Repairing Decision-theoretic Policies Using Goal-oriented Planning

Ch. Mies [1] and **A. Ferrein** [2] and **G. Lakemeyer** [2]

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 plan repair might pay off rather than replan 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 was 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. [2, 3, 28]). 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. DT planning has been deployed for the decision making in a variety of robotic applications. 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 [5], which marries DT planning with the well-known robot programming framework Golog [24, 29]. 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 [17, 15], the Golog dialect Readylog is proposed which is well-suited to formulate the behavior of soccer robots in the RoboCup. Besides support for continuous change and probabilistic projections, also DT planning as proposed in [5] is integrated into their Readylog interpreter. Readylog was used for controlling soccer robots in the past [12, 17], but turned out useful also for the abstract behavior specification of soccer agents [13]. In dynamic domains like robotic soccer it turns out that policies become easily invalid during execution due to failing actions. The observation is that slight deviation 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 describe an approach, how policies can be repaired. The rest of this paper is organized as follows. In Section 2 we briefly elucidate 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. In Section 4 we show our transformation of the world description between the situation calculus and PDDL as well as the repairing itself. Section 5 presents first results of the plan repair algorithm in simulated soccer as well as in the well-known Wumpus world. We conclude with Section 6 discussing some related work there.

2 Planning Background

2.1 DT Planning in Readylog

Readylog [16, 17], a variant of Golog, is based on Reiter's variant of the situation calculus [29, 25], 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* [29].

[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

READYLOG integrates several extensions made to GOLOG like loops and conditionals [24], recursive procedures, but also less standard constructs like the nondeterministic choice of actions as well as extensions exist for dealing with continuous change [22] and concurrency [10], allowing for exogenous and sensing actions [8] and probabilistic projections into the future [21], or decision-theoretic planning [5] employing Markov Decision Processes (MDPs), into one agent programming framework [16, 17, 4]. For specifying the behaviors of an agent or robot the following constructs exist: (1) sequence $(a; b)$, (2) nondeterministic choice between actions $(a|b)$, (3) solve an MDP ($solve(p, h)$, p is a GOLOG program, h is the MDP's solution horizon), (4) test actions ($?(c)$), (5) event-interrupt ($waitFor(c)$), (6) conditionals ($if(c, a_1, a_2)$), (7) loops ($while(c, a_1)$), (8) condition-bounded execution ($withCtrl(c, a_1)$), (9) concurrent execution of programs ($pconc(p_1, p_2)$), (10) stochastic actions, (11) probabilistic (offline) projection ($pproj(c, a_1)$), and (12) procedures ($proc(name(parameters), body)$) to name the most important ones.

READYLOG uses a transition semantics as proposed by [10] with CONGOLOG. The program is interpreted in a step-by-step fashion where a transition relation defines the transformation from one step to another. In this transition semantics a program is interpreted from one configuration $\langle \sigma, s \rangle$, a program σ in a situation s, to another configuration $\langle \delta, s' \rangle$ which results after executing the first action of σ, where δ is the remaining program and s' the situation resulting of the execution of the first action of σ. The one-step transition function *Trans* defines the successor configuration for each program construct. In addition, another predicate *Final* is needed to characterize final configurations, which are those where a program is allowed to legally terminate.

To illustrate the transition semantics, let us consider the definition of *Trans* for some of the language constructs:

1. $Trans(nil, s, \delta, s') \equiv false$
2. $Trans(\alpha, s, \delta, s') \equiv Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s)$
3. $Trans([\delta_1; \delta_2], s, \delta, s') \equiv Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta, s') \vee \exists \delta'. \delta = (\delta'; \delta_2) \wedge Trans(\delta_1, s, \delta', s)$
4. $Trans(\sigma_1 \,||\, \sigma_2, s, \delta, s') \equiv \exists \gamma. \delta = (\gamma \,||\, \sigma_2) \wedge Trans(\sigma_1, s, \gamma, s') \vee \exists \gamma. \delta(\sigma_1 \,||\, \gamma) \wedge Trans(\sigma_2, s, \gamma, s')$

1. Here *nil* is the empty program, which does not admit any further transitions.
2. For a *primitive action* α we first test if its precondition holds. The successor configuration is $\langle nil, do(\alpha, s) \rangle$, that is, executing α leads to a new situation $do(\alpha, s)$ with the *nil* program remaining.
3. The next definition concerns an action sequence $[\delta_1; \delta_2]$, where it is checked whether the first program is already final and a transition exists for the second program δ_2, otherwise a transition of δ_1 is taken.
4. $\sigma_1 \,||\, \sigma_2$ denotes that σ_1 and σ_2 can be executed concurrently. Here the definition of $Trans$ makes sure that one of the two programs is allowed to make a transition without specifying which. This corresponds to the usual interleaved semantics of concurrency.

We only sketched the transition semantics here. For a concise overview of the transition semantics we refer the interested reader for example to [10, 9]. Note that the transition semantics allows for a natural integration of sensing and online execution of programs.

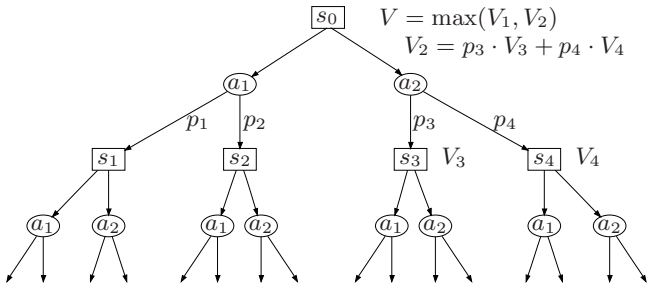

Figure 1. Decision tree search in READYLOG

In this paper we focus on repairing DT policies. Therefore, we discuss in little more detail, how decision-theoretic planning is conducted in the READYLOG framework. From an input program, which leaves several choices open, the forward-search value iteration algorithm, which was originally proposed in [5], computes an optimal policy. We call the input programs plan skeletons. Fig. 1 illustrates the planning. For each stochastic action outcome, the algorithm inserts a new branch and annotates the tree with the probability of occurrence of this outcome. When the whole tree of possible actions is spanned, the value at each branch point is calculated as shown in the figure. The different outcomes of stochastic actions are called *nature's choice points* and are all represented in the final policy, which is a tree branching over nature's choices. *Agent choice points* in the plan skeleton are optimized away when calculating the policy. At each agent choice point the best choice is taken according to the value of the respective branch.

An optimal policy π is then executed with READYLOG's runtime system. We refer to [16] 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. To detect these differences, in [16] it was proposed to introduce special markers into the policy. With these markers one can easily detect, when a policy becomes invalid and can no longer be executed. We want to stress that these discrepancies between the MDP model in READYLOG and the real execution trace occurs because our models are in general not precise enough. We cannot foresee and model all possible outcomes that can happen. Against this backdrop, we are using fully observable MDPs as an idealized model.

We will show in Section 3 how the concept of markers works formally and how it can be detected when a policy has become invalid during execution.

2.2 PDDL and SGPlan

The *Planning Domain Definition Language* (PDDL) is a family of formal standard languages to define planning domains. It was first proposed by McDermott in 1998 [26]. It is used as the description language for the bi-annual International Plan-

ning Competitions (IPC) and was since then further developed to meet the requirements of the planning competitions. In this work, we basically use PDDL2.2. The most important language features of this PDDL-version are the following: (1) basic actions; (2) objects with types; (3) conditional action effects; (4) fluents; (5) metrics.

The *basic actions* have a STRIPS style, and are used to define the possible actions. An action definition consists of a precondition formula stating at which world states the action is applicable, the effect formula defining the action's effect to the world when applied, and the action parameters, which is a list of all used variables. For pruning the search tree, it is important that objects have certain *types*. Only objects of the correct type may be unified with the variables or predicates. There exists a default super-type *object* in PDDL. *Conditional action effects* are a huge help in defining actions. The construct (`when` : φ_{Pre} φ_{Effect}) defines such an effect. If φ_{Pre} holds when the action is applied, the effect φ_{Effect} is added. Nothing is changed otherwise. In PDDL planning systems, such an action a is usually transformed into two actions a_1 and a_2, where the precondition of a_i is the one of a combined with φ_{Pre} or its negation, respectively. The effect is combined with φ_{Effect} if φ_{Pre} was added to the precondition, e.g. the if-condition was fulfilled. Unlike READYLOG, *fluents* only may take numerical values. Due to the object-oriented approach of PDDL, which especially maintains a finite state space, it is not possible to have numerical values as arguments of predicates. Fluents are defined with the keyword `function`. There exist five ways to change a fluent's value by applying one of the following actions: `increase`, `decrease`, `scale-up` and `scale-down` for relative assignments and `assign` for absolute assignment. *Metrics* 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 $false$. The first important time point is the initial state. When PDDL-subsets without time representation (as PDDL2.2) are considered, other important time points are set after actions are applied. More details about PDDL can be found in [18, 14, 19].

For our system, we used the metric PDDL2.2 planner SGPLAN, which won the last two IPC's in the metric domain, which is the one of our interest. The basic architecture of SGPLAN follows a hierarchical planning approach and decomposes the overall goal into several sub-goals. These are ordered such that no sub-goal interferes with another one. Then the sub-goals are planned on. In many cases, not all actions are needed and so SGPLAN can prune the search tree by not considering unused actions resulting in a substantially smaller search space. The sub-plans, fulfilling the sub-goals, are then used to build a plan solving the entire planning problem. A more detailed description of SGPLAN is given in [6] and [23].

3 Policy Execution Monitoring

As we pointed out, READYLOG distinguishes between an off-line mode, in which the interpreter is switched during plan generation, and an on-line mode for executing the calculated policies and interacting with the environment. Consequently, there are also two classes of failures. One class contains failures that can be predicted already at planning time, failures of the other class can only be detected at execution time. In previous work [16] it was shown how discrepancies between plan and execution could be detected by introducing markers into the policy which keep track of the model assumptions made during plan generation. The policy was discarded in these cases and re-planning was initiated. We restate the idea of [16] in the following to give the basic idea of annotating policies with markers as we extend the marker concept for detecting failures at planning time in Section 3.2. Run-time failures, or flaws as we call them, need a different treatment. We show how flaws could be detected in Section 3.3.

3.1 Introduction

As we have stated in Section 2, Readylog uses a special operator $solve(h, f, p)$ for a program p, a reward function f, and a fixed horizon h, which initiates decision-theoretic planning in the on-line transition semantics (cf. also [15]).

$$\begin{aligned} &Trans(\mathbf{solve}(h, f, p), s, \delta, s') \equiv \\ &\quad \exists \pi, v, pr.BestDo(p, s, h, \pi, v, pr, f) \wedge \\ &\quad \delta = applyPol(\pi) \wedge s' = s \end{aligned}$$

The predicate *BestDo* first calculates the policy for the whole program p. The policy π is then scheduled for on-line execution as the remaining program. In the case of the robot's position, for example, policy generation works with an abstract model of the robot's movements so that robot positions in future states can simply be computed without having to appeal to actual sensing. Making use of such models during plan generation requires that we monitor whether π remains valid during execution as discrepancies between the model and the real-word situation might arise. Monitoring is handled within special *applyPol* transitions, whose definition we omit here. Note that the *solve* statement never reaches a final configuration as further transitions are needed to execute the calculated policy. To keep track of the model assumptions we made during planning, we introduce special markers into the policy. Hence, in the definition of *BestDo* we have to store the truth values of logical formulas. For conditionals this means:

$$\begin{aligned} &BestDo(\mathbf{if}\ \varphi\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2\ \mathbf{endif}; p, s, h, \pi, v, pr) \stackrel{def}{=} \\ &\quad \varphi[s] \wedge \exists \pi_1.BestDo(p_1; p, s, h, \pi_1, pr) \wedge \\ &\qquad \pi = \mathfrak{M}(\varphi, true); \pi_1 \vee \\ &\quad \neg\varphi[s] \wedge \exists \pi_2.BestDo(p_2; p, s, h, \pi_2, v, pr) \wedge \\ &\qquad \pi = \mathfrak{M}(\varphi, false); \pi_2 \end{aligned}$$

Thus, for conditionals we introduce a marker into the policy that keeps track of the truth value of the loop condition. We prefix the generated policy with a marker $\mathfrak{M}(\varphi, true)$ in case φ turned out to be true in s and $\mathfrak{M}(\varphi, false)$ if it is false. While-loops are treated in a similar way. The treatment of a test action $\varphi?$ is even simpler, since only the case where φ is true matters. If φ is false, the current branch of the policy is terminated, which is indicated by the *nil* poliy, i.e. the empty

policy.

$$\begin{aligned}
&BestDo(\varphi?; p, s, h, \pi, v, pr) \stackrel{def}{=} \\
&\quad \varphi[s] \wedge \exists \pi'. BestDo(p, s, h, \pi', v, pr) \wedge \\
&\quad\quad \pi = \mathfrak{M}(\varphi, true); \pi' \vee \\
&\quad \neg\varphi[s] \wedge \pi = nil \wedge pr = 0 \wedge v = reward(s)
\end{aligned}$$

With keeping track of the model assumption in conditions this way, it is easy to to detect when a policy became invalid due to discrepancies between plan and execution time.

3.2 Marking Possible Failures during Planning

As already stated above possible failures of program conditions φ which occur during policy execution can be detected by markers of the form $\mathfrak{M}(\varphi, v)$. The results of tests, conditionals and while loops can thus be validated during execution. For the complete definition of the semantics, we refer to [15].

Another source of a failure is when action preconditions are not fulfilled. In our previous work [16], these cases were not treated explicitly. In the case where none of the action choices were possible, the resulting policy simply was the empty policy. As the empty policy directly terminates, policy execution directly terminated. It was possible to exploit this behavior, as we did not need to know the reason for the policy to terminate. However now, when we try to repair a policy, we need to know when a policy cannot be executed any longer due to failing action preconditions. Thus, we introduce the marker $\mathfrak{M}(notPoss)$ into the policy when action a was not executable at planning time, i.e.

$$\begin{aligned}
&BestDo(a; p, s, h, \pi, v, pr) \stackrel{def}{=} \\
&\quad Poss(a, s) \wedge \exists \pi', v', pr'. BestDo(p, a; s, h-1, \pi', v', pr') \\
&\quad\quad \wedge \pi = a; \pi' \wedge v = reward(s) + v' \wedge pr = pr' \vee \\
&\quad \neg Poss(a, s) \wedge \pi = \mathfrak{M}(notPoss) \wedge pr = 0 \wedge v = 0
\end{aligned}$$

In the case of a stochastic action, the predicate $BestDoAux$ with the set of all outcomes for this stochastic action is expanded. $choice'(a) \stackrel{def}{=} \{n_1, \ldots, n_k\}$ is an abbreviation for the outcomes of the stochastic action a.

$$\begin{aligned}
&BestDo(a; p, s, h, \pi, v, pr) \stackrel{def}{=} \\
&\quad \exists \pi', v'. BestDoAux(choice'(a), a, p, s, h, \pi', v', pr) \wedge \\
&\quad \pi' = a; senseEffect(a); \pi' \wedge v = reward(s) + v'
\end{aligned}$$

The resulting policy is $a; senseEffect(a); \pi'$. The pseudo action *senseEffect* is introduced to fulfill the requirement of full observability. For similar reasons, predicates $senseCond(\varphi, n)$ need to be defined for each action outcome to find out which outcome occurred when executing the stochastic action (see e.g. [15] for a thorough discussion). The remainder policy π' branches over the possible outcomes and the agent must be enabled to sense the state it is in after having executed this action. The remainder policy is evaluated using the predicate $BestDoAux$. The predicate $BestDoAux$ for the (base) case that there is one outcome is defined as

$$\begin{aligned}
&BestDoAux(\{n_k\}, a, \delta, s, h, \pi, v, pr) \stackrel{def}{=} \\
&\quad \neg Poss(n_k, s) \wedge \pi = \mathfrak{M}(notPoss) \wedge \\
&\quad\quad v = 0 \wedge pr = 0 \vee \\
&\quad Poss(n_k, s) \wedge senseCond(n_k, \varphi_k) \wedge \\
&\quad\quad (\exists \pi', v'. pr'). BestDo(\delta, do(n_k, s), h, \pi', v', pr') \wedge \\
&\quad\quad \pi = \varphi_k?; \pi' \wedge v = v' \cdot prob(n_k, a, s) \wedge \\
&\quad\quad\quad pr = pr' \cdot prob(n_k, a, s)
\end{aligned}$$

If the outcome action is not possible, we insert the marker $\mathfrak{M}(notPoss)$ indicating that the outcome action n_k is not possible. Otherwise, if the current outcome action is possible, the remainder policy π' for the remaining program is calculated. The policy π consists of a test action on the condition φ_k from the *senseCond* predicate with the remainder policy π' attached. The case for more than one remaining outcome action is defined as

$$\begin{aligned}
&BestDoAux(\{n_1, \ldots, n_k\}, a, p, s, h, \pi, v, pr) \stackrel{def}{=} \\
&\quad \neg senseCond(\{n_1, \ldots, n_k\}) \wedge \pi = \mathfrak{M}(unexpAct) \wedge \\
&\quad\quad v = 0 \wedge pr = 0 \vee \\
&\quad \neg Poss(n_1, s) \wedge BestDoAux(\{n_2, \ldots, n_k\}, p, s, h, \pi, v, pr) \vee \\
&\quad Poss(n_1, s) \wedge (\exists \pi', v', pr'). \\
&\quad\quad BestDoAux(\{n_2, \ldots, n_k\}, p, s, h, \pi', v', pr') \wedge \\
&\quad \exists \pi_1, v_1, pr_1. BestDo(p, do(n_1, s), h-1, \pi_1, v_1, pr_1) \wedge \\
&\quad senseCond(n_1, \varphi_1) \wedge \pi = \textbf{if } \varphi_1 \textbf{ then } \pi_1 \textbf{ else } \pi' \textbf{ endif} \wedge \\
&\quad v = v' + v_1 \cdot prob(n_1, a, s) \wedge pr = pr' + p_1 \cdot prob(n_1, a, s)
\end{aligned}$$

The difference to the previous $BestDoAux$ predicate is that the other outcomes are recursively interpreted, and that the resulting policy now consists of a conditional instead of a test action as in the previous case. The value for the outcome is clearly the value of the remaining policy which has n_1 as prefix weighted by the probability of occurrence of n_1 plus the value gathered by the other possible outcomes. Similarly the probability of success pr is calculated. This is very much conform with the definition of stochastic actions as given in [16, 15]. New in the definition is the first disjunction, where $\neg senseCond(\{n_1, \ldots, n_k\})$ is an abbreviation for the formula $senseCond(n_1, \varphi_1) \wedge \cdots \wedge senseCond(n_k, \varphi_k) \wedge \neg\varphi_1 \wedge \cdots \wedge \neg\varphi_k$. This formula checks, if there are loopholes in the definition of the sense conditions. In general, the sense conditions of a stochastic action need to be modeled in a way that exactly one of the modeled outcome occur when executing the action. With the special marker $\mathfrak{M}(unexpAct)$ we can detect if an not-modeled action outcome was observed.

By now, we have defined several kinds of possible policy failures and their markers. If we consider a policy π being constructed this way, several markers can be inserted into the policy. Regarding π as a tree branching in nature's choice points as shown in Fig. 1, it might happen that all children in a sub-tree are marked with failure markers. In this case the whole sub-tree is not executable and can be skipped. To detect this as early as possible, we also can mark the parent with a failure marker.

If the system faces a violated condition marker, a repair is

initiated. The transition semantics therefore is as follows:

$$\begin{aligned}Trans(applyPol(\mathfrak{M}(\varphi,v);\pi),s,\delta,s') \equiv s=s' \wedge \\ \neg violated(\varphi,v,s) \wedge \delta = applyPol(\pi) \vee \\ violated(\varphi,v,s) \wedge \exists\pi'.repair(s,\pi') \wedge \delta = applyPol(\pi') \vee \\ \neg\exists\pi'.repair(s,\pi') \wedge \delta = nil\end{aligned}$$

The term $violated(\varphi, v, s)$ is an abbreviation for the formula $\phi[s] \wedge v = false \vee \neg\phi[s] \wedge v = true$ denoting a violation of the condition marker $\mathfrak{M}(\varphi, v)$. Note that the $repair(s, \pi)$ predicate succeeds if the policy currently being executed can be repaired in situation s. The main idea is that a repair plan is executed in order to establish a world state in which the remainder of the current policy can be executed. $repair(s, \pi)$ succeeds if such a plan has been calculated. The policy π consists of the repair plan as well as the remainder of the original policy. Note that it might happen that such a repair plan cannot be calculated. In these cases $repair(s, \pi)$ fails and a re-planning is initiated by instantiating δ with the *nil* program. For further details of the repairing process we refer to to Section 4. If a $\mathfrak{M}(notPoss)$ is faced by *Trans* a repair is initiated directly:

$$\begin{aligned}Trans(applyPol(notPoss;\pi),s,\delta,s') \equiv s=s' \wedge \\ \exists\pi'.repair(s,\pi') \wedge \delta = applyPol(\pi') \vee \\ \neg\exists\pi'.repair(s,\pi') \wedge \delta = nil\end{aligned}$$

Note that all other failure markers treated analogously.

3.3 Detecting Execution Flaws

While the failure marker presented in the previous section are a means to mark possible sources for a policy to become invalid, 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. But with this procedure we still have no means to decide when a policy is worth to try and repair it. One option is not to repair at all, though we claim and show later that under certain circumstances a plan repair can pay off. Another option is to try to repair every policy which has become invalid. Consider, for example, a robot accomplishing a policy for attacking the opponent goal. His policy will include actions like *intercept-ball*, *dribble*, and *shoot*. During the journey towards the opponent goal it might be that the robot loses the ball. With a simple intercept action control over the ball could be regained and the remainder of the policy becomes executable again. On the other hand, if the robot cannot shoot because of a hardware failure and due to that the policy to score may become invalid as some preconditions of the shoot action are violated, a plan repair seems senseless. As, in general, it is hard to decide automatically in which cases it might be useful to try and 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 constructs $guardEx(\varphi, \pi, repair)$ and $guardEx(\psi, \pi, replan)$ to attach conditions φ, or ψ resp., to a policy π which must hold during the execution of π. In the former case a violation of the condition given by φ should invoke a repair action, while the latter states that if ψ does not hold, a re-planning in the current situation should be initiated. Concurrent to the policy execution, our monitoring loop checks whether the conditions attached to a policy hold or not.

$$\begin{aligned}Trans(guardEx(\psi,\delta,replan),s,\delta,s') \equiv \\ \psi[s] \wedge \exists\delta'.Trans(\delta,s,\delta',s') \wedge \\ \delta = guardEx(\psi,\delta',replan) \vee \\ \neg\psi[s] \wedge s=s' \wedge \delta = nil\end{aligned}$$

If ψ holds, the next step of $Trans$ is performed before the next check of ψ is done recursively. Otherwise, the execution terminates by executing the *nil* program. This then results in the policy to fail, and a re-planning will be initiated by the robot control program. The case of plan repair is defined as

$$\begin{aligned}Trans(guardEx(\psi,\delta,repair),s,\delta,s') \equiv \\ \psi[s] \wedge \exists\delta'.Trans(\delta,s,\delta',s') \wedge \\ \delta = guardEx(\psi,\delta',repair) \vee \\ \neg\psi[s] \wedge \exists\pi.repair(s,\pi) \wedge s=s' \wedge \\ \delta = guardEx(\psi,applyPol(\pi),repair) \vee \\ \neg\psi[s] \wedge \neg\exists\pi.repair(s,\pi) \wedge s=s' \wedge \delta = nil\end{aligned}$$

If the condition holds, the policy is executed one step by $Trans$ and then ψ is checked again. Otherwise, a repair is initiated by $repair(s, \pi)$ resulting in a new policy π that is executed afterwards. If the repairing is not possible, the whole execution stops by the *nil* program. Note that ψ will always fail during the execution of the transition policy, i.e. the repair plan, as long as the condition is not established in the world again. Therefore, the check of ψ is aborted as a side effect of the repairing process. 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 exists three ways how a policy repair is initiated: (1) a failure marker occurs during policy execution; (2) a repairable policy constraint is violated; and (3) a policy failure in the remainder policy was detected by simulating it.

4 Repairing Policies

In the previous section we showed how we could determine whether or not 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 $guardEx(\varphi, \pi, repair)$. 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 the next step.
2. *Translate state and goal description to PDDL.* Next, we translate the desired world state into the goal description

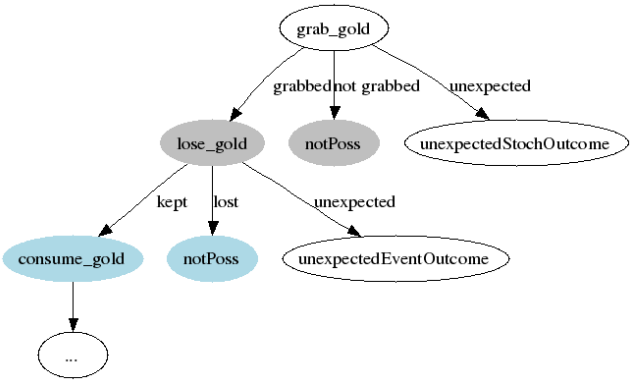

Figure 2. Example of a Desired Reference World State.

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.

3. *Plan step.* The initial and goal state description are transmitted to an external planner via 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.

4.1 Desired Reference World State

With our execution monitoring facility which checks after each action whether the policy constraints hold or not, we know after which action the policy has become invalid. In order to continue its execution we have to restore a world state in which its remainder is executable. As mentioned above the execution of the last action a caused the policy π to fail. Thus, we restore the world state that would have been valid after a's execution. We can calculate the desired reference world state s^* by projecting the world state s valid before a's execution by a, i.e. $s^* = do(a, s)$ with a is a primitive action.

If a was a stochastic action, we assume that the outcome gaining the highest reward is the most desirable and restore a world state where the branch leading to this rewards becomes applicable. This is a heuristic in order to calculate exactly one reference world state since PDDL is not able to deal with several goal states. Figure 2 shows an example for reference world states. When a policy execution failure is detected at blue or gray node labeled with *notPoss*, its same-colored sibling shows its desired world state.

Before starting planning, we store the current world state, that is, we store all fluent values of the current situation s. During the execution of the policy, we log all performed actions and the occurred outcomes of stochastic actions and events. We call this log the *policy execution trace*. By projecting this trace from s we calculate the reference situation according to the current real world state, i.e. the situation that was valid during planning exactly for the current course of actions. By modifying the last entry of the policy execution trace to be the highest valued successor of the last executed action as explained above we calculate the execution trace leading to the desired reference world state. We want to emphasize that the idea to take the highest-valued successor is an heuristic and can be sub-optimal. But we need to decide for exactly one desired reference world state since PDDL cannot deal with several goal states.

Note that the remainder of π is executable in s^* since it was part of the initial plan. If we find an action sequence to reach s^*, we can execute the remainder policy and the policy repair works out. It may happen on the other hand that s^* cannot be reached from the current world state. Then the policy repair fails.

4.2 Transforming Golog Situations to PDDL States

In the next step of our repair algorithm, we have to construct a PDDL description of our planning problem. As initial PDDL state, we have to transform the current world state. The PDDL goal state is the translation of the desired reference world state defined in the last section. The resulting PDDL plan can then be translated back to ReadyLog. If it is executed before the remainder of the current policy, it is executable again. The state description has to consist of as less atoms 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 needs 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 that values that are important to the remainder of the policy. We introduced this kind of pruning for more complex domains like simulated robot soccer. For example, a policy for a pass between player p_1 and player p_2 does not necessarily need to know anything about the positions of the other players. Therefore, we only translate p_1's and p_2's positions.

4.3 Translating Back Plans to Policies

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 actions can occur in the plan. Thus, each PDDL action can be translated to a sequence of ReadyLog actions with specified outcome. For example, the PDDL action `grab_gold` from the Wumpus domain is translated to the ReadyLog action description $[grab\_gold; select\_outcome(2)]$ denoting that the gold has been grabbed successfully. $select\_outcome(n)$ is an action which simply identifies which outcome of a stochastic action's outcome needs to be chosen. For a deterministic action, we do not have to select a specific outcome. This sequence is then translated to a *transition policy* that is executed before the remainder of the failed policy. The translation process is again straight-forward. The art is to find a set of PDDL

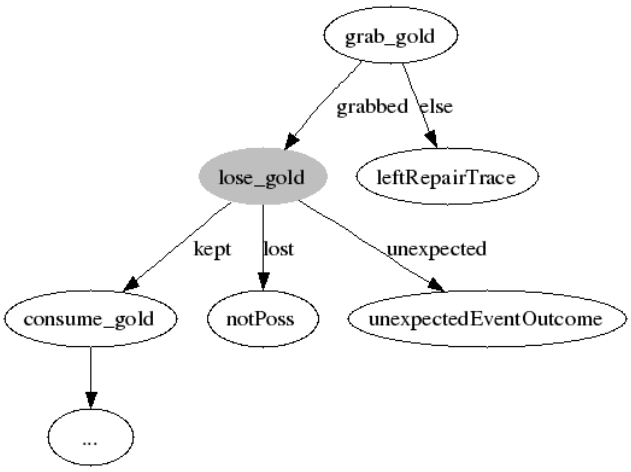

Figure 3. Example of a Repaired Policy.

actions that can be mapped to ReadyLog. Note that the resulting policy does not allow any branching over nature's choices, i.e. all stochastic actions or events must result in the outcome that is specified in the PDDL plan sequence by the *select_outcome*(n) statement. Any other possible outcome is marked with an additional failure marker $\mathfrak{M}$(*leftRepairTrace*). Thus, the transition policy represents a path of one possible execution trace. The remainder of the failed policy is attached to the leaf representing the successful execution of the transition policy. Figure 3 shows an example of a repaired policy. The original policy is the one of Figure 2. The flaw was that the first grab_gold action failed and the gold was not grabbed. Then, with *Trans* plan repair is initiated. The desired reference world state was the situation where the gold has been grabbed successfully. The sub-tree above the gray node in Figure 3 represents the repair plan, which consist of the single action grab_gold. Note that the two nodes resulting from the "not grabbing" and the "unexpected stochastic outcome" are not visible since they are subsumed by the $\mathfrak{M}$(*leftRepairTrace*). The sub-tree under the gray node including itself is the remainder of the original policy that is executable again.

5 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 with or move to a not-modeled direction with 30 percent. 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 agents 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 at least as complex as planning from scratch [27]. Therefore, we compared the run-times 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 [7]). 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.

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 of 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 for 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 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

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	

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 takes 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.

6 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.

Although it was shown by Nebel and Koehler [27] 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 [20], 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.

In [1], contingency plans are specified to provide action patterns when previously created plans fail. A similar idea, to combine Golog with an external planning system based on a PDDL description, was proposed in [7]. For the future work, we plan to combine their results with ours.

Clearly, our experimental results at the current stage resemble only a first proof of concept. Further and extended testing needs to be conducted. This particularly means to compare our method not only with brute-force Golog planners but also with other state-of-the-art planners in several other domain, similar as is done in [11]. Finally, we have to compare our approach with other state-of-the-art POMDP planners. We assume fully observable MDPs as an idealistic model for generating our policies. As there might be discrepancies between the action theory leading to a policy and the real execution, the problem that policies can fail, occurs. As such is our approach driven from the practical experience of programming robots and agents with the logic-based language Readylog and the observation that often only small errors in the execution lead to a complete failure of the optimal pol-

icy. Here, we have to stress again that READYLOG only calculates partial policies, not taking the whole state space into account. For our future work, we need to take other models like POMDPs into account as well.

Acknowledgements

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] M. Beetz. Structured reactive controllers. *Journal of Autonomous Agents and Multi-Agent Systems*, 2(4):25–55, 2001.
[2] R. Bellman. *Dynamic Programming*. Princeton University Press, Princton, NJ, 1957.
[3] D. Bertsekas. *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall, 1987.
[4] L. Böhnstedt, A. Ferrein, and G. Lakemeyer. Options in readylog reloaded - generating decision-theoretic plan libraries in golog. In *In Proc. KI-07*, pages 352–366, 2007.
[5] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proc. AAAI-00*, pages 355–362. AAAI Press, 2000.
[6] Y. Chen, W. Hsu, and B. Wah. SGPlan: Subgoal Partitioning and Resolution in Planning, 2004.
[7] J. Claßen, P. Eyerich, G. Lakemeyer, and B. Nebel. Towards an integration of golog and planning. In *Proc. IJCAI-07*. AAAI Press, 2007.
[8] G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In H. Levesque and F. Pirri, editors, *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, pages 86–102. Springer, 1999.
[9] G. De Giacomo, H. Levesque, and S. Sardiña. Incremental execution of guarded theories. *Computational Logic*, 2(4):495–525, 2001.
[10] G. De Giacomo, Y. Lsperance, and H. Levesque. ConGolog, A concurrent programming language based on situation calculus. *Artificial Intelligence*, 121(1–2):109–169, 2000.
[11] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76(1–2):35–74, July 1995.
[12] F. Dylla, A. Ferrein, and G. Lakemeyer. Specifying multi-robot coordination in ICPGolog – from simulation towards real robots. In *AOS-4 at IJCAI-03*, 2003.
[13] F. Dylla, A. Ferrein, G. Lakemeyer, J. Murray, O. Obst, T. Röfer, S. Schiffer, F. Stolzenburg, U. Visser, and T. Wagner. Approaching a formal soccer theory from behaviour specifications in robotic soccer. In P. Dabnicki and A. Baca, editors, *Computer in Sports*, pages 161–185. WIT Press, 2007. to appear.
[14] S. Edelkamp and J. Hoffmann. PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Technical Report 195, Institute für Informatik, December 2003.
[15] A. Ferrein. *Robot Controllers for Highly Dynamic Environments with Real-time Constraints*. Doctoral dissertation, RWTH Aachen University, Germany, 2008.
[16] A. Ferrein, C. Fritz, and G. Lakemeyer. On-line Decision-Theoretic Golog for Unpredictable Domains. In *Proc. KI-04*, 2004.
[17] A. Ferrein, C. Fritz, and G. Lakemeyer. Using golog for deliberation and team coordination in robotic soccer. *KI*, 19(1):24–30, 2005.
[18] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Artificial Intelligence Research*, 20:51–124, 2003.
[19] A. Gerevini and D. Long. Plan Constraints and Preferences in PDDL3. Technical Report RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia, Italy, 2005.
[20] G. D. Giacomo, R. Reiter, and M. Soutchanski. Execution Monitoring of High-Level Robot Programs. In *In Proc. KR-98*, pages 453–465, 1998.
[21] H. Grosskreutz. Probabilistic projection and belief update in the pgolog framework. In *The Second International Cognitive Robotics Workshop, (CogRob-00)*, pages pages 34–41. ECAI-00, 2000.
[22] H. Grosskreutz and G. Lakemeyer. cc-Golog – An Action Language with Continous Change. *Logic Journal of the IGPL*, 2002.
[23] W. Hsu, B. Wah, R. Huang, and Y. Chen. New Features in SGPlan for Handling Soft Constraints and Goal Preferences in PDDL3.0. *Proc. ICAPS-06*, June 2006.
[24] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
[25] J. McCarthy. Situations, Actions and Causal Laws. Technical report, Stanford University, 1963.
[26] D. McDermott. 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.
[27] B. Nebel and J. Koehler. Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis. Technical Report RR-93-33, DFKI GmbH, 1993.
[28] M. Puterman. *Markov Decision Processes: Discrete Dynamic Programming*. Wiley, New York, 1994.
[29] R. Reiter. *Knowledge in Action*. MIT Press, 2001.