Using hybrid planning for plan reparation

Patrick Bechon[1], Magali Barbier[1], Charles Lesire[1], Guillaume Infantes[1] and Vincent Vidal[1]

***Abstract*— In this work we propose a plan repair algorithm designed to be used in a real-life setting for a team of autonomous robots. This algorithm is built on top of a hybrid planner. This planner mixes partial order planning and hierarchical planning. This allows the creation of a plan with temporal flexibility while using human knowledge to improve the search process.**

Simulation shows that repairing increases the number of solved problems or at least reduces the number of plans explored. The algorithm uses the same hierarchical knowledge as the underlying planner, thus needing no more human modelling to properly run. We show that using this knowledge can help the reparation, even when some half-executed abstract actions are present in the plan.

I. Introduction

This paper presents a decisional architecture allowing a team of heterogeneous mobile robots to be autonomous in the achievement of a real-life mission. To execute a mission, an initial plan is computed offline, taking into account all the robots and mandating rendezvous to synchronize them. When an unexpected event happens, the involved robots must amend their own part of the plan. This paper mainly deals with this update of their, now broken, local plan.

In a first approach one can use the planning algorithm used for generating the initial plan to compute a new plan from scratch. This is called *replanning*. Another approach is to use the initial plan and amend it to generate a new plan. This is called *reparation*. One advantage of plan repair is its ability to modify the current plan as little as possible: the *plan stability*. Those terms are introduced in [7]. The stability is important in cases where several robots are collaborating thus reducing the amount of synchronisation between them. The proposed architecture is based on plan reparation, but a comparison with replanning is shown.

To improve the efficiency and scalability of both the planning and the reparation processes, our architecture uses hierarchical planning. This means that additional information can be given to the planner and is used in a heuristic to guide the search. In our case, hierarchical information takes the form of abstract actions that encompass several other actions, described with their own preconditions and effects. We do not include hierarchical information regarding different reparation strategies or different events to which we have to be robust. Only information pertaining to the case of planning is needed, not strategies to deal with every corner case of what could go wrong during execution. This paper presents how we use this information during the repair process and how it improves the performance of the overall architecture.

After a summary of the related work, the first part will describe the inner workings of the planning algorithm. Then we will describe the repair algorithm. The last part contains results obtained on several hierarchical benchmarks.

[1]Onera — The French Aerospace Lab, F-31055, Toulouse, France `firstname.lastname@onera.fr`

II. Background and related work

The developed planning technique is sometimes called *hybrid planning*, a mix between partial order planning and hierarchical planning. Partial Order Planning (also called POCL for Partial Order Causal Link) is used by several planners such as UCPOP [17], VHPOP [22], CPT [20]. It allows to output plans with temporal flexibility but is usually slower than other planning techniques. Hierarchical planning is mainly known through Hierarchical Task Network [6]. It allows the user to specify abstract actions that bundle together several other actions and guide the planner to prune the search tree. In their initial form, HTNs do not allow temporal planning. Several hybrid planners exist, such as PANDA [19], FAPE [5] or HiPOP [2]. Their goal is to combine the temporal reasoning and flexibility of POP algorithms with the efficiency of HTNs.

Plan reparation is an active research topic. Some algorithms take a local approach, where the planner can combine moves forward and moves backward during the search [11]. But one has to find a way to avoid loops in this setting, and heuristics have to be designed to efficiently search the space. Another approach is to plan for a subset of the unexpected events: *conditional planning* [18]. The plan is then created with some choices left to be made during execution but this cannot deal with really unexpected events. A related idea is *probabilistic planning* [12], where the uncertainty is factored into the planning process and produces a policy. Some work for the DARPA Coordinators project, eg. [1], starts with an initial plan but assumes that no agent can have the global view of the problem at any given time. Plan reparation is closely related to *case-base reasoning* [13] where one wants to adapt a general plan to a particular situation. The difference with our approach is that we do not specify rules to repair, the reparation uses the same knowledge than when planning. Yet another approach is to not only store a plan but also the reasoning that led to this plan. So when something changes, the algorithm can follow the same reasoning and create a new plan. This is done for instance in [15], [21].

Plan repair using a POP algorithm is introduced in [11]. Unlike the work presented here, no hierarchical knowledge is used during planning or reparation. Plan repair using a hybrid planner has been proposed in [3]. The authors did

not provide any implementation or comparison of the results of their algorithm, only formal properties using a replay of the search procedure leading to the initial plan.

Other architectures have been proposed to execute and repair onboard failures for robots or team of robots. IDEA [16] provides an architecture based on timelines, but incorporates several kinds of planners. In this case a deliberative planner is tasked to compute a global plan while a reactive planner is tasked to return a locally-executable plan when needed. T-REX [14] is another architecture mixing different planners, each with a different horizon and response time. The global planning is done by exchanging messages between those planners (called *reactors*), each of them responsible for a different level of abstraction.

Our goal is to use the same planning algorithm in all the layers of the hierarchy. This method removes the need of synchronizing mechanisms between the different planners while allowing the same depth of reasoning, that would not be possible without hierarchy. The ability to output a flexible plan, through POP reasoning, is important to reduce the load of the planner during the execution.

III. Planning

HiPOP [2] is used as the underlying hybrid planner.

A. Partial order planning with abstract actions

1) STRIPS planning: The state of the world is described as a set of positive literals (STRIPS representation). PDDL2.1 [8] is used to describe the planning problem. A problem instance is defined by $\mathcal{A}$, the set of available actions, I, the set of literals representing the initial state and G, the set of literals representing the goal.

An action is defined by $\mathcal{P}$, the set of literals representing its preconditions, $\mathcal{E}$, the set of literals representing its effects, d, its duration, $\mathcal{M}$, a set of partial plans (called methods) and $\mathcal{C}$, a set of conflicts described in [2].

The actions such that $\mathcal{M} = \mathcal{C} = \emptyset$ can be executed and are called elementary actions. Those actions can be described in PDDL. The other actions, called abstract, cannot be executed. They must be "instantiated" (ie. replaced by one of their methods) for the plan to be valid.

The aim of the planning algorithm is to find a sequence of actions that reaches the goal from the initial state. The algorithm explores the space of partial plans to find a plan achieving every literal of the goal. Since an action can be scheduled multiple times, it is represented as a step in a plan. A step s is a tuple $s = (a, t_s, t_e)$ where a is an action, t_s and t_e are indexes of timepoints. They represent respectively the start time and end time of s. We call s an elementary step if a is elementary and an abstract step if a is abstract.

A partial plan P is defined by $\mathcal{S}$, a set of steps, $\mathcal{TC}$, a set of (simple temporal) constraints over the time points defined by $\mathcal{S}$, $\mathcal{CL}$, a set of causal links, $\mathcal{H}$, a set of hierarchical relationships and Abs, a set of execution times.

A Simple Temporal Network [4] over the timepoints of the steps can be associated to each plan. A causal link between two steps represents the fact that one step is creating, as an effect, a precondition of another step. This induces a precedence relationship between timepoints. Two steps are said to be causally linked if there exists a causal link from one to the other. A temporal link induces a precedence relationship between two timepoints.The execution times define the absolute execution time for some timepoints. When set in the past this represents an executed step. When in the future this defines a deadline. The hierarchical relationship describes a forest among steps: a set of trees whose nodes are steps. An elementary step is a leaf. An uninstantiated abstract step is added as a leaf but during its instantiation all new steps introduced become its children. A step introduced outside an instantiation is a *root* step.

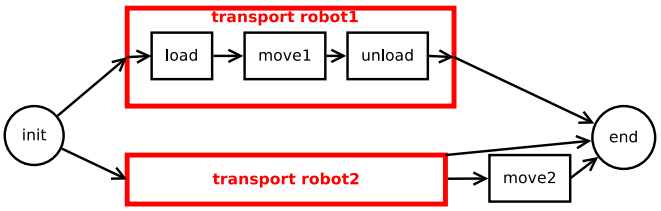

Fig. 1. Example of a simple partial plan. Abstract actions are in thick red, elementary are in black. Round actions are dummy actions that respectively create the initial state and consume the goals. Arrows represent causal links. Both *transport* actions and *move2* are root actions. The actions *load*, *move1* and *unload* are children of *transport robot1*.

Given a partial plan P with no uninstantiated steps, we can define its *elementary* plan as the partial plan obtained by removing all abstract steps, emptying $\mathcal{H}$, and changing all the causal links and temporal links to point to elementary steps instead. This is the plan obtained when we remove all the hierarchical information of P. If P is a solution to a planning problem, its elementary plan will also be a solution.

One can also compute $\mathcal{F}$, the set of flaws of a partial plan. Those flaws, each of them preventing the plan to be a solution to the problem, can be of three types: an open link when a step precondition is not guaranteed, a threat when a step could delete a literal while a causal link is active and an abstract flaw when an abstract step is not yet instantiated.

The planning algorithm is an adaptation of a Partial Order Planning, shown on Algorithm 1. The *PopBestPlan* and *PopBestFlaw* procedures are based on heuristics described in [2]. The *Resolvers* procedure computes all partial plans that solve a particular flaw. The *InitialPartialPlan*, in the planning setting, returns a minimum plan: no steps, no causal links or temporal links and open links corresponding to the goal.

Algorithm 1: Basic POP algorithm

$\Pi = \{InitialPartialPlan(I, G)\}$;
while $\Pi \neq \emptyset$ **do**
  $P = PopBestPlan(\Pi)$;
  if $\mathcal{F}(P) = \emptyset$ **then**
    return P ;
  $f = PopBestFlaw(\mathcal{F}(P))$;
  $\Pi = \Pi \cup Resolvers(\mathcal{A}, P, f)$;
return $\emptyset$

2) Deadlines: A deadline is defined as a goal (a literal) that must be achieved at a set time, for example "The robot should be at its charging station 3 hours after the beginning of the mission". We modelled it as a step inserted into the initial plan. The associated action has the goal as a precondition and its start time is in Abs with the value of the deadline, so it is possible to have multiple deadlines simultaneously.

3) Temporal flexibility: The presence of the STN allows the final plan to be temporally flexible: each step is not associated with an absolute time but with some temporal constraints. It is possible to compute absolute bounds (earliest and latest dates) and to update those bounds when new information is acquired, for instance when a point is executed. All timepoints are assumed to be controllable so a delayed action may invalidate the STN and trigger a repair.

4) Allowed list: Planning with abstract and elementary actions together can lead the planner to explore several branches of the search tree leading to the same elementary plan. It may also allow the planner to return a plan that does not conform to what the user intends by disregarding abstract actions and by planning only with elementary actions. So the user-defined knowledge also contains the list of actions that can be used as root actions: steps using this action can be inserted in the plan without being the child of another step, to solve an open link.

In Fig. 1 both *transport* actions and *move* actions were on this list. *Load* and *unload* were not, so the planner always inserts two matching steps together.

5) Low priority predicate: For the planner to use several layers of abstraction together, it has to focus on some parts of the problem first. The user can specify a set of open link that has to be solved after the abstract links, usually the position of the robots. This allows the planner to instantiate all the steps before taking care of adding some steps to do the transition between the abstract steps.

B. Execution

The execution algorithm keeps updated the STN associated with the plan. At any point, if a controllable timepoint has all its preceding timepoints executed, its execution is launched. We assume that the execution can decide itself when an abstract step starts or ends since they have no influence on the world and cannot be executed. When the algorithm is notified at the end of a step, it updates the STN with the execution of the associated uncontrollable timepoint.

IV. Reparation

For repairing, we assume that we have access to the current problem, the current time and the initial plan. The execution of the plan may have started, meaning that some steps in the plan can be already executed and that no step can be scheduled before the current time.

1) General repair procedure: To repair, we kept the forward-search mechanism of HiPOP. To ensure completeness, we iteratively compute a starting partial plan by removing steps in the failed plan until a solution is found or everything is removed. The overall process is described in Fig. 2. The different steps used are:

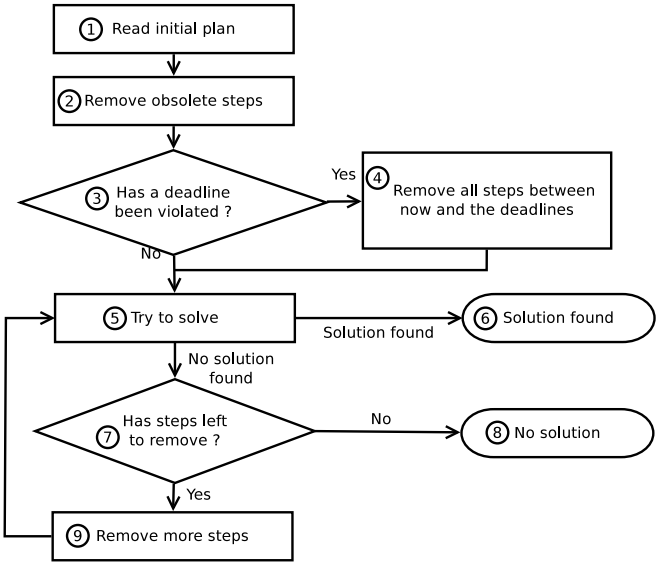

Fig. 2. Repair process

Remove obsolete steps (2): remove all the steps that are in the failed plan but whose action is not in the domain anymore. This also removes useless steps, ie. steps that do not have a chain of causal links to one of the goals.

Violated deadlines (3-4): A deadline is violated if the STN associated to the plan is temporally coherent without it but incoherent with. In this case, all steps between the current time and the deadline times are removed.

Solving (5-6) The current plan is used as a starting plan Π for the planning algorithm (Algorithm 1). It is the root of the search tree. Its flaws are recomputed. Since no backward steps are taken, all steps, causal links and temporal links in Π will be in the output.

Remove more steps (7-9): At each new call it will remove all steps that were previously causally linked to an already removed step. The already executed steps cannot be removed. Nor can be the steps introduced to create a deadline. But constraints implied by those delete steps have to be kept.

While this algorithm is enough to repair, we present here contributions that will improve the repair process, mainly using the hierarchical information encoded in the plan.

2) Removing action in hierarchy: When removing an action from the plan, one should preserve the hierarchy when possible. Children of the same actions are semantically linked together, they all contribute to the same higher-level goal. Since we have more information than just causal links, we can use it to remove more actions or to try to preserve the user intent of the abstract description.

If a root step is removed, all its children are removed (recursively if one of its children is also abstract). This preserves the fact that non-root actions should not be in the plan if not part of an abstract action. If a non-root step is removed, its parent is uninstantiated: the planner will try to

find an alternative method to accomplish this action.

In the case where the parent step has a child that is already executed, it is impossible to remove the whole step. For instance, in Fig. 1 the hierarchy may forbid the insertion of a *load* step without its matching *unload*. If the *unload* step fails then the planner must be able to insert a single *unload* step to create a valid plan. So, in such case, we leave all the other siblings in the plan and allow the planner to insert any elementary actions, by adding all elementary steps to the *allowed list* presented in III-A.4. At the next call to **remove more steps**, more siblings will be deleted until all the non-executed siblings are eventually removed.

3) Interaction with deadlines: If an abstract step has at least one child step executed but is before the deadline, it is necessary to remove its non-executed children to be able to achieve the deadline even if this means not respecting the user intend. Since this introduces an abstract step not conforming to the abstract description, we need to allow the planner to insert all the elementary actions to repair this instance of the abstract action. So we consider, as in the previous part, all the elementary actions as part of the *allowed list*.

4) Dealing with forbidden action: If an elementary action is forbidden, methods using this action might not be complete any more. For instance in the previous example, if the *move* action used in the *transport* step is forbidden, the instantiation of a *transport* step will not be possible using the same method.

But HiPOP (and hybrid planning in general) is robust to non-complete methods ie. methods which need the introduction of other steps for their instantiation to be valid. So we still allow all abstract actions, even those using the forbidden action, to be inserted. But during the instantiation all forbidden steps and their associated causal links are not introduced into the plan. This could introduce open links or threats that the planner will have to solve. We then modified the heuristic to account for the fact that in presence of an abstract step whose methods used a forbidden action, the planner will have to find an alternative. So the heuristic cost of those steps is computed as the cost of achieving the effects of the forbidden actions, as if there were open links.

5) Formal properties: The repair process makes heavy use of the underlying planning algorithm. So the properties of the reparation depend on the properties of the planning process. We assume here that the planning process is sound and complete.

Soundness All plans output by the algorithm are an output of the planning algorithm (Fig. 2). So if the planning algorithm is sound, the repair algorithm is also sound.

Completeness As shown on Fig. 2, the only case where the repair algorithm cannot find a solution is when the initial plan has no removable steps and the planner cannot find a solution. This is a *replan*, and the planner did not find a solution. If the planner is complete, there exists no solution. So the repair process is complete.

V. Experimental evaluation

The first domain used for the evaluation is *survivors* [2]. The world is represented as a grid, some survivors are scattered on this grid and hospitals are available in the grid. A set of robots, organized in teams, must explore each location and take each survivor to a hospital. The abstract actions allow the planner to reason in terms of teams of robots exploring zones whereas elementary actions only deal with individual robots and locations. The second domain is called *parking*. We used a 3D model of a parking and a robotic path planning library [10] to generate a set of allowed actions in this parking. The goal of the two robots is to explore every intersection of this parking. In addition, they must also be in communication (ie. on the same spot) at a precise time. The planner has the choice of this point.

The main metrics used are the total time needed for the repair algorithm to compute a plan, the number of plans explored and the similarity between the initial plan and the repaired plan. We use the definition of [7]: if P_1 and P_2 are two plans, their similarity is measured as the number of steps that are in P_1 but not in P_2 plus the number of steps that are in P_2 but not in P_1.

Given an initial problem, we compute an initial plan, then simulate its execution. Due to the lack of space, the inner working of the simulation is only sketched here. The simulation may increase the duration of a step or fail one action permanently, modifying the problem. At each unexpected event, a reparation is triggered only if necessary. For instance if a step takes more time than planned, the execution process updates the STN. If the plan is still valid, the execution continues (this is when the temporal flexibility of the plan is important). If the plan becomes invalid, the reparation process is triggered and the planner cannot remove the already executed steps.

The results are presented on Table I. The experiments were run on a desktop computer (Intel Xeon 2.67GHz, 3GB of RAM) with a timeout of 10 minutes. For the *parking* domain, one initial plan was computed. For the *survivors* domain, three initial plans were computed. For each initial plan and each type of defect, 15 instances of this defect are randomly chosen. The execution is then launched. When the defect occurs during execution, the planner is called to repair the current plan, both with and without using the hierarchy. Each column of the table represents a kind of defect:

- *durOnce*: One step of the initial plan, occurring before the deadline, is delayed of a fixed offset.
- *failOnce*: One step in the initial plan fails and its associated action is forbidden. The planner is called when the first step using this action should start.
- *failOnce-move*: Same as before except that only motion steps can fail. They are usually easier to repair since they can be replaced by a sequence of moves bypassing the troublesome path.
- *-replan*: When the planner is called, it deletes every removable step (so it only *replans*).

Regarding the delays, only the *parking* domain is interest-

***Parking* domain**					
	durOnce	failOnce	failOnce-move	failOnce-move-replan	failOnce-replan
Number of successes	12/12	15/12	15/12	15/6	13/8
Mean number of plans explored	39.17/256.00	3434.87/2700.08	2704.27/620.42	1162.73/664.33	386.54/1722.12
Mean percentage of steps changed	9.22/7.62	23.69/9.75	32.91/18.09	90.07/71.28	79.38/49.73
***Survivors* domain**					
	durOnce	failOnce	failOnce-move	failOnce-move-replan	failOnce-replan
Number of successes	45/45	45/45	45/45	40/45	36/43
Mean number of plans explored	0.00/0.00	39.07/293.16	6.18/8.87	91.25/278.04	127.61/378.44
Mean percentage of steps changed	0.00/0.00	7.38/7.93	6.82/6.72	18.78/69.78	24.37/84.80

TABLE I

RESULTS OF THE REPAIR PROCESS. EACH CELL CONTAINS TWO VALUES: ONE FOR THE HIERARCHICAL ALGORITHM ON THE LEFT AND ONE FOR THE NON-HIERARCHICAL ALGORITHM ON THE RIGHT. MEANS ARE COMPUTED ON SUCCESSFUL INSTANCES.

ing since it is the only one with a deadline that can invalidate the plan. We only delayed steps that happened before the deadline (else no repair would have been triggered). Of the 15 executions, 9 triggered a repair and 3 were impossible to repair. On all successful reparations, the hierarchical repair explored less plans than the non-hierarchical repair.

Regarding the action failures, in the *parking* domain, when actions fail, the success rate of the reparation with hierarchy is better (almost always 100% success rate) than without. The mean number of plans explored increases when the hierarchy is used, but this allows the algorithm to solve more problems. The map is not strongly connected in this problem, so when an action fails the bypassing route can be difficult to find. This explains why the number of plans explored is not that reduced, in *parking*, when only move steps fail.

The number of steps changed also increases when using the hierarchy in both cases. This is expected since using abstract actions can introduce steps that would not be part of the minimal plan or focus the search on a suboptimal but more general approach. The number of steps changed also increases when using *replanning* instead of *repairing*. This is expected since *replanning* means discarding all scheduled steps to replace them with new ones.

In the *survivors* domain, the *reparation* with and without abstract actions both solved all problems. But when using abstract actions, the number of plans explored drops. The number of steps changed is steady. There is a difference with *parking* in the number of plans explored if a move action fails. Since the grid is much more connected, it is way easier to repair motion failure rather than general failure.

The *replanning* is made harder if an abstract step is started but not finished when the reparation is triggered. The reparation is designed to allow the planning process to insert elementary actions to deal with this situation. But when asked to replan, allowing all elementary actions in addition to the abstract action can be detrimental. The same effect can be seen when planning with abstract actions while still allowing all actions to be inserted as root actions. Thus there exist several branches in the search tree leading to the same elementary plan, and if the planning algorithm explores them concurrently this actually penalizes it to have more choices. This explains why the performance of the reparation are poorer in this case.

We also ran tests in another setting. If the defect in the plan does not impact the action currently being executed, then all the special cases for half-executed actions do not apply anymore. This could happen for instance if a sensor breaks down but should not be used in the following 10 minutes. The plan is invalid but the repair process will not impact the actions currently being executed.

So we designed tests where, instead of running a simulation, we modify the problem being solved and try to repair it, on the *survivors* domain, called the *offline* setting. This allowed to run our test on bigger plans since no simulation is involved. Those experiments were run on an Intel X5670 processor running at 2.93Ghz with 24GB of RAM and a timeout of 10 minutes.

Detailed results between the *replan* and *repair* strategies for *offline* repair are presented on Fig. 3. The best *repair* algorithm is compared to the best *replan* algorithm. Each marker represents a different type of defect introduced. Square is for a change in the initial position of the robots. Cross is for a change in the position of one survivor. Circle is a new survivor that has to be rescued. As expected, the number of steps changed (on the right hand side of Fig. 3) is limited in case of reparation whereas replanning means that all steps can be modified, even the one that should not be impacted by the unexpected event. The use of reparation also allows to drastically diminish the number of plans explored (centre plot). This does not translate to big changes in the total time used during the resolution of the problem (left hand side, log scaled); it still reduces the planning time in the majority of cases.

VI. DISCUSSION

This paper presents a repair algorithm built on top of a hybrid planner. Results highlight that this algorithm can use the user-defined knowledge to repair more efficiently, especially in the offline setting. But common problems arising during execution with abstract action, namely half-executed abstract steps, were dealt with.

The presence of abstract actions in the plan gives some information to the repair process: it bundles actions that share the same goal together. Without it, the reparation must rely on causal links to know which steps should be removed when something fails. This strategy can work quite well when the robot is currently executing the part of the plan that has to be repaired. For instance if a step fails, there may exist a combination of steps that accomplishes the same task (taking another route if this is a motion step, sending another robot to

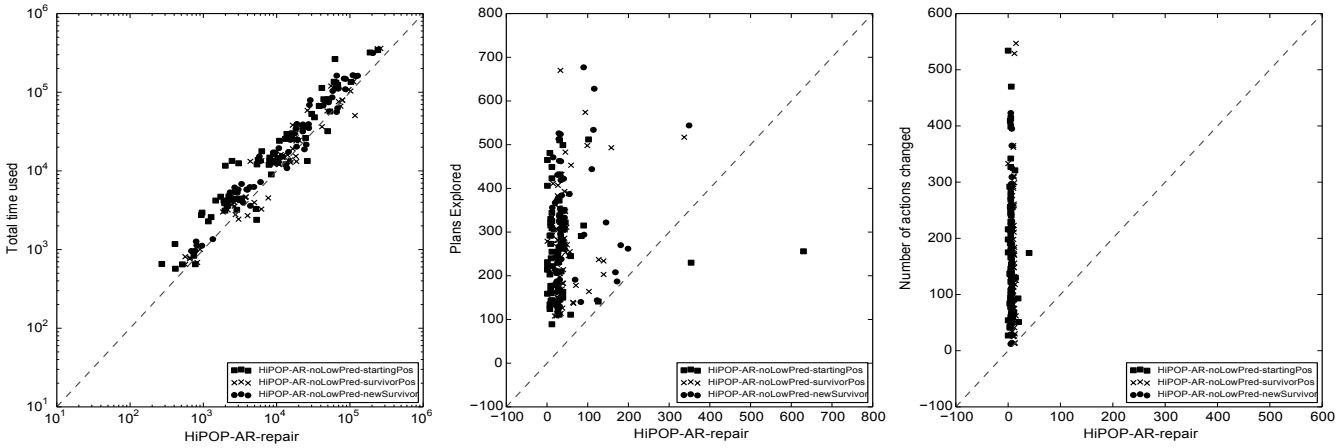

Fig. 3. Results on the offline setting. Each figure plots a metric from the best *replan* setting with respect to the best *repair* setting. From left to right: total time of the planning process in ms, number of plans explored, number of steps changed between the initial plan and the repaired plan. Each kind of unexpected event is associated with a different marker

do the task, etc.). But when the robot can foresee the problem in advance, the presence of abstract actions can be more useful. It gives information about what steps exactly were introduced to support the now obsolete steps, and remove all of them together (which is impossible if those enabling steps were already executed). This explains why the improvements in the offline setting are much more important. If during execution some events invalidate steps in the future, then the abstract information can be very valuable. But unexpected events usually impact both: on the current actions but also in the future. For instance if a path is blocked, it prevents the current move but also all future moves.

In all tests, we assume that robots were able to communicate together so the plan can be computed and executed by a single agent. This means that the repair algorithm is able to add steps for any robot and to remove any step from the plan. To deal with a real distributed architecture one would need to first merge the plans of all the robots able to communicate. Previous work [9] has shown that using POP is a good plan representation for merging plans but abstract actions still have to be taken into account. Then preventing the algorithm to add, delete or constrain the steps of a given set of agents would allow the remaining robots to plan for themselves, even if the communication is lost with the other group. And this algorithm has proven that it can deal with those "locked" steps. Work is under way to further demonstrate this aspect.

The results shown here only concern the reparation of a single defect, but for the execution of a full mission it is important to be able to iterate the execute/repair cycle until the end of the mission. First experiments show that this is possible with this algorithm, but as soon as the two algorithms repair the plan differently it is difficult to fairly compare them.

The next step will be to use a full-fledged robotic simulator to broaden the spectrum of defects that can be simulated, and to reduce the gap between our tests and real-life use of the architecture. This will lead to experiments with real robots in an outdoor setting for patrol missions.

REFERENCES

[1] L Barbulescu, Z B. Rubinstein, S F. Smith, D E. Wilkins, and T L. Zimmerman. Distributed Scheduling Agents for Disaster Response. *ICAPS SPARK Workshop*, pages 7–14, 2010.

[2] P Bechon, M Barbier, G Infantes, C Lesire, and V Vidal. HiPOP : Hierarchical Partial-Order Planning. *STAIRS*, 2014.

[3] J Bidot, B Schattenberg, and S Biundo. Plan repair in hybrid planning. *KI 2008: Advances in Artificial Intelligence*, 2008.

[4] R Dechter, I Meiri, and J Pearl. Temporal constraint networks. *Artificial intelligence*, 49(1):61–95, 1991.

[5] F Dvorák, R Barták, A Bit-Monnot, F Ingrand, and M Ghallab. Planning and Acting with Temporal and Hierarchical Decomposition Models. *PAIR*, 2014.

[6] K Erol, J Hendler, and D S Nau. HTN planning: Complexity and expressivity. In *AAAI*, pages 1123—-1128, 1994.

[7] M Fox, A Gerevini, D Long, and I Serina. Plan Stability: Replanning versus Plan Repair. *ICAPS*, 2006.

[8] M Fox and D Long. PDDL2. 1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*, 2003.

[9] M A Hashmi and A El Fallah Seghrouchni. Merging of Temporal Plans Supported by Plan Repairing. *ICTAI*, 2010.

[10] P Koch, C Robin, and A Degroote. Gladys: 0.2.5. January 2015.

[11] R V D Krogt and M De Weerdt. Plan Repair as an Extension of Planning. *ICAPS*, pages 161–170, 2005.

[12] N Kushmerick, S Hanks, and DS Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 1995.

[13] S M Lee-Urban. *Hierarchical Planning Knowledge for Refining Partial-Order Plans*. PhD thesis, Lehigh University, 2012.

[14] C McGann, F Py, K Rajan, H Thomas, R Henthorn, and R McEwen. A deliberative architecture for AUV control. In *ICRA*, pages 1049–1054. IEEE, 2008.

[15] H Mufioz-Avila and F Weberskirch. Planning for manufacturing workpieces by storing, indexing and replaying planning decisions. *AIPS*, 1996.

[16] N Muscettola, G A Dorais, C Fry, R Levinson, and C Plaunt. IDEA: Planning at the Core of Autonomous Reactive Agents. *IWPSS*, 2002.

[17] JS Penberthy and D Weld. UCPOP: A sound, complete, partial order planner for ADL. *KR*, 1992.

[18] MA Peot and DE Smith. Conditional nonlinear planning. *AIPS*, 1992.

[19] B Schattenberg. *Hybrid Planning And Scheduling*. PhD thesis, Ulm University, Institute of Artificial Intelligence, 2009.

[20] V Vidal and H Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170(3):298–335, 2006.

[21] I Warfield, C Hogg, S Lee-Urban, and H Munoz-Avila. Adaptation of Hierarchical Task Network Plans. *FLAIRS*, 2007.

[22] HLS Younes and RG Simmons. VHPOP: Versatile heuristic partial order planner. *JAIR*, 20:405–430, 2003.