2014 IEEE 26th International Conference on Tools with Artificial Intelligence

Robust Plan Execution in Multi-Agent Environments

Cesar Guzman, Pablo Castejon, Eva Onaindia
Dept. de Sistemas Informáticos y Computación
Universitat Politècnica de València
Valencia, Spain
{*cguzman,pcastejon,onaindia*} *at dsic.upv.es*

Jeremy Frank
NASA Ames Research Center
Moffet Field, USA
{*Jeremy.D.Frank*} *at nasa.gov*

***Abstract*—This paper presents a novel multi-agent reactive execution model that keeps track of the execution of an agent to recover from incoming failures. It is a domain-independent execution model, which can be exploited in any planning control application, embedded into a more general multi-agent planning framework. The multi-agent reactive execution model provides a mechanism allowing an agent to respond to failures that prevent completion of a task when another agent is not able to repair the failure by itself. The model exploits the reactive planning capabilities of agents to come up with a solution at runtime, thus preventing agents from having to resort to replanning. We show the application of the proposed model for the control of multiple autonomous space vehicles.**

***Keywords*-multi-agent, reactive planner; dynamic execution; monitoring plan execution; reactive execution agent; coordination; unpredictable environment**

I. Introduction

Automated planning is very helpful for an effective control in sophisticated applications like manufacturing industry, power and telecommunication networks, robotics or space exploration [16, 18, 7, 5]. A controlled and controllable planning and execution (P&E) system for such real-world applications requires integrating automated planning with a plan execution engine to account for possible conflicts between planning decisions and unexpected events, including failures, arising in the environment.

Most P&E applications rely on single-agent architectures [10, 1, 4] that include the functionalities required to support continuous modification and updating of a current working plan. Systems like IxTeT-eXeC [13] or TPOPEXEC [17] work under a continual planning approach [14, 3, 11], continuously interleaving planning and execution. Some P&E systems unify deliberative (planning) and reactive (execution) behaviors under a single planning technology and model representation [2] while others maintain independent modules for planning and execution together [12].

Multi-agent planning (MAP) systems are viewed as extensions of P&E single-agent architectures for distributed problem solving [22, 21, 15]. One common characteristic of MAP architectures is that the multi-agent infrastructure is specifically used for supporting the deliberative machinery (planning) whereas the need for reactive mechanisms (plan execution) is basically relegated to the individual agent level. Thus, when an executor agent encounters a failure during plan execution it either chooses the immediate next feasible action on the basis of the current context, it accommodates some sort of reactivity by having task-specific assessors that abstractly plan how to accomplish the failed task [15], or it exhibits a reactive behavior and responds to a plan failure by consulting a plan library of predefined, static plans [20]. However, some planning control applications need to rely on a robust plan execution capable of a reactive response that repairs the non-executable part of the plan and allows the flow of the execution to engage with the rest of the (executable) plan. Under these premises, reactive planning as the utilization of a library of reactive plans that accounts for all possible contingencies in the world is unaffordable; and abstract repair plans are not sufficient nor executable. Other approaches resort to reactive model-based programming languages to combine the flexibility of reactive execution languages and the reasoning power of deliberative planners, and thus automate the process of reasoning about low-level system interactions [23, 6]. However, these approaches require the programmer specifically design the plans and contingencies that will ensure a high degree of success in each particular application.

Our aim is to present a general-purpose model for reactive and robust multi-agent plan execution; that is, a model whose underlying principle is to operate over bounded-size structures which have been generated within a limited time. This way, we are able to provide runtime deliberative responses which can be found in the bounded-size structures. The primary issue for a robust plan execution in a MAP application is to make executive agents capable of repairing their own plan failures. But, if this is not possible, a cooperative behaviour that fosters a joint repair among agents is desirable. Unlike most MAP architectures, we propose a novel architecture and a general reactive execution model in which execution agents request for help to others when they are not able to repair their plans by themselves. Our proposal grants a high degree of reactivity and robustness as it always attempts first to repair failures at runtime, either individually or collectively, before resorting to replanning, i.e., asking the deliberative planner for a new executable

1082-3409/14 $31.00 © 2014 IEEE
DOI 10.1109/ICTAI.2014.65

plan.

This paper is organized as follows. The next section outlines the global P&E architecture where the reactive execution model is integrated. Next, we present the single-agent repairing process and the multi-agent approach. Finally, we show a real-world application and last section concludes.

II. PLANNING AND EXECUTION ARCHITECTURE

PLANINTERACTION[1] is a multi-agent P&E architecture where agents autonomously execute a set of tasks in a simulated or real world, monitor the plan execution attending to potential discrepancies, and take decisions for repairing or replanning in case of a plan failure. In this section, we first provide a general description of PLANINTERACTION and then we describe the internal architecture of an execution agent.

PLANINTERACTION builds upon PELEA [12], a single-agent architecture in which an agent is endowed with capabilities for generating a plan, execution, plan monitoring and learning. An agent in PLANINTERACTION is created as a combination of such capabilities, thus providing a great flexibility to set up any multi-agent configuration. In this work, we define two types of agents: *planning agents*, with capabilities for calculating a plan, and *execution agents*, in charge of the plan execution and plan monitoring. Moreover, each planning agent is associated to a different execution agent, both representing a single autonomous entity of the problem (Fig. 1). This particular configuration adapts to contexts in which the autonomous entities have their own particular planning task to solve and they execute in a common environment[2].

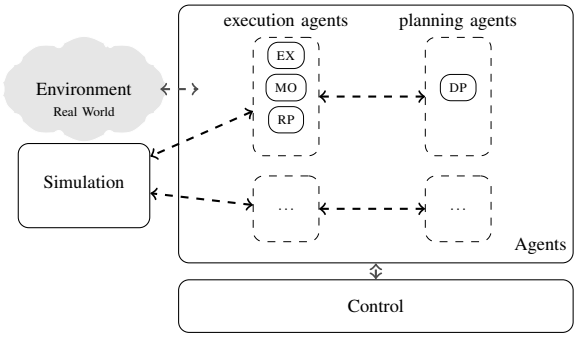

Figure 1. Multi-Agent PLANINTERACTION Architecture

A planning agent comprises a deliberative planner (**DP**) that computes a plan for the planning task of the autonomous entity. The associated execution agent is composed of three modules; the *Execution module* (**EX**), responsible of reading and communicating the current state to the rest of modules as well as executing the actions of the plan calculated by the DP; the *Monitoring module* (**MO**), which receives the state resulting from the execution of some action of the plan from the EX and checks whether the next action of the plan is executable in the resulting state (*plan monitoring*); and the *Reactive Planner module* (**RP**), which is used when a plan failure is detected by the MO in order to promptly find a plan that brings the current state to one from which the plan execution can be resumed.

The EX, MO, and RP modules operate the reactive execution module of an executor agent within PLANINTERACTION. This can be seen in more detail in Fig. 2. An action plan Π calculated by the DP of the planning agent is sent to the MO for its execution. Π consists of a series of actions to be executed at given time steps, each of which makes a deterministic change to the current world state. The elapsed time from one time step to the next defines an execution cycle. The reactive model of an execution agent follows several execution cycles, performing the scheduled action in each cycle until the plan execution is completed. Fig. 3 depicts a plan Π that contains five actions $(a_1, \ldots, a_5)$ scheduled in five execution cycles.

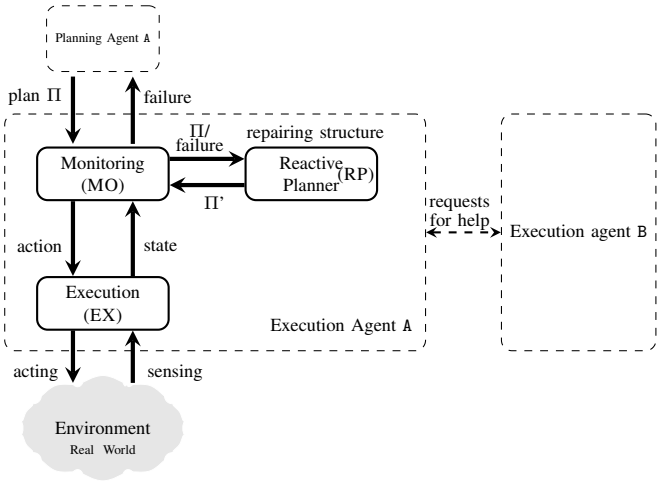

Figure 2. Flow of the reactive execution model

The MO sends the plan Π to the RP, which in turn creates a *repairing structure* for a fragment of Π of length l called *plan window*. l is the number of actions or execution cycles in the plan window, and this value depends on the time the RP is given to build the repairing structure, thus resulting in a bounded-size data structure. For instance, for the plan shown in Fig. 3, the RP might create a repairing structure for a plan window whose length varies from two to five actions depending on the available time. The plan window determines the number of repairable actions and the associated repairing structure encodes alternative plans to repair any of the actions included in the plan window.

Once the time of the RP expires, and a repairing structure

[1] *http://servergrps.dsic.upv.es/planinteraction/*

[2] Depending on the type of problem, other possible configurations are to define a single planning agent with multiple executors or multiple planning agents with a single executor.

has been calculated for a particular plan window, the MO monitors the variables of the first action of the window (e.g., the variables of the action a_1 in Fig. 3). If the sensed values of the action variables match the required values for the action to be executed, the MO sends the scheduled action to the EX for its execution (see the flow in the executing agent of Fig. 2). Then, the EX returns the state resulting from executing the action and the MO monitors the variables of the next action of the window in the new current state. If a mismatch is detected during the plan monitoring, the MO calls the RP, which will make use of the repairing structure created for the plan window to fix the failing action. The RP obtains a new plan Π' that solves the failure and replaces the old plan Π, attaining likewise the goals of the planning task.

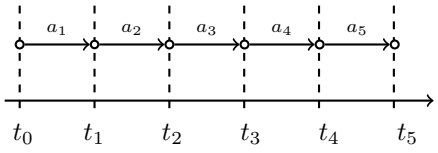

Figure 3. Example of a plan Π composed of 5 execution cycles

An important feature of the reactive model is that while the EX is executing the actions of the plan window, the RP is already calculating the next plan window and associated repairing structure. Typically, the time for the RP to compute the repairing structure of the subsequent plan window is the time that the EX will take to execute the actions included in the current window. Therefore, the more actions in the current plan window, the more time to create the next repairing structure and, consequently, the more actions will likely be included in the subsequent plan window. This working scheme gives our model an *anytime* behaviour, thus guaranteeing the RP can be interrupted at anytime and will always have a repairing structure available to attend an immediate plan failure.

The above description outlines the self-repair flow of an execution agent. Since a plan window will typically cover a small fragment of the plan, and the associated repairing structure only comprises plans that enable to link the current state to any state reachable within the plan window, it may be the case the agent does not find a repair plan for a failing action. In such a case, the execution agent may opt for a reactive repair with the help of the other agents (this is depicted in Fig. 2 through the link "requests for help") before resorting to replanning, i.e., calling its planning agent for a new executable plan. Replanning is not only a costly operation; if replanning takes too long, it may require halting the agent's execution. Consequently, disposing of a robust execution model is always desirable. Next section describes the single-agent repair process and the subsequent section the multi-agent approach.

III. Single-agent repairing process

Let's take the plan shown in Fig. 3 as the plan Π that an execution agent receives from its planning agent. An action $a \in \Pi$ is defined as a set of preconditions, $pre(a)$, and a set of effects, $eff(a)$. Preconditions and effects are represented by means of fluents[3]. Given a world state S, action a is executable in S if $pre(a) \in S$; in such a case, the execution of a results in a new state S' which contains the fluents in $eff(a)$ plus all those fluents in S that have not been altered by the execution of a.

When the RP receives Π from the MO, it starts calculating the plan window and the associated repairing structure ($\mathcal{T}$ for short) that fit within the given time. Generally speaking, $\mathcal{T}$ is a partial-state search tree that encodes recovery plans for a plan window of Π. A *partial state* denotes the minimal set of fluents that must hold in a world state S for the plan to be executable in S. For example, let's suppose the agent has already executed action a_1 and receives S as the resulting state from this execution. If $pre(a_2) \in S$, we can guarantee that a_2 is executable next. But, if we want to ensure the rest of the plan $[a_2, a_3, a_4, a_5]$ is executable in S, we must check that S includes the proper set of fluents for this to happen. This set of fluents is called a *partial state*.

Fig. 4 graphically shows the $\mathcal{T}$ for a plan window of length 3. Nodes G_i represent *regressed partial states* because they are calculated by regressing the goals of the problem through the actions of the plan. More specifically, let a be an action and G a partial (goal) state such that $eff(a) \subseteq G$. The *partial state* G' in which a can be applied is calculated by the *regressed transition function* $\Gamma(G, a)$, defined as:

$$G' := \Gamma(G, a) := G \setminus eff(a) \cup pre(a) \quad (1)$$

where G' represents the minimal set of fluents that must hold in a world state in order to generate G by means of the execution of a. The regressed partial state approach was first used by PLANEX [8] to supervise the execution of a sequence of actions. Roughly, the regression of a fluent over an action is a sufficient and necessary condition for the satisfaction of the fluent following the execution of the action. The work in [9] formalizes this concept in the situation calculus language.

The root node of $\mathcal{T}$ is the last partial state of the plan window. The tree nodes are partial states and arcs are labeled with the actions defined in the planning task. The expansion of $\mathcal{T}$ consists in finding relevant actions for each fluent of a partial state by recursively applying the regressed transition function. That is, $\mathcal{T}$ encodes recovery plans to reach a partial state from another partial state.

$\mathcal{T}$ is the bounded-size structure that the execution agent will use to repair a failure in an action of the plan window. Let's suppose that the current world state is S and that a_1

[3]A fluent is a tuple $\langle v, p \rangle$ where v is a variable and p represents the value the variable takes on

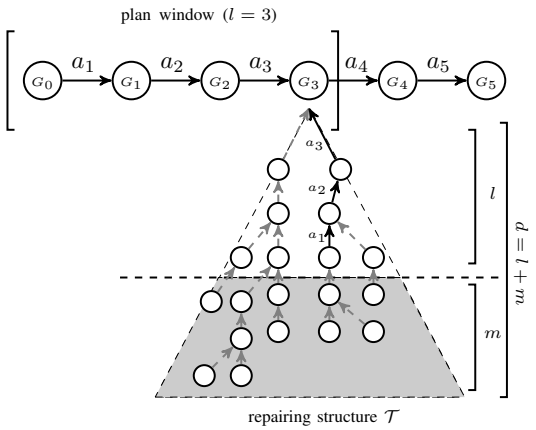

Figure 4. Repairing structure for a plan window of length 3

cannot be executed in S. Fixing a failure means the RP applies a path-finding algorithm between two nodes of $\mathcal{T}$: G_o, a origin state, and, G_t, a target state. G_o is a set of relevant fluents that hold in S ($G_o \subseteq S$); and G_t is a partial state of the plan window. The first attempt to repair the failure in a_1 is to reach $G_t = G_0$ from a node which represents a situation that holds in S. The RP performs a backward search from G_0 until it finds a state G_o in $\mathcal{T}$ such that $G_o \subseteq S$. If such a path does not exist then the second attempt is to set $G_t = G_1$ and the same process is repeated. Note that a_1 will no longer be involved in the second attempt since a recovery plan to G_1 would continue the plan execution with action a_2. If a path is not found either, next attempt will be $G_t = G_2$, and so on until $G_t = G_3$. Hence, the purpose of a plan repair is to find a plan to a partial state of the plan window, thus guaranteeing the rest of the plan is executable and that the calculated repairing structures remain valid.

It is important to highlight that the closer G_t is to the root node of $\mathcal{T}$, the more choices to find a recovery path. This can be graphically observed in Fig. 4, where the depth of $\mathcal{T}$ is $l+m$, being l the length of the plan window. The subtree of depth l comprises the partial states of the plan window. The subsequent m levels contain more partial states; the deeper the tree, the more possibilities to find a state G_o which fluents hold in the current world state. Consequently, if the repairing task is to reach G_0 and, thereby, actions a_1, a_2 and a_3 must be included in the recovery plan, the path-finding algorithm restricts the search to the m levels of the shadowed area in figure 4 (particularly, to the single branch that departs from G_0). In contrast, if the repairing task is to reach G_3 then the search area to find a state that matches S is far larger because no actions of the plan window must be included in the recovery plan.

In conclusion, when the target state is one of the early states of the plan window, we find fewer alternatives of repairing but the recovery path guarantees more stability with respect to the actions of the plan window. However, if the target state is closer to the root node, more choices to find a recovery plan exist although the found plan might not keep any of the actions of the plan window. Clearly, the more flexibility, the less stability.

Finally, a note on the time-bounded construction of $\mathcal{T}$. The RP has a limited time to build $\mathcal{T}$, and the size of $\mathcal{T}$ is determined by the length of the plan window (l) and the depth (d) of the tree. We estimate the time to generate $\mathcal{T}$ with respect to some selected l and d through the application of a predictive model. We trained the predictive model with the solution plans obtained from the execution of diverse planning benchmarks.

IV. Multi-agent execution model

The focus of the multi-agent execution model is to provide agents with a cooperative behavior so that they engage together in a multi-agent repair (MAR) task when an agent is not capable of solving a failure by itself. In the following, we describe the flow of the multi-agent execution model and the MAR approach.

A. Flow of the multi-agent execution model

The flow of the individual reactive execution model presented in Fig. 2 is extended to a multi-agent context as depicted in Fig. 5. The two new modules are: *publicizing* and *multi-agent repairing* (MAR).

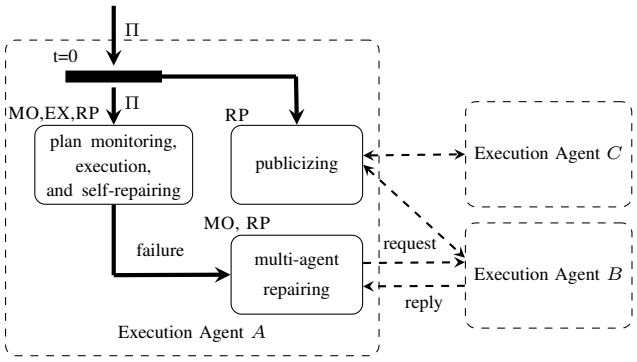

Figure 5. Flow of the multi-agent reactive execution model.

Let's suppose Π_A is the plan of an agent A, and that the first repairing structure calculated by A results in a plan window of $l = 3$ and $\mathcal{T}_{G_3}$, where G_3 is the root node. Simultaneously to the construction of $\mathcal{T}_{G_3}$, agent A publicizes two types of information by a broadcast message:

(1) **partial states**: agent A publicizes the partial states of its plan window $(G_0, \ldots, G_3)$ to the rest of participating agents. The aim of this communication is to make the rest of agents aware of the fluents that A might need to be repaired in case that a self-repair does not fix

a failure in Π_A. The participating agents filter out the fluents they are not able to achieve; for the remaining fluents, they create a single search space, similar to a repairing structure, per fluent. For instance, if A publicizes the fluent $\langle v, d\rangle$ and variable v is not defined in agent B's problem, then B will ignore such a fluent. Likewise if variable v is defined in B's problem but the value d is unknown to B. Otherwise, for each unfiltered fluent $\langle v, d\rangle$, B creates a repairing structure $\mathcal{T}_{\langle v,d\rangle}$, which will be used to search for a solution in case A encounters a failure that affects its fluent $\langle v, d\rangle$.

(2) **services**: agent A also publicizes the variables that appear in the effects of the actions defined in its planning task (services). That is, A publicizes the variables he can modify in order to inform the rest of agents how he could contribute in a MAR task. This way, agents are informed of the capabilities of A in case a failure occurs in their plans. As with the partial states, participating agents filter out the variables unknown to them.

In other words, the key principle of publicizing is: (1) to inform about what an agent may require during its plan execution if a failure occurs (obtained from the partial states of the plan window), and (2) to inform about what an agent can do for the others (extracted from the agent's actions).

Then, if a failure occurs in Π_A, and agent A is not able to fix it through a self-repair, he will invoke a MAR task, which is explained in the next section.

B. Multi-agent repairing task

The multi-agent repairing (MAR) task is depicted in Fig. 6. It consists of three following stages:

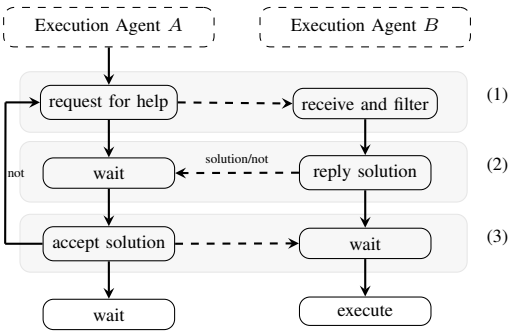

Figure 6. multi-agent reactive process.

(1) **requesting for help:** if a fluent $\langle v, d\rangle$ of the agent A fails, A communicates with the agents that have publicized v among their services. Let's assume that B is one of these agents and that a_1 is the failing action in the execution of Π_A. Then, A sends B the partial state G_0 in order to be able to execute the next action, a_1, of its plan window. Assuming $G_0 = \{\langle v, d\rangle, \langle v', d'\rangle\}$, B filters out the unknown variables and keeps $G_0 = \{\langle v, d\rangle\}$.

(2) **reply solution:** in a multi-agent context, repairing a failure means the agent tries to reach a fluent by affecting its own execution as less as possible. Consequently, B has to consider its own plan window during the MAR task. First, B uses the search space that he built for $\langle v, d\rangle$ during the publicizing phase of agent A and applies a a path-finding algorithm that obtains a recovery plan for this fluent. Second, he merges this recovery plan into his own repairing structure to come up with a final plan that fixes $\langle v, d\rangle$ and carries on with his plan execution. Finally, B informs agent A about the number of execution cycles that are needed to achieve $\langle v, d\rangle$.

(3) **accepting the solution:** A receives the response from B and accepts or rejects the solution. Normally, it will accept the solution that achieves the fluent $\langle v, d\rangle$ in the fewer execution cycles, rejecting the solutions offered by the rest of agents. B receives the acceptance message and proceeds with the execution of the merged plan. If A does not receive any reply solution or rejects all the possible solutions, the stage (1) starts again with the next partial state G_1 associated to the plan window. If there is no solution to any of the partial states in its plan window, A resorts to replanning and calls its deliberative planner.

The next section shows the application of this procedure to a planetary Mars domain.

V. Planetary Mars domain, a real-world application example

In this section, a representative mission of a hypothetical Planetary Mars domain is used to illustrate the operation of the reactive execution model. In this mission, two rovers and a lander are deployed on the Martian surface, and a spacecraft is in Mars orbit. The rovers analyze rocks or soils and communicate the results to a lander, which in turn sends the results to a control center situated in Earth. The spacecraft orbits Mars and generates maps for the rovers. In case of a failure in the plan of a rover, it communicates with the deliberative planner in Earth to have a new executable plan. The current Curiosity mission on Mars [19] consists of a single rover whose plan is generated and uplinked daily. Curiosity has the ability to avoid hazards while traversing, but performs few other replanning functions. Existing orbital imagery can be used to determine areas of potentially interesting science, but is too low resolution to provide precise science targets, or traversal maps for rovers. Unexpected events and failures are always handled by replanning on the ground. Finally, no Mars missions with cooperative rovers have been deployed. While futuristic, the motivation behind this hypothetical problem is to show how rovers and spacecraft with on-board reactive repairing capabilities can reduce the communication overhead with the Earth and still accomplish their mission. We show here the behaviour of

the RP for repairing a failure with a self-repair process or through a MAP task.

We have two rovers, A and B, and a spacecraft, C. The rovers *analyze* rocks (r), or soils (s), and *communicate* the results to a Lander (L), which in turn sends the results to the Earth. The task of C is to *fly* all over the terrain to create good map routes and *communicate* them to the rovers. The waypoint w_2 is the initial location of L and of both rovers, A and B. The initial location of C is w_3. L remains always in its initial location w_2. Rovers have good maps to travel between two waypoints. The mission of A is to use the microscopic camera to analyze rocks located in w_3, communicate the results to L, and *navigate* to the initial position w_2. The mission of B is to analyze a sample of soil located in w_1, communicate the results to L, and navigate to w_3. The mission of C is to create good maps to travel from w_3 to w_1, and from w_1 to w_2. Rovers A and B can only communicate to L if the device transmission is not disabled, i.e. either the fluents $\langle$trans-A,$true\rangle$ and $\langle$trans-B,$true\rangle$ are present, respectively, in the current world state. The deliberative planner on Earth computes the plans Π_A= $\langle a_1, a_2, a_3, a_4\rangle$, Π_B= $\langle b_1, b_2, b_3, b_4\rangle$, Π_C= $\langle c_1, c_2\rangle$ for A, B, and C, as shown in Fig. 8.

Fig. 8 depicts the execution control flow of our application example. For each agent (A, B, and C), we show:

- Π_X: it is the plan of agent X along with the partial states G_i that must hold at each time step of the plan.
- $\mathcal{T}_{G_j}$: it is the current repairing structure used by agent X in a self-repair task.
- $\mathcal{T}_{\langle v,p\rangle}$: individual repairing structures that agent X creates per each fluent received from the other agents during the publicizing phase.
- P: this represents the publicizing state. For example, the services publicized by agent C are the variable maps-w_y-w_z, affected by the action fly C w_y w_z (fly the spacecraft from waypoint w_y to w_z), and the variable link-X-w_y-w_z, affected by the action send-maps C X w_y w_z, by which the spacecraft C sends agent X navigation maps to travel from w_y to w_z (see Fig. 7).
- R/F: the agent receives and filters the information received from another agent.
- SR: the agent executes a self-repair process.
- MR: the agent repairs the failure of another agent through the application of MAR task.

Firstly, before agents execute their plans Π_A, Π_B, and Π_C, they publicize the partial states associated to the first repairing structure and the services they provide (variables they can modify). For example, the first repairing structure of the rover A is $\mathcal{T}_{G_3}$, and A communicates the partial states G_0, G_1, G_2, and G_3 along with the variables comm-rock and comm-soil, which represent that the results of analyzing the rocks or the soils are respectively communicated to L. Agents B and C filter out the fluents they cannot achieve

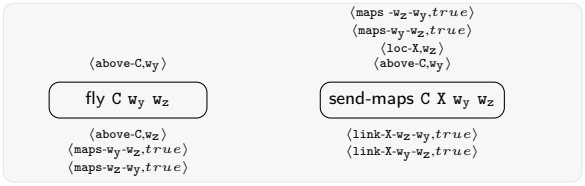

Figure 7. Actions of the spacecraft C. Variables are above-C: location of the spacecraft C; loc-X: location of rover X; link-X-w_y-w_z: map of the agent X to travel from w_y to w_z; and maps-w_y-w_z: general maps to travel from w_y to w_z.

with the actions defined in their planning tasks. For example, B will ignore the fluent $\langle$link-A-w_3-w_2,$true\rangle$ in the partial states publicized by agent A because there is nothing that B can do about it. And the spacecraft C will filter out the fluent $\langle$comm-rock,$true\rangle$ because C can only help with the generation of travel maps.

In this example, four possible situations may arise and are described as follows:

- *correct execution:* No failures are encountered during the execution of Π_A, Π_B, and Π_C.
- *self-repair:* Let's assume the following situation. A wants to execute the first action a_1, Navigate A w_2 w_3, which changes the fluent $\langle$loc-A, $w_2\rangle$ to $\langle$loc-A, $w_3\rangle$, and when the MO verifies the preconditions of a_1, it detects a failure due to the path from w_2 to w_3 is blocked; i.e., the fluent $\langle$link-A-w_2-w_3, $false\rangle$ is present in the current world state S. Hence, at this point, we have S = {$\langle$loc-A,$w_2\rangle$, $\langle$loc-L,$w_2\rangle$, $\langle$link-A-w_2-w_3,$false\rangle$, $\langle$trans-A,$true\rangle$, $\langle$loc-rock-w_3,$true\rangle$, $\langle$loc-soil-w_1,$true\rangle$, $\langle$link-A-w_1-w_2,$true\rangle$, $\langle$link-A-w_2-w_1,$true\rangle$, $\langle$link-A-w_1-w_3,$true\rangle$, $\langle$link-A-w_3-w_1,$true\rangle$} [4]. The MO calls the RP and communicates it the fluents in S as well as the time point of the faulty action a_1. The RP repairs the plan failure to the state G_1 of $\mathcal{T}_{G_3}$ by adding the actions a_1', Navigate A w_2 w_1, and a_1'', Navigate A w_1 w_3. Thereby, the rover recovers from the plan failure by applying a self-repair process that finds an alternative path to w_3.
- *requesting for help:* Let's suppose now that rover A wants to execute the repaired plan Π_A={$a_1', a_1'', a_2, a_3, a_4$}, particularly the first action a_1', which changes the fluent $\langle$loc-A, $w_2\rangle$ to $\langle$loc-A, $w_1\rangle$, and let's suppose the MO detects a failure because the fluent $\langle$link-A-w_2-w_1, $false\rangle$ is present in the current world state S; i.e. the link from w_2 to w_1 is no longer available.

  In this situation, A is in w_2 and cannot navigate to any other waypoint (w_1 or w_3). Thus, the RP is not able to reach any target state within its repairing structure $\mathcal{T}_{G_3}$.

[4]Variables are loc-A: location of rover A;loc-L: location of the lander L;link-A-w_i-w_j: map of the agent A to travel from w_i to w_j; and trans-A: device transmission of the agent A is enabled or disabled.

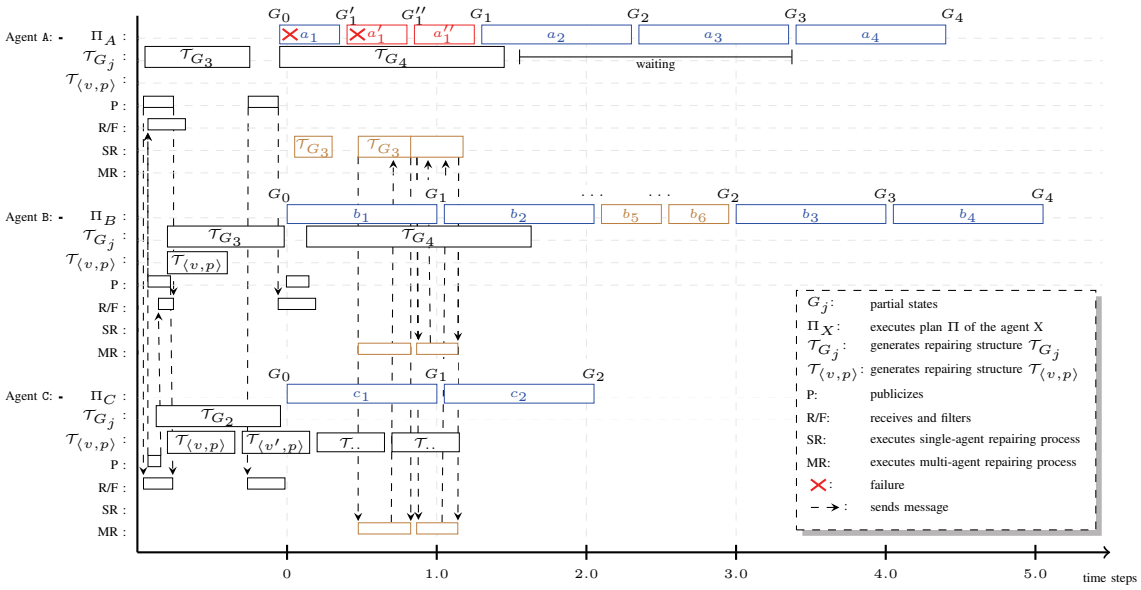

Figure 8. Flow of the multi-agent reactive execution model in a real-world application example

In this case, A requests for help to the agent C, which has publicized the services link-A-w_2-w_1 and link-A-w_2-w_3. A sends the first partial state of its plan window (state G'_1) that needs to be reached in order to be able to execute the action a'_1. C receives the information and starts the MAR task (see row MR of the agent C in Fig. 8) using its repairing structure $\mathcal{T}_{G_2}$ and the tree $\mathcal{T}_{\langle v',p\rangle}$, where $\langle v', p\rangle$ represents the fluent $\langle$link-A-w_2-w_1,$true\rangle$.

- *new requesting for help:* in the case that agent C is not able to create a map that allows A to navigate from w_2 to w_1, rover A gives up finding a solution for executing the action Navigate A w_2 w_1, and it decides to repeat the MAR task with its next partial state (state G''_1). In this state, having A in w_2 is no longer necessary but other fluents must be achieved to execute a''_1. If agents cannot either find a feasible repair that allows A to execute a''_1, A may opt for repeating the same process with the following partial state or call its deliberative planner instead. As for a MAR task, the last resort of A is to request whether someone can directly achieve $\langle$comm-rock,$true\rangle$, a fluent that is present in its last partial state. And it happens that B finds a recovery plan that achieves $\langle$comm-rock,$true\rangle$ after executing its own plan because B has also capabilities to analyze rocks and to communicate the results to the lander.

In summary, the information comprised in the repairing structures allows the RP to easily find a solution plan to a failure because the search trees encompass the contingencies (failures) most likely to happen in the real world. The multi-agent approach allows agents to exploit several alternatives to repair a plan at runtime before replanning.

VI. Conclusions

This paper presents the extension of a single-reactive execution model to a multi-agent context. More specifically, we designed a recovery execution model for robust plan execution, which implements a multi-agent repairing task for solving plan failures among multiple agents. The model allows dealing with the heterogeneity of the execution agents to be executed, and is embedded into PlanInteraction, a more general architecture for multi-agent planning and execution.

An execution agent operates over bounded-size structures. A bounded-size structure is a search tree that accommodates a representation of the potential failed world states that might occur during the plan execution. By using this repairing structure, an execution agent can return deliberative responses at runtime. Furthermore, agents are endowed with capabilities to repair their own plan failures, and otherwise request for collaboration to perform a cooperative multi-agent repair task.

The paper shows the application of the multi-agent execution model in a planetary Mars domain. This is the scenario that has primarily motivated this work although the model is designed to be a domain-independent recovery mechanism. We can conclude that the proposed repairing structures constitute a flexible mechanism to find recovery plans in contrast to the rigidity of the responses of most reactive systems. All in all, the multi-agent reactive execution model is viewed as a perfect trade-off between deliberation and reactivity.

ACKNOWLEDGMENT

This work has been partially supported by the Spanish MICIIN project TIN2011-27652-C03.

REFERENCES

[1] M. Ai-Chang, J. Bresina, L. Charest, A. Chase, J.-J. Hsu, A. Jonsson, B. Kanefsky, P. Morris, K. Rajan, J. Yglesias, B. Chafin, W. Dias, and P. Maldague, MAPGEN: Mixed-initiative planning and scheduling for the Mars Exploration Rover mission, *IEEE Intelligent Systems*, vol. 19, no. 1, Feb. 2004, pp. 8–12.

[2] 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.

[3] M. Brenner and B. Nebel, Continual planning and acting in dynamic multiagent environments, *Autonomous Agents and Multi-Agent Systems*, vol. 19, no. 3, 2009, pp. 297–331.

[4] A. Cesta, G. Cortellessa, S. Fratini, and A. Oddi, Developing an End-to-End Planning Application from a Timeline Representation Framework, in *IAAI-09. Proceedings of the 21st Innovative Applications of Artificial Intelligence Conference, Pasadena, CA, USA*, 2009.

[5] S. Chien, B. Cichy, A. Davies, D. Tran, G. Rabideau, R. Castaño, R. Sherwood, D. Mandl, S. Frye, S. Shulman, J. Jones, and S. Grosvenor, An autonomous earth-observing sensorweb, *IEEE Intelligent Systems*, vol. 20, 2005, pp. 16–24.

[6] R. T. Effinger and B. C. Williams, Dynamic controllability of temporally-flexible reactive programs, in *ICAPS*. AAAI, 2009, pp. 122–129.

[7] M. G. Félix Ingrand, Robotics and artificial intelligence: A perspective on deliberation functions, *AI Communications*, vol. 27, no. 1, 2014, pp. 63–80.

[8] R. E. Fikes, P. E. Hart, and N. J. Nilsson, Learning and executing generalized robot plans, *Artificial Intelligence*, vol. 3, 1972, pp. 251–288.

[9] C. Fritz and S. A. McIlraith, Monitoring plan optimality during execution, in *ICAPS*, 2007, pp. 144–151.

[10] M. P. Georgeff and A. L. Lansky, Reactive reasoning and planning, in *Proceedings of AAAI-87 Sixth National Conference on Artificial Intelligence*, Seattle, WA (USA), Jul. 1987, pp. 677–68.

[11] M. Ghallab, D. S. Nau, and P. Traverso, The actor's view of automated planning and acting: A position paper, *Artif. Intell.*, vol. 208, 2014, pp. 1–17.

[12] C. Guzmán, V. Alcázar, D. Prior, E. Onaindia, D. Borrajo, J. Fdez-Olivares, and E. Quintero, Pelea: a domain-independent architecture for planning, execution and learning, in *ICAPS Workshop on Scheduling and Planning Applications woRKshop (SPARK)*, 2012, pp. 38–45.

[13] S. Lemai and F. Ingrand, Interleaving temporal planning and execution: Ixtet-exec, in *Proceedings of the ICAPS Workshop on Plan Execution*, 2003.

[14] ——, Interleaving temporal planning and execution in robotics domains, in *AAAI Conference on Artificial Intelligence*. AAAI, 2004, pp. 617–622.

[15] V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. N. Prasad, A. Raja, R. Vincent, P. Xuan, and X. Q. Zhang, Evolution of the GPGP/TAEMS domain-independent coordination framework, *Autonomous Agents and Multi-Agent Systems*, vol. 9, no. 1-2, 2004, pp. 87–143.

[16] M. G. Marchetta and R. Forradellas, An artificial intelligence planning approach to manufacturing feature recognition, *Computer-Aided Design*, vol. 42, no. 3, 2010, pp. 248–256.

[17] C. J. Muise, J. C. Beck, and S. A. McIlraith, Flexible execution of partial order plans with temporal constraints, in *IJCAI*, 2013.

[18] C. Piacentini, V. Alimisis, M. Fox, and D. Long, Combining a temporal planner with an external solver for the power balancing problem in an electricity network, in *ICAPS*, 2013.

[19] J. A. Samuels, The mars curiosity rover mission: remotely operating a science laboratory on the surface of another planet. in *17th Annual International Space University Symposium, Strasbourg, France, March 5, 2013.* Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2013., 2013.

[20] L. P. Sebastian Sardiña, A BDI agent programming language with failure handling, declarative goals, and planning, *Autonomous Agents and Multi-Agent Systems*, vol. 23, no. 1, 2011, pp. 18–70.

[21] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa, The RETSINA MAS infrastructure, The special joint issue of Autonomous Agents and MAS, Volume 7, Nos. 1 and 2, Tech. Rep., 2001.

[22] D. Wilkins and K. Myers, A multiagent planning architecture, in *AIPS*, 1998, pp. 154–162.

[23] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, Model-based programming of intelligent embedded systems and robotic space explorers, *Proceedings of the IEEE: Modeling and Design of Embedded Software*, vol. 91, no. 1, 2003, pp. 212–237.