Repairing Decision-theoretic Policies Using Goal-oriented Planning

Christoph Mies[1], Alexander Ferrein[2], and Gerhard Lakemeyer[2]

[1] Fraunhofer-Institut für Intelligente Analyse- und Informationssysteme IAIS
Schloss Birlinghoven, D-53754 St. Augustin
christoph.mies@iais.fraunhofer.de
[2] Knowledge-based Systems Group
RWTH Aachen University
Ahornstrasse 55, D-52056 Aachen
{ferrein,gerhard}@kbsg.rwth-aachen.de

Abstract. In this paper we address the problem of how decision-theoretic policies can be repaired. This work is motivated by observations made in robotic soccer where decision-theoretic policies become invalid due to small deviations during execution; and repairing might pay off compared to re-planning from scratch. Our policies are generated with ReadyLog, a derivative of Golog based on the situation calculus, which combines programming and planning for agents in dynamic domains. When an invalid policy is detected, the world state is transformed into a PDDL description and a state-of-the-art PDDL planner is deployed to calculate the repair plan.

1 Introduction

Using decision-theoretic (DT) planning for the behavior specification of a mobile robot offers some flexibility over hard-coded behavior programs. The reason is that decision-theoretic planning follows a more declarative approach rather than exactly describing what the robot should do in a particular world situation. The programmer equips the robot with a specification of the domain, a specification of the actions and their effects, and the planning algorithm chooses the optimal actions according to a background optimization theory which assigns a reward to world states. With this reward, certain world states are preferred over others and goal-directed behavior emerges. The theory behind DT planning is the theory of Markov Decision Processes (e.g. [1]). The MDP model allows for stochastic actions, the solution of such an MDP leads to a behavior policy which optimizes the expected cumulated reward over the states which were traversed during planning. An interesting approach is the integration of DT planning techniques into other existing robot programming languages. One of these approaches is the language DTGolog proposed in [2], which marries DT planning with the well-known robot programming framework Golog [3, 4]. They follow the idea to

combine decision-theoretic planning with explicit robot programming. The programmer has the control over the planning algorithm for example by restricting the state space for the search for a policy.

These techniques have been successfully deployed in dynamic real-time domains as well. In [5] it is shown how the GOLOG dialect READYLOG is used for formulating the behavior of soccer robots in the RoboCup [6] domain. In dynamic and uncertain domains like robotic soccer it turns out that policies become easily invalid at execution due to failing actions. The observation is that slight deviations in the execution can make the policy fail. Consider for example, when a ball should be intercepted and the robot does not have the ball in its gripper afterwards, or a move action where the robot deviated slightly from the target position. The result is, however, that the remainder policy cannot be executed anymore as preconditions for subsequent actions are violated. Nevertheless, often simple action sequences like $goto(x, y); turn(\theta); intercept\text{-}ball$ are the solution to the problem. In this paper we sketch our approach, how DT policies, which were computed with READYLOG, can be repaired. The rest of this paper is organized as follows. In Section 2 we briefly introduce the formalisms we use in our approach, namely the language READYLOG and the language PDDL. Section 3 addresses the execution monitoring for detecting when a policy has become inapplicable and outline the transformation of the world description between the situation calculus and PDDL. In Section 4 we show first results of the plan repair algorithm in simulated soccer as well as in the well-known Wumpus world. We conclude with Section 5, also discussing some related work there.

2 Planning Background

2.1 DT Planning in Readylog

READYLOG [5], a variant of GOLOG, is based on Reiter's variant of the situation calculus [4, 7], a second-order language for reasoning about actions and their effects. Changes in the world are only due to actions so that a situation is completely described by the history of actions starting in some initial situation. Properties of the world are described by *fluents*, which are situation-dependent predicates and functions. For each fluent the user defines a successor state axiom specifying precisely which value the fluent takes on after performing an action. These, together with precondition axioms for each action, axioms for the initial situation, foundational and unique names axioms, form a so-called *basic action theory* [4]. READYLOG integrates several extensions made to GOLOG like loops, conditionals and recursive procedures, but also less standard constructs like the nondeterministic choice of actions as well as extensions exist for dealing with continuous change and concurrency allowing for exogenous and sensing actions and probabilistic projections into the future, or decision-theoretic planning employing Markov Decision Processes (MDPs), into one agent programming framework [5]. In this paper we focus on repairing DT policies. From an input program, which leaves several choices open, READYLOG computes an optimal policy (cf. also [2]). The policy is a tree branching over so-called *nature's choices*. These choice points represent the different outcomes of a stochastic action. *Agent choice points* in the plan skeleton, on the other hand, are optimized

away when calculating the policy. An optimal policy π is then executed with READYLOG's run-time system. We refer to [8] for further details. Important for this work is to know that the planning process is done off-line, based on models of the world, while the execution of the policy naturally is on-line. Discrepancies between the model and the real execution might occur.

2.2 PDDL and SGPlan

The *Planning Domain Definition Language* (PDDL) is a family of formal standard languages to define planning domains, dating back to work by McDermott [9]. It is used as the description language for the bi-annual International Planning Competitions (IPC) and was since then further developed to meet the requirements of the planning competitions. Here, we basically use PDDL2.2. The most important language features of this PDDL-version are the following: (1) domain objects with types (2) basic actions and actions with conditional effects (3) fluents, which as opposed to READYLOG may only take numerical values (4) metrics which are used to measure the quality of a plan. A metric is defined in the problem definition and is a function over fluents. It can either be maximized or minimized. A PDDL *world state* is a collection of predicates that are *true* at certain time points. Using the closed world assumption all predicates not listed in a world state are assumed to be *false*. More details about PDDL can be found in [10]. For our system, we used the metric PDDL2.2 planner SGPLAN [11], which won the last two IPCs. The basic architecture of SGPLAN follows a hierarchical planning approach and decomposes the overall goal into several non-interfering sub-goals.

3 Policy Execution Monitoring and Policy Repair

3.1 Marking Possible Failures and Detecting Execution Flaws

As we pointed out, READYLOG distinguishes between an off-line mode for plan generation, and an on-line mode for executing the calculated policies and interact with the environment. Consequently, we can distinguish two classes of failures. One class contains failures which can be detected at planning time, failures of the other class can not. As we said earlier, READYLOG makes use of models during DT planning. These include (stochastic) action models together with their effects. The effect axiom (or successor state axioms, to be precise) of a primitive action in the situation calculus and with it in READYLOG describes how the world evolves from world situation to world situation due to actions. It is in general impossible to design these models in such a way that they are free of errors. It does happen that unforeseen action effects occur in reality. As this problem is inevitable, we thus need at least account for detecting these failures. We extend the previous concept of [8] of inserting special markers into the policy. Originally, for conditions φ occurring in loops or if-then statements a marker $\mathfrak{M}(\varphi, v)$ was inserted which stored the truth value of φ at planning time and allowed to re-evaluate it at execution time to compare the values. It could thus be detected, when a model assumption in conditions, test actions and loops

became invalid. We add markers allowing to detect unforeseen action outcomes and failing action preconditions.

While the failure marker are a means to mark possible sources for a policy to become invalid at planning time, during execution we need to monitor if a policy is still executable by re-evaluating marker conditions and comparing the conditions with their reference values taken during plan generation. As described in [8] with some special transition in the implementation of READYLOG is it relatively straight-forward to check these conditions. For space reasons we cannot introduce it formally here. As, in general, it is hard to decide automatically in which cases it might be useful to repair or to cancel the execution of the policy, we lay it into the hand of the system designer to decide under which condition a policy shall be repaired or not. To this end, we introduce the construct $guardEx(\varphi, \pi)$ to attach condition φ to a policy π which must hold during its execution. A violation of the condition given by φ should initiate a re-planning in the current situation. Concurrent to the policy execution, a monitoring loop checks whether the condition attached to a policy hold or not. Additionally, after each execution of an action the monitoring loop simulates the remainder of the current policy in order to check if it is still executable. Therefore, all outcomes of each action are projected starting at the current world state. Thereby all precondition axioms and conditions in the policy are validated and tested in the projected world state where they are to be applied. If any of these checks fails the policy is corrupt and is subject to repairing or re-planning. Summarizing, there are two ways how a policy repair is initiated: (1) a failure marker occurs during policy execution; and (2) a policy failure in the remainder policy was detected by simulating it.

3.2 Repairing Policies

In the previous section we showed how we could determine whether a policy is corrupt, i.e. it cannot be executed until its end for some reasons, and shall be repaired. Now assume a policy π that has become invalid due to a violation of the condition φ in $guardEx(\varphi, \pi)$. To repair the policy we have to conduct the following steps:

1. *Calculate the desired world state:* The desired world state is the world state in which the remainder policy becomes executable again. This state serves as the goal description in step 2. With the formal specification of the actions in READYLOG it is possible to compute the preconditions for the remainder policy to become executable again.
2. *Translate state and goal description to PDDL.* Next, we translate the desired world state into the goal description of PDDL. Furthermore, in order to be able to perform goal-directed planning, we need an initial state description; we therefore translate the current state to a PDDL description and use it as the initial state. The state description has to be as small as possible as the run-time of PDDL-planners is, in general, directly related to the number of ground actions. Therefore, we have to restrict our translation to the important and salient parts of our situation calculus world description.

- *Restrict to salient fluents.* The goal state should only refer to fluents that occur in the policy. Thus for the PDDL state translation, we can ignore all fluents that are not mentioned by the READYLOG policy.
- *Restrict fluent domains:* The idea is to translate only those values that are important to the remainder of the policy. A policy repair for a pass between player p_1 and player p_2 does not necessarily need anything to know about the positions of the other players. Therefore, we only translate p_1's and p_2's positions.

Note that in our current implementation, the translation between the situation calculus and PDDL has to be done by hand. For each pair of READYLOG and PDDL domain definitions, the domain axiomatizer has to provide a valid translation.

3. *Plan step.* The initial and goal state description are transmitted to an external planner via the file system, and the planning process is initiated. In our current implementation we make use of SGPlan. Note that, as we are using the abstract PDDL description, we can plug any other PDDL planner to our system easily.
4. *Re-translate the repair plan.* When a plan has been generated, we have to re-translate it to a READYLOG policy which can be executed then. After having calculated a PDDL repair plan with the external planner, it has to be executed before the remainder of the failed policy. Since PDDL is a deterministic language, no nondeterministic action can occur in the plan. Thus, each PDDL action can be translated to a sequence of READYLOG actions with specified outcome.

4 First Empirical Results

For proving the concept of the presented plan-repair scheme, we applied it to the toy domain WUMPUSWORLD, where an agent has to hunt a creature, the Wumpus, in a maze environment with pits and traps while searching for a pile of gold. The domain is modeled in a stochastic way, that is, the basic actions like move or grab_gold have stochastic effects, they can succeed or fail with certain probabilities. Moreover, exogenous events can occur. For example, the agent may lose the gold or move to a wrong position in the grid due to such events. In these cases, plan repair is invoked. The plans the agent performs are decision-theoretic policies with a plan horizon of three, meaning that the agent plans ahead the next three actions. In this paper we propose to connect READYLOG to an external PDDL planner. The first important question is whether or not plan repair pays off in the given application domain, as it was shown that, in general, plan repair is as complex as planning from scratch [12]. Therefore, we compared the runtimes of planning from scratch (DT) each time a policy became invalid with an iterative deepening depth-first brute-force planner (BF) in READYLOG itself (this is very similar to the comparison made in [13]). Next, we compared this to the PDDL plan repair scheme we propose in this paper to get information about the extra computational overhead to transfer the world states between READYLOG and the PDDL planner.

Table 1. Results of WumpusWorld.

Setup	*Run time*		*Plan. time*		*Repair time*	
	# runs	av. [s]	# plans	av. [s]	# repairs	av. [s]
STANDARD_PDDL	91	19.53	740	1.23	722	0.99
STANDARD_BF	87	13.95	713	1.25	676	0.12
STANDARD_NR	93	17.26	1213	1.27	N/A	
SMALL_PDDL	87	14.62	713	1.21	759	0.33
SMALL_BF	89	13.57	728	1.26	617	0.11
SMALL_NR	95	16.58	1251	1.23	N/A	
HORIZON_PDDL	90	58.71	612	7.51	629	0.97
HORIZON_BF	89	53.13	606	8.21	704	0.12
HORIZON_NR	91	92.85	921	9.11	N/A	

The results are shown in Tab. 1. We defined three evaluation setups STANDARD, SMALL and HORIZON. The scenarios where we applied the PDDL repair mechanism are suffixed with _PDDL, the ones with brute-force planning are suffixed with _BF and the ones without repairing with _NR (no repair). The first column represents the number of successful runs (out of 100 runs in total) and their average run-time. A run is successful when the agent arrives at the target cell with the gold in its hands. The second column contains the number of generated policies together with their average computation time; the number of calls of the repairing method as well as their average computation times are presented in the last column. All run-times are measured in real elapsed seconds. Note that there is no repairing data available if the plan repair was not performed (the _NR cases). Therefore, the number of generated plans for these scenarios roughly equals the number of repairs plus plans in the other cases. For the first row of the table, this means that each of the 91 successful runs took on average 19.53 seconds. In total, 740 policies have been generated, each of which took 1.23 second on average to be computed. 722 times the repair routine was called due to failing plans. On average, it took 0.99 seconds to establish the successful repair plan. Note that the number of repairs can exceed the number of plans. This is the case when a repaired plan fails again. The scenario STANDARD is the starting point for our evaluation. The DT planning depth is 3 in that setup. Thus, the lengths of the DT plans and the repair plans are roughly equal. The average computation time for PDDL repair (column 3) is slightly lower than the DT planning time (column 1). The average run-time of STANDARD_PDDL is higher than the one of STANDARD_DT. The advantage is taken by the overhead of the repair mechanism which is mostly determined by the policy execution simulation. The setup SMALL equals STANDARD except the smaller size of the WumpusWorld. It contains 54 cells, 8 walls and 2 holes whereas STANDARD contains 180 cells, 16 walls and 2 holes. This leads to a significant faster PDDL repairing w.r.t. STANDARD_PDDL. The gap between repairing and DT re-planning is large enough to be faster than the pure DT planning approach SMALL_NR, which does not change in comparison to STANDARD_NR. The times of SMALL_BF are also very similar to the ones of STANDARD_BF. With the last setup, HORIZON, we wanted to check the influence of an increased plan horizon to the overall

run-times. It has an increased plan horizon of 4 instead of 3 as in `STANDARD` and `SMALL`. This leads to a significantly higher DT planning time since our method has an exponential run-time w.r.t. the horizon. Therefore, `HORIZON_PDDL` is much faster than its pure DT planning companion. Again, the times of the brute-force approach are not influenced by this scenario variation. The times for plan repair for PDDL and BF equal those with horizon 3. As one can see from these results, the brute-force method is superior to PDDL in all WumpusWorld scenarios. The reason lies in the simple structure of the scenario. The computation time of the brute-force method depends on the number of ground actions. There exists only 9 ground actions (4 move actions, 4 shoot actions and grab_gold). Another issue is the very preliminary interface between Readylog and the PDDL planner. We make use of the file system to communicate the world states between PDDL and Readylog, the brute-force planner is integrated into Readylog. There is a connection between the simplicity of the action description and the run-time of the brute-force method. To check this, we applied the method to a more complex and realistic domain, the simulated soccer domain. Two teams of 11 agents play soccer against each other. The available actions are move, pass, dribble, score and intercept. Each action may take arguments. Thus, the number of ground actions (where the argument variables are substituted by values) is exponential in the size of the domain. This complexity enables the PDDL repairing to dominate the brute-force repairing due to SGPlan's superior search heuristics. Our scenario is as follows: an agent has to intercept the ball and to dribble to the opponent's goal. Thereby, the dribble path is planned, i.e. the field is divided into rectangular cells and the agent has to dribble from one cell to an adjacent one. We then artificially cause the policy execution to fail after some while in order to force policy repair. For a DT planing horizon of 2, i.e. the policy consists of one intercept and one dribble action, DT planning takes 0.027 and PDDL repairing 0.138 seconds on average. If the horizon is 3, the DT planning time increases to 0.302 seconds whereas the PDDL repairing time does not change. Thus, repairing with PDDL is faster than re-planning in this scenario. This confirms the evaluation of `HORIZON`. But in contrast to WumpusWorld, the BF repairing takes 0.305 seconds on average and is slower than the PDDL method. This gives more evidence to our observation that the performance of BF is related to the complexity of the action description. Our results show that plan repair in principle is useful in dynamic domains. We take these preliminary results as the starting point for future investigations.

5 Conclusions

In this paper we sketched our plan repair framework, where invalid DT policies generated by Readylog are repaired using an external PDDL planner. To this end, it must be detected when a policy becomes invalid, and how the world state must look like in order to continue the remainder policy. Then, we map our Readylog world state to a PDDL description and calculate a repair plan with an external PDDL planner. In our current implementation we use SGPlan. The repair plan is then translated back into the Readylog framework and executed before the remainder policy is continued. Although it was shown by Nebel and

Koehler [12] that plan repair is at least as hard as planning from scratch, it turns out that plan repair works in practice under certain circumstances. In our case, the assumption is that only slight deviations in the execution make our policy invalid. Thus, only simple repair plans are needed to reach a state where the remainder policy can be executed again. A similar approach was proposed in [14], which integrate monitoring into a GOLOG-like language with a transition semantics similar to that of READYLOG. Their system also performs a monitoring step after each action execution and performs repair actions before the remainder of the failed policy in order to restore its executability. A similar idea, to combine GOLOG with an external planning system based on a PDDL description, was proposed in [13]. For the future work, we plan to combine their results with ours.

Acknowledgments

This work was supported by the German National Science Foundation (DFG) in the Priority Program 1125, *Cooperating Teams of Mobile Robots in Dynamic Environments*. We would like to thank the anonymous reviewers for their helpful comments.

References

1. Puterman, M.: Markov Decision Processes: Discrete Dynamic Programming. Wiley, New York (1994)
2. Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S.: Decision-theoretic, high-level agent programming in the situation calculus. In: Proc. AAAI-00, AAAI Press (2000) 355–362
3. Levesque, H., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming **31**(1-3) (1997) 59–83
4. Reiter, R.: Knowledge in Action. MIT Press (2001)
5. Ferrein, A., Fritz, C., Lakemeyer, G.: Using golog for deliberation and team coordination in robotic soccer. KI **19**(1) (2005) 24–30
6. RoboCup: The Robocup Federation. http://www.robocup.org (2007)
7. McCarthy, J.: Situations, Actions and Causal Laws. Technical report, Stanford University (1963)
8. Ferrein, A., Fritz, C., Lakemeyer, G.: On-line Decision-Theoretic Golog for Unpredictable Domains. In: Proc. KI-04. (2004)
9. McDermott, D.: PDDL – The planning Domain Definition Language, Version 1.2. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
10. Edelkamp, S., Hoffmann, J.: PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Institute für Informatik (December 2003)
11. Hsu, W., Wah, B., Huang, R., Chen, Y.: New Features in SGPlan for Handling Soft Constraints and Goal Preferences in PDDL3.0. Proc. ICAPS-06 (June 2006)
12. Nebel, B., Koehler, J.: Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis. Technical Report RR-93-33, DFKI GmbH (1993)
13. Claßen, J., Eyerich, P., Lakemeyer, G., Nebel, B.: Towards an integration of golog and planning. In: Proc. IJCAI-07, AAAI Press (2007)
14. Giacomo, G.D., Reiter, R., Soutchanski, M.: Execution Monitoring of High-Level Robot Programs. In: In Proc. KR-98. (1998) 453–465