GOTCHA: AN AUTONOMOUS CONTROLLER FOR THE SPACE DOMAIN

J. Ocón Alonso [(1)], J.M. Delfa [(1)], De la Rosa Turbides, T. [(2)], García-Olaya, A. [(2)], Escudero Martín, Y. [(2)]

[(1)] *GMV Aerospace and Defence, Isaac Newton 11 PTM, Tres Cantos, 28760 Madrid, Spain, jocon@gmv.com*
[(2)] *Universidad Carlos III de Madrid, Departamento de Informática Avda. de la Universidad, 30 28911 Leganés (Madrid) trosa@inf.uc3m.es, agolaya@inf.uc3m.es, yescuder@inf.uc3m.es*

ABSTRACT

GOTCHA is a project which aims to develop a robotic autonomous controller for the space domain. This paper discusses the GOTCHA system, the key design decisions taken, as well as the main conclusions reached so far in the frame of this project.

1. INTRODUCTION

The aim of the GOTCHA system (*GOAC TRL Increase convenience enhancements hardening and application extension*) is to have a template for an autonomous controller that can be instantiated for different space robots. This would allow having a design and a development procedure suitable to be used as on-board flight-software for future space robotic missions.

The proposed GOTCHA system will be able to offer goal-based commanding (in which the system can be commanded via dynamic high-level goals); dynamic re-planning (changing the plan on real-time when the current plan is no longer valid); selectable level of autonomy (on-board planning can be activated or deactivated from ground); and domain independent reasoning through the use of Automated Planning techniques. In addition, in GOTCHA's design we are taking into account the limited computing and memory resources typical of on-board space computers.

In this paper, we provide a detailed overview of the proposed GOTCHA system and its related designed components at deliberative, executive and functional layers. Then, we present some preliminary results, which show some successful decision-making capabilities. Finally, we present the conclusions.

2. SYSTEM CONCEPT

GOTCHA is based on GOAC [1], a previous TRP activity lead by GMV. Like GOAC, GOTCHA uses an interleaved planning and execution schema in which deliberation and reaction are harmonized via a controller or Agent.

Both GOAC and GOTCHA are based on the T-REX architecture [2] which is an evolution of IDEA [3]. T-REX divides a system in a hierarchy of components, called reactors, each one being on charge of a specific part of the system. A T-REX instantiation is comprised of a set of control loops, also known as *reactors* that encapsulate a set of deliberative and reactive behaviours. The number of reactors is fully configurable; in our architecture, a single reactor conforms the deliberative layer, meanwhile other reactors provide the executive part and interface with the functional layer. The system as a whole can be seen as a three layer architecture [4]: meanwhile the functional level layer deals with the spacecraft sensors and actuators, the executive layer transforms the high-level commands received from the deliberative layer into low level behaviours executable by the functional layer, and also transforms the sensors data into high-level knowledge understandable by the deliberative level. Finally the deliberative layer uses Automated Planning techniques to create plans fulfilling the goals received from ground and sends the appropriate commands to the executive layer.

In GOTCHA's architecture, a *reactor* is the owner of a set of internal *timelines*, being each timeline a state variable that can have a finite set of states, and is evolving in time. For instance, a reactor in charge of controlling the movement of the rover can have a *RoverBase* timeline with two possible statuses: *At(x,y)* (indicating that it is static at position (x,y) or *GoingTo(w,z)* – which would indicate that it is moving to position (*w,z).* Timelines owned by a reactor are considered *internal* to this reactor, which means that only this reactor can set its value. A goal can be posted to a reactor to request any of its internal timelines to have a value for a given time, this piece of information being the so-called *token*. In addition, other reactors can subscribe to the internal timelines of a given reactor, so that they are informed whenever there is a change of its value. These are considered as *external timelines* by the reactor that subscribes to them, which means its value is set by the reactor that owns the timeline, but any reactor subscribed will be informed by the *Agent* whenever there is a change in these timelines.

The system state can be therefore seen as a set of timelines (state variables) that are evolving in time, being the agent responsible to ensure that all reactors share a common view of all timelines at a given time [5]

3. TIMELINE-BASED VS ACTION-BASED PLANNING: PROS AND CONS.

3.1. The GOAC Approach

In GOAC, the deliberative layer generated plans of actions using a timeline-based planner. In Timeline-based planning, systems are modelled as a series of interrelated finite state machines. For example, we can have a state machine representing the possible states and transitions of the pan and tilt unit (PTU), another one for the camera or the navigation of the robot, and so on. A timeline is defined as the temporal evolution of each of them, i.e. the temporal sequence of states holding for it. Valid transitions between states in a timeline are usually temporally related to states holding in other timelines by means of Allen's temporal relations. As an example, the *PTU* timeline cannot transit from its *idle* to its *pointing-at* state unless the *navigation* timeline is at the *stopped* state, to avoid the PTU moving while the system is navigating. Or the *communication* timeline can only transit to its *communicating-picture* state after the *camera* timeline has passed by its *taking-picture* state. Timeline-based planning is founded on two key concepts: *tokens* and *temporal flexibility*. A token is any of the possible states a given timeline can be at a certain temporal point. Goals are given as future values of the timelines, or in other words, tokens we want to hold at specific future times. The planner has to find a valid sequence of transitions over states in the timelines, which, starting from the initial states of all the timelines, allows the goal tokens to hold at their desired times. A plan describes the sequence of states all the timelines must pass by until goals are achieved, together with the times these states must be achieved and their temporal durations. Plans are flexible, that is, the time boundaries (start, end, and duration) of planned tasks are not fixed, but are expressed as temporal intervals with lower and upper bounds. An example of an element of the plan would be *camera.taking-picture at [0,100] [1,200] [1,199]*, meaning that the camera timeline must be at its taking-picture state starting in any moment between 0 and 100, ending in any moment between 1 and 200, and having a duration in the range 1 to 199. Plans created following this approach are more robust in uncertain environmental conditions than predefined and rigid sequences of activities.

Since the GOAC controller is based on timelines, and this capability remains unchanged within the GOTCHA controller, a key restriction of GOTCHA is that the planner must generate temporal plans in a timeline-based format. If using any other planning paradigm the produced plans must be mapped to the proper values for timelines that uses the GOAC executive, making possible interleaving planning with execution.

3.2. The GOTCHA approach

Action-based planning, more commonly known as Classical Planning [6] offers an alternative way of modelling a system from an Automated Planning perspective. Although not as popular as timeline-based planning for space applications, it was the first planning approach applied to a real robot [6] and is now being widely used in robotics applications [7][8][9]. Instead of modelling a system as a series of interrelated state machines, this planning paradigm models the system by the actions it can perform. Using the standard Planning Domain Description Language (PDDL) [10] a durative action a can be described using a tuple $\{Pre_a, Eff_a, Dur_a\}$, where Pre_a defines the conditions under which the action can be applied, Eff_a describes the changes the action produces in the state when it is executed, and Dur_a is the temporal duration of the action. Goals are expressed as a future desirable state, or more commonly, sub-state. A plan is a list of actions achieving the goals from the initial state, together with their earliest starting times. PDDL uses predicate logic and divides the description of the planning task into two different files: the domain file contains the description of the actions and the states the system can be, while the problem file contains the description of the initial state and the goals.

Although GOTCHA uses an action-based planner, it provides an integrated model for planning and scheduling: internally GOTCHA uses timelines to communicate among the different control loops (reactors) that are running during execution. Therefore the plan, initially conceived as a set of actions by the action-based planner, is converted into an equivalent set of timelines that are used by the executive. This is performed by a single component, the mission planner reactor that, as in the case of GOAC, has a tight integration with the executive.

4. THE PLANNING REACTOR

The Planning Reactor is the centre of the deliberative layer. It consists of a novel mission planner that combines classical planning and timeline-based planning. As other T-REX reactors, it communicates with the other reactors through the T-REX Agent interface. This reactor receives high-level goals and is in charge of decomposing them into lower level actions that are posted to other reactors. The main processes carried out by the Planning Reactor are deliberation of a plan for achieving high-level goals, the dispatching of low-level actions which will be executed by other reactors, and the monitoring of the execution of the plan through a synchronization with the external observations. Given a PDDL domain and the initial state, this layer transforms the high-level goals, received as tokens (values for timelines) from ground or even other reactors (for instance, a scientific agent) into

PDDL predicates, which are then used to generate a PDDL problem. Then, it creates a plan that achieves such goals. In order to do that, it makes use of OPTIC [11], a PDDL-based temporal planner capable of dealing with temporal problems and goals with preferences. This plan is then translated into a timeline-based plan using a simple temporal network (STN). The resulting plan is used to send actions to other reactors through the T-REX Agent at their earliest start time (assuming everything works as expected). The next subsections describe this layer in detail.

4.1. The PDDL domain

The domain specifies the types of objects that are relevant for reasoning, their properties and the relations among them (both as predicates that can be true or false) and the actions the system can perform. In other words, in the domain it is needed to describe the state (objects, properties and relations) and the actions so a plan can be found in such a way that it can be later translated to a timelines format.

Currently the objects considered for reasoning are the waypoints, the PTU angles and the id of the files used to communicate images.

To represent the different states, the PDDL predicates in our model can be of any of the following types:

- Internal predicates: They encode the states of the Planner Reactor's internal timelines. To distinguish them, they have the "internal_" prefix tag.
- External Predicates: They encode the states of external timelines (e.g. PTU, camera). These predicates have the "state_" prefix tag.
- Regular Predicates: They are the remaining predicates, not falling in the previous categories. They are used mainly to keep record of already visited states or to keep causality relations between different actions.

The current domain contains the following actions:

- Perform Experiment: it is a wrapper action that encompasses all lower-level actions that allow the achievement of a high-level goal.
- Move Robot base: it moves the robot from an origin to a destination. The path to follow and the time it takes are calculated externally (see 4.6).
- Move PTU: it moves the pan and tilt unit to a desired position.
- Take Picture: it captures a snapshot with the onboard camera.
- Communicate: represents the action of sending a previously recorded data file.

Depending on the evolution and the needs of the project, the domain could contain additional actions, for instance a drill action, for performing such action when the rover will be able to accomplish it.

4.2. Automatic generation of PDDL problems

As seen, to be able to reason, a PDDL planner needs the domain and the problem files. The problem includes the initial state and the goals to be achieved. However, new goals might arrive dynamically, in which case only the initial state of the system remains valid. In consequence, it is required to generate online a PDDL problem as new goals arrive. The way that we do this is by extracting all relevant objects and their predicates from the initial state and the incoming goals. Instead of including all the known objects and predicates into the problem, we just add those that are either related to the initial state or the goals[1]. To illustrate, assume the grid example in Fig. 1, where a robot is initially located at coordinates 0-0, *(state_robotBase_at w0-0)*, and its PTU is pointing at angle 0-0 *(state_ptu_pointingat h0-0)*. Fig. 2 shows a fragment of the initial PDDL problem before any goal is received, where only initial conditions are defined. There are two dynamic goals, G_1 and G_2, which are (*mission_communicating f1 w4-5 h35-45*) and (*mission_communicating f2 w5-2 h55-45*) respectively, meaning that two images must be taken, at waypoints (4,5) and (5,2), with headings 35º-45º and 55º-45º, and communicated to ground control with file ids f1 and f2, respectively. From the initial state and the goals we can extract the objects - waypoints, heading, and files - that are relevant for our problem. By doing this, we abstract away all the details that are not needed for that particular problem, as other waypoints, headings or files that could exist, and only represent the relevant ones.

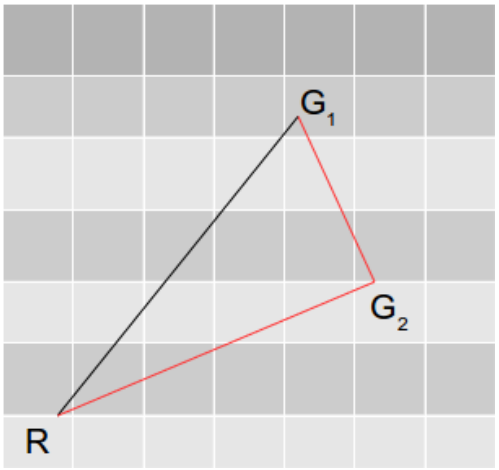

Figure 1: A rover has to achieve G1 and G2 (7x7 grid).

[1] Notice that this is totally domain-dependent, although some automatic approaches will be studied in the future.

```
(:objects
       w0-0 - waypoint
       h0-0 - heading)
(:init
       (state_robotBase_At w0_0)
       (state_ptu_pointingAt h0-0)
       (state_camera_idle)
       (state_communication_idle)
       (internal_mission_idle)
       (communicationvw_visible))(:   goal
)
```

Figure 2: A fragment of PDDL problem with initial conditions defined.

Fig. 3 shows the final PDDL problem for the current example, which includes the relevant objects, their relationships in the form of predicates and the goals. Notice that in this case after receiving the goals, objects for the new waypoints, headings and file ids have been created. Also numeric predicates (known as *fluents*) describing the time to navigate between the waypoints have been added. In this case the value of the fluents will be calculated externally (see 4.6).

```
(:objects
       w0-0 w4-5 w5-2 - waypoint
       f1 f2 - file_id
       h0-0 h35-45 h55-45 - heading)
(:init
       (state_robotBase_At w0_0)
       (state_ptu_poinyingat h0-0)
       (state_camera_idle)
       (state_communication_idle)
       (internal_mission_idle)
       (communicationvw_visible)
       (=(ef-pathplanner-time w0-0 w4-5) x)
       (=(ef-pathplanner-time w4-5 w0-0) x)
       (=(ef-pathplanner-time w0-0 w5-2) x)
       (=(ef-pathplanner-time w5-2 w0-0) x)
       (=(ef-pathplanner-time w4-5 w5-2) x)
       (=(ef-pathplanner-time w5-2 w4-5) x)
)
(: goal (and
(mission_communicating f1 w4-5 h35-45)
(mission_communicating f2 w5-2 h55-45)))
```

Figure 3: A fragment of the final PDDL problem that includes relevant objects, numeric state variables, and goals. New elements added after the goals are received are marked in bold.

The online generation of PDDL problems provides flexibility and reduces the complexity of the problems since it only considers relevant features. Usually, a PDDL problem definition requires including all the details of the problem the planner needs to solve it. However, the larger the domain the poorer the performance of the planner. Therefore, as a consequence of the online generation of PDDL problems, the scalability of the planner improves in large domains.

4.3. Preferences

As previously mentioned, GOTCHA handles preferences. This means that, goals may have different levels of priority depending on how important achieving them is. A goal with a priority equal or higher than 10 is called *hard goal* and must be included in the plan. A goal with a priority equal or lower than 9 is called *soft goal* and could be included in the plan or not, depending on the availability of resources (battery, time…) to achieve it. An incoming dynamic goal has therefore a priority that needs to be considered in the PDDL problem definition in order to provide the planner this information. The way that PDDL handles preferences is by using *penalties*, which can be seen as the cost it would be paid if the goal is not reached by the plan solution. That way, not achieving soft goals implies a higher plan cost, so we will ask the planner to minimize the cost of the plan found, which will lead it to include as many soft goals as possible. Given a goal g with priority p, we define the penalty of g as:

$$penalty(g) = max\_time + 50 * p$$

where *max_time* is the latest end time among all the pending (to achieve) goals. By computing the penalty in this way, we assure that the penalty that the plan does not reach a goal is higher than the total time of the plan, and that the planner does not discard any achievable soft goal.

Continuing with our example, Fig. 4 shows the definition of preferences in PDDL assuming that *max_time* is equal to 100, G_1 is a hard goal, and G_2 is a soft goal with priority 8 (a penalty of 500). The planner takes into account these penalties and tries to find a plan that minimizes the metric defined in the PDDL problem.

```
(: goal (and
(mission_communicating f1 w4-5 h35-45)
(preference g0
(mission_communicating f2 w5-2 h55-45)))
(: metric minimize
        (+ (total-time)
           (* (is-violated g0) 500)))
```

Figure 4: Example of the definition of preferences in a PDDL problem.

4.4. Plan generation and conversion to timeline format

Once the PDDL problem has been created, it is sent, along with the domain, to the planner. The planner returns a temporal plan where each action of the plan is tagged with its earliest starting time. To add flexibility to the time each action is executed, we used the time points proposed by the plan together with the latest time in which a goal should be achieved to build a simple temporal network (STN). The solution to the STN is computed with the Floyd-Warshall algorithm [12]. This

solution provides time intervals that are used as time boundaries for executing timeline transitions.

4.5. Goal dispatching and synchronization

The goal dispatching in the Planning Reactor is the process of determining if there is any transition in the current plan that should occur in the current execution time. If such transition exists and involves any external timeline, the planner reactor will post through the agent the T-REX tokens containing the goal state that these timelines need to achieve in a given time interval.

The process of synchronization inside the Planning Reactor has the objective of having a consistent view of the real state of the scenario. In general, observations dominate planning expectations, therefore this process is in charge of checking (1) whether the observations received from external timelines match the expected next transition of the plan, and (2) whether the current state of any external timeline has exceed the time boundaries. A discrepancy in the first case occurs when there is an unexpected observation, e.g., the rover reaches a waypoint different from the one requested. The synchronization occurs in every tick, therefore is easy to verify if there is a discrepancy for the second case, e.g., no observation is being received before the time boundary for the next transition. In both failure cases the Planning Reactor needs to replan.

When re-planning occurs, the Planning Reactor must restore the real state of the scenario. The actual real state will consist of the expected state, updated with the observations received. To illustrate, assume that a robot performs a *GoingTo(w0-0,w4-1)* action; the expected state would be *At(w4-1)* along with all the other external timelines (e.g.: PTU, camera, etc.) that remain unchanged when the action is performed. If the observation received from the *Agent* is *At(w4-0)*, the current real state would consist of all the external predicates in the expected state, but with *At(w4-1)* updated to *At(w4-0).*

In addition to the definition of the current real state, the Planning Reactor must update the list of goals to achieve; those goals that have been already achieved would be deleted from the list of goals. Once both real state of scenario and goals are updated, the Planning Reactor creates a new problem (as explained in 4.2) and finds a plan (as explained in 4.4).

4.6. Planner Enhancements

One of the main concepts of GOTCHA related to planning, inherited from QuijoteExpress [13] is the use of external functions: planning generation is not anymore an exclusive task of the mission planner, but a collaborative one in which the mission planner query external expert systems about specific aspects of the plan. For example, in a Mars rover scenario, the mission planner could ask the path planner for an estimation of the time and energy required to do a traverse, taking this information into account in order to generate a more accurate plan. To guarantee that GOTCHA can be easily adapted to different types of domains, the mission planner offers a standard interface, named *external function interface* (see box below); in order to connect an expert system to the mission planner, it just need to implement this interface

External Function Interface

```
class ExternalFunction {
public:
virtual                    std::vector<NumericVariable>
evaluateFunction(std::vector<PddlArg> parameters) = 0
```

In addition to the implementation of the interface, the activities in the domain requiring information from an external function need to add a reference to such function. In the following example, the *goingto* action relies on *ef-pathplanner* function to compute the time and battery required for a traverse between two waypoints.

External Function in Domain

```
(:durative-action robotbase_goingto
:parameters (…)
:duration (= ?duration (ef-pathplanner-time ?wpfrom
?wpto) )
:condition (and
   (at start (> (amount-available ?bat) (ef-pathplanner-
battery ?wpfrom ?wpto)))
…
)
:effect (and
 (at start (decrease (amount-available ?bat) (ef-
pathplanner-battery ?wpfrom ?wpto) ))
…
))
```

The following references are required: the variable duration receives the value estimated by the function; a precondition indicating that the amount of battery available needs to be bigger than the estimated battery required for the activity; finally, an effect to decrease the battery in the amount estimated by the function. In the following example, it is indicated that the pathplanner time between waypoints w0-0 and w4-1 must be computed:

External Function in Problem

```
(= (ef-pathplanner-time w0-0 w4-1) x)
```

Another important feature of the planner is the capability to represent hierarchical models. The model is organized in a top-down hierarchy, so that the missions can be exclusively defined in terms of high-

level goals, leaving the low-level details to the model. As a result, routine operations of the mission are greatly simplified. This type of representation is achieved in GOTCHA in a transparent manner, without the need of additional source code, thanks to PDDL. A high level goal h_g achieved by a number of low level actions h_{l1}, ..., h_{ln}, contains at least n conditions that are satisfied by h_{l1},... h_{ln}.

```
(:durative-action complexgoal
        :condition (and
                (at start (state_subgoal))
        ...
)

(:durative-action subgoal
        :condition (…)
        :effect (and
                (at start (state_subgoal))
                ...
```

One final concept derived from QuijoteExpress is *sufficient planning*. The idea is to allow the generation of valid *partial* plans in conditions where key information is missing. For example, imagine that a mars rover needs to do three activities: a traverse to waypoint wp1, some science, a second traverse to wp2 and further science. Imagine that the first traverse (within the field of view of the robot) and both scientific activities are highly deterministic, while the second traverse (beyond the field of view), cannot be determined. With sufficient planning, the second traverse only receives an estimation of the resources and time required, but is not actually planned in detail. It is also marked with a flag to remind the planner that it must be re-planned prior to its execution. If no solution is found, even for the most optimistic estimation, then we know that all the goals cannot be achieved. Otherwise, the execution can start, and the second traverse will be refined at due time.

5. INTERFACES

5.1. Interfaces to the functional layer

Being conceived as a generic template, the GOTCHA system will have to interface with many different robotic assets. Nowadays, one of the main RCOS that provides support for many different robotic platforms is ROS. Therefore one of the main works that have been performed has been the integration of the functional layer with the GOTCHA agent. A series of "ROS Adapters" were developed. These ROS Adapters link ROS Actions and ROS States with the timelines used by the agent, the following timelines are envisaged:

- *RobotBase*: with states *Idle* and *GoingTo(x,y)*
- *Camera*: with two possible states: *Idle* and *TakingPictureAt(x,y)*
- *PTU (Pan Tilt Unit)*: With states *Idle* and *PointingTo(x,y)*
- *CommunicationTimeline*, with two possible states: *idle* and *communicating*

Readers can easily identify that there is a one-to-one mapping between the actions as defined in the PDDL domain and a given state of a timeline.

When the plan has a new action to be executed (like f.i. the rover to go to a given position), a *ROS Adapter* is used to trigger the corresponding ROS action, that performs the movement of the robot. This ROS Adapter is a SW component that acts as the interface with the functional layer

In addition, reactors can subscribe to *ROS States* to update incoming observations from the robot sensors. The work on ROS Adapters was based on the previous work done by GMV in the frame of the PERIGEO project [14]

The development of this interface based on ROS actions and states opens the door to access multiple commercial robots, and eases the instantiation of the controller to multiple commercial robotic assets supported by ROS in the laboratory environment. As part of the project activities, a testing environment using the GOTCHA controller in combination with the gazebo simulator and the ROS framework has been setup.

In order to target the space domain, a further activity that is planned is to adapt the functional layer of the RAT-LRM [15] (currently using ROS) to the TASTE framework [16], by wrapping its components into TASTE functions.

5.2. Interfaces with ground

The interfaces to ground are heavily based on telecommand files. The reason for using files is that, it is assumed that for a planetary rover, the high latency and the existence of communication windows will make impossible direct telecommanding. Commanding will then be performing by uploading files, each file containing a given set of telecommands. Once successfully uploaded, these commands will be executed by the agent. Therefore commanding at the rover will be performed following a transactional scheme: either all the telecommands inside a file received are executed or no telecommand from the file is executed. The last case can eventually happen if the upload is not successful (due to communication problems, lack of on-board space, etc.).
A single file is used to define the level of autonomy of

the system. The levels of autonomy correspond to the ones defined in the ECSS standards [17] (ranging from E1 to E4). Different files are to be used for each of the commanding at each autonomy level, with a different syntax. An example of the TCs at level 1 is shown in the following figure:

Tc-ID	Description	Parameters	Functional Layer Equivalence
CINIT	Initialise robot	N/A	InitialiseRobot
CMOV	Go to a given absolute position	X,Y	MoveRobot X,Y
CPTU	Move the PTU to a given direction	Pan, Tilt	MovePTU Pan Tilt
CCOM	Send a picture file to ground	Id	Communicate Id
CCAM	Take Snapshot	Id	Camera Id
CLSTP	Stop Robot		StopRobot

Figure 5: Low-level (E1) telecommands

6. VALIDATION

Although GOTCHA is conceived as a generic robotic controller, able to be used in many different space robotic assets, the main use case we are targeting in the frame of the project is a planetary exploration rover. Two different scenarios are envisaged: the virtual reference scenario and field tests.

Virtual tests will be performed by executing a set of test cases using the RAT VFS [15], in charge of controlling the virtual LRM and providing a TM/TC interface to the Ground Segment, with the aim of demonstrating the feasibility of the deliberative and executive layers against a robotics simulator.

The test scenarios to be run in the field trials are aimed to assess the fulfilment of the requirements and also to characterize the performance of the system in different situations, so for instance at E4 planning level [17] the system will plan a set of high level goals, perform both a nominal execution, execution in which unexpected events during the plan execution will lead to re-planning, and dynamic injection during execution of new goals, that will enforce the system to re-plan. Execution for the lowest levels of execution (E1, E2, E3) [17] will be also oriented to tackle nominal execution and execution under failures. In addition, some tests will be oriented to inject errors that will lead to an automatic decrease of the level of autonomy due to the impossibility to re-plan.

7. CONCLUSIONS

The main sources of novelty within the GOTCHA with respect to the GOAC system are the following

- GOTCHA uses a PDDL action-based planner. Action-based planners model a system by the actions it is able to perform (see section 3.1). PDDL (reference) is a standard planning modelling language (Planning Domain Description Language). Some advantages of this approach are (1) modelling: action-based models are generally easier to define than their timeline-based counterparts, (2) standard: the use of the current standard for modelling planning tasks, PDDL, facilitates the improvement of the GOTCHA system when new advances in automated planning technology appear, and (3) performance: PDDL planners employ powerful domain-independent heuristics that allow planners to improve the system's performance in comparison to that of their timeline-based counterparts. This is a crucial factor in space scenarios due to the scarce on-board computer resources.

- It provides an integrated model for planning and scheduling: internally GOTCHA uses timelines to communicate among different control loops in the execution. Therefore the plan, initially conceived as a set of actions by the action-based planner, is converted into an equivalent set of timelines for execution. This is performed by a single component, the mission planner reactor that, as in the case of GOAC, has a tight integration with the executive. The mission planner has been designed to collaborate closely with the executive layer and any other decision-level component (guidance system, arm motion planner, etc.) taking advantage of a well-defined set of interfaces, based on timelines. These specialised planners will provide relevant information in terms of duration of activities and resource consumption to the mission planner in order to generate a more accurate plan.

- Dynamic reasoning on goals: the planner takes into account new goals as they arrive or discards goals that are no longer needed when generating new plans or when re-planning. Dynamic goal reasoning will allow the system to increase its level of autonomy in case humans so decide.

- Handling of preferences: the system is able to work with goals with different levels of priority, trying to accommodate all of them in the plan if possible, or automatically discarding the lowest priority ones if not.

- As part of the GOTCHA activities, an interface has been developed to the ROS ecosystem. This will pave the way for a future instantiation of the system in robotic assets supported by ROS.

Future activities include the embedding of the functional layer components under the TASTE framework as well as to investigate the use of TASTE as a correct-by-design model-driven environment allowing designers to wrap the functional capabilities of the robot.

8. REFERENCES

1. **Ceballos A, Bensalem S, Cesta A, et al.** "*A goal-oriented autonomous controller for space exploration"*. Proceedings of the 11th symposium on advanced space technologies in robotics and automation, Noordwijk, Netherlands, 12–14 April 2011.
2. **McGann, C., Py F., Rajan, K., Ryan, J. & Henthorn, R. (2008)** *"Adaptive Control for Autonomous Underwater Vehicles"* AAAI'08 Proceedings of the 23rd national conference on Artificial intelligence - Volume 3 Pages 1319-1324
3. **N. Muscettola et al.** "*IDEA: Planning at the Core of Autonomous Reactive Agents".* Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space American Association for Artificial Intelligence, 2002
4. **E. Gat,**et el "*Three-Layer Architectures"*, In: Artificial Intelligence and Mobile Robots", pages 195-210 MIT/AAAI Press (http://robotics.usc.edu/~maja/teaching/cs584/papers/tla.pdf).
5. **Py, F., Rajan, K., McGann, C**. *A Systematic Agent Framework for Situated Autonomous Systems* AAMAS'2010 Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 2 Pages 583-590
6. **R.E. Fikes and N.J. Nilsson**. *STRIPS: A new approach to the application of theorem proving to problem solving.* Artificial Intelligence, 2(3-4):189-208, 1971.
7. **Quintero, E., García-Olaya, Á., Borrajo, D., & Fernández, F.** (2011). *Control of Autonomous Mobile Robots with Automated Planning*. Journal of Physical Agents 5(1):3-13.
8. **Bernardini, S., Fox, M., & Long, D.** (2014). *Planning the Behaviour of Low-Cost Quadcopters for Surveillance Missions*. In Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS) 445-453.
9. **González, J.C., Pulido, J.C. & Fernández, F**. (2017). *A three-layer planning architecture for the autonomous control of rehabilitation therapies based on social robots:*, Cognitive Systems Research (CSR) 43:232-249.
10. **Fox, M., & Long, D.** (2003). *PDDL2.1: an Extension to PDDL for Expressing Temporal Planning Domains.* Journal of Artificial Intelligence Research (JAIR), 20(1), 61-124.
11. **Benton, J., Coles, A., & Coles, A.** (2012). *Temporal Planning with Preferences and Time-Dependent Continuous Costs*. In Proceedings of the 22nd International Conference on Automated Planning and Scheduling, 2–10.
12. **Dechter, R., Meiri, I. & Pearl, J**. (1991). *Temporal constraint networks*. Artificial intelligence, 49(1-3):61-95.
13. **J. M. Delfa Victoria, N. Policella, Y. Gao, and O. V. Stryk.** "*Quijoteexpress - A novel Apsi planning system for future space robotic missions*". In ASTRA, 2013
14. PERIGEO – Spanish National Project IPT-20111022 "Aerial platform's for research and orbital testing". 2015
15. **Medina, A,, Mollinedo, L., Kapellos, K.,, Crespo, C., Poulakis, P**. *"Design and realization of a rover autonomy testbed* "ASTRA conference, 2015
16. **Perrotin, M., Conquet, E., Dissaux, P., Tsiodras, T., Hugues, H.** (2010), *The TASTE Toolset: turning human designed heterogeneous systems into computer built homogeneous software*. In: ERTS 2010, Toulouse (2010)
17. ECSS Secretariat. (ESA/ESTEC), "*ECSS-E-70-11 Space Segment Operability*" (August, 2005)

ACKNOWLEDGEMENTS

We would like to thank the ESA / ESTEC and in particular to those members of ESA that are supervising the project (M. Van Winnendael, M. Azkárate and J. Ramachandran) for their support and guidance in the development of this activity