Plan Repair in Hybrid Planning

Julien Bidot*, Bernd Schattenberg, and Susanne Biundo

Institute for Artificial Intelligence
Ulm University, Germany
firstname.lastname@uni-ulm.de

Abstract. We present a domain-independent approach to plan repair in a formal framework for hybrid planning. It exploits the generation process of the failed plan by retracting decisions that led to the failed plan fragments. They are selectively replaced by suitable alternatives, and the repaired plan is completed by following the previous generation process as close as possible. This way, a stable solution is obtained, i.e. a repair of the failed plan that causes minimal perturbation.

1 Introduction

For many real-world domains, hybrid approaches that integrate hierarchical task decomposition with action-based planning turned out to be most appropriate [1]. On the one hand, human expert knowledge can be represented and exploited by means of tasks and methods, which describe how abstract tasks can be decomposed into pre-defined plans that accomplish them. On the other hand, flexibility to come up with non-standard solutions, to overcome incompleteness of the explicitly defined solution space, or to deal with unexpected changes in the environment results from the option to insert tasks and primitive actions like in partial-order-causal-link planning (POCL). Furthermore, hybrid planning enables the generation of abstract solutions as well as of plans whose prefixes provide courses of primitive actions for execution, while other parts remain still abstract, ready for a refinement in later stages. With these capabilities hybrid approaches meet essential requirements complex real-world applications, such as mission or project planning, impose on AI planning technology. However, the problem of how to deal with execution failures in this context has not been considered in detail yet. In general, there are various alternatives for achieving this including contingency planning, replanning, and repair.

In this paper, we introduce an approach to plan repair in hybrid planning. The motivation is twofold. Firstly, real-world planning problems often result in complex plans (task networks) with a large number of causal, temporal, and hierarchical dependencies among the tasks involved. In order to keep the plans manageable, constructs such as conditionals can therefore only sparingly be used and need to be left for those plan sections which are most likely affected by uncertainty during execution. Replanning is not appropriate in this context either. Given the complexity of the planning domain, it is not reasonable to build a new plan from scratch in order to address just a single and

* This work has been supported by a grant of Ministry of Science, Research, and the Arts of Baden-Württemberg (Az: 23-7532.24-14-1).

A. Dengel et al. (Eds.): KI 2008, LNAI 5243, pp. 169–176, 2008.

exceptional execution failure. Secondly, plan stability is essential in this context. This means, the modified plan should be as similar to the failed plan as possible and should only differ at positions that definitely need to be altered to compensate for the failure. The reason is that large parts of the plan may be unaffected by the failure, some parts may have been executed already, others may require commitments in terms of resource allocations or activities from third parties that have already been requested or even carried out. In a word, perturbation of the plan at hand should be minimized in order to avoid any amount of unnecessary cancellation actions and confusion.

Most plan repair methods known from the literature (cf. Section 5) take aim at non-hierarchical plans and use local search techniques to modify a failed plan by removing and inserting plan steps. In contrast to these approaches, our plan repair method covers hierarchical task decomposition and relies on the plan generation process instead of operating on the failed plan itself.

We present a general *refinement-retraction-and-repair algorithm* that revises the plan generation process of the failed plan by first retracting those development steps that led to the failed plan fragments. In a second step, the repaired plan is produced by following the previous generation process as close as possible. This means to replace the failure-critical development steps by alternatives and redo the uncritical ones. The rationale behind this procedure is as follows: it is realistic to assume that all initial goals persist and that the underlying domain model is adequate and stable. Thus, an execution failure is taken as caused by exceptional conditions in the environment. Nevertheless, such a failure can disturb several dependencies within the complex plan structure and may require revisions that go beyond adding and removing plan steps and include even the decomposition of abstract tasks. Information gathered during the original plan generation process is reused to avoid decisions that inserted the failure-critical plan elements and explore the possible alternatives instead, thereby enabling an efficient and minimally invasive plan repair.

In the following, we first describe the hybrid planning framework. We then introduce the plan repair problem and provide some extensions to the formal framework that allow us to explicitly address failure-affected plan elements. After that, the generic plan repair algorithm is presented. It is based on a hybrid planning system that records information about plan generation processes. The plan repair algorithm exploits information about the generation of the original plan and uses least-discrepancy heuristics to efficiently search for stable repair plans. Finally, we review related work and conclude with some remarks.

2 A Hybrid-Planning Framework

The hybrid planning framework is based on an ADL-like representation of states and *primitive tasks*, which correspond to executable basic actions. States as well as preconditions and effects of tasks are specified through formulae of a fragment of first-order logic. *Abstract tasks*, which show preconditions and effects as well, and *decomposition methods* are the means to represent human expert knowledge in the planning domain model. Methods provide *task networks*, also called *partial plans* that describe how the corresponding task can be solved. Partial plans may contain abstract and primitive tasks.

With that, hierarchies of tasks and associated methods can be used to encode the various ways a complex abstract task can be accomplished. Besides the option to define and exploit a reservoir of pre-defined solutions this way, hybrid planning offers first-principles planning based on both primitive and abstract actions.

Formally, a domain model $D = \langle \mathtt{T}, \mathtt{M} \rangle$ consists of a set of task schemata $\mathtt{T}$ and a set $\mathtt{M}$ of decomposition methods. A partial plan is a tuple $P = \langle TE, \prec, VC, CL \rangle$ where TE is a set of *task expressions* (plan steps) $te = l{:}t(\overline{\tau})$ with t being the task name and $\overline{\tau} = \tau_1, \ldots, \tau_n$ the task parameters; the label l serves to uniquely identify the steps in a plan. $\prec$ is a set of *ordering constraints* that impose a partial order on plan steps in TE. VC are *variable constraints*, i.e. co-designation and non-co-designation constraints $v \doteq \tau$ resp. $v \not\doteq \tau$ on task parameters. Finally, CL is a set of *causal links* $\langle te_i, \phi, te_j \rangle$ indicating that formula ϕ, which is an effect of te_i establishes (a part of) the precondition of te_j. Like in POCL, causal links are the means to establish and maintain causal relationships among the tasks in a partial plan.

A *planning problem* $\pi = \langle D, P_{\text{init}} \rangle$ consists of a domain model D and an initial task network P_{init}. Please note that purely action-based planning problems given by state descriptions s_{init} and s_{goal}, like in POCL planning are represented by using distinguished task expressions te_{init} and te_{goal}, where s_{init} are the effects of te_{init} and te_{goal} has preconditions s_{goal}. The solution of a planning problem is obtained by transforming the initial task stepwise into a partial plan P that meets the following *solution criteria*: (1) all preconditions of the tasks in P are supported by a causal link, i.e. for each precondition ϕ of a task te_j there exists a task te_i and a causal link $\langle te_i, \phi, te_j \rangle$ in P; (2) the ordering and variable constraints of P are consistent; (3) the ordering and variable constraints ensure that none of the causal links is threatened, i.e. for each causal link $\langle te_i, \phi, te_j \rangle$ and each plan step te_k that destroys the precondition ϕ of te_j, the ordering constraints $te_i \prec te_k$ and $te_k \prec te_j$ are inconsistent with the ordering constraints $\prec$. If all task expressions of P are primitive in addition, P is called an *executable solution*. The transformation of partial plans is done using so-called *plan modifications*, also called *refinements*. Given a partial plan $P = \langle TE, \prec, VC, CL \rangle$ and domain model D, a plan modification is defined as $\mathtt{m} = \langle E^{\oplus}, E^{\ominus} \rangle$, where $E^{\oplus}$ and $E^{\ominus}$ are disjoint sets of elementary additions and deletions of so-called *plan elements* over P and D. The members of $E^{\ominus}$ are elements of P, i.e. elements of TE, $\prec$, VC or CL, respectively. $E^{\oplus}$ consists of new plan elements, i.e. task expressions, causal links, ordering or variable constraints that have to be inserted in order to refine P towards a solution. This generic definition makes the changes explicit that a modification imposes on a plan. With that, a planning strategy, which has to choose among all applicable modifications, is able to compare the available options qualitatively and quantitatively [1,2]. The application of a modification $\mathtt{m} = \langle E^{\oplus}, E^{\ominus} \rangle$ to a plan P returns a plan P' that is obtained from P by adding all elements in $E^{\oplus}$ to P and removing those of $E^{\ominus}$. We distinguish various classes of plan modifications.

For a partial plan P that has been developed from the initial task network of a planning problem, but is not yet a solution, so-called *flaws* are used to make the violations of the solution criteria explicit. Flaws list those plan elements that constitute deficiencies of the partial plan. We distinguish various flaw classes including the ones for unsupported preconditions of tasks and inconsistencies of variable and ordering constraints.

It is obvious that particular classes of modifications are appropriate to address particular classes of flaws while others are not. This relationship is explicitly represented by a *modification trigger function* α, which is used in the algorithm presented in [3], that relates flaw classes to suitable modification classes.

The resolution procedure is as follows: 1) the flaws of the current plan are collected; 2) relevant modifications are applied (generation of new plans); 3) the next plan to refine is selected, and we go to 1. This loop repeats, until a flawless plan is found and returned.

3 Formal Representation of the Plan-Repair Problem

After a (partly) executable solution for a planning problem has been obtained, it is supposed to be given to an execution agent that carries out the specified plan. We assume that a dedicated *monitoring component* keeps track of the agent's and the world's state and notices deviations from the plan as well as unexpected behaviour of the environment. It is capable of recognizing failure situations that cause the plan to be no longer executable: actions have not been executed, properties that are required by future actions do not hold as expected, etc. The monitor maps the observation onto the plan data structure, thereby identifying the *failure-affected elements* in the plan. This includes the break-down of causal links, variable constraints that cannot be satisfied, and the like. Note that this failure assessment is not limited to the immediately following actions in the executed plan fragments, as the plan's causal structure allows us to infer and anticipate causal complications for actions to be executed far in the future.

With such a failure description at hand, the system tries to find an alternative fail-safe plan for the previously solved problem, taking into account what has already been executed and is, therefore, viewed as *non-retractable decisions*. Remember that we assume the planning domain model to be valid and the failure event to be an exceptional incident. We hence translate the execution failure episode into a *plan-repair problem* that we will solve relying on the same domain model that we used for the previous problem.

As a prerequisite, we define the notions of failure descriptions and non-retractable decisions. In order to express exceptional incidents we augment a domain model with a set of particular primitive task schemata that represent non-controllable environmental processes: for every fluent, i.e. for every property that occurs in the effects of a task, we introduce two *process schemata* that invert the truth value of the respective fluent. Every failure-affected causal link will impose the plan-repair problem to contain a process, the effect of which falsifies the annotated condition. We assume that failures can be unambiguously described this way – if a disjunctive failure cause is monitored, the problem has to be translated into a set of alternative plan-repair problems, which is beyond the scope of this paper.

Regarding the representation of non-retractable decisions, every executed plan element, i.e. either an executed task or a constraint referring to such, has to occur in the repair plan. To this end, we make use of meta-variables of plan elements such as task expressions over given schemata in D, causal links between two task expression meta-variables, and ordering or variable constraints on appropriate meta-variables. Meta-variables are related with actual plan elements by *obligation constraints*, which serve in this way as a structural plan specification. We thereby distinguish two types

of constraints: existence obligations, which introduce meta-variables, and assignment obligations, which are equations on meta-variables and plan elements. A set of obligation constraints OC is *satisfied* w.r.t. a plan P, if (1) for every existence obligation in OC, a respective assignment obligation is included as well and (2) the assignment obligations instantiate a plan schema that is consistent with P.

Given a planning problem $\pi = \langle D, P_{\text{init}} \rangle$ and a plan $P_{\text{fail}} = \langle TE_{\text{fail}}, \prec_{\text{fail}}, VC_{\text{fail}}, CL_{\text{fail}} \rangle$ that is a partially executed solution to π that failed during execution, a plan-repair problem is given by $\pi^r = \langle D^r, P^r_{\text{init}} \rangle$ that consists of the following components: D^r is an extended domain model specification $\langle \mathtt{T}, \mathtt{M}, \mathtt{P} \rangle$ that concurs with D on the task schemata and method definitions and provides a set of process schemata $\mathtt{P}$ in addition. The initial plan $P^r_{\text{init}} = \langle TE_{\text{init}}, \prec_{\text{init}}, VC_{\text{init}}, CL_{\text{init}}, OC_{\text{init}} \rangle$ of the plan-repair problem is an obligation-extended plan data structure that is obtained from P_{init} and P_{fail} as follows. The task expressions (incl. te_{init} and te_{goal}), ordering constraints, variable constraints, and causal links are replicated in P^r_{init}. The obligation constraints OC_{init} contain an existence obligation for every executed plan element in P_{fail}. Finally, if broken causal links have been monitored, appropriate existence obligations for process meta-variables are added to OC_{init} and causally linked to the intended link producers. E.g., let $\langle te_a, \varphi, te_b \rangle$ be a broken causal link, then OC_{init} contains existence obligations for the following meta-variables: v_a stands for a task expression of the same schema as te_a, v_φ represents a process that has a precondition φ and an effect $\neg\varphi$, and finally $v_{CL} = \langle te_a, \varphi, te_\varphi \rangle$.

An obligation-extended plan $P^r = \langle TE, \prec, VC, CL, OC \rangle$ is a solution to a plan-repair problem $\pi^r = \langle D^r, P^r_{\text{init}} \rangle$, if and only if the following conditions hold: (1) $\langle TE, \prec, VC, CL \rangle$ is a solution to the problem $\langle D^r, P_{\text{init}} \rangle$ where P_{init} is obtained from P^r_{init} by removing the obligations. (2) The obligations in P^r are satisfied. (3) For every ordering constraint $te_a \prec te_b$ in $\prec$, either OC contains an assignment obligation for te_a or none for te_b.

Solution criterion (1) requires the repaired plan to be a solution in the sense of Section 2, and in particular implies that P^r is an executable solution to the original problem as well. (2) ensures that the non-retractable decisions of the failed plan are respected and that the environment's anomaly is considered. Criterion (3) finally verifies that the repair plan stays consistent w.r.t. execution; i.e., in the partial order, not yet executed tasks do not occur before executed ones.

The problem specification and solution criteria imply appropriate flaw and plan modification classes for announcing unsatisfied obligations and documenting obligation decisions. A specific modification generation function is used for introducing the appropriate processes. With these definitions, the repair mechanism is properly integrated into the general hybrid planning framework and a simple replanning procedure can easily be realized: The obligation-aware functions are added to the respective function sets in the algorithm presented in [3], which in turn is called by $\mathbf{simplePlan}(P^r_{\text{init}}, \pi^r)$. Note that the obligation extension of the plans is transparent to the framework. With the appropriate flaw and modification generators, the algorithm will return a solution to the repair problem, and while the new plan is developed, the obligation assignments will be introduced opportunistically. Apparently, the solution obtained by replanning

Require: Planning problem $\pi^r = \langle D^r, P^r_{\text{init}} \rangle$ and plan P_{fail}

1: **planRepair**($P_{\text{fail}}, \pi^r, \texttt{M}$)**:**
2: $P_{\text{rh}} \leftarrow$ **retract**($P_{\text{fail}}, \texttt{M}$) {Retract modifications}
3: **while** $P_{\text{rh}} \neq$ **fail do**
4: $P_{\text{safe}} \leftarrow$ **autoRefine**($P_{\text{rh}}, \texttt{M}$) {Remake modifications}
5: $P_{\text{new}} \leftarrow$ **simplePlan**(P_{safe}, π^r) {Call the algorithm of [3]}
6: **if** $P_{\text{new}} \neq$ **fail then**
7: **return** P_{new}
8: **else**
9: $P_{\text{rh}} \leftarrow$ **retract**($P_{\text{new}}, \texttt{M}$) {Retract modifications}
10: **return fail**

Algorithm 1. The refinement-retraction-and-repair algorithm

cannot be expected to be close to the previous solution. Furthermore, all knowledge of the previous successfull plan generation episode and in particular that concerning the un-affected portions of the plan, is lost. The following section will address these issues.

4 The Plan-Repair Algorithm

In this section, we describe a repair algorithm that allows us to find efficiently an executable solution plan to π^r, which is as close to a failed one (P_{fail}) as possible, by reusing the search space already explored to find P_{fail}.

In contrast to replanning, our plan-repair procedure starts from P_{fail}. Our objective w.r.t. search is to minimize the number of modifications to retract, remake, and newly add (minimal invasiveness). To achieve this efficiently we focus on the modifications $\texttt{M}$ that our planning system applied to obtain P_{fail} from the initial planning problem. We thus partition the modifications $\texttt{M}$ into the following two subsequences: $\texttt{M}_{\texttt{fail}}$ is the sequence of plan modifications that are related to the failure-affected elements of P_{fail}; $\texttt{M}_{\texttt{rest}} = \texttt{M} \setminus \texttt{M}_{\texttt{fail}}$ is the sequence of plan modifications that are not related to the failure-affected elements of P_{fail}. During our refinement-retraction-and-repair procedure, all modifications of $\texttt{M}_{\texttt{fail}}$ have to be retracted from P_{fail}.

Once an execution-time failure is detected by our monitoring system, we repair the current failed plan as described by Algorithm 1. It first retracts hybrid planning modifications within the *retraction horizon* (line 2). The retraction horizon corresponds to the first partial plan P_{rh} that is encountered during retraction of modifications in $\texttt{M}$ in which no failure-affected elements occur. The retraction step chooses the modifications to retract given $\texttt{M}$. At this step of the algorithm the retraction horizon comprises all the modifications made from the last one inserted in $\texttt{M}$ until the first modification in $\texttt{M}$ that is related to one or more failure-affected plan elements. In line 3, we have a consistent partial plan, which we name P_{rh}. The next step of Algorithm 1 consists in refining the current partial plan (P_{rh}) automatically by remaking every possible modifications in $\texttt{M}_{\text{rest}}$ (line 4); we have then a consistent partial plan that we name P_{safe}. We then call line 5 algorithm `simplePlan` presented in [3] to refine P_{safe} into an executable solution P_{new}. If no consistent plan is found, then the algorithm retracts modifications (line 9). Each time function `retract` is called (line 9) after backtrackings of `simplePlan`, the

retraction horizon includes one more modification included in $\mathtt{M}_{\mathtt{rest}}$. Each time function `autoRefine` is called after backtrackings of `simplePlan` (line 4), we remake one less modification of $\mathtt{M}_{\mathtt{rest}}$; i.e., we do not remake the modifications of $\mathtt{M}_{\mathtt{rest}}$ that are beyond the original retraction horizon. This algorithm runs iteratively until a plan (P_{new}) without flaws is found (line 7), or when no modification can be retracted any longer (line 9). The retraction step (lines 2 and 9) has a linear cost, which depends on the number of modifications to retract. Note that P_{fail}, P_{rh}, P_{safe}, and P_{new} represent each a repair plan P^r, i.e. a partial plan with obligation constraints.

During the retraction of modifications related to committed plan elements, the corresponding assignment obligations are no longer satisfied. During the refinement phase of Algorithm 1 (line 4), obligation flaws related to execution-time failures are addressed by automatically inserting environment processes.

There is no combinatorial search from P_{fail} to P_{safe} (Algorithm 1, lines 2-4).

Our retraction-refinement search algorithm can be extended to find a new plan P_{new} that is similar to P_{fail}. For achieving this, our procedure uses least-discrepancy heuristics when refining P_{safe} (Algorithm 1, line 5). The heuristics guide search with respect to the modifications that have led to plan P_{fail}: function f^{select} prefers the modifications that are similar to the modifications in $\mathtt{M}_{\mathtt{rest}}$; i.e., the modification-selection strategy prefers modifications that belong to the same class and that correspond to the same flaw class.

5 Related Work

Plan repair is rarely addressed in the context of hierarchical planning. However, there is a number of studies in the field that are relevant to this objective.

Fox et al. [4] demonstrate with an empirical study that a plan-repair strategy can produce more stable plans than those produced by replanning, and their system can produce repaired plans more efficiently than replanning. Their implementation uses local-search techniques and plan-oriented stability metrics to guide search. To what extent our approach may benefit from using such metrics is subject to future work.

The planning system of Yoon et al. [5] computes a totally-ordered plan before execution; each time it is confronted with an unexpected state, it replans from scratch without reusing previous planning effort.

Nebel and Köhler [6] report a theoretic and practical study about plan reuse and plan generation in a general situation. Using the non-hierarchical propositional STRIPS planning framework, the authors show that modifying a plan is not easier than planning from scratch. Recent work in hierarchical case-based planning includes the approach based on SHOP presented by Warfield et al. [7].

Kambhampati and Hendler [8] present some techniques for reusing old plans. To reuse an old plan in solving a new problem, the old plan, along with its annotations, has to be found in a library of plans and is then mapped into the new problem. A process of annotation verification is used to locate applicability failures and suggest refitting tasks.

Drabble et al. [9] propose plan-repair mechanisms to integrate several pre-assembled repair plans into an ongoing and executing plan when action effects fail for a limited number of reasons.

Van der Krogt and de Weerdt [10] endow a partial-order causal link planning system with capabilities of repairing plans. In this context, plan repair consists of adding and removing actions. During the execution of a plan, their monitoring system may observe uncontrollable changes in the set of facts describing a state, which makes the plan fail.

6 Conclusion

We presented a novel approach to plan repair that is capable of dealing with hierarchical and partial-order plans. Embedded in a well-founded hybrid planning framework, it uses the plan generation process of the failed plan to construct a repair. In a first phase, all plan refinements that introduced failure-affected plan elements – possibly including even task decompositions – are retracted from the failed plan. After that, additional obligation constraints are inserted into the resulting partial plan in order to ensure that already executed plan steps will be respected by the repair plan. The repair procedure constructs a solution by replacing the failure-critical plan modifications and replaying as many of the uncritical ones as possible, thereby achieving stability of the repair plan. The flexibility of our hybrid planning framework, where plan deficiencies and modifications are explicitly defined, enables the repair of various plan and execution failures including both the removal of plan steps or entire plan fragments that became obsolete and the insertion of extra tasks coming up.

References

1. Biundo, S., Schattenberg, B.: From abstract crisis to concrete relief–a preliminary report on combining state abstraction and HTN planning. In: Proc. of ECP 2001, pp. 157–168 (2001)
2. Schattenberg, B., Bidot, J., Biundo, S.: On the construction and evaluation of flexible plan-refinement strategies. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNCS (LNAI), vol. 4667, pp. 367–381. Springer, Heidelberg (2007)
3. Schattenberg, B., Weigl, A., Biundo, S.: Hybrid planning using flexible strategies. In: Furbach, U. (ed.) KI 2005. LNCS (LNAI), vol. 3698, pp. 258–272. Springer, Heidelberg (2005)
4. Fox, M., Gerevini, A., Long, D., Serina, I.: Plan stability: Replanning versus plan repair. In: Proc. of ICAPS 2006, pp. 212–221 (2006)
5. Yoon, S., Fern, A., Givan, R.: FF-Replan: A baseline for probabilistic planning. In: Proc. of ICAPS 2006, pp. 352–359 (2006)
6. Nebel, B., Köhler, J.: Plan reuse versus plan generation: A theoretical and empirical analysis. Artificial Intelligence 76, 427–454 (1995)
7. Warfield, I., Hogg, C., Lee-Urban, S., Muñoz-Avila, H.: Adaptation of hierarchical task network plans. In: FLAIRS 2007, pp. 429–434 (2007)
8. Kambhampati, S., Hendler, J.A.: A validation-structure-based theory of plan modification and reuse. Artificial Intelligence 55, 193–258 (1992)
9. Drabble, B., Tate, A., Dalton, J.: Repairing plans on-the-fly. In: Proceedings of the NASA Workshop on Planning and Scheduling for Space (1997)
10. van der Krogt, R., de Weerdt, M.: Plan repair as an extension of planning. In: Proc. ICAPS 2005, pp. 161–170 (2005)