Intelligent Supervision for Robust Plan Execution

Roberto Micalizio, Enrico Scala, and Pietro Torasso

Università di Torino corso Svizzera 185, 10149 Torino
{micalizio,scala,torasso}@di.unito.it

Abstract. The paper addresses the problem of supervising the execution of a plan with durative actions in a just partially known world, where discrepancies between the expected conditions and the ones actually found may arise. The paper advocates a control architecture which exploits additional knowledge to prevent (when possible) action failures by changing the execution modality of actions while these are still in progress. Preliminary experimental results, obtained in a simulated space exploration scenario, are reported.

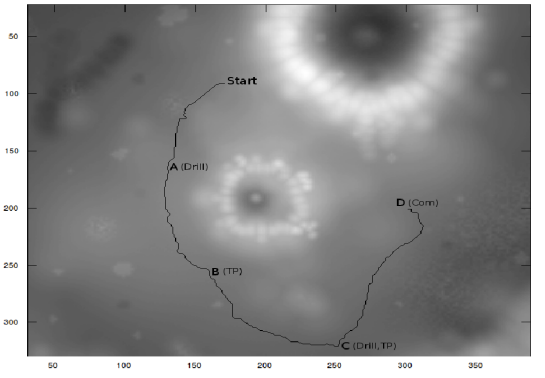
Keywords: Plan Execution, Intelligent Supervision, Robotic Agents, Control Architecture.

1 Introduction

In the last years a significant amount of interest has been devoted to the area currently known as "planning in the real world", in order to weaken some of the assumptions made in classical planning. While relevant results have been obtained for innovative planning techniques such as conditional and contingent planning, it is worth noting that these forms of planning may be quite expensive from a computational point of view and in many cases it is very hard to anticipate all possible contingencies, since the execution of an action can be perturbed by *unexpected* events which may cause the failure of the action itself. To handle these issues, some methodologies [1,2,3] propose to monitor the execution of a plan and to invoke a repair strategy, typically based on a re-planning step, as soon as action failures are detected. These methodologies, however, are unable to intervene during the execution of an action, as the repair is invoked just after the occurrence of an action failure, that is, when the plan execution has been interrupted. In principle, this problem could be mitigated by anticipating which actions in the plan will not be executable in the future (i.e., threatened actions, see for example [4]), so that the repair strategy can be invoked earlier. Unfortunately, it is not always possible to anticipate the set of threatened actions as the plan executor (i.e., the agent) may have just a partial knowledge of the world where it is operating.

In this paper we propose a control architecture for robust plan execution whose aim is to avoid (at least in some cases) the occurrence of action failures. To reach this goal, the proposed architecture exploits a temporal interpretation

R. Pirrone and F. Sorbello (Eds.): AI*IA 2011, LNAI 6934, pp. 151–163, 2011.



Capital letters A, B, C, D denote the sites the rover has to visit. Each site is tagged with the actions to be performed when the site has been reached: DRILL refers to the drill action, TP to take picture, and COM to data transmission. More actions can be done at the same site, see for instance target C. The black line connecting two targets is the route, predicted during a path planning phase, the rover should follow during a navigate action.

Fig. 1. An example of daily mission plan

module for detecting agent's behavioral patterns which, over a temporal window, describe deviations from the nominal expected behavior. When such potentially hazardous situations are detected, another module, the Active Controller, can decide to change the modality of execution of the current action by taking into account the capability of alternative modalities in alleviating the discrepancy between the actual behavior and the nominal one.

The paper is organized as follows: section 2 describes a space exploration scenario used to exemplify the proposed approach; section 3 presents a basic control architecture which just reacts to action failures, an improved architecture which tries to prevent failures is discussed in section 4; section 5 reports some preliminary experimental results; finally, in section 6, the conclusions.

2 A Motivating Example

This section introduces a space exploration scenario, where a mobile robot (i.e., a planetary rover) is in charge of accomplishing explorative tasks. This scenario presents some interesting and challenging characteristics which made it particularly interesting for the plan execution problem. The rover, in fact, has to operate in a hazardous and not fully observable environment where a number of unpredictable events may occur.

In our discussion, we assume that the rover has been provided with a mission plan covering a number of scientifically interesting sites: the plan includes navigation actions as well as exploratory actions that the rover has to complete once a target has been reached; for instance the rover can:
- drill the surface of rocks;
- collect soil samples and complete experiments in search for organic traces;
- take pictures of the environment.

All these actions produce a certain amount of data which are stored in an onboard memory of the rover until a communication window towards Earth becomes available. In that moment the data can be uploaded; see [5] for a possible

solution tackling the communication problem in a space scenario. For example, a possible daily plan involves: `navigate(Start,A); drill(A); navigate(A,B); tp(B); navigate(B,C); drill(C); tp(C); navigate(C,D); com(D)`. This plan is graphically represented in Figure 1 where a map of a portion of the Martian soil is showed.[1]

It is easy to see that some of these actions can be considered atomic (e.g., take picture), some others, instead, will take time to be completed. For instance, a navigate action will take several minutes (or hours), and during its execution the rover moves over a rough terrain with holes, rocks, slopes. The safeness of the rover could be threatened by too deep holes or too steep slopes since some physical limits of the rover cannot be exceeded. In case such a situation occurs, the rover is unable to complete the action. Of course, the rover's physical limits are taken into account during the synthesis of the mission plan, and regions presenting potential threats are excluded *a priori*.

However, the safeness of the rover could also be threatened by terrain characteristics which can hardly be anticipated. For instance, a terrain full of shallow holes may cause high-frequency vibrations on the rover, and if these vibrations last for a while they may endanger some of the rover's devices. This kind of threat is difficult to anticipate from Earth both because satellite maps cannot capture all terrain details, and because this threat depends on the rover's contextual conditions, such as its speed.

In the following of the paper we propose a control architecture which recognizes potential threats while actions are still under way, and reacts to them by tuning the *execution modality*. For instance, the navigation action can be associated with two execution modalities: high-speed and reduced-speed; slowing down the rover's speed can mitigate the harmful effects of disconnected terrains. As we will see, the solution we propose is sufficiently flexible to change the execution modality not only when threats have been detected, but also when threats terminate and nominal execution modalities can be restored.

3 Basic Control Architecture

As said above, plan execution monitoring becomes a critical activity when a given plan is executed in the real world; differences between the (abstract) world assumed during the planning phase and the actual world may lead, in some cases, to a failure in plan execution. Monitoring the plan execution is hence necessary but it is just the first step (detecting plan failures): a plan repair mechanism should be subsequently activated in order to restore (if possible) nominal conditions. Many plan repair techniques rely on a (re)planning phase to overcome the failure of an action. In some cases, however, this technique may be difficult to apply and too costly, so it should be limited as far as possible. For this reason, in the following we propose a control architecture which tries to limit the necessity of replanning by preventing, as far as possible, the occurrence of plan failures.

[1] In the picture, different altitudes are represented in a grey scale where white corresponds to the highest altitude, and black to the lowest.

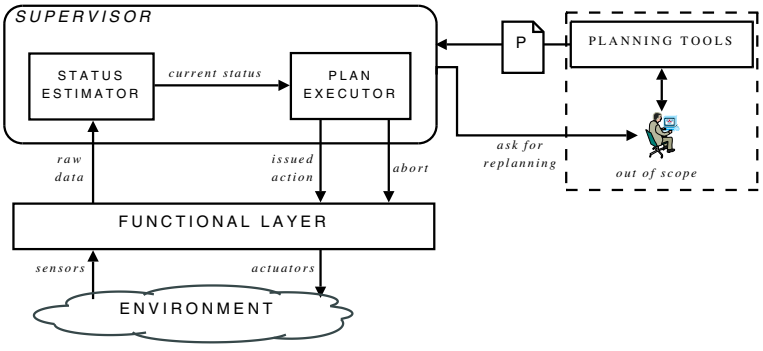


Fig. 2. A basic control architecture

```
(:durative-action navigate
parameters : (?r - rover ?y - site ?z - site)
duration : (= ?duration navigation-time ?z ?p)
condition : (and(at start(at ?r ?y)
        (over all (and(≤ (pitch-derivative ?r) 5)
              (≤ (roll-derivative ?r) 5)
              (≤ (pitch ?r) 30)
              (≤ (roll ?r) 30))))
effect :(and(at start(not (at ?r ?y)))
              (at end(at ?r ?z)))
```

Fig. 3. An example of durative action

```
01 while there are actions in P to be performed
02   rdata_t ← getRawDataFromFL(t)
03   state_t ← StateEstimator(rdata_t)
04   if (checkInvariants(inv_a, state_t) = violation
       ∨ actualDuration(a) > duration(a))
05       send abort to FL
06       ask for replanning
07   if checkOutcome(eff_a, state_t) = succeeded
08       a ←getNextAction(P)
09       if checkPreconditions(pre_a, state_t) =
                    not satisfied
10           ask for replanning
11       else submit a to FL
12   t ← t + 1
```

Fig. 4. The basic control strategy

For the sake of exposition, before presenting the complete control architecture, we introduce a basic version which just detects failures and reacts to them by aborting the current execution (see Figure 2).

We assume that the given mission plan P is a totally ordered sequence of action instances, which are modeled in PDDL 2.1 [6] (this formalism, in fact, allows to deal with atomic as well as durative actions). Note that, besides preconditions and effects, PDDL 2.1 allows the definition of invariant conditions which the planner must guarantee to maintain during the synthesis of the mission plan. These invariant conditions are exploited in our approach not only in the planning phase, but also to check whether the rover's safeness conditions are maintained during the plan execution. For instance, the "over-all" construct of the navigate action shown in Figure 3 specifies which conditions on the rover's attitude (i.e., the combination of pitch and roll) are to be considered safe, and hence must hold during the whole execution of the action.

The basic architecture of Figure 2, includes two main levels: the Supervisor and the Functional Level (FL). The Supervisor is in charge of managing the execution of the plan P and its control strategy is reported in Figure 4. At each time instant t, the State Estimator gets the raw data provided by the FL and produces an internal representation of the current rover's state possibly by

making qualitative abstractions on the raw data. The Plan Executor matches the estimated state $state_t$ against the invariant conditions of the action a currently in execution: in case such conditions are not satisfied or the execution of the action a lasts over the duration indicated in the plan, the action is considered failed. In this case, the Plan Executor aborts the execution (by sending an appropriate abort command to the FL) and asks for a new repair plan P'. In case $state_t$ is consistent with the action model, the Plan Executor establishes whether a has been completed with success (of course, the outcome of a is *success* when the expected effects eff_a hold in $state_t$); in the positive case, the next action in P becomes the new current action to be performed. The new action is actually submitted for execution only after the validation of its preconditions against the current state; in case the action is not executable, the Plan Executor asks for a recovery plan.

The second level of the architecture is the FL, which, from our point of view, is an abstraction of the rover's hardware able to match the actions issued by the Supervisor into lower level commands for the rover's actuators. In doing so, the FL may exploit services such as localization, obstacle avoidance, short range path planning, path following (see [7,8]).

4 Improving the Control

To be more effective, the Supervisor must be able to anticipate plan failures and actively intervene during the execution, not only for aborting the current action, but also for changing the way in which that action is going to be performed. Unfortunately, the pieces of information contained in the mission plan are not sufficient for this purpose and the Supervisor needs additional sources of information complementing the ones in the plan.

4.1 Knowledge for the Active Control

Execution Trajectories. The first extension we introduce is closely related to the actual execution of an action. In the PDDL2.1 model, in fact, one just specifies (propositional) preconditions and effects, but there may be different ways to achieve the expected effects from the given preconditions. For instance, the action $navigate(A, B)$ just specifies that: 1) the rover must be initially located in A and 2) the rover, after the completion of the navigate action, will be eventually located in B; but nothing is specified about the intermediate rover positions between A and B. This lack of knowledge is an issue when we consider the problem of plan execution monitoring. For the monitoring purpose, in fact, it becomes important to detect erroneous behaviors while the action is still under execution. For this reason, we associate each durative action instance a with a parameter trj_a, that specifies a trajectory of nominal rover states. More formally, trj_a=$\{s_0, \ldots, s_n\}$, where s_i $(i : 0..n)$ are, possibly partial, rover states at different steps of execution of a. We just require that both $s_0 \vdash pre_a$ and $s_n \vdash eff_a$ must hold. Therefore, trj_a represents how the rover state should

evolve over time while it is performing a. For example, let a be *navigation(A,B)*, trj_a maintains a sequence of waypoints which sketches the route the rover has to follow. Of course, the actual execution of the navigation action may deviate from the given trajectory for a number of reasons (e.g., unexpected obstacles may be encountered along the way). In principle, a deviation from the nominal trajectory does not necessarily represent an issue; in our extended approach, the Supervisor takes the responsibility for tracking these deviations and deciding when they signal anomalies to be faced.

Temporal Patterns. The trajectory associated with an action instance traces a preferable execution path, but it is not sufficiently informative to detect potentially dangerous situations. For example, even though the robot is accurately following the trajectory associated with a navigation action, the safeness of the rover could be endangered by a terrain that can be rougher than expected. Taking into account just the invariant conditions associated with the navigation may not prevent action failures; these conditions, in fact, represent the physical limits the rover should never violate, and when they are violated any reaction may arrive too late. To avoid this situation, the Supervisor must be able to anticipate anomalous conditions before they become so dangerous to trigger an abort. In our approach we associate each action type with a set temporal patterns that describe how the rover should, or should not, behave while it is performing a specific action. Differently from a trajectory, the temporal patterns are defined on sequences of events which abstract relevant changes in the rover state. In the paper we propose the adoption of the chronicles formalism [9] for encoding these temporal patterns. Intuitively, a chronicle is a set of events, linked together by time constraints modeling possible behaviors of a dynamic system over time. The occurrence of events may depend both on the activities carried on by the system itself and on the contextual conditions of the environment where the system is operating.

Execution Modalities. The last extension we introduce consists in associating each action type with a set of *execution modalities*. An execution modality does not interfere with the expected effects of the action; it just represents an alternative way for reaching the same effects. The basic idea is that, while the temporal patterns can be used to anticipate dangerous conditions, the execution modalities could be used to reduce the risk of falling in one of them. For example, a navigate action is associated with the set of execution modalities $mods(navigate)$={*nominal-speed*, *reduced-speed*}. It is easy to see that in both cases the rover reaches the expected position, but the two modalities affect the navigation in different ways.

4.2 Improved Control Architecture

Relying on the additional pieces of knowledge discussed above, we propose the improved control architecture depicted in Figure 5; three new modules have been added: the State Interpreter (SI), the Temporal Reasoner (TR), and the Active

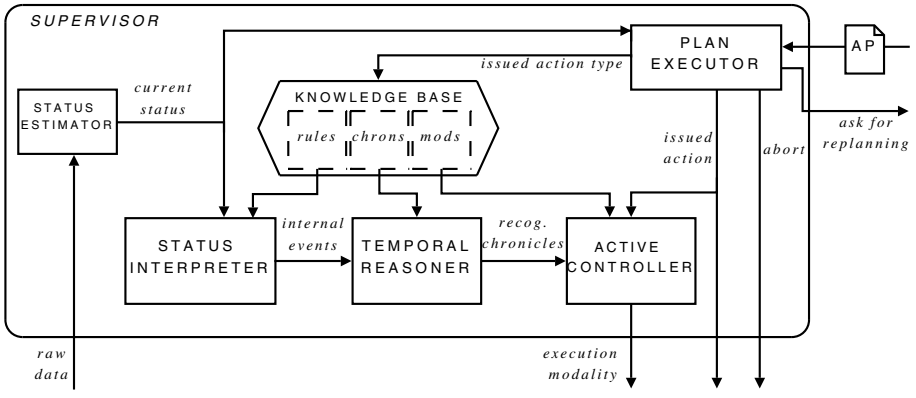


Fig. 5. The improved control architecture

Controller (AC); moreover, a Knowledge Base (KB) is also added to provide the modules with the knowledge associated with a specific action type.

The Supervisor receives in input a plan AP (i.e., an augmented plan where each action is provided with the trj_a parameter discussed above). The actual execution of AP is under the control of the Plan Executor (PE), as in the basic architecture (see Figure 4). The first improvement to the PE strategy is the check that the current $state_t$ is consistent with the constraints imposed by the trajectory trj_a. This improvement is implemented by changing the line 04 in Figure 4 as follows:

if (checkInvariants(inv_a, $state_t$) = *violation*) $\vee$ (actualDuration(a) > duration(a)) $\vee$ (trajectoryDeviations(trj_a, $state_t$)=*relevant*)

In this way, the PE emits an abort also when the execution of a deviates significantly from the expected trajectory trj_a.[2]

A second and more relevant improvement is about the exploitation of the temporal patterns associated with action a. Since the evaluation of the current execution w.r.t. relevant temporal patterns is a complex activity which requires the coordination of different modules and the decision to changing execution modality (when required), we summarize this process in the macro function *ActiveMonitoring* (depicted in Figure 6). The PE, responsible for the coordination of the internal modules of the Supervisor, invokes *ActiveMonitoring* just before the assessment of an action outcome (see the algorithm in Figure 4, line 07). In the following, we first describe the idea at the basis of the *ActiveMonitoring* and then we sketch how each involved module actually operates.

As said above, *ActiveMonitoring* is aimed at emitting an execution modality relying on the set of chronicles that have been recognized at a given time instant. Since chronicles capture events, it is up to the *StateInterpreter* module to look at the history of the rover state for generating internal events which highlight

[2] Since in this paper we are more interested in the problem of correcting the execution by means of the selection of an appropriate execution modality, we do not provide further details about *trajectoryDeviations*.

ActiveMonitoring(a, t, $state_t$)
  $H \leftarrow$ append(H, $state_t$)
  $rules_a \leftarrow$ get-interpretative-rules(KB, $acttype(a)$)
  $events_t \leftarrow$ **StateInterpreter**(H, $rules_a$)
  $RC \leftarrow \emptyset$
  $chronicles_a \leftarrow$ get-chronicles(KB, $acttype(a)$)
  for each event $e_t \in events_t$
    $chr_a \leftarrow$ get-relevant-chronicle(e_t, $chronicles_a$)
    if **TemporalReasoner**(chr_a, e_t) **emits** *recognized*
      $RC \leftarrow RC \cup \{chr_a\}$
  if $RC \neq \emptyset$
    $mods_a \leftarrow$ get-execution-modalities(KB, $acttype(a)$, RC)
    ActiveControl(RC, $mods_a$) **emits** mod to FL

Fig. 6. The strategy for the active monitoring

relevant changes in the rover state. This process is performed in the first three lines of *ActiveMonitoring*: *StateInterpreter* module generates at each time t the set of internal events $events_t$. Each event $e_t \in events_t$ is subsequently sent to the *TemporalReasoner* (i.e., a *CRS*), which consumes the event and possibly recognizes a chronicle chr_a. All the chronicles recognized at time t are collected into the set RC, which becomes the input for the *ActiveController*. This last module has the responsibility for selecting, among a set of possible execution modalities $mods_a$, a specific modality to be sent to the FL.

The State Interpreter generates the internal events by exploiting a set of interpretative rules in $rules(atype)$ (where $atype$=$acttype(a)$). These interpretative rules have the form *Boolean condition* $\rightarrow$ *internal event*. The Boolean condition is build upon three basic types of atoms: state variables x_i, state variable derivates $\delta(x_i)$, and abstraction operators $qAbs(x_i, [t_l, t_u]) \rightarrow qVals$ which map the array of values assumed by x_i over the time interval $[t_l, t_u]$ into a set of qualitative values $qVals = \{qval_1, \ldots, qval_m\}$. For example, the following interpretative rules:
$(\delta(roll) > limits_{roll} \vee \delta(pitch) > limits_{pitch}) \rightarrow severe\text{-}hazard(roll, pitch)$
is used to generate a *severe-hazard* event whenever the derivate value of either roll or pitch exceeds predefined thresholds in the current rover state.
Another example is the rule:

$$attitude(roll, [t_{current} - \Delta, t_{current}]) = \text{nominal} \wedge$$
$$attitude(pitch, [t_{current} - \Delta, t_{current}]) = \text{nominal} \rightarrow safe(roll, pitch)$$

Where *attitude* is an operator which abstracts the last Δ values of either roll or pitch (the only two variables for which this operator is defined) over the set {nominal, border, non-nominal}.

Note that set of internal events can be partitioned according to the apparatus they refer to; for instance, the *attitude* and *severe-hazard* refer to the rover's mobility; *low-power* instead refers to the rover's battery. We assume that at each time t, $events_t$ can maintain at most one event referring to a specific device.

```
chronicle plain-terrain {
 occurs( (N, +oo), plain-conditions[pitch,
      roll], (t, t+W) )
 when recognized {
  emit event(plain-terrain[pitch,roll],t);
 }
}
```

```
chronicle hazardous-terrain {
event(medium-hazard[pitch, roll], t1 )
event(medium-hazard[pitch, roll], t2 )
event(severe-hazard[pitch, roll], t3)
t1<t2<t3 ; t2-t1<W1 ; t3-t2<W1
when recognized { emit
  event(hazardous-terrain[pitch,roll],t)} }
```

Fig. 7. Two chronicle examples in the space exploration scenario

The Temporal Reasoner is essentially a Chronicle Recognition System (CRS) similar to the one proposed by Dusson in [9]. For simplicity, in our approach we assume that each event e_t can be consumed by exactly one active chronicle *chr*; the function *get-relevant-chronicle* in Figure 6 selects such a chronicle from $chronicles_a$ so that the TR receives in input just the event e_t which can be analyzed by the appropriate chronicle.

A chronicle example associated with the navigation action is given in Figure 7; it allows the Supervisor to indentify a potentially hazardous terrain. This chronicle is recognized when at least N *severe-hazard* events (regarding the parameters `pitch` and `roll`) have been detected within an interval of W time instant. The basic idea is that the safeness of the rover may be endangered when it moves at a high speed along a too rough terrain; this kind of threat can be captured by detecting hazardous variation of the roll and pitch parameters in a short time window. Indeed, the event *hazard*, resulting from an interpretation process over the rover's state variables, denotes that, although the rover's state has not violated the physical constraints (and hence it is still nominal), it may become anomalous in the near future.

The Active Controller accomplishes two important activities. First, it selects an execution modality to be issued towards the FL. In principle, such a selection should correct the current robot's behavior smoothly; that is, on one side, the AC's strategy should not be too reactive in order to avoid abrupt changes in the robot's behavior which may be as dangerous as the threat to face; and on the other side, the AC should be able to restore the nominal execution modalities when it is reasonable to presume that no menace is expected in the near future. In our current and preliminary solution, however, the AC is still purely reactive matching a recognized chronicle with a specific execution modality. Second, the AC updates some parameters of the current action according the execution modalities it emits. For instance, when a navigation action is slowed down, it will take more time to be completed, this extra time must be taken into account by the PE during its job. Due to lack of space, we cannot provide further details on this point.

Running example Let us consider the action `navigate(A,B)`, and let us assume that the actual terrain is rougher than expected and causes repeated vibrations while the rover is moving from `A` to `B` . The basic architecture would handle this situation by aborting the navigate action, this would have a dramatic

effect on the mission plan as the following actions could not be performed. On the contrary, the improved architecture is able to anticipate the threat and to intervene by slowing down the rover, this change of modality reduce the abrupt changes in pitch and roll, so that the action can be completed with success. It is worth noting that the execution modality is changed again (returning to nominal) when the pitch and roll parameters are nominal for a while (see chronicle `plain-terrain` which takes care of this temporal pattern).

5 Experimental Results

The experimental scenario. The approach described in the paper has undergone to a first validation by using as test bed the space exploration scenario previously introduced. The planetary environment has been represented as a Digital Elevation Model (DEM); we assumed that an initial DEM D_{init}, presumably computed from satellites images, is available, and we used it for synthesizing a set of rover's missions. In particular, by taking into account the terrain's characteristics, we have subdivided the rover's missions into two classes: *easy* and *difficult*. Note that the planning phase verifies the feasibility of each navigate action by invoking a path planner that, relying on D_{init}, assesses the validity of the invariant conditions associated with this action type (see Figure 3) and provides also a trajectory in terms of way points.

Obviously, D_{init} is just an approximation of the real terrain, therefore the actual execution of a mission plan may be affected by unexpected environmental conditions. For simulating the discrepancies between D_{init} and the real terrain, we have altered the original DEM by adding a random noise on the altitude of each cell. In our experiments, we have considered 6 noise degrees: from 10 cm to 15 cm, and for each of them we have generated 320 cases: 160 for the *easy* class and 160 for the *difficult* one.

Altogether, in our experiments we have considered up to 1920 navigate actions differing with one another for their starting and ending points, and their length.

To prove the effectiveness of our control architecture, we have simulated the execution of both *easy* and *difficult* cases in each noisy DEM comparing the responses of the two architectures, the basic and the improved, presented above.

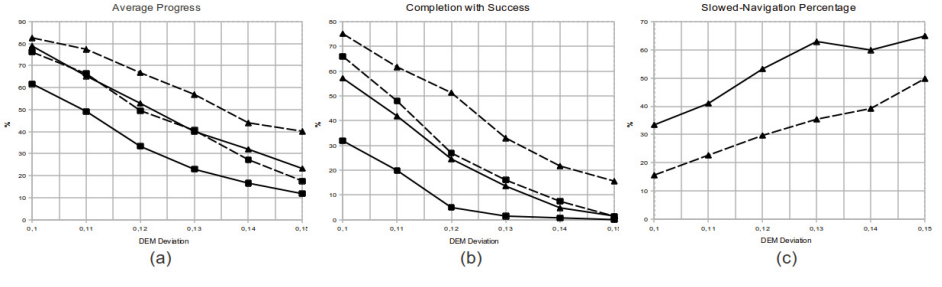


Fig. 8. Experimental results

A simplified simulator of the FL has been implemented in order to generate with a frequency of 1Hz the set of raw data the Supervisor (either basic or improved) has to interpret. For measuring the robustness of the plan execution and for providing some insights in the ability of the Supervsior in tolerating variations in the DEM, we are reporting data about three main parameters concerning the execution of the *navigate* actions:

1) the percentage of navigate actions that were completed successfully.
2) the percentage of progress actually done by the rover with respect to the whole trajectory, computed taking into account both the navigations that were actually completed and the aborted ones.
3) The percentage of steps the navigation has been performed in the slowdown modality w.r.t. the whole trajectory. Of course this datum is relevant just for the improved architecture.

Figure 8 summarizes the results of the tests. The graphs show the average values for the class of difficult cases (solid line), and for the class of the easy ones (dashed line). Each bullet corresponds to the average value of 160 navigations; squares denote the responses of the basic architecture, triangles denote the responses the improved architecture. It is easy to see that the improved architecture always provides better results than the basic one as concerns both the percentage of success and the progress. As expected, in the *difficult* cases, the gains are significant even for small DEM deviations, whereas in the *easy* ones, the gain becomes relevant for larger deviations. The results also show that the mechanism of active control is quite powerful but cannot avoid failures when the noise degree grows too much. A final remark concerns the cost of the intelligent monitoring: while the computational cost is negligible, there is an impact on the actual execution that we estimate as the percentage of steps performed in reduced-speed modality, showed in Figure 8.c. It is easy to see that this percentage is proportional to the noise degree and to hardness of the navigation.

6 Discussion and Conclusions

This paper addresses the problem of robust plan execution, when the environment may (slightly) differ from the one known (or assumed) during the planning phase and unexpected contingencies may arise. Previous works in literature have faced this problem by endowing the plan executor (e.g., a mobile robot) with some form of autonomous behavior. For instance, the control architectures discussed in [7,8,10] support the robot's autonomy by means of three layers of control: the highest one is devoted to the decisional aspects and it is typically based on one (or even more) (re)planning module(s).

Recent works on planning have faced the problem of recovering from an action failure by synthesizing a repairing plan on the fly (e.g., see [3,2,1]). These approaches, however, have been designed to intervene only after a failure has occurred (and therefore when the plan execution has been interrupted). In this

paper we propose a control architecture aimed at reducing the necessity of invoking a replanner by preventing, when possible, the occurrence of an action failure. In particular, we have shown how the formalism of chronicles [9] can be used to model patterns of the robot's behavior over a temporal window, and how these patterns are subsequently exploited for anticipating threats. The chronicles formalism represents a viable and efficient solution to the problem of interpreting online the raw data coming from the environment, and hence reasoning about the environment in more abstract terms.

In order to keep potential threats under control, we propose to correct the current robot's behavior through the selection of execution modalities, whose effect is to change the way in which the current action is actually carried on while an action is still in execution. The proposed methodology is therefore a way to enhance the robot's autonomy as it can flexibly switch among modalities according to its contextual conditions. This idea is not completely new, also in [8] the authors suggest a methodology for adjusting the way in which an action is carried on depending on the rover's context. For instance, their navigation has three modalities related to the rover's speed: low, medium, and high. However, their context is just a snapshot (i.e., a set of variables) of the current conditions; conversely, we propose to maintain a "temporal" context by means of chronicles. This allows us to predict how the context will evolve, and hence to anticipate a change of modality. With regard to this point, a central role is played by the Active Controller; in its current preliminary implementation, the AC selects just one execution modality at each time instant. As future work we intend to extend the functionality of this module by allowing the selection of multiple modalities; to reach this result, however, the AC needs to know the dependencies existing between different modalities in order to estimate how their effects could interfere with each other. We are currently investigating the adoption of (probabilistic) causal networks as a possible way to face this challenge.

References

1. Bouguerra, A., Karlsson, L., Saffiotti, A.: Monitoring the execution of robot plans using semantic knowledge. Robotics and Autonomous Systems 56, 942–954 (2008)
2. Bozzano, M., Cimatti, A., Roveri, M., Tchaltsev, A.: A comprehensive approach to on-board autonomy verification and validation. In: ICAPS 2009 Workshop on Verification and Validation of Planning and Scheduling Systems (2009)
3. Micalizio, R.: A distributed control loop for autonomous recovery in a multi-agent plan. In: Proc. IJCAI 2009, pp. 1760–1765 (2009)
4. Micalizio, R., Torasso, P.: Monitoring the execution of a multi-agent plan: Dealing with partial observability. In: Proc. ECAI 2008, pp. 408–412 (2008)
5. Musso, I., Micalizio, R., Scala, E., et al.: Communication scheduling and plans revision for planetary rovers. In: Proc. of i-SAIRAS 2010 (2010)
6. Fox, M., Long, D.: Pddl2.1: An extension to pddl for expressing temporal planning domains. Journal of Artificial Intelligence Research 20, 61–124 (2003)
7. Alami, R., Chatila, R., Fleury, S., Ghallab, M., Ingrand, F.: An architecture for autonomy. International Journal of Robotics Research 17(4), 315–337 (1998)

8. Calisi, D., Iocchi, L., Nardi, D., Scalzo, C., Ziparo, V.A.: Context-based design of robotic systems. Robotics and Autonomous Systems (RAS) - Special Issue on Semantic Knowledge in Robotics 56(11), 992–1003 (2008)
9. Dousson, C., Le Maigat, P.: Chronicle recognition improvement using temporal focusing and hierarchization. In: IJCAI 2007, pp. 324–329 (2007)
10. Nesnas, I.A.: Claraty: A collaborative software for advancing robotic technologies. In: Proc. of NASA Science and Technology Conference (2007)