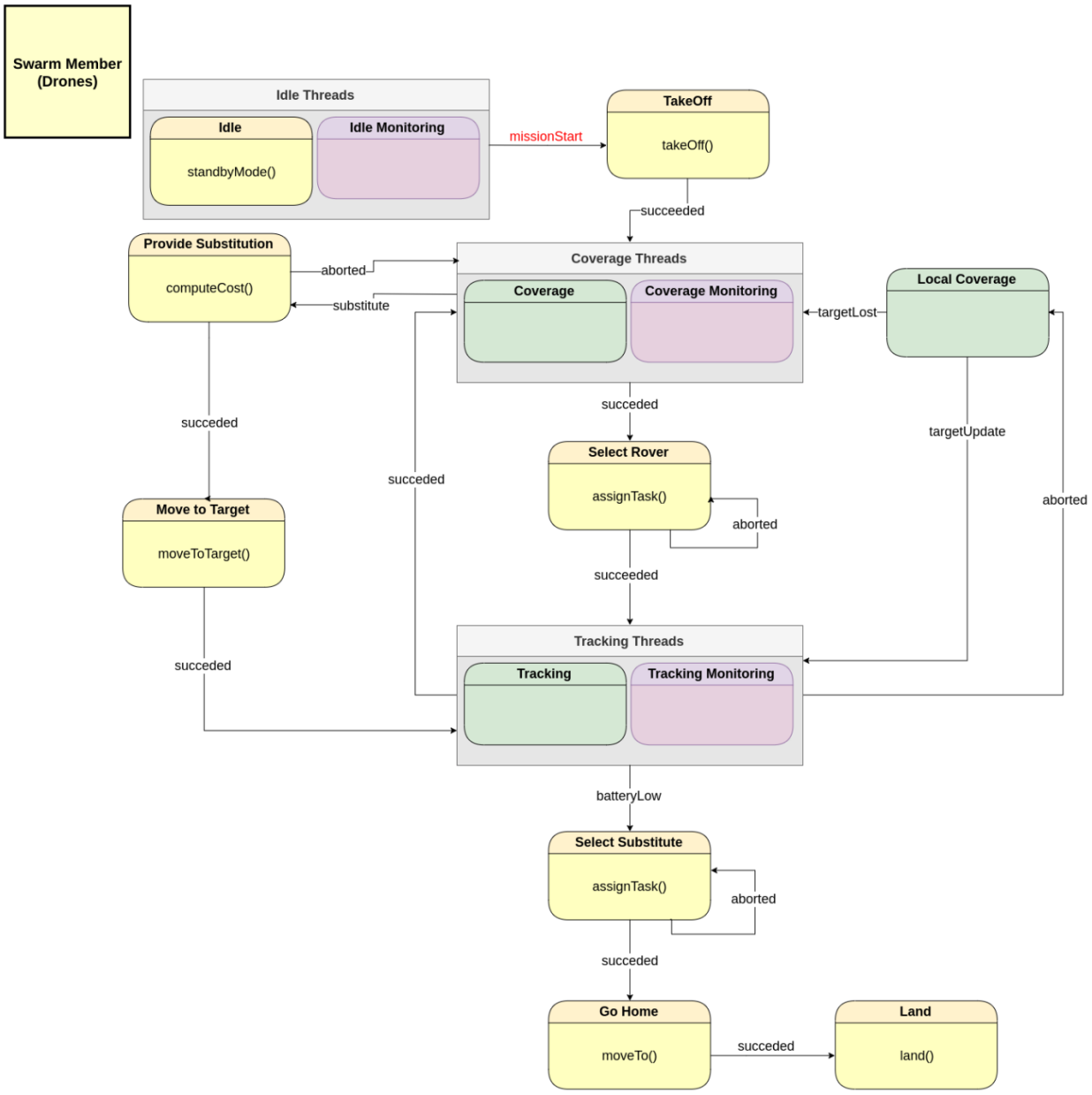


Figure 11 Final drone state machine for SAR scenario

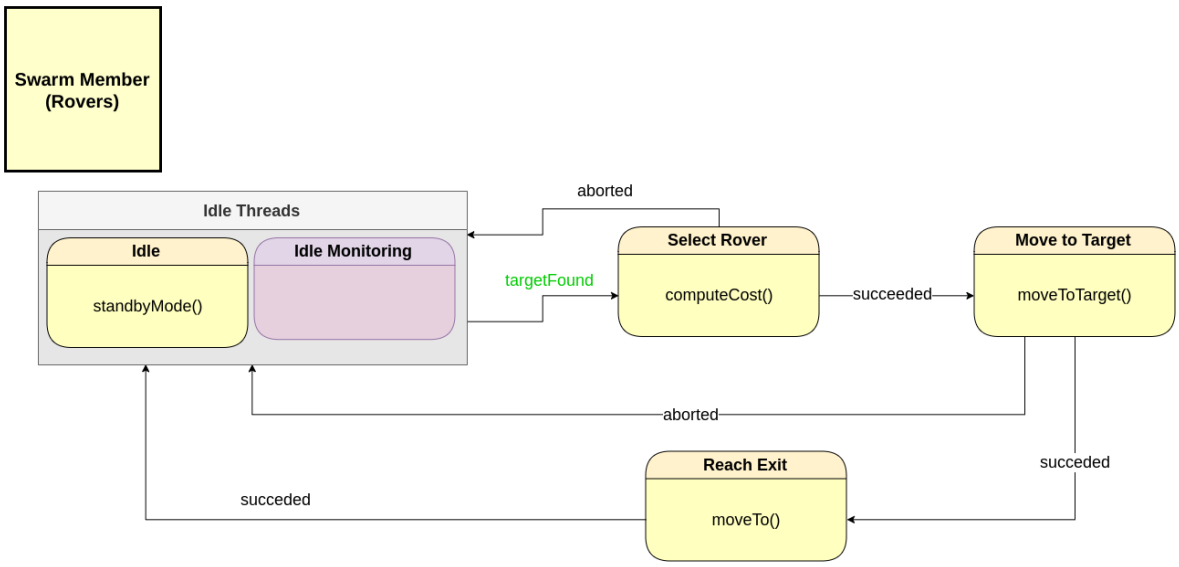


Figure 12 Final rover state machine for SAR scenario

5 Conclusions

In this deliverable we provide the final swarm modeling library: we introduced the final library structure and all relate components. We use the concepts of state machines as introduced in the previous deliverables and updated them for all three use cases: logistics, automotive, and search & rescue. to describe the behavior of the use cases, with a focus on the search and rescue use case. Especially for the automotive use case, we provided a set of algorithms that can be used within the state machine.

Acronyms

Acronym	Explanation
DoA	Description of Action
FSM	Finite State Machine
KPI	Key Performance Indicators
UML	Unified Modeling Language

References

[1] „Dijkstra's shortest path algorithm," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/. [Zugriff am 12 06 2019].

[2] „Dijkstra Algorithm: Short terms and Pseudocode," Geographic Information Technology Training Alliance, 13 05 2014. [Online]. Available: http://www.gitta.info/Accessibiliti/de/html/Dijkstra_learningObject1.html. [Zugriff am 12 06 2019].

[3] R. B. Games, „Introduction to the A* Algorithm," Red Blob Games, 26 05 2014. [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html. [Zugriff am 12 06 2019].

[4] I. Tinca, „From Dijkstra to A Star (A*), Part 1: The Dijkstra Algorithm," Big Data Zone, 08 02 2019. [Online]. Available: https://dzone.com/articles/from-dijkstra-to-a-star-a. [Zugriff am 13 06 2019].

[5] N. Fan, „Dijkstra's Algorithm," Youtube, 24 11 2012. [Online]. Available: https://www.youtube.com/watch?v=gdmfOwyQlcI. [Zugriff am 12 06 2019].

[6] E. Sukaj, „What is the best alternative to Dijkstra's Algorithm?," Slant, 2015. [Online]. Available: https://www.slant.co/options/11584/alternatives/~dijkstra-s-algorithm-alternatives. [Zugriff am 12 06 2019].

[7] A. Patel, „Introduction to A*," Red Blob Games, 2019. [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html. [Zugriff am 12 06 2019].

[8] „A* Search Algorithm," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/. [Zugriff am 13 06 2019].

[9] A. Shimbel, „Structure in communication nets," *Proceedings of the Symposiuum on Information Networks,* Bd. Polytechnic Press of the Polytechnic Institute of Brooklyn, pp. 199-203, 1955.

[10] R. Bellman, „On a routing problem,“ *Quarterly of Applied Mathematics,* Bd. MR 0102435, Nr. 16, pp. 87-90, 1958.

[11] L. Ford, „Network Flow Theory,“ RAND Cooperation, 1956.

List of figures