TOWARDS ROBUST EXECUTION OF MISSION PLANS FOR PLANETARY ROVERS

Roberto Micalizio, Enrico Scala, and Pietro Torasso

Dipartimento di Informatica, Università di Torino, corso Svizzera 185 - 10149 Torino (Italy), {micalizio, scala, torasso}@di.unito.it

ABSTRACT

The paper discusses an on-line control architecture for plan execution whose main aim is to prevent (at least in some cases) the occurrence of action failures. To reach this result, the plan to be executed has been enriched with additional knowledge about the intermediate conditions (i.e., invariant conditions), which must be satisfied during the execution of (durative) actions. This knowledge is used by a Temporal Reasoner to detect anomalous situations that may endanger the safeness of the plan executor. Whenever an anomaly has been detected, the proposed control architecture tries to prevent a failure by changing the execution modality of the action while it is still in progress. Preliminary experimental results, obtained in a simulated space exploration scenario, are reported.

Key words: Plan Execution, Intelligent Supervision, Robotic Agents, Control Architecture.

1. INTRODUCTION

Planning the activities of a mobile robot is a complex task, which many planners address by means of abstract models of the real world. In recent years there has been a substantial amount of work for reducing the gap between these abstract models and the real world, and innovative planning techniques have been proposed in literature. Nevertheless, the problem of plan execution in not completely known or observable environments (such as the space exploration scenario) still remains open and techniques for assuring robust execution are required. Some recent methodologies [BKS08, BCRT09, Mic09] face the problem of plan execution by establishing a closed loop of control including (among others) on-line monitoring, which is responsible for the detection of action failures, and plan repair, which is typically based on a re-planning step and it is devoted to restore (if possible) the nominal execution conditions. 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 a space exploration domain, however, there are many reasons why it may be difficult to adopt control methodologies just based on re-planning. First of all, the computational power onboard of a planetary rover is in general insufficient to solve a complex task such as the synthesis of a new mission plan. More important, in this scenario a mission plan is the result of a long process involving a team of human experts: technicians and scientists have to cooperate to build a plan which reaches relevant scientific targets without endangering the rover integrity and taking into account resource consumption and physical limits; any change to the mission plan is therefore negotiated among team members. Due to such a complexity in synthesizing a mission plan, the rover is typically not allowed to significantly deviate from the given plan, and hence, in case of action failures, the rover can just interrupt the execution and wait for instructions (i.e., a new mission plan) from a Ground Control Station (GCS) on Earth.

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, we associate each plan action with a set of execution modalities which, similarly to [CIN08], represent alternative ways to complete a given action. When an action is submitted for the execution, the proposed architecture is in charge of establishing, according to the current context ([CIN08]), the initial execution modality of that action. Such an initial modality, however, can be adjusted on-line while the action is still in progress. In fact, during the actual action execution, the control architecture propose exploits a temporal interpretation module for detecting agent’s behavioral patterns which describe deviations from the nominal expected behavior over a time window. When potentially hazardous situations are detected, another module, the Active Controller, can decide to adjust the current execution modality 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 on-board memory of the rover until a communication window towards Earth becomes available. In that moment the data can be uploaded; see [MMSea10] 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.\(^1\)

\(^1\)In the picture, different altitudes are represented in a grey scale where white corresponds to the highest altitude, and black to the lowest.

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 \textit{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 \textit{execution modality}. For instance, the navigation action can be associated with three execution modalities: cruise-speed (the nominal one), high-speed and reduced-speed. In the following, we will show how having such a set of modalities allows us to flexibly respond to the contextual conditions: on the one hand, we can reduce the rover’s speed in order to mitigate the harmful effects of a disconnected terrain; on the other hand, we can restore the cruise-speed (or even set the high-speed) when the terrain conditions get better. 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 replanning 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 the occurrence of plan failures. Before that, however, we need to characterize the notion of mission plan. More precisely, we assume that the given mission plan $P$ is a totally ordered sequence of action instances, which are modeled in PDDL 2.1 [FL03] (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. 

3. BASIC CONTROL ARCHITECTURE

For the sake of exposition, before presenting the complete control architecture, we introduce a basic version, see Figure 2, which just detects failures and reacts to them by aborting the current execution. Such an architecture includes two main levels: the Supervisor and the Functional Level (FL). The Supervisor is in charge of managing the execution of the plan $P$. More precisely, the Status Estimator (SE) gets the raw data provided by the FL and produces an internal representation of the current rover’s status possibly by making qualitative abstractions on the raw data. The Plan Executor (PE) is responsible for the actual execution of the actions in the given mission plan. This means that the PE must be able to determine: first, when an action can be submitted for execution to the FL (i.e., when the action preconditions are satisfied in the state estimated by the SE), and second, when an action has been carried out successfully. Whenever the PE detects an anomalous condition (e.g., an action takes longer than expected) it aborts the execution (by sending an appropriate abort command to the FL) and asks for a new repair plan. Further details about the control strategy will be provided in the next section about the improved control architecture.

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, motion planning, path following (see [ACF+98, CIN+08]).

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 $\text{navigate}(A, B)$ 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 $tr_{ja}$, that specifies a trajectory of nominal rover states. More formally, $tr_{ja} = \{ s_0, \ldots, s_n \}$, where $s_i (i : 0..n)$ are, possibly partial, rover states at different steps of execution of $a$. We just require that both $s_0 + \text{pre}_a$ and $s_n + \text{eff}_a$ must hold. Therefore, $tr_{ja}$ represents how the rover status should evolve over time while it is performing $a$. For example, let $a$ be $\text{navigation}(A, B)$, $tr_{ja}$ maintains a sequence of waypoints which sketches the route the rover
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 of 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 status. In the paper we propose the adoption of the chronicles formalism [DLM07] 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 \( \text{mods}(\text{navigate}) = \{\text{cruise-speed}, \text{reduced-speed}, \text{high-speed}\} \). 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 4; three new modules have been added: the Status Interpreter (SI), the Temporal Reasoner (TR), and the Active 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 2).

Figure 5 shows the high-level strategy of the PE which involves two main cases: the case in which a new action has to be submitted (lines 06 through 11), and the case in which an action is still in execution (lines 12 through 19) and the Supervisor has to decide when and how intervening. Underlined lines represent substantial improvements w.r.t. the basic, purely reactive, approach previously discussed.

More precisely, a first improvement is represented by the exploitation of the trajectory \( trj_a \) in order to assess how far the action execution is deviating from the expected behavior (see line 12 of the algorithm). The PE is therefore able to emit 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 func-

\(^2\)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 \( trajectoryDeviations \).
chrons mods

ask for replanning and exit
t
send abort to FL
Submit
18
16
if
13
12
rdata
04
a
07
if
06
status
05
if
01
t
(a
ask for replanning and exit
currentmod
03
this step is very similar to the "context aware" modality into account the current status of the rover (see line 10), Note that when a new action is submitted for execution,
the PE determines an initial execution modality taking
of the set of chronicles that have been recognized at a given time instant. Since chronicles
capture events, it is up to the Status Interpreter module to look at the history of the rover status for generating
relevant changes in the rover status. This process is performed in the first three lines of ActiveMonitoring: Status Interpreter 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

ActiveMonitoring($a, t, status_t, currentmod_a$)

$H \leftarrow \text{append}(H, status_t)$
rules$_a \leftarrow$ get-interpretative-rules($KB, actype(a))$

$events_t \leftarrow \text{StatusInterpreter}(H, rules_a)$

$RC \leftarrow \emptyset$

chronicles$_a \leftarrow$ get-chronicles($KB, actype(a)$)

for each event $e_t \in events_t$

$\text{chr}_a \leftarrow$ get-relevant-chronicle($e_t, chronicles_a$)

if $\text{TemporalReasoner}(\text{chr}_a, e_t)$ emits recognized

$RC \leftarrow RC \cup \{\text{chr}_a\}$

if $RC \neq \emptyset$

mods$_a \leftarrow$ get-execution-modalities($KB, actype(a), RC$)

$newmod_a \leftarrow \text{ActiveController}(RC, mods_a, currentmod_a)$

$currentmod_a \leftarrow$ newmod$_a$

submit $currentmod_a$ to FL

figure 4. the improved control architecture

Figure 5. The Plan Executor’s high-level control strategy.

Figure 6. The strategy for the active monitoring.

tion ActiveMonitoring (depicted in Figure 6). The PE, responsible for the coordination of the internal modules of the Supervisor, invokes ActiveMonitoring in line 20. Note that when a new action is submitted for execution, the PE determines an initial execution modality taking into account the current status of the rover (see line 10), this step is very similar to the “context aware” modality selection discussed in [CIN’08]. 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 Status Interpreter module to look at the history of the rover status for generating internal events which highlight relevant changes in the rover status. This process is performed in the first three lines of ActiveMonitoring: Status Interpreter 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 $\text{chr}_a$. All the chronicles recognized at time $t$ are collected into the set $RC$, which becomes the input for the ActiveController as well as the current execution modality. This last module has the responsibility of updating the current execution modality by selecting a new one from the set of possibilities $mods_a$, the new modality will then be emitted towards the FL.

The Status Interpreter generates the internal events by exploiting a set of interpretative rules in rules($atype(a)$). These interpretative rules have the form Boolean condition $\rightarrow$ internal event. The Boolean condition is build upon three basic types of atoms: status variables $x_i$, status variable derivates $\delta(x_i)$, and abstraction operators $qAbs(x_i, [t_i, t_a]) \rightarrow qVals$ which map the array of values assumed by $x_i$ over the time interval $[t_i, t_a]$ into a set of qualitative values $qVals = \{qval_1, \ldots, qval_m\}$. For example, the following interpretative rules:

$(\delta(roll) > limitSroll) \lor (\delta(pitch) > limitSpitch) \rightarrow$ severe-hazard($roll, pitch$)

is used to generate a severe-hazard event whenever the
chronicle plain-terrain {
  occurs((N, +∞), 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)
  }
}

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

Figure 8. Experimental results.

derivate value of either roll or pitch exceeds predefined thresholds in the current rover status.

Another example is the rule:
\[
\text{attitude}(\text{roll}, [t_{\text{current}} - \Delta, t_{\text{current}}]) = \text{nominal} \land \\
\text{attitude}(\text{pitch}, [t_{\text{current}} - \Delta, t_{\text{current}}]) = \text{nominal} \\
\rightarrow \text{safe}(\text{roll}, \text{pitch})
\]

Where \text{attitude} is an operator which abstracts the last \(\Delta\) 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 \text{attitude} and \text{severe-hazard} refer to the rover’s mobility; \text{low-power} instead refers to the rover’s battery.

We assume that at each time \(t\), \text{events}_t can maintain at most one event referring to a specific device.

The \textbf{Temporal Reasoner} is essentially a Chronicle Recognition System (CRS) similar to the one proposed by Dusson in [DLM07]. For simplicity, in our approach we assume that each event \(e_t\) can be consumed by exactly one active chronicle \(\text{chr}\); the function get-relevant-chronicle in Figure 6 selects such a chronicle from \(\text{chronicles}_a\) 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 identify a potentially hazardous terrain. This chronicle is recognized when at least \(N\) \text{severe-hazard} events (regarding the parameters \text{pitch} and \text{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 \text{hazard}, resulting from an interpretation process over the rover’s status variables, denotes that, although the rover’s status is still nominal, it may become anomalous in the near future.

The \textbf{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. Second, the AC update 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. It is to see that only the AC is able to estimate such an extra time as it controls the rover’s speed. So far, however, some of problems related with this activity are still open.

\textbf{Running example} Let us consider the action \text{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. 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 Supervisor 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.

Implementation. To implement the Supervisor’s strategy we have used PLEXIL and its environment (the Universal Executive). PLEXIL is a lightweight but powerful language developed by the NASA [VEJ+]05, which provides a useful event driven framework, supporting concurrent tasks. While PLEXIL has been exploited for the realization of the control strategies depicted in figures 5 and 6 (and hence for the coordination of the different internal modules of the Supervisor), each internal module has been developed as a single piece of software in Java or in C++. The experiments have run on a laptop Intel Core2 2.16 GHz, 2 GB RAM.

6. DISCUSSION AND CONCLUSIONS

The paper has addressed 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 [ACF+98, CIN+08, Nes07] 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 [Mic09, BCRT09, BKS08]). These approaches, however, have been designed to intervene only after a failure has occurred (and therefore when the plan execution has been interrupted). In principle, this problem could be mitigated by anticipating which actions

---

1. PLEXIL has been designed for very low performance systems, and it has been successfully tested on a VxWorks OS (see [VJP06]).
2. PLEXIL is downloadable at http://plexil.wiki.sourceforge.net
in the plan will not be executable in the future (see for example the set of threatened actions in [MT08]), so that the repair strategy can be invoked earlier. Unfortunately, this is not always possible as the plan executor has in general just a partial knowledge of the world where it is operating.

In this paper we have proposed 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 [DLM07] 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 the raw data coming from the environment, and hence reasoning about the environment in more abstract terms. In some scenarios, this problem is still open as the environment is just partially observable and the amount of ambiguous information to deal with may be significantly huge (see [BKZ10]).

In order to keep potential threats under control, we have proposed 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 [CIN+08] 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


