Roberto Micalizio. Action Failure Recovery via Model-Based Diagnosis and Conformant Planning. COMPUTATIONAL INTELLIGENCE. 29 (2) pp: 233-280.
DOI: 10.1111/j.1467-8640.2012.00444.x

Computational Intelligence, Volume 000, Number 000, 2011

Action Failure Recovery via Model-Based Diagnosis and Conformant Planning

Roberto Micalizio

Dipartimento di Informatica, Università di Torino,
corso Svizzera 185, 10149 - Torino,
tel. +39 011 6706735, email: micalizio@di.unito.it

A plan carried on in the real world may be affected by a number of unexpected events, *plan threats*, which cause significant deviations between the intended behavior of the plan executor (i.e., the agent) and the observed one. These deviations are typically considered as action failures.

This paper addresses the problem of recovering from action failures caused by a specific class of plan threats: *faults* in the functionalities of the agent. The problem is approached by exploiting techniques of the Model-Based Diagnosis (MBD) for detecting failures (*plan execution monitoring*) and for explaining these failures in terms of faulty functionalities (*agent diagnosis*). The recovery process is modeled as a replanning problem aimed at fixing the faulty components identified by the agent diagnosis. However, since the diagnosis is in general ambiguous (a failure may be explained by alternative faults), the recovery has to deal with such an uncertainty. The paper advocates the adoption of a conformant planner, which guarantees that the recovery plan, if it exists, is executable no matter what the actual cause of the failure is.

The paper focuses on a single agent performing its own plan, however the proposed methodology takes also into account that agents are typically situated into a multi-agent scenario and that commitments between agents may exist. The repair strategy is therefore conceived to overcome the causes of a failure while assuring the commitments an agent has agreed with other team members.

Key words: Model-Based Diagnosis, Plan Execution Monitoring, Conformant Planning

1. INTRODUCTION

An agent performing a plan in the real-world should be in charge of monitoring its environment and the actual effects of its actions as they may fail for a number of reasons. In order to make the phase of plan execution robust to failures, many strategies to plan repair have been recently proposed (Gerevini and Serina, 2000; van der Krogt and de Weerdt, 2005a; Fox *et al.*, 2006). The basic idea of these approaches is that, during the plan execution, changes in the goals (e.g., new goals can be added), or changes in the environment (e.g., a door expected to be open is actually closed) make the current plan no longer adequate for achieving the desired goals. The plan execution is therefore stopped, and a plan repair mechanism is activated to adjust the current plan to the situation actually encountered at execution time.

As pointed out by Cushing and Kambhampati (2005), however, plan repair cannot assume that execution failures are independent of the agent's behavior; when such assumption is made, the agent might repeat indefinitely the same error. For instance, let us consider the blocks world domain and assume that an agent fails in picking a block up because of an error in calculating the movements of its arm, in this case adapting the initial plan by introducing a new *pick up* action may resolve the problem. However, if the same *pick up* action fails due to a fault in the agent's arm, there will be no advantage in trying to execute a *pick up* action again since the action would inevitably fail. To overcome this situation, one should first remove the root causes of the failure (i.e., the fault in the handling apparatus), and then attempt to repair the plan by inserting in the original plan a new *pick up* action.

In this paper we intend to complement previous approaches to plan repair by taking into account the problem of recovering from action failures caused by *faults*. To this end, we adopt Model-Based Diagnosis (MBD) in order to detect action failures (*plan execution monitoring*) and to explain these failures in terms of faulty functionalities (*agent diagnosis*).

Our idea, in fact, is that the first step for handling effectively an action failure consists in

removing the root causes of the failure identified by the agent diagnosis, and restoring the nominal conditions in the agent functionalities. After this fundamental step, either the agent resumes the execution of the original plan from the same point where it was stopped; or, if required, a plan repair mechanism is invoked to adjust the rest of the plan. In this paper, we will focus on the problem of recovering from an action failure so that the execution of the original plan can be resumed.

One of the main challenges in pursuing this objective is that the agent diagnosis cannot be anticipated; thus the recovery strategy must be based on a planner which synthesizes on-the-fly a recovery plan, and whose goal consists in fixing the faulty functionalities mentioned by the agent diagnosis. Moreover, the agent diagnosis is typically ambiguous (several faults can explain the same action failure), thereby the planner synthesizing the recovery plan must be able to deal with such ambiguity. To cope with these problems, we adopt a conformant planner as this kind of planners is able to deal with ambiguous initial states, and assures that the recovery plan, when it exists, is executable even though the actual cause of the failure is not precisely known.

Albeit the recovery strategy we propose is based on a single agent that can just change its own plan, we also consider the problem from a wider point of view by situating the agent into a multi-agent setting. When an agent shares its environment with other agents, it has to consider that its recovery plan may interfere with the plans the other agents are carrying on. In this paper our objective is to make the recovery process of an agent transparent to all the other agents. This means that, on the one side the recovery cannot acquire new resources to avoid negotiations with other agents; on the other side, the recovery must guarantee that the commitments an agent has already agreed with other team members will be preserved. To model such a multi-agent setting, we adopt the notion of Multi-Agent Plan (MAP) (Durfee, 2001), in which commitments and dependencies among agents are explicitly modeled as precedence and causal links between action instances.

A further difficulty in dealing with a multi-agent scenario is that, when a recovery plan does not exist (e.g., an agent cannot fix a fault on its own), the impaired agent can become a latent menace for the other agents; for instance, an agent may lock indefinitely critical resources. We try to limit the impact of unrecoverable faults by switching the goal of the recovery strategy from "fixing the faulty functionalities" to "reaching a safe status", that is, a state where the impaired agent does not hold any resource. As we will see, the main advantage of the proposed approach is that the conformant planner used for the synthesis of a recovery plan can also be used for the synthesis of a plan to the safe status.

Organization The paper is organized as follows. The next section recalls some basic notions about classical planning and multi-agent planning; these concepts are then exemplified in section 3 where we briefly introduce a multi-agent scenario that will be used to illustrate the proposed methodology throughout the paper. Section 4 delineates the main control strategy that allows an agent to coordinate with other teammates and to supervise the progress of its own local plan. The following three sections, 5, 6, and 7, discuss the three activities involved in the control strategy: the monitoring, the diagnosis and the recovery, respectively. In particular, the recovery relies on a conformant planner, which is presented in section 8 where we also motivate why a recovery plan must satisfy a so stringent requirement. In section 9 we go back to the example and show how the control strategy actually intervenes during the execution of a multi-agent plan by detecting and recovering from action failures. The effectiveness of the repair methodology is discussed in section 10, in which an extensive experimental analysis is presented. Finally, in section 11 the proposed approach is compared with other relevant works in the literature.

2. BACKGROUND

In this section we briefly recall some basic notions on classical and multi-agent planning which will be useful in the rest of the paper.

Classical Planning Classical planning is traditionally formalized in terms of propositional logics[1]. According to Nebel (2000), in the propositional framework a system state s is modeled as a subset of

[1]See for instance the seminal works about STRIPS (Fikes and Nilsson, 1971), and more recently the Planning Domain Definition Language (PDDL) (Ghallab *et al.*, 1998; Fox and Long, 2003)

literals in Σ, that is the set of all the propositional atoms modeling the domain at hand, each of which may appear in its positive or negated form. A plan operator (i.e., an *action instance*) $o \in 2^{\Sigma} \times 2^{\widehat{\Sigma}}$ is defined by a set of preconditions $pre \subseteq \Sigma$ and its effects $\mathit{eff} \subseteq \widehat{\Sigma}$; where $\widehat{\Sigma}$ is the set Σ augmented with $\bot$ (i.e., false) and $\top$ (i.e., true).
The application of an operator o to a state s is defined as $App : 2^{\widehat{\Sigma}} \times o \rightarrow 2^{\widehat{\Sigma}}$

$$App(s,o) = \begin{cases} (s - \neg \mathit{eff}(o)) \cup \mathit{eff}(o), & \text{if } s \models pre(o) \text{ and } s \not\models \bot \text{ and } \mathit{eff}(o) \not\models \bot \\ \{\bot\}, & \text{otherwise.} \end{cases}$$

Of course, actions are deterministic: when the preconditions $pre(o)$ are satisfied s the effects $\mathit{eff}(o)$ are always achieved.

A planning problem is the tuple $\Pi = \langle \Sigma, O, I, G \rangle$ where:

- Σ is the set of propositional atoms, also called fact or fluent;
- $O \subseteq 2^{\Sigma} \times 2^{\widehat{\Sigma}}$ is the set of available plan operators;
- $I \subseteq \widehat{\Sigma}$ is the initial state;
- $G \subseteq \Sigma$ is the goal state.

Solving a planning problem requires to find a sequence of plan operators that when applied to the initial state I reaches the goal state G.
Multi-Agent Planning A multi-agent plan (MAP) can be seen of as an extension of a Partial-Order Plan (POP) (Weld, 1994) where deterministic actions, rather than being assigned to a single agent, are distributed among a number of cooperating agents in a team $\mathcal{T}$. Since agents share the same environment, they also share the same set of critical resources RES; i.e. resources that can only be accessed in mutual exclusion. A MAP has therefore to achieve the expected goals while guaranteeing the consistent access to the resources. The formalism we adopt for modeling a MAP is a simplified version of the formalism presented by Cox *et al.* (2005). In our view, a MAP instance P is the tuple $\langle \mathcal{T}, RES, A, E, C, R \rangle$, where:

- $\mathcal{T}$ is the team of agents; agents will be denoted by the letters i and j;[2]
- RES is the set of critical resources available in the environment; in this paper we assume that all the resources are renewable: they can be locked and relinquished by an agent, but are never consumed; for instance, renewable resources are tools, locations, objects, and so on;
- A is the set of the action instances the agents have to execute. Each action instance $a \in A$ is assigned to a specific agent $i \in \mathcal{T}$;
- E is a set of precedence links between actions: a precedence link $a \prec a'$ in E indicates that a' can start only after the completion of action a;
- C is a set of causal links of the form $cl : a \xrightarrow{q} a'$; the link cl states that the action a provides the action a' with the service q (q is an atom occurring both in the effects of a and in the preconditions of a');
- R is a set of precedence links specifically used to rule the access to the resources. Such a kind of link has the form $a \prec_{res} a'$, meaning that action a precedes a' in the use of resource $res \in RES$.

While causal links model the exchange of services between agents, precedence links in R guarantee the *concurrency requirement* (Roos and Witteveen, 2009), for which two actions a and a', assigned to different agents, cannot be executed at the same instant if they require the same resource res; the two actions must be serialized by adding either $a \prec_{res} a'$ or $a' \prec_{res} a$ in R.
The goal G achieved by the MAP instance P consists of a conjunction of propositions that must hold in the final state. In general, an agent is responsible for providing just a sub-goal and hence a subset of the propositions mentioned in G.

The problem of synthesizing the MAP P is outside the scope of this paper, possible solutions have been addressed in (Boutilier and Brafman, 2001; Jensen and Veloso, 2000; Cox *et al.*, 2005). Independently of how the MAP P has been built, we assume it satisfies the following properties:

[2] In principle, agents could be heterogeneous; i.e., two agents may have different actuators; but for the sake of simplicity in the discussion, we will assume that all the agents are of the same type.

go `(AG, FROM, TO)`
pre: `place(FROM), place(TO), position(AG, FROM)`
eff: `position(AG, TO), ¬position(AG, FROM)`

load `(AG, FROM, OBJ)`
pre: `place(FROM), position(AG, FROM), at(OBJ, FROM), loaded(AG, empty)`
eff: `at(OBJ, AG), loaded(AG, OBJ), ¬at(OBJ, FROM), ¬loaded(AG, empty)`

unload `(AG, TO, OBJ)`
pre: `place(TO), position(AG, TO), at(OBJ, AG), loaded(AG, OBJ)`
eff: `at(OBJ, TO), loaded(AG, empty), ¬at(OBJ, AG), ¬loaded(AG, OBJ)`

FIGURE 1. The set of action templates in the office domain.

- *executable*: the plan is deadlock free, and the concurrency requirement is satisfied;
- *correct*: all the domain-dependent constraints on the use of resources are satisfied, and the global goal is actually reached if no unexpected event occurs during the plan execution.

Local plans In our approach, each agent has just a partial view of the multi-agent plan P; in particular, an agent i knows just its own local plan $P^i{=}\langle A^i, E^i, C^i, T^i_{in}, T^i_{out}, R^i_{in}, R^i_{out}\rangle$: A^i, E^i and C^i have the same meaning as A, E and C, respectively, restricted to actions assigned to agent i. Thus E^i and C^i only contain links between actions in A^i. Whereas the sets T^i_{in}, T^i_{out}, R^i_{in}, and R^i_{out} contain links between actions of different agents. More precisely, T^i_{in} is a set of incoming causal links of the form $a \xrightarrow{q} a'$ where a' belongs to A^i and a is assigned to another agent j in $\mathcal{T}$; T^i_{out} is a set of outgoing causal links $a \xrightarrow{q} a'$ where $a \in A^i$ and $a' \in A^j$ $(i \neq j)$. Similarly, R^i_{in} is a set of incoming precedence links of the form $a \prec_{res} a'$ where a' belongs to A^i and a is assigned to another agent j in the team; finally, R^i_{out} is a set of outgoing precedence links $a \prec_{res} a'$ where $a \in A^i$ and $a' \in A^j$.

For the sake of simplicity, we assume that each local plan P^i extracted from P is *totally ordered*, and hence P^i can be seen as the ordered sequence of actions $[a^i_0, a^i_1, \ldots, a^i_n, a^i_\infty]$. Where, as usual, a^i_0 and a^i_∞ are two pseudo actions modeling, respectively, the initial state and the sub-goal of the local plan P^i (Weld, 1994).

3. AN EXAMPLE

Let us consider a simple applicative scenario where two agents, `A1` and `A2`, provide a delivery service in an office. The task domain is modeled in propositional terms through the following set of atoms:

- `place`: denotes the resources available in the environment. In our example, we are going to consider an office with two desks, `place(Desk1)` and `place(Desk2)`, a repository for the parcels, `place(Rep)`, and a parking area `place(Parking)`. The repository and the two desks are critical resources that can be accessed by no more than one agent at any time instant, while `Parking` is not constrained and hence many agents can be simultaneously located in it;
- `position`: models the position of an agent, for instance `position(A1, Rep)` means that the agent `A1` is located within the repository;
- `at`: models the position of an object, namely the parcels the agents have to deliver to the desks; for instance `at(Pack1, Rep)` means that the parcel `Pack1` is stored into the repository `Rep`;
- `loaded`: indicates whether an agent is carrying an object or not, the atom `loaded(A1, Pack1)` states that `A1` is loaded with the parcel `Pack1`, whereas `loaded(A1, empty)` means that `A1` is not carrying any object.

Figure 1 shows some examples of action templates in our office domain. The `go` action template models the movement of an agent from a place `FROM` to another place `TO`. While the `load` and `unload` action templates describe how an agent can operate on a parcel. Of course, these templates need to be instantiated w.r.t. concrete atoms in order to obtain an action instance that an agent can actually perform.

```
place(Desk1), place(Desk2), place(Rep), place(Parking)
position(A1,Rep), position (A2, Desk2)
loaded(A1, empty), loaded(A2, Pack2)
at(Pack1, Rep), at(Pack2, A2)
```

FIGURE 2. The atoms which are initially true.

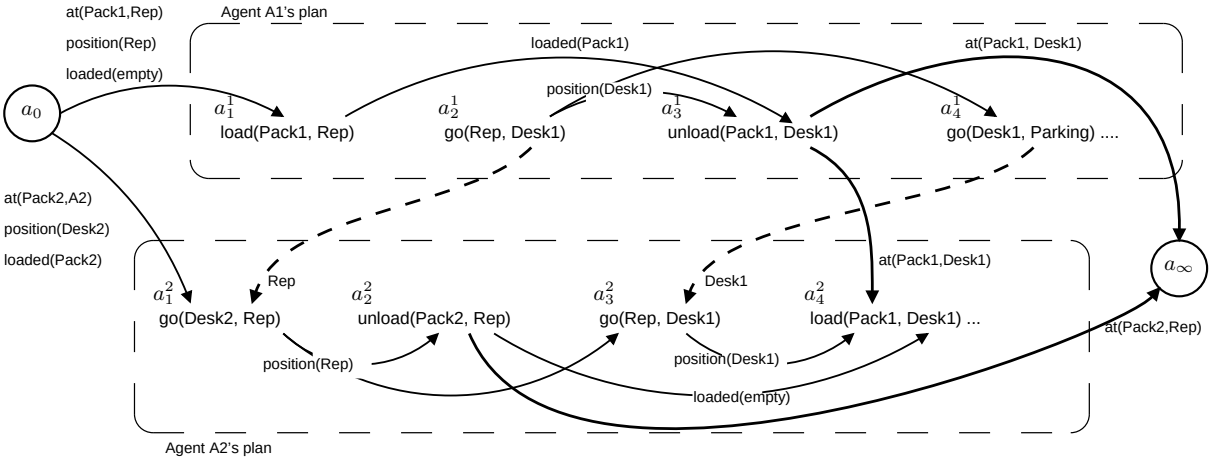

FIGURE 3. The MAP instance P assigned to the team of agents.

Let us assume that, as a general rule, agent `A1` is in charge of delivering parcels to desks picking them up from the repository; while agent `A2` is in charge of collecting parcels from desks and taking them back to the repository. In our specific case, we have that agent `A1` is assigned to deliver parcel `Pack1` to `Desk1` while agent `A2` has to move the two parcels `Pack1` and `Pack2` back to the repository. Of course, agent `A2` expects to collect parcel `Pack1` from `Desk1`, and hence agent `A1` has to deliver that parcel first.

The initial set of atoms is given in Figure 2: agent `A1` is in the repository and it is empty, so it is ready to load parcel `Pack1`; agent `A1` is in `Desk2` and already loaded with parcel `Pack2`. Figure 3 shows a portion of the multi-agent plan assigned to the two agents. For the sake of readability of the picture, the action instances and atoms have been simplified by removing the agent identifier; it can be easily determined by considering that all the actions of the same agent are encircled by a dashed frame, and hence all the actions and atoms within the same frame refer to the corresponding agent. For instance, the local plan assigned to agent `A1` involves four actions: load `Pack1` from `Rep`, move from `Rep` to `Desk1`, unload `Pack1` onto `Desk1`, and move from `Desk1` to the parking area.

The multi-agent plan resembles a Directed Acyclic Graph (DAG) where nodes are action instances and edges between nodes represent either causal or precedence dependencies between actions. Precedence links ruling the access to the resources are labeled with the name of the resource they refer to. For example, the link between a_2^1 and a_1^2 is labeled with the resource name `Rep`; it means that `Rep` will be released by agent `A1` after the execution of action a_2^1, and it will be acquired by agent `A2` to perform action a_1^2. Causal links are depicted as solid arrows and labeled with the atom provided by one action to the other; e.g., the link between actions a_3^1 and a_4^2 means that the atom `at(Pack1, Desk1)` will be provided by agent `A1` to agent `A2` as an effect of action a_3^1. In this case, the atom labeling a causal link resembles the concept of service exchanged between agents. Since local plans are totally ordered, precedence links between two consecutive actions are omitted for the sake of readability.

Let us assume that, during the plan execution phase, a fault affects the robotic arm of agent `A1` while it is performing action a_1^1; as a consequence, parcel `Pack1` has not been loaded on-board the agent: the action has failed. If agent `A1` is not controlling the execution of its local plan, it will ignore the occurrence of the action failure and continue the execution of its local plan: the following

`go` action is successfully carried out as the movement from `Rep` to `Desk1` is not affected by the fault in the robotic arm. However, as soon as agent `A1` tries to perform action a_3^1, it gets stuck since one of the action preconditions (namely, the atom `loaded(A1, Pack1)`) is missing: agent `A1` is in a deadlock condition as it will wait indefinitely for a service that only the agent itself can provide. At the same time, the resource `Desk1` is locked and no other agent can access it; thus the fault in `A1`'s robotic arm will propagate in the whole plan and will also affect the actions `A2` is responsible for. Moreover, even though the system is blocked, no recovery strategy is activated as no failure has been detected.

This simple example shows the necessity of detecting action failures as soon as possible and properly reacting to them. The methodology we propose in this paper handles action failures in two ways: first, it tries to recover from a failure by fixing the faulty components; second, when the faulty components cannot be fixed, the methodology tries to reduce the impact of the failure by moving the impaired agent into a "safe condition", that is a condition where the agent does not hold any resource and does not hinder other teammates.

4. DISTRIBUTED CONTROL STRATEGY

In a multi-agent setting, where the execution of a multi-agent plan is carried on in parallel by a team of cooperating agents, it appears natural to adopt a distributed approach also for the control of the plan execution. This means that each team member must be able to:

- Determine when the preconditions of the next action to perform are satisfied;
- Activate the execution of the next action;
- Determine the outcome of the last performed action;
- In case of non-nominal outcome, activate a strategy for recovering from the failure so that the plan execution can be resumed.

To get these results, we propose a local control task consisting of three main activities:

- *plan execution monitoring*, corresponding to the *fault detection* phase in the MBD terminology, tracks the status of the agent over time while the plan execution is in progress, and detects deviations between the expected and the observed agent's behavior. In this paper, such deviations can only be due to non-observable faults affecting the agent's functionalities;
- *agent diagnosis*, corresponding to the *fault identification* phase in MBD, explains the detected action failures in terms of faults in the agent's functionalities;
- *action recovery* aims at bringing the status of an agent back to its nominal conditions; the recovery has to fix the components that have been qualified as faulty by the agent diagnosis. In this paper we propose a local recovery strategy which only intervenes on the plan of the impaired agent while it is completely transparent for all the other agents in the team.

Since agents execute their actions concurrently, they need to coordinate with one another in order to avoid the violation of the constraints defined during the planning phase. Effective coordination is obtained by exploiting the causal and precedence links in P. As pointed out by Decker and Li (2000), in fact, coordination between agents i and j is required when i provides j with a service q. This is explicitly modeled in our framework by a causal link $cl : a_h^i \xrightarrow{q} a_k^j$ in C, which belongs both to T_{out}^i and to T_{in}^j as an effect of the decomposition of P. Therefore, on the one hand $cl \in T_{out}^i$ tells agent i to send agent j a message about the service q as soon as the action a_h^i has been completed; on the other hand, $cl \in T_{in}^j$ tells agent j that to perform action a_k^j a message about the service q has to be received from agent i. Likewise, the consistent access to the resources is a form of coordination which involves precedence links.
It follows that coordination among agents is only possible when the following requirement is satisfied.

Requirement 1: *Observability requirement.* Each agent observes (at least) the direct effects of the actions it executes.

Note that this requirement must be satisfied even under the hypothesis that no fault can occur as the plan execution is distributed and agents are not synchronized with one another. Moreover, since we are interested in monitoring the plan execution even when something wrong occurs, we also assume:

LocalControlLoop(P^i)
00 $t \leftarrow 0$
01 **while** there are actions in P^i to be performed
02 $a_l^i \leftarrow$**nextAction**(P^i);
03 **if** preconditions of a_l^i are satisfied
04 $\langle$EXECUTE $a_l^i\rangle$
05 $obs^i(t+1) \leftarrow$ $\langle$ gather observations $\rangle$
06 *outcome* $\leftarrow$**Monitoring**($a_l^i, t, obs^i(t+1)$)
07 **if** *outcome* is nominal
08 mark a_l^i as performed
09 **else**
10 $D^i \leftarrow$ **Diagnosis**(a_l^i)
11 $P^i \leftarrow$ **Recovery**(P^i, a_l^i, D^i)
12 **if** P^i is empty
13 $\langle$STOP EXECUTION$\rangle$
14 **else** $\langle$WAIT$\rangle$ /*some preconditions are not satisfied yet*/
15 $t \leftarrow t+1$

FIGURE 4. The local control strategy.

Assumption 1: *Determinable outcome.* The amount of observations an agent receives are always sufficient to determine the outcome of every action the agent performs.

This means that right after the execution of an action, the agent is able to determine whether the action effects have been achieved or not. Intuitively, the outcome of an action can be either *succeeded* or *failed*; this concept will be formalized in section 5 together with the process that leads an agent to the detection of action failures.

The control strategy is sketched[3] in Figure 4 showing the high-level steps followed by an agent i while it is carrying on its local plan P^i. Each iteration of the while loop takes care of the execution of a single action: first, the agent determines its next action a_l^i to be performed (line 02), then it verifies whether the action preconditions are satisfied or not (line 03). Note that, since the plan P is correct, and since an agent immediately discovers the failure of one of its own actions (Assumption 1), the preconditions of an action can be unsatisfied only when some services that other agents have to provide are still missing. Thus, when action a_l^i is not executable yet, the agent keeps waiting for the missing services.
On the other hand, when the action preconditions are satisfied, action a_l^i is actually performed in the real world (line 04).

The outcome of an action is determined by the Monitoring activity (line 06) relying on the set of observations received by the agent. When a non-nominal outcome is detected, the Diagnosis (line 10) is activated in order to explain such an outcome and to provide the Recovery strategy with useful pieces of information. Recovery returns either a new local plan P^i (overcoming the problems which caused the non-nominal outcome) or an empty plan when the recovery strategy failed in finding a solution. In this last case, the execution of the local plan P^i is stopped. Of course, when the local recovery fails in handling a failure, a global plan repair strategy might be activated, such an option, however, is outside the scope of this paper.

Note that the while loop ends with the increment of the time t (line 15), this represents just a local clock for the agent i, namely, the agents are not temporally synchronized with one another.

In the rest of the paper we will examine these three activities (monitoring, diagnosis and recovery), providing for each of them a formalization in terms of Relational Algebra operators.

[3]In order to keep the algorithm simple, the interagent communication has been omitted.

5. MONITORING THE EXECUTION OF A LOCAL PLAN

In this section we address the first step of the control loop previously discussed: the plan execution monitoring. As sketched in section 4 the main objective of the monitoring task consists in detecting discrepancies between the expected behavior of agent i and its actual, observed activity.

To reach this objective, the monitoring needs two types of action models. The first type is the one used during the planning phase, where just the nominal evolution of an action is represented in terms of preconditions and effects. These models are used by the monitoring to create expectations about the correct agent's behavior. The second type of models is an *extended* version of the previous one which also includes anomalous evolutions of an action; that is, how an agent behaves when a fault occurs during the execution of a given action. This second type of models is used to trace the actual agent's behavior. By comparing the expected to the actual agent's behavior, the monitoring task is able to detect discrepancies and hence action failures.

In the remainder of this section we first formalize the extended action models and then the process that leads to the detection of action failures.

5.1. Agent Status

While the propositional language appears to be adequate during the planning phase (see Section 2), it becomes awkward to deal with when one has to consider the plan execution phase. Previous works on plan diagnosis (Roos and Witteveen, 2009; de Jonge *et al.*, 2009) have already shown that a state-based description of the world, with non-Boolean status variables, is more effective as it is easier to update when actions are actually performed, especially when unexpected deviations may occur. In this subsection we introduce the notion of agent status as a set of (non-Boolean) status variables; note that this representation is not in contrast with the propositional representation given above, in fact it is always possible to map each propositional atom into one (or more) status variables, and vice versa (Brenner and Nebel, 2009).

Given an agent $i \in \mathcal{T}$, the status of agent i is modeled by a set of discrete status variables VAR^i; for each variable $v \in VAR^i$, $dom(v)$ denotes the finite domain of v. The set VAR^i is partitioned into three subsets END^i, ENV^i and HLT^i:

- END^i maintains endogenous status variables modeling the internal state of agent i; for instance, in the example sketched in section 3, endogenous variables are: *position* where $dom(position)$={`Rep`, `Desk1`, `Desk2`, `Parking`}, and *loaded* where $dom(loaded)$ = {`Pack1`, `Pack2`, `empty`}; the `empty` value is added to model an agent which is not carrying any parcels.
- ENV^i maintains variables concerning the environment where agent i operates, namely, the status of the available resources. Let RES be the set of available resources, for each resource $res_k \in RES$, ENV^i includes a private variable $res_{k,i}$ whose domain is {*in-use*, *not-in-use*, *unknown*}: $res_{k,i}$=*in-use* indicates that resource res_k is being used by agent i at the current time; $res_{k,i}$=*not-in-use* means that resource res_k is assigned to agent i but it is not currently used; finally, $res_{k,i}$=*unknown* means that the agent i has no access to resource res_k. As we discuss in Section 7, the distinction between a resource in use, unused or unknown, is essential during the recovery phase to determine the objective of a repairing plan.
  Of course, since a resource variable is duplicated in as many copies as there are agents, maintaining the consistency among all these copies could be an issue. In our approach, however, this issue does not arise since conflicts for accessing the resources are solved at planning level. In fact, the precedence links in R guarantee that, at each time t, a resource res_k can only be used by a specific agent i; thereby for any other agent $j \in \mathcal{T} \setminus \{i\}$ the status of the res_k is *unknown*.
  In the rest of the paper we will denote as $AvRes(i,t)$ (available resources) the subset of resources that agent i holds at time t; i.e., $AvRes(i,t)$ ={$res_k \in RES | res_{k,i} \neq unknown$}.

- HLT^i is the set of variables modeling the health status of agent i. In fact, while in many approaches to plan repair (see e.g., (Gerevini and Serina, 2000; van der Krogt and de Weerdt, 2005a)) plan deviations are due to unexpected changes in the environment; in our framework plan deviations are caused by faults in some agent's functionalities. For this reason, we associate each agent functionality f with a variable $v_f \in HLT^i$ modeling the current operative mode of f; $dom(v_f) = \{ok, abn_1, \ldots, abn_n\}$, where ok denotes the nominal mode, while $abn_1, \ldots, abn_n$ denote anomalous or degraded modes. For example, the agent's arm is modeled via the variable *hnd* (i.e., handling functionality) whose domain is the set {*ok*, *blocked*}; while the agent's battery is modeled by the variable *pwr* whose domain is {*ok*, *low*, *flat*}.

It is important to note that, while the variables in END^i and in ENV^i can be mapped with the propositional formalism previously discussed, the variables in HLT^i have been added to capture the possible occurrence of faults, which is an aspect not considered during the planning phase.

The status of agent i is therefore a complete assignment of values to the variables in VAR^i. It is worth noting that the status of an agent evolves over time according to the actions it executes. Thus, an agent status is a snapshot of the agent taken at a given step of the plan execution. As we will see, the monitoring task has to consider consecutive agent states, so it is convenient to have a copy of the variables in VAR^i labeled with the time instant they refer to. In the rest of the paper, we denote as VAR^i_t the copies of the status variables encoding the status of agent i at time t.

5.2. Extended Action Models

The main purpose of an extended action model is to describe how an action can deviate from its expected behavior when a subset of agent's functionalities is not operating under the nominal mode; we propose to represent these extended action models as relations:

Definition 1: The extended model of action a^i_l is the tuple:
$\langle AVAR(a^i_l), PRE(a^i_l), EFF(a^i_l), \Delta(a^i_l)\rangle$, where:

- $AVAR(a^i_l) = \{v_1, \ldots, v_m\} \subseteq VAR^i$ is the subset of *active* status variables; i.e., they are relevant for capturing all the changes which may occur in the status of agent i during the execution of action a^i_l.
- $PRE(a^i_l) \subseteq dom(v_1) \times \ldots \times dom(v_m)$, is the set of nominal preconditions of action a^i_l.
- $EFF(a^i_l) \subseteq dom(v_1) \times \ldots \times dom(v_m)$, is the set of nominal effects of a^i_l.
- $\Delta(a^i_l) \subseteq PRE(a^i_l) \times EFF(a^i_l)$ is the transition relation binding preconditions to effects.

All the *passive* variables in $VAR^i \setminus AVAR(a^i_l)$ are assumed to be persistent during the execution of action a^i_l.

Note that, since action a^i_l is executed at a given time t, the transition relation $\Delta(a^i_l)$ models the changes in the agent status from time t (when the action starts) to time t+1 (when the action ends). Hereafter $a^i_l(t)$ will denote that agent i started the execution of a^i_l at time t; for readability, we will omit the time whenever it is not strictly required.

TABLE 1. The extended model of action a^1_2:`go(Rep,Desk1)` (a simplified version).

		active variables at time t				*active variables at time t+1*			
		pos	**loaded**	**pwr**	**engTmp**	**pos**	**loaded**	**pwr**	**engTmp**
1	*nominal*	`Rep`	*empty*	*ok*	*ok*	`Desk1`	*empty*	*ok*	*ok*
2	*nominal*	`Rep`	*obj*	*ok*	*ok*	`Desk1`	*obj*	*ok*	*ok*
3	*degraded*	`Rep`	*empty*	*low*	*ok*	`Desk1`	*empty*	*low*	*ok*
4	*degraded*	`Rep`	*empty*	*ok*	*hot*	`Desk1`	*empty*	*ok*	*hot*
5	*faulty*	`Rep`	*obj*	*low*	*ok*	`Rep`	*obj*	*low*	*ok*
6	*faulty*	`Rep`	*obj*	*ok*	*hot*	`Rep`	*obj*	*ok*	*hot*
7	*faulty*	`Rep`	*obj*	*ok*	*ok*	`Rep`	*obj*	*low*	*ok*
8	*faulty*	`Rep`	*obj*	*ok*	*ok*	`Desk1`	*obj*	*low*	*ok*

Since not all the agent's functionalities are typically required during the execution of action a_l^i, we highlight the subset of relevant functionalities through the following subset:

$$healthVar(a_l^i) = HLT^i \cap AVAR(a_l^i)$$

the functionalities included in this set are essential for the successful completion of action a_l^i; in fact:

Property 1: The outcome of action $a_l^i(t)$ is *succeeded* if and only if for each variable $v \in healthVar(a_l^i)$, v assumes the value *ok* at time t.

Namely, when all the functionalities $healthVar(a_l^i)$ are nominal, action a_l^i behaves deterministically and its outcome is *succeeded*; whereas when at least one functionality is not nominal, the action behaves non-deterministically and its outcome might be *failed*.

Running example. Let us consider the `go` action a_2^1 of our running example. The template for the nominal model of such an action is given in propositional terms in Figure 1; the corresponding extended action model is sketched in Table 1 and it is represented as a relation; i.e., as a set of tuples. The active variables mentioned in the extended model are: *pos* (i.e., position), *loaded*, *pwr* (i.e., the power level of the battery) and *engTmp* (i.e., the temperature of the engine); these two last variables are health status variables, and hence there are included in $healthVar(a_2^1)$.

Each row of the extended model represents a state transition relating the active variables at time t to the same active variables at time t+1. The first two of them are nominal transition as they model the expected effect of the `go` action when the agent is empty or is loaded, respectively. The third and fourth rows model degraded conditions; that is, even though the agent is somehow impaired, it can move from `Rep` to `Desk1` when it is empty. However, in the same healthy conditions, the agent cannot move when it is loaded with an object. The last four rows represent faulty transitions; in particular, rows 7 and 8 show how the same fault in the battery (a drop from *ok* to *low* in the power level), may have non deterministic impacts on the `go` action: the agent cannot even leave `Rep` (row 7); the agent moves to `Desk1` as expected (row 8), but after this step it will not be able to move anymore.

Note that the model in the table is very partial; as we will discuss in the experimental results section, the action models actually used during our tests may include more than 100 state transitions, and mention more than 20 active variables.

5.3. Fault Detection

Agent belief state. Requirement 1 guarantees that after the execution of an action at time t, agent i receives at time t+1 a set of observations - denoted as $obs^i(t+1)$ - that conveys pieces of information about the action's effects. Although we assume that observations are correct, they are in general incomplete: an agent can just directly observe the status of its available resources, and the value of a *subset* of variables in END^i (i.e., not all the variables in END^i are observed at each time). Whereas variables in HLT^i cannot be observed and their actual value can only be inferred. As a consequence, the monitoring task cannot be so accurate to precisely estimate the current status of agent i. In general, the monitoring is able to infer a *belief state*; i.e., a *set* of alternative agent's states which are all consistent with the received observations and hence are all possible. In the remainder of the paper, $\mathcal{B}^i(t)$ will refer to the belief of agent i inferred at time t.

Estimating the agent belief state. The estimation process aims at predicting the status (i.e., the belief state) of agent i at time $t+1$ after the execution of action $a_l^i(t)$; it is formalized in terms of the Relational Algebra operators as follows (see the Appendix for a short introduction about the Relational Algebra operators).

Definition 2: Let $\mathcal{B}^i(t)$ be the belief state of agent i, and let $\Delta(a_l^i(t))$ be the transition relation of action a_l^i executed at time t; the agent belief state at time $t+1$ results from:

$$\mathcal{B}^i(t+1) = \text{PROJECTION}_{VAR_{t+1}^i}(\ \text{SELECTION}_{obs^i(t+1)}\ (\mathcal{B}^i(t)\ \ \text{JOIN}\ \Delta(a_l^i(t))))$$

Let us examine this expression in detail. The first step for estimating the new belief state is the *join* operation between $\mathcal{B}^i(t)$ and $\Delta(a_l^i(t))$; this is the predictive step by means of which all the possible

agent states at time $t+1$ are estimated. More precisely, each tuple in $\Delta(a_l^i(t))$ can be thought of as having the form $\langle activeVariables_t, activeVariables_{t+1}\rangle$, describing the transition from the active variables at time t to the active variables at time t+1. The (natural) join operator compares $activeVariables_t$ to the current belief state $\mathcal{B}_t^i$, and if $\mathcal{B}_t^i \vdash activeVariables_t$, the tuple is included within the join result and brings the assignments in $activeVariables_{t+1}$ as an estimation of the next agent status. Since all the variables which are not active are assumed persistent, the result of the join operation is a new relation having the form $\langle VAR_t^i, VAR_{t+1}^i\rangle$; that is to say, a relation mapping an agent state at time t (i.e., belonging to $\mathcal{B}_t^i$) with a possible agent state at time t+1 (built by means of the extended action model). Of course, such a relation might contain spurious estimates; the selection over $obs^i(t+1)$ is therefore used to refine the result of the join operation by pruning off all the estimates which are inconsistent with the agent's observations. Finally, the belief state $\mathcal{B}^i(t+1)$ results from the projection over the status variables of agent i at time $t+1$.

Under the assumption that the action model $\Delta(a_l^i(t))$ is correct and complete the following property holds.

Property 2: Let $\mathcal{B}^i(t)$ be the belief state of agent i at time t, and let $\hat{s} \in \mathcal{B}^i(t)$ be the actual status of the agent i at time t; given the extended action model $\Delta(a_l^i(t))$, the agent belief state $\mathcal{B}^i(t+1)$ inferred according to Definition 2 always includes the actual state $\hat{s}'$ of agent i at time t+1.

Proof. By contradiction, let us assume that the state $\hat{s}'$ does not belong to the belief state $\mathcal{B}^i(t+1)$; this might happen for one of the following reasons:

(1) a state transition $\langle \hat{s}, \hat{s}'\rangle$ is missing in $\Delta(a_l^i(t))$, but this is against the assumption of correctness and completeness of the action model;

(2) the state transition $\langle \hat{s}, \hat{s}'\rangle$ is included in $\Delta(a_l^i(t))$, but the actual state $\hat{s}$ is not included in $\mathcal{B}^i(t)$, but this is against the premises of the property;

(3) the state $\hat{s}'$ is correctly estimated via the join operator, but it is filtered out during the selection operation which prunes off all the agent states inconsistent with the observations. But this is against the assumption that the available observations, though incomplete, are correct.

□

Property 2 assures that the monitoring is correct as the actual status of an agent is always traced during the plan execution.

Inference of an action outcome. To infer the outcome of an action we adopt a conservative approach asserting that an action has outcome *succeeded* when the following condition holds:

Definition 3: Given the belief $\mathcal{B}^i(t+1)$, the outcome of action $a_l^i(t)$ is *succeeded* if and only if $\forall s \in \mathcal{B}^i(t+1), s \vdash \mathit{eff}(a_l^i)$.

This condition states that action a_l^i is successfully completed when the nominal effects of the action hold in every state s belonging to the belief state inferred just after the execution of the action. This means that we adopt a rather pessimistic approach, in fact it is sufficient that the action effects are not satisfied in just one state in $\mathcal{B}^i(t+1)$ to conclude that action $a_l^i(t)$ has outcome *failed*. [4]

The Monitoring Task After these premises, the monitoring task which is part of the control loop introduced in Section 4 is summarized in the pseudo-code of Figure 5. The first step consists in the estimation of the next belief state $\mathcal{B}^i(t+1)$ according to Definition 2 (line `00`). To determine whether the action outcome is *succeeded* or not, we build a temporary relation $Temp$ as the join between $\mathcal{B}^i(t+1)$ and the nominal action effects $\mathit{eff}(a_l^i)$ (line `01`); $Temp$ will keep the same states as $\mathcal{B}^i(t+1)$ only if the nominal effects of a_l^i are satisfied in every state of $\mathcal{B}^i(t+1)$; therefore, when $Temp$ is a proper subset of $\mathcal{B}^i(t+1)$ we can conclude that the action outcome is *failed* (line `02`); otherwise, the

[4] Definition 3 can be relaxed along the line discussed in (Micalizio and Torasso, 2008): whenever it is not possible to certainly assert the successful or unsuccessful completion of an action, the action outcome remains *pending*; observations received in the future will be used for discriminating between *succeeded* and *failed*.

Monitoring(a^i, t, $obs^i(t+1)$)
00 $\mathcal{B}^i(t+1) \leftarrow$ PROJECTION$_{VAR^i_{t+1}}$(SELECTION$_{obs^i(t+1)}$ ($\mathcal{B}^i(t)$ JOIN $\Delta(a^i_l(t))$))
01 $Temp \leftarrow \mathcal{B}^i(t+1)$JOIN $eff(a^i_l)$
02 **if** $Temp \subset \mathcal{B}^i(t+1)$ **return** *outcome failed*
03 **else return** *outcome succeeded*

FIGURE 5. The Monitoring task.

action outcome is *succeeded* (line 03).
See the Appendix for an analysis of the computational complexity of the monitoring task implemented by means of the Ordered Binary Decision Diagrams (OBDDs).
Running example. Let us assume that agent A1 has successfully completed action a^1_1, so it is now located into the repository Rep and loaded with Pack1. The agent then executes the subsequent go action to move from Rep to Desk1. To monitor this action, the agent exploits the model in Table 1, where the entries 2 and 5 through 8 play an active role in estimating the possible next status of the agent. After the execution of the go action, the agent observes its current position and discovers that it is located in Rep (*pos*=Rep), this observation is used to refine its belief state and the result is showed in Table 2. It is easy to see that such a belief models an erroneous situation as the nominal expected effect of action a^1_2 is *pos*=Desk1, which is not satisfied in any state included into the belief. Thus the agent concludes that the go action has failed.

6. AGENT DIAGNOSIS

In this section we describe the model-based approach adopted for diagnosing action failures; before that, however, we recall some basic definition about Model-Based Diagnosis (MBD). Intuitively, MBD can be viewed as an interpretation process that, given a model of the system under consideration and a set of observations about the system behavior, provides an indication of the presence or absence of faults in the components of the system itself.
One of the first formal (logic-based) theories of diagnosis is the *consistency-based diagnosis* proposed by Reiter (Reiter, 1987). In a consistency-based setting, the system to be diagnosed is described as a triple $\langle SD, COMPS, O\rangle$ where:

- SD (*system description*) denotes a finite set of formulae in first-order predicate logic, specifying only the system *normal* structure and behavior;
- $COMPS$ (*components*) is the set of system components; each component $c \in COMPS$ can be qualified as behaving either abnormally $ab(c)$, or nominally $\neg ab(c)$;
- OBS is a finite set of logic formulae denoting the *observations*.

We have a diagnostic problem when the hypothesis that all the components in $COMPS$ behave nominally ($H_{NOM} = \bigcup_{c\in COMPS}\{\neg ab(c)\}$) is inconsistent with the system description and the available observations. In other words, when $SD \cup OBS \cup H_{NOM} \models \bot$ we have detected an anomalous behavior of the system. Solving a diagnostic task requires to find a subset $D \subseteq COMPS$ of components that, when qualified as abnormal, make the new hypothesis $H=\{ab(c)|c \in D\} \cup \{\neg ab(c)|c \in COMPS \setminus D\}$ consistent with the system description and the observations: $SD \cup H \cup OBS \not\models \bot$. The subset D is therefore a diagnosis since it identifies a subset of components whose anomalous behavior is consistent with the observations.

Console and Torasso (Console *et al.*, 1991; Console and Torasso, 1991) have introduced the notion of *abductive diagnosis* by including within the system description not only the nominal states of the system but also its abnormal states and the corresponding abnormal observations. Having

TABLE 2. The agent belief state at the time of the failure of action a^1_2.

	pos	**loaded**	**pwr**	**engTmp**	**hnd**
i_1	Rep	Pack1	*ok*	*high*	*ok*
i_2	Rep	Pack1	*low*	*ok*	*ok*

such a kind of system description, the diagnostic inference aims at finding a subset of components D such that:
(1) $SD \cup H \models OBS$ and
(2) $SD \cup H$ is consistent.
Where H is again the hypothesis $\{ab(c)|c \in D\} \cup \{\neg ab(c)|c \in COMPS \setminus D\}$. This means that in the abductive diagnosis we are interested in identifying a subset of components whose anomalous behavior is not only consistent with the observations, but *explains* them. Moreover, having explicit fault models of the system components, it is also possible to identify the specific abnormal behaviors affecting the impaired components.

Diagnosing action failures We show now how our framework matches to the MBD concepts previously introduced. Since we are interested in detecting and diagnosing action failures as soon as they occur, the model of an action corresponds to a portion of the system description to be used during the diagnostic inferences. Moreover, since we aim at recovering from action failures by fixing faults, we are interested in an action model which allows us to infer an abductive diagnosis: we do not want just to know which components (i.e., agent's functionalities) are faulty, but also in which way.

The extended action model introduced in the previous section satisfies this requirement. In particular, Property 1 states that the outcome of an action is *succeeded* if and only if all the functionalities in *healthVar* behave nominally; otherwise, the action fails. This means that the agent diagnosis must explain the failure of an action by singling out a functionality (or a set of functionalities) which cannot be assumed to be nominal.

More formally, in our framework a diagnostic problem is the tuple

$$\langle \Delta(a_l^i(t)),\ healthVar(a_l^i), \mathcal{B}^i(t), obs^i(t+1) \rangle$$

where:
- $\Delta(a_l^i(t))$ is that portion of system description relevant for the diagnosis of action a_l^i,
- $healthVar(a_l^i)$ is the set of agent's functionalities which can be qualified as faulty,
- $\mathcal{B}^i(t)$ represents a piece of contextual information about the current status of agent i,
- $obs^i(t+1)$ is the set of available observations.
Solving such a diagnostic problem means finding a hypothesis $H^i(t+1)$ (i.e., an assignment of values to the variables in $healthVar(a_l^i)$ at time $t+1$), such that

$$\mathcal{B}^i(t) \ \widetilde{\cup}\ \Delta(a_l^i(t)) \ \widetilde{\cup}\ H^i(t+1) \vdash obs^i(t+1) \tag{1}$$

Where the symbol $\widetilde{\cup}$ means *combined with*. The agent diagnosis is obviously a subset D^i of variables in $healthVar(a_l^i)$ such that

$$H^i(t+1)=\{v \in D^i | v \neq ok\} \cup \{v \in healthVar(a_l^i) \setminus D^i | v = ok\}$$

Namely, D^i singles out a subset of functionalities which might have behaved erroneously during the execution of action a_l^i.

Since we are dealing with relations, symbol $\widetilde{\cup}$ in expression (1) corresponds to a join operator between relations; moreover, the entailment ($\vdash$) can be matched to the selection operator of the Relational Algebra; expression (1) thus becomes:

$$\text{SELECTION}_{obs^i(t+1)}(\mathcal{B}^i(t) \text{JOIN } \Delta(a_l^i(t)) \text{JOIN } H^i(t+1)) \tag{2}$$

that we have already met, though in a slightly different form, in Definition 2 where we have described the process for estimating the next belief state $\mathcal{B}^i(t+1)$, which represents a synthesis of the estimation process. This means that for inferring the agent diagnosis we have just to extract the hypothesis $H^i(t+1)$ from $\mathcal{B}^i(t+1)$:

Definition 4: Let $a_l^i(t)$ be an action whose outcome *failed* is detected within the belief state $\mathcal{B}^i(t+1)$, the qualification hypothesis explaining such a failure is

$$H^i(t+1)=\text{PROJECTION}_{healthVar(a_l^i)}\mathcal{B}^i(t+1)$$

From which the agent diagnosis results as $D^i=\{v \in healthVar(a_l^i) | v \neq ok \text{ in } H^i(t+1)\}$

Property 3: The agent diagnosis D^i inferred according Definition 4 is correct as it always includes the actual explanation for the action failure.

Proof. The proof directly follows from Property 2, which guarantees that the actual state $\hat{s}'$ of agent i at time $t+1$ is included in $\mathcal{B}^i(t+1)$. Since the agent diagnosis is inferred by projecting the belief status $\mathcal{B}^i(t+1)$ over the health status variables of agent i, it follows that D^i includes, among others, the exact (anomalous) health status of agent i. □

Running example. Given the failure of the `go` action a_2^1, agent `A1` infers a qualification hypothesis by projecting the belief state in Table 2 over the variables in *healthVar(`go`)*, namely, the variables *pwr* and *engTmp*. Consequently, ambiguous agent diagnosis is the disjunction $D^{\texttt{A1}} = \{pwr = low \vee engTmp = hot\}$; according to the extended action model in Table 1, in fact, it is sufficient that either the battery (*pwr*) or the engine (*engTmp*) is in an anomalous mode, to prevent the agent from moving when it is loaded with an object.

7. RECOVERING FROM ACTION FAILURES: THE MAIN STRATEGY

In this section we introduce the strategy we propose for recovering from an action failure. The basic idea of this strategy, sketched in Figure 6, is that the agent first tries to self-repair its impaired functionalities and then it resumes the plan execution from the same action where it was stopped. Of course, the recovery builds up the results of the monitoring and diagnosis activities, so it takes in input the failed action a_l^i, the last inferred belief state $\mathcal{B}^i(t+1)$, and the corresponding agent diagnosis; the output consists of a recovery plan that brings agent i from its erroneous current situation back to a nominal condition; when the recovery plan does not exist, an empty plan is returned.

The strategy starts by determining the "healthy" agent state $\mathcal{H}$, which represents the desired situation where all the functionalities assumed to be faulty by the agent diagnosis have been fixed (line `00`). Thus, in line `01`, the recovered state $\mathcal{R}$ is synthesized as the conjunction of $\mathcal{H}$ and the nominal preconditions of the failed action a_l^i; $\mathcal{R}$ represents the situation from which the plan execution can be resumed by trying again action a_l^i. The subsequent step consists in the synthesis of a plan Pr^i reaching the state $\mathcal{R}$. When Pr^i exists, the plan repair strategy returns a recovery plan consisting of two parts: first the plan segment Pr^i which fixes the faulty functionalities, and then the original plan segment $[a_l^i, \ldots, a_\infty^i]$ (line `04`).

On the other hand, when it is not possible to repair all the malfunctioning functionalities, the agent cannot complete the plan segment $[a_l^i, \ldots, a_\infty^i]$ which is aborted. In this case, the strategy tries to bring agent i into a *safe status* $\mathcal{S}$; i.e., a situation where the agent does not represent an obstacle for the other teammates. Therefore, even though agent i cannot completely recover from its failure, it can try to limit the impact of its failure in the whole system. If a plan Ps^i to the safe status exists, Ps^i is returned to the control loop which is in charge of executing it (lines `06-09`).

Finally, when also the plan to safe status does not exist, the repair strategy returns an empty plan; in this case the agent interrupts the execution of its local plan avoiding further damages.

In the rest of this section we formalize the two planning problems that have to be solved in order to achieve either the repaired status $\mathcal{R}$ or the safe status $\mathcal{S}$. Next section will go into details of the conformant planner we use to solve these planning problems and explain why we need such a kind of planner.

7.1. Plan to $\mathcal{R}$

The repaired status $\mathcal{R}$ is the conjunction of three conditions:
1) all the functionalities mentioned into D^i have been fixed;
2) the preconditions of action a_l^i are satisfied;
3) for any action in the segment $[a_{l+1}^i, \ldots, a_\infty^i]$, no open preconditions are left.

The first condition removes the causes of the failure of a_l^i, while the second and the third ones assure that the plan segment $[a_l^i, \ldots, a_\infty^i]$ is executable.
A plan Pr^i reaching $\mathcal{R}$ is called *recovery plan* and can be found by resolving the following planning problem:

Recovery$(P^i, a_l^i, \mathcal{B}^i(t+1), D^i)$
00 $\mathcal{H} \leftarrow$ **inferHealthyStatus**$(a_l^i,\ D^i)$
01 $\mathcal{R} \leftarrow \mathcal{H} \wedge pre(a_l^i)$
02 $Pr^i \leftarrow \langle$PLAN TO $\mathcal{R}\rangle$
03 **if** Pr^i is not empty
04 **return** $P^{*i} \leftarrow$ **concatenate**$(Pr^i, [a_l^i, \ldots, a_\infty^i])$
05 **else**
06 $\mathcal{S} \leftarrow$ **inferSafeStatus**$(a_l^i,\ AvRes(i,t))$
07 $Ps^i \leftarrow \langle$PLAN TO $\mathcal{S}\rangle$
08 **if** Ps^i is not empty
09 **return** Ps^i
10 **else return** $\emptyset$

FIGURE 6. The plan repair strategy.

Definition 5: The plan Pr^i=$[ar_0^i, \ldots, ar_\infty^i]$ achieving $\mathcal{R}$ is a solution of the planning problem $\langle \mathcal{I}, \mathcal{F}, \mathcal{A}\rangle$; where:

- $\mathcal{I}$ (initial state) corresponds to the agent belief state $\mathcal{B}^i(t+1)$ inferred at the time of the failure of action $a_l^i(t)$;

- $\mathcal{F}$ (final state) coincides with $\mathcal{R}$=$(\bigwedge_{\forall v \in D^i} v = ok) \wedge pre(a_l^i) \wedge grantedServices(l)$;

- $\mathcal{A}$ = is the set of action models which can be used during the planning process.

Where $grantedServices(l)$= $\{q | q \in eff(a_k), k < l, q \in pre(a_h), h > l\}$ are all those services achieved by actions preceding a_l^i and consumed by actions following a_l^i; these services must be included in $\mathcal{R}$ to avoid they are removed by some collateral effects of the recovery plan.

Note that $\mathcal{A}$ contains all the actions the agent can perform, including actions which restore the nominal behavioral mode in the agent's functionality. For example, a low charge in the battery (*pwr=low*) is fixed by means of a `recharge` action whose effect is *pwr=ok*; similarly, a high temperature in the engine ($engTmp = hot$) is mitigated by a `refill` of a coolant fluid whose effect is *engTmp=ok*. In principle, these repairing actions could also be part of the original plan; for instance, a recharge action could be planned after a number of actions so as to prevent the agent running out of power. It is important to note, however, that some faults cannot be autonomously repaired by the agent; for example, there is no repair action that can fix a blocked arm (*hnd=blocked*); these faults can be repaired only by human intervention.

As mentioned earlier, when the plan Pr^i=$[ar_0^i, \ldots, ar_\infty^i]$ exists, the agent i yields its new local plan $P^{*i} = [ar_0^i, \ldots, ar_\infty^i] \circ [a_l^i, \ldots, a_\infty^i]$; where $\circ$ denotes the concatenation between two plans (i.e., the second plan can be executed just after the last action of the first plan has been completed).

Property 4: The plan P^{*i} is feasible and executable.

Proof. Both plan segments $[ar_0^i, \ldots, ar_\infty^i]$ and $[a_l^i, \ldots, a_\infty^i]$ are feasible and executable on their own as each of them has been produced by a specific planning step. Moreover, by construction, ar_∞^i corresponds to a state where both the preconditions a_l^i and the $grantedServices(l)$ are satisfied. It follows that the whole plan P^{*i} is feasible and executable. □

7.2. Plan to $\mathcal{S}$

The plan Pr^i may not exist as agent i could be unable to autonomously repair its faults. The impaired agent, however, should not be left in the environment as it could become a latent menace for the other team members; for instance, the agent could lock indefinitely critical resources preventing others from acquiring them. For this reason, the repair strategy tries to move agent i into a safe status $\mathcal{S}$ in which i releases all its resources. Also this step is modeled as a planning problem:

Definition 6: The plan Ps^i=$[as_0^i, \ldots, as_\infty^i]$ achieving $\mathcal{S}$ is a solution of the plan problem $\langle \mathcal{I}, \mathcal{F}, \mathcal{A}\rangle$; where:

- $\mathcal{I}$ (initial state) corresponds to the agent belief state $\mathcal{B}^i(t+1)$ (as in the previous case);
- $\mathcal{F}$ (final state) is the safe status $\mathcal{S} = \bigwedge_{\forall res_k \in AvRes(i,t)} res_{k,i} = \textit{not-in-use}$;

- $\mathcal{A}$ is the set of action models which can be used during the planning process.

Since it is the result of a single planning phase, if the plan Ps^i exists, then it is also feasible and executable. When plan Ps^i exists, it becomes the new local plan assigned to agent i; that is $P^{*i} = Ps^i$, and all the actions in the segment $[a^i_l, \ldots, a^i_\infty]$ are aborted.

8. A CONFORMANT PLANNER

The recovery strategy introduced in the previous section strongly relies on a planner in order to synthesize either Pr^i or Ps^i; the recovery strategy, however, must satisfy the following demanding requirements.

Requirement 2: *Locality.* The recovery strategy can only impose local changes in P^i: no new resources can be acquired for achieving either $\mathcal{R}$ or $\mathcal{S}$.

The acquisition of a new resource would require an explicit synchronization with other agents and hence the impact of the recovery strategy would affect, besides P^i, also the local plans of some other agents. This first requirement imposes that the planner can just exploit the resources $AvRes(i,t)$, already acquired by agent i at the time of the failure.

Requirement 3: *Conformant Plan.* Since the belief state $\mathcal{B}^i(t+1)$ is in general ambiguous (the actual health status of the agent is not precisely known), the planning phase must produce a *conformant* plan.

This requirement assures that when the repairing plan Pr^i exists, it is also executable, namely, the agent can carry out that plan without the risk of getting stuck during the execution of the recovery plan.

In the remainder of the paper, we will show how these two requirements shape the planner used for the recovery purpose.

8.1. Refining action models

In order to integrate the planning phase within the control loop previously introduced, we formalize a conformant planning algorithm based on the same Relational language we have so far used to formalize the monitoring and the diagnostic activities. In other words, we want to use the same action models envisaged for the monitoring purpose. These models, however, are too rich for the planning purpose: an extended action model $\Delta(a)$ describes all the evolutions of action a taking into account the possible occurrence of faults during its execution. From the repair point of view, however, it is sufficient to restrict the action model $\Delta(a)$ to those transitions which are consistent either with the agent diagnosis D^i or with the nominal health status of the agent. In fact, on the one hand action a could be performed under the unhealthy status assumed by D^i. On the other hand, the same action could be performed when the faulty functionalities have been fixed and the nominal status has been restored.

Given an extended action model $\Delta(a)$, the corresponding refined model $\widehat{\Delta}(a)$ restricted by the agent diagnosis D^i is:

Definition 7:
$$\widehat{\Delta}(a) = \begin{cases} \{\Delta(a) \text{ JOIN } pre(a)\} \cup \{\Delta(a) \text{ JOIN } D^i\} & \text{if } \{\Delta(a) \text{ JOIN } D^i\} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$\widehat{\Delta}(a)$ is a subset of $\Delta(a)$ and it is defined only when action a can be performed given the agent diagnosis D^i, in such a case it includes all the state transitions which are either nominal or consistent with D^i. Otherwise, the refined model is empty.

TABLE 3. The model of action a_2^1:`go(Rep,Desk1)` refined for the repair purpose.

		active variables at time t				*active variables at time t+1*			
		pos	**loaded**	**pwr**	**engTmp**	**pos**	**loaded**	**pwr**	**engTmp**
1	*nominal*	`Rep`	*empty*	*ok*	*ok*	`Desk1`	*empty*	*ok*	*ok*
2	*nominal*	`Rep`	*obj*	*ok*	*ok*	`Desk1`	*obj*	*ok*	*ok*
3	*degraded*	`Rep`	*empty*	*low*	*ok*	`Desk1`	*empty*	*low*	*ok*
4	*degraded*	`Rep`	*empty*	*ok*	*hot*	`Desk1`	*empty*	*ok*	*hot*

An example of refined action model is given in Table 3 and refines the model of the `go` action previously sketched in Table 1; it is easy to see that the refined model only contains the nominal state transitions and the degraded state transitions which are consistent with the agent diagnosis $D^{\mathtt{A1}} = \{pwr{=}low \vee engTmp{=}high\}$.

8.2. Conformant Planning: Preliminaries

The conformant planning algorithm we propose is based on the same predictive mechanism we have already used during the monitoring phase, and hence on the notion of agent belief state. In the following discussion, however, we consider the synthesis of a plan rather than the plan execution; thus, when we mention an agent belief state, we will not intend a status estimation made after the actual execution of an action, but an estimation made after the application (i.e., execution hypothesis) of an action to a given state.
After this premise, we define a conformant action as follows:

Definition 8: An action a is *conformant* w.r.t. an agent belief state $\mathcal{B}$ iff its (refined) model $\widehat{\Delta}(a)$ is applicable in every state $s \in \mathcal{I}$

Applying an action a to an agent belief $\mathcal{B}$ yields the relation:

$$\mathcal{P} = \mathcal{B} \text{ JOIN } \widehat{\Delta}(a) \tag{3}$$

$\mathcal{P}$ is a set of state transitions of the form $\langle s, s' \rangle$ where s and s' are agent states before and after the application of a, respectively. The action a is conformant when each state $s \in \mathcal{B}$ matches with at least a transition in $\widehat{\Delta}(a)$. Of course, when $\mathcal{P}$ is empty, action a is not applicable in $\mathcal{B}$. In all the intermediate conditions, where $\mathcal{P}$ is not empty but some states in $\mathcal{B}$ do not participate to the join, the action is applicable but it is not conformant.

Definition 9: Given a plan candidate $\pi{=}a_0, \ldots, a_h$, and an initial agent belief state $\mathcal{I} = \mathcal{B}_0$, the application of π to $\mathcal{B}_0$ is the relation $\mathcal{P}_{h+1} = (((\mathcal{B}_0 \text{ JOIN } \widehat{\Delta}(a_0)) \text{ JOIN } \widehat{\Delta}(a_1)) \ldots) \text{ JOIN } \widehat{\Delta}(a_h)$

$\mathcal{P}_{h+1}$ is a set of trajectories, each of which has the form $\langle s_0, s_1, \ldots, s_{h+1} \rangle$ where, except for s_0 which is an agent state in $\mathcal{B}_0$, each agent state s_{k+1} $(k : 0..h)$ results by applying action a_k to state s_k.
The result of π is the agent belief state after the application of π to $\mathcal{B}_0$:

$$\mathcal{B}_{h+1} = \text{PROJECTION}_{h+1} \mathcal{P}_{h+1}{}^{5}$$

For short, we will denote as $\pi(\mathcal{B}_0)$ the agent belief state obtained by applying π to $\mathcal{B}_0$.

In the previous section we have said that the repair strategy has to solve the planning problem $\langle \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle$, where the initial state $\mathcal{I}$ coincides with the agent belief $\mathcal{B}^i(t+1)$ upon which the failure of action a_l^i has been detected; the final state $\mathcal{F}$ is either $\mathcal{R}$ or $\mathcal{S}$; and $\mathcal{A}$ is the set of action models to be used.

Definition 10: The plan candidate $\pi{=}\, a_0, \ldots, a_h$ is a *conformant* solution for the planning problem $\langle \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle$ iff:

[5]To simplify the notation, the predicate $h{+}1$ of the projection operator stands for VAR_{h+1}^i; i.e., the agent status variables at the $(h{+}1)$-th step.

- $\pi(\mathcal{I}) \vdash \mathcal{F}$; i.e., the goal $\mathcal{F}$ is satisfied in each state $s \in \pi(\mathcal{I})$,
  and
- $a_0, \ldots, a_h$ are all *conformant* action instances; i.e., each action a_k is conformant w.r.t. the (intermediate) agent belief state $\mathcal{B}_k$ extracted from $\mathcal{P}_k$ $(k : 0..h)$.

To verify whether an intermediate action a_k is conformant, it is sufficient to assess the following property.

Property 5: a_k is conformant iff $\mathcal{B}_k \cap \overline{\mathcal{B}_k^{ext}} = \emptyset$.

Where $\mathcal{B}_k = \text{PROJECTION}_k \mathcal{P}_k$, and $\mathcal{B}_k^{ext} = \text{PROJECTION}_k \mathcal{P}_{k+1}$; namely, $\mathcal{B}_k^{ext}$ is the agent belief state at the k-th plan step (as $\mathcal{B}_k$), but extracted from $\mathcal{P}_{k+1}$ after the application of action a_k.

Proof. By definition, action a_k is conformant iff it is applicable in every state $s \in \mathcal{B}_k$. If a_k is not conformant, there must exist at least one state $s \in \mathcal{B}_k$ where a_k is not applicable; as a consequence when the corresponding model $\widehat{\Delta}(a_k)$ is joined with $\mathcal{B}_k$, the state s does not participate to build the relation $\mathcal{P}_{k+1}$. Namely, the state s is missing in $\mathcal{B}_k^{ext}$. This means that s belongs to the dual set $\overline{\mathcal{B}_k^{ext}}$, and hence when a_k is not conformant, $\mathcal{B}_k \cap \overline{\mathcal{B}_k^{ext}}$ cannot be empty.

Now, let us assume that $\mathcal{B}_k \cap \overline{\mathcal{B}_k^{ext}} = \emptyset$ holds and conclude that a_k must be conformant. The intersection between $\mathcal{B}_k$ and $\overline{\mathcal{B}_k^{ext}}$ can be empty only when $\mathcal{B}_k$ equals $\mathcal{B}_k^{ext}$; this means that the application of the refined model $\widehat{\Delta}(a_k)$ to $\mathcal{B}_k$ preserved in P_{k+1} each state $s \in \mathcal{B}_k$. It follows that $\widehat{\Delta}(a_k)$ has been applied in each state included in $\mathcal{B}_k$, and hence a_k is conformant. □

8.3. Search for a conformant solution

The conformant planning algorithm we propose adopts a forward-chaining approach that from the initial state $\mathcal{I}$ finds a plan reaching the goal state $\mathcal{F}$. More precisely, the algorithm realizes a breadth-first strategy which carries on all the plan candidates built at a given step.

To formalize this strategy we introduce the macro-operator Φ, defined as follows:

Definition 11: $\Phi = \bigcup_{a \in \mathcal{A}} \{\widehat{\Delta}(a)$ such that a is executable given the resources in $AvRes(i,t)\}$.

In other words, Φ is a set of refined models which just includes the actions that agent i can perform by exploiting the resources it already holds.

Our basic idea is to use Φ as a means for pruning the space of plan candidates looking for plans whose actions are all included in Φ; in fact, the following property holds

Property 6: Given a planning problem $\Pi : \langle \mathcal{I}, \mathcal{F}, \mathcal{A} \rangle$, any conformant solution π for Π that satisfies the locality requirement can only consist of actions in Φ.

Proof. By definition we have that: 1) Φ includes refined action models which are consistent with the agent diagnosis, and this is a prerequisite for an action to be conformant; 2) Φ includes only the actions that are executable given the set of resources the agent already holds.
Let us assume that there exists a conformant plan π including at least one action a whose model is not in Φ. If π is conformant, action a must be executable given the agent diagnosis $\mathcal{D}^i$ (i.e., $\widehat{\Delta}(a) \neq \emptyset$), and hence it has not been included into Φ as it uses resources outside $AvRes(i,t)$, it follows that π does not satisfies the locality requirement. On the other hand, if a just exploits resources in $AvRes(i,t)$, a has been discarded from Φ as its refined model $\widehat{\Delta}(a)$ is empty, but this implies that π is not conformant against the initial hypothesis. □

Relying on the previous property, and since $\mathcal{I}$ equals the initial belief state $\mathcal{B}_0$, we can generalize expression (3) as follows

$$\mathcal{P}SET_0 = \mathcal{B}_0 \text{ JOIN } \Phi \tag{4}$$

$\mathcal{P}SET_0$ is a set of $\mathcal{P}$ relations and represents the set of all the plan candidates consisting of one single action. (Of course, only those actions which are applicable in $\mathcal{B}_0$ are actually part of the set

ConformantPlan$(\mathcal{I}, \mathcal{F}, \mathcal{A}, AvRes(i,t), D^i)$
00 $\Phi \leftarrow$ **Build-**$\Phi(\mathcal{A}, AvRes(i,t), D^i)$
01 $\pi \leftarrow \emptyset$
02 $\mathcal{PSET}_0 \leftarrow \mathcal{I}$
03 $h \leftarrow 0$
04 *solved*$\leftarrow$*false*
05 **while** *not solved and* $h < MAXDEPTH$
06 $\quad \mathcal{PSET}_{h+1} \leftarrow \mathcal{PSET}_h$ JOIN Φ
07 $\quad \mathcal{PSET}_{h+1} \leftarrow$ **PruneNotConformant**$(\mathcal{PSET}_h, \mathcal{PSET}_{h+1})$
08 $\quad$ **if** $\mathcal{PSET}_{h+1}$ is *empty* **return** $\emptyset$
09 $\quad$ *solved* $\leftarrow$ **CheckGoal**$(\mathcal{PSET}_{h+1}, \mathcal{F})$
10 $\quad$ **if** *solved*=*true*
11 $\quad\quad \pi \leftarrow$**ExtractPlan**$(\mathcal{PSET}_{h+1})$
12 $\quad$ **else** $h \leftarrow h + 1$
13 **return** π

FIGURE 7. The high-level algorithm for the synthesis of a conformant plan.

$\mathcal{PSET}_0$.) We can further generalize expression (4) in order to model the space of the (conformant) plan candidates incrementally built by applying Φ in succession:

$$\mathcal{PSET}_{h+1} = \mathcal{PSET}_h \text{ JOIN } \Phi \tag{5}$$

$\mathcal{PSET}_{h+1}$ is the result of h+1 successive applications of Φ; it maintains all the plan candidates built by extending one step longer the previous set of candidates $\mathcal{PSET}_h$.

Intuitively, our planning algorithm first extends the space of the plan candidates by applying Φ, and then it verifies whether there exists a plan $\pi \in \mathcal{PSET}_{h+1}$ representing a conformant solution for the problem at hand (Definition 10). Of course, since the algorithm goes forward, we need a constant $MAXDEPTH$ which limits the depth of the search and guarantees the termination of the algorithm. In other words, $MAXDEPTH$ represents the maximum number of times that it is possible to apply Φ. If a solution is not found in less than $MAXDEPTH$ steps, the plan search terminates with a failure.

The high-level planning algorithm is showed in Figure 7; in the first five lines, some important structures are initialized: Φ is set by invoking the Build-Φ function which operates according to equation (11); the sought plan π is set to empty; $\mathcal{PSET}_0$ (the initial explored space) just includes the initial state $\mathcal{I}$; h, set to zero, counts the number of times Φ has been applied; finally, *solved* is a Boolean flag set to *false*, if a conformant plan is found, this flag will be set to *true* and the search will be stopped. After these preliminary steps, the algorithm loops until either a conformant solution has been found or the number of iterations becomes greater than $MAXDEPTH$.

At each iteration, the algorithm builds a new set of plan candidates $\mathcal{PSET}_{h+1}$ by applying Φ to the previous set $\mathcal{PSET}_h$ (line 06); $\mathcal{PSET}_{h+1}$ is therefore refined by pruning off all those plan candidates which are not conformant, function PruneNotConformant relies on Property 5 to achieve this objective (line 07). Note that, after the invocation of function PruneNotConformant, $\mathcal{PSET}_{h+1}$ could become empty, this may happen when no conformant action exists, in that case the search is stopped and an empty plan is returned (line 08).

When the space of plan candidates is not empty, function CheckGoal checks whether at least one candidate π leads to the goal $\mathcal{F}$; more precisely, π leads to a state $\mathcal{B}_{h+1}$ where the condition $\mathcal{B}_{h+1} \vdash \mathcal{F}$ is satisfied. If this is the case, the conformant plan π is extracted from $\mathcal{PSET}_{h+1}$ and the flag *solved* is turned to *true* (lines 09-11). Conversely, h is incremented and the loop is repeated.

If a solution has not been found after $MAXDEPTH$ iterations, the loop is stopped and an empty plan is returned.

Note that, since the search proceeds in a breadth-first manner and keeps all the plan candidates found at a given iteration, there is no need to backtrack. Of course, an efficient implementation of the algorithm becomes a critical issue; since the sizes of the relations might be very huge, the operations between relations might become computationally expensive. A possible way to mitigate the problem consists in the adoption of symbolic formalisms for encoding the relations in a compact form. The work by Darwiche and Marquis (2002) describes and compares with one another different method-

ologies for compiling knowledge into symbolic representations. Relying on the results presented by Darwiche and Marquis, we have adopted in our implementation the formalism of the Ordered Binary Decision Diagrams (OBDDs). The computational cost of the algorithm implemented by means of OBDD operators is discussed from an empirical point of view in Section 10; for a theoretical analysis of its computational complexity, see the Appendix.

Theorem 1: Let h be the depth of the search space; when $h < MAXDEPTH$, the conformant planning algorithm is correct and complete.

Proof. In this proof we demonstrate that:

- if it exists a plan π, having h steps, such that $\pi(\mathcal{I}) \vdash \mathcal{F}$, then:
  1) π is a solution,
  2) π belongs to $\mathcal{PSET}_h$,
  3) the algorithm finds it;
- otherwise, if $\mathcal{PSET}_h$ becomes empty, then no conformant solution exists (even with a number of steps greater than $MAXDEPTH$).

With Property 6 we have already shown that any conformant plan satisfying the locality requirement, if it exists, is a sequence of action instances in Φ. Since Φ is finite and h limited by $MAXDEPTH$, the space of plan candidates is finite too. We demonstrate that the planning algorithm carries on an exhaustive search within this space and that it does not miss solutions. We show this by induction:
Hypothesis: After h iterations $\mathcal{PSET}_h$ maintains all the conformant plans satisfying the locality requirement, each of these plan candidates has h actions.
Base Case: for $h = 1$ we have that $\mathcal{PSET}_1$, built as $\mathcal{PSET}_0$ JOIN Φ, maintains all the conformant plan candidates satisfying the locality requirement consisting of just one action. Since $\mathcal{PSET}_0$ equals $\mathcal{I}$ the basic case is trivially satisfied (see equation 4).
Inductive step: We show that, at the $(h+1)$-th iteration, the synthesis of $\mathcal{PSET}_{h+1}$ does not lose solutions and keeps all the possible conformant plans:

(1) $\mathcal{PSET}_{h+1} = \mathcal{PSET}_h$ JOIN Φ:

   (a) for each plan candidate $\pi \in \mathcal{PSET}_h$ the algorithm builds a set of new plan candidates

$$X[\pi] : \bigcup_{a \in \Phi} \{\pi' |\ \pi' = \pi \circ \langle a \rangle\}$$

   Namely, for each action $a \in \Phi$, the algorithm gets a new plan candidate π' by appending a in π;
   (b) $\mathcal{PSET}_{h+1}$ is the union of the sets of plan candidates $X[\pi]$ for each $\pi \in \mathcal{PSET}_h$.

$$\mathcal{PSET}_{h+1} = \bigcup_{\pi \in \mathcal{PSET}_h} X[\pi]$$

   Since a plan candidate with h+1 steps can only be obtained by appending an action to a plan candidate with h steps, and since $\mathcal{PSET}_h$ maintains all the (conformant) plan candidates of length h (for the inductive hypothesis), $\mathcal{PSET}_{h+1}$ contains all the possible plan candidates having h+1 steps, though some of them may not be conformant.

(2) The PruneNotConformant function prunes off from the $\mathcal{PSET}_{h+1}$ previously computed all the plan candidates π' which are not conformant; so after this step:

$$\mathcal{PSET}_{h+1} = \{\pi' = \pi \circ \langle a \rangle |\ \pi \in \mathcal{PSET}_h \text{ and } a \text{ is conformant w.r.t. } \pi(\mathcal{I})\}.$$

   Of course, $\mathcal{PSET}_{h+1}$ becomes empty when no plan $\pi \in \mathcal{PSET}_h$ can be extended with any conformant action, and hence no plan candidates with h+1 steps can be built. Otherwise, when $\mathcal{PSET}_{h+1}$ is not empty, it maintains *all and only* the conformant plans of length $h+1$ obtained by appending an action to a plan $\pi \in \mathcal{PSET}_h$.
(3) Finally, the algorithm looks for a plan reaching the goal; function CheckGoal returns true when there exists a plan $\pi \in \mathcal{PSET}_{h+1}$ such that $\pi(\mathcal{I}) = \mathcal{F}$.

We have therefore demonstrated that the algorithm performs an exhaustive search and that no solution is lost: for $h < MAXDEPTH$, the algorithm is correct and complete. □

The algorithm is not complete in general; when it terminates for $h >= MAXDEPTH$, we can only assert that a solution with fewer than $MAXDEPTH$ actions does not exist, but we cannot state whether solutions involving a greater number of actions exist or not.

Corollary 1: When the algorithm terminates with success, the extracted solution π is optimal in terms of number of applied actions.

In other words, no any other conformant plan reaching the goal with less actions than π exists; this is a direct consequence of the exhaustive search the algorithm carries on.

Corollary 2: When the algorithm terminates with success, $\mathcal{P}SET_{h+1}$ maintains all the possible optimal solutions with exactly h+1 actions.

In fact, the search carries on all the possible plan candidates found at a given step of the search. So far we have not exploited this property as our current implementation returns the first conformant solution that has been found. As future work we may consider to return the best plan w.r.t. some preference criteria.

8.4. Advantages and limits of a conformant planner

Since the recovery strategy has to deal with ambiguous agent diagnoses, it has to exploit a planner which is able to deal with belief states and non-deterministic actions. To cope with this issue, in this paper we have proposed the adoption of a conformant planner, but there are other kinds of planners that can deal with ambiguity as well. For instance, contingent planning (Peot and Smith, 1992) is the problem of finding conditional plans given incomplete knowledge about the initial world and uncertain action effects. The basic idea of conformant planning is to anticipate, at planning time, all the possible contingencies that may arise during the plan execution phase; for each contingency a contingent plan is inferred; sensing actions are thus used to identify the contingencies at execution time. Thereby, a contingent plan is not a linear plan, as in the conformant case, but it similar to a tree where the agent's actions are interleaved with sensing actions. Contingent planning, however, does not seem to fit adequately the needs of the recovery strategy we have discussed. First of all, it is not easy in our scenario to anticipate all the possible contingencies. In fact, even though the set of possible faults is finite, the same fault occurring in different contexts may have different consequences [6]. Thus, predicting all the possible combinations of faults and contextual conditions in which those faults can occur may become unpractical. Moreover, contingent planning explicitly requires that at execution time the sensing actions can observe the happening of contingencies; i.e., faults, which are typically not observable.

The choice of a conformant planner is therefore motivated not only by the need to deal with ambiguity, but also by the lack of assumptions about the agent observability during the recovery procedure. Conformant planning, however, has also some drawbacks; first of all, it is computationally expensive (see the Appendix); in addition to that, the conformant requirement imposes a very stringent constraint on the possible solutions, and it is easy to miss a recovery plan. The conformant planner, in fact, tries to fix all the faults assumed by the agent diagnosis; therefore, it is sufficient that one of them is non-recoverable to conclude that no recovery plan exists, even though the actual fault is recoverable.

A possible way to mitigate this problem could be to consider preferred diagnoses rather than all the plausible diagnoses. One could synthesize a recovery plan just taking into account the most probable fault, and in that case a conformant planner would not even be required. Of course, the solution obtained in this way would not have any guarantee neither about its executability nor about

[6] See for instance the action model in Table 1 where a low charge in the battery has different consequences depending on the fact that the agent is carrying an object or not.

its effectiveness in restoring the nominal conditions. In fact, even if a fault is more probable than others, it may not be the actual fault. Another possible solution could be the adoption of active diagnosis techniques to better identify the actual fault. In active diagnosis, the agent is required to perform tests intended to confirm or disconfirm a fault. We leave this aspect of the recovery problem for future research.

9. RUNNING EXAMPLE

In this section we show how to recover from the failure of action a_2^1 that has been diagnosed as $D^{\texttt{A1}} = \{pwr = low \vee engTmp = hot\}$. First of all, we determine the recovered state $\mathcal{R}$ to be reached. As discussed above, $\mathcal{R}$ must represent an agent state where the faulty functionalities mentioned in the agent diagnosis have been fixed and the preconditions of a_2^1 are satisfied (the agent is ready to resume the execution of the original plan). It is easy to see that the recovered state $\mathcal{R}$ coincides with the variable assignments: (*pos*=`Rep`; *loaded*=`Pack1`; $pwr = ok$; $engTmp = ok$); namely, the agent is positioned in `Rep`, loaded with `Pack1`, and both the functionalities *pwr* and *engTmp* are in their nominal mode *ok*.

After this step, the relation Φ is built by considering the set of resources currently available to the agent, namely: {`Rep`, `Desk1`, `Pack1`, `Parking`}; agent `A1` has the exclusive access to the first three resources, while the last one is a non-constrained resource and hence it is always available to any agent. It follows that Φ={`go(A1,Rep,Parking)`; `go(A1, Parking, Rep)`; `go(A1, Rep, Desk1)`; `go(A1, Desk1, Rep)`; `go(A1, Desk1, Parking)`; `go(A1, Parking, Desk1)`; `load(A1, Pack1, Rep)`; `unload(A1, Pack1, Rep)`; `load(A1, Pack1, Desk1)`; `unload(A1, Pack1, Desk1)`; `recharge(A1)`; `refill(A1)`}.

Note that Φ also includes the repairing actions (`recharge` and `refill`) which can be used to fix the impaired functionalities; these actions can only be performed within the `Parking` area. As an example of repair action, Table 4 shows the relational model for the `recharge` action; the first entry of this table states that when the agent's battery is already charged, `recharge` has no effect; whereas the second one models the transition from the non-nominal *low* power to the nominal *ok*, and hence the fault is fixed.

The result of the planning process is sketched in Figure 8 , which shows how the $\mathcal{PSET}$ is incrementally built by successive applications of Φ starting from the initial agent belief $\mathcal{I}$ (see Table 2). For simplicity, the picture does not include in Φ the actions involving `Desk1` as this resource is not useful for the recovery purpose. In the picture, boxes represent agent belief states, whereas ellipses represents the action instances included in Φ. An arrow from a box b to an ellipsis e indicates that the action e is conformant w.r.t. b. An arrow from an ellipsis e to a box b indicates that the application of e yields the new agent belief b.

Actions along a path of solid arrows form plans leading to the final state $\mathcal{F}$, whereas actions along paths of dashed arrows represent infinite plans looping on "do"/"undo" actions. Gray boxes represent agent belief states along a conformant solution. From the picture it is apparent that two alternative solutions have been found:

Solution 1	Solution 2
(1) `unload(A1, Pack1, Rep)`	(1) `unload(A1, Pack1, Rep)`
(2) `go(A1, Rep,Parking)`	(2) `go(A1, Rep, Parking)`
(3) `recharge(A1)`	(3) `refill(A1)`
(4) `refill(A1)`	(4) `recharge(A1)`
(5) `go(A1, Parking,Rep)`	(5) `go(A1, Parking, Rep)`
(6) `load(A1, Pack1,Rep)`	(6) `load(A1, Pack1, Rep)`

The two solutions just differ with each other for the order with which the repair actions are performed within the parking area; our recovery strategy selects and returns one of them non-deterministically.

It is important to say that, for efficiency reasons, the conformant planner we have implemented maintains a history of visited states so that infinite loops of do/undo actions can be pruned off from the $\mathcal{PSET}$ structure speeding up the search.

As an alternative example, let us assume that agent `A1` fails in loading `Pack1`; action a_1^1

Table 4. The refined model for repair action `recharge(Parking)`.

active vars. at t		*active vars. at t+1*	
pos	**pwr**	**pos**	**pwr**
`Parking`	*ok*	`Parking`	*ok*
`Parking`	*low*	`Parking`	*ok*

terminates with a non-nominal outcome and the possible explanation inferred by the diagnostic analysis is $D^{\texttt{A1}}$={*hnd*=blocked} (i.e., the handling functionality is out of order). Since no repair action exists to fix the handling apparatus, such a fault cannot be autonomously repaired by agent `A1`; this means that the first step of our recovery strategy fails. If the agent simply stopped its activities, it would lock the repository preventing agent `A2` from using it. To mitigate the effects of such a failure, the conformant planner is invoked again with the aim of moving `A1` into a safe status. In this case, `A1` is in its safe status when it is located in `Parking`. A conformant plan to the safe status exists (except the handling functionality, all the other functionalities work properly), and this plan consists of a single action: `go(A1, Rep,Parking)`. The plan to safe status becomes the new `A1`'s plan, which releases the repository; of course, the subgoal assigned to `A1` can no longer be achieved unless a global repair strategy is activated.

10. EXPERIMENTAL ANALYSIS

In this section we give an empirical evaluation of the control loop we propose, implemented by exploiting the Ordered Binary Decision Diagrams (OBDDs) to efficiently encode the relations that are used at different steps of the loop. See the Appendix for some insights on how the operators of the Relational Algebra can be mapped into OBDDs operators, and for a computational analysis from a theoretical point of view.

In our experiments[7] we have (software) simulated a service-robot scenario where a team of robotic agents offers a "mail delivery service" in an office block. Resources are parcels, clerks' desks, doors, and repositories of parcels. Resources are constrained: desks, doors and repositories can be accessed by only one agent per time; at most one parcel can be put on a desk, whereas many parcels can be stored within a repository.

We have simulated a fairly large environment involving: 7 offices, 14 desks (two per office), 18 parcels, 7 doors connecting each office with a corridor, and 2 repositories located into the corridor. A parking area is also located into the corridor; this area is not constrained so many agents can move there simultaneously; an agent can perform a repairing action only when it is located into a parking area.

This environment is encoded within the status of an agent through 51 variables: 5 for the agent's health status, 7 for the agent's endogenous status, and 39 for the environment where the agent operates (these last variables are the private copies each agent keeps about the shared resources).

To test how good our approach scales up, we have considered four alternative scenarios including from 2 to 8 agents. In each scenario we have simulated the execution of 40 MAPs, whose main characteristics are reported in Table 5 (SCN2 stands for "scenario with two agents", SCN4 for "scenario with four agents", and so on). Each plan requires the involved agents to move parcels from a repository to a desk and vice versa. In most cases, an agent delivers a parcel to a desk and collects another parcel from another desk into a different office.

Since a subgoal consists in delivering a parcel to a desk or in storing a parcel into a repository, subgoals are achieved at different steps of the execution, not just at the end of the plan. Of course, when agents execute their plans under nominal conditions, they achieve all their subgoals as the given MAP is assumed to be correct; otherwise, the occurrence of a fault typically prevents the agents from

[7]The experimental data have been collected from a series of tests carried on a PC equipped with CPU: Intel Core 2, 2.40Gh; RAM: 3,24 GB; Windows XP OS. Agents have been implemented as Java (JDK 1.6) threads; coordination among agents has been realized by means of the exchange of XML messages. OBDDs have been made available via the JavaBDD package (Whaley, 2007), which relies on the BuDDy library.

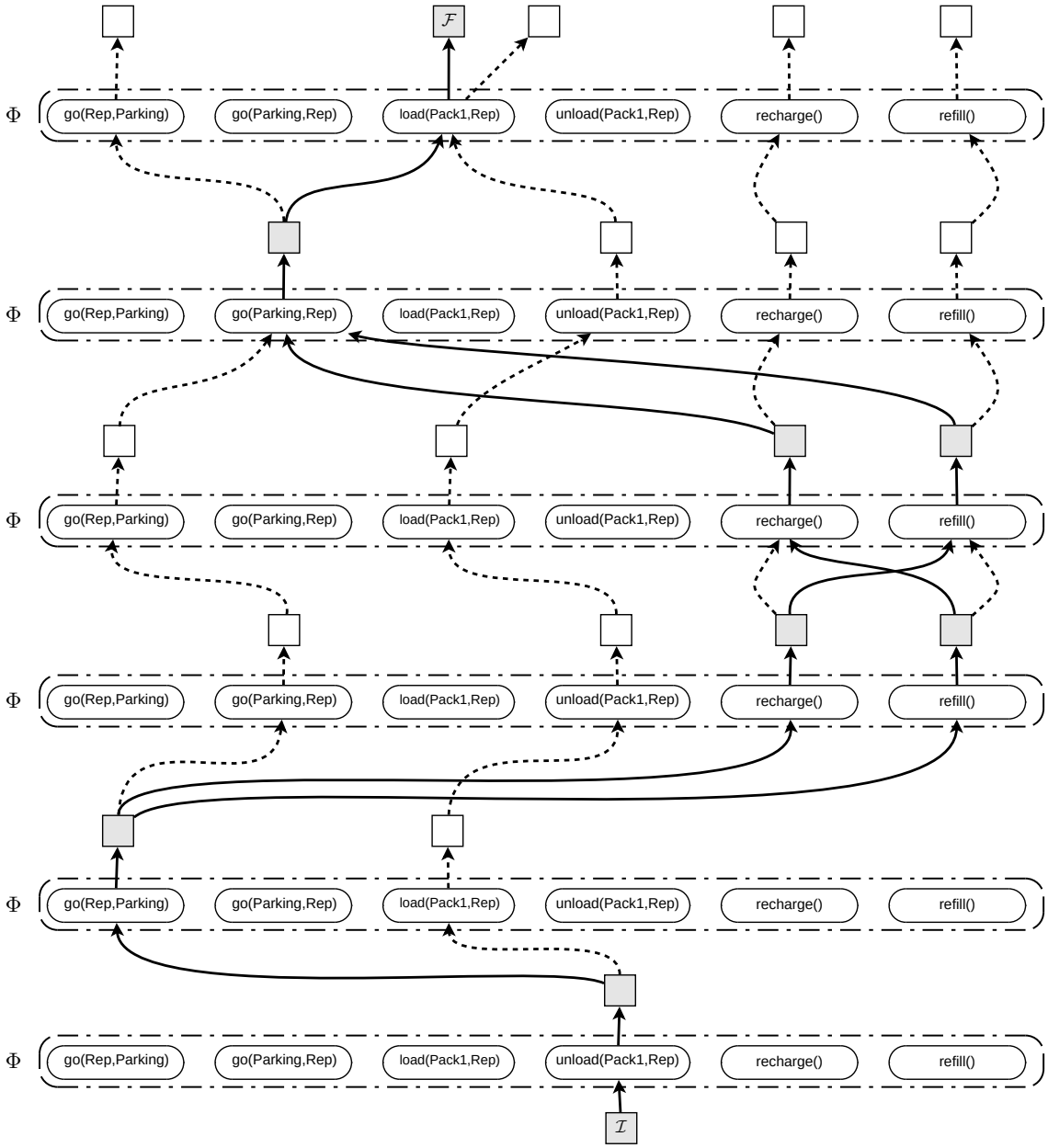

FIGURE 8. The $\mathcal{PSET}$ structure built during the conformant planning process.

achieving some of the desired subgoals; as we will see, a fault may have a huge impact affecting not only the agent where the fault occurs, but also other teammates. In the following analysis we use the number of achieved subgoals as a measure both of the impact of a fault in the plan, and of the effectiveness of the proposed repair strategy which tries to mitigate such an impact.

Table 6 reports relevant data about the execution of these plans under nominal conditions. Under this hypothesis, no time has been spent in the attempt of recovering from an action failure; thus we just show the average CPU time required by the monitoring phase in the four scenarios. In fact the CPU time just depends on the sizes of the OBDDs encoding the relations mentioned in Definition 2 (namely, the agent belief state and of the action model); while it does not depend on the number of agents involved in the team. It is evident that the monitoring activity is carried out very

Table 5. Main characteristics of the simulated plans (avg. values) in each scenario.

	SCN2	SCN4	SCN6	SCN8
# actions (avg.)	140	312	308	444
# casual links (avg.)	430	846	818	1062
# subgoals (avg.)	43	114	99	148
# actions per agent (avg.)	70	78	51	56
# subgoals per agent (avg.)	22	28	17	19

Table 6. Monitoring under nominal conditions: CPU time and OBDD dimensions.

	All four scenarios	
CPU time [msec] action monitoring	11 ± 3 (avg.)	14 (max.)
CPU time [msec] local plan monitoring	1072 ± 260 (avg.)	1355 (max.)
# nodes within an agent belief	403 ± 20 (avg.)	635 (max.)
# states within an agent belief	11 ± 2 (avg.)	16 (max.)
# nodes within an action model	432 ± 82(avg.)	860 (max.)
# state transitions within an action model	70 ± 26 (avg.)	129 (max.)

efficiently: estimating the belief state after the execution of a single action is in the order of just ten msec.; such a time must be compared to the actual time (in the order of seconds or even minutes) that a robotic agent takes for completing an action in the real world (service robots typically move very slowly especially when they share the environment with humans).

Concerning the recovery problem, a key parameter to be set is the $MAXDEPTH$ constant. Such a value must be chosen taking into account that if it is too low, one can lose solutions; whereas if it is too high one could waste time looking for a solution that does not exist, moreover a too long recovery plan may have a deleterious effect on the local plans of the other agents. In general, one can adopt some domain-dependent heuristics to determine a reasonable value for $MAXDEPTH$. For instance, in our experiment we have observed that, according to the disposition of doors and resources, the longest round-trip from a resource to the parking area, including one repair action, requires 15 actions; so we have set $MAXDEPTH$ constant to 15.

In order to prove the effectiveness of our recovery strategy, we have perturbed each plan by randomly injecting a fault in one agent's functionality. Three typologies of faults have been considered: *repairable* faults can autonomously be fixed by the agent through appropriate repair actions; *non-blocking* faults cannot autonomously be repaired by the agent but the affected agent may be able to move into a safe-status; finally, *blocking* faults have the worst impact as the affected agent cannot even move into a safe-status. In our experiments we have assumed that the first type of faults has a greater probability than the second one, which in turn is more probable than the last one. The chart in Figure 9 shows the distribution of faults in the four scenarios.

The experimental evaluation has been conducted by comparing four alternative strategies. The first strategy, the easiest one, is *no-repair*: the agent in trouble does not react to an action failure, it just stops the execution of its local plan. The second strategy, called *safe-status*, aims at moving the agent in trouble into a safe-status after the occurrence of an action failure. In the third strategy, *repair*, the agent tries to repair its own plan by invoking the conformant re-planner, when the repair process fails the agent stops the plan execution. The last strategy, the one proposed in this paper, is a combination of the two previous strategies: first the agent tries to repair its plan, when this step fails the agent tries to move into a safe-status; this strategy will be denoted as *r+s* (repair + safe-status).

The efficiency of the different repair strategies has been measured by taking into account both the CPU time and the memory occupation (in terms of number of OBDD nodes) for each of them when applied in the four scenarios. Table 7 shows the CPU time spent by the conformant planner for

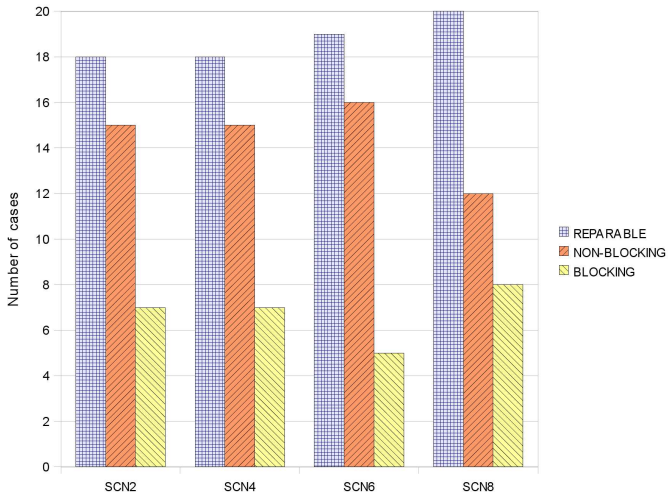

FIGURE 9. The distribution of faults over the four scenarios.

TABLE 7. SCN2 - Repair costs: CPU time in ms (in brackets the number of cases).

	repaired cases	cases moved to safe status	blocked cases
r+s	841 ± 129 [18 cases]	876 ± 138 [15 cases]	0 [7 cases]
repair	838 ± 128 [18 cases]	-	0 [22 cases]
safe status	-	112 ± 27 [33 cases]	0 [7 cases]
no-repair	-	-	- [40 cases]

TABLE 8. SCN2 - Repair costs: OBDDs encoding Φ and $\mathcal{PSET}$.

	All four strategies	
# nodes in Φ	6195 ± 697 (avg.)	7897 (max)
# actions in Φ	36 ± 5 (avg.)	58 (max)
CPU time to compute Φ [msec]	46 ±10 (avg.)	56 (max)
	r+s	
# nodes in $\mathcal{PSET}$	288 ± 62 (avg.)	1351 (max)
# repairing plans found	6 ± 2 (max)	33 (max)
# repairing plan length	10 ± 3 (avg.)	14 (max)

synthesizing either a recovery plan or a plan to safe status in SCN2. This time has been calculated according to the type of fault; e.g., *r+s* takes 841 msec (on average) when the case at hand is affected by a *repairable* fault; the same strategy takes a bit longer when it has to recover from a *non-blocking* fault as the conformant planner is invoked twice (first planning to $\mathcal{R}$, and then planning to $\mathcal{S}$). It is worth noting that, when the fault is blocking, the strategy *r+s* does not waste time in a vain search; this happens because the recovery strategy is driven by the agent diagnosis: it is sufficient to include a blocking fault among the explanations to conclude that a conformant solution does not exist. Note that the table also reports the number of cases in which each strategy has been activated successfully: the *r+s* strategy successfully intervenes in all the 18 repairable faults in this set of cases.

Table 8 shows the cost of the four strategies from the point of view of the dimensions (in terms of OBDD nodes) of two important structures: Φ and $\mathcal{PSET}$. First of all, since Φ is built up by joining refined action models, its dimension is independent of the actual strategy being used; moreover, even though Φ is a complex structure including a significant number of action models, the CPU time for computing it is almost negligible. The size of $\mathcal{PSET}$, on the other hand, which maintains the current set of plan candidates, depends on the strategy being used; for brevity we just show

TABLE 9. SCN4 - Repair costs: CPU time in ms (in brackets the number of cases).

	repaired cases	cases moved to safe status	blocked cases
r+s	841 ± 129 [18 cases]	876 ± 137 [15 cases]	0 [7 cases]
repair	750 ± 200 [18 cases]	-	0 [22 cases]
safe status	-	72 ± 25 [33 cases]	0 [7 cases]
no-repair	-	-	- [40 cases]

TABLE 10. SCN4 - Repair costs: OBDDs encoding Φ and $\mathcal{P}SET$.

	All four strategies	
# nodes in Φ	3945 ± 930 (avg.)	7932 (max)
# actions in Φ	27 ± 6 (avg.)	58 (max)
CPU time to compute Φ [msec]	46 ± 10 (avg.)	78 (max)
	r+s	
# nodes in $\mathcal{P}SET$	177 ± 48 (avg.)	708 (max)
# repairing plans found	3 ± 1 (max)	14 (max)
# repairing plan length	10 ± 3 (avg.)	13 (max)

TABLE 11. SCN6 - Repair costs: CPU time in ms (in brackets the number of cases).

	repaired cases	cases moved to safe status	blocked cases
r+s	1009± 130 [18 cases]	1053±135 [13 cases]	0 [9 cases]
repair	1007 ± 146 [18 cases]	-	0 [22 cases]
safe status	-	109 ± 34 [31 cases]	0 [9 cases]
no-repair	-	-	- [40 cases]

TABLE 12. SCN6 - Repair costs: OBDDs encoding Φ and $\mathcal{P}SET$.

	All four strategies	
# nodes in Φ	6125 ± 1107 (avg.)	10814 (max)
# actions in Φ	33± 7 (avg.)	70 (max)
CPU time to compute Φ [msec]	38 ± 6 (avg.)	93 (max)
	r+s	
# nodes in $\mathcal{P}SET$	252 ± 53(avg.)	413 (max)
# repairing plans found	4 ± 1 (max)	7 (max)
# repairing plan length	9 ± 2 (avg.)	11 (max)

some characteristics of this structure for the *r+s* strategy, captured when a solution has been found (i.e., when it gets the biggest size). Note that the dimension of $\mathcal{P}SET$ remains tractable in all the four scenarios (see tables 10, 12 and 14); this is an empirical demonstration that the *r+s* strategy is feasible in practice even though the theoretical computational cost of the conformant planner is exponential in the length of the plan (see the Appendix).

The analysis of the repair strategies in scenarios SCN4, SCN6 and SCN8 is similar; the results are showed in tables 9 through 14.

The effectiveness of the four strategies is synthesized in Figure 10, reporting the average number of plan actions performed by the agents in the four scenarios, and in Figure 11, where the percentage of achieved subgoals is showed. From the graphs it is apparent that *r+s* mitigates the harmful effects of a fault more effectively than the other strategies; in fact with *r+s* is activated, the agents can perform more actions of the original plan, and hence can achieve the greater number of subgoals. The chart in Figure 11 makes evident how the impact of a fault is mitigated by *r+s*: when *no-repair* is active, the percentage of achieved subgoals ranges from 40 to 63; this percentage rises to 71 / 84

TABLE 13. SCN8 - Repair costs: CPU time in ms (in brackets the number of cases).

	repaired cases	cases moved to safe status	blocked cases
r+s	1677 ± 45 [20 cases]	1765 ± 47 [11 cases]	0 [9 cases]
repair	1657 ± 35 [20 cases]	-	0 [20 cases]
safe status	-	217 ± 34 [31 cases]	0 [9 cases]
no-repair	-	-	- [40 cases]

TABLE 14. SCN8 - Repair costs: OBDDs encoding Φ and $\mathcal{PSET}$.

	All four strategies	
# nodes in Φ	9057 ± 1220 (avg.)	12426 (max)
# actions in Φ	37 ± 5 (avg.)	78 (max)
CPU time to compute Φ [msec]	46 ± 7 (avg.)	94 (max)
	r+s	
# nodes in $\mathcal{PSET}$	413 ± 62 (avg.)	529 (max)
# repairing plans found	10 ± 2 (avg)	13 (max)
# repairing plan length	10 ± 3 (avg.)	14 (max)

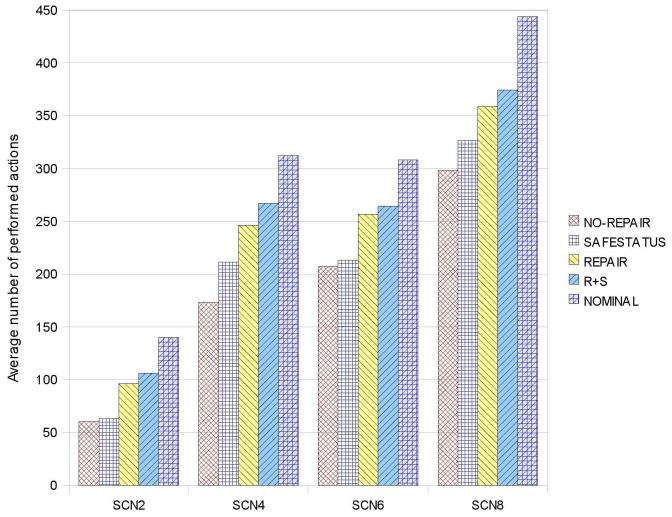

FIGURE 10. The average number of performed actions in the four scenarios: comparison between the repair strategies and the nominal situation.

when *r+s* is active.
Of course, *r+s* requires more CPU time than other strategies, but the difference with *repair* is very small; *safestatus* is cheaper than *r+s* and *repair* but its effects are limited; finally, *no-repair* is the cheapest, but, as expected, reaches the worst results.

From this analysis we can conclude that:

- the monitoring process can be considered *on-line* as it is sufficiently efficient to follow the actual execution of an action performed by a robotic agent;
- both monitoring and repair are able to deal with environments involving a significant number of resources;
- in the worst cases the repair strategy takes 1.6 sec. to find a repairing plan, this time is acceptable in many domains involving mobile service-robots which, for safety requirements, typically move very slowly;

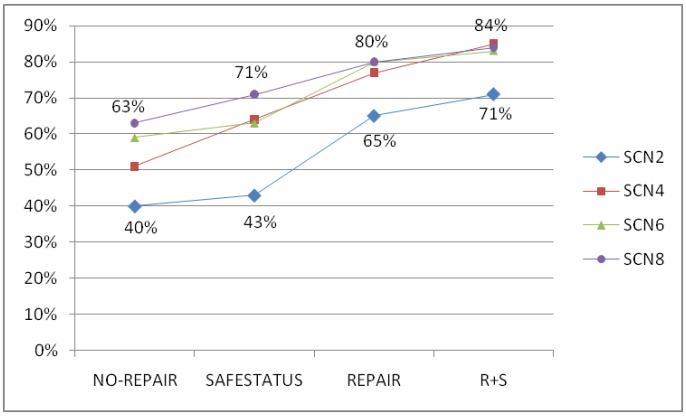

FIGURE 11. The percentage of achieved subgoals by the four repair strategies with the four scenarios.

- the number of actions within Φ (up to 78) demonstrates that the conformant planner is able to deal with non-trivial planning problems.

11. RELATED WORKS

There are two main research areas related to the work presented in this paper: *plan execution monitoring and diagnosis* and *plan repair*. The approach we have discussed, in fact, represents a first step for the integration of plan execution and plan repair within a single model-based framework. We start our discussion on related works by considering the plan repair area since the motivations of this work stems from that area.

In addressing the plan repair problem, a fundamental question must be answered: what is the best solution between synthesizing a new plan from scratch and adapting the existing (failed) plan? Despite Nebel and Koehler (1995) have shown that *plan adaptation* represents a real advantage w.r.t. plan from scratch only when it reuses as much of the original plan as possible, many researchers believe that plan adaptation still deserves to be investigated and a number of plan adaptation approaches have been proposed, see for example PRODIGY/ANALOGY (Veloso *et al.*, 1995), SPA (Hanks and Weld, 1995), PRIAR (Kambhampati and Hendler, 1992) , GPG (Gerevini and Serina, 2000).

The term *plan adaptation* comes from the case-based planning area, and refers to the process of modifying an existing plan π to solve a new problem, which is similar to the one solved by π; for instance, the old problem and the new one only differ for few facts in the initial and goal states. Thus the original intent of plan adaptation is to solve a new planning problem rather than repairing an existing plan (see e.g., Francis *et al.* (1995); Hanks and Weld (1995); Hammond (1986); Kambhampati (1990)).

More recent works (e.g., van der Krogt and de Weerdt (2005a); Cushing and Kambhampati (2005); Fox *et al.* (2006)) advocate that plan adaptation can be a viable means to get robust plan execution in dynamic domains. These works, in fact, point out that plans are in general synthesized by using abstract models of the worlds where they are going to be executed; since these abstract models just approximate the real world, discrepancies between the expected status of the world (as predicted by the models), and the actual one may be detected when a plan is actually executed. In order to cope with the new, unexpected conditions, the old plan must be replaced by a new one (i.e., the old plan needs to be repaired).

van der Krogt and de Weerdt (2005a) propose a basic template for plan repair which includes two basic steps: first, actions that prevent the achievement of the goal(s) are removed; and then, actions that bring the agent closer to the goal(s) are introduced. The first phase is also called plan unrefinement, whereas the second one is called plan refinement (see also Kambhampati (1997)). Plan unrefinement is driven by a heuristic in order to find an appropriate portion of the plan to be removed; whereas, plan refinement consists of a planning process that has to fill the gap left by the

unrefinement step.

In the approach we have discussed we do not have an explicit phase of plan unrefinement. However, the planning phase can implicitly realize this step by including into the repairing plan actions which compensate the effects of actions that have already been executed. In the example discussed in section 9, for instance, both the two repairing solutions start with an *unload* action which compensate the effect of a previous *load*.

In Fox *et al.* (2006), the authors introduce the notion of *plan stability* as a metric to assess the safeness of a repaired plan. In fact, when the repaired plan is close to the original one it is easier for humans to understand the changes. Moreover, a plan is a means to communicate future intentions to other agents and, to do this successfully, repaired plans should be as close as possible to the original plan (Cushing *et al.*, 2008; Cushing and Kambhampati, 2005).

Since an agent is typically situated in an environment shared with other agents, some authors have addressed the problem of plan repair in a multi-agent setting. van der Krogt and de Weerdt (2005b) discuss a planning methodology where self-interested agents achieve globally consistent local plans by means of an iterative plan repair process. In their approach each agent infers a local plan autonomously, during this process the agent can decide either to achieve a goal by itself or to ask some other agent for it. When an agent accepts to achieve a goal for another agent, it has to change its original plan in order to include among its goals the new one; its local plan is therefore adapted (i.e., repaired) to this end. The methodology is very interesting but introduces some assumptions which make it unsuitable for our scenario; for example, it is assumed that agents cannot share resources and that agents have a complete knowledge about its own problem. Both assumptions are acceptable during a planning phase, but they are not in plan execution.

Another approach to multi-agent replanning is discussed in (Zhang *et al.*, 2007); the authors present a distributed refinement strategy to construct a graph plan fixing errors occurred during the execution of a multi-agent plan; this work, however, does not take into account that failures may be due to faults in the agent's functionalities.

Extensions of Markov Decision Processes (MDPs) have also been proposed to solve the problem of coordinating a team of agents. For example, Decentralized MDPs (Dec-MDPs) have been proposed in (Bernstein *et al.*, 2002) to model teams of agents that operate under *collective observability*. Collective observability is a condition where, although each agent is unable to independently identify the global world state, the union of all the observations received by the team members at a given time yield a fully observed state. Other approaches are based on *factored* MDPs (Boutilier *et al.*, 2000; Guestrin *et al.*, 2001; Guestrin and Gordon, 2002); while standard MDPs enumerate all the possible states, factored MDPs take advantage of conditional independencies among state variables and yield more compact state representations, and possibly more compact policies. Although these approaches are mainly concerned with the planning problem in a multi-agent scenario, they could also be adopted to solve the plan repair problem.

The literature on plan execution monitoring is very broad; seminal studies on this topic come from the field of industrial control, where it is often referred to as the problem of fault detection and isolation (FDI) (Pettersson, 2005). Since the end of 1970s, a number of monitoring methodologies have been validated in industrial applications see (Chen and Patton, 1999; Gertler, 1998) for an overview.

Roughly speaking, execution monitoring in robotics can be categorized into two main families of approaches: *model-based* and *model-free*. In model-based approaches, a model of the system (robot and environment) is used to generate expectations about the nominal behavior of the robot; these expectations are subsequently compared to the available observations received during the actual execution in order to detect discrepancies. Examples of architectures in this category include, among others, the Remote Agent by Muscettola *et al.* (1998), the LAAS proposal (Alami *et al.*, 1998) and ROUGE (Haigh and Veloso, 1998). Model-free approaches, on the other hand, do not use predictive models, but pattern recognition mechanisms in order to learn on-line both nominal and faulty behaviors (Pettersson *et al.*, 2003, 2005). It is easy to see that the present paper falls into the first category.

Most of the approaches in the literature assume that action failures are consequences of unexpected environment conditions; in this paper we adopt a vision similar to the one discussed in (Birnbaum *et al.*, 1990), where the outcome of an action is related to the health status of the agent

performing it. A non nominal action outcome is therefore explained in terms of domain-dependent threats which have affected the health status of the agent. In our work, plan threats are faults in the agent's functionalities; the proposed extended action models capture the relation between the agent health status and action outcomes.

The work by Birnbaum et al. has been recently extended to the multi-agent scenario. To the best of our knowledge, however, only a few approaches tackle plan execution monitoring and diagnosis in a multi-agent scenario. In (Roos and Witteveen, 2009) the authors consider the multi-agent plan as the system to be diagnosed, and hence they introduce the notion of *plan diagnosis* as the subset of actions whose failure is consistent with the anomalous observed behavior of the system; thus this work just focuses on the detection of abnormal actions. In (de Jonge *et al.*, 2009) the same authors complement plan diagnosis with the notion of secondary diagnosis. More precisely, while the plan diagnosis is the primary diagnosis identifying the failed actions; the secondary diagnosis explains why those actions have failed. Roos et al. distinguish three possible secondary plan diagnoses: agent diagnosis (faults in the agent's functionalities), equipment diagnosis (faults in the resources used by the agents), and environment diagnosis (unexpected changes in the environment). To achieve this result, the authors exploit action models which share some similarities with the extended models we have proposed; in fact, their action models explicitly mention the functionalities and resources required for the successful completion of the action; moreover, known faulty models of the agent's functionalities can be used to partially predict the next agent status.

At the current stage of development, the portion of our framework related to plan diagnosis is a subset of the framework by Roos et al. But in this work we are more interested in the whole control loop from detection to recovery, and we have considered here just the agent diagnosis to simplify the discussion. Nonetheless, Roos et al. have opened some appealing research lines along which our strategy could be extended. Indeed, we have started working on diagnosing failures caused by unexpected changes in the environment, see (Micalizio and Torasso, 2008, 2009). From the recovery point of view, however, the control strategy must still be extended to deal with unexpected exogenous events. For instance, a new general strategy must be conceived to match an exogenous event with a recovered state the affected agent should reach in order to recover from a failure.

A different notion of diagnosis is the one discussed in (Kalech and Kaminka, 2007) where the authors introduce the *social diagnosis* to find out the cause of coordination failures. In their approach the authors do not explicitly consider plans, they rather model a hierarchy of *behaviors*: each agent selects independently from others the more appropriate behavior given its own beliefs. When an agent selects a behavior which is not in accordance with the ones selected by its teammates, a disagreement has been detected, and a diagnostic process is activated in order to determine its root causes. In this case, the social diagnosis explains the disagreement by individuating portions of the agents' beliefs which are one another in conflict.

12. SUMMARY AND CONCLUSIONS

Cushing and Kambhampati (2005) have observed that, to repair a plan after the occurrence of an action failure, one has to consider not only the unexpected changes occurred in the environment, but also faults affecting the agent's functionalities. In their opinion, thus, a repair strategy has to get two results: on the one side it must adjust the original plan according to new environment conditions; and on the other side, it must also fix (if possible) the agent's faults.

The paper originates from this observation and represents an attempt of complementing previous works on plan repair (see e.g., van der Krogt and de Weerdt (2005a); Fox *et al.* (2006)), which typically assume that the agent is fault-free and that the causes of an action failure are due to the environment. In this work we have focused on those situations where an action failure is due to faults; and we have proposed a recovery strategy which aims at restoring a healthy status in the agent's functionalities.

The paper points out some important challenges one has to face to solve the problem of action failure recovery. First of all, there is the need of detecting action failures and relating them to faults. Since the nominal action models used during the planning phase are inadequate for this purpose, we have introduced *extended action models* which allow a mapping between faults and action failures. However, since faults may have non-deterministic effects, mapping an action failure with a fault is

not a trivial task as a number of alternative explanations may be possible. To cope with this problem, the paper adopts techniques from the Model-Based Diagnosis (MBD) for detecting and diagnosing action failures. Finally, since faults and their contextual conditions are difficult to anticipate, it is not possible to exploit precompiled recovery solutions; so the paper presents a recovery strategy, which based on a conformant planner and driven by the agent diagnosis, synthesizes on-the-fly a recovery plan fixing the (assumed) faulty functionalities.

The paper contributes to the plan repair area in a number of ways. First of all, it presents in a formal way a unique framework where three main activities - monitoring, diagnosis, and recovery - are integrated with one another within a closed loop of control. The framework has been discussed in terms of relations and Relational Algebra operators between relations. This has two immediate advantages: 1) it is easy to define extended action models with non-deterministic effects in terms of relations; 2) the Relational Algebra is a general language so the control loop can be presented independently of its actual implementation.

A further contribution of the paper is that the recovery strategy, albeit conceived for a single agent, can successfully be adopted within a multi-agent setting. As noticed by Roth *et al.* (2007), in fact, in many multi-agent domains agents operate independently of one another for long periods and coordinate their actions in just few occasions. The basic idea of our strategy is to intervene in the plan segments where an agent operates isolated from others; the recovery, however, preserves the plan segments where the agent interacts with other team members. Therefore, when the recovery terminates successfully, only the plan of the agent in trouble has been changed, and this change is completely transparent to the other members of the team.

Limits of the framework and future developments The proposed local strategy is meant to be a first attempt to overcome an action failure trying to change the plan of the impaired agent only; when a recovery plan does not exist, the impaired agent tries to move into a safe status where it does not represent a menace for other agents. It may be possible, however, that such a safe status is not achievable or even missing; in that case the faulty agent gets stuck and locks a subset of resources indefinitely. A possible way to overcome this limitation consists in complementing the local strategy with a team-based strategy, where agents cooperate with one another to overcome a single failure. In particular, in a team-based strategy agents would be able to acquire and release resources freely, without the limitations imposed by locality requirement. This would help an agent to build a recovery plan by acquiring new resources. Moreover, when a recovery plan does not exist, the faulty agent could coordinate with the other teammates in order to move from place to place without locking resources indefinitely. In other words, the safe status of a faulty agent would not be a place where it can isolate itself, but a property or a high-level goal supported by a form of continual planning.

The work by van der Krogt and de Weerdt (2005b) discusses an interesting approach to plan repair in a multi-agent scenario where agents negotiate goals and services, however, the work introduces some assumptions which make the approach not directly applicable in our scenario. For instance, the authors assume that the problems of all the agents are mutually distinct, meaning that agents cannot use the same resources. Of course, this assumption can be hardly applied in a service-robots scenario, where resources are actually shared and agents are typically involved in conflicts/negotiations for accessing them. An interesting research line is to extend the framework by van der Krogt et al. in order to relax their assumptions and meet our scenario.

Another possible extension is about the agent diagnosis. In the current solution the recovery mechanism tries to find a plan fixing all the faults assumed by the agent diagnosis, which in general is ambiguous. Thereby some of the repairing actions included within the recovery plan are not really necessary. Redundant repairing actions may represent an issue especially when they are expensive and time consuming. A possible solution to mitigate the problem could be the adoption of "active diagnosis" techniques to refine the ambiguous agent diagnosis. The basic idea of active diagnosis is to perform a number of actions intended to test the agent's functionalities assumed to be faulty: test actions are therefore used to confirm or disconfirm the hypotheses made by the agent diagnosis. Of course, active diagnosis techniques do not guarantee to precisely identify the actual fault affecting an agent, but they can significantly reduce the ambiguity of the agent diagnosis, and hence reduce the number of redundant repairing actions in the recovery plan.

REFERENCES

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *International Journal of Robotics Research*, **17**(4), 315–337.

Bernstein, D., Givan, R., Immerman, N., and Zilberstein, S. (2002). The complexity of centralized control of markov decision processes. *Mathematics of Operations Research*, **27**(4), 819–840.

Birnbaum, L., Collins, G., Freed, M., and Krulwich, B. (1990). Model-based diagnosis of planning failures. In *Proc. AAAI90*, pages 318–323.

Boutilier, C. and Brafman, R. I. (2001). Partial-order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research*, **14**, 105–136.

Boutilier, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *Artificial Intelligence*, **121**(1-2), 49–107.

Brenner, M. and Nebel, B. (2009). Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-agent Systems*, **19**, 297–331.

Bryant, R. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, **35**(8), 677–691.

Bryant, R. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computer Surveys*, **24**, 293–318.

Chen, J. and Patton, R. (1999). *Robust Model-Based Fault Diagnosis for Dynamic Systems*. Kluwer Academic Publishers, Boston, MA,USA.

Cimatti, A. and Roveri, M. (2000). Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, **13**, 305–338.

Console, L. and Torasso, P. (1991). Integrating models of correct behavior into abductive diagnosis. *Computational Intelligence*, **7**(3), 133–141.

Console, L., Theseider, D., and Torasso, P. (1991). On the relation between abduction and deduction. *Journal of Logic and Computation*, **1**(5), 661–690.

Cox, J. S., Durfee, E. H., and Bartold, T. (2005). A distributed framework for solving the multiagent plan coordination problem. In *Proc. International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'05)*, pages 821–827.

Cushing, W. and Kambhampati, S. (2005). Replanning: A new perspective. In *Poster Program of the International Conference on Automated Planning and Scheduling (ICAPS'05).*

Cushing, W., Benton, J., and Kambhampati, S. (2008). Replanning as deliberative re-selection of objectives. In *Technical report. ASU CSE.*

Darwiche, A. and Marquis, P. (2002). A knowledge compilation map. *Journal of Artificial Intelligence Research*, **17**, 229–264.

de Jonge, F., Roos, N., and Witteveen, C. (2009). Primary and secondary diagnosis of multi-agent plan execution. *Journal of Autonomous Agents and Multi-Agent Systems*, **18**, 267–294.

Decker, K. and Li, J. (2000). Coordinating mutually exclusive resources using gpgp. *Journal of Autonomous Agents and Multi-Agent Systems*, **3**(2), 113–157.

Durfee, E. H. (2001). Mutli-agent systems and applications. chapter Distributed problem solving and planning, pages 118–149. Springer-Verlag New York, Inc., New York, NY, USA.

Fikes, R. and Nilsson, N. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, **2**(3-4), 189–208.

Fox, M. and Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, **20**, 61–124.

Fox, M., Gerevini, A., Long, D., and Serina, I. (2006). Plan stability: Replanning versus plan repair. In *Proc. International Conference on Automated Planning and Scheduling (ICAPS'06)*, pages 212–221. AAAI Press.

Francis, A. G., Jr., and Ram, A. (1995). A domain-independent algorithm for multi-plan adaptation and merging in least-commitment planners. In *AAAI Fall Symposium: Adaptation of Knowledge Reuse*, pages 19–25.

Gerevini, A. and Serina, I. (2000). Fast plan adaptation through planning graphs: local and systematic search techniques. In *Proc. International Conference on Artifical Intelligence Planning and Scheduling (AIPS'00)*, pages 112–121.

Gertler, J. (1998). *Fault Detection and Diagnosis in Engineering Systems*. New York, NY, USA.

Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., and Wilkins,

D. (1998). Pddl - the planning domain definition language. In *Proc. International Conference on Artifical Intelligence Planning and Scheduling*.

Guestrin, C. and Gordon, G. (2002). Distributed planning in hierarchical factored mpds. In *Proc. of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, pages 197–206.

Guestrin, C., Koller, D., and Parr, R. (2001). Multiagent planning with factored mdps. In *Proc. of Advances in Neural Information Processing Systems (NIPS'01)*, pages 1523–1530.

Haigh, K. and Veloso, M. (1998). Planning, execution and learning in a robotic agent. In *Proc. International Conference on Artifical Intelligence Planning and Scheduling*, pages 120–127.

Hammond, K. J. (1986). Chef: A model of case-based planning. In *AAAI*, pages 267–271.

Hanks, S. and Weld, D. (1995). A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research (JAIR)*, **2**, 319–360.

Jensen, R. M. and Veloso, M. M. (1999). Obdd-based universal planning: Specifying and solving planning problems for synchronized agents in non-deterministic domains. *LNCS*, **1600**, 213–249.

Jensen, R. M. and Veloso, M. M. (2000). Obdd-based universal planning for synchronized agents in non-deterministic domains. *Journal of Artificial Intelligence Research*, **13**, 189–226.

Kalech, M. and Kaminka, G. A. (2007). On the design of coordination diagnosis algorithms for teams of situated agents. *Artificial Intelligence*, **171**, 491–513.

Kambhampati, S. (1990). Mapping and retrieval during plan reuse: A validation structure based approach. In *AAAI*, pages 170–175.

Kambhampati, S. (1997). Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, **18**(2), 67–97.

Kambhampati, S. and Hendler, J. (1992). A validation structure based theory of plan modification and reuse. *Artificial Intelligence*, **55**, 193–258.

Lam, M., S., Whaley, J., Livshits, V. B., Martin, M., C., Avots, D., Carbin, M., and Unkel, C. (2005). Context-sensitive program analysis as database queries. In *Proc. of the twenty-fourth ACM SIGMOD Symposium on Principles of Database Systems*, pages 1–12.

Lhoták, O. and Hendren, L. (2004). Jedd: a bdd-based relational extension of java. *SIGPLAN Not.*, **39**, 158–169.

Lind-Nielsen, J. (2003). Buddy: A binary decision diagram package. http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/index.html/.

Micalizio, R. and Torasso, P. (2008). Monitoring the execution of a multi-agent plan: Dealing with partial observability. In *Proc. of European Conference on Artificial Intelligence (ECAI'08)*, pages 408–412.

Micalizio, R. and Torasso, P. (2009). Agent cooperation for monitoring and diagnosing a map. In *Multiagent System Technologies, 7th German Conference, MATES*, volume 5774 of *LNCS*, pages 66–78.

Muscettola, N., Nayak, P., and Williams, B. (1998). Remote agent: to boldly go where no ai system has gone before. *Artificial Intelligence*, **103**, 5–47.

Nebel, B. (2000). On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, **12**, 271–315.

Nebel, B. and Koehler, J. (1995). Plan reuse versus plan generation: a theoretical and empirical analysis. *Artificial Intelligence*, **76**(1), 427–454.

Peot, M. and Smith, D. E. (1992). Conditional nonlinear planning. In *Proc. International Conference on Artifical Intelligence Planning and Scheduling(AIPS'92)*, pages 39–46.

Pettersson, O. (2005). Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, **53**, 73–88.

Pettersson, O., Karlsson, L., and Saffiotti, A. (2003). Model-free execution monitoring in behavior-based mobile robotics. In *Proceedings of the International Conference on Advanced Robotics (ICAR)*, pages 864–869.

Pettersson, O., Karlsson, L., and Saffiotti, A. (2005). Model-free execution monitoring by learning from simulation. In *Proc. of Computational Intelligence in Robotics and Automation (CIRA)*, pages 505–511.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, **32 (1)**, 57–96.

Roos, N. and Witteveen, C. (2009). Models and methods for plan diagnosis. *Journal of Autonomous*

Agents and Multi-Agent Systems, **19**(1), 30–52.

Roth, M., Simmons, R., and Veloso, M. (2007). Exploiting factored representations for decentralized execution in multi-agent teams. In *in Proc. International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'07)*, pages 469–475.

Torta, G. and Torasso, P. (2007). On the role of modeling causal independence for system model compilation with obdds. *AI Communications*, **20**(1), 17–26.

van der Krogt, R. and de Weerdt, M. (2005a). Plan repair as an extension of planning. In *Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS'05)*, pages 161–170.

van der Krogt, R. and de Weerdt, M. (2005b). Self-interested planning agents using plan repair. In *Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS'05)*, pages 36 – 44.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., and Blythe, J. (1995). Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, **7**, 189–197.

Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine*, **15**(4), 27–61.

Whaley, J. (2007). Javabdd package - project summary page at. *http:\\javabdd.sourceforge.net*.

Zhang, J. F., Nguyen, X. T., and Kowalczyk, R. (2007). Graph-based multi-agent replanning algorithm. In *Proc. International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'07)*, pages 798–805.

APPENDIX Relational Algebra Operators

This appendix summarizes some basic notions about relations and operators of the Relational Algebra.

A relation R consists of a schema R_σ and of a set of contained tuples R_τ. The schema is an ordered list of attributes, or variables, R_σ=$\langle v_1, \ldots, v_n \rangle$, where each variable v_i $(i : 1..n)$ has a finite domain $dom(v_i)$. Each tuple $t = \langle d_1, \ldots, d_n \rangle$ in R_τ represents an assignment of values to the variables in the relation schema such that $d_i \in dom(v_i)$ for each $i : 1..n$ (namely, $t \in dom(v_1) \times \ldots \times dom(v_n)$).

The Relational Algebra operators we are interested in are:

- *union*: given two relations R and S defined over the same set of attributes (R_σ equals S_σ); the union $R \cup S$ is a new relation T, defined on the same set of attributes as R and S, containing all the tuples of R and all the tuples of S without duplicates; formally: T_σ=R_σ=S_σ, and $T_\tau = R_\tau \cup S_\tau$.
- *intersection*: given two relations R and S defined over the same set of attributes; the intersection $R \cap S$ is a new relation T, having the same schema as R and S, and containing the tuples which belong both to R and to S: $T_\tau = R_\tau \cap S_\tau$. Of course, T might result to be empty.
- *difference*: given two relations R and S defined over the same set of attributes; the difference $R - S$ is a new relation T, having the same schema as R and S, and containing the tuples which belong to R but which do not appear in S: $T_\tau = R_\tau - S_\tau$.
- *Cartesian product*: given two relations R and S defined over schemas which are one another disjointed ($R_\sigma \cap S_\sigma = \emptyset$), the Cartesian product $R \times S$ is a new relation T such that:

  1. T_σ=$R_\sigma \cup S_\sigma$
  2. $T_\tau = R_\tau \times S_\tau$= $\{t$ is a tuple in T_τ such that $t = t_1 \cup t_2, \forall\ t_1 \in R_\tau, \forall\ t_2 \in S_\tau\}$: each tuple in R is combined with each tuple in S, the result is stored in T.

- *selection*: given a relation R and a predicate ρ, the selection $\text{SELECTION}_\rho R$ is a new relation T such that:

  1. T has the same schema as R
  2. each tuple in T_τ satisfies the predicate ρ

  ρ is defined over (a subset of) the schema variables and typically represents a restriction of the domains of these variables.
- *projection*: given a relation R and a subset $\gamma \subseteq R_\sigma$ of variables , $\text{PROJECTION}_\gamma R$ is a new relation T such that:

1. $T_\sigma = \gamma$
2. T_τ is the projection of R_τ over the variables in γ: all the variables which are not in γ are discarded, duplicated tuples are removed.

- join: in the paper we only utilize the *natural-join* operator, which is a specific case of join, for brevity however we use the generic term *join*. Let R and S be two relations such that $R_\sigma \cap S_\sigma \neq \emptyset$ (the two relations share a subset of variables); let ω be such a subset of shared variables. The (natural) join operation, denoted as R JOIN S, is the new relation T defined as:

1. T_σ=$R_\sigma \cup S_\sigma$ (without duplication of variables)
2. $T_\tau = \{t \text{ is a tuple} \mid t = t_1 \cup t_2,\ \forall\ t_1 \in R_\tau,\ \forall\ t_2 \in S_\tau, \text{ such that } \ \forall v \in \omega,\ t_1(v) = t_2(v)\}$

The natural join combines a tuple t_1 in R with a tuple t_2 in S only when the two tuples are related to each other as they assign the same values to the shared variables in ω.

APPENDIX Computational Analysis

In this work we have presented in a declarative manner a methodology to control the execution of a plan by integrating model-based diagnosis and conformant planning. Both these activities have been formalized in terms of relations and operations between relations.
The computational complexity of the proposed algorithms strongly depends on how efficiently these relations are handled; any operation on relations, in fact, can easily become a bottleneck. Before addressing the computational analysis, it is therefore necessary to briefly discuss how the algorithms have been implemented.

Basic concepts on OBDDs

We have adopted the formalism of the Ordered Binary Decision Diagrams (OBDDs) (Bryant (1986)) to symbolically, and hence compactly, encode the relations involved by the control strategy. OBDDs are a well-known mathematical tool, which compactly encode complex Boolean functions as rooted directed acyclic graphs having only two leaves: zero (i.e., false) and one (i.e., true). While OBDDs have initially been developed to test hardware failures, they are becoming widely used in many areas of the Artificial Intelligence for efficiently representing and manipulating large state spaces (see for instance, diagnosis of component-based systems (Torta and Torasso (2007)), and different kind of planning (Cimatti and Roveri (2000); Jensen and Veloso (2000))). Some basic operators on OBDDs are reported in Table 15 (see also Bryant (1986)). In the table, the Boolean function f is represented by a reduced function graph G containing $|G|$ vertices, and similarly for the functions f_1 and f_2. The meaning of these operators is as follows:

- the *reduce* operator gets the canonical form of a Boolean function f; i.e., given a specific variables ordering, the reduce operator gets a graph G whose size is minimal
- the binary logical operations between two Boolean functions f_1 and f_2 is realized by the *apply* operator working on the graphs G_1 and G_2, respectively encoding the two functions. The computational complexity in the worst case is the product of the sizes (i.e., number of vertices) of the two graphs.
- the *restrict* operator substitutes a constant b to a variable x_i in time almost linear in the number of vertices.

TABLE 15. OBDD operators and their complexity.

operator	result	time	size
Reduce(f)	G reduced to canonical form	$O(\|G\| \cdot \log \|G\|)$	$\|G\|$
Apply(op, f_1, f_2)	$f_1 \langle op \rangle f_2$	$O(\|G_1\| \cdot \|G_2\|)$	$\leqslant \|G_1\| + \|G_2\|$
Restrict(x_i, b, f)	$f\|_{x_i=b}$	$O(\|G\| \cdot \log \|G\|)$	$\leqslant \|G\|$

The adoption of the OBDDs is not a *per se* panacea; since the computational complexity of the OBDDs operators depends on the sizes of the involved OBDDs, it is important to get minimal-size OBDDs. However, the size of an OBDD depends on the variable ordering chosen during the reduce operation, a wrong variable ordering may result in an exponential growth of the OBDD size. It has been demonstrated that finding an optimal variable ordering minimizing the size of an OBDD is a **NP-hard** problem. Heuristics are typically used to cope with this issue; for instance, many approaches keep close variables which depend on one another (Jensen and Veloso (1999); Torta and Torasso (2007)).

The problem of how translating a relation into an OBDDs has been addressed in previous works (Bryant (1992); Lhoták and Hendren (2004)). In this paper we just give some hints on how to solve it. First of all, we observe that any n-ary relation R can be encoded as a Boolean function f_R:

$$f(d_1, \ldots, d_n) = \begin{cases} 1 & \text{if } \langle d_1, \ldots, d_n \rangle \in R[\tau] \\ 0 & \text{otherwise} \end{cases}$$

For instance, given the belief state in Table 2, the corresponding Boolean function is defined as $f_{belief} : dom(pos) \times dom(loaded) \times dom(pwr) \times dom(engTmp) \times dom(hnd) \rightarrow \{0, 1\}$, and it assumes the value 1 only for the two combinations of values reported in the table, namely, $\langle$ `Rep`, `Pack1`, *ok*, *high*, *ok* $\rangle$ and $\langle$ `Rep`, `Pack1`, *low*, *ok*, *ok* $\rangle$; the function is 0 for any other combination of values. By means of a proper renaming of variables, also the relations representing action models (which report the same status variables at two consecutive time instants) can be translated into Boolean functions. For instance, the extended model of action *go* (see Table 1) is modeled by the Boolean function
$f_{go} : dom(pos) \times dom(loaded) \times dom(pwr) \times dom(engTmp) \times$
$dom(posNxt) \times dom(loadedNxt) \times dom(pwrNxt) \times dom(engTmpNxt) \rightarrow \{0, 1\}$.
As in the previous case, the function f_{go} assumes the value *true* for each combination of values corresponding to an entry of Table 1, *false* in any other case.
Note that we have heuristically chosen the variable ordering $\{pos, posNxt, loaded, loadedNxt \ldots\}$ in order to obtain an OBDD G_{go} efficiently encoding the Boolean function f_{go}; in fact, the value of a "next" variable will strongly depend on the value of the corresponding "current" variable.

Once we have translated each relation into a Boolean function, we can observe that each operator of the Relational Algebra can be mapped to an operation (or a combination of operations) between Boolean functions (see Lam *et al.* (2005)). Consider for example the predictive step of the monitoring process, realized by a JOIN operator between the current belief state and the action model, this step is translated into the Boolean expression $f_{belief} \wedge f_{go}$.

Note also that some of the packages implementing OBDDs provide useful operators which correspond to a sequence of Relational operations. For instance, natural join operations are frequently followed by project operations to eliminate unnecessary attributes. The OBDD operation `relprod`, provided by the BuDDy package Lind-Nielsen (2003), efficiently combines this sequence in a single operation. Similarly, the select and project operations can be combined into a single`restrict` operation between OBDDs (Lam *et al.*, 2005).

Computational Analysis

In this section we discuss separately the computational complexity of the three main tasks (monitoring, diagnosis and recovery) included within the proposed control strategy by taking into account their OBDD-based implementation.

Plan Execution Monitoring. The main goal of the monitoring activity consists in estimating the agent belief state $\mathcal{B}^i(t+1)$ given the previous belief state $\mathcal{B}^i(t)$ and the extended model of action $a_l^i(t)$. This process has been formalized in Definition 2. To assess the computational complexity of such a process, we have first to consider how it is actually translated into OBDD operators. Let G_{belief} be the OBDD encoding $\mathcal{B}^i(t)$, G_{action} be the OBDD encoding the model of action $a_l^i(t)$, and G_{obs} be the OBDD encoding the available observations $obs^i(t+1)$; then the Relational expression

$$\text{SELECTION}_{obs^i(t+1)}(\mathcal{B}^i(t) \text{JOIN } \Delta(a_l^i(t)) \tag{6}$$

is translated into the following expression on OBDDs:

$$G_{int} = apply(\wedge, G_{obs}, apply(\wedge, G_{belief}, G_{action})) \tag{7}$$

Where G_{int} is the intermediate OBDD resulting from the two subsequent apply operations. Relying on Table 15, we can conclude that the complexity for getting G_{int} is

$$O(|G_{belief}| \cdot |G_{action}| \cdot |G_{obs}|) \tag{8}$$

The last relational operation to obtain the belief state $\mathcal{B}^i(t+1)$ is the projection over the status variables in VAR^i_{t+1} (i.e., the "next" variables). To remove a variable v from an OBDD one can exploit the restrict operation, in particular, one has to invoke the restrict operator for each value in $dom(v)$. The projection is therefore translated into a number of restrict operations over G_{int}, which is proportional to the number of variables to be discarded (namely, the variables in VAR^i_t), and to the cardinality of the variable domains. Let $maxDomSize = \text{argmax}_{v \in VAR^i} |dom(v)|$ be the greatest domain cardinality; the computational cost for the projecting G_{int} over VAR^i_{t+1} is

$$O(maxDomSize \cdot |VAR^i| \cdot |G_{int}| \log |G_{int}|) \tag{9}$$

Although this computational result might not appear completely satisfactory, it is important to note that it is an estimation of the very worst case. Bryant himself states that this theoretical complexity is not so frequent in practice, and he conjectures that the actual complexity of an apply operation between two OBDDs G_1 and G_2 is $O(|G_1| + |G_2| + |G_3|)$, where G_3 is the resulting OBDD (Bryant, 1986). Indeed, the experimental results discussed in section 10 have shown that the monitoring process is actually carried out very efficiently and that the size of the OBDDs encoding the agent belief states and the action models are kept small.

Agent diagnosis. The inference of the agent diagnosis in Relational terms is given in Definition 4, it simply consists in a projection of the belief state $\mathcal{B}^i(t+1)$ over the health status variables. Let $G_{newBelief}$ be the OBDD encoding such an agent belief state; the computational complexity for projecting $G_{newBelief}$ over the variables in $healthVar(a^i_l)$ is

$$O(maxDomSize \cdot |VAR^i \setminus healthVar(a^i_l)| \cdot |G_{newBelief}| \log |G_{newBelief}|) \tag{10}$$

In fact, also in this case the projection is translated into a number of restrict operations which is proportional to the number of variables that must be removed and to the size of their domains.

Recovery. The recovery strategy is formalized in the algorithm of Figure 6; this algorithm first determines which goal must be reached (either $\mathcal{R}$ or $\mathcal{S}$), and then invokes the conformant planner trying to find a plan getting the desired goal. It is easy to see that the real complexity of the recovery process is in the solution of a conformant planning problem rather than in the recovery strategy itself; for this reason, we will only examine the computational cost of the conformant planner.
The pseudo-code of the conformant planner is given in Figure 7. To analyze the computational complexity of this algorithm, we focus our attention on those steps in which the size of the relations grows or might represent an issue; in fact, these are the same steps where the OBDD implementation might become inefficient.
In particular, there are two steps which deserve our attention. The first step we consider is

$$\mathcal{PSET}_{h+1} \leftarrow \mathcal{PSET}_h \text{ JOIN } \Phi \tag{11}$$

which extends the plan candidates in $\mathcal{PSET}_h$ one step longer by applying the Φ relation. Since $\mathcal{PSET}_h$ is built incrementally, we can unfold it as

$$\begin{aligned} \mathcal{PSET}_h &\leftarrow \mathcal{PSET}_{h-1} \text{ JOIN } \Phi \\ \mathcal{PSET}_{h-1} &\leftarrow \mathcal{PSET}_{h-2} \text{ JOIN } \Phi \\ &\dots \\ \mathcal{PSET}_1 &\leftarrow \mathcal{PSET}_0 \text{ JOIN } \Phi \end{aligned}$$

Where $\mathcal{PSET}_0$ coincides with the initial state $\mathcal{I}$.
As we have noticed earlier, the relational *join* is translated into an apply operation between OBDDs,

it follows that expression (11) is translated into the following OBDD operation

$$G_{\mathcal{P}SET_{h+1}} = apply(\wedge, G_{\mathcal{P}SET_h}, G_\Phi) \tag{12}$$

which in turn is unfolded as:

$$\begin{aligned} G_{\mathcal{P}SET_h} &= apply(\wedge, G_{\mathcal{P}SET_{h-1}}, G_\Phi) \\ &\dots \\ G_{\mathcal{P}SET_1} &= apply(\wedge, G_\mathcal{I}, G_\Phi) \end{aligned}$$

Where G_Φ and $G_\mathcal{I}$ are the OBDDs encoding the Φ relation and the initial state $\mathcal{I}$, respectively; and $G_{\mathcal{P}SET_k}$ is the OBDD encoding $\mathcal{P}SET_k$ (for $k : 0..h+1$).
The computational complexity for getting $G_{\mathcal{P}SET_{h+1}}$ is therefore

$$O(|G_\mathcal{I}| \cdot |G_\Phi|^{h+1}) \tag{13}$$

The second critical step we consider is:

$$\mathcal{P}SET_{h+1} \leftarrow \textbf{PruneNotConformant}(\mathcal{P}SET_h, \mathcal{P}SET_{h+1}) \tag{14}$$

which discards from the new space of plan candidates all those candidates which are not conformant plans. As we have shown, this result can be obtained by exploiting Property 5; namely, when $\mathcal{B}_h \cap \overline{\mathcal{B}_h^{ext}} = \emptyset$ the plan candidate including $\mathcal{B}_h$ is a conformant plan as it is built by a sequence of h conformant actions.
Thus, we have first to extract the agent belief state $\mathcal{B}_h$ from $\mathcal{P}SET_h$ and $\mathcal{B}_h^{ext}$ from $\mathcal{P}SET_{h+1}$. A naïve implementation based on the restrict operation on OBDDs would be very inefficient, for instance the computational cost for getting $G_{\mathcal{B}_h}$ would be

$$O(maxDomSize \cdot |VAR^i|^{h-1} \cdot |G_{\mathcal{P}SET_h}| \log |G_{\mathcal{P}SET_h}|) \tag{15}$$

In fact, we have to remove h-1 times the status variables in VAR^i in order to obtain a single agent belief state. The computational complexity for extracting $\mathcal{B}_h^{ext}$ is similar.
To get a more efficient implementation, we have exploited the `relprod` operator made available by the BuDDy package, this operator has the same complexity as the apply operation, but has the advantage of existentially quantifying the variables we are not interested in, and hence pruning them off from the final result. More precisely, we can obtain $\mathcal{B}_h$ as

$$G_{\mathcal{B}_h} = \texttt{relprod}(G_{\mathcal{P}SET_h}, G_{\mathcal{P}SET_{h-1}}) \tag{16}$$

In fact all the variables mentioned in $G_{\mathcal{P}SET_{h-1}}$ are removed from the final result; the cost of this operation is $O(|G_{\mathcal{P}SET_h}| \cdot |G_{\mathcal{P}SET_{h-1}}|)$.
In a similar way we can obtain the OBDD $G_{\mathcal{B}_t^{ext}}$ from $\mathcal{P}SET_{h+1}$.
After this step, we have just to verify whether $\mathcal{B}_h \cap \overline{\mathcal{B}_h^{ext}}$ is equal to zero. The dual belief state $\overline{\mathcal{B}_h^{ext}}$ is obtained in constant time as it is sufficient to invert the two special nodes **1** and **0** in $G_{\mathcal{B}_t^{ext}}$.
The intersection between $\mathcal{B}_h$ and $\overline{\mathcal{B}_h^{ext}}$ is again implemented by means of an apply operator, so its computational complexity in worst case tends to be the product of the sizes of the involved OBDDs. Note that, as far as the agent belief states remain small, the cost of such an intersection is negligible w.r.t. the cost for extracting $\mathcal{B}_h$ and $\mathcal{B}_h^{ext}$.
According to Property 5, if the intersection $\mathcal{B}_h \cap \overline{\mathcal{B}_h^{ext}}$ is not empty, it means that we have applied a non-conformant action. Thus $\mathcal{P}SET_{h+1}$ must be refined by pruning off all the non-conformant actions. This can be done by the following `relprod` operation

$$G_{\mathcal{P}SET_{h+1}} = \texttt{relprod}(G_{\mathcal{P}SET_{h+1}}, G_{\mathcal{B}_h}) \tag{17}$$

Which discards from $\mathcal{P}SET_{h+1}$ all the plan candidates leading to $\mathcal{B}_h$ and has complexity $O(|G_{\mathcal{P}SET_{h+1}}| \cdot |G_{\mathcal{B}_h}|)$. Also this operation is negligible w.r.t. the cost of extracting $\mathcal{B}_t$ and $\mathcal{B}_t^{ext}$; so we can conclude that the computational cost of the PruneNotConformant step is dominated by

$$O(|G_{\mathcal{P}SET_h}| \cdot |G_{\mathcal{P}SET_{h-1}}|) + O(|G_{\mathcal{P}SET_{h+1}}| \cdot |G_{\mathcal{P}SET_h}|) \approx O(2 \cdot |G_{\mathcal{P}SET_{h+1}}| \cdot |G_{\mathcal{P}SET_h}|) \tag{18}$$

Substituting equation (13) in equation (18) we have that the computational complexity of the conformant planner is dominated by

$$O(2 \cdot |G_\mathcal{I}|^2 \cdot |G_\Phi|^{2(h+1)}) \tag{19}$$

Again, this result might appear below the expectations, however, the experimental analysis we have discussed in Section 10 demonstrate how such a worst case is quite infrequent in practical cases. In particular, we have empirically demonstrated that the size of the $\mathcal{P}SET_h$ structure does not grow exponentially. This is a consequence of the fact that, each time the space of plan candidates is extended, not all the actions in Φ can be actually applied as conformant actions, and hence the real branching factor is well below $|\Phi|$.

As a final remark about the conformant planner, it is easy to see that the worst situation the algorithm may encounter is when it has to explore the whole space of plan candidates (limited by $h < MAXDEPTH$) before returning an answer. This may happen when solutions with less than $MAXDEPTH$ do not exist, or when there exists at least a solution with exactly $MAXDEPTH-1$ actions. It must be noticed, however, that when a problem has no solution, we have experimentally observed that the $\mathcal{P}SET_h$ structure in many cases becomes empty in very few iterations of the algorithm. This happens because the algorithm discovers that no action in Φ can be applied as a conformant action. It follows that the hardest problems to be solved are those whose solutions include a number of actions very close to $MAXDEPTH$.