PLANNING-BASED CONTROLLERS FOR INCREASED LEVELS OF AUTONOMOUS OPERATIONS

Simone Fratini, Sebastian Martin, Nicola Policella, and Alessandro Donati

European Space Agency, ESA-ESOC
Robert-Bosch-Strasse 5, 64293, Darmstadt, Germany
name.lastname@esa.int

ABSTRACT

Upcoming missions will require a higher degree of remote operations to increase quality and quantity of science return. Remote operations are certainly a challenging scenario, mainly because communication delays and errors lead to objective difficulties for the operator to promptly enable opportunistic science, activate contingency recovery procedures or to anticipate and react to tracked events. AI planning-based control layers have demonstrated to be able to entail autonomy, but modeling still constitute a bottleneck for the use of these technologies. In this paper we discuss how to design a controller based on the ESA GOAC platform to robustify the line operator - communication layer - remote system in the context of remote operations, with particular attention on discussing a general methodology to design efficient models for the AI planning engine. The approach presented has been tested on a real system made of a communication network, to simulate the communication issues, and a rover, as controlled device. The communication system and rover used are part of the ESA METERON infrastructure (Multi-Purpose End-To-End Robotic Operations Network).

Key words: Control, Autonomy, AI, Teleoperations.

1. INTRODUCTION

Upcoming missions either targeting earth observation, planets or deep space, will require a higher degree of onboard autonomy operations to increase quality and quantity science return, to minimize close-loop space-ground decision making, and to enable new operational scenarios. Autonomy operations will induce a new distribution of processes and functions between ground and space; on-ground routine operations workload will be reduced and refocused on supervisory role while operator intervention will be required only for non-nominal situations.

In this context remote operations are certainly an interesting scenario where autonomy can provide added value. In fact, communications, in remote operations, are affected by non-continuous communication and by relevant radio communication propagation delay and transmission errors, with subsequent difficulties for the operator to promptly enable opportunistic science, activate contingency recovery procedures or to anticipate and react to tracked events.

In this paper we propose the use of a domain independent, model-based control layer between the operator and the system to be operated, in order to robustify the line operator - communication layer - remote system in two ways: on one side the controller can be located at the operator side (see Figure 1, left), providing a unified view of the communication layer plus remote system that allows the operator to reduce the impact of the delay and errors introduced by the communication channel. On the other side the controller can be located at the remote system side, rising the abstraction level of the platform commanding and providing to the operator the view of a more robust autonomous, goal oriented system, that reduces the amount of controls to be transmitted and reduce the need of fine grained synchronization of commands for the remote system (Figure 1, right).

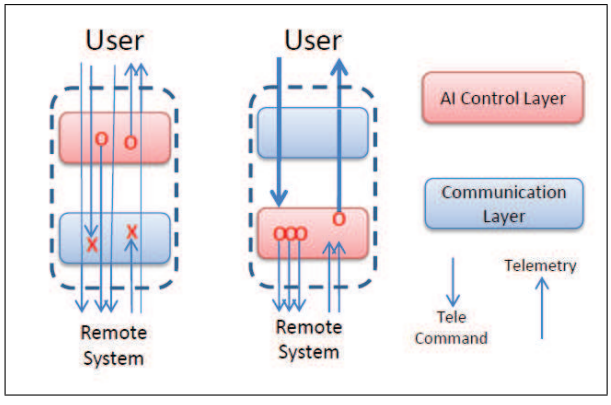

Figure 1. Conceptual Architecture

The AI controller has been implemented by means of an ESA platform deployed for the "Goal Oriented Autonomous Controller" (GOAC [2]) initiative. The GOAC platform entails rapid prototyping of model-based controllers for complex physical systems exploiting AI planning technologies. The platform takes in input a model of the system to be controlled and a set of user goal to be (hopefully) achieved by the controlled system. The controller plans for user goals and supervises plan execution by ingesting the telemetry of the controlled system. The controller is based on a domain independent planner, entailing the reusability of the same technology in differ-

ent domains (by changing only the model). As described in the following sections, we have designed two GOAC-based controllers, one to be used upstream of the communication channel and one to be used downstream. The technology is the same, but the models and the roles of the model change. Besides all the obvious advantages of using model-based technologies (the high level of code reusabilty above all), there is usually a common problem: the difficulty of modeling. This paper aims also at discussing this problem from a methodological point of view. In this sense the objective of this paper with respect to modeling, is not to present complex models executable with the GOAC platform, but to discuss *how* to build a model to cope with some common problems in remote operations.

The proposed approach has been tested on a real system made of a communication network, to simulate the communication issues, and a rover, as controlled device. The communication system and rover used are part of the METERON infrastructure (Multi-Purpose End-To-End Robotic Operations Network), one of the projects established within ESA to test networking protocols and operations for future human exploration scenarios to moon, mars and other celestial bodies[1]. It contains a sophisticated simulator and operational network for in-orbit testing and validation of novel communication techniques under consideration for future human exploration, such as Disruption Tolerant Network (DTN)[3] concepts and technologies (Communications).

To generate more meaningful telemetry and telecommand streams, a rover called MOCUP is part of this network. Via its user interface software named MOPS, it allows simple operations such as waypoint navigation, execution of command stacks, capturing, transmission and display of pictures as well as transmission of status telemetry. This system was used successfully for the METERON OPSCOM-1 experiment of October 23, 2012, where an astronaut onboard the ISS controlled the MOCUP rover on ground in ESOC, Germany over the DTN network. With re-using this infrastructure and hardware, the AI controller layer proposed in this paper has already been validated in a realistic environment.

2. THE AI CONTROL LAYER

The AI control layer is an instantiation of the GOAC platform. The GOAC platform is the result of the ESA Goal Oriented Autonomous Controller (GOAC) initiative, aimed at defining a new generation of software autonomous controllers to support increasing levels of autonomy for robotic task achievement. In particular, goal of the GOAC architecture is to generate on-board plans, dispatch activities for execution, and recover from off-nominal conditions. GOAC has been designed as a principled integration of different software solutions (see [2, 6]) to be a *goal-oriented* system embedding automated model-based planning and execution. GOAC encapsulates the long-standing notion of a *sense-deliberate-act* cycle, where sensing, planning and execution are interleaved. The architecture consists of a set of *reactors*, each of them encapsulating an independent control loop. There is a well-defined messaging protocol to synchronize the independent control loops: exchanging *facts* and *goals* between reactors. Each reactor ingest *observations* of the current state either from the platform or other reactors, and send *goals* to be accomplished (to the platform or to the other reactors). The whole control loop is then split into subproblems, following a "divide-and-conquer" approach to complexity that simplifies the scalability and increases the flexibility of the architecture (see [8] for a comprehensive discussion of the principles behind the executive engine of GOAC).

The number and hierarchy of reactors to be instantiated is a problem-specific design decision. Goals follow a top-down flow, from an abstract mission level reactor (close to the user), to more specific reactors, down to a "command-dispatcher" reactor (close to the platform), which translates goals into requests to the functional layer of the controlled system. Observations flow in a bottom-up direction. At the bottom of the hierarchy, the command dispatcher reactor synthesizes observations from replies sent by the functional layer, and forwards these to reactors placed at higher levels of abstraction, which in turn generate more abstract observations to state the level of accomplishment of the received goals. Hence, the goals in input to the highest level reactor are the user goals for the controller, the goals sent out by the lowest level reactor (the command dispatcher) are commands for the platform. Vice-versa, the observations in input to the command dispatcher constitute the telemetry from the platform, while observations sent out by the highest level reactor constitute the status of accomplishment of the user's goals (see Figure 2).

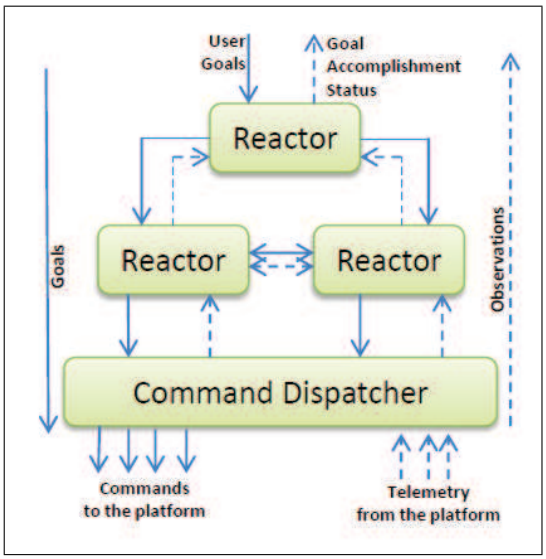

Figure 2. Reactors

The flow of observations and goals among the reactors implement, respectively, the sense and the act steps of the loop. The planning step is a model-based process based on *timeline* planning. Each reactor embeds a timeline-based planner with its own model. The model comprises a set of state variables, and describes the allowed transitions as well as a set of synchronization rules. Synchronization rules describe causal and temporal relationships between state variables (see for instance [7, 4, 5] about timeline-based modeling and planning). The advantage

of using a domain-independent planning technology is at hand: different reactors differ only on the model, while the software is reused across the different reactors. The only exception is the command dispatcher reactor: this reactor does not do planning but conversion of timelines into commands for the functional layer and the other way around (telemetry from the platform into timelines). The reactors that actually implement a planning loop are defined *deliberative*.

Hence the tailoring of the GOAC platform consists in: (1) choosing the number and hierarchy of the reactors for the specific problem; (2) design the model for each deliberative reactor and (3) implement a command dispatcher for the connection between the platform and the functional layer of the system to be controlled. The number and hierarchy of the reactors depends on the level of autonomy that the controller has to provide to the platform and the complexity of the model. A generic controller based on the GOAC platform aimed at providing autonomy requires at least one deliberative reactor (for planning and re-planning) in cascade with a command dispatcher (to connect the planner to the platform).

The need of managing goals at different levels of abstraction and different temporal scopes requires to increase the depth of the reactors graph. Two reactors in cascade for instance allows an HTN approach dividing the model of abstract, flexible high level goals (to be refined into lower-levels goals and planned/optimized over a wide temporal horizon) from the modeling of more concrete, better specified (in time and value) lower-levels goals (to be planned and monitored in a smaller temporal horizon). The model for high level goals is more complex from the planning point of view but leads to more stable plan (which does not need to be re-planned frequently) while the model for low-level goals is simpler and allows fast re-planning for more frequent failures. Hence to increase the depth of the reactors graph entails the possibility of designing models with different levels of abstraction.

The need for modeling sub-problems with different functional scopes requires to increase the width of the reactors graph. Two reactors in parallel allows modeling for multiple planning and execution monitoring processes to be carried on in parallel. Let's take for instance the example of complex platform payloads. Each payload needs to be operated in parallel with occasional logical and temporal synchronizations. A model for managing all of them would become quickly very complex, with no real need of modeling all the logic in the same model. Multiple reactors in parallel simplify drastically the model and allows synchronizations based on the post goal/receive observation interaction provided natively by the platform. Hence to increase the width of the reactors graph entails the possibility of designing models with different functional scopes.

For our purposes we have designed 4 reactors and we have used them (1) in a cascade configuration of 3 reactors for the downstream controller (see Figure 3, top) and (2) in a cascade of 2 reactors for the upstream configuration (Figure 3, bottom). The 4 reactors were: the *Mission Manager*, the *Platform Manager*, the *Communication Repair* and the *MOPS* reactor. The first three reactors are deliberative, the last one is the command dispatcher.

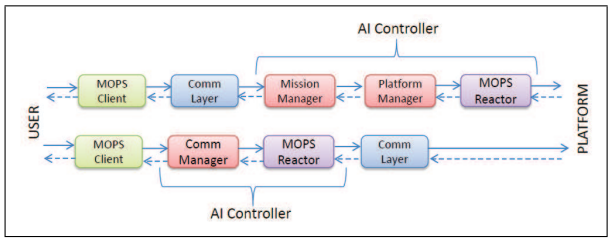

Figure 3. Reactors Architecture

The platform manager reactor aims at modeling a "virtual platform" able to perform simple navigation tasks in a robust manner. The model is designed to be able to re-plan for simple obstacle avoidance and plan autonomously for opportunistic science. The mission manager reactor is designed to plan for complex user tasks, like monitoring an area and taking pictures around interesting objects. The communication repair reactor is designed to robustify the communication between the operator and the remote system. The MOPS reactor encapsulates the connection with the rover trough the network and constitute the logical interface with the platform.

3. DELIBERATIVE REACTORS

The role of deliberative reactors is to plan timelines for achieving goals, monitor the execution status of the timeline and re-plan timelines when problem occurs in the execution. The behavior of a GOAC deliberative reactor depends on the *model* and on various policies. The most interesting ones are the *goal planning policy* (GPP) and the *goal re-planning policy* (GRP) (see [6] for details on how how a GOAC deliberative reactor is implemented and the available policies). The GPP specifies how the planner is feed with goals asynchronously received by the reactor, the GRP specifies what to do with the goals whose plan failed execution. The design of a specific reactor implies (1) the definition of a model for the planner, (2) the choice of a goal planning policy and (3) the choice of the re-planning policy. Following sections describe the objectives, the models and the policies used by the deliberative reactors designed for the experiments described in this paper.

3.1. The Platform Manager Reactor

The platform manager reactor aims at modeling a "virtual platform" able to perform simple tasks in a robust manner. The model is designed to allow the rover to (1) recover from a set of predefined not nominal conditions and (2) plan on-board activities to achieve unexpected sub-goals triggered by the status of the platform. Obstacle avoidance and plan autonomously for opportunistic science are two examples of capabilities entailed by this model.

Regarding the problem of recovering from failures, we consider a generic platform modeled with a timeline TL_P

(platform timeline) with two types of states: *Stable* states S and *Acting* states A, with the goal expressed by specifying the stable status we want to get. The nominal behavior would be a sequence of stable and acting states S $\rightarrow$ A $\rightarrow$ S.... To add the capability of reacting to failure conditions, we add a set of error states E_i (one for for each possible failure) plus a path $E_i \rightarrow P_i \rightarrow$ A of states to go trough to recover the failure (see Figure 4, left). A failure forces an error state on the timeline TL_P that models the platform. The use of a re-planning policy that re-plans for the last goal in execution (a stable status in this case) with this type of model forces, in case of failure, the execution of the recovering procedure for the failure occurred from the error state to the acting states in execution when the failure occurred. Section 5 shows an example of implementation of this pattern to entail simple obstacle avoidance.

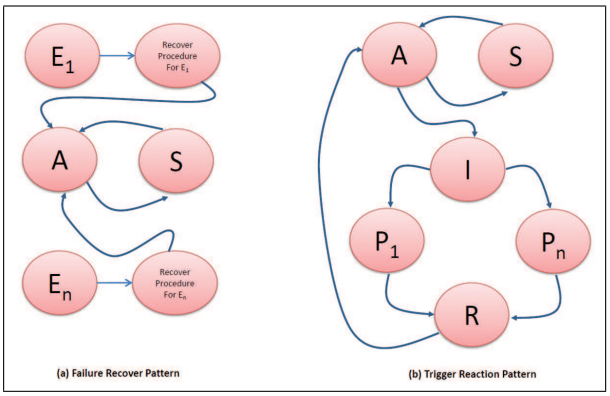

Figure 4. Platform Manager Modeling Patterns

To achieve unexpected sub-goals triggered by the status of the platform (for example to react when an interesting object is found), we need to model at least two different scenarios: a "nominal" one, i.e. what is the configuration of the telemetry expected in default conditions (that does not need a reaction) and (2) one (or more) not nominal scenarios, i.e. what are the configurations of the telemetry that need a reaction and how to react. There is a number of requirements for the plans that can be generated from the model to guarantee the correctness of the behavior that can be obtained executing these plans. First of all a nominal plan has to fail if the telemetry configuration changes during the execution into a status that require a reaction (this is to enforce the reaction, if the envelope for the nominal plan accidentally contains configurations of the telemetry that needs a reaction, the platform might not react when needed). Secondly, a plan to react to a given configuration of the telemetry must fail with any other configuration (included the nominal one), to ensure the right reaction of the platform and to avoid an "over-reaction" (i.e. a reaction in nominal conditions). Finally, any plan for the not nominal conditions must re-generate the initial conditions for the nominal scenario (to handle multiple reactions).

Considering again a generic platform modeled with the timeline TL_P with acting and stable states. We want to be able to react to a set of telemetry values modeled as a set of values obs_i of a timeline TL_T (telemetry timeline). We assume obs_0 as the nominal value for the telemetry (no action is required), while the other values require a reaction (obs_i is supposed to require a sub-plan P_i). To react to a generic trigger obs_i we conceptually add to the state machine modeling the platform the paths A$\rightarrow$I$\rightarrow P_i \rightarrow$R$\rightarrow$A, where the states I (for *Interrupt*) and R (for *Restore*) aim a modeling the procedures to interrupt the current activities and to restore the default status of the platform[1] (see Figure 4, right). Besides that, we synchronize the values of A and P_i with the values of the timeline TL_T that triggers the reactions. In order to guarantee at the modeling level that nominal plan must fail if the telemetry configuration changes during the execution into a status that require a reaction, we force the nominal action A to occur during the value obs_0 of TL. In order to ensure that a plan for a reaction P_i will fail with any other configuration of telemetry than obs_i, P_i will be synchronized to occur during obs_i. Finally, to guarantee that the platform will be able to perform A after P_i, R will be synchronized to restore the value obs_0 at its end (see Figure 5). Section 5 shows an example of implementation of this pattern to entail opportunistic science.

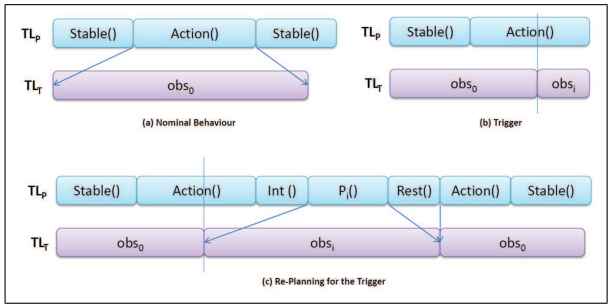

Figure 5. Trigger Synchronization

3.2. The Mission Manager Reactor

The mission manager reactor is designed to manage complex user tasks, like monitoring an area, take stereographic picture of interesting objects or zig-zagging between two locations. What is interesting to discuss beside the model in itself is the interaction between the model of this reactor and the model of the platform reactor described above. Let us suppose R_T the reactor on the top (the mission manager in our case) and R_B the reactor on the bottom (the platform manager in our case). The model of R_T is designed to achieve a set of high level tasks T_i, while R_B has to provide a set of low level functionalities such that T_i can be modeled as a set of them. In order to simplify the model of R_T and wrap the complexity of R_B, instead of thinking R_B as a set of operations it is useful to model it as a set of states that it can achieve (autonomously) and define T_i as a sequence of states to be achieved instead of a sequence of operations to perform.

Hence in general, supposing a model M_B for R_B and a model for M_T for R_T, let T_i the tasks we want to achieve with M_T and S_i the interesting states that can be achieved with M_B, we (1) define a timeline TL_G (G stands for

[1] Depending on the actual behavior of the platform and the reactivity requested, I may not be necessary if the platform autonomously interrupts its activities when launching a trigger or R might be included directly in P_i.

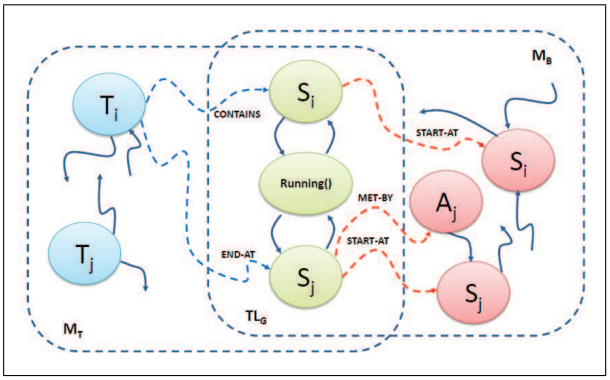

Figure 6. Mission Manager Modeling Pattern

"goal timeline") with n possible states S_i plus the state RUNNING() (the running status wrap the complexity of how actually M_B changes its status); (2) we add TL_G both to M_T and M_B; (3) we add to M_B the proper synchronizations with the corresponding states in the M_B's timelines; a status S_i on TL_G is synchronized to start at the corresponding status of the timeline in M_B (we call this type of goal *status achievement goal*) or to be met by the action that can achieve that status (*action achievement goal*); (4) we add to M_T the proper definitions of T_i in terms of S_i[2] (see Figure 6). Regarding the modeling of T_i in terms of S_i, T_i has to contain all the S_i to be achieved and has to be synchronized to end at the achievement of the last status that has to be met by the sub-plan. Section 5 shows an example of instantiation of this pattern to implement high-level tasks in terms of low-level platform functionalities.

3.3. The Communication Repair Reactor

The GOAC architecture has been designed to support deployment of software autonomous controllers to generate on-board plans and to dispatch activities for execution. As briefly described in Section 2, GOAC is based on a well-defined messaging protocol to synchronize independent control loops: exchanging *facts* and *goals* between reactors. Each reactor ingest *observations* of the current state either from the platform or other reactors, and send *goals* to be accomplished to the platform or to the other reactors. In theory an underlying assumption is that there is no communication errors among the reactors, hence any difference between planned and observed timelines is theoretically due only to unexpected outcomes of planned actions and/or unforeseen events.

In practice there might more reasons for discrepancy among planned and executed timelines (interpreted as failures during execution): errors in the models of the deliberative reactors, errors in the implementation of the command dispatcher and errors in communication among the reactors (lost goals or observations). The GOAC model-based approach is flexible enough to cope with these errors. In fact in principle there is no difference between an error during execution and any other type of error mentioned above, since they all lead to a discrepancy on timelines that entails re-planning. In other word, as we design a model for reacting to unexpected outcomes of planned actions, it is possible to design a model to cope with errors in sending goals and receiving observations.

Let us consider again a generic platform modeled with two types of states: a set of stable states S_i and a set of acting states A_i. We assume for the communication reactor a model conceptually similar to the one described for the reactor on the bottom in Section 3.2 and on the right in Figure 6. Hence we have 2 timelines for the model of the communication repair reactor: a goal timeline TL_G and a platform timeline TL_P. TL_P is a timeline shared with the command dispatcher, and takes values S_i and A_i, while TL_G takes values S_i and RUNNING().

Errors can occur in the flow of goals and observations from and to TL_P. If we consider for TL_P a status S_i and as goal to achieve a status S_j, the transition of the planned timeline $S_i \rightarrow A \rightarrow S_j$ would generate a flow of information among the communication repair reactor and the platform reactor (that in this configuration wraps the communication channel) of 1 goal (A) and 2 observations (A to acknowledge the start of the action and S_j to communicate the end of the action and the new stable status of the platform). Hence we can have 3 types of failures (see Figure 7 (a)): (MG) missed goal, when A is sent but no received, (MA) missed ack for the goal, when A is received, the action starts but the ack that the action has started is missed, and (MS) missed status change, when the action has finished, the new status S_j is sent to the deliberative reactor but is not received. (MG) and (MS) are less critical, because in both cases the real status of the platform is stable and during re-planning no dangerous actions are performed by the platform. (MA) is more critical because the model of the platform is of a status stable while the platform is actually acting.

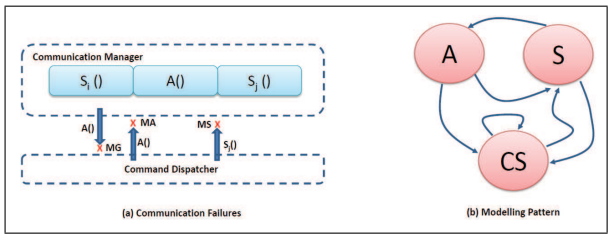

Figure 7. Communication Repair Reactor

We can deal with the problems mentioned above by means of timeouts, a user defined re-planning policy and a slightly modified model for TL_P. Regarding the model for TL_P, we add a status CS (for check status), which effect is to stop the platform (if it is acting) and to force it to provide its current status (see Figure 7 (b)). Regarding timeouts, let us suppose that we can calculate a "reasonable"[3] timeout τ_{out} for the transition $S_i \rightarrow S_j$. If we impose τ_{out} as upper bound for the transition of the platform into S_j, we can have 3 possible failures (see Figure 8): (1)

[2] This pattern works under the assumption that all the S_i used to define a task T_i belongs to the same model M_B . In case of tasks defined in terms of states belonging to different models, more complex synchronization patterns have to be used. The description of these patterns is out of the scope of this paper.

[3] Reasonable here means that we know the platform able to perform the task in nominal conditions in a given time, and we set the timeout a bit higher than that.

at τ_{out} the timeline is still in S_i; (2) a straight transition $S_i \rightarrow S_j$ appears on the timeline; (3) at τ_{out} the timeline is in the status A. The case (2) indicates a communication error MA (missed ack). Obviously the platform has changed status but we missed the ack that the platform was acting. The case (1) can be the result of either a MG or MA error: being apparently the platform still in S_i either it did not move because the goal has not been received or it was actually moving but the ack of the move has been lost and the timeout has been reached. To sort out which is the case, we use a re-planning policy that forces first CS followed by the goal that was originally planned. This has the effect of restoring the status of the timeline allowing a consistent re-planning for the original goal. Regarding the third case, i.e. a value A at the timeout, it can be the result of either a MS error (the platform got the stable status but we haven't received the confirmation) or a generic error (the platform is still acting but not able to reach the goal by the timeout). Also in this case, the user-defined re-planning policy forces a check of the status before starting the re-planning process.

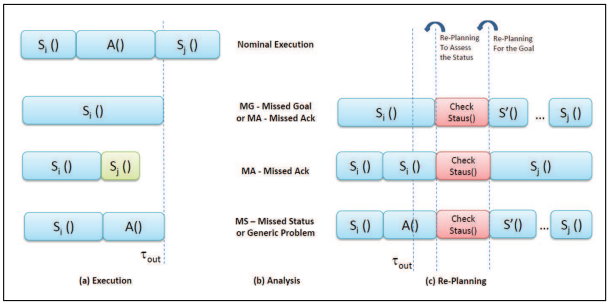

Figure 8. Communication Repair Reactor Failures

The models of deliberative reactors implemented for the experimental evaluation are described in Section 5, while the next Section describes the implementation of the command dispatcher.

4. THE MOPS REACTOR

The connection to the MOCUP Rover platform was highly facilitated due to the MOPS control software and its external interfaces. The MOPS (METERON Operations Software) user interface itself connects to the MOCUP rover via simple String commands. The commands send are headed by an identification number, and replied by the platform with acknowledgment of reception messages, and later by an execution status. In addition, the platform periodically sends status information such as current rover position, ultrasonic distance sensor readings and battery status to the MOPS software.

MOPS, programmed in the python language, has an API allowing for sending all available commands via external applications, as well as to register to the rover command responses and telemetry. This API was used to interface the MOPS reactor with the MOPS software. As the MOPS software can be deployed and connected to MOCUP from remote computers or be run heedlessly directly on the platform itself, the rover can be interfaced by a controller located remotely over the network or by a controller directly deployed on the rover.

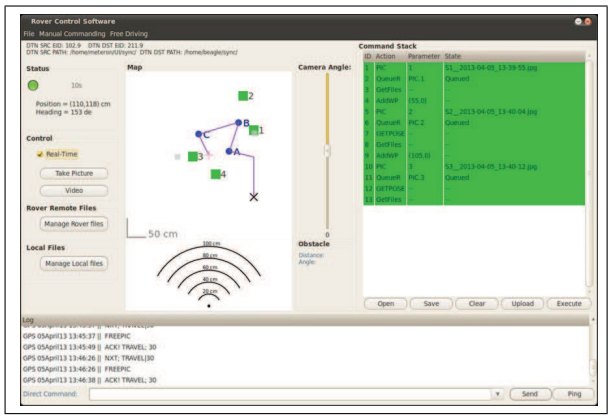

Figure 9. The MOPS user interface for MOCUP rover control

As MOPS/MOCUP allow only sequential execution of commands, the MOPS Reactor uses a FIFO queue to process goals and their responses. Goals received by the controller are translated in the reactor into the corresponding String commands understandable for MOPS/MOCUP and queued for execution. Separate processes forward the commands to MOPS which forwards the commands over the network to MOCUP. They also monitor and process responses forwarded back to MOPS from MOCUP. At any time, the controller application can then subsequently poll and react on new status information of each goal depending whether execution has started, execution was successful or execution has failed.

The MOPS reactor in the architecture plays the role of command dispatcher. The rover is able to move between two points in space given their coordinates $\langle ?x, ?y \rangle$ (taking into account that the rover may get stuck in between two points because of an obstacle) and the orientation $?h$ it has to reach. Besides that, the rover can take and store pictures. Hence MOPS reactor has been implemented to be able to execute the goals GOTO$(?x, ?y, ?h)$, to move the rover in $\langle x, y \rangle$ with final orientation h, TAKEPICTURE(), to take a picture in the current position and CHECKSTATUS(). The reactor provides for the observations AT$(?x, ?y, ?h)$ when the rover is standing in $\langle x, y \rangle$ with an orientation h (observation provided after having executed a command GOTO$(?x, ?y, ?h)$ or after a command CHECKSTATUS()), STUCKAT$(?x, ?y, ?h)$ when the rover is stuck in $\langle x, y \rangle$ with an orientation h (an error status), PICTURETAKENAT$(?x, ?y, ?h)$, after having took a picture or after a command CHECKSTATUS() (if the last goal successfully executed was to take a picture) and TARGETAT$(?x, ?y, ?h)$ when an interesting object is detected in $\langle x, y, h \rangle$ (while moving). We assume that a transition GOTO$(?x, ?y, ?h) \rightarrow$ AT$(?x, ?y, ?h)$ denotes a successful move to $\langle x, y, h \rangle$, while a transition GOTO$(?x, ?y, ?h) \rightarrow$ AT$(?x', ?y', ?h')$ or GOTO$(?x, ?y, ?h) \rightarrow$ STUCKAT$(?x', ?y', ?h')$, with $?x \neq ?x'$ or $?y \neq ?y'$ denotes an unsuccessful move to $\langle x, y, h \rangle$[4] with the rover standing in $\langle x', y' \rangle$ with orienta-

[4]For this simple model we consider a failure not reaching the position, with no care of the actual orientation of the rover. Once that the rover has reached the proper position no failure is considered possible

tion h' while moving (status AT) or stuck because of an obstacle during the path (status STUCKAT). A transition AT$(?x, ?y, ?h) \rightarrow$ GOTO$(?x', ?y', ?h')$ denotes the rover starting to move from a point $\langle x, y, h\rangle$ to a point $\langle x, y, h\rangle$. Finally, we do not consider possible a failure in taking a picture, hence the only possible transition involving the command TAKEPICTURE() will be AT$(?x, ?y, ?h)\rightarrow$ TAKEPICTURE() $\rightarrow$PICTURETAKENAT$(?x, ?y, ?h)$.

5. EXPERIMENTAL SETTING

The MOCUP rover is built on LEGO® bricks and the LEGO NXT 2.0 Mindstorms kit. It is extended by an ARM Linux Beagleboard with 1Ghz CPU and 500Mb of RAM with a Debian Linux operating system. The USB Ports of the board are used to interface with the NXT brick to control the LEGO® motors for movement and the ultrasonic sensors for obstacle detection, a webcam for taking pictures and a wireless dongle for connection to the network. The rover control software is started from the Linux system and communicating via the METERON simulator to the MOPS user interface.

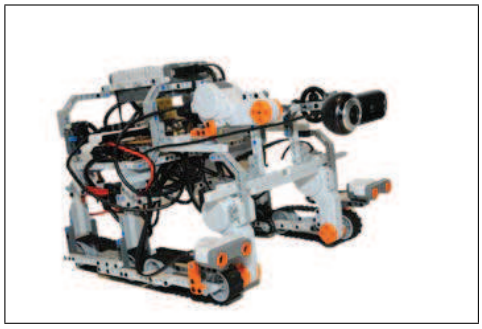

Figure 10. The MOCUP Rover

The ultrasonic sensors can detect obstacles in the front and rear, and can triangulate positions of objects in the front. If an obstacle is detected too close during movements, the rover is automatically stopped to prevent collisions, and an "interrupt" response is send back to the user interface. The network delays of the simulator have been set to a realistic ISS scenario with around 6-7 seconds of round-trip time for each transmitted packet. As a Delay Tolerant Network protocol is used in the communications chain, delays can be adjusted for various scenarios without the danger of data packets timing out during transmission. In addition, the simulator allows for pausing - hence delaying any data packet, as well as dropping packets to simulate data loss on the communications chain.

We have designed for the experimental evaluation 3 models, one for each of the 3 deliberative reactors described in the previous sections.

Regarding the Platform Manager reactor, we have designed a model to entail simple obstacle avoidance and the capability of performing opportunistic science while navigating the environment. The model is made of 3 timelines: the *goal timeline* TL_G, the *platform timeline* TL_P and the *trigger timeline* TL_T. The synchronization between the goal timeline and the platform timeline follows the pattern described in Figure 6. On TL_G is possible to post the goals AT$(?x, ?y, ?h)$, to move the platform and PICTURETAKENAT$(?x, ?y, ?h)$, to take a picture in the current position of the platform. The goals executable by the command dispatcher are posted on the platform timeline TL_G , while on TL_T the platform raises the triggers when an interesting object is found. Figure 12(a) shows the transition graph for TL_P to entail obstacle avoidance. This is an instantiation of the failure pattern (Figure 4, left), where STUCKAT$(?x, ?y, ?h)$ is the error status. The long path on the right is the recover procedure for the error status (when the rover is stuck, it takes a picture then moves around the obstacle).

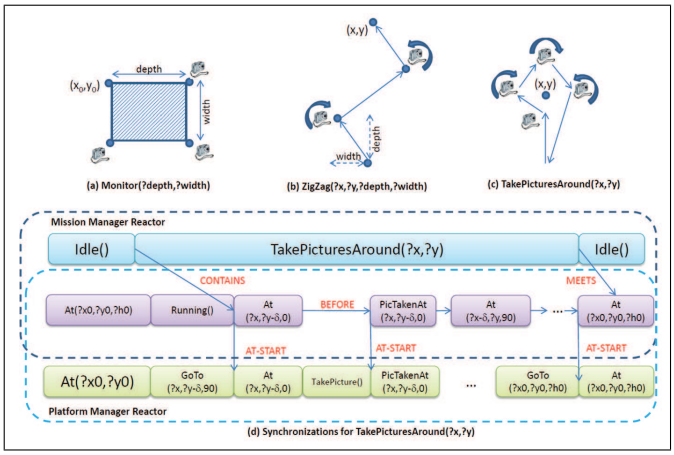

Figure 11. Mission Manager Tasks

To show an example of a model for opportunistic science, we suppose the rover equipped with a system to detect scientific targets while navigating the environment. The trigger timeline TL_T has 2 values: NOTARGET() (nominal status), TARGETAT$(?x, ?y, ?h)$ when an interesting object is detected in $\langle x, y, h\rangle$. The reaction consists in stopping the platform while moving, coming back to $\langle x, y, h\rangle$, take a picture of the target, restore the status of the trigger timeline and keep moving towards the destination (see Figure 12(b)). This is an instantiation of the triggering pattern (Figure 4, right), where the plan (P_i in the pattern) for a trigger T_i TARGETAT$(?x, ?y, ?h)$ consists in GOTO$(?x, ?y, h)\rightarrow$TAKEOPPPICTURE$(?x, ?y, h)$, and the restore plan for T_i (R in the pattern) consists in restoring the value NOTARGET() on TL_T. In this instantiation of the pattern we do not need a plan for interrupting the current action, since the platform is able to change direction without stopping if a new command is received.

In the mission manager reactor we have implemented 3 high level tasks (see Figure 11, (a)-(c)): MONITORING$(?depth, ?width)$ (to monitor an area and take pictures at the corners, coming back to the starting point), ZIGZAG$(?x, ?y, ?depth, ?width)$ (provides for a zigzag move to $\langle x, y\rangle$ with a fix amplitude and depth) and TAKEPICTUREAROUND$(?x, ?y)$ (to take 4 pictures all around a target in $\langle x, y\rangle$ and coming back to the starting point). The model is made of 2 timelines: the *mission goal timeline* TL_M and the *platform goal timeline* TL_G (the model implements the pattern in Figure 6). TL_G is the goal timeline of the platform reactor described above, TL_M is the goal timeline of the mission manager. TL_M

during rotation to get the final orientation.

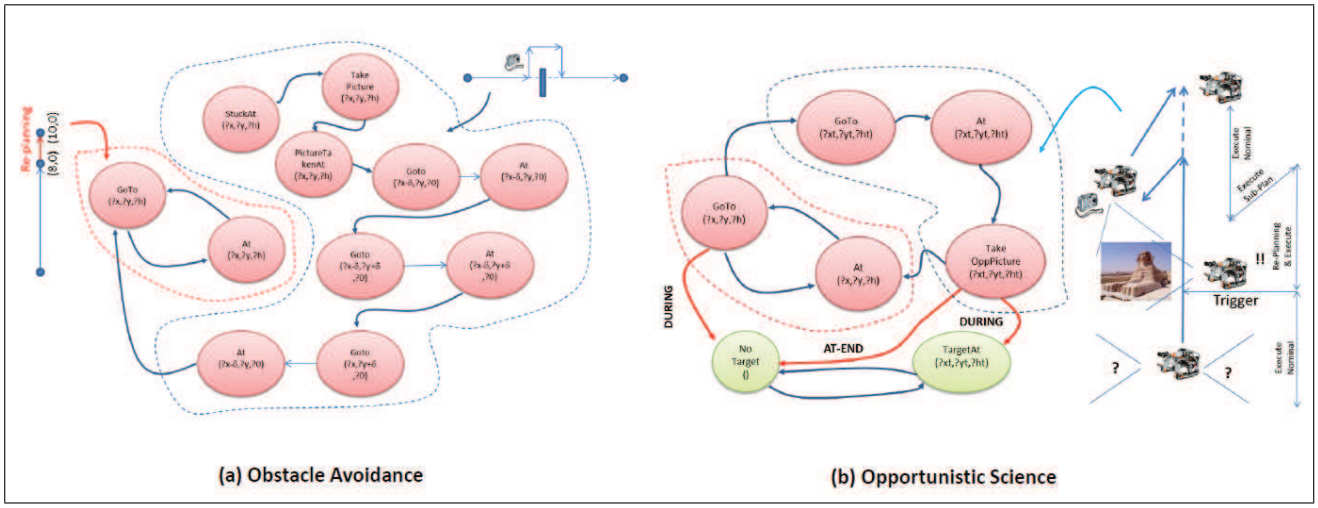

Figure 12. The Platform Reactor

can take the value IDLE() plus all the high level tasks described above.

The models have been tested (a) upstream with a communication manager reactor, conceptually similar to a platform manager reactor but implementing the behavior in Figure 8 (without obstacle avoidance and opportunistic science capabilities), and a command dispatcher (ingesting manually failures in the command dispatcher) and (b) downstream with a mission manager reactor on top of a platform manager reactor and a command dispatcher.

6. CONCLUSIONS

Space has been often a fertile field for the introduction of AI based advanced planning and scheduling technologies. In fact, the AI model-based approach allows reusing of software modules across different scenarios because of the great flexibility introduced by the symbolic representation of goals, constraints, logic, parameters to be optimized and so on.

However, a great effort and amount of time in general is necessary to understand domains and problems, capturing all the specificity, and create the model. This modeling difficulties constitute a not trivial barrier for the practical adoption of AI model-based P&S technologies, seriously harming the great advantage that, in theory, the approach could bring.

In this paper we discuss how to build models for an AI planning-based, domain independent control layer in the context of remote operations. The control layer has been used in two ways: on one side the controller is located at the operator side, providing a unified view of the communication layer plus remote system that allows the operator to reduce the impact of the delay and errors introduced by the communication channel. On the other side the controller is located at the remote system side, rising the abstraction level of the platform commanding and providing to the operator the view of a more robust autonomous, goal oriented system, that reduces the amount of controls to be transmitted and reduce the need of fine grained synchronization of commands for the remote system.

The experimental evaluation shows that is possible to provide, with a little effort in writing additional code, robustness and levels of autonomy for an existing architecture originally designed for remote operations. We have implemented the AI controllers with the GOAC platform. If the effort in writing code was minimal, with such an approach a not trivial efforts have been required for modeling. This paper is an initial step to address the problem of modeling in the context of AI planning for autonomy.

REFERENCES

1. W. Carey, P. Schoonejans, B. Hufenbach, K. Nergaard, F. Bosquillon de Frescheville, J. Grenouilleau, and A. Schiele. Meteron: A mission concept proposal for preparation of human-robotic exploration. In *Proceedings of the Global Space Exploration Conference, Washington D.C.*, 2012. GLEX-2012,01,2,6,x12697.
2. A. Ceballos, S. Bensalem, A. Cesta, L. de Silva, S. Fratini, F. Ingrand, J. Ocón, A. Orlandini, F. Py, K. Rajan, R. Rasconi, and M. van Winnendael. A Goal-Oriented Autonomous Controller for space exploration. In *Proceedings of the ASTRA 2011*, 11^{th} *Symposium on Advanced Space Technologies in Robotics and Automation*, 2011.
3. Vinton G. Cerf, Scott C. Burleigh, Robert C. Durst, Kevin Fall, Adrian J. Hooke, Keith L. Scott, Leigh Torgerson, and Howard S. Weiss. Delay-tolerant networking architecture, April 2007. Internet Research Task Force RFC4838.
4. J. Frank and A. Jonsson. Constraint based attribute and interval planning. *Journal of Constraints*, 8(4):339–364, 2003.
5. S. Fratini and A. Cesta. The APSI Framework: A Platform for Timeline Synthesis. In *Proceedings of the 1st Workshops on Planning and Scheduling with Timelines at ICAPS-12, Atibaia, Brazil*, 2012.
6. S. Fratini, A. Cesta, R. De Benedictis, A. Orlandini, and R. Rasconi. Apsi-based deliberation in goal oriented autonomous controllers. In *ASTRA 2011. 11^{th} Symposium on Advanced Space Technologies in Robotics and Automation*, 2011.
7. N. Muscettola. HSTS: Integrating Planning and Scheduling. In Zweben, M. and Fox, M.S., editor, *Intelligent Scheduling*. Morgan Kauffmann, 1994.
8. K. Rajan and F. Py. T-REX: Partitioned Inference for AUV Mission Control. In G. N. Roberts and R. Sutton, editors, *Further Advances in Unmanned Marine Vehicles*. The Institution of Engineering and Technology (IET), 2012.