Introduction

In this paper, we develop techniques for using classical planning systems to provide decision support to human planners: specifically to help human planners identify and fix problems in their plans that can arise from uncontrolled events in the execution environment. Most automated planning systems assume that the objective of the system is to generate a plan that will achieve some goal or goals in the state of the world. In many practical applications, however, the primary planning responsibility rests with human planners, either because planning systems are not capable of handling the full complexity of the planning application, or because the human users are not willing to cede planning responsibility.

In this paper, we show how to use any automated planning system to analyze a plan to identify ways that uncontrolled (disturbance) actions could cause the plan to fail in execution, and produce counterexample traces that would show how failures could occur. The intent is that human planners could use these traces as guidance in improving their plans, perhaps by incorporating our technique in an interactive plan critiquing system. Our proposal is a planning analog to the use of model-checking systems to verify critical hardware and software systems.

We describe how our system, MURPHY, translates a plan into what we call a “counter planning” problem, combining a representation of the initial plan with the definition of a set of uncontrolled actions. These uncontrolled actions may be the actions of other agents in the environment, either friendly, indifferent or hostile, or they may be events that simply occur. The result of this translation is a disjunctive planning problem, which we further process in order to play into the strengths of existing classical planners. Using this formulation, a classical planner can find counterexample traces that illustrate ways a plan may go awry.

We present empirical results in order to demonstrate the practicality of using classical planners as plan verifiers. Our experiments probe the difficulty of the counter planning problem. We vary the difficulty of the counter planning problems along a number of dimensions, including the number of agents available to the counter-planner, and the agents’ initial configuration. We show how these affect the difficulty of counter-planning. We also compare the use of a classical planner with the use of a more conventional verification tool: NuSMV (Cimatti et al. 2002b). Our results show that even for very difficult planning problems, MURPHY can efficiently generate counter planning instances. We also show that for most of the counter planning problems, a PDDL planner (FD (Helmert 2006)) can rapidly compute counterexamples, where they exist, or verify that no counterexample exists. They also show that the planner can do this more efficiently than the general verification system, NuSMV.

Why Analyze Against Disturbances?

While critiquing of plans has been a fixture even in the early days of the AI planning literature (Sacerdoti 1977; Erol et al. 1995), these works have typically focused on completely specified models and static worlds where only the planner agent can change the state of the world. In dynamic worlds, we must consider how to make our plans robust to the actions of other agents, and the first step in doing so is to identify where these plans are threatened by the actions of another agent.

As an example, consider the following multi-agent, dynamic planning domain description, called Escape Monster. Figure 1 illustrates this domain.

An explorer in the jungle, shown here at grid cell (9, 1), wishes to reach the flag at grid cell (1, 10). The jungle has a river and mountains that are impassable. There are various items in the jungle that the explorer can collect in order to build bridges and rafts (to cross rivers), or to blast tunnels (to pass through mountains). The black squares indicate other obstacles, that cannot be overcome with the aid of tools. There is also a monster in the jungle: in this example it is at square (10, 8). Initially, the monster is not aware of the explorer, and is harmless. However, if the explorer gets into the monster’s line of sight, the monster will start to take actions to catch and kill the explorer. The monster can see any grid cells in the four directions from its current location. If the monster and the explorer are ever at the same location, the monster will kill the explorer.

One possible plan is as follows:

The explorer first collects the rope at grid cell (10, 4) and the logs at (7, 5). After collecting these items, the explorer will return to the bottom of the grid, then move along the south and west edges to the river at (1, 5), build a raft, cross the river, and move north to reach the flag.

Unfortunately, this plan is flawed, and our system, MURPHY might present a counterexample such as:
The explorer moves north to collect the rope at $(10, 4)$. While collecting the rope, the monster sees the explorer. The monster starts to move, collects the mine at $(8, 8)$, blasts a tunnel through the mountains and catches the explorer.

Identifying such vulnerabilities can be crucial; the outcome of executing a flawed plan could be catastrophic and irreversible. Counterexamples like those our system produces would provide decision support to human planners in improving their plans. A planner who sees the above counterexample might revise the plan so that:

The explorer first moves north and west to collect the logs, then turns south to get the nails. The explorer will return to the bottom of the grid, then move along the south and west edges to the river at $(1, 5)$, picking up the hammer on the way. When the explorer reaches the river, she will use the hammer, nails and logs to build a bridge across the river, then cross the river, and move north to reach the flag.

This revised plan will not cause the explorer to enter the monster’s line of sight, so she will reach her goal safely.

Our Translation Method

Problem input format. We use the Planning Domain Description Language (PDDL) (Ghallab et al. 1998; Fox and Long 2003) of the International Planning Competition (IPC) to define our domains and problems. In particular, we use the ADL dialect of PDDL: ADL provides reasonable expressive power for more complex domains, through quantification and conditional effects, while still being supported by a relatively large number of freely-available planners.

Preliminary definitions. Let $L$ be the set of all positive literals in a function-free first-order language over a set of constants, $C$. A state, $s$ is a set of positive literals, $s \subseteq L$. A classical planning problem is a tuple $P = (L, s_0, g, O)$, where $s_0$ is the initial state, $g$ is the goal (a set of ground literals of $L$), and $O$ is a set of operators. Each operator $o \in O$ is a triple $o = (name(o), \text{pre}(o), \text{eff}(o))$, where $name(o)$ is $o$’s name and argument list. $\text{pre}(o)$ is the preconditions of the operator, a logical formula formed by using the usual logical operators for conjunction, disjunction, negation, and bounded quantification. An action $\alpha$ is a ground instance of an operator, $\alpha, \alpha = o/\sigma$, where $\sigma$ is a set of bindings for the arguments of $o$. $\text{pre}(\alpha)$ and $\text{eff}(\alpha)$ are the preconditions and effects with the substitution $\sigma$ applied.

If a state $s$ satisfies $\text{pre}(\alpha)$, then $\alpha$ is executable in $s$. The effects of an operator $\alpha, \text{eff}(\alpha)$, are (in general) a list of quantified implications. When an action, $\alpha$, is applied in a state, $s$, where $s \models \text{pre}(\alpha)$, the effects of $\alpha$ in $s$ (we overload $\text{eff}$ to $\text{eff}(\alpha, s)$) are the set of positive ($\text{tp}(E)$) and negative ($\text{nf}(E)$) literals, $E$, such that $\text{eff}(\alpha), s \models E$. If $\alpha$ is executable in $s$, the result of applying $\alpha$ in $s$ is

$$\text{result}(s, \alpha) = s \cup \left( \bigcup_{e \in \text{p}(\text{eff}(\alpha, s))} e \right) - \left( \bigcup_{e \in \text{n}(\text{eff}(\alpha, s))} e \right)$$

A plan is a sequence $\pi = (\alpha_1 \prec \alpha_2 \prec \ldots \prec \alpha_n)$ of actions. We overload $\text{result}$ for a sequence of actions $\pi$ such that $\text{result}(s, \epsilon) = s$ and $\text{result}(s, \alpha\pi) = \text{result}(\text{result}(s, \alpha), \pi)$. If any of the $\alpha$ in $\pi$ is not executable in the corresponding state, the plan is ill-formed. $\pi$ is a solution for the classical planning problem $P$ if $\pi$ is well-formed with respect to $s_0$ and $\text{result}(s, \pi) \models g$.

Counter planning problems. We define a counter planning problem, for a solved planning problem, $\langle P, \pi \rangle$ (from now on we will simply refer to a counter planning problem wrt the plan, $\pi$) as $C = \langle P, \pi, L', c_0, A \rangle$. $L' \supseteq L$ is a set of positive literals that is a superset of the literals of $P$, and $c_0 \supseteq s_0$ is an augmented initial state, a superset of the initial state. The set of literals and the initial state are typically augmented in order to capture the state of some agent or agents outside the control of the original planner. $A = U \cup O$ is the set of operators of the counter planning problem. $U$ is a set of uncontrolled operators modeled in the same way as the operators $O$. Note that $U \cap O = \emptyset$. When necessary for clarity, we will refer to the original operators (actions) as controlled operators (actions).

A counterexample for $\pi$ wrt $C$ is a sequence of actions, $\Pi$, that is an interleaving of $\pi$ with some uncontrolled actions $\pi - \Pi = \pi|\pi - \Pi$ containing some controllable $\alpha$ such that $\Pi = \Pi'\alpha\Pi''$ where $\Pi'$ is executable, but $\alpha, \text{result}(\Pi')$ is not executable ($\text{result}(c_0, \Pi') \not\models \text{pre}(\alpha)$). A solution for $C$ is a counterexample if one exists, or $\bot$.

MURPHY: A New Translation Procedure. We have developed a translation that takes as input (1) a PDDL planning domain; (2) a PDDL planning problem; (3) a plan that solves the domain + problem; (4) a counter planning domain extension with new objects and uncontrolled operators. The translation yields a new counter planning problem and domain that can be solved to find a counterexample or, if searched exhaustively without finding a trace, indicates that

\[\text{result}(s, \alpha) = s \cup \left( \bigcup_{e \in \text{p}(\text{eff}(\alpha, s))} e \right) - \left( \bigcup_{e \in \text{n}(\text{eff}(\alpha, s))} e \right)\]
no counterexample exists. There are three components to the translation process: (1) generating a “skeleton” for the original plan that ensures that the original plan actions occur in the proper order; (2) formulating a goal expression that will cause a planner to search for a counterexample and (3) encoding the resulting disjunctive goal expression in a way that is amenable to the use of a standard PDDL planner.

First, we create new operators that replicate the actions in the plan. That is, for each action in the original plan, we create a new operator definition that combines the original operator definition with the parameters actually used in the plan. So, if the original action is \( \alpha = \alpha/\sigma \) we create a new operator, \( \tilde{\alpha}(\sigma) = \alpha/\sigma \). This is a simple matter of applying bindings. The second part of the skeleton construction is to force these operators to be inserted in the counterexample plan in the proper order. This is done by creating a new predicate, \( P \), and a family of new constants, \( p_0 \ldots p_n \) for an original plan of length \( n \). \( P \) will act as a counter: if \( P(p_k) \in s \) then the last action from the original plan to be executed is \( \pi_k \). This may be ensured by adding \( \neg P(p_i) \) to \( \text{pre}(\tilde{\alpha}_i) \) and for \( i \geq 1 \), adding \( P(p_{i-1}) \), as well. Finally, we add \( P(p_i) \) to \( \text{eff}(\tilde{\alpha}_i) \). The above is sufficient to ensure that the actions appear in an output plan, they will only appear in the right order (and will permit the introduction of arbitrary uncontrolled actions into the sequence). The goal (see below) will ensure that the actions do appear.

The second step in the translation is to formulate the goal for the counterplanning. There are two pieces to formulating the goal: one is to formulate the goal of defeating the preconditions of an action, and the second is to do this at the right time in the counterexample. The first is, in some sense, trivial. To formulate precondition failure for \( \alpha_i \) the goal is simply \( \neg \text{pre}(\alpha_i) \). The second part of the solution is to make sure that \( \neg \text{pre}(\alpha_i) \) holds “right before” the execution of \( \alpha_i \). But this is equivalent to “after the execution of \( \alpha_{i-1} \) and before the execution of \( \alpha_i \). So the counterplanning goal is:

\[
\bigcup_{0 \leq i \leq n} \neg \text{pre}(\alpha_i) \land \neg P(p_i) \land (\text{if } i \geq 1, P(p_{i-1})) \quad (1)
\]

Unfortunately for us, the PDDL planners we know of do not handle disjunctive goals well, and indeed some will not process them at all.\(^2\) Fortunately, PDDL planners do perform disjunctive reasoning: they make or decisions when choosing between operators. Accordingly, as the final step in building the counter planning problem, we translate the disjunctive counter planning goal into a set of pseudo operators. We add a distinguished new zero-argument predicate, \( G() \), that is not true in the initial state, \( c_0 \). In order to generate these pseudo-operators and give them simple preconditions, we repeatedly apply deMorgan’s laws to the formulas, and transform them to disjunctive and negation normal form.\(^3\) For each of the disjuncts in (1), above, we create a new operator, whose precondition is that disjunct, and whose single effect is to make \( G \) true. The actual goals that are given to the planner in order to search for a counterexample then, is simply this new proposition, \( G() \).

The soundness of the above translation procedure follows immediately from its construction. Given access to a sound and complete classical planner, we have a sound and complete algorithm for deciding a counter planning problem as defined above. But see the future work discussion for limitations of this definition.

**Experimental Evaluation**

We implemented an algorithm, called MURPHY, that uses our translation technique, and therefore, is able to use any classical planner to search for counterexamples. We did an experimental investigation of the following questions:

1. *Is the translation method of MURPHY practical? Is the outcome of translation effective for a classical planner to use for planning?*

2. *In planning domains that are difficult for a verification tool, how much does a classical planner improve the performance for plan verification?*

For the classical planner, we used FASTDOWNWARD (FD) (Helmer2006). For the verification tool, we used NuSMV (Cimatti et al. 2002b), a model-checker using Binary Decision Diagrams (BDDs) (Bryant1992).

We chose two planning domains: an enhanced version of the Airports problem domain from IPC-2004 (Hoffmann and Edelkamp2005) and a new planning domain, called Escape Monster. For each planning domain we developed a suite of counter planning problems, using the MURPHY translation algorithms, and tested them on FD and NuSMV.

**Airport.** This domain is modeled on ground operations at Munich airport. The challenge for the planner is to find plans to get aircraft between gates and the runway, both arriving and departing. For counter planning, we added uncontrolled ground vehicles (e.g., repair trucks) and aircraft (e.g., general-aviation aircraft). The trucks start from some initial location in the airport and move around to repair taxiways, possibly blocking aircraft movements in the original plan. The general-aviation aircraft arrive and depart, and if they push back from a gate in either case, they require the airport’s push-back resources (tugs), which can prevent the original planner’s aircraft from leaving their gates.

For the suite of plans to verify in our experiments, we used FD to generate solutions to the IPC-2004 Airport nontemporal ADL problems. We generated three types of Airport counter planning problems for each original plan: (1) there is both general-aviation aircraft and a repair truck in the airport; (2) there is general-aviation but no repair trucks; and (3) there is no general-aviation but a repair truck. For each of these types we generated 5 sets of random initial locations of the general-aviation and trucks.

**Escape Monster.** The rationale for using the Escape Monster domain, introduced above, was that we observed that in the airports domain, the counter-planning was primarily a matter of path-finding. The Escape Monster domain

---

\(^2\)Malte Helmert confirmed this, personal communication.

\(^3\)While in the worst case, this transformation can cause an explosion in formula size, our experimental results, even for complex actions such as the ones in the Airports domain, show that in practice the growth is acceptable.
adds more complicated sub-goaling, because the monster may need to collect some tools and overcome obstacles, rather than simply finding a path to a blocking (in this case, capturing) position. For this reason, the Escape Monster domain is more complex than the well-known pursuit-evasion problems (e.g., (Moldenhauer and Sturtevant 2009; Ishida and Korf 2002)).

We have generated 20 random problems in this domain without monsters. FD’s solutions for these problems served as the plans to be verified. For each original planning problem and for each case where we have one or two monsters, we have randomly varied the initial locations of the monsters in the world, creating 10 counter problems for each case, a total of 400 counter planning scenarios.

We ran all of our experiments on Amazon Elastic Compute Cloud (EC2). Each run was performed on a Debian system using 2.66GHz Intel Xeon CPU with 7.5 GB memory. We measured the average running times of MURPHY in both Airport and Escape-Monster domains. In both domains, MURPHY translated the original planning problems into counter planning problems very quickly: for the Escape Monster problems, the average CPU time for MURPHY was 0.212 seconds with a standard deviation of 0.036. This shows that there was little variance across this experiment set of 400 counter-problem instances.

For the Airport problems, MURPHY was still fast; Figure 2 shows the results as a function of the solution length for the original planning problems and the difficulty levels of those problems.

Figure 2: Average running (CPU) times for MURPHY as a function of solution length in Airport planning problems and the difficulty levels of those problems.

Table 1: Average CPU times (seconds) for NuSMV and FD in Airport counter-planning problems P1 – P10.

<table>
<thead>
<tr>
<th>Problem</th>
<th>FD</th>
<th>NuSMV</th>
</tr>
</thead>
<tbody>
<tr>
<td>P01</td>
<td>0.68</td>
<td>2.26</td>
</tr>
<tr>
<td>P02</td>
<td>0.66</td>
<td>2.91</td>
</tr>
<tr>
<td>P03</td>
<td>0.98</td>
<td>45.62</td>
</tr>
<tr>
<td>P04</td>
<td>1.94</td>
<td>–</td>
</tr>
<tr>
<td>P05</td>
<td>2.58</td>
<td>–</td>
</tr>
<tr>
<td>P06</td>
<td>3.33</td>
<td>–</td>
</tr>
<tr>
<td>P07</td>
<td>3.36</td>
<td>–</td>
</tr>
<tr>
<td>P08</td>
<td>5.32</td>
<td>–</td>
</tr>
<tr>
<td>P09</td>
<td>5.07</td>
<td>237.02</td>
</tr>
<tr>
<td>P10</td>
<td>1.79</td>
<td>–</td>
</tr>
</tbody>
</table>

SMV, NuSMV’s input language, based on FD’s translator. FD translates PDDL into SAS+, which can readily be translated into an SMV automaton.

Table reports the average running times of FD and NuSMV to generate counterexamples on the easiest Airport problems. Each data point is an average of the running times of the planners on counter planning problems for each IPC-2004 problem. These results show that FD was significantly faster than NuSMV in generating counterexamples and it was able to do so in large problems. NuSMV could not generate solutions to most of these counter planning problems in 30 minutes, the time limit for our experiments, when the original Airport plans contained more than two airplanes.

In Escape Monster, NuSMV was able to solve only two cases of 40 (2 counter planning variations each of 20 plans), one where it found a trivial counterexample and 11 cases where it was able to demonstrate that no counterexample existed. In no case was NuSMV faster than FD.

To evaluate the capability of FD for counter planning, we performed experiments with the full problem sets from the Airports and Escape Monster domains. As shown by Figure 3, FD was able to quickly search and detect that there was no counterexample to the input counter planning problems. It did not return an answer in 10 cases within the 30 minutes time limit. Figure 4 shows that FD was similarly fast in generating counterexamples: in only 2 cases did it fail to return a solution in the time limit. Figures 5 and 6 show similar results on the Escape Monster domain.

Related Work

The work that is most closely related to MURPHY is work on model-checking verification (Clarke 2008). In model-checking, a formal model of a transition system is checked, through exploration of its reachable state space, to determine whether it satisfies some property or properties. The NuSMV system (Cimatti et al. 2002a), which we used in our comparative experiments, is a high performance model checking system based on the use of binary decision diagrams to compactly encode representations of reachable state space. Often when model checking protocols or controllers, one will take the product composition of the implementation to be verified, together with models of uncontrolled systems (the plant for a controller, or participants in
Figure 3: CPU times for FD in the Airport counter-planning problems, where it performed an exhaustive search but could not find a solution to the problems.

Figure 4: CPU times for FD in the Airport counter-planning problems, where it generated solution counterexamples.

Figure 5: CPU times for FD in the Escape Monster counter-planning problems, where it performed an exhaustive search, without finding a counterexample.

Figure 6: CPU times for FD in Escape Monster, where it generated solution counterexamples.

a protocol), and see if the product satisfies the desired properties. Those verification queries are very similar to ours. There is a close relationship between the heuristic search used in modern classical planners, and that used in model-checkers, as witness the presence of the “Promela” scenarios in the IPC (Hoffmann and Edelkamp 2005). Our experiments show that exploiting the particular features of a PDDL planner for counter planning can outperform the use of a more generic verification system.

We do not know of any such system that uses a counterexample generator like MURPHY to systematically address issues that arise through uncontrolled actions. We believe that a system like MURPHY would be a valuable component in a mixed-initiative planning system.

Replanning and plan repair (e.g., (Yoon, Fern, and Givan 2007; Kambhampati and Hendler 1992)) are closely related, but aim at fixing issues in execution, rather than before. Our work is orthogonal to replanning and plan repair, since our work aims at avoiding the need for repair during execution. That said, our work could usefully be informed by a theory of plan repair during execution. It would be useful if MURPHY could differentiate between counterexamples that would be easy or difficult to repair. For example, if a plan makes an arbitrary choice between moving down corridors $A$ and $B$, which are close to each other, and one can be blocked, that is not as interesting as a case where a plan depends on crossing a single bridge, and that bridge can be blocked, because in the second case it would be harder to fix the plan at execution time.

MURPHY was inspired by compilation techniques that translate more complex problems into classical planning, for solution by “off the shelf” planners (e.g., (Baier, Fritz, and McIlraith 2007; Alford, Kuter, and Nau 2009; Baier, Bacchus, and McIlraith 2007; Palacios and Geffner 2007)).

Conclusions and Future Work

In this paper we have presented an algorithm for translating a classical plan, its domain, and a set of possible disturbances into a counter planning problem that can be solved by a classical planner. The counter planning problem will either yield a counterexample showing how disturbances could cause a plan failure, as guidance for improving the plan, or will report that the plan cannot be adversely affected by the un-

---

4Promela is the input language of the SPIN model-checker.
controlled actions in the model. We presented an empirical demonstration of the translation algorithm and its use in combination with a high-performance planning system.

We are considering a number of refinements and extensions of this work. One extension we are actively investigating is a broader definition of plan failure/counterexample. The definition presented earlier considers as failures only plans where an action cannot be executed, because its preconditions have been clobbered. However, given the presence of conditional effects, an additional class of failures may be considered: a plan all of whose actions can be executed, but which nevertheless fails to achieve its goal because of interference from uncontrolled actions: \( \Pi = \pi | \bar{\pi} \) such that \( \text{result}(\pi_0, \Pi) \not\in g \). Our translation algorithm may readily be extended to cover this class of failures.

We are currently working on developing a variant of Logistics where thieves might steal items from trucks or airplanes without disrupting the movements of those vehicles. Similarly, we are developing a variant of Escape-Monster domain where some monster may destroy the items the explorer collects, instead of killing the explorer, which in turn counter the outcomes of the execution of a plan since the explorer will not be able to eliminate obstacles.

We would also like to investigate realistic limitations on uncontrolled agents. We are considering cases where there are limitations of the number, cost or resources consumed by the uncontrollable actions. In some cases, we would like to constrain the uncontrolled actions to be directed by plausible goals. For example, the trucks in the airports domain might be constrained to motions consistent with moving to and from particular gates. This is quite difficult to do because of the counterfactuals involved: the truck is driving as if the aircraft was not present; if the aircraft was not present, the truck would have reached its chosen destination.

To improve our experimental evaluation, we are currently studying our translation from PDDL into NuSMV. We believe other translations are possible and we will investigate this experimentally. In addition, we are currently working on including the model-checker SPIN in our tests as well.

Finally, we would like to explore counterexample search for more expressive plans, particularly temporal plans and hierarchical plans. These are both more typical of human organizational planning than the simple atemporal plans we have worked on to date.

Acknowledgments
The project or effort depicted is supported by DARPA and the U.S. Army Research Laboratory under contract W911NF-11-C-0037. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Approved for Public Release, Distribution Unlimited.

Thanks to the following people for miscellaneous assistance: to Tom Marble for making it possible for us to perform our Amazon EC2 tests; to Malte Helmert for helpful suggestions and for access to the FASTDOWNWARD planner; to Derek Long for the VAL PDDL validator; to Malte, Derek and Patrik Haslum for discussions about PDDL; to Fondazione Bruno Kessler for making NuSMV freely available.

References


Palacios, H., and Geffner, H. 2007. From conformant into classical planning: Efficient translations that may be complete too. In Boddy, M. S.; Fox, M.; and Thiébaux, S., eds., ICAPS, 264–271. AAAI.
