# Introduction

## Overview

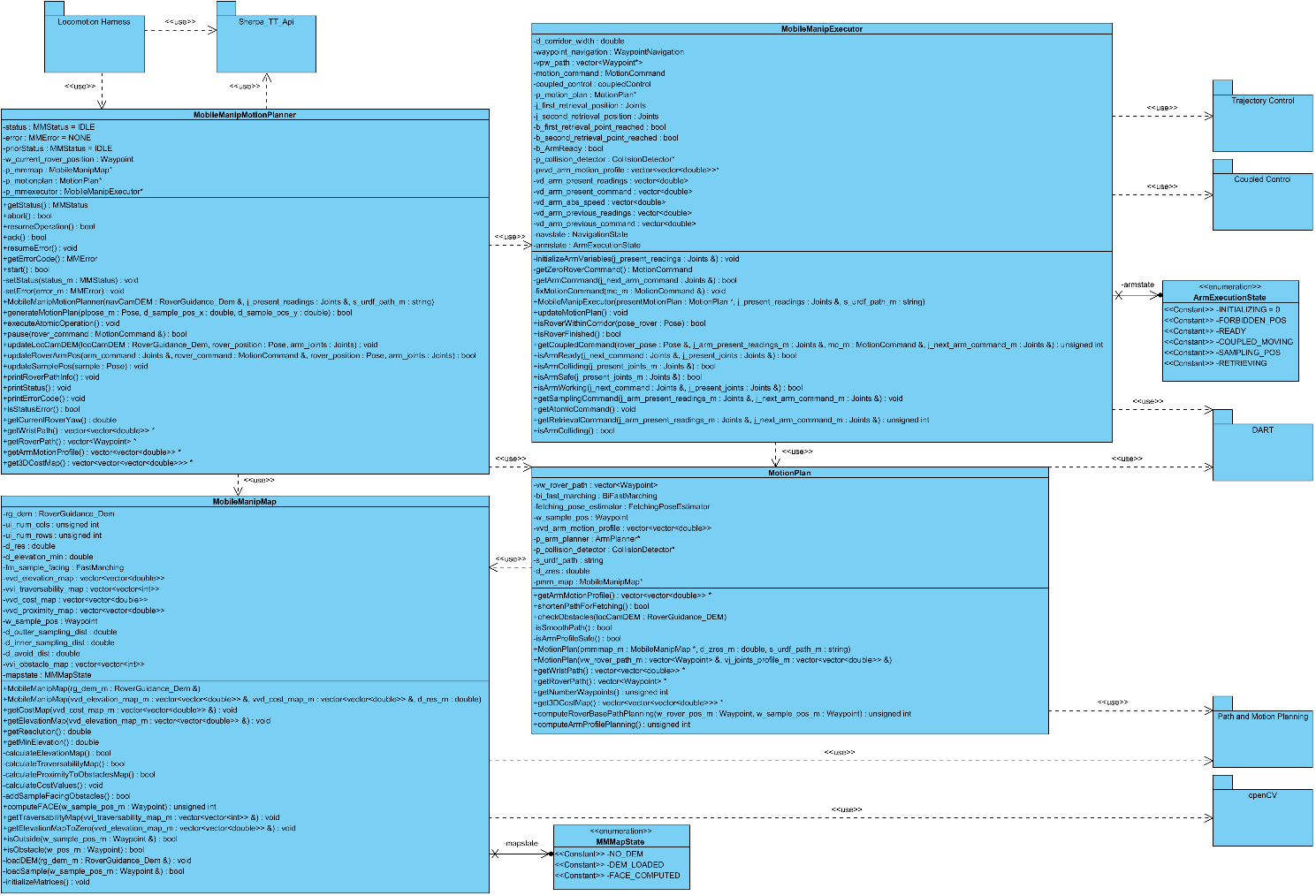
The objective of this document is to provide an initial UML model of the Mobile Manipulation subsystem and how it could be integrated within the whole system. Moreover, a brief description of the used algorithms and their location into the UML class diagram is provided. Finally, a Sequence Diagram is provided in order to show interaction between defined classes and the rest of the system.

# UML Model

## Class diagram

The proposed software architecture is briefly based on four classes as shown in Figure 1. The main class is the *MobileManipMotionPlanner*, which serves as the interface with the harness/guidance component. This class receives the DEM using the same data structure as defined by Airbus for the Rover Guidance. It is so important to include the DEM error within this data structure. Once received, an instance of the *MobileManipMap* class is created. This class includes the current DEM and calculates the obstacles and cost map that would be used to later generate a motion plan. Once a motion plan is generated using the respective class (*MotionPlan*), it can be executed by creating an instance of the *MobileManipExecutor* class, which contains the motion plan.

There are two types of operation this software is expected to be able to perform. First, the atomic operation, consisting on exclusively controlling the arm to make a certain movement. Second, a coupled arm-rover motion, which must be produced by means of Path and Motion Planning libraries based on the Fast Marching Method (in 2D and 3D). The latter consists on coordinating the rover and the arm to reach the location of a sample and place the end effector on it. Then, it is expected to generate the motion to execute a particular task to do something with the sample, e.g. place the end effector in contact with the sample, pick up or drop a sample of soil.

**

*Figure 1. Class diagram of the Mobile Manipulation subsystem.*

### Table of main functions for Harness

|  |  |
| --- | --- |
| **Function** | **Used to** |
| bool **generateMotionPlan**(proxy\_library::Pose, double x, double y) | It computes the rover path and the position commands profile for the arm joints. Returns TRUE if successful (the status goes from IDLE to READY\_TO\_MOVE), FALSE if any error occurred (status goes to ERROR). |
| bool **start**() | If the MM status is READY\_TO\_MOVE, takes it to EXECUTING\_MOTION\_PLAN and then TRUE is returned. Otherwise it will take the status to ERROR, with error code IMPROPER\_CALL, and returns FALSE. |
| bool **abort**() | If status is:  -READY\_TO\_MOVE: status returns to IDLE. Function returns TRUE.  -EXECUTING\_MOTION\_PLAN, EXECUTING\_ARM\_OPERATION or PAUSE: status goes to RETRIEVING\_ARM. Function returns TRUE.  -Any other: status goes to ERROR, with error IMPROPER\_CALL. Function returns FALSE. |
| bool **updateRoverArmPos**(proxy\_library::Joints&, proxy\_library::MotionCommand&, proxy\_library::Pose, proxy\_library::Joints) | It provides, by reference, commands for the arm joints and the rover, according to their positions. Returns TRUE if, according to the input positions, the execution is working fine, FALSE in case something went wrong or the execution just finished (use getStatus() to check). In affirmative case, depending on the current positions the MM status will transition between EXECUTING\_MOTION\_PLAN, EXECUTING\_ARM\_OPERATION , RETRIEVING\_ARM and FINISHED, following that order. |
| bool **pause**(MotionCommand&) | Takes the MM status to PAUSE and returns TRUE, only if current status is either EXECUTING\_MOTION\_PLAN, EXECUTING\_ARM\_OPERATION or RETRIEVING\_ARM. A rover command to stop is also provided by reference. Otherwise status goes to ERROR, with error code IMPROPER\_CALL, and FALSE is returned. |
| bool **resumeOperation**() | Changes status from PAUSE to the previous one, then returns TRUE. If at that moment status is not PAUSE, status goes to ERROR, with error code IMPROPER\_CALL, and FALSE is returned. |
| bool **ack**() | It can only be called in FINISHED status, otherwise instead of a TRUE it will return a FALSE and the error code will be IMPROPER\_CALL. |
| void **resumeError**() | Its functioning depends on the type of error that is existing. For example, in case of the IMPROPER\_CALL error code, the MM status returns to the one before entering into ERROR status. |
| bool **isStatusError**() | Returns whether the MM status is ERROR or not. |
| MMError **getErrorCode**() | Returns the code of the existing error. |
| MMStatus **getStatus**() | Returns the MM status. |
| void **printRoverPathInfo**() | Prints information about the computed path in the console. |
| void **printStatus**() | Prints the current status in the console. |
| void **printErrorCode**() | Prints the error code in the console. |
| double **getCurrentRoverYaw**() | Returns the last yaw angle registered in radians. |
| std::vector<std::vector<double>>\* **getWristPath**() | Returns a pointer to the 3-D path followed by the arm wrist. |
| std::vector<base::Waypoint> **getRoverPath**() | Returns a pointer to the rover path. |
| std::vector<std::vector<double>>\* **getArmMotionProfile**() | Returns a pointer to the profile of joints positions planned. |
| std::vector<std::vector<std::vector<double>>>\* **get3DCostMap**() | Returns a pointer to the 3D tunnel cost map created. |

## Sequence Diagram

### Initial plan execution

An example of the use of the proposed design is shown in Figure 2. This diagram illustrates an initial execution of a plan to move the rover and the manipulator in a coupled way to reach a sample. First, the harness component calls the *MobileManipMotionPlanner* constructor to instantiate the class. This constructor has the rover surrounding DEM, with all of its metadata, as parameter. This DEM is processed in order to calculate the obstacles and cost maps. Later on, the harness component sends a motion plan that is based on the initial rover pose (position and orientation) and the estimated sample location. This method generates the rover path and manipulator trajectory using the Fast Marching Method (FMM) algorithm. It is based on the provided information and the *MobileManipMap* object. If the sample can be reached, it provides a *MotionPlan* object, which is later sent to the *MobileManipExecutor* through the use of the class constructor. Once it is stored, the subsystem would be ready to run the motion plan, which would begin once the *start()* method is called.

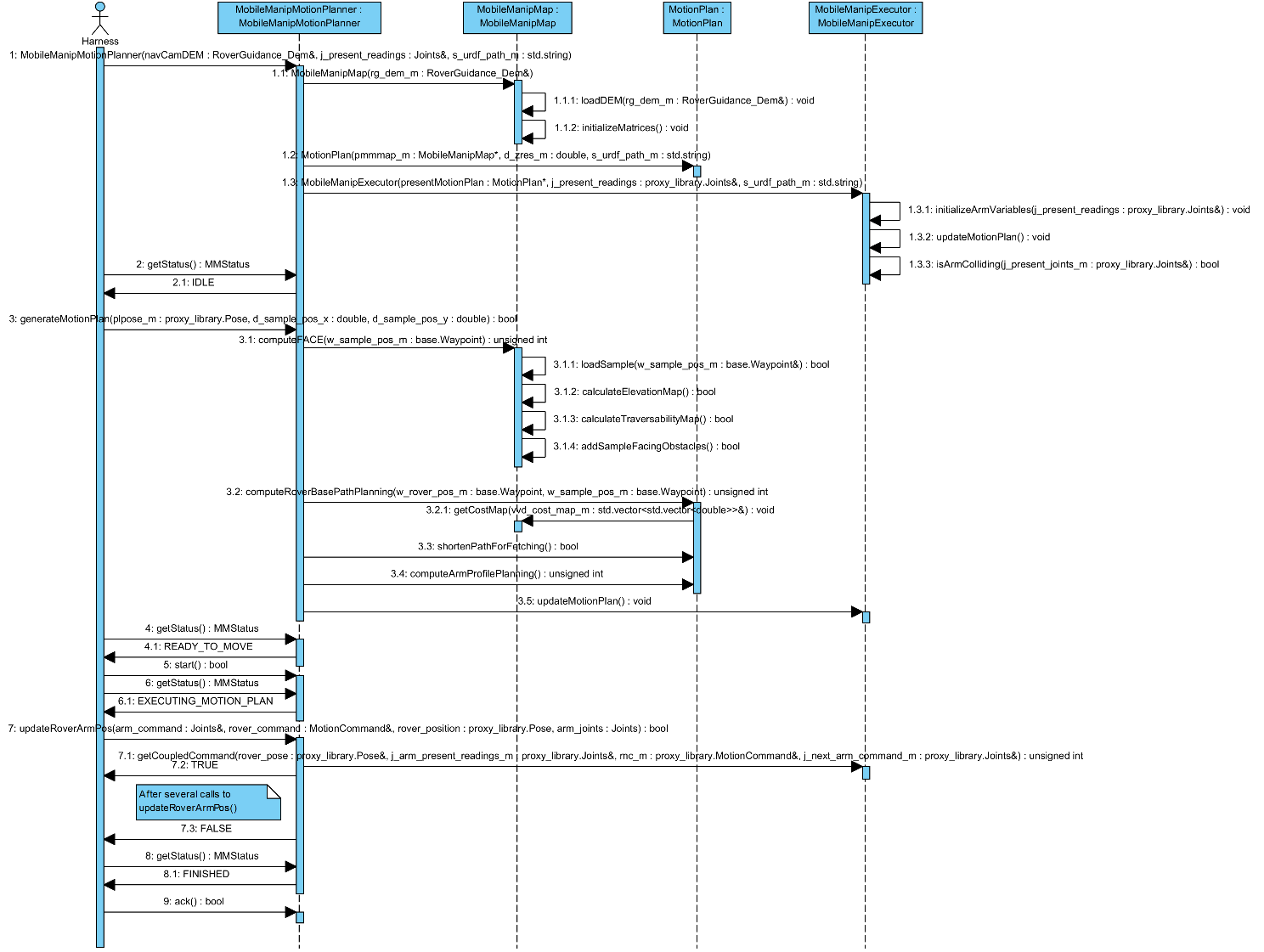


Figure 2. Sequence diagram for the initial execution of a plan.

### Atomic Operation

Instead of executing a fully coupled arm-rover motion, an operation involving exclusively the arm can be chosen. In this case, as depicted in Figure 3, the harness calls to *executeAtomicOperation()* at the IDLE state. Then a Motion Plan is created consisting on only moving the arm to perform an operation. Later on, the state changes to EXECUTING\_ARM\_OPERATION and the position commands for the arm joints are generated.

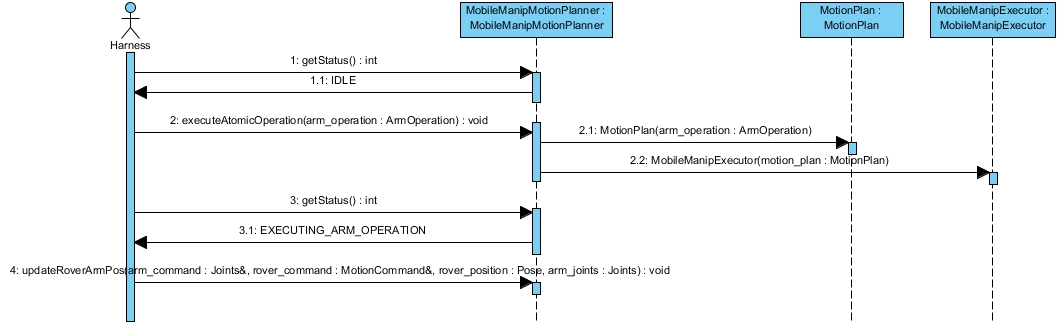


Figure 3. Sequence diagram for an atomic operation.

### Pause operation

During the execution of the motion plan can arise the need of pausing it. Figure 4 depicts the sequence diagram in which this case is represented.

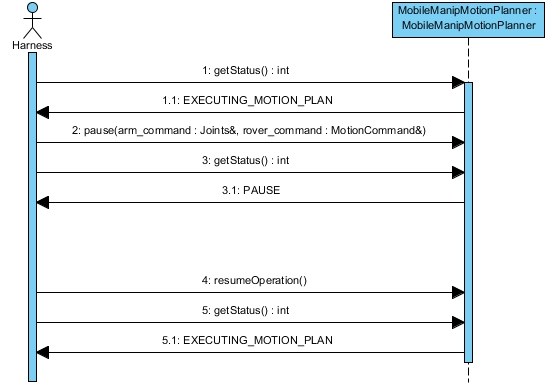


Figure 4. Sequence diagram for pausing the execution

### Error handling example

Figure 5 illustrates how an error during the motion plan execution arises and is handled. In this case the error code is EXCESSIVE\_DRIFT (see table I), which is caused by the rover being too far from the planned path. As result, the planner enters into the ERROR state and, according to how this error must be handled, the Motion Plan is computed again.

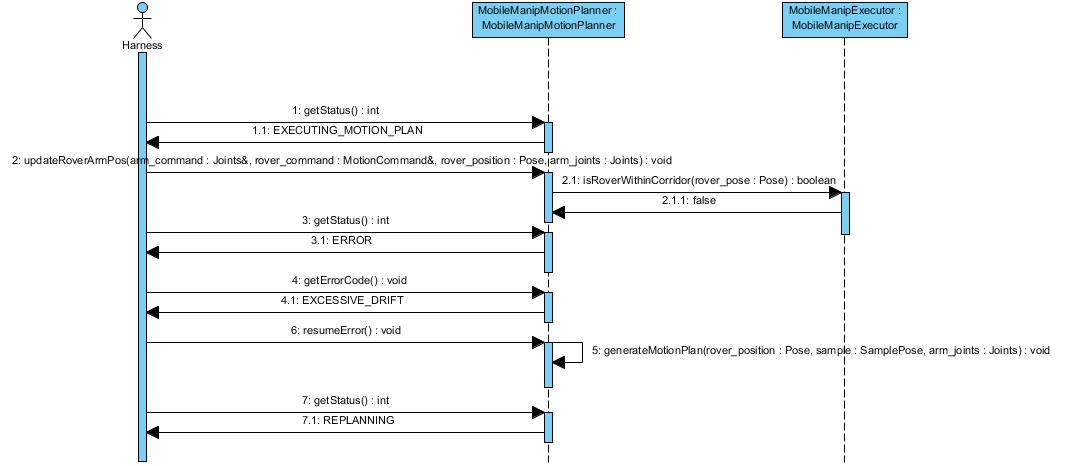


Figure 5. Sequence diagram for an error handling case

### LocCam DEM update and replanning

Figure 6 shows what happens when a new DEM from the Localization Camera (LocCam) is received and contains an obstacle within it. In this case, *checkObstacles()* returns a true value, indicating there is in fact an obstacle on the rover path. Therefore, the Motion Plan is generated again. COMPLETAR ESTO!!!!!!!

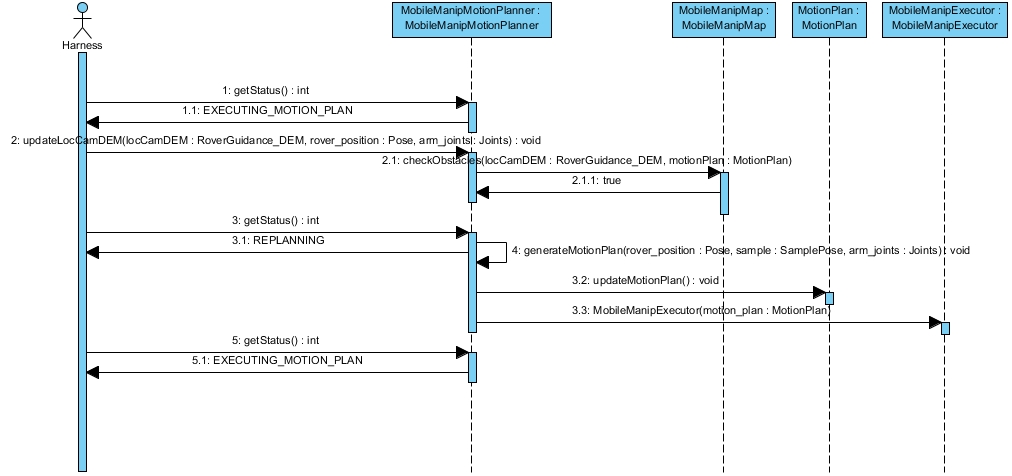


Figure 6. Sequence diagram for a replanning

## State-machine Diagram

Figure 7 depicts a diagram with the different states of the components and their transitions. The meaning of each state is denoted as follows:

* **IDLE**: The state in which the component starts when it is created and finish a motion. It serves as a starting point to either perform a complete rover-arm operation or an atomic arm operation.
* **GENERATING\_MOTION\_PLAN**: The software computes the optimal motion plan for the rover-manipulator system in order to reach a the sample location.
* **READY\_TO\_MOVE**: Once the Motion Plan is successfully computed, the component waits until the corresponding function to start the execution, *start(),* is called.
* **EXECUTING\_MOTION\_PLAN**: During this state, the component provides the commands to the rover and the arm according to their current positions and the computed Motion Plan.
* **PAUSE**: During the execution of the motion plan, it shall arise the need of making the system stop moving. Later, the software can resume this execution and continue the planned motion.
* **REPLANNING**: In case an obstacle is detected, the sample location is changed or some errors arise, the component goes into this state. Here, the cost map is updated and the Motion Plan is computed again.
* **EXECUTING\_ARM\_OPERATION**: In this state, the rover base is stopped, while the arm motion is controlled to perform a preprogramed operation.
* **RETRIEVING\_ARM:** The arm is controlled to reach the home position.
* **FINISHED**: After the whole operation has concluded, the program enters into this final state, waiting for an *ack()* to go to the idle state and wait for a new motion plan.
* **ERROR**: in case the program encounters with any error during its execution, it goes to this state. According to the type of error, the function *resumeError()* will act according to Table 1.

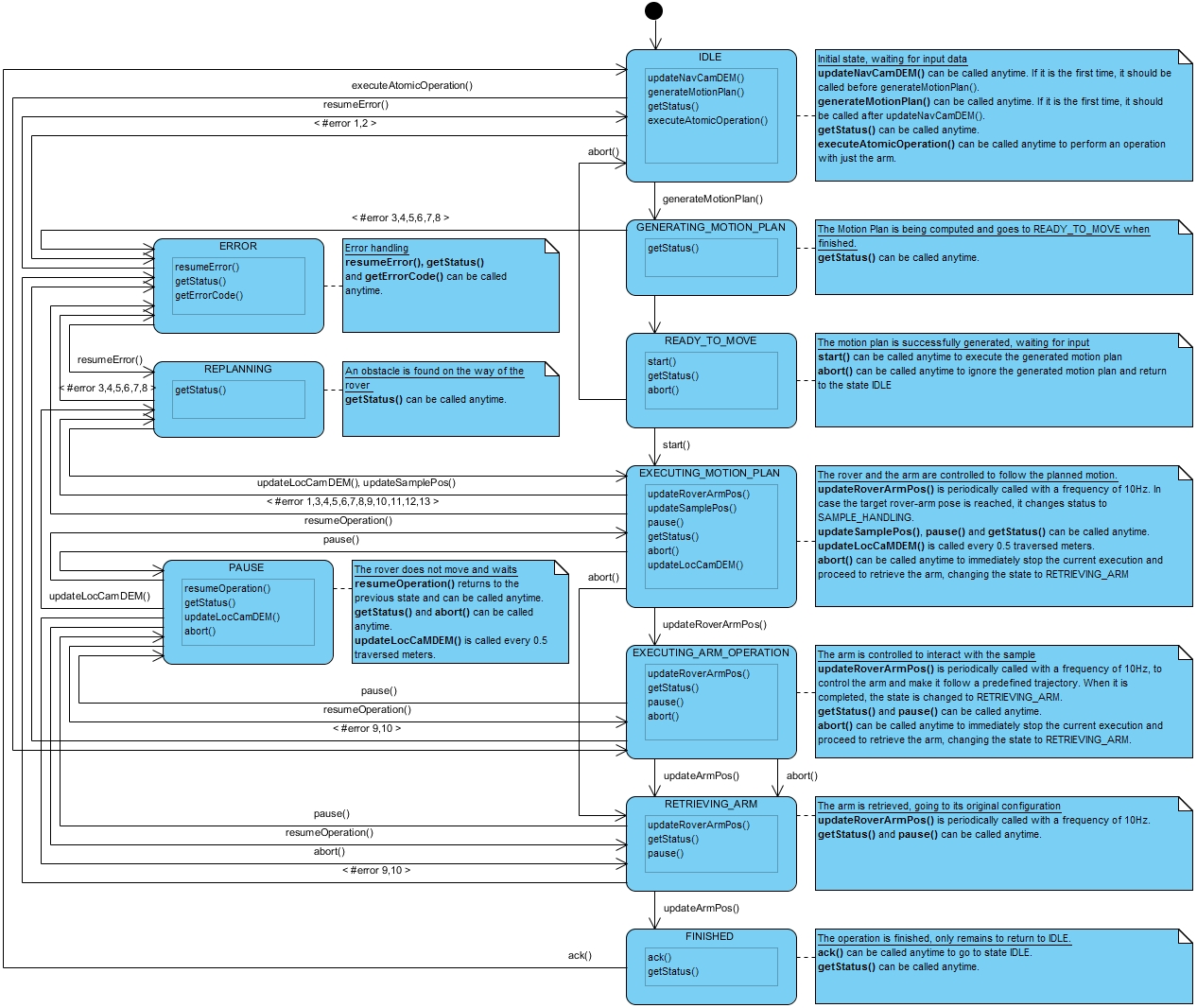


Figure 7 State machine Diagram

# Algorithms

## Motion plan and map generation

The objective of the motion planner is to reach the sample position and place the rover manipulator close to it, i.e. 10-20cm far. The proposed algorithm for rover path and arm trajectory generation is based on the Fast Marching Method (FMM), making use of 2D and 3D workspaces respectively. This method computes the numerical solution of a wave that propagates through the environment starting from a certain point. The rate at which the wave propagates depends on the cost assigned to every part of such environment, which is discretised into a grid. Thus, FMM computes the minimal time at which the wave arrives at each grid node. Then, by making use of the gradient descent method, a path is retrieved from any point to the one from which the wave started expanding. The main advantages of this method are:

* Smooth trajectories generation. Unlike other methods like A\* or D\*, the turning angles of the paths obtained through FMM are not restricted at all. Besides, the location of the waypoints making up these paths are not constrained to the location of the grid nodes, meaning they can be anywhere within the workspace. In this way, it is not necessary to apply any post-processing to the path to smooth it.
* Optimal solution. FMM numerically solves the propagation of the wave using the eikonal equation, an expression that correlates the propagation rate with a cost value defined at any workspace point. In this way, the retrieved paths always tend to be optimal, and the only error committed is due to the grid discretization. Other grid-search based methods like Field-D\* or Theta\* cannot ensure this, since, although the computed paths can be also smooth, they make use of estimation methods that introduce more error and, in some cases, can produce suboptimal solutions.
* Computer complexity similar to other path planning algorithms with less features, using a Dijkstra-based grid-search method to visit each grid node. The computer complexity is similar to A\* and D\*, the most typical path planning algorithms. However, as shown, this method is much better in some features.
* Parallelization. Since FMM computes the optimal solution of a wave propagation, by using its bi-directional version it can be parallelized: two waves can be propagated, one originated from the start and the other from the goal location. Then, they encounter at an intermediate point and, because of the nature of FMM, the whole path between start and goal is the concatenation of the path between the intermediate point and the start and between the intermediate point and the goal. This would be useful in the case of using multiple cores processors.

Therefore, the proposed algorithm is able to generate the rover path and the manipulator end effector trajectory, given a 2D cost map, the initial rover pose and the sample location, which corresponds to the desired final manipulator end effector position. Then, by using the inverse kinematics, a profile of the rover joint references can be generated depending on the relation between the rover path and the end effector trajectory.

Finally, since the cost map has a direct effect on the resulting path and end effector trajectory, the error committed to build it is here relevant. In this sense, the rover pose and the sample position, as well as the DEM, should be provided with their respective accuracy. The sum of all estimation errors provided have an impact on the uncertainty of the end effector position as shown in Figure 8. In this figure, the main reference frames from the rover and manipulator are shown. The first one is the rover position frame with respect to the world frame. Any error on the rover pose is extended to the end effector frame, e.g. a yaw error would increase the end effector position error based on the distance between the rover and the end effector frames (*L*). On the other hand, an error on the sample location would also increase the total error committed by the manipulator.

Taking into consideration there errors, a sphere can be defined. It represents the error space, i.e. the manipulator end effector would be in any place within the sphere. Therefore, the size of this sphere is proportional to the amount of introduced error. Assuming the manipulator has a Force/Torque sensor on the end effector, the vertical error could reduce by detecting the instant time the manipulator is in contact with the surface. It would belong to the final stage of the manipulator movement.

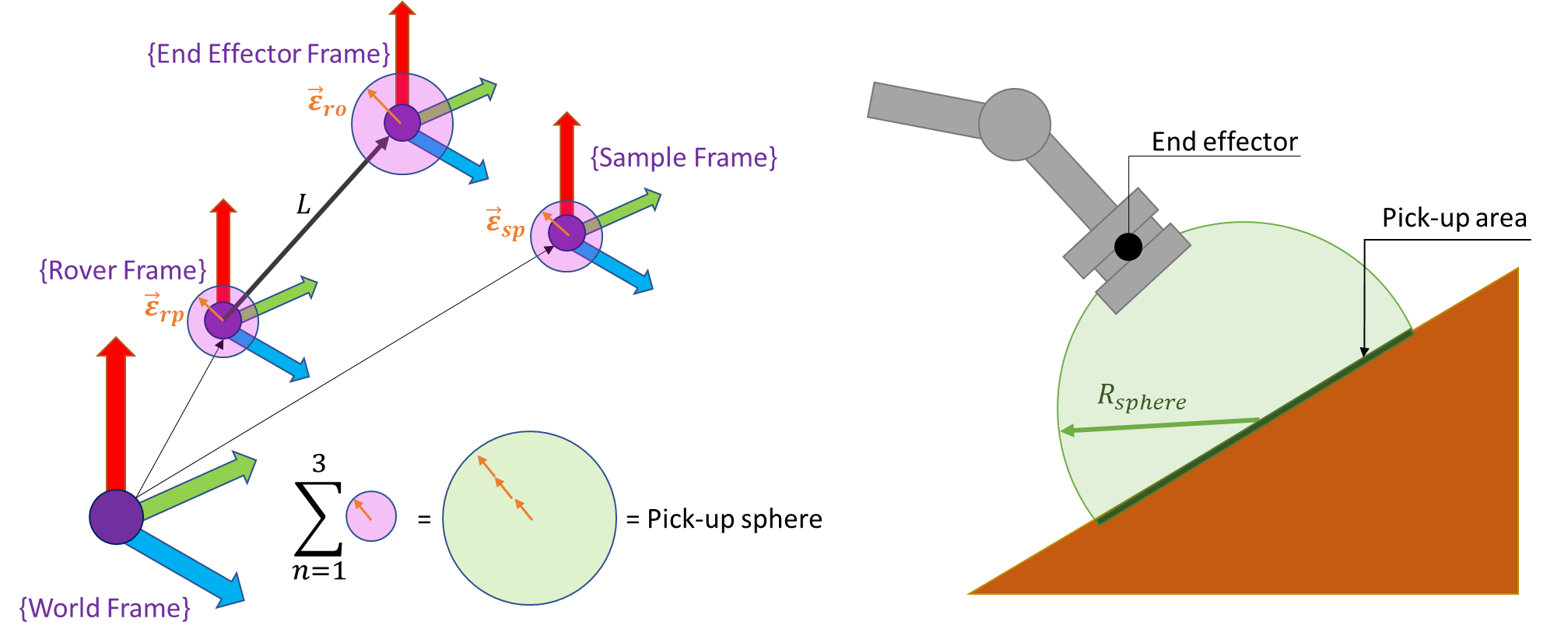


Figure 8.Relation between the estimation errors.

The provided DEM is processed to detect obstacles in the surrounding area of the rover. The size of these obstacles may be larger due to the amount of estimation error introduced. Therefore, reachability of some samples could be seen as non feasible by the algorithm because of the DEM error, although in the reality they could be in fact reached. For example, in Figure 9 there is a small corridor in position (5,15) that could be closed if there were a big estimation error on the DEM. Therefore, in some cases, the sample could be reached even when it could be reached in the reality.

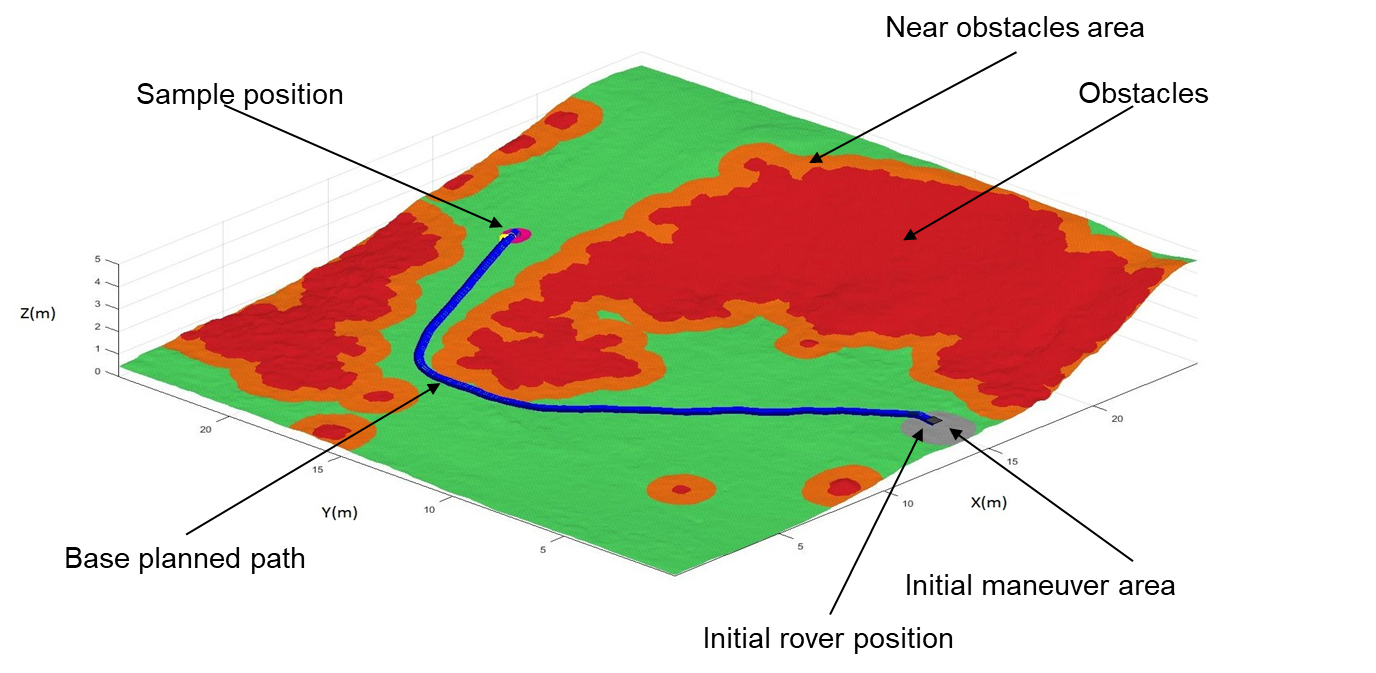


Figure 9.Cost map example.

## Sample handling tasks

Once the motion plan has been achieved, the next step is to perform a task with the sample. Two tasks are proposed, and one of them would be finally implemented:

* Place the manipulator end effector close to the sample. Emulating an instrument attached to the manipulator, which would need to do some kind of analysis with the sample e.g. spectrography.
* Pick up or drop a sample of soil. Using a special end effector, the manipulator would be able to pick up or drop a sample of soil on the final location.

To perform on these tasks, a preprogramed motion would be executed with the support of a Force/Torque sensor. This sensor would allow the system to know the manipulator end effector is in contact with the surface.

# Error handling

During the execution of the Mobile Manipulation component some issues can arise. Therefore, it is considered the use of methods to handle a certain series of errors. These are shown in the following table, together with the code designated to identify each of them.

Table 1. Summary of main errors and how to handle them

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Origin State | Code | # | Error Description | | Action |
| IDLE /  EXECUTING\_MOTION\_  PLAN | POOR\_DEM | 1 | Input is poorly defined | The input DEM is not properly built (i.e. mismatch between parameter sizes) | Return control to the higher component level (Harness). |
| IDLE | POOR\_CONFIG | 2 | The config file either misses some information or it is not fine | Return control to the higher component level (Harness). |
| GENERATING\_  MOTION\_  PLAN / REPLANNING / EXECUTING\_MOTION\_  PLAN | OOB\_ROVER\_POS | 3 | Position out of DEM boundaries | Rover position is not located within DEM | Return control to the higher component level (Harness). |
| OOB\_GOAL\_POS | 4 | Goal Position is not located within DEM |
| OBS\_ROVER\_POS | 5 | Position on obstacle area | For some reason, the rover is in forbidden area | Return control to the higher component level (Harness). |
| OBS\_GOAL\_POS | 6 | Goal located deep within obstacles | Return control to the higher component level (Harness). Goal shall be discarded. |
| DEGEN\_PATH | 7 | Internal error, the produced path is degenerate. This can be due to narrow corridors. | |  |
| COLLIDING\_PROF | 8 | Internal error, the generated arm profile is unsafe. | |  |
| DEVIATED\_PROF | 9 | Internal error, the generated arm profile is unfeasible. | |  |
| GOAL\_TOO\_CLOSE | 10 | The rover is too close to the sample to create a coupled operation. | |  |
| UNREACH\_GOAL | 11 | After attempting to compute the rover path, goal is not accessible | | Return control to the higher component level (Harness). it shall be discarded. |
| UNCERT\_GOAL | 12 | Target Pose is uncertain due to the sum of uncertainties (DEM and Target high errors) | |
| EXECUTING\_  MOTION\_  PLAN / EXECUTING\_  SAMPLE\_  HANDLING | NON\_RESP\_ARM | 13 | Arm does not respond. | | Return control to the higher component level (Harness). |
| COLLIDING\_ARM | 14 | Torque sensors detect the arm is colliding with something | | Withdraw the arm, then return control to the higher component level (Harness). |
| FORB\_ARM\_POS | 15 | The arm is at a forbidden configuration (maybe colliding or too risky) | |  |
| INCOMPLETE\_INPUT | 16 | Input information to create the commands is partially or totally missing. | |  |
| EXECUTING\_  MOTION\_  PLAN | NON\_RESP\_ROVER | 17 | Rover does not respond. | | Return control to the higher component level (Harness). |
| EXCESSIVE\_DRIFT | 18 | The rover is too far from the planned path | | Recompute the motion plan. |
| UNCERT\_HEADING | 19 | Too much uncertainty due to accumulated heading error | | Return control to the higher component level (Harness). |
| (ANY) | IMPROPER\_CALL | 20 | A function was called when it was not supposed to. | | Return to the previous state before entering into the ERROR state. |