Decisional Autonomy of Planetary Rovers

Félix Ingrand, Simon Lacroix, Solange Lemai-Chenevier[1] and Frederic Py[2]
LAAS-CNRS, University of Toulouse, France

Abstract

To achieve the ever increasing demand for science return, planetary exploration rovers require more autonomy to successfully perform their missions. Indeed, the communication delays are such that teleoperation is unrealistic. Although the current rovers (such as MER) demonstrate a limited navigation autonomy, and mostly rely on ground mission planning, the next generation (e.g. NASA Mars Science Laboratory and ESA Exomars) will have to regularly achieve long range autonomous navigation tasks.

However, fully autonomous long range navigation in partially known planetary-like terrains is still an open challenge for robotics. Navigating hundreds of meters without any human intervention requires the robot to be able to build adequate representations of its environment, to plan and execute trajectories according to the kind of terrain traversed, to control its motions and to localize itself as it moves.

All these activities have to be planned, scheduled, and performed according to the rover context, and controlled so that the mission is correctly fulfilled. To achieve these objectives, we have developed a temporal planner and an execution controller, that exhibit plan repair and replanning capabilities. The planner is in charge of producing plans composed of actions for navigation, science activities (moving and operating instruments), communication with Earth and with an orbiter or a lander, while managing resources (power, memory, etc) and respecting temporal constraints (communication visibility windows, rendezvous, etc). High level actions also need to be refined and their execution temporally and logically controlled. Last, in such critical applications, we believe it is important to deploy a component which protects the system against dangerous or even fatal situations resulting from unexpected interactions between subsystems (e.g. move the robot while the robot arm is unstowed) and/or software components (e.g. take and store a picture in a buffer while the previous one is still being processed).

In this article we review the aforementioned capabilities, that have been developed, tested and evaluated on board our rovers (Lama and Dala). After an overview of the architecture design principle adopted, we summarize the perception, localization and motion generation capabilities – required by autonomous navigation – and their integration and concurrent operation in a global architecture. We then detail the decisional components : a high level temporal planner which produces the robot activity plan on board, and temporal and procedural execution controllers. We show how some failures or execution delays are being taken care of with online local repair, or replanning.

I. Introduction

The control structure of an autonomous rover must allow the integration of both decision-making – *i.e.*, planning – and reactive capabilities. Indeed, the correct execution of rover missions requires the rover software to anticipate possible situations and adequate actions to cope with them. Tasks must also be instantiated and refined at execution time according to the actual context, which is not necessarily known at planning time, and the agent must react in a timely fashion to events. The following requirements for a planetary rover system are thus to be considered:

- Programmability: The rover should be able to achieve multiple tasks described at an abstract level. Its basic capabilities should therefore be easily combined according to the task to be executed.
- Autonomy and adaptability: the rover should be able to carry out its actions and to refine or modify the task and its own behavior according to the current goal and execution context as it perceives it.

[1]Currently at Centre National des Etudes Spatiales, Toulouse, France.
[2]Currently at Monterey Bay Aquarium Research Institute, CA, USA.

- Reactivity: the rover has to take into account events with time bounds compatible with the correct and efficient achievement of its goals (including its own safety) and the dynamics of the environment.
- Consistent behavior: the reactions of the robot to events must be guided by the objectives of its task.
- Robustness: the control architecture should be able to exploit the redundancy of the processing functions.

Other important requirements for autonomous agents in general, such as learning, are not considered here.

The above requirements call for the coexistence of deliberative and reactive behaviors in the system. Its architecture should therefore embed interacting subsystems performing according to different temporal properties.

In this paper, we present a framework to integrate deliberative planning, plan repair and execution control that takes into account resource level updates and temporal constraints, together with the subsystems responsible for actions execution. The work presented here summarizes years of development and experimental validations on the Marsokhod model rover "Lama", and now with the iRobot ATRV "Dala" (Fig. 1).

Fig. 1. The two rovers Lama and Dala, fully equipped with on-board sensing and processing capabilities.

The article is organized as follows: the next section summarizes how the various processes required by an autonomous rover are organized within a generic software architecture. This architecture integrates a full range of capabilities, from low-level motion control to high level planning algorithms, satisfying the requirements mentioned above. Section III briefly presents some of the most important functions required to achieve the basic navigation task, namely "reaching a given goal position" – this section is an overview, as it summarizes prior work previously published. Section IV then details how these capabilities are controlled by the higher level decisional processes, that plan the rover activities considering time constraints, control the execution of the navigation functions, and reactively re-plan

activities according to the way the rover actually behaves on the terrain. Then, section V presents the execution control mechanism which acts as a safety bag [1] in the LAAS architecture and enforces some strict rules on acceptable behavior. Finally, some experimental results are presented in section VI.

II. Software Architecture for Autonomy

It is clear that the implementation of several task-oriented and event-oriented closed-loops for achieving both anticipation capabilities and real-time behavior cannot be done on a single system with homogeneous processes, hence the need for a software architecture. This research field is becoming increasingly active as systems are more and more complex and provide advanced decisional capabilities. Our goal here is not to present a complete state of the art. Nevertheless, we should briefly present another architecture which is being developed for similar applications, and which has reached an interesting level of integration.

The CLARAty architecture [2]–[4] proposes an organization of software components along two "axes": an abstraction axis (i.e. from low level hardware to high level software processing) and a functional/decisional axis. This results in an architecture, which has two layers (a functional layer and a decisional layer). Yet, in each of these layers, one can have components at a very low level of abstraction (i.e. close to the hardware) from which one can build up higher level software components.

Over the last two decades, LAAS has developed a framework, a global architecture, that enables the integration of processes with different temporal properties and different representations. This architecture (Fig. 2) decomposes the robot system into tree main levels, having different temporal constraints and manipulating different data representations [5]. From bottom up, the levels are:

- The *functional* level includes all the basic built-in robot action and perception capacities. These processing functions and control loops (e.g., image processing, obstacle avoidance, motion control, etc.) are encapsulated into controllable communicating *modules*. These modules are activated by *requests* sent by the decisional level, according to the task to be executed. They send *reports* upon completion and may export data in *posters* to be used by other modules.
- The *decisional level* includes the abilities of producing the task plan and supervising its execution, while being reactive to events from the lower levels. The coexistence of these two features, a time-consuming planning process, and a time-bounded reactive execution process poses the key problem of their interaction and their integration to balance deliberation and reaction at the decisional level.
- The *execution control* level is the interface between the decisional and the functional levels. It controls and coordinates the execution of the functions distributed over the various functional level modules according to the task requirements. It achieves this by filtering the requests according to the current state of the system and a formal model of allowed and forbidden states. Despite its clear role in the architecture, it is embedded in the functional level.

We detail next the functional level, the decisional level in section IV, then the execution control level in section V.

III. Functional Level

The functional level/layer embeds the various functions that endow the rover with the ability to perform autonomous navigation. "Autonomous navigation" is here understood as the capacity to reach a given goal (position) without any operator intervention.

A. Overall approach to autonomous navigation

Depending on the specification of the goal and on the environment context, the achievement of an autonomous navigation task can require very different capabilities. Indeed, reaching a close goal in a perfectly known, almost obstacle free area is a very different task from reaching a far away goal in poorly known terrain, or from reaching a goal specified as a particular object designated by the

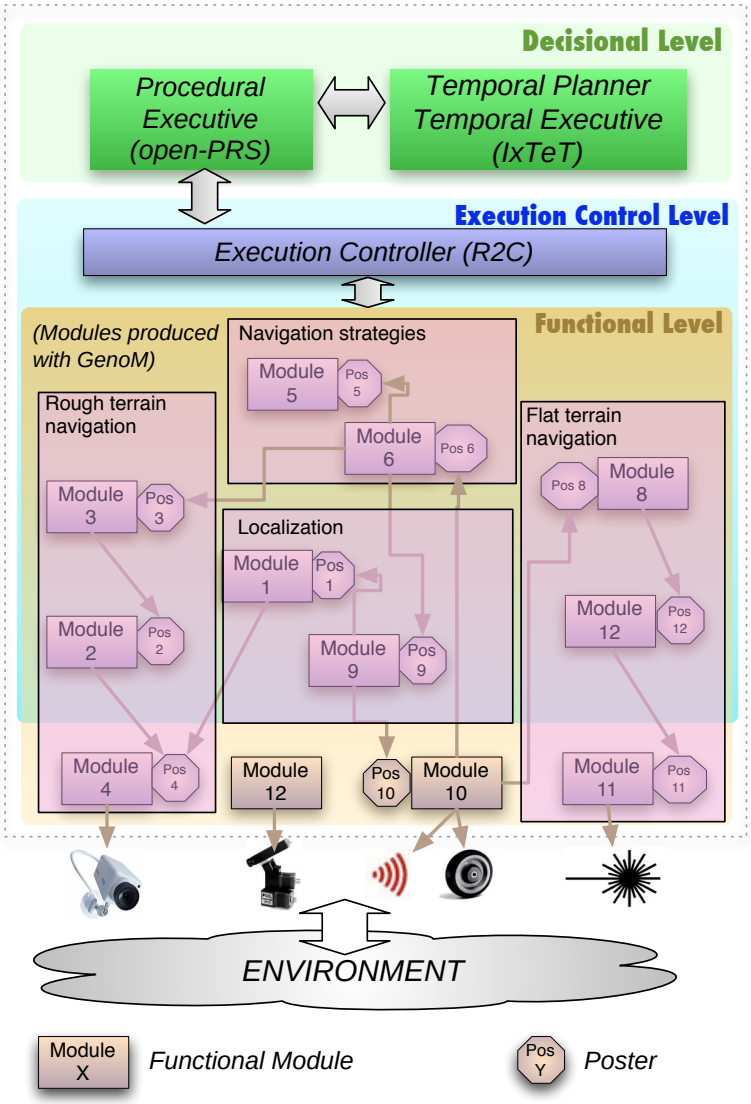

Fig. 2. The LAAS architecture. The functional level is here populated with dummy modules, to illustrate that sets of modules can be composed to provide higher modalities.

operators. We therefore define various *navigation modes*, adapted to the navigation task at hand: each navigation mode is a particular instance of the perception/decision/action loop [6]. For instance, one can consider the following modes:

- Flat terrain mode: if the terrain is mainly flat with only some sparse obstacles, simple obstacle avoidance or trajectory planning schemes on a binary traversable/obstacle terrain representation can be sufficient to reach a given goal.
- 3D mode: when the terrain exhibits a more complex geometry, a 3D description of the environment and algorithms that consider the chassis geometry are required to determine feasible paths.
- Long range traverses: regardless of the kind of terrain the rover has to cross, it must be endowed

with an ability to select paths at the itinerary level, especially to avoid getting trapped in dead-ends.
- Target reaching mode: this particular mode is devoted to position the rover with respect to a designated science target, so that it can readily deploy measuring instruments once it reaches it [7]. The specific feature of this mode is that during motion, the robot must keep track of the relative position of the target.

Of course, the fact that the rover is endowed with various different navigation modes calls for the ability to *select* the mode to apply, according to the given goal and environment context. This selection is quite a high level decision. It can be fulfilled by the operators, as it is the case for the MERs [8] or as it is foreseen for the ESA ExoMars rover [9]. But in the case of long range traverses, where the goal is located beyond the operator line of sight[1], future rovers will have to handle this decision autonomously, as the operator may not be aware of the terrain characteristics.

This section summarizes two navigation modes that have been integrated within our architecture, and the way they are selected during long range traverses (sections III-B to III-D). Algorithms for rover localization, an essential functionality, are briefly described in section III-E.

B. Flat terrain navigation modes

The problem of navigating on easy terrains, *i.e.* essentially flat with sparse obstacles, has received a lot of attention in the robotics community, and many approaches efficiently solve it. The main difficulty is the obstacle detection process.

1) Using stereovision data:

a) Obstacle detection: We developed a method that produces a probabilistic description of the terrain on the basis of stereovision data [10]. The method is a classification procedure that produces a probabilistically labeled polygonal map, akin to an occupancy grid representation. It relies on a discretization of the perceived area, that defines a *cell image*. Every cell is labeled with a non-parametric Bayesian classifier, using a learning base built off-line by an operator from sample images. On line, an attribute vector is computed for each cell, and the Bayes formula yields the determination of the partial probabilities for each cell to correspond to an obstacle (Fig. 3).

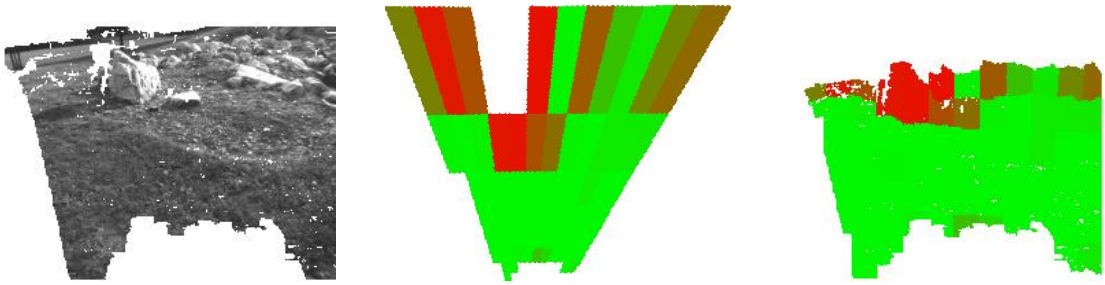

Fig. 3. An example of classification result. From left to right: image showing the pixels to which the stereovision algorithm could associate a depth, description of the perceived area with the cells partial probabilities to be an obstacle (the greener the more traversable the terrain), and re-projection of these probabilities in the sensor frame.

Thanks to the probabilistic description, local maps perceived from different viewpoints can be very easily merged into a global description – provided the rover is properly localized of course. The fusion procedure is performed on a Cartesian grid, in the pixels of which are encoded cell partial probabilities (Fig. 4).

[1]Or at least beyond the area for which precise 3D knowledge of the terrain is available.

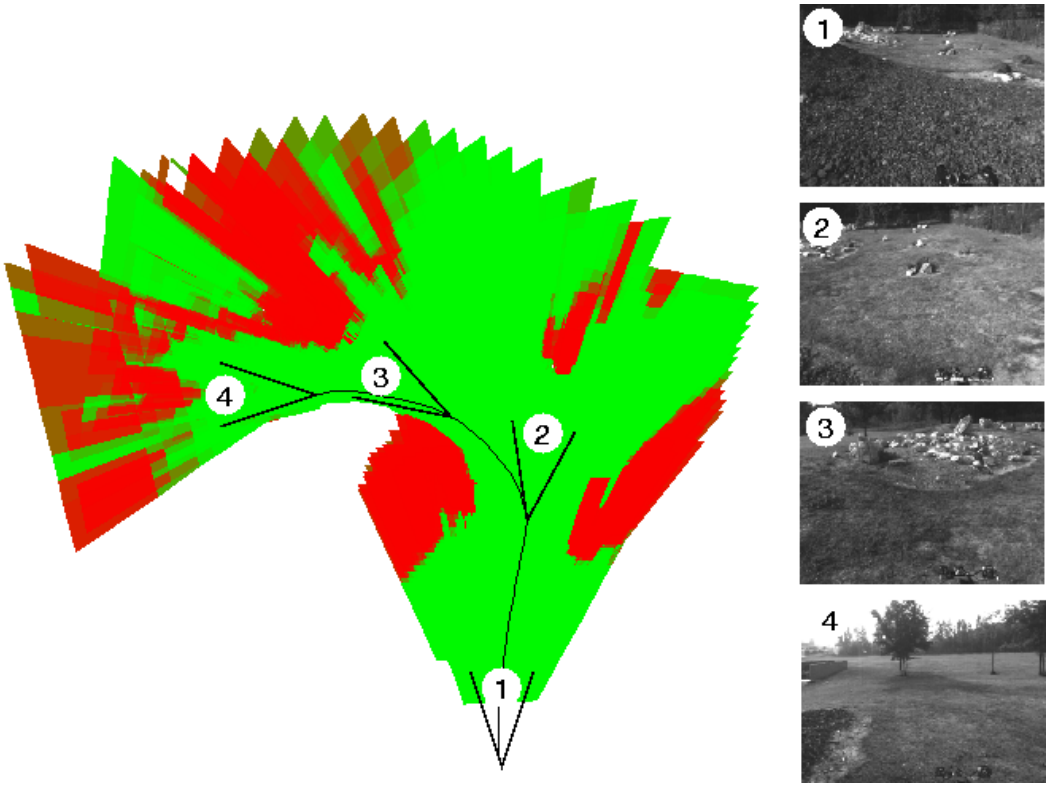

Fig. 4. A qualitative terrain model integrating 50 stereo pairs, acquired along a $20m$ long trajectory.

b) Path generation: On the basis of this probabilistic obstacle representation, various ways to find paths can be applied. In previous work, we proposed a potential-field based approach [11], however approaches that evaluate a set of elementary trajectories (figure 5), similarly to the Morphin algorithm [12], proved to yield better results.

2) Using laser range data: Laser range finders require too much energy to be exploited on planetary rovers. However we conducted many experiments with such a sensor on board the robot Dala. The approach is an extension of the "vector field histogram" approach introduced in [13]: it consists in analyzing range scans to select obstacle avoidance motions – details can be seen in [14].

C. 3D navigation mode

When the terrain does not exhibit flat traversable areas, a more complex way to generate motions is necessary. The principle of the approach sketched here is to virtually conform the rover chassis on a digital terrain map, in order to select admissible trajectories. Note that although such an approach will find trajectory in non flat terrains, autonomously traversing really rough areas remains a difficult challenge that calls for more sophisticated techniques [15], [16].

A digital elevation map (DEM) is a straightforward way to represent the terrain geometry [17], [18]. A precise enough model can be easily built by computing the mean elevation of the data points on the DEM cells. Provided the robot is localized with a precision of the order of the cell size, data acquired from several view-points can be merged into a global map (Fig. 6).

Motions are selected on the basis of the DEM in a very similar manner as in section III-B, except that the motions are now evaluated by conforming the rover chassis model with the digital elevation map. The

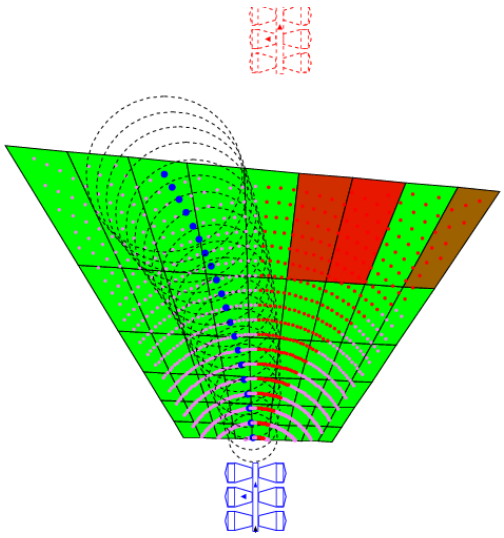

Fig. 5. Evaluation of elementary trajectories. Positions are evaluated along trajectories corresponding to various steering angles. To consider safe paths in case of trajectory deviations, the robot width is enlarged as it proceeds along the trajectory.

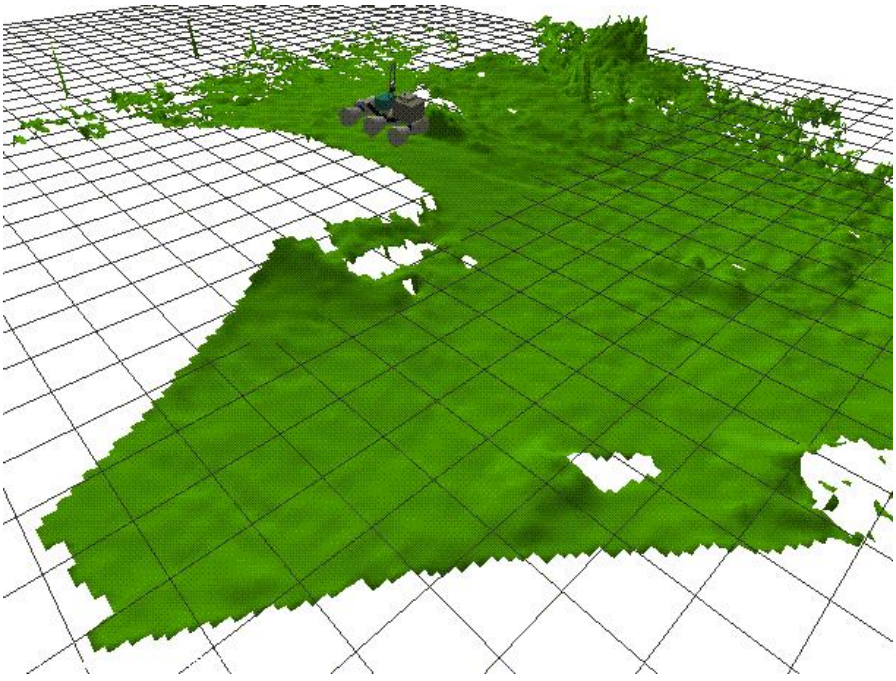

Fig. 6. Digital elevation map built with the same data used to built the model of Fig. 4 (The displayed grid is $1 \times 1m$, the map resolution is $0.1 \times 0.1m$).

notion of obstacle depends on the capacity of the locomotion system to overcome terrain irregularities. This capacity can be expressed as a set of constraints to satisfy: stability constraints, ground/rover body collision constraints, and internal configuration constraints when the chassis is articulated. All these constraints are evaluated along the analyzed trajectories: the chassis attitude and internal configurations

Fig. 7. A predicted placement of the robot on the DEM (left), and the corresponding real placement on the terrain (right).

is computed with the digital elevation map (figure 7), and the predicted configuration is then used to compute a "dangerousness" value for the considered position. Each considered trajectory has an associated *cost* that integrates the elementary costs of the successive configurations it defines, and an *interest*, which is the Dubbins distance to reach the goal from the arc end. The arc to execute is the one that maximizes the interest/cost ratio, and an A* algorithm is used to efficiently explore the tree formed by the evaluated trajectories (more details can be found in [19]).

D. Long range traverses

When the goal to reach is beyond the rover line of sight, motions are generated according to a two-stage process: the *navigation planner* first selects a sub-goal and the navigation mode to apply to reach it, and then the selected navigation mode is applied. Such an approach is typical to tackle long range navigation [10], [20], [21], and we extended it to consider the perception task to execute (direction of sight) once the sub-goal is reached. Indeed, the decisions related to motion and perception must be taken jointly, as they are strongly interdependent: executing a motion requires a model of e environment beforehand, and to acquire some specific data, motion is often necessary to reach the adequate observation position. Our approach to jointly specify the sub-goal to reach and the perception task to apply exploits the graph defined by the probabilistic model depicted in section III-B, and a model of the perception process expressed in terms of information gain – details on this approach can be seen in [22].

E. Rover localization

The ability for the rover to localize itself is *essential* to autonomous navigation. Indeed, navigation tasks are expressed in localization terms[2], the environment models built by the robot must be *spatially consistent*, and the closed-loop execution of the planned trajectories calls for the precise knowledge of the rover motions.

Robot self-localization is actually one of the most important functional issues to tackle in autonomous navigation, which is confirmed by the vast amount of contributions to this problem in the literature, and the variety of approaches. In the absence of any external structure (*e.g.* radioed bearings), no single localization algorithm that uses on-board data can be robust enough to fulfill the various localization needs during long range navigation: a set of *concurrent and complementary* algorithms have to be developed and integrated for that purpose. Various developments have been integrated on board the

[2]With an exception for the target reaching mode, where the goal location is not explicit

robots Lama and Dala: 3D odometry [23], visual motion estimation [24]. More recently, we investigated visual SLAM approaches[3], integrating stereovision and panoramic imaging [25].

From the overall decisional point of view, all the localization processes are “passive", in the sense that their activation is simply triggered by the selection of a motion mode – *e.g.* precise visual odometry may not be required when dealing with easy terrains.

F. Summary

Fig. 8 shows the functional level defined by the main navigation capabilities. This level is structured in *modules*, as defined by the tool GenoM (Generator of Modules) [26]. Each module is a specific instance of a generic module model, so that the behavior and the interfaces of the module operational functions obey standard specifications, which helps the systematization of the integration of several modules. Given a library that achieves a function, GenoM automatically produces a new module according the generic module specifications from a synthetic description of the library. The description is a form to fill, that declares the module services and specifies their parameters (possible qualitative results, exported data, temporal and logical characteristics, etc). GenoM modules may produce posters (octagons on the figure), which contain data produced by the module. Scan contains the laser scan produced by the Laser RF module, Obs the obstacle map produced by Aspect, Speed the speed reference produced by NDD, etc. The modules are activated with requests coming from OpenPRS through the R^2C. So the control flow is produced by the procedure executing, while the data flow is made through posters reading/writing.

On the sole basis of this formal description, GenoM produces a module that can run under various operating systems, and a set of interface libraries to communicate with the module using various programming languages (C, Tcl/Tk, Ruby, and of course OpenPRS).

The standardization associated to the formal description allows to easily develop a complex functional level, in which tens of modules can be integrated.

Note that other instances of this functional level have also been used on Gromit, an iRobot ATRV Junior used at NASA Ames in various experiments (IDEA [27], and cooperative exploration with K9 [28]). In these particular experiments, the upper part of the architecture is different. Nevertheless, OpenPRS was kept to manage the communications between the IDEA agents and the functional level.

IV. Decisional Level : Temporal Planning, Temporal Executive and Procedural Executive

Various strategies have been applied to implement a decisional level/layer. Some approaches propose the use of a strong executive, enhanced with deliberative capabilities. Procedural executives, such as TDL [29] or OpenPRS [30], support action decomposition, synchronization, execution monitoring and exception handling. In PROPEL [31], and PropicePlan [32], the planner is used by the executive to anticipate by simulating subplans, or to generate a new subplan corresponding to the current situation. These systems mostly provide a reactive behavior, whereas a look-ahead far in the future is necessary to achieve a strong level of autonomy (e.g. to manage the level of a limited resource during the entire mission).

Other approaches propose to interleave planning and execution in a continuous way. The planning process remains active to adapt the plan when new goals are added or to resolve conflicts appearing after a state update; and actions which are ready to be executed are committed to execution even if the plan is not yet completely generated. Examples include IPEM [33], based on the classical Partial Order

[3]The benefits of SLAM are poor when performing traverses that mainly consists in moving forward, but one can foresee that future rovers will carry on numerous tasks in given areas, thus requiring the ability to be precisely localized in already explored and mapped areas.

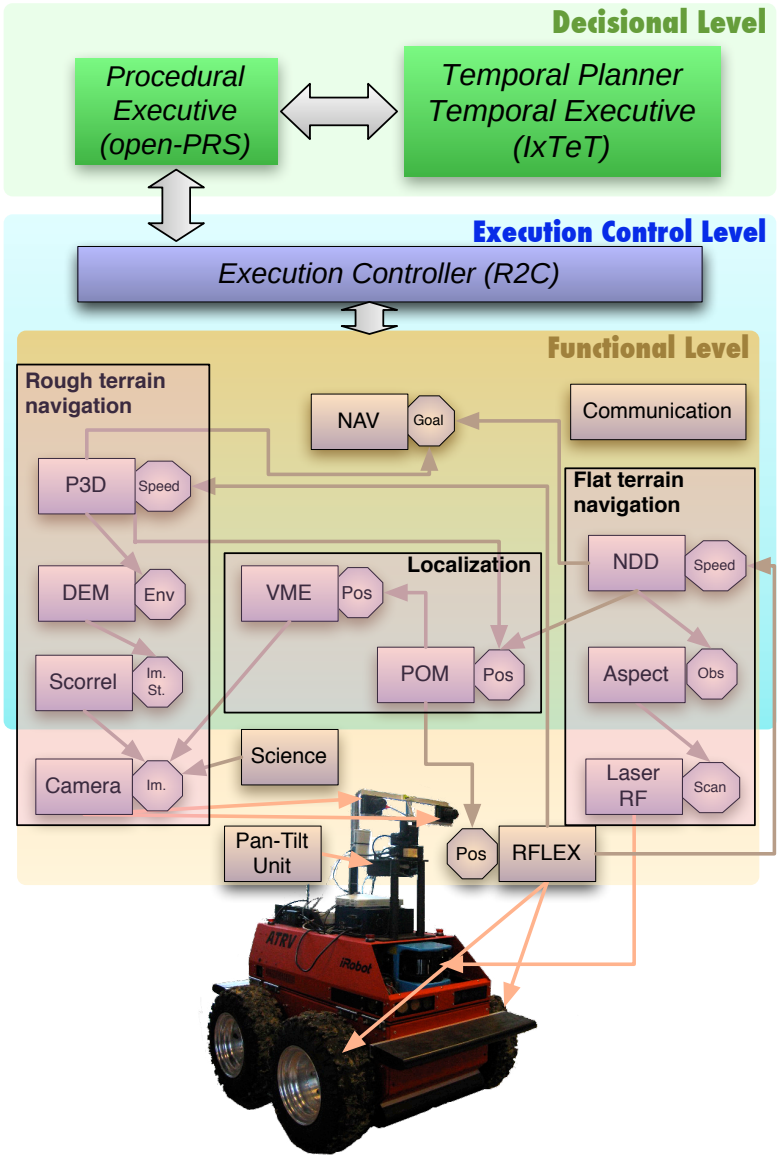

Fig. 8. An instance of the architecture with the functional level that embeds the capabilities mentioned in this section. Modules involved in the flat terrain navigation mode are on the right (mainly module Laser RF, ASPECT and NDD), and the modules involved in the 3D navigation mode are on the left (loop SCORREL, DEM, P3D). RFLEX just uses either one or the other speed reference produced. Arrows between modules indicate data flow – for readability purpose, only the main ones are shown.

Planning framework; ROGUE [34], which has been deployed on mobile robots; or the multi-agent architecture CPEF [35].

Still, very few approaches really take into account timing problems, such as actions with duration and goals with deadlines, and their effects on the planning and execution processes. Examples of temporal systems include CASPER and IDEA. In the CASPER system [36], state and temporal data are regularly updated and potential future conflicts are resolved using iterative repair techniques. However this approach does not handle conflicts which appear within the replanning time interval. The IDEA

approach [37] uses temporal planning techniques at any level of abstraction (mission planning as well as reactive execution). This system offers look-ahead abilities with flexible planning horizons and the use of a common language at any level of abstraction, but does not yet handle non-unary resources. Another interesting approach is Titan/Kirk [38]. Kirk can be viewed as a plan selector and executive. It looks into a precompiled TPN (Temporal Plan Network) and selects a thread that looks consistent. This thread is a paratially ordered plan offering some flexibility. It is then executed by Titan (via translation of the TPN into an RMPL program) which exploits a model of the plant with explicit defintion of possible failures to decide what is the best action to do to reach local objectives and avoid/repair possible failures. To do so it interleaves action ad state estimation using a markov decision process.

The LAAS decisional level integrates in one process: deliberative temporal planning, plan repair, temporal plan execution control; and in a second process procedural execution control. Overall, these components take into account resource level updates and temporal constraints.

The breakdown in two processes is mostly due to the data and models used by each component. The procedural executive manipulates procedures; considering that it interacts with lower level, it needs a fast reacting main loop (few hundredth of a second). While the temporal planner and temporal executive manipulate actions and plans with a much higher reaction time (few seconds).

The temporal planner and executive explicitly represent and reason about time. These two componants are fully integrated in the IxTeT planner. This one is able to interact with the user and the functional level through the procedural executive (OpenPRS). First, IxTeT produces a plan to achieve a set of goals provided by the user. The plan execution is controlled by both procedural and temporal executives as follows. The temporal executive decides when to start or stop an action in the plan and handles plan adaptations. OpenPRS expands and refines the action into commands to the functional level, monitors its execution and can recover from predefined failures. It finally reports to IxTeT upon the action completion. This cycle enables interaction with the controlled system by taking into account runtime failures and timeouts, and updating the plan accordingly. Interleaving plan repair and execution is motivated by the facts that some parts of the plan may remain valid and executable, and that the plan can be temporally flexible and thus allow postponing and inserting actions. Plan repair uses nonlinear planning techniques under certain assumptions discussed in this paper. The IxTeT planning system has the following properties: a temporal representation based on state variables and a Partial Order Causal Link (POCL) planning process that generates flexible plans based on CSP[4] managers, particularly the time-map relies on a Simple Temporal Network (STN) [39].

This section is organized as follows. We first present the Procedural Executive OpenPRS. We then focus on the IxTeT system, detail the dynamic replanning and repair mechanism, and address the issues raised by interleaving planning and execution processes on the same plan.

A. Procedural Executive: OpenPRS

There are a large number of Procedural Executive available. We can cite TDL, Situation Calculus, Rap, Exec, etc. They all share the same philosophy even if they differ in their details. The procedural executive remains a critical and important part of our architecture, yet, its capabilities, its properties and its implementation have already been discussed at length in various papers [5], [30]. Nevertheless, we briefly present it here, focusing on its integration with the other components.

In the context of robot experiments, the *Procedural Reasoning System* (in our case OpenPRS) has the following interesting properties:

- a semantically high-level language allowing for the representation of goals, subgoals, events and conditional operational procedures,
- the ability to deal with different activities in parallel,
- a bounded reaction time to new events is guaranteed.

[4]Constraints Satisfaction Problem.

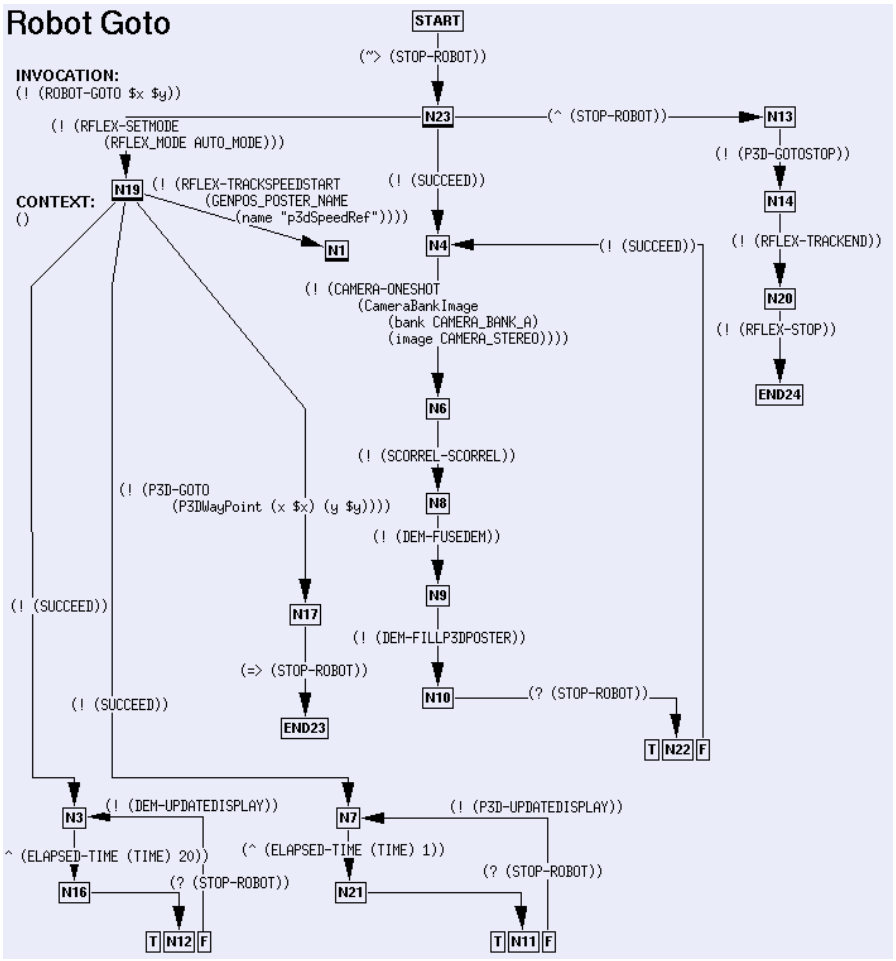

Fig. 9. Example of an OpenPRS procedure. This Robot Goto procedure will be called when an IxTeT MOVE is executed in the plan. The nodes with "thick" bottom correspond to split nodes: the outgoing branches are executed in parallel. Most edge labels correspond to requests which are sent to the corresponding functional modules. For example CAMERA-ONESHOT, correspond to the ONESHOT request of the CAMERA module, DEM-FUSEDEM is the FUSEDEM request of the DEM module, etc.

In the context of our autonomous rovers experiment, and more generally in the LAAS architecture for autonomous systems, OpenPRS interacts with the functional modules (through the R^{2}C Execution Control presented in section V), with the temporal executive, but also with the user. OpenPRS provides a number of functionalities presented below.

OpenPRS main activity is to refine high level "actions" into lower level "requests" sent to functional modules. Typically, these actions (such as a move) belong to the plan produced by the planner, yet they cannot be directly executed, without being refined by executing an appropriate procedure[5] – an example is shown Fig. 9. Note that the library can contain several such procedures for each action, that are applicable in different contexts: in such a case OpenPRS automatically selects the proper one (contexts are often exclusive, if not, OpenPRS choose randomly among the applicable one).

Apart from refining, OpenPRS also performs local recovery. All subgoals are automatically reposted

[5]A TakePicture action, for instance, corresponds to the sequence of requests `OneShot` (take a raw image) and `Save` (compress and store the image in a specific location) sent to the Camera module.

until a "complete" failure is reached. By complete failure, we mean that all the applicable procedures (with all possible context bindings) have failed. For example, a procedure to take a science picture may be applicable using one camera or another. If for some reason, the first instance fails (the camera is faulty), the second instance of the procedure will be executed. Moreover, often procedures are written in such a way that they test at run time what is the best execution path to take according to the context (requests reports, resource availability, etc.), and may be written to recover from immediate failures.

OpenPRS asynchronously reacts to reports from requests execution, and proper handling of non nominal termination through the execution of "recovery" procedures when available. It also asynchronously reacts to action execution requests from the temporal executive[6]. OpenPRS also stores in its database the current "state" of the robot, and monitor resource usage. It sends to the temporal executive and planner relevant information (for the planning and plan repair process). For example it reports on the action execution (actions termination status, the system state and resources levels), but it also passes new goals (given by the user).

OpenPRS also executes initialization procedures. Starting up the robot functional components and properly initializing them can be quite complex, as the sequencing and timing must be properly defined. Finally, OpenPRS should have a standby procedure to rely on while plan execution is aborted and replanning is in progress. In test lab situation, the robot just stays still. One could imagine that in real exploration, the robot puts itself in some safe mode.

OpenPRS internal components are:

- **A database** It contains facts representing the system view of the world, and is constantly and automatically updated as new events appear (internal events or external events from an operator, the temporal planner or the functional modules). Thus the database can contain symbolic and numerical information such as the position of the robot or the status reports of requests.
- **A library of procedures** Each procedure describes a particular sequence of subgoals, actions and tests that may be performed to achieve given goals[7] or to react to certain situations. Procedures provide classical programming control structure (if-then-else, while, etc) as well as more advanced construct such as parallelism, semaphore handling, etc.
- **A task graph** It corresponds to a dynamic set of tasks currently executing. Tasks are dynamic structures which keep track of the state of execution of the intended procedures and of the state of their posted subgoals.

An interpreter runs these components: it receives new events and internal subgoals, selects appropriate procedures based on the new situation and places them on the task graph, chooses one task and finally executes one step of its active procedure. This can result in a primitive action (e.g. a request sent to a module), or the establishment of a new subgoal.

Last, the recent developments around OpenPRS have greatly improved its integration with GenoM functional modules. Using the Transgen tool, one can automatically generate an OpenPRS kernel with all the encoding/decoding functions, as well as the OpenPRS procedures to call all the requests and access all the posters provided by a given set of functional modules. This may not be an architectural advantage, nevertheless, the perfect integration of tools greatly reinforces the use of the decisional tools among a community traditionally more focused on the functional aspects of robotics.

[6]The temporal executive only manages the temporal aspect of the execution, while the procedural one focuses on the refinement process

[7]In PRS, *goals* correspond to the description of a desired state along with the behavior to reach/test this state. Thus goals can be: *achieve* (!) a condition, *test* (?) a condition, *wait* (^) for a condition to become true, passively *preserve* a condition or actively *maintain* a condition while doing something else. Apart from goals, the possible instructions in a procedure include explicit addition (=>) or removal (~>) of facts in the database, standard programming structures (if-then-else, while, etc.) and an extensive use of variables.

B. The Temporal Planner : IxTeT

The planner in IxTeT is a lifted POCL temporal planner based on CSPs [40]. Its temporal representation describes the world as a set of *attributes*: logical attributes (e.g. robot_position($?r$)[8]), which are multi-valued functions of time, and resource attributes (e.g. battery_level()) for which one can specify borrowings, consumptions or productions. We note $LgcA$ and $RscA$, respectively the sets of logical and resource attributes. $LgcA_g$ and $RscA_g$ designate the sets of all possible instantiations of these attributes.

The evolution of a logical attribute value is represented through the proposition *hold*, which asserts the persistence of a value over a time interval, and the proposition *event*, which states an instantaneous change of value. The propositions *use*, *consume* and *produce* respectively specify over an interval the borrowing, the consumption or the production at a given instant of a resource quantity.

```
task MOVE(?initL,?endL)(st,et){
  ?initL,?endL in LOCATIONS;
  event(ROBOT_POS():(?initL,IDLE_POS),st);
  hold(ROBOT_POS():IDLE_POS,(st,et));
  event(ROBOT_POS():(IDLE_POS,?endL),et);
  event(ROBOT_STATUS():(STILL,MOVING),st);
  hold(ROBOT_STATUS():MOVING,(st,et));
  event(ROBOT_STATUS():(MOVING,STILL),et);
  hold(PTU_POS():FORWARD,(st,et));
  variable ?di,?du,?dist;
  variable ?duration;
  distance(?initL,?endL,?di);
  distance_uncertainty(?du);
  ?dist = ?di * ?du;
  speed(?s);
  ?dist = ?s * ?duration;
  contingent ?duration = et − st;
}latePreemptive
```

Fig. 10. Example of move action model.

```
task MOVE_PTU(?initL,?endL)(st,et){
  timepoint end_heat;
  ?initL,?endL in PTU_POSITIONS;
  hold(ROBOT_STATUS():STILL,(end_heat, st));
  event(PTU_STATUS():(COLD, HEAT),st);
  hold(PTU_STATUS():HEAT,(st,end_heat));
  event(PTU_STATUS():(HEAT,MOVING),end_heat);
  hold(PTU_STATUS():MOVING,(end_heat,et));
  event(PTU_STATUS():(MOVING,COLD),et);
  hold(PTU_INIT():TRUE,(st,et));
  hold(PTU_POS():?initL,(st,end_heat));
  event(PTU_POS():(?initL,PTU_POS_IDLE),end_heat);
  hold(PTU_POS():PTU_POS_IDLE,(end_heat,et));
  event(PTU_POS():(PTU_POS_IDLE,?endL),et);
  (end_heat − st) in [10,12];
  contingent (et − st) in [16,20];
}latePreemptive
```

Fig. 11. Example of move_ptu action model.

As shown Fig. 10, an action (also called *task*) consists of a set of *events* describing the change of the world induced by the action, a set of *hold* propositions expressing required conditions or the protection of some fact between two events, a set of resource usages, and a set of constraints on the timepoints and variables of the action. Note the *contingent* keyword used to express that this duration is not controllable and should not be modified by the planner.

A plan relies on two CSP managers. A Simple Temporal Network (STN) handles the timepoints and their binary constraints (ordering, duration, etc.). The other CSP manages atemporal symbolic and numeric variables and their constraints (binding, domain restriction, sum, etc.). Mixed constraints between temporal and atemporal variables can also be expressed [41] (e.g. the relation between the distance, speed and duration of a move $?dist = ?speed * (et - st)$). These CSP managers compute

[8] $?r$ represents a variable.

for each variable a minimal domain which reflects only the necessary constraints in the plan. Thus the plan is least committed and as much as possible flexibility is left for execution.

The plan search explores a tree $\mathcal{T}$ in the partial plan space. In a POCL framework, a partial plan is generally defined as a 4-tuple (A, C, L, F), where A is a set of partially instantiated actions, C is a set of constraints on the temporal and atemporal variables of actions in A, L is a set of causal links[9] and F is a set of flaws. A partial plan stands for a family of plans. It is considered to be a valid solution if all its possible instances are coherent, that is F is empty.

The root node of the tree $\mathcal{T}$ consists of the initial state (initial values of all instantiated attributes), expected availability profiles of resources, goals to be achieved (desired values for specific instantiated attributes) and a set of constraints between these elements. The branches of $\mathcal{T}$ correspond to resolvers (new actions or constraints) inserted into the partial plan in order to solve one of its flaws. Three kinds of flaws are considered:

– *Open conditions* are events or assertions that have not yet been established. Resolvers consist of finding an establishing event (in the plan or a new action) and adding a causal link that protects the attribute value between the establishing event and the open condition.

– *Threats* correspond to pairs of *event* and *hold* whose values are potentially in conflict. Such conflicts are solved by adding temporal or binding constraints.

– *Resource conflicts* are detected as over-consuming sets of potentially overlapping propositions. Resolvers include insertion of resource production action, action reordering, etc

Thus, a planning step consists of detecting flaws in the current partial plan, selecting one, choosing a resolver in its associated list of potential resolvers and inserting it into the partial plan. This planning step is repeated until a solution plan is found. When a dead end is reached (flaws remain but no resolver are available), the search backtracks on a previous choice. The algorithm is complete and the flaw and resolver choices are guided by diverse heuristics discussed in [40]. Note that the search is stopped as soon as a valid plan is found.

The advantages of the CSP-based functional approach are numerous in the context of plan execution. Besides the expressiveness of the representation (handling of time and resources), the flexibility of plans (partially ordered and partially instantiated, with minimal constraints) is well-adapted to their execution in an uncertain and dynamic environment. Plans are actually constrained at execution time. Finally, the planner, performing a search in the plan space, can be adapted to incremental planning and plan repair.

Nevertheless, there are still open problems such as how to handle the controllability issue. Regular propagation in STN, and by extension in the atemporal CSP, may shrink a temporal interval which may not be "controllable" by the planner. As a result, the execution may fail, not because the action model is wrong, but because the planner took some "freedom" with respect to what it is allowed to control. For example, STN propagation may shrink a move originally allowed to execute between 5 and 7 minutes to 5 and 6 minutes. Of course, one could make a model which maximizes all durations but produces plan with a poor makespan.

The 3DC+ algorithm addresses this problem and was first introduced by [42] to address this problem and propose the STNU approach. We have implemented it in IxTeT, and we are thus able to produce plans which are dynamically controllable with waits. The resulting plans may not be as "efficient" as one produced without 3DC+, but it is more robust and still more efficient than a plan where all non controllable actions have been maximized.

C. The Temporal Executive : IxTeT (again)

The temporal executive part of IxTeT is more "recent" than the planner part [43]. The temporal executive controls the temporal network of the plan produced by IxTeT by deciding the execution

[9] A causal link $a_i \xrightarrow{p} a_j$ denotes a commitment by the planner that a proposition p of action a_j is established by an effect of action a_i. The precedence constraint $a_i \prec a_j$ and binding constraints for variables of a_i and a_j appearing in p are in C.

order of actions execution and by mapping the timepoints at their execution time. Upon bootstrapping, IxTeT produces first a full plan for the initial situation and the original goal. Only when this first plan is ready, the temporal executive kicks in and starts execution. The execution of an action a with grounded parameters p_a, starting timepoint st^a, ending timepoint et^a, and identifier i_a is started by sending the command to the procedural executive OpenPRS. If the action is *non preemptive*, et^a is not controllable, and IxTeT just monitors if a is completed in due time (e.g. in et^a temporal window $[et^a_{lb}, et^a_{ub}]$). Otherwise et^a is controllable: if the action does not terminate by itself, it is stopped as soon (respectively as late) as possible if a is *early (resp. late) preemptive*.

IxTeT integrates in the plan the reports sent by OpenPRS upon each action execution completion. A report returns the ending status of the action (*nominal*, *interrupted* or *failed*) and a partial description of the system state. If nominal, it just contains the final levels of the resources, if any, used by the action. Otherwise, it also contains the final values of the other state variables relevant to the action.

Besides completion reports, IxTeT also reacts to exogenous events coming through OpenPRS to insert a new goal (from user requests) but also to sudden alterations of a resource capacity (e.g. the loss of a memory bank used to store science data).

In any case, while execution is taking place, various events from OpenPRS can forbid further execution of the plan, and may invalidate it:
– *temporal failures* The STN constrains each timepoint t to occur inside a time interval $[t_{lb}, t_{ub}]$. Thus two types of failure lead to an inconsistent plan: the corresponding event (typically, the end of an action) happens *too early* or *too late* (time-out).
– *action failure* The system returns a non nominal report (e.g. the robot was blocked by unexpected obstacles).
– *resource level adjustment* If an action has consumed more or produced less than expected, the plan may contain future resource contentions.

When these occur, and if an event produced an inconsistency (either temporal or on attributes) IxTeT starts and controls the processes of plan adaptation. To take advantage of the temporal flexibility of the plan, the dynamic replanning strategy has two steps. A first attempt is to repair the plan while executing its valid part in parallel. If this fails or if a timepoint times out, the execution is aborted and IxTeT completely replans from scratch.

Yet, to distribute planning on several cycles raises two problems:
Which plan does the concurrent execution relies on, especially if no solution has been found? This plan has to be *executable*. At each planning step, the node is labeled if the current partial plan is *executable*. When the maximum duration for this cycle has elapsed, the last labeled partial plan becomes *ExecutingPlan*.
Which plan and which search tree the planning process relies on in the next cycle? If no change has been made meanwhile (no timepoint execution, no message reception), the search tree can be kept as is and further developed during the next *plan repair* part. However, if the plan has been modified, a new search tree whose root node is the new *ExecutingPlan* is used, and the planning decisions made in previous cycles are committed.

Basically, all modifications made to *ExecutingPlan* have to guarantee that an *executable* plan is available after each phase of the cycle. If this condition does not hold, the cycle is stopped and a complete replanning is mandatory. During a cycle without plan repair, *ExecutingPlan* remains a solution plan.

V. Execution Control Level : the R^{2}C

The Execution Control Level has not been "present" in all instances of our architecture. Yet, its role is paramount, even more in critical applications where errors resulting from planning and plan refinement may lead to the loss of the mission.

As shown in Fig. 2, the execution control level sits in between the decisional level and the functional level. Note that it is different from the Procedural Executive (OpenPRS) presented in section IV-A

and from the Temporal Executive (integrated in IxTeT) presented in section IV-B. The R^{2}C (Requests and Reports Checker) role is to prevent the autonomous system to enter dangerous states which could lead to catastrophic or fatal consequences. For example to prevent the rover to reorient its antenna while a communication is taking place, or to move while the arm is unstowed, etc. Of course, no decisional level should ever produce such a "faulty" sequence of actions. Yet, we have seen that in the general case, a planner cannot produce a sequence of actions at the level provided by our functional modules (the plan produced by IxTeT are at a level higher than the level at which the functional modules are commanded). Therefore, some refinement mechanisms are required. In our architecture the Procedural Executive provides such mechanisms. However, as seen in section IV-A, OpenPRS flexibility, parallelism and expressiveness has a cost. These procedures are written by hand, and there is no way one can guarantee that unexpected race conditions or particular execution interleavings will never occur. So despite the use of models (actions and procedures) at the decisional level, one need a "low level" fault detection mechanism to prevent such problem to occur.

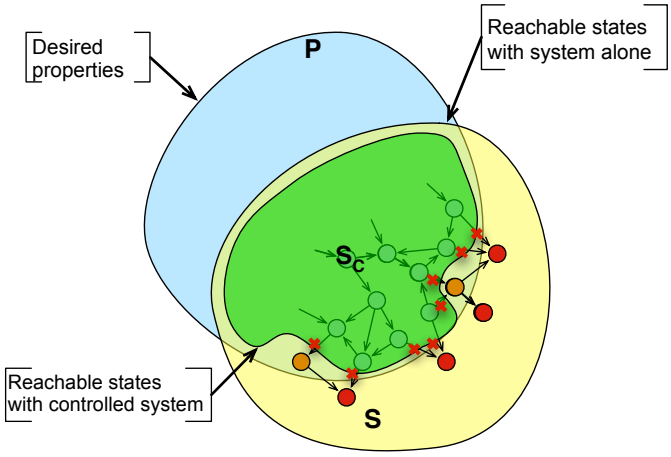

Fig. 12. States reachable by the system alone, and with a controller.

The main idea supporting execution control is to offer a component that makes sure the system will never reach a "faulty" state (Fig. 12). It is a fault protection system acting as a filter or "safety bag" [44]. To efficiently control the system (e.g. the functional level), the execution controller must respect some basic requirements:

- *Observability:* The component must have the ability to monitor and catch all events that may lead to a system problem. Note that in our current scheme, this prevents modules from sending requests to other modules. All requests must come from the decisional level (OpenPRS in our case).
- *Controllability:* The execution control must be able to control the system to avoid faulty states. In our current implementation this controllability is limited to the possibility to block commands and kill activities. Nevertheless, it appeared to be sufficient in our applications.
- *Synchronous and Bounded cycle time:* The component must act under a synchronous hypothesis (i.e. computation and communication take virtually no time). Apart from avoiding asynchronous formalism difficulties, this allows us to have a cleaner formal model of the system behavior and state transitions. In a practical way, this implies that the system will run as a loop with a bounded cycling time, offering, by this way, guarantees on the overall system reactivity.
- *Verifiability:* The execution control component has to offer a formalism and a representation that allow the developer to check if it safely controls the system behavior.

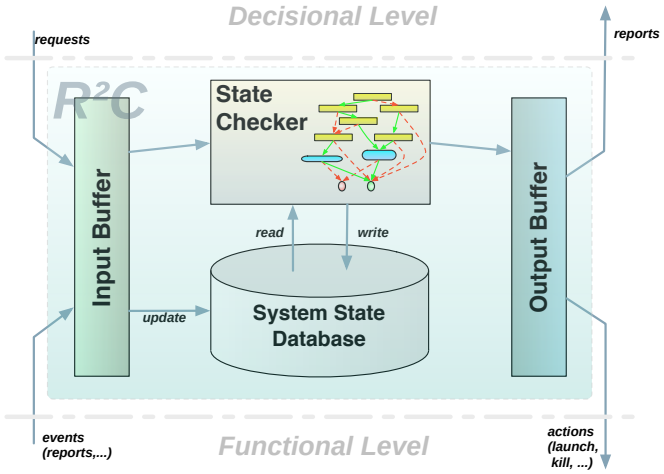

Fig. 13. The R^{2}C internal structure.

As shown Fig. 13, the R^{2}C captures all events that change the system state (request of services, reports of service execution, ...), updates its current system state representation and checks if these new events do not lead into an inconsistent state. If they do, then it takes actions (rejecting requests, killing services, suspending processes, ...) to keep the system consistent. Otherwise, it allows the request and the reports to be passed as expected.

The main component of the R^{2}C is thus the *state checker*. It encodes the constraints of the system which specify the acceptable and unacceptable states. To specify these constraints we have defined a language to model the system and its evolution. The model we use is presented in more detail in [45], [46]. We illustrate it here with an example:

```
check {
never: running(camera.oneshot()) && !last(camera.init(?mode));
always: last(camera.init(?mode) with ?mode!=LOW)
             => !( running(platine.move(?pos))
                      && running(camera.oneshot()) );
}
```

Fig. 14. Example of constraints compiled in the R^{2}C. The first one specifies that a camera can take a picture (`camera.oneshot`) if and only if the camera has already been initialized (`camera.init`). The second one specifies that one can only take a picture and move the PTU at the same time if the camera is in low resolution mode .

The user who programs the R^{2}C specifies a number of constraints such as the one of Fig. 14. The role of the R^{2}C is to maintain these constraints satisfied in all the states of the system. If some of the components of the resulting logical formula are "given", others are controllable. In this case it can reject or kill service instances of *platine.move* or *camera.take* to maintain consistency.

The R^{2}C may be seen as a component maintaining a formula true. Still, such an approach is reasonable in our context if and only if the R^{2}C deductions are fast enough to keep the synchronous hypothesis "acceptable". The approach we propose to satisfy this requirement is to use Ordered Binary Decision Diagrams (OBDDs, see [47]). This graph based data structure expresses logical formulas with the following properties:

- The resulting structure is a complete factorization of the initial formula. This implies that a

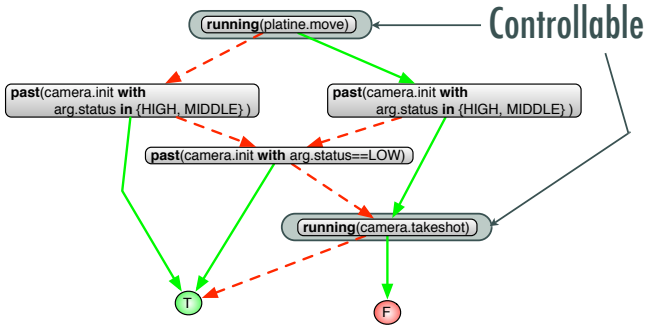

Fig. 15. Simple OCRD corresponding to the constraints expressed in Fig. 14. Note that two nodes are controllable, thus to reach the T node, the R^{2}C may have to kill/prevent the platine to move, or the camera to take a picture.

predicate value is checked only once.

- The traversal is bounded (complexity is on the order of the number of variables).
- We can validate it using formal techniques.

OBDDs are used to express first order logic formulas. This is not sufficient when our model is more complex with the introduction of constraints. Thus we have defined an OBDD like data structure named OCRD[10](see [48]).

This data structure is quite similar to OBDDs but is able to express formulas with predicates with simple constraints. The construction algorithm of one OCRD is almost identical to the OBDD one. It differs on the introduction of predicates which are similar but with different constraints on the attributes. In this case the compiler makes a partition of the constraints and split nodes accordingly. An example of such OCRD is given in Fig. 15 where the predicate **past** `(camera.init)` was split into two symbolic variables according to the constraints applied to its arguments.

VI. Experimentation and Results

We illustrate in this section the capabilities and the performances of the completely integrated system. This architecture and tools have been used on a large number of robots (Lama, Rackham, Gromit, etc), with different setups, using some of the components. Nevertheless, they have also been used all together on a real robot (Dala, a B21), as well as with a simulated environment integrated with GenoM and the functional level [49]. In these particular complete experiments, we ran various scenario, but also various navigation modalities. For example if outdoors we run the rough terrain navigation presented above, indoors, and for the sake of a "faster" navigation we use the flat terrain NDD based navigation mode [14]. So we are able to report on specific and focus experiments as well as complete integration which give the true experimental validation of our approach.

As for the high level planning component, we present here our experimental results with an example of a scenario for a rover exploration mission. In such a domain, the quantitative effects and durations can be estimated in advance for planning but are accurately known only at execution time (e.g. the actual compression rate of an image or the actual duration of a navigation task), thus requiring regular updates and look-ahead capabilities to manage unforeseen situations and resource levels. We also illustrate the use of 3DC+ algorithm in order to produce a more robust plan (with respect to execution).

We set up an exploration mission scenario which requires the robot Dala to achieve three types of goals (see Fig. 17): "take pictures of specific science targets" (in locations (0.5,-0.5), (4.5,-0.5), (1.5,-2.5)), "communicate with a ground station during visibility window" ($W_1[117-147]$), and "return to

[10]OCRD: Ordered Constrained Rules Diagram.

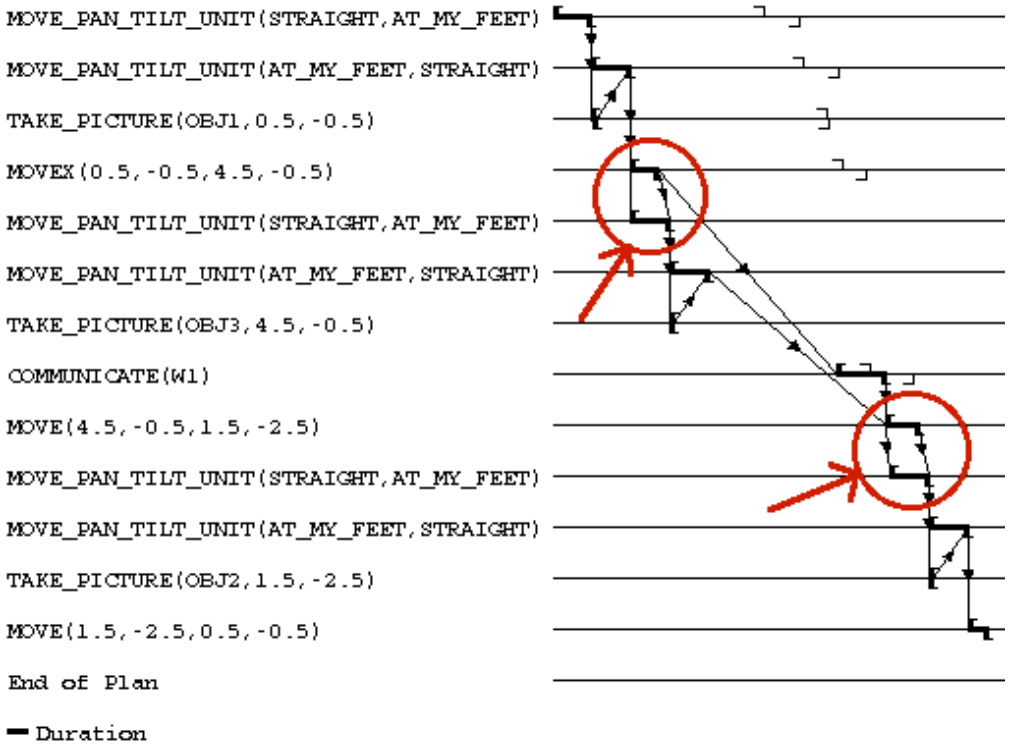

Fig. 16. Initial plan produced with the use of 3DC+ (note the flexibility left, the dependencies and the parallelism).

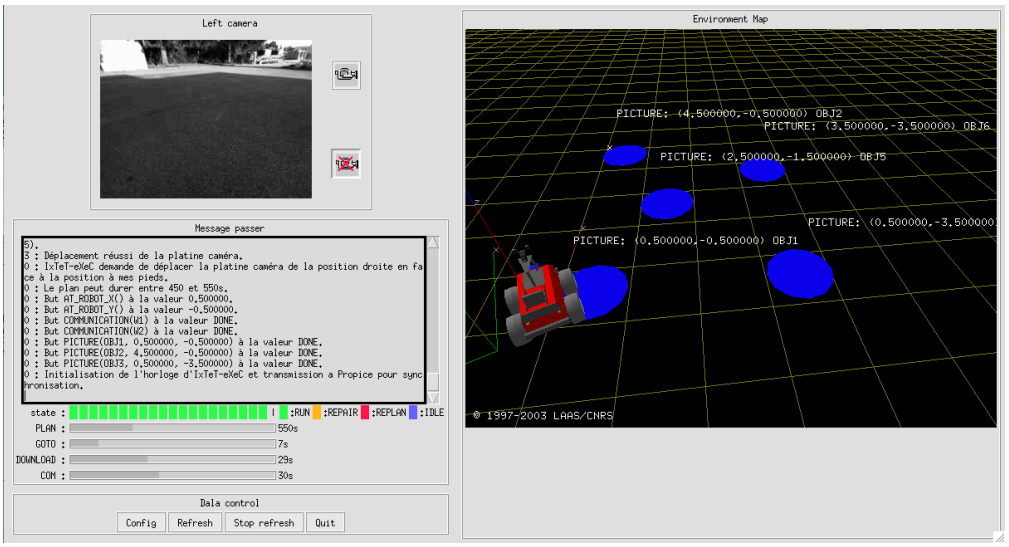

Fig. 17. DALA GUI showing the goals of the exploration mission. The environment model is empty because the robot has not yet started its mission.

location (0.5,-0.5) before time 500". Dala runs a 3 GHz Pentium IV (1 GB memory) under Linux and is equipped with the following sensors: odometry and a stereo camera pair mounted on a pan&tilt unit (PTU). Five main actions are considered at the mission planning level: take_picture, move_ptu (Fig. 11), move (Fig. 10), download_images, communicate. The first three actions are performed by Dala, while the last two are realistically simulated.

There are specific constraints attached to each tasks. The pan&tilt unit must be warmed up ten seconds before it can move. During a move action (of the rover), the camera must be pointed at a specific angle in order to provide the best perception of the environment. Thus the move and the move_ptu actions are mutually exclusive, however the pan&tilt unit can be warmed up during the "end of the move". It allows us to start a move_ptu action before the end of the move that precedes it without "stopping" the move itself. Yet, to do so, we need 3DC+ to correctly produce and execute this plan. Without this, IxTeT produces a plan which may shorten the duration of the move to its lower limit and we will most likely get a temporal failure. One can see Fig.16 that in the initial plan the end of the two move actions is overlapped by a move_ptu action. At this stage, the IxTeT Plan Viewer used to produce this screen dump does not show the wait introduced by the 3DC+ algorithm. But the plan produced will start the move_ptu as soon as the move is completed or ten seconds before its latest termination date.

The plan execution is controlled by both executives as follows. IxTeT decides when to start or stop an action in the plan and handles plan adaptations. OpenPRS expands the action into commands to the functional level[11], monitors its execution and can recover from specific failures. It finally reports to IxTeT upon the action completion.

This mission (the corresponding initial plan with 3DC+ is shown in Fig. 16) has been executed by Dala under IxTeT control (with total cycle duration$= 3s$). The initial plan with 3DC+ was produced in $7.1s$.

Fig. 18 shows the duration of each phase of the cycle for a run with 3DC+.

On similar hardware, OpenPRS takes negligible time compared to the planner and the functional modules. The R^2C compiles 14 rules in less than half a second, and produces an OCRD of a maximum depth of 28 nodes. The processing cycle time is less than $0.4ms$, which shows that such approach can be integrated without any penalties.

VII. Conclusions

The autonomous operation of a planetary rover requires methods for navigation (terrain perception, localization, motion planning and execution), as well as task planning and execution control. These various processes have to be embedded in a single architectural concept providing for consistent global behavior of the robot system, and respecting the temporal properties of each function. The space application context stresses not only the requirements for operational autonomy because of time delays, but it also adds temporal constraints and deadline on mission execution to cope with visibility and communication windows, or energy and resource availability. This article presented a complete framework for autonomous rover operation, that has been developed over several years and succesfully demonstrated on different rovers. To some extent, it sums up a large body of work, and shows that running the complete system, with fairly advanced decisional capabilities, is now possible on regular hardware.

This architectural concept has a more general applicability. It has been successfully implemented on other experimental mobile robots operating indoors, with different navigation capacities, and has also been considered for autonomous spacecraft operations [50], [51].

Ongoing work and research on the LAAS architecture and its various components are now focused on the dependability aspects of the system. For example we are adding formal temporal and timed

[11]For the download_images and communicate actions, specific procedures simulate the visibility windows and the gradual download of images.

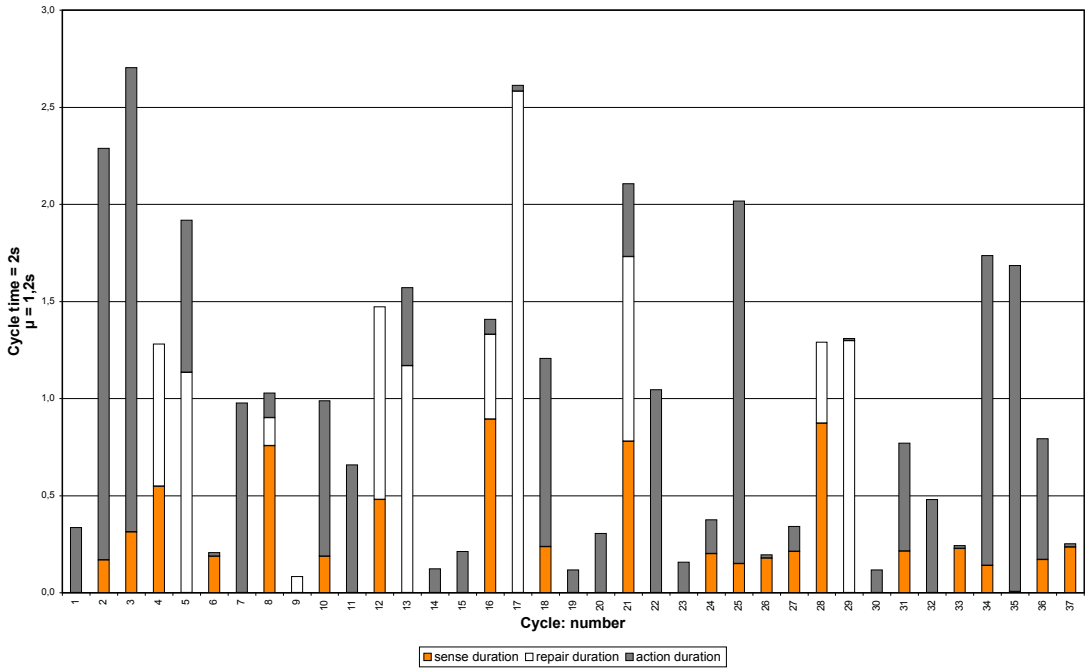

Fig. 18. Cycle duration of plans. *Sense* corresponds to the part of the cycle used to integrate new information in the plan, *repair* is self explanatory, and *action* consists in determining which timepoint has to be executed and processing them or detecting time-out. One can see repairs over more than one cycle (12-13, 28-29). Note that overall, the cycle is kept under the 3 seconds time cap.

models to the functional level modules, in order to automatically produce a controller that is safe by design. Similarly, we are studying ways to improve the robustness of the planning system with respect to action model errors.

ACKNOWLEDGMENT

The authors wish to acknowledge the participation of the following researchers in the work described here: Rachid Alami, Raja Chatila, Matthieu Gallien, Sara Fleury, Matthieu Herrb, Anthony Mallet.

REFERENCES

[1] J. Fox and S. Das, *Safe and Sound, Artificial Intelligence in Hazardous Applications*, ISBN 0-262-06211-9. The American Association for Artificial Intelligence Press, 2000.

[2] R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The claraty architecture for robotic autonomy," in *IEEE 2001 Aerospace Conference*, March 2001.

[3] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons, and T. Estlin, "Claraty and challenges of developing interoperable robotic software," in *International Conference on Intelligent Robots and Systems (IROS)*, Nevada, Oct. 2003, invited paper.

[4] M.G. Bualat, G.C. Kuntz, A.R. Wright, and I.A. Nesnas, "Developing an autonomy infusion infrastructure for robotic exploration," in *Proceedings of the 2004 IEEE Aerospace Conference*, Big Sky, Montana, Mar. 2004.

[5] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *IJRR*, 1998.

[6] S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila, "Autonomous rover navigation on unknown terrains: Functions and integration," *International Journal of Robotics Research*, vol. 21, no. 10-11, pp. 917–942, Oct-Nov. 2002.

[7] L. Pedersen, M. Bualat, C. Kuntz, R. Sargent, R. Washington, and W. Wright, "Instrument deployment for mars rovers," in *IEEE International Conference on Robotics and Automation, Taipei (Taïwan)*, Sept. 2003, pp. 2525–2542.

[8] J. Biesiadecki, C. Leger, and M. Maimone, "Tradeoffs between directed and autonomous driving on the mars exploration rovers," in *International Symposium on Robotics Research, San Francisco, CA (USA)*, Oct 2005.

[9] G. Gianfiglio, "Exomars project status," in *8th ESA Workshop on Advanced Space Technologies for Robotics and Automation, Noordwijk (The Netherlands)*, Nov. 2006.

[10] S. Lacroix and R. Chatila, "Motion and perception strategies for outdoor mobile robot navigation in unknown environments," in *Experimental Robotics IV*, O. Khatib and J.K. Salisbury, Eds., vol. 223 of *Lecture Notes in Computer and Information Science*, pp. 538–547. Springer, 1995.

[11] H. Haddad, M. Khatib, S. Lacroix, and R. Chatila, "Reactive navigation in outdoor environments using potential fields," in *International Conference on Robotics and Automation, Leuven (Belgium)*, May 1998, pp. 1232–1237.

[12] R. Simmons, E. Krotkov, L. Chrisman, F. Cozman, R. Goodwin, M. Hebert, L. Katragadda, S. Koenig, G. Krisnaswamy, Y. Shinoda, W. Whittaker, and P. Klarer, "Experience with rover navigation for lunar-like terrains," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, Pittsburgh, Pa (USA)*, 1995.

[13] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, June 1991.

[14] J. Minguez and L. Montano, "Nearness diagram (nd) navigation: collision avoidance in troublesome scenarios," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 1, pp. 45–59, Feb. 2004.

[15] A. Kelly, *An Intelligent Predictive Control Approach to the High Speed Cross Country Autonomous Navigation Problem*, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA (USA), 1995.

[16] K. Iagnemma and S. Dubowsky, "Mobile robots in rough terrain: Estimation, motion planning, and control with application to planetary rovers," in *Springer Tracts in Advanced Robotics (STAR) Series*, vol. 12. Springer, June 2004.

[17] I.S. Kweon and T. Kanade, "High-resolution terrain map from multiple sensor data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 2, pp. 278–292, Feb. 1992.

[18] P. Ballard and F. Vacherand, "The manhattan method : A fast cartesian elevation map reconstruction from range data," in *IEEE International Conference on Robotics and Automation, San Diego, Ca. (USA)*, 1994, pp. 143–148.

[19] D. Bonnafous, S. Lacroix, and T. Siméon, "Motion generation for a rover on rough terrains," in *International Conference on Intelligent Robots and Systems, Maui, Hawai (USA)*, Oct. 2001.

[20] A. Stentz and M. Hebert, "A complete navigation system for goal acquisition in unknown environments," *Autonomous Robots*, vol. 2, no. 2, Aug. 1995.

[21] S. Singh, R. Simmons, M.F. Smith, A. Stentz, V. Verma, A. Yahja, and K. Schwehr, "Recent progress in local and global traversability for planetary rovers," in *IEEE International Conference on Robotics and Automation, San Francisco, Ca (USA)*, 2000, pp. 1194–1200.

[22] J. Gancet and S. Lacroix, "Pg2p: A perception-guided path planning approach for long range autonomous navigation in unknown natural environments," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, Las Vegas (USA)*, Oct. 2003.

[23] T. Peynot and S. Lacroix, "Enhanced locomotion control for a planetary rover," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, Las Vegas (USA)*, Oct. 2003.

[24] A. Mallet, S. Lacroix, and L. Gallo, "Position estimation in outdoor environments using pixel tracking and stereovision," in *IEEE International Conference on Robotics and Automation, San Francisco, Ca (USA)*, April 2000, pp. 3519–3524.

[25] T. Lemaire and S. Lacroix, "Long term SLAM with panoramic vision," *To appear in Journal of Field Robotics*, 2007.

[26] S. Fleury, M. Herrb, and R. Chatila, "Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture," in *International Conference on Intelligent Robots and Systems*, Grenoble (France), 1997, vol. 2, pp. 842–848.

[27] A. Finzi, F. Ingrand, and N. Muscettola, "Model-based executive control through reactive planning for autonomous rovers," in *IROS 2004 (IEEE/RSJ International Conference on Intelligent Robots and Systems)*, Sendai, Japan, 2004.

[28] P. Aschwanden, V. Baskaran, S. Bernardini, C. Fry, M. Moreno, N. Muscettola, C. Plaunt, D. Rijsman, and P. Tompkins, "Model-unified planning and execution for distributed autonomous system control," in *AAAI Fall Symposium on Spacecraft Autonomy*, 2006.

[29] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *IRS*, 1998.

[30] F. Ingrand, R. Chatila, R. Alami, and F. Robert, "Prs: A high level supervision and control language for autonomous mobile robots," in *IEEE International Conference on Robotics and Automation*, Mineapolis, USA, 1996.

[31] R. Levinson, "A general programming language for unified planning and control," *Artificial Intelligence*, vol. 76, 1995.

[32] O. Despouys and F. Ingrand, "Propice-plan: Toward a unified framework for planning and execution" in *ECP*, 1999.

[33] J. Ambros-Ingerson and S. Steel, "Integrating planning, execution and monitoring," in *AAAI*, 1988.

[34] K.Z. Haigh and M.M. Veloso, "Planning, execution and learning in a robotic agent," in *AIPS*, 1998.

[35] K.L. Myers, "Cpef: Continuous planning and execution framework," *AI Magazine*, vol. 20, no. 4, pp. 63–69, Winter 1999.

[36] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, "Using iterative repair to improve the responsiveness of planning and scheduling," in *AAAI*, 2000.

[37] A. Finzi, F. Ingrand, and N. Muscettola, "Robot action planning and execution control," in *IWPSS 2004, 4th International Workshop on Planning and Scheduling for Space, June 23 - 25*, 2004.

[38] P. Kim, B. Williams, and M. Abramson, "Executing reactive, model-based programs throgh graph-based temporal planning," in *IJCAI*, 2001.

[39] R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks," in *KR*, 1989.

[40] P. Laborie and M. Ghallab, "Planning with sharable resource constraints," in *IJCAI*, 1995.

[41] R. Trinquart and M. Ghallab, "An extended functional representation in temporal planning : towards continuous change," in *ECP*, 2001.

[42] T. Vidal, P. Morris, and N. Muscettola, "Dynamic Control of Plans With Temporal Uncertainty," in *IJCAI*, Seattle, Washington, USA, 2001, pp. 494–502.

[43] S. Lemai, *IxTeT-eXeC: planning, plan repair and execution control with time and resource management*, Ph.D. thesis, LAAS-CNRS and Institut National Polytechnique de Toulouse, France, 2004.
[44] P. Klein, "The Safety Bag Expert System in the Electronic Railway Interlocking System ELEKTRA," *Expert Systems with Applications*, pp. 499–560, 1991.
[45] F. Py and F. Ingrand, "Dependable execution control for autonomous robots," in *IROS 2004 (IEEE/RSJ International Conference on Intelligent Robots and Systems)*, Sendai, Japan, 2004.
[46] F. Py, *Contrôle d'exécution dans une architecture hiérarchiése pour systèmes autonomes (in French)*, Ph.D. thesis, LAAS-CNRS and Institut National Polytechnique de Toulouse, France, 2005.
[47] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
[48] F. Ingrand and F. Py, "An execution control system for autonomous robots," in *IEEE International Conference on Robotics and Automation*, Washington DC (USA), May 2002.
[49] S. Joyeux, A. Lampe, R. Alami, and S. Lacroix, "Simulation in the LAAS Architecture," in *Proceedings of Principles and Practice of Software Development in Robotics (SDIR2005), ICRA workshop*, Barcelona, Spain, April 2005.
[50] J. Gout, S. Fleury, and H. Schindler, "A new design approach of software architecture for an autonomous observation satellite," in *Proceedings of iSAIRAS*, 1999.
[51] M-C. Charmeau G. Verfaillie, "A generic modular architecture for the control of an autonomous spacecraft," in *IWPSS 2006, 5th International Workshop on Planning and Scheduling for Space, October 22nd-25th, Space Telescope Science Institute, Baltimore, MD. USA*, 2006.