Plan Reversals for Recovery in Execution Monitoring

Thomas Eiter and **Esra Erdem** and **Wolfgang Faber**
Institut für Informationssysteme 184/3,
Technische Universität Wien
Favoritenstraße 9-11, 1040 Wien, Austria

Abstract

In this paper, we introduce a new method to recover from discrepancies in a general monitoring framework where the agent finds some explanations (points of failure) for discrepancies. According to this method, the agent finds a reverse plan to backtrack to a diagnosed point of failure and subsequently continues with the original plan. This method is appealing given that such a reverse plan is short with respect to the overall plan to be executed. While a reverse plan could be computed online by solving a planning problem, we present a potentially more efficient method: We first build offline a reverse plan library by finding reverse plans for action sequences, and then use this library online to construct reverse plans. The former part is done by reducing the problem of finding pairs of action sequences and reverse plans (of a certain length) to a conformant planning problem; for the latter, we present a polynomial time algorithm. Furthermore, we analyze the complexity of finding reverse plans, and obtain that the presented reduction is reasonable in general.

Introduction

When an agent is situated in a nondeterministic environment, execution of a plan may need to be monitored to ensure that the plan does not fail to achieve the goal. For example, imagine a shopping agent that accidentally picks the wrong, but more expensive, milk from a shelf. Later, at the cashier, she might realize that she does not have enough money to pay, and the remainder of her shopping plan is obsolete.

Execution monitoring may help to reveal that things go wrong and to recover from any detected execution failure. To this end, the agent may determine a discrepancy between the actual and the expected state of the world. Such a discrepancy usually implies a failure (at least in the agent's belief), for which a recovery should be sought. A diagnosis of the discrepancy can be useful to find a reasonable plan recovery. In the previous scenario, by monitoring, the agent might discover earlier the wrong milk in the shopping cart and conclude that she did not pick the right one. She then can return the expensive milk, grab the right one instead, and continue with the execution of the rest of her shopping plan.

In this paper, we are concerned with a single agent in a logic-based monitoring framework. We consider in particular the issue of plan recovery. Detecting discrepancies and generating diagnoses for them has been considered in (Eiter, Erdem, & Faber 2004).

Among the possible strategies for recovery are replanning—replacing the remaining plan by a new plan, patch planning—adding actions to be executed before the remaining plan, and backtracking—adding actions which lead to a diagnosed point of failure and executing the remaining plan from there. In this paper, we will focus on backtracking.

Backtracking to a diagnosed point of failure can be done in various ways such as by (a) finding a reverse plan, (b) reversing the given plan, (c) reversing action occurrences, or (d) restarting actions. Here, each way can be regarded as a special case of the preceding one. The most general one, (a), amounts to solving a planning problem, and any plan is acceptable. In (b), this plan should be assembled from a reverse plan library, while in (c), each executed action is undone by some reverse action fetched from the library. Finally, restartable actions in (d) do not require undoing. Clearly, more specific backtracking methods will be less often applicable in general.

The main contributions of this paper are summarized as follows:

- We formally define and illustrate the notions of a reverse action and a reverse plan for an action in a general framework, addressing both items (b) and (c) above. These notions are extended to multi-step reversals where a sequence of actions can be reversed, and to conditional reversals where the applicability of a reversal is subject to a condition on the current state.
- We consider offline computation of a reverse plan library and online computation of a reverse plan from a given library. We show that the former can be achieved by conformant planning, and, for the latter, we provide a polynomial-time algorithm.
- Finally, we analyze the complexity of some problems related to the computation of a reverse plan library, including the problem of finding a reverse action or a reverse plan.

In order to keep our results at a general level, we describe execution monitoring in a general action representation framework as in (Turner 2002). In this framework, an action description can be represented by a transition diagram—a directed graph whose nodes correspond to states and whose edges correspond to action occurrences. It can accommodate nondeterminism, concurrent actions and dynamic worlds. Such action representations can be obtained from domain descriptions in STRIPS-based languages or in more expressive action description languages, such as $\mathcal{C}+$ (Giunchiglia *et al.* 2003) or $\mathcal{K}$ (Eiter *et al.* 2002). This allows us to use systems like CCALC, CPLAN, and DLV$^{\mathcal{K}}$ for reasoning about actions.

In the logic-based frameworks (Giacomo, Reiter, & Soutchanski 1998; Soutchanski 1999; 2003), execution monitoring is described in the situation calculus as in (Reiter 2001), which makes them applicable to Golog programs (Levesque *et al.* 1997). In (Fichtner, Großmann, & Thielscher 2003), execution monitoring is described in the fluent calculus (Thielscher 2001) for FLUX programs (Thielscher 2004). These works differ from ours with respect to plan recovery mainly in that plan reversals are not studied in detail (see the last section).

In the following, first we present the action representation framework and the planning framework we consider, and briefly discuss our monitoring framework. Then, we precisely describe the reverse of an action and a plan, and extend these definitions to handle various cases. After that, we discuss how to compute offline plan reversals, and how to exploit this library of plan reversals online for plan recovery. After a complexity analysis of these problems, we conclude with a discussion of the related work.

Preliminaries

We consider the action representation and planning framework described in (Turner 2002).

Action representation framework

We begin with a set $\mathcal{A}$ of action symbols and a disjoint set $\mathcal{F}$ of fluent symbols. Let *state*$(\mathcal{F})$ be a formula in which the only nonlogical symbols are elements of $\mathcal{F}$. This formula encodes the set of states that correspond to its models. Let *act*$(\mathcal{F}, \mathcal{A}, \mathcal{F}')$ be a formula whose only nonlogical symbols are elements of $\mathcal{F} \cup \mathcal{A} \cup \mathcal{F}'$, where $\mathcal{F}'$ is obtained from $\mathcal{F}$ by priming each element of $\mathcal{F}$. Then the formula

$$\mathit{state}(\mathcal{F}) \wedge \mathit{act}(\mathcal{F}, \mathcal{A}, \mathcal{F}') \wedge \mathit{state}(\mathcal{F}') \qquad (1)$$

encodes the set of transitions that corresponds to its models. That is,

(i) the start state corresponds to an interpretation of the symbols in $\mathcal{F}$,

(ii) the set of actions executed corresponds to an interpretation of actions executed correspond to an interpretation of the symbols in $\mathcal{A}$, and

(iii) the end state corresponds to an interpretation of the symbols in $\mathcal{F}'$.

Formula (1) is abbreviated as *tr*$(\mathcal{F}, \mathcal{A}, \mathcal{F}')$.

Example 1 [(Giunchiglia *et al.* 2003)] Putting a puppy in water makes the puppy wet, and drying a puppy with a towel makes it dry. With the fluents

$$\mathcal{F} = \{\mathit{inWater}, \mathit{wet}\},$$

and actions

$$\mathcal{A} = \{\mathit{putIntoWater}, \mathit{dryWithTowel}\}$$

the states can be described by the formula

$$\mathit{state}(\mathcal{F}) = \mathit{inWater} \supset \mathit{wet}.$$

Since there are three interpretations of $\mathcal{F}$ satisfying the formula above, there are 3 states:

$$\{\mathit{inWater}, \mathit{wet}\}, \{\neg \mathit{inWater}, \mathit{wet}\}, \{\neg \mathit{inWater}, \neg \mathit{wet}\}.$$

The action occurrences can be defined as follows:

$$\begin{aligned}&\mathit{act}(\mathcal{F}, \mathcal{A}, \mathcal{F}') = \\ &\quad (\mathit{inWater}' \equiv \mathit{inWater} \vee \mathit{putIntoWater}) \wedge \\ &\quad (\mathit{wet}' \equiv (\mathit{wet} \wedge \neg \mathit{dryWithTowel}) \vee \mathit{putIntoWater}) \wedge \\ &\quad (\mathit{dryWithTowel} \supset (\neg \mathit{inWater} \wedge \neg \mathit{putIntoWater}))\end{aligned}$$

The last line expresses that $\mathit{dryWithTowel}$ is executable when $\mathit{inWater}$ is false and it is not executable concurrently with $\mathit{putIntoWater}$.

For instance, the interpretation

$$\{\neg \mathit{inWater}, \mathit{wet}, \mathit{dryWithTowel}, \neg \mathit{putIntoWater}, \neg \mathit{inWater}', \neg \mathit{wet}'\}$$

satisfies *tr*$(\mathcal{F}, \mathcal{A}, \mathcal{F}')$, therefore it describes a transition:

$$\langle \{\neg \mathit{inWater}, \mathit{wet}\}, \{\mathit{dryWithTowel}\}, \{\neg \mathit{inWater}, \neg \mathit{wet}\} \rangle.$$

Note that the interpretation of $\mathcal{A}$ above describes the occurrence of $\mathit{dryWithTowel}$.

The meaning of a domain description can be represented by a transition diagram—a directed graph whose nodes correspond to states and whose edges correspond to action occurrences. In a transition diagram, a "trajectory" of length n is obtained by finding a model of

$$\mathit{tr}_n(\mathcal{F}, \mathcal{A}) = \bigwedge_{t=0}^{n-1} \mathit{tr}(\mathcal{F}_t, \mathcal{A}_t, \mathcal{F}_{t+1})$$

where each $\mathcal{F}_i$ (resp., each $\mathcal{A}_i$) is the set of fluents obtained from $\mathcal{F}$ (resp., $\mathcal{A}$) by adding time stamp i to each fluent symbol (resp., each action symbol). A trajectory is an alternating sequence of states and action occurrences that correspond to interpretations of fluent and action atoms, respectively. A trajectory is of the form

$$S_0, A_0, S_1, \ldots, S_{n-1}, A_{n-1}, S_n \qquad (2)$$

where each S_i is the state that corresponds to (the interpretation of fluents in) $\mathcal{F}_i$, and each A_i is the action occurrences that correspond to (the interpretation of action atoms in) $\mathcal{A}_i$. In the rest of the paper, S and S_i denote states and A and A_i denote action occurrences.

Example 2 According to the transition diagram for the action description of Example 1 presented in Figure 1, here is a trajectory:

$$\begin{aligned}&\{\neg \mathit{inWater}_0, \mathit{wet}_0\}, \{\mathit{dryWithTowel}_0\}, \\ &\{\neg \mathit{inWater}_1, \neg \mathit{wet}_1\}, \{\mathit{putIntoWater}_1\}, \\ &\{\mathit{inWater}_2, \mathit{wet}_2\}\end{aligned}$$

expressing that the wet puppy is first dried with a towel and then put into the water.

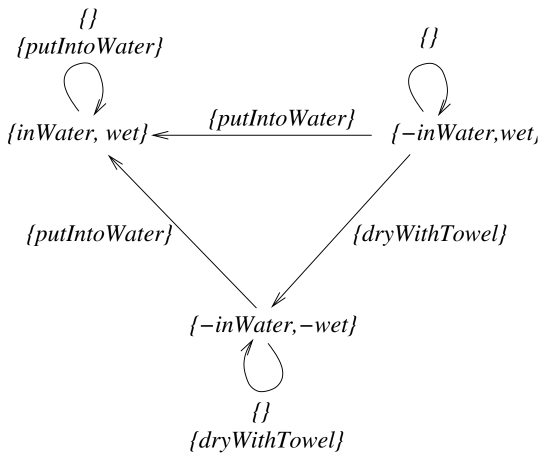


Figure 1: The transition diagram for the action description of Example 1.

Planning framework

In a planning problem, an initial state is described by a formula $\mathit{init}(\mathcal{F})$ (where $\mathit{init}(\mathcal{F}) \models \mathit{state}(\mathcal{F})$), and the goal is described by a formula $\mathit{goal}(\mathcal{F})$. A plan of length n is obtained from any model of

$$\mathit{init}(\mathcal{F}_0) \wedge \mathit{tr}_n(\mathcal{F}, \mathcal{A}) \wedge \mathit{goal}(\mathcal{F}_n). \tag{3}$$

The plan is the sequence of action occurrences that correspond to the interpretation of action atoms. It is of the form

$$\langle A_0, \ldots, A_{n-1} \rangle \tag{4}$$

where each A_i is the action occurrences that correspond to the interpretation of action atoms in $\mathcal{A}_i$.

A *trajectory for a plan P* (4) is a trajectory

$$S_0, A'_0, S_1, \ldots, S_{n-1}, A'_{n-1}, S_n$$

such that $A_i = A'_i$ ($0 \leq i < n$), $S_0 \models \mathit{init}(\mathcal{F}_0)$, and $S_n \models \mathit{goal}(\mathcal{F}_n)$. In the following, we will denote by T_P the set of all trajectories for a plan P.

Example 3 Consider a version of the blocks world in which a block B can be thrown from location $L1$ to another location $L2$. Unfortunately, throwing is not accurate and so the block may end up on any location, and not necessarily on $L2$.

With the fluents $on(B, L)$ ("block B is on location L"), and the actions $\mathit{throw}(B, L, L1)$ ("throw block B from location L to location $L1$"), the states, i.e., $\mathit{state}(\mathcal{F})$, can be defined by the conjunction of the following formulas:[1]

- every block should be on some location:

$$\bigvee_{L} on(B, L); \tag{5}$$

- if a block is on some location then it is not anywhere else:

$$on(B, L) \supset \bigwedge_{L \neq L1} \neg on(B, L1); \tag{6}$$

- a block cannot have more than one block on itself:

$$on(B1, B) \supset \bigwedge_{B1 \neq B2} \neg on(B2, B); \tag{7}$$

- every block is "supported" by the table (here $\mathit{supported}(B)$ is an auxiliary propositional variable defined in terms of $on(B, L)$):

$$\mathit{supported}(B). \tag{8}$$

Formula $\mathit{act}(\mathcal{F}, \mathcal{A}, \mathcal{F}')$ can be defined by the conjunction of the following formulas:

- the preconditions of $\mathit{throw}(B, L, L1)$:

$$\mathit{throw}(B, L, L1) \supset (on(B, L) \wedge \bigwedge_{B1} \neg on(B1, B)); \tag{9}$$

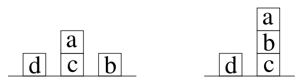
- the effects of $\mathit{throw}(B, L, L1)$, and inertia:

$$\begin{array}{l} on(B, L1)' \supset \\ \quad (on(B, L1) \vee \bigvee_{L, L2} \mathit{throw}(B, L, L2)); \end{array} \tag{10}$$

- no-concurrency:

$$\begin{array}{l} \mathit{throw}(B, L, L1) \supset \\ \quad \bigwedge_{B1 \neq B, L2, L3} \neg \mathit{throw}(B1, L2, L3). \end{array} \tag{11}$$

Consider, in this domain, a planning problem $\mathcal{P}$ with the following initial and goal states:

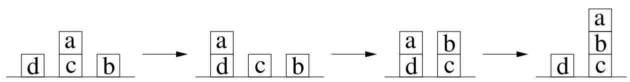


initial goal

A solution to this problem would be the plan

$$P = \langle \mathit{throw}_0(a, c, d), \mathit{throw}_1(b, \mathit{table}, c), \mathit{throw}_2(a, d, b) \rangle.$$

A unique trajectory T_P exists for P, shown in the following graph:



0 1 2 3

An *evolution of a state* S_i reached at time stamp i from an initial state S_0 after action occurrences $A_0, \ldots, A_{i-1}$ of a plan P is a trajectory

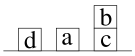
$$S_0, A_0, \ldots, S_{i-1}, A_{i-1}, S_i \tag{12}$$

obtained by finding a model of the formula

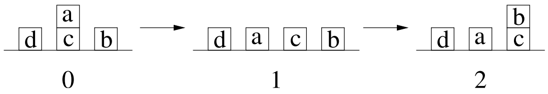
$$\mathit{init}(\mathcal{F}_0) \wedge \mathit{tr}_i(\mathcal{F}, \mathcal{A}).$$

[1]In the following, $B, B1, B2$ range over a finite set of block constants, and $L, L1, L2, L3$ range over the set of location constants that consists of the set of block constants and the constant table.

Example 4 Assume that during the execution of plan P of Example 3, at time stamp 2, the following state S_2 is observed:



There is one evolution E_P of the state S_2 reached at time stamp 2 from the initial state S_0:



A *conformant plan* is a plan (4) such that every evolution (12) ($0 \leq i \leq n$) of any state can be extended to a trajectory for the plan:

$$\forall \mathcal{F}_0, \ldots, \mathcal{F}_n \exists \mathcal{F}'_0, \ldots, \mathcal{F}'_n \; init(\mathcal{F}'_0) \\ \wedge \bigwedge_{t=0}^{n-1} \left(init(\mathcal{F}_0) \wedge tr_t(\mathcal{F}, A) \supset tr(\mathcal{F}_t, A_t, \mathcal{F}'_{t+1})\right) \\ \wedge \left(init(\mathcal{F}_0) \wedge tr_n(\mathcal{F}, A) \supset goal(\mathcal{F}_n)\right). \tag{13}$$

The formula above expresses the following conditions on plan (4). The first conjunct expresses that there is at least one possible initial state. The second conjunct describes the executability of the plan. The last conjunct ensures that execution of the plan at an initial state leads to a goal state.

Example 5 Plan P of Example 3 is not a conformant plan for the planning problem $\mathcal{P}$, witnessed by evolution E_P, in which $throw(a, d, b)$ is not executable. No conformant plan made of *throw* actions exists for $\mathcal{P}$.

Now assume that another action is available, by which the agent can carry a block B safely to a location L (denoted by $carry(B, L)$), such that B will definitely reside on L after the execution of this action. With this new action, the definition of $act(\mathcal{F}, \mathcal{A}, \mathcal{F}')$ is modified as follows. The conjunction (9) with the formula below describes the preconditions of actions:

$$carry(B, L) \supset \bigwedge_{B1} \neg on(B1, B), \tag{14}$$

and the conjunction of the following with (11) describes no-concurrency:

$$(carry(B, L) \supset \\ (\bigwedge_{B1,L2,L3} \neg throw(B1, L2, L3) \wedge \\ \bigwedge_{B1 \neq B, L2} \neg carry(B1, L2))). \tag{15}$$

The effects of actions, and inertia are defined, instead of (10), with the conjunction:

$$(carry(B, L) \supset on(B, L)') \wedge \\ (on(B, L1)' \supset \\ (on(B, L1) \vee carry(B, L1) \vee \\ \bigvee_{L,L2} throw(B, L, L2))). \tag{16}$$

A conformant plan for $\mathcal{P}$ then is

$$P' = \langle carry_0(a, d), carry_1(b, c), carry_2(a, b)\rangle.$$

In the following sections, $\mathcal{F} \equiv \mathcal{F}'$ denotes $\bigwedge_{f \in F} f \equiv f'$.

Monitoring Framework

According to our framework, for monitoring the execution of a plan relative to a set of intended trajectories, the monitoring agent

1. checks whether there is a discrepancy between the current state and the corresponding states of the given trajectories;
2. if no discrepancy is detected then continues with the execution of the plan; otherwise, tries to find a diagnosis of discrepancies by examining the given trajectories against evolutions of the current state;[2]
3. if a diagnosis is found then recovers from the discrepancies by backtracking to the diagnosed point of failure and executing the plan from that point on; otherwise, finds another plan from the current state to reach a goal state.

Consider, for instance, the setting of Example 4. Here, a discrepancy can be observed between the state S_2 and the trajectory T_P at time stamp 2. The point of failure for this discrepancy is the state S_0 at time stamp 0 because, the evolution E_P of S_2 "deviates" from the trajectory T_P at time stamp 0 in state S_0. That is, the states of T_P and E_P are identical at time stamp 0, but they differ at time stamp 1. The reason is that, in E_P, block a ended up on the table rather than on block d when executing $throw(a, c, d)$. This point of failure can be used to recover from the discrepancy above. In particular, the agent can backtrack to the state S_0 from the state S_2 and then execute P again. In the following, we will describe how such a backtracking can be done, and we will give formal specifications of that. More details about points of failure can be found in (Eiter, Erdem, & Faber 2004).

Backtracking

In the process of recovering from discrepancies, backtracking from the state where a discrepancy is observed to a state diagnosed as a point of failure can be done in several ways. For instance, one can compute a plan to reach the point of failure, or "undo" every action till the point of failure. An action can be undone by executing one of its "reverse actions" or by executing a "reverse plan". Sometimes, several actions can be undone by executing a reverse plan. In the following, we will make these concepts precise.

Reverse actions and plans

We define a reverse of an action below relative to a given action description represented by a transition diagram.

An action A' is a *reverse action* for A, if, for all $\mathcal{F}$ and $\mathcal{F}'$, the formula

$$revAct(\mathcal{F}, \mathcal{F}'; A, A') = \\ tr(\mathcal{F}, A, \mathcal{F}') \supset \\ (tr(\mathcal{F}', A', \mathcal{F}) \wedge \forall \mathcal{F}'' \, (tr(\mathcal{F}', A', \mathcal{F}'') \supset \mathcal{F} \equiv \mathcal{F}''))$$

is true (i.e., $\forall \mathcal{F} \forall \mathcal{F}' revAct(\mathcal{F}, \mathcal{F}'; A, A')$ is a tautology).

The formula above expresses the following condition about actions A and A'. Take any two states S, S' (described

[2] In our framework, like in (Fichtner, Großmann, & Thielscher 2003), the detected discrepancies may not be relevant to the successful execution of the rest of the plan.

by the interpretations of fluents in $\mathcal{F}$ and $\mathcal{F}'$ respectively) such that executing A at S leads to S'. Then executing A' at state S' always leads to S.

Example 6 In the setting of Example 5, $carry(B, L)$ is a reverse action for $throw(B, L, L1)$. Indeed, consider any two states S, S' such that, for some block B, and for some locations $L, L1$, executing $throw(B, L, L1)$ at state S leads to state S'. Due to the preconditions of $throw(B, L, L1)$, i.e., (9), we know that S is a state at which block B is on location L and block B is clear. Due to the nondeterministic effect of $throw(B, L, L1)$ and due to inertia, i.e., (16), we know that S' is a state at which block B is on some location $L2$, block B is clear, and other blocks are in the same locations as they are in S. Due to the preconditions of $carry(B, L)$, i.e., (14), we can execute $carry(B, L)$ at state S'. Due to the deterministic effect of $carry(B, L)$ and due to inertia, i.e., (16), carrying block B onto location L leads to the state S'' at which B is on location L, and other blocks are in the same locations as they are in S. That is, $S'' = S$.

A *reverse plan for an action* A is a sequence $\langle A_0, \ldots, A_{m-1}\rangle$ ($m \geq 0$) of actions such that, for all $\mathcal{F}$ and $\mathcal{F}'$, the formula

$$\begin{array}{l} revPlan(\mathcal{F}, \mathcal{F}'; A, [A_0, \ldots, A_{m-1}]) = \\ \quad tr(\mathcal{F}, A, \mathcal{F}') \supset \\ \qquad \forall \mathcal{F}_0, \ldots, \mathcal{F}_m \exists \mathcal{F}'_1, \ldots, \mathcal{F}'_m (\mathcal{F}_0 \equiv \mathcal{F}' \supset \\ \qquad\quad (\bigwedge_{t=0}^{m-1} (tr_t(\mathcal{F}, A) \supset tr(\mathcal{F}_t, A_t, \mathcal{F}'_{t+1})) \wedge \\ \qquad\quad (tr_m(\mathcal{F}, A) \supset \mathcal{F}_m \equiv \mathcal{F}))) \end{array}$$

holds.

The formula above expresses the following condition about an action A and an action sequence $\langle A_0, \ldots, A_{m-1}\rangle$. Take any two states S, S' (described by the interpretations of fluents in $\mathcal{F}$ and $\mathcal{F}'$ respectively) such that executing A at S leads to S'. If the action sequence $\langle A_0, \ldots, A_{m-1}\rangle$ is executable at state S', then it always leads to S. The executability condition of $\langle A_0, \ldots, A_{m-1}\rangle$ is described above by the formula $\bigwedge_{t=0}^{m-1} (tr_t(\mathcal{F}, A) \supset tr(\mathcal{F}_t, A_t, \mathcal{F}'_{t+1}))$.

Note that $revPlan(\mathcal{F}, \mathcal{F}'; A, [A_0])$ is equivalent to $revAct(\mathcal{F}, \mathcal{F}'; A, A_0)$.

Multi-step reversals

We can further generalize the notion of reversing by considering plans, rather than actions, to be reversed. There are two motivations for this generalization: It might not always be possible to find reverse plans for single actions, but only for sequences of actions. Also, a reverse plan for an action sequence might be shorter than a reverse plan obtained by concatenating reverse plans for subsequences (as in Example 7).

A sequence $\langle A'_0, \ldots, A'_{m-1}\rangle$ ($m \geq 0$) of actions is a *reverse plan for an action sequence* $\langle A_0, \ldots, A_{k-1}\rangle$ ($k > 0$), if, for all $\mathcal{F}$ and $\mathcal{F}'$, the formula

$$\begin{array}{l} multiRev(\mathcal{F}, \mathcal{F}'; [A_0, \ldots, A_{k-1}], [A'_0, \ldots, A'_{m-1}]) = \\ \quad \exists \mathcal{F}_0, \ldots, \mathcal{F}_k \ (\mathcal{F} \equiv \mathcal{F}_0 \wedge tr_k(\mathcal{F}, A) \ \wedge \mathcal{F}' \equiv \mathcal{F}_k) \supset \\ \qquad \forall \mathcal{F}'_0, \ldots, \mathcal{F}'_m \exists \mathcal{F}''_0, \ldots, \mathcal{F}''_m (\mathcal{F}'_0 \equiv \mathcal{F}' \supset \\ \qquad\quad \bigwedge_{t=0}^{m-1} \Big(tr_t(\mathcal{F}', A') \supset tr(\mathcal{F}'_t, A'_t, \mathcal{F}''_{t+1}) \Big) \wedge \\ \qquad\quad (tr_m(\mathcal{F}', A') \supset \mathcal{F}'_m \equiv \mathcal{F})) \end{array}$$

is true.

The formula above is very similar to $revPlan(\mathcal{F}, \mathcal{F}'; A, [A_0, \ldots, A_{m-1}])$. The only difference is that, in the premise of the formula, a trajectory is considered instead of a single transition.

Note that $multiRev(\mathcal{F}, \mathcal{F}'; [A_0], [A'_0, \ldots, A'_m])$ is equivalent to $revPlan(\mathcal{F}, \mathcal{F}'; A_0, [A'_0, \ldots, A'_m])$.

Example 7 In the setting of Example 5, a reverse plan for the action sequence

$$\begin{array}{l} \langle throw(B1, L1, L2), carry(B1, L3), \\ \ throw(B1, L3, L4), carry(B1, L5)\rangle \end{array}$$

is $\langle carry(B1, L1)\rangle$. Indeed, executing the action sequence above at a state S changes the location of block $B1$ from location $L1$ to location $L5$, without changing the locations of other blocks. Carrying block $B1$ to location $L1$ at this new state brings the blocks world back to its state S.

Conditional reversals

In the above, a reverse plan is defined for an action sequence at any reachable state. However, at some such states, an action sequence may not admit any reverse plan. That is, an action sequence may have a reverse plan under conditions that do not hold at every reachable state. To make plan reversals applicable in such cases, we extend the concept of a reverse plan to a "conditional reverse plan" that takes state information into account.

A sequence $\langle A'_0, \ldots, A'_{m-1}\rangle$ ($m \geq 0$) of actions is a ϕ-*reverse plan for an action sequence* $\langle A_0, \ldots, A_{k-1}\rangle$ ($k > 0$) if, for any $\mathcal{F}$ and $\mathcal{F}'$, the formula

$$\begin{array}{l} \phi(\mathcal{F}') \supset \\ \qquad multiRev(\mathcal{F}, \mathcal{F}'; [A_0, \ldots, A_{k-1}], [A'_0, \ldots, A'_{m-1}]). \end{array}$$

is true. Here $\phi(\mathcal{F}')$ is a formula whose only free variables are $\mathcal{F}'$.

Example 8 Consider a variant of Example 5 with another deterministic action of carrying a block B from a location L to another location $L1$: $carry2(B, L, L1)$. Note that $\langle carry2(B, L4, L1)\rangle$ is not a reverse plan for the action sequence

$$\langle throw(B, L1, L2), carry(B, L3), throw(B, L3, L4)\rangle,$$

as B need not be on $L4$ after the execution of this action sequence, due to the nondeterministic effects of $throw(B, L3, L4)$. However, it is a ϕ-reverse plan, with $\phi(\mathcal{F}')$ being $on(B, L4)'$.

Computation

A *reverse plan item*, RPI, is a tuple of the form (AS, R, ϕ) such that R is a ϕ-reverse plan for the (nonempty) action sequence AS, where $\phi = \phi(\mathcal{F})$. An RPI is *single-step*, if $|AS| = 1$, i.e., AS consists of a single action, and *unconditional*, if ϕ = *true*. A *reverse plan library* L is a (finite) set of RPIs; it is called *single-step* (resp., *unconditional*), if each RPI in it is single-step (resp.,unconditional). For instance, according to Example 6,

$$(\langle throw(B, L, L1)\rangle, \langle carry(B, L)\rangle, true)$$

is a single-step unconditional RPI; whereas, according to Example 8, the RPI

$$(\langle throw(B,L1,L2), carry(B,L3), throw(B,L3,L4)\rangle,\\ \langle carry2(B,L4,L1)\rangle,\\ on(B,L4))$$

is neither single-step nor unconditional.

We now discuss how to compute offline a reverse plan library and how to exploit it online during plan recovery.

Offline RPI computation

Given an action description D represented by a transition diagram $tr(\mathcal{F},\mathcal{A},\mathcal{F}')$, one can compute RPIs (AS,R,ϕ) for every action sequence AS by solving conformant planning problems P_{rev} defined relative to a modification D_{rev} of D. The solutions to these planning problems can then be used to fill a reverse plan library.

We discuss first how to compute all unconditional RPIs for AS. For that we define D_{rev} and P_{rev} as follows. We consider the fluents in $\mathcal{F}_{rev} = \mathcal{F} \cup \tilde{\mathcal{F}}$, where $\tilde{\mathcal{F}} = \{\tilde{f} \mid f \in \mathcal{F}\}$ consists of new fluent symbols. For D_{rev}, the states are defined by the interpretations of $\mathcal{F}_{rev}$ such that $state(\mathcal{F})$. The transition function is defined by formula $tr_{rev}(\mathcal{F}_{rev},\mathcal{A},\mathcal{F}'_{rev})$ where fluent values in $\tilde{\mathcal{F}}$ are copied to $\tilde{\mathcal{F}}'$:

$$tr_{rev}(\mathcal{F}_{rev},\mathcal{A},\mathcal{F}'_{rev}) = (tr(\mathcal{F},\mathcal{A},\mathcal{F}') \wedge \tilde{\mathcal{F}} \equiv \tilde{\mathcal{F}}').$$

For P_{rev}, the initial state is defined by the formula $init_{rev}(\mathcal{F}_{rev})$, which encodes all possible states over $\mathcal{F}$, and which additionally duplicates all fluents of $\mathcal{F}$ to $\tilde{\mathcal{F}}$:

$$init_{rev}(\mathcal{F}_{rev}) = (state(\mathcal{F}) \wedge \mathcal{F} \equiv \tilde{\mathcal{F}}).$$

The goal conditions are defined by the formula $goal_{rev}(\mathcal{F}_{rev})$ which makes sure that the fluent values for $\mathcal{F}$ (which have been changed by actions according to $tr(\mathcal{F},\mathcal{A},\mathcal{F}')$) are equal to those in $\tilde{\mathcal{F}}$ (which are equal to the initial state fluent values):

$$goal_{rev}(\mathcal{F}_{rev}) = (\mathcal{F} \equiv \tilde{\mathcal{F}}).$$

Any conformant plan for P_{rev} of length ≥ 1 represents reverse plans:

Theorem 1 *Let P_{rev} be the planning problem defined relative to the action description D_{rev}, and let $\langle A_0,\dots,A_{n-1}\rangle$ $(0 < n)$ be a conformant plan of P_{rev}. Then for any $i \in \{1,\dots,n\}$, $\langle A_i,\dots,A_{n-1}\rangle$ is a reverse plan of $\langle A_0,\dots,A_{i-1}\rangle$ relative to D.*

Example 9 Consider the setting of Example 5. The associated planning problem P_{rev} admits the following conformant plans of length 2:

$$\langle throw(X,Y,Z), carry(X,Y)\rangle$$

$(X \in \{a,b,c,d\}, Y \neq Z, Y,Z \in \{table,a,b,c,d\})$. These plans give rise to a single-step, unconditional plan library.

The complexity of deciding the existence of reverse actions and repair plans of up to polynomial length (longer ones do not seem to be economical) can be characterized as follows.

Theorem 2 *Deciding if a given action description D has some pair (AS,R), such that AS is an action sequence of length $m > 0$ and R is a reverse action (resp. a reverse plan of length n bounded by a polynomial) for AS, is Σ_2^p-complete (resp. Σ_3^p-complete). Hardness holds also if AS is fixed or if $m = 1$, and, in case of reverse plans, for $n = 2$.*

Membership follows from the structures of the formulas given in the previous sections, and from the reduction to conformant plans. Hardness can be shown by reductions from evaluation of suitable fragments of QBFs (2-QSAT and 3-QSAT), respectively.

These results follow from the fact that checking whether an action (resp. a plan of length $n > 1$) R is a reverse action (resp. plan) for a (possibly fixed) AS in D is coNP-complete (resp. Π_2^p-complete).

Note that, when only sequential actions and short plans (length bound by a constant) are considered, the complexity of finding a reverse plan drops to that of checking (i.e. coNP or Π_2^p).

The reduction from finding reverse plans to conformant planning (which is also Σ_3^p-complete (Turner 2002; Eiter *et al.* 2004)) is reasonable from a complexity point of view. However, for the simpler case of finding reverse actions, it is not immediate that P_{rev} falls into a simpler class. Conformant planners might therefore not be able to detect that solving it is an easier task. To overcome this problem, we have created an alternative reduction (which we omit for space reasons) of finding reversal actions to a conformant planning problem in which actions are always executable, which is known to be Σ_2^p-complete (Turner 2002; Eiter *et al.* 2004).

Online reverse plan assembly

We first consider reverse plan libraries L which contain only single-step RPIs. In this case, a reverse plan for a given action sequence $AS = A_0,\dots,A_{i-1}$ (occurring in a plan) from the reached state, S_i, can be done by the algorithm presented in Figure 2. Here, exec(S,R) computes the state which results from executing the action sequence $R = \langle B_1,\dots,B_k\rangle$ starting in state S.

Proposition 1 *(i)* S-REVERSE(AS,S_i,L) *is a polynomial-time algorithm.*
(ii) S-REVERSE(AS,S_i,L) *correctly outputs, relative to L, a reverse plan RP for AS starting at S_i, and the resulting state S_0; or it determines that such a reverse plan does not exist.*

Example 10 Consider the setting of Example 5. Take L to be the derived single-step unconditional plan library of Example 9. Consider the plan P of Example 3 with the observed state S_2 of Example 4. Then, as described earlier, the point of failure is state S_0. To produce a reverse plan for the action sequence $\langle throw(a,c,d), throw(b,table,c)\rangle$, to reach the point of failure from state S_2, we call

$$\text{S-REVERSE}(\langle throw(a,c,d), throw(b,table,c)\rangle, S_2, L).$$

Algorithm S-REVERSE(AS, S_i, L)

Input: Action sequence $AS = \langle A_0, \ldots, A_{i-1}\rangle$, $i \geq 0$, state S_i, single-step reverse plan library L;
Output: Reverse plan RP from L for AS from state S_i and resulting state, S_0, or "no" if none exists

S := S_i; RP := ϵ ; /* empty plan */
for each $j = i-1, i-2, \ldots, 0$ **do**
  if some $(\langle A\rangle, R, \phi) \in L$ exists
      s.t. $A{=}A_j \wedge \phi(S){=}\mathit{true}$ **then**
    begin
      RP := RP + R;
      S := exec(S,R); /* undo A_j */
    end
  else return "no";
return (RP, S)

Figure 2: Algorithm S-REVERSE to compute plan reversals using a single-step plan library.

According to S-REVERSE, after initialization, for $j{=}1$, the only match is

$$(\langle \mathit{throw}(b, \mathit{table}, c)\rangle, \langle \mathit{carry}(b, \mathit{table})\rangle, \mathit{true}).$$

Therefore, $RP := \langle \mathit{carry}(b, \mathit{table})\rangle$ and S is computed as the state of time stamp 1 in E_P. For $j = 0$, the only match in L is

$$(\langle \mathit{throw}(a, c, d)\rangle, \langle \mathit{carry}(a, c)\rangle, \mathit{true});$$

hence, $S = S_0$ and $RP := \langle \mathit{carry}(b, \mathit{table}), \mathit{carry}(a, c)\rangle$ are finally returned.

When we consider a multi-step plan library, i.e., not necessarily a single-step plan library, finding a reverse plan RP is trickier since RP may be assembled from L in many different ways, and state conditions might exclude some of them. For instance, take

$$AS = \langle A, B, C\rangle,$$

and

$$L = \{(\langle A, B\rangle, \langle D\rangle, \phi_1), (\langle C\rangle, \langle E\rangle, \phi_2), (\langle A\rangle, \langle F\rangle, \phi_3), (\langle B, C\rangle, \langle G\rangle, \phi_4)\}.$$

We can assemble the action sequence $\langle A, B, C\rangle$ from $\langle A, B\rangle$ and $\langle C\rangle$, or from $\langle A\rangle$ and $\langle B, C\rangle$. However, in the former case, ϕ_1 might be false at the state resulting from reversing C by E, while, in the latter case, ϕ_3 might be true at the state resulting from reversing the action sequence $\langle B, C\rangle$ by the action G. Therefore, we need to consider choices and constraints when building a reverse plan.

Fortunately, this is not a source of intractability, and a reverse plan from L can be found in polynomial time (if one exists) by the algorithm REVERSE in Figure 3, which generalizes algorithm S-REVERSE.

The auxiliary array S in the algorithms in Figure 3 is used to keep state information when available, i.e., each $S[j]$ contains information about the state S_j before the action occurrences A_j. The main algorithm, REVERSE, initializes every $S[j]$ ($j < i$) of S to $\perp$ since we do not have information about the states $S_0, \ldots, S_{i-1}$ initially. The recursive algorithm REVERSE1 updates S whenever new knowledge is gained. For instance, if the action A_{i-1} can be reversed at state S_i, then we can get some information about state S_{i-1} and modify $S[j-1]$ accordingly. Having such state information available in S helps us find a reverse plan for the action sequence AS from L. Also, it prevents us explore the same search space over and over.

Algorithm REVERSE(AS, S_i, L)

Input: Action sequence $AS = \langle A_0, \ldots, A_{i-1}\rangle$, $i \geq 0$, state S_i, reverse plan library L;
Output: Reverse plan RP from L for AS from state S_i and resulting state, S_0, or "no" if none exists

Set $S[0] := \perp, \ldots, S[i-1] := \perp$ and $S[i] := S_i$;
RP := REVERSE1(i);
if RP = "no" **then return** "no"
else return (RP,$S[0]$)

Algorithm REVERSE1(j)

Input: integer j, $0 \leq j \leq i$ (=$|AS|$);
Output: Reverse plan RP for $\langle A_0, \ldots, A_{j-1}\rangle$ from $S[j]$, or "no"

if $j = 0$ **then return** ϵ ; /* empty plan */
for each $(As, R, \phi) \in L$
      s.t. As is a suffix of $\langle A_0, \ldots, A_{j-1}\rangle$ **do**
  if $\phi(S_j){=}\mathit{true} \wedge S[j-|As|] = \perp$ **then**
    begin
      $S[j-|As|]$:= exec($S[j]$,R); /* undo As from S_j*/
      RP := REVERSE1$(j-|As|)$;
      if $RP \neq$ "no" **then return** R + RP
    end
return "no"

Figure 3: Algorithm REVERSE to compute plan reversals using a multi-step plan library.

The algorithm REVERSE starts constructing a reverse plan for an action sequence $A_0, \ldots, A_{j-1}$ by considering its suffixes As. For efficiently determining all As in L, we can employ search structures such as a trie (or indexed trie) to represent L: consider each node of the trie labeled by an action so that the path from the root to the node would describe an action sequence. If the node describes an action sequence As such that (As, R, ϕ) is in L then the node is linked to a list of all RPIs (As', R, ϕ) in L where $As{=}As'$. With such a trie representation, all matching suffixes As in L can be determined in time $O(\ell_{\max}(L))$, where $\ell_{\max}(L) = \max\{|As| \mid (As, R, \phi) \in L\}$.

Notice that exec($S[j], R$) is called at most i times, and that each action in $A_0, \ldots, A_{i-1}$ is covered by exactly one such undo.

Theorem 3 (i) $\text{REVERSE}(AS, S_i, L)$ *is a polynomial-time algorithm. Its running time is of order* $O(|AS|(|L| \cdot \phi_{\max}(L) + \ell_{\max}(L) + exec_{\max}(\mathcal{A})))$, *where* $\phi_{\max}(L)$ *and* $exec_{\max}(\mathcal{A})$ *bound the times to evaluate* $\phi(S)$ *for formula* ϕ *in L and exec(S,A) for any action occurrence A, respectively.*

(ii) $\text{REVERSE}(AS, S_i, L)$ *correctly outputs, relative to L, a reverse plan* RP *for* AS *starting at* S_i*, and the resulting state* S_0*; or it determines that such a plan does not exist.*

Related Work and Conclusion

We have introduced a new method to recover from discrepancies in plan execution, according to which the agent finds a reverse plan to backtrack to a diagnosed point of failure and continues with the execution of the original plan from that point on.

This method is different from the plan recovery approaches of the other logic-based monitoring frameworks (Giacomo, Reiter, & Soutchanski 1998), (Soutchanski 1999) (Soutchanski 2003), and (Fichtner, Großmann, & Thielscher 2003) as follows. In (Giacomo, Reiter, & Soutchanski 1998), backtracking is not considered; instead, a new plan is computed so that executing it followed by the remaining plan would lead to a goal situation from the current situation. In (Soutchanski 1999), the authors consider restartable plans so that the agent can backtrack to a past nondeterministic choice point without having to compute a plan. After identifying the latest nondeterministic choice point, the agent executes the plan from that point on, to reach a goal situation from the current situation. If the agent cannot reach a goal situation then she identifies the next past nondeterministic choice point, and follows the recovery procedure as above. In (Soutchanski 2003), backtracking is considered in connection with inserting corrective plans as in (Giacomo, Reiter, & Soutchanski 1998), by a recursive recovery procedure like the one in (Soutchanski 1999). The agent computes a plan from the current situation to reach the latest nondeterministic choice point. If executing the plan from that point on does not lead to a goal situation then the agent tries to recover by inserting a corrective plan at that point. If the agent cannot find a corrective plan then she finds the next past nondeterministic choice point, and follows the recovery procedure as above. In (Fichtner, Großmann, & Thielscher 2003), backtracking is not considered. If a diagnosis is found then some predefined plans are executed to achieve the goals; otherwise, a new plan is computed to reach to a goal situation from the current situation.

The idea of backtracking for recovery is similar to "reverse execution" in program debugging (Zelkowitz 1973; Agrawal, DeMillo, & Spafford 1991), where every action is undone to reach a "stable" state. Our method is more general because it does not require an execution history to be able to undo actions.

Implementation of the algorithms above, including the generation of conditional reverse plan libraries, is on our future agenda. Another future direction is to extend the recovery framework above, from propositional logic, to a strips-style predicate language.

Acknowledgments

Thanks to Mikhail Soutchanski for useful discussions. This work was supported by FWF (Austrian Science Funds) under project P16536-N04.

References

Agrawal, H.; DeMillo, R. A.; and Spafford, E. H. 1991. An execution backtracking approach to program debugging. *IEEE Software* 8(3):21–26.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2002. A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system. *Artificial Intelligence* 144(1–2):157–211.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2004. A logic programming approach to knowledge-state planning, semantics and complexity. *ACM TOCL* 5(2).

Eiter, T.; Erdem, E.; and Faber, W. 2004. Finding explanations for discrepancies in plan execution.

Fichtner, M.; Großmann, A.; and Thielscher, M. 2003. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae* 57(2–4).

Giacomo, G. D.; Reiter, R.; and Soutchanski, M. 1998. Execution monitoring of high-level robot programs. In *Principles of Knowledge Representation and Reasoning*, 453–465.

Giunchiglia, E.; Lee, J.; Lifschitz, V.; McCain, N.; and Turner, H. 2003. Nonmonotonic causal theories. *Artificial Intelligence*. To appear.

Levesque, H. J.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1-3):59–83.

Reiter, R. 2001. *Knowledge in action: Logical Foundations for specifying and implementing dynamical systems*. MIT Press.

Soutchanski, M. 1999. Execution monitoring of high-level temporal programs. In *Proc. of IJCAI Workshop on Robot Action Planning*.

Soutchanski, M. 2003. High-level robot programming and program execution. In *Proc. of ICAPS Workshop on Plan Execution*.

Thielscher, M. 2001. The concurrent, continuous Fluent Calculus. *Studia Logica* 67(3):315–331.

Thielscher, M. 2004. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*. To appear.

Turner, H. 2002. Polynomial-length planning spans the polynomial hierarchy. In *Proc. of Eighth European Conf. on Logics in Artificial Intelligence (JELIA'02)*, 111–124.

Zelkowitz, M. 1973. Reversible execution. *Communications of ACM* 16(9):566.