Multi-agent reactive planning for solving plan failures

Cesar Guzman Alvarez[1], Pablo Castejon[1], Eva Onaindia[1], and Jeremy Frank[2]

[1] Universitat Politècnica de València, Camino de Vera s/n, Valencia, Spain
[2] NASA Ames Research Center, Moffet Field, USA

Abstract. In this paper we present a multi-agent reactive planning mechanism for recovering from plan failures with the help of multiple agents. Our contribution is twofold: a proposal of a dynamic execution architecture embedded into a more general multi-agent planning framework, and a mechanism based on state-transition systems that allows execution agents to reactively and cooperatively attend a plan failure during execution. Specifically, we propose a flexible dynamic execution architecture that allows agents to find solutions for a successful plan execution during a plan failure.

Keywords: reactive planner, multi-agent planner, coordination, execution

1 Introduction

Most planning-and-execution applications rely on single-agent architectures that include the functionalities required for a continuous planning, namely sensing the state, generating the problem at hand, planning, executing the plan, monitoring the execution for failures, and replanning; for example, space and robotics applications of platforms as Mapgen [1], APSI [5], PRS [8], or IxTeT [9]. Typically, these architectures incorporate a deliberative component augmented with reactive behaviors [14], unify deliberation and execution under a single planning technology and model representation [2] or maintain independent modules for planning and execution in an integrated way [10].

Multi-agent planning (MAP) systems are viewed as extensions of planning-and-execution single-agent architectures for cooperative distributed problem solving [16, 11, 12]. One common characteristic of these 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 resorts to a centralized manager that sends messages to the planning agents requesting a solution [16]; it accommodates some sort of reactivity by having task assessors that only abstractly plan how to accomplish the failed task [12]; or the executor agent is a Beliefs, Desires and Intentions (BDI) agent that exhibits a reactive behavior and responds to a plan failure by consulting a plan library of predefined, static plans [13]. In any case, almost all of the MAP architectures rely on cooperative behaviors for the construction of plans but execution failures are individually attended by each agent.

Reactive planning architectures have been largely investigated in the area of automated planning. The first approaches to reactive planning exploited abstraction as a means to implement quick response mechanisms, like the Procedural Reasoning System (PRS) [8], a framework for symbolic reactive control systems in dynamic environments, or the Reactive Action Package (RAP) [7], a system designed for the reactive execution of symbolic plans. The usage of hierarchical control structures [3] or semi-reactive architectures [9], which provide rough plans when a quick response is required in the presence of unforeseen events, are also very popular in reactive planning. However, none of the reactive frameworks embedded in MAP systems have ever exploited the idea of cooperatively solving a plan failure at execution time by using the reactive capabilities of the agents in the system. The problem here lies in the difficulty of merging the reactive plan representation of multiple agents and reactively responding to an agent's request.

In this paper, we present a first approach to reactively and cooperatively solve a plan failure in a MAP system. Our work builds upon PELEA [10], a component-based single-agent architecture able to perform planning, execution, monitoring and repairing in an integrated way, and PLANINTERACTION[3], a multi-agent planning architecture that integrates PELEA agents into a multi-agent system. Within the PLANINTERACTION platform, we propose a dynamic execution architecture and a recovery mechanism based on state-transition systems that allow executor agents to quickly respond to unexpected events before resorting to a computationally expensive replanning solution.

This paper is organized as follows. Next section provides the main features of PELEA, PLANINTERACTION, and presents the Dynamic Execution Architecture embedded in PLANINTERACTION. Section 3 introduces the state-transition system used by the reactive planner and section 4 presents an example of application. Finally, last section concludes and presents our future research lines.

2 PlanInteraction: an architecture for multi-agent plan interaction

PLANINTERACTION[4] is a multi-agent planning-and-execution architecture, which allows agents to autonomously perform science targets, 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.

PLANINTERACTION is built upon PELEA [10], a Planning, Execution and LEarning Achitecture for a single-agent. PELEA provides an agent with onboard capabilities to generate, execute, and monitor a plan. It also provides an agent with learning capabilities. The main components of PELEA that are used in PLANINTERACTION are:

– *Execution module* (**EX**). This is the starting point of the architecture. The *EX* captures as input a planning task, which current state is read from the environment through the sensors. The *EX* module is thus in charge of reading and communicating the current state to the rest of modules as well as executing the tasks in the environment.
– *Monitoring module* (**MO**). Besides the current state, the *EX* also sends the *MO* the planning task to solve. The *MO* calls a deliberative planner in order to obtain a plan

[3] http://servergrps.dsic.upv.es/planinteraction/
[4] This work is supported by project TIN2011-27652-C03-01

(see below) and, once obtained, it sends the actions to execute to the *EX*. The *EX* reports the *MO* the state resulting from executing each action and the *MO* performs the plan monitoring process, i.e. it checks whether the received state matches the expected state or not and determines the existence or lack of a plan failure.

- *Deliberative Planner module* (**DP**). The *DP* receives a planning task and generates a plan for the task. This module is also invoked when it is necessary to fix (repair or replan) a plan. The planner is also responsible for selecting the variables that the module *MO* must check during the plan execution. Any state-of-the-art planner can be used as the *DP* module.

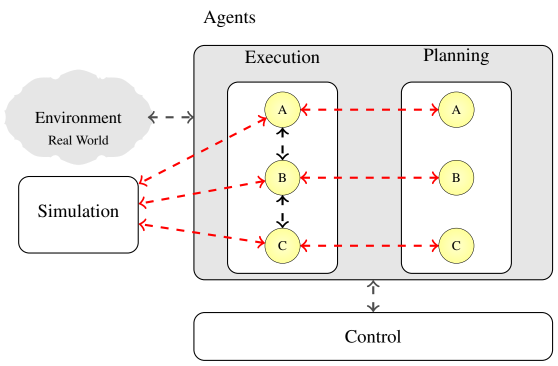


Fig. 1. Multi-Agent Plan Interaction Architecture

PLANINTERACTION is a flexible and domain-independent architecture implemented by integrating PELEA agents in an open MAP platform called MAGENTIX2 [15]. As we can see in Fig. 1, the architecture consists of three main modules:

- *Control* is responsible of registering agents in the system, initializing the internal clock, handling the problem information and controlling conditions for the system termination. The internal clock manages the time of all agents in the system.
- *Simulation* represents the simulated state of the world. Agents will be able to access the information in the simulated environment as well as to modify it through the execution of the actions in their plans.
- *Agents* comprises the set of agents of the problem. An agent in PLANINTERACTION represents any combination of the PELEA modules. The composition of modules in each agent depends on the problem specification and agent's capabilities. Thus, we can have the three modules (*EX*, *MO* and *DP*) embedded in a single PELEA-like agent; we can also opt for creating planning agents that only comprise the *DP* module, thus providing agents with capabilities for planning and repairing plans, and execution agents that comprise the *EX* and *MO* modules, with capabilities for tracing plan execution and plan monitoring (this is the configuration shown in Fig. 1). In some applications, we might even want to have several different EX modules but a single *MO*; for example, a robotics application where one monitor controls the operations of several robots moving in a shared space.

The particular architecture configuration we have chosen for our purposes is shown in Fig. 1. The simulation module is not required in case we are working in a real-world scenario. Each execution agent comprises the modules *EX* and *MO*, and each planning agent contains a *DP* module. Specifically, in the problem shown in section 4 we have three parties, two rovers (A, and B) and one spacecraft (C), each one containing a planning agent and an execution agent.

In this type of configuration, we distinguish three different coordination levels: 1) *Planning-Planning Coordination*, between planning agents when they have to jointly generate a solution plan for a particular task; 2) *Execution-Execution Coordination* focuses on the coordination between execution agents when they attempt to resolve a plan failure at execution time; 3) *Execution-Planning Coordination* takes place when execution agents are not capable of reaching a solution during the execution-execution coordination and so they have to resort to their planning agents so as to find a new plan for solving the task. In this paper, we specifically focus on execution-execution coordination.

2.1 Dynamic Execution Architecture

In this subsection, we present the multi-agent Dynamic Execution Architecture (DEA) we have implemented within the PLANINTERACTION framework. DEA is particularly devoted to implement the execution-execution coordination referred above. Our goal is to come up with DEA in which execution agents gather together at execution time for repairing some failure that happened in the environment before resorting to a more computationally expensive planning-execution coordination.

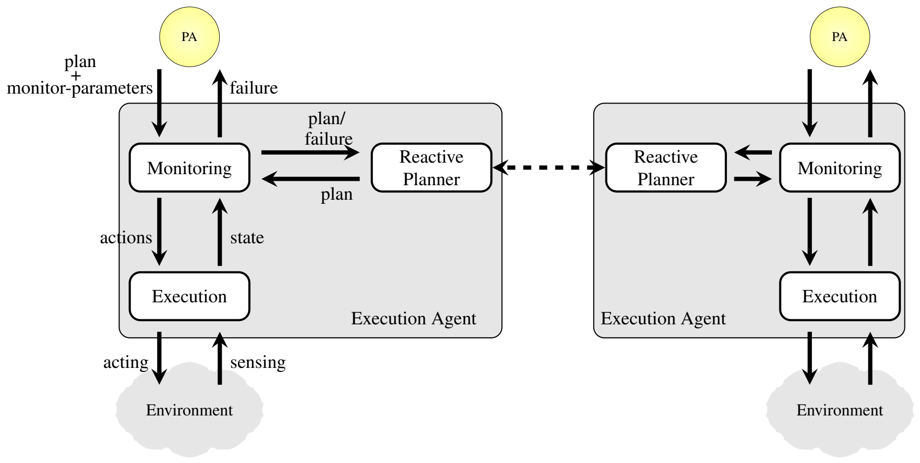


Fig. 2. Multi-Agent Dynamic Execution Architecture.

DEA is composed of three modules (Fig. 2 left): a *MO* module, a Reactive Planner (*RP*), and an *EX* module. The control flow of the architecture (Fig. 2) begins when the planning agent sends the plan and the parameters to be monitored (*monitor-parameters*) to the *MO* module of the execution agent. The *MO* sends the plan to the *RP*, which transforms the plan into a set of reactive state-transition systems (explained in the next section) that will be later used for a reactive plan fixing. Meanwhile, the *MO* sends a set of executable (*actions*) to the *EX*, which executes them in the specified order (*acting*), senses the dynamic part of the state from the environment (*sensing*), and sends it to the *MO*. The *MO* receives the information from the sensors (*state*) and checks the parameter values before sending the next action to execute. If a discrepancy in the value of a variable is found, the anomaly (*failure*) is reported by the *MO* to the *RP*, which uses the stored reactive structure for a rapid intervention. If the *RP* finds a solution, it sends a new plan to the *MO*, which in turn sends the next action to the *EX*. In case the *RP* is not able to fix the problem, it can request other agents for help. This implies the activation of a communication protocol that performs the execution-execution coordination. If the other agents cannot find a solution, then the *RP* informs the *MO*, which reports the plan failure to the planning agent for that it repairs the plan or find a new executable plan.

3 Reactive Planner

We briefy summarize some basic concepts of plannnig that we will need to explain the *RP* module. It is important to note that we work at the same abstraction level both within a planning and an execution agent. While many architectures translate high-level planning specifications into low-level execution structures, we keep the same level of abstraction. This is by no means a loss of generality as it has no impact at all in the design of the reactive planning module.

In classical planning, a planning task of an agent is defined as a tuple $\langle I,A,G\rangle$, where I represents the agent's initial state of the world, G is a partial world state that represents the goals that the agent wants to achieve and A is the set of actions of the agent. In our setting, we assume we have a set of entities or domain agents, each one with its particular planning task to solve. Agents share the initial state I and, possibly, some of the actions in A, depending on the agents' skills; however, each agent has its specific G to accomplish.

A state of the world is modeled through a finite set of state variables V, each associated to a finite domain D_v of mutually exclusive values. A $fluent$ is a tuple $\langle v,d\rangle$, which indicates that the variable $v \in V$ takes the value $d \in D_v$. Therefore, a world state s is defined as a set of fluents.

An agent's action is a transition function $a \in A$ that when applied to world state s gives rise to a new state s'. Specifically, an action a is defined as tuple $a = \langle pre(a), eff(a)\rangle$, where $pre(a)$ is a finite set of fluents that represents the preconditions of a, the fluents that must hold in s in order to apply a in s. And $eff(a)$ is a finite set of operations that change the value of fluents. An operation of the form $(v = d)$ adds a fluent $\langle v,d\rangle$ to state s' and also removes fluents of the form $\langle v,d'\rangle$ such that $d' \neq d$ in state s'. We will denote by $eff(a)^+$ the fluents added to s' and by $eff(a)^-$ the fluents removed in s'.

An agent's plan is defined as $\pi = \langle a_1, a_2, \ldots, a_n\rangle$, a sequence of actions that solves its planning task. This way, action a_1 is applied in state I resulting in a new state, say s_1, then a_2 is applied in s_1 resulting in a new state and so on. Given a world state s and an action a, the result of executing a in s is $result(s, \langle a\rangle) := s \setminus eff(a)^- \cup eff(a)^+$

if the action is applicable in s, i.e., $pre(a) \subseteq s$. Otherwise, $result(s, \langle a \rangle)$ is undefined. The result of executing π in a state is recursively defined by $result(s, \langle a_1, \ldots, a_n \rangle) := result(result(s, \langle a_1, \ldots, a_{n-1} \rangle), a_n)$, and $result(s, \langle \rangle) = s$.

When the *MO* module of an execution agent receives a plan $\pi = \langle a_1, a_2, \ldots, a_n \rangle$, the *MO* sends action a_1 to execute and, simultaneously, sends π to the *RP* module. The reactive planner converts the plan π into a reactive structure that allows it to attend future failures during the plan execution. Specifically, the *RP* builds a state-transition system [4] that represents the plan π and extends this basic representational structure by adding new world states (along with their corresponding transitions) that denote failed states that are likely to be reached during the plan execution. State-transition systems are commonly used in model-checking planning approaches [6].

Definition 1. *A state-transition system consists of a set of (world) states and transitions between states, which are labeled with actions from the set A. A state-transition system T is defined as a 4-tuple* $T = \langle F, S, A, R \rangle$*, where:*

- *F is a finite set of fluents*
- $S \subseteq 2^F$ *is a finite set of states*
- *A is a finite set of actions over S*
- *R is the state-transition function* $R : S \times A \rightarrow S$ *that represents a transition between two states labeled with an action from A.*

Given a plan $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ of an execution agent, we will denote the sequence of states generated through a successful execution of π as $S = \langle s_0, s_1, s_2, \ldots, s_{n-1}, s_G \rangle$, where:

- $s_0 = I$ is the initial state, and s_G is the final state such that $G \in s_G$
- $result(s_0, a_1) = s_1, result(s_1, a_2) = s_2, \ldots result(s_{n-1}, a_n) = s_G$

The plan π and the sequence of states S traversed by a successful plan execution define the *basic* T of an execution agent. Particularly, S is the sequence of states; F is the set of fluents contained in all of the states in S; A is the set of actions in π; and R is the *result* function. In the following, we will refer to a basic state-transition system as a tuple $T = \langle \pi, S \rangle$.

3.1 Extending the basic state-transition system

Once the reactive planner builds T from the plan of an execution agent, the next step is to extend it by including new states and transitions in T. We can distinguish two main situations when a plan failure occurs. The **first situation** is that the failure in the action execution makes the agent remain exactly at the same state that it was initially. In other words, this situation occurs when $result(s_i, a_{i+1}) = s_i$. In this case, the agent finds itself in the same state and an alternative course of actions from s_i is necessary, if possible, to reach s_{i+1}. This situation is usually due to a malfunction in the action execution that leaves the world unaffected. The **second situation** typically arises when exogenous events occur in the environment. In this case, after the action execution, the agent is neither at s_i nor s_{i+1} but in a different state, where at least one of the fluents that model the resulting world state does not have the expected value.

Given a basic state-transition system $T = \langle \pi, S \rangle$ of an execution agent, T is extended to account for the first situation as follows: for each state $s_i \in S$, we search for alternative transition paths that connect s_i to another state in S; that is, we search for applicable actions in s_i (actions such that $pre(a) \subseteq s_i$), other than a_{i+1}, that lead to any forward recovery state $s_{i+1}, s_{i+2}, \ldots, s_G$. This process is performed through a forward state-space search. On the other hand, we might consider there is only one desirable recovery state to reach: that is, the following state in the sequence. If the *MO* module detects an error when trying to reach state s_i, it is likely the *MO* would request a plan to reach solely s_{i+1} as this solution would permit then to continue with the execution of the next action in π. This is so because in reactive planning a quick response that allows to continue with the plan execution is preferable over a more time-consuming solution. The consideration of exogenous events in $T = \langle \pi, S \rangle$ is addressed as follows:

1) For each pair (s_i, a_{i+1}) such that $s_i \in S$, $a_{i+1} \in \pi$ and a_{i+1} is applicable in s_i, we create a list $L = \{\langle v, d \rangle\}$ that contains the fluents that appear in $pre(a_{i+1})$.
2) Let v be a variable of a fluent $\langle v, d \rangle$ in L whose domain is $D_v = \{d_1, \ldots, d_m\}$; for each value $d_i \neq d$, we create a new state $s' = s \setminus \langle v, d \rangle \cup \langle v, d_i \rangle$. This operation is repeated for the variables that appear in all fluents in L.
3) We connect the new states generated in 2) to another state in S by following the same procedure explained above.

Therefore, we consider all possible contingencies in the value of the variables that appear in the preconditions of the actions of π. It is also possible that none of the states in S are reachable from a state generated due to an exogenous event; that is, there is no transition path from the new state to any of the states in S.

An extended state-transition system T' is a tuple $T' = \langle \pi, S, A', S' \rangle$, where π and S are the components of the basic state-transition system T, and A' and S' are the added transitions and states, respectively.

3.2 Fixing a plan failure

The list *monitor-parameters* that the *MO* module receives from the planning agent (see Fig. 2) contains elements of the form $\langle v, d, t_s, t_e \rangle$, where v is the variable, d is the expected value for v and $[t_s, t_e]$ is the time interval during which v is expected to take value d. When the *MO* receives the *state* from the *EX* after the execution of an action, the *MO* checks whether all the fluents contained in the *state* match the items in the *monitor-parameters* list or not. If not, the *MO* calls the *RP* to inform about a plan failure, passing the *RP* the current world *state* received from the sensors of the *EX* (we will call this state s_{cur} in the following), the recovery state to reach (s_{rec}) and the particular set of fluents that need to be repaired; i.e, the failed variables along with their expected values.

Assume $T' = \langle \pi, S, A', S' \rangle$ is the extended state-transition system defined in the *RP*. When the *RP* receives s_{cur} and s_{rec} from the *MO*, it performs the following operations: 1) find a state in $S \cup S'$ such that $s_{cur} \in S \cup S'$; 2) find a state in S such that $s_{rec} \in S$; 3) apply a simple version of the Dijkstra's algorithm to find a path from s_{cur} to s_{rec}. The state s_{rec} will always be a state from S as the objective is to take the execution back to a state from the original plan. The state s_{cur} will always be a state from either S

(first situation) or S' (second situation) since we consider all possible fluents that may appear in the state-transition system model. However, it might be the case there is not a transition path from s_{cur} to s_{rec} in whose case the *RP* will resort first to the other agents.

Multi-Agent Reactive Planner. The idea of cooperatively solving a plan failure at execution time requires to combine the state-transition systems of the agents involved in the system. Let EA_1 and EA_2 be two execution agents and $T_1 = \langle \pi_1, S_1, A'_1, S'_1 \rangle$ and $T_2 = \langle \pi_2, S_2, A'_2, S'_2 \rangle$ be their extended state-transition systems, respectively. First thing to note is that the planning tasks of the agents are different and so will be their sets of actions A_1 and A_2 and, equivalently, the set of fluents and states in T_1 and T_2. However, EA_2 will be able to help EA_1 fix its plan failure if the fluent to be repaired belongs to the knowledge shared between both agents. The *shared data* by the entities of the problem is defined at planning time before the deliberative planner builds the plans for the agents. Assume $\langle v, d \rangle$ is the fluent to repair that the *RP* of EA_1 sends to the *RP* of EA_2. EA_2 will match its s_{cur_2} in T_2 as well as the states in $S_2 \cup S'_2$ in which $\langle v, d \rangle$ holds. If a path from s_{cur_2} to any of the states holding $\langle v, d \rangle$ exists then EA_2 can actually help EA_1. The final response will depend on how this transition path deviates from its plan π_2 and the cooperative behavior defined in the agents. Finally, if EA_2 cannot actually help EA_1 or is *not willing* to, EA_1 will call its planning agent for replanning the problem, i.e., find a new plan.

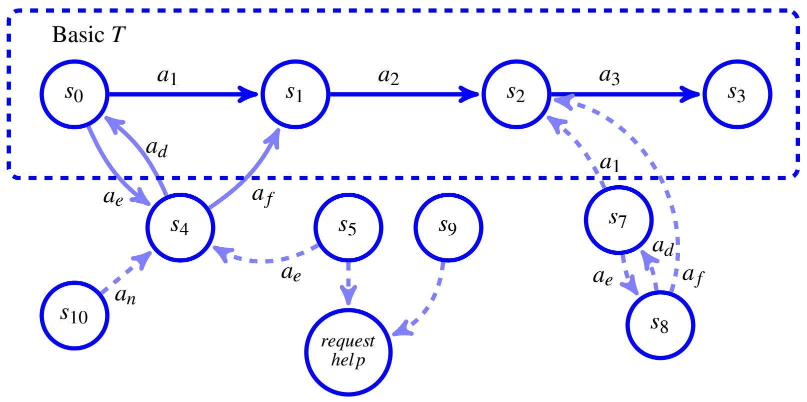
4 MARS Domain, an example of application

In this section, we will show how our reactive architecture works on a possible Mars Domain scenario. Space missions of NASA have rovers in Mars. When a plan failure occurs, rovers communicate with a control center on Earth for repairing the failure. NASA is interested in providing rovers with on-board reactive planning and execution capabilities, so that they can perform the reparation by themselves or with the help of other agents in a timely fashion and thus reducing the communication overhead with the Earth.

We present a simple example that shows the state-transition system of a rover for repairing future plan failures. Let's suppose we have a rover A, whose mission is to analyze rocks and communicate the results to a Lander L, which in turn sends the results to the Earth. There are three waypoints $\{\mathsf{w}_1, \mathsf{w}_2, \mathsf{w}_3\}$ located on the surface; w_2 is the initial location of L and the rover A. Rover A has good maps to travel from w_1 to both w_2 and w_3, and from waypoint w_2 to w_3. The mission of the rover is to use the microscopic camera to analyze rocks in waypoint w_1 and communicate the results to L. The deliberative planner computes the plan $\pi = \langle a_1, a_2, a_3 \rangle$ for A (see Fig. 3 bottom).

Using the plan π, the *RP* module generates the basic transition system T, which consists of the states $S = \langle s_0, s_1, s_2, s_3 \rangle$, and transitions labelled with actions a_1, a_2, a_3, as shown in Fig. 3.

Then, the RP extends T by considering the two situations explained in 3.1: those cases in which the rover remains in the same state after executing an action, and exogenous events. In the first situation, T is extended with state s_4 and transitions $A' = \langle a_e, a_d, a_f \rangle$. The transition a_e represents the action (navigate A w_2 w_3), which changes the fluent $\langle pos_A, \mathsf{w}_2 \rangle$ to $\langle pos_A, \mathsf{w}_3 \rangle$ and generates the new state s_4. The transition a_f,



$\pi = \langle a_1{:}(\mathit{navigate}\ A\ w_2\ w_1),\ a_2{:}(\mathit{sample\ rock}\ A\ \mathit{roverAstore}\ w_1),\ a_3{:}(\mathit{commSample\ rock}\ A\ L\ w_1\ w_2)\ \rangle$

Fig. 3. State-transition system T for A.

which is a path to the state s_1, represents the action (navigate A w_3 w_1), which changes the fluent $\langle pos_A, \mathsf{w}_3\rangle$ to $\langle pos_A, \mathsf{w}_1\rangle$.

T is now augmented with the states that may arise with the appearance of exogenous events. The states labeled as $s_5, s_7, s_8, s_9, s_{10}$ are incorporated to T along with the transitions shown in Fig. 3[5]. For example, s_5 represents a situation in which the rover finds the path from w_2 to w_1 is blocked. Then, s_5 will be the same state as s_0 except that the fluent $\langle path_{w_2 w_1}, true\rangle$ is not present in s_5. Since A can reach w_3 from s_5, the transition a_e also connects s_5 to s_4. Another example of exogenous event is s_9, which represents the lander is not in w_2, where it was supposed to be to communicate the results. As the position of L is unknown to rover A, it is not possible to reach a state of S from s_9, so the only possible solution is to request for help to the other rovers or to the Earth. If, on the contrary, the rover A knew that L is in w_3 then a transition from s_9 to s_3 would appear in the state transition system.

As it can be observed, the information comprised in the state transition system T allows rover A to quickly find a solution to a failure because all the possible contingencies which can be modeled with the variables in the agent's domain are considered in T. Additionally, it is easy to combine the information from two or more different state transition systems.

5 Conclusions and Future Works

In this paper, we have presented a first approach to a recovery mechanism from plan failures that makes use of state-transition systems as flexible reactive structures. Each agent comprises its own transition system that accommodates a subset of the possible failed states that the agent would encounter during the execution of its plan. Through

[5] For simplicity, we do not show all the states that would be generated.

this reactive structure, we can easily locate the failed state and find, if possible, a sequence of transitions that lead the agent to a recovery state. Additionally, agents can use their state-transition systems to respond to agents' requests. In future works, we intend to exploit this research direction to create a conflict resolution mechanism for solving plan failures among multiple agents using reactive *teamworks* at execution time that work together in the accomplishment of a cooperative goal.

References

1. Ai-Chang, M., Bresina, J., Charest, L., Chase, A., Hsu, J.J., Jonsson, A., Kanefsky, B., Morris, P., Rajan, K., Yglesias, J., Chafin, B., Dias, W., Maldague, P.: MAPGEN: Mixed-initiative planning and scheduling for the Mars Exploration Rover mission. IEEE Intelligent Systems 19(1), 8–12 (Feb 2004)
2. Aschwanden, P., Baskaran, V., Bernardini, S., C. Fry, M.M., Muscettola, N., Plaunt, C., Rijsman, D., Tompkins, P.: Model-unified planning and execution for distributed autonomous system control. In: Workshop on Spacecraft Autonomy: Using AI to Expand Human Space Exploration. AAAI Press (2006)
3. B. Browning, J. Bruce, M.B.M.V.: Stp: Skills, tactics and plays for multi-robot control in adversarial environments. IEEE Journal of Control and Systems Engineering 219, 33–52 (2005)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
5. Cesta, A., Cortellessa, G., Fratini, S., Oddi, A.: Developing an End-to-End Planning Application from a Timeline Representation Framework. In: IAAI-09. Proceedings of the 21^{st} Innovative Applications of Artificial Intelligence Conference, Pasadena, CA, USA (2009)
6. Cimatti, A., Roveri, M., Bertoli, P.: Conformant planning via symbolic model checking and heuristic search. Artif. Intell. 159(1-2), 127–206 (2004)
7. Firby, R.J.: Task networks for controlling continuous processes. In: Artificial Intelligence Planning Systems: Proceedings of the First International Conference, 1994. pp. 49–54. Morgan Kaufmann Pub (1994)
8. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning. In: Proceedings of AAAI-87 Sixth National Conference on Artificial Intelligence. pp. 677–68. Seattle, WA (USA) (Jul 1987)
9. Ghallab, M., Laruelle, H.: Representation and control in IxTeT, a temporal planner. In: Proceedings of the 2nd International Conference on AI Planning Systems (1994)
10. Guzman, C., Alcazar, V., Prior, D., Onaindia, E., Borrajo, D., Fernández-Olivares, J., Quintero, E.: Pelea: a domain-independent architecture for planning, execution and learning. In: Scheduling and Planning Applications woRKshop (SPARK) ICAPS 2012 (2012)
11. Katia P. Sycara, Massimo Paolucci, M.V.V.J.A.G.: The retsina mas infrastructure. Autonomous Agents and Multi-Agent Systems 7(1-2), 29–48 (2003)
12. Lesser, V., Decker, K., Wagner, T., Carver, N., Garvey, A., Horling, B., Neiman, D., Podorozhny, R., Prasad, M.N., Raja, A., Vincent, R., Xuan, P., Zhang, X.Q.: Evolution of the gpgp/taems domain-independent coordination framework. Autonomous Agents and Multi-Agent Systems 9(1-2), 87–143 (2004)
13. Sebastian Sardiña, L.P.: A bdi agent programming language with failure handling, declarative goals, and planning. Autonomous Agents and Multi-Agent Systems 23(1), 18–70 (2011)
14. Simmons, R.: Concurrent planning and execution for autonomous robots. In: IEEE International Conference on Robotics and Automation. pp. 46–50 (1992)
15. Such, J.M., Garcia-Fornes, A., Espinosa, A., Bellver, J.: Magentix2: a Privacy-enhancing Agent Platform. Engineering Applications of Artificial Intelligence (2012)
16. Wilkins, D.E., Myers, K.L.: A multiagent planning architecture (1998)