# Reasoning with Conditional Plans in the Presence of Incomplete Knowledge

**Ronald P. A. Petrick**
**Department of Computer Science**
**University Of Toronto**
**Toronto, Ontario**
**Canada M5S 1A4**
***rpetrick@cs.utoronto.ca***

**Fahiem Bacchus**
**Department of Computer Science**
**University Of Toronto**
**Toronto, Ontario**
**Canada M5S 1A4**
***fbacchus@cs.utoronto.ca***

**Abstract**

We present a simple set of inference rules for reasoning about the effects of actions in a conditional plan. The rules allow us to make additional conclusions about a plan at every stage of its execution, and to augment an agent's knowledge state in the presence of incomplete knowledge. We can use the rules to refine an agent's knowledge after a particular execution of a plan has completed, and to improve an agent's ability to generate plans. Furthermore, this enhancement gives us the ability to plan for certain types of temporally oriented goals, e.g., goals that require some initial state condition be restored by the end of the plan. We have implemented this mechanism inside of a planner, and we demonstrate the planner's increased ability to solve a variety of interesting planning problems.

## Introduction

In this paper we address the problem of controlling agents under conditions of accurate but incomplete knowledge. A common example of this scenario would be that of a software agent that can reasonably assume the accuracy but not the completeness of its knowledge.

To achieve control over such environments the agent must be able to use its current incomplete knowledge to generate plans that it can infer will achieve its goals. This inference is complicated by the fact that it must be performed entirely at plan time, prior to receiving any information that might be generated by its execution. Furthermore, after the agent has executed the plan it must be able to update its knowledge so as to make that knowledge as complete as possible given information gathered during the plan's execution. Both of these tasks require an ability to reason about the effects of a plan that is about to be, or has just been, executed. We present a mechanism for accomplishing this kind of reasoning within the framework developed in (Petrick and Bacchus 2002), and demonstrate how it can be used to reach further conclusions after a plan has been executed and to augment our ability to generate correct plans.

The utility of reasoning with plans is best illustrated by a series of examples.[1] Say that we have a bottle of liquid, a healthy lawn, and two actions: pour-on-lawn and sense-lawn. Pour-on-lawn pours some of the liquid on the lawn, and also has the conditional effect that if the liquid is poisonous the lawn becomes dead in the successor state. Sense-lawn simply senses whether or not the lawn is dead. Say that we execute the sequence of actions ⟨pour-on-lawn, sense-lawn⟩ after which we come to know that the lawn is dead. An intuitively obvious additional conclusion is that the liquid is poisonous. The question is: how do we automate this kind of inference?

It should be noted that the conclusion "poisonous" requires non-trivial inference. It does not follow from either of the individual actions executed. The pour-on-lawn action in of itself provides no information about whether or not it changed the state of the lawn, so we cannot know if poisonous holds after the action is executed. Similarly, sense-lawn simply returns the status of the lawn; by itself it says nothing about how the lawn became dead. Further evidence that a non-trivial inference process is at work is provided when we consider our knowledge that in the initial state the lawn is not dead. It is not hard to see that without this knowledge the conclusion poisonous is not justified.

An elaboration of this example is where we have an additional action drink, with the conditional effect that if the liquid is poisonous then we have been poisoned. If the action sequence ⟨drink, pour-on-lawn, sense-lawn⟩ results in sensing a dead lawn, we can also conclude that we have been poisoned. Notice that here we need to infer not only that the liquid is poisonous in the final state of the execution, as we did in the previous example, but rather that the liquid was in fact poisonous in the *initial state*. It is only when we can reach this temporally indexed conclusion that we can also conclude that the drink action caused poisoned: that action was executed in the initial state not in the final state.

These kinds of temporally indexed conclusions are also needed when trying to achieve restore goals (Golden and Weld 1996). Restore goals require that the final state return a condition to the status it had in the initial state. We might not know what the initial status of the condition was, however. Hence, it can be difficult to infer that a plan does in fact restore this status. With additional reasoning we can sometimes infer the initial status of the condition, and thus be in a position to try to ensure that the plan restores it.

Finally, consider the effects of the plan ⟨pour-on-lawn, sense-lawn⟩ *prior* to its execution. Before the plan has been executed, we do not know what the outcome of the sense-

[1]The first of these was communicated to us by David Smith.

lawn action will be. All we know at plan time is that in the initial state we have that the lawn is not dead. Nevertheless, by reasoning about the two possible outcomes of the sense-lawn action, we can conclude that if the outcome is "not lawn-dead" we will come to know that the liquid is not poisonous, and that if the outcome is "lawn-dead" we will come to know that the liquid is poisonous. Thus we can conclude at plan time that, irrespective of the actual outcome of the plan, we will come to know whether the liquid is poisonous.

The rest of the paper is organized as follows. First, we examine in greater detail the inferences that are required to solve the above examples. To automate these inferences we need a formal framework, so next we present a previously developed framework for planning under incomplete knowledge (Bacchus and Petrick 1998; Petrick and Bacchus 2002). Using this framework, we present a method for automating the required inferences, and show how the mechanism can be used to extend the planner's ability to generate plans. Finally, we illustrate the power of this approach with a number of examples that have been solved by our extended planner.

## Required inferences

To automate the kinds of inferences presented above we must analyze in more detail the steps involved. A critical element in these inferences is the Markov assumption described in (Golden and Weld 1996) which we will make throughout this paper. The Markov assumption has two parts. First is an assumption that we have complete knowledge of action effects and non-effects.[2] The second component of the Markov assumption is that the agent's actions are the only source of change in the world. Unlike the previous case, this assumption is restrictive and has to be examined carefully when dealing with other agents (or nature) that could be altering the world concurrently. Nevertheless, even in domains where there is concurrent activity, the agent might still know that (or reasonably assume that) the facts it is reasoning about were only changed by its own actions.

Consider the action sequence ⟨pour-on-lawn, sense-lawn⟩ after it has been executed and we have come to know that the lawn is dead. This action sequence produces a sequence of three worlds: $W_0$ the initial world, $W_1$ the world after executing pour-on-lawn, and $W_2$ the world after executing sense-lawn. We know that lawn-dead does not hold in $W_0$, and that lawn-dead holds in $W_2$. Reasoning backwards we see that sense-lawn does not change the status of lawn-dead. Hence lawn-dead must have held in $W_1$. But then, not lawn-dead held in $W_0$ and lawn-dead held in $W_1$, which means that pour-on-lawn must have produced a change in lawn-dead. Since lawn-dead is altered by a conditional effect of pour-on-lawn, it must be that the antecedent of the condition, poison, was true in $W_0$ when pour-on-lawn was executed. Furthermore, poison is not affected by pour-on-lawn, nor by sense-lawn. Hence, poison must be true in $W_1$ as well as in $W_2$.

[2] In general, action non-determinism can be pushed into the non-determinism of the state in which it is being executed (e.g., (Bacchus *et al.* 1999)). Hence, we lose no generality with this assumption.

When we extend the action sequence to ⟨drink, pour-on-lawn, sense-lawn⟩, with corresponding world sequence $W_0$, $W_1$, $W_2$, $W_3$, we first need to infer that since drink does not alter lawn-dead, not lawn-dead must hold in $W_1$. Using the same reasoning as above we can conclude that poison holds in $W_1$, and since poison is not changed by drink it also holds in $W_0$. Now, since drink was executed in $W_0$, we conclude that it must have created the effect poisoned in $W_1$. Poisoned is unaffected by pour-on-lawn and sense-lawn, hence we conclude poisoned also holds in $W_2$ and $W_3$.

Finally, if we want to reason at plan time about the plan ⟨pour-on-lawn, sense-lawn⟩, we can consider the two possible outcomes of sense-lawn: either it senses that the lawn is dead, or it senses that the lawn is not dead. If it sensed lawn-dead, the previous example shows that we would know lawn-dead in the final state of the plan. If it sensed not lawn-dead, we perform the same type of inferences, but at the critical step we conclude that since not lawn-dead is true in $W_0$ as well as $W_1$, pour-on-lawn did not alter lawn-dead, and hence the antecedent of its conditional effect must not have been true in $W_0$. That is, not poisonous must have been true in $W_0$, and since poisonous is not changed by the two actions, it must also be true in the final state of the plan. Hence, irrespective of the actual outcome of executing the plan, the agent will arrive in a state where it either knows poison or knows not poison, and we can conclude that the plan will allow the agent to know whether or not poison.

These examples show that the basic inferences required to reach the conclusions we want are fairly simple. Nevertheless, they can add significantly to the agent's ability to deal with incompletely known environments.

## PKS

To realize these kinds of inferences we cast our work in the formal framework for planning under incomplete knowledge developed in (Bacchus and Petrick 1998; Petrick and Bacchus 2002). In this section we present an overview of this framework, but first we discuss some alternative formalisms.

A number of other works have addressed the problem of planning under incomplete information, e.g., (Pryor and Collins 1996; Bertoli *et al.* 2001; Bonet and Geffner 2000; Anderson *et al.* 1998; Golden and Weld 1996).

A recent approach to planning under incomplete knowledge has been to find ways of efficiently representing all of the possible configurations of the world (possible worlds) that are compatible with the agent's knowledge. The main technique is to utilize symbolic representations, BDDs (Bryant 1992), to compactly represent this set of possible worlds. Action effects are then reasoned about by examining their effect on the entire set of possible worlds. Although this approach can in some instances be quite efficient, its ultimate scalability remains a question. In particular, the number of possible worlds grows exponentially with the number of distinct fluents in the world, so it is not clear that even compact symbolic representations will remain compact.

The SADL representation of (Golden and Weld 1996) is closer to the approach we take here. SADL examines the middle ground between expressiveness and the complexity of reasoning. Whereas representing the complete set

of possible worlds provides a high degree of expressiveness, it also provides the worst complexity. SADL, like our approach, tries to restrict expressiveness in order to achieve greater efficiency. The language was also designed to address very similar issues in planning under incomplete knowledge. However, the approach to reasoning with plans (regression) and planning (partial order planning) are very different to the approach we use here, and empirical evidence from previously published works suggests that their approach is much less efficient.

Our approach utilizes the PKS (Planning with Knowledge and Sensing) framework (Petrick and Bacchus 2002). This framework is based on a generalization of STRIPS. In STRIPS, the state of the world is represented by a database and actions are represented as updates to that database. In PKS, the agent's knowledge (rather than the state of the world) is represented by a set of databases; actions are represented as updates to these databases. Thus, actions are represented at the knowledge level as modifications to the agent's knowledge (the databases) rather than as modifications to the state of the world.

Modelling actions as database updates leads naturally to a simple forward-chaining approach to finding plans, and in (Petrick and Bacchus 2002) empirical evidence is provided to demonstrate the efficiency and effectiveness of this approach. The database representation of the agent's knowledge is computationally efficient, but restricts the types of knowledge that can be expressed. In particular, it places strong limits on disjunctive knowledge in order to achieve its efficiency.

In somewhat more detail, PKS utilizes four different databases to represent the agent's knowledge. The contents of these databases are formalized using a standard first-order modal logic of knowledge. In particular, there is a fixed mapping between the contents of the databases and a collection of formulas of the modal logic. Thus, any configuration of the databases corresponds to a set of logical formulas that precisely characterize the agent's knowledge state. The details of this mapping are given in (Bacchus and Petrick 1998). Briefly, the different databases utilized are as follows:

**$K_f$:** The first database is much like a standard STRIPS database, except that both positive and negative facts are allowed and we do not apply the closed world assumption. In particular, $K_f$ can include any ground literal, $\ell$, and intuitively $\ell \in K_f$ means that we know $\ell$. $K_f$ can also contain formulas specifying knowledge of the value of various functions on fixed arguments.

**$K_w$:** The second database is designed to address plan time reasoning about sensing actions. If the plan contains an action to sense the fluent $f$, at plan time all that the agent will know is that *after* it has executed the action it will either know $f$ or know $\neg f$. At plan time the actual value of this fluent remains unknown. Intuitively, $\phi \in K_w$ means that the agent either knows $\phi$ or knows $\neg\phi$, and that at execution time this disjunction will be resolved. For example, the action sense-lawn adds lawn-dead to $K_w$: i.e., the effect of this action on the agent's knowledge is to place it in a state where it knows whether or not lawn-dead is true.

$K_w$ plays a particularly important role at plan time when it comes to generating conditional plans that branch based on information that the agent will resolve at execution time. In a conditional plan it is only legitimate to branch on "know-whether" facts. In particular, we are guaranteed that at execution time the agent will have sufficient information at that point in the plan's execution to know which branch to take. This guarantee satisfies one of the important conditions for plan correctness in the context of incomplete knowledge put forward in (Levesque 1996).

**$K_v$:** The third database is a specialized version of $K_w$ designed to store information about various function values the agent will come to know at execution time. $K_v$ can contain any unnested function term whose value is guaranteed to be known to the agent at execution time. $K_v$ is used for plan time modelling of sensing actions that return numeric values. For example, $\mathit{size}(\mathit{paper.tex}) \in K_v$ means the agent knows that at execution time the size of *paper.tex* will become known.

**$K_x$:** The fourth database contains information about a particular type of disjunctive knowledge, namely "exclusive or" knowledge of literals. Entries in $K_x$ are of the form $(\ell_1|\ell_2|\ldots|\ell_n)$, where each $\ell_i$ is a ground literal. Intuitively, such a formula represents knowledge of the fact that "exactly one of the $\ell_i$ is true." In particular, if one of these literals becomes known, we immediately come to know that the other literals are false. For example, if $(\mathit{infected}(I_1)|\mathit{infected}(I_2)) \in K_x$ then the agent knows that one and only one of $\mathit{infected}(I_1)$ or $\mathit{infected}(I_2)$ is true. This form of incomplete knowledge is common in planning.

On top of these databases PKS implements an inference algorithm that works by examining the database contents to draw various conclusions about what the agent does and does not know (or know-whether). The inference algorithm is efficient but incomplete and is presented in (Bacchus and Petrick 1998). It is used to determine whether or not an action's preconditions hold, whether or not various conditional effects of the action should be activated, and whether or not the plan achieves the stated goal.

As mentioned above, actions are represented as updates to the set of databases, with these updates representing the effects of the actions on the agent's knowledge. This approach is best illustrated by formalizing the *pour-on-lawn*, *drink*, and *sense-lawn* actions from the poisonous liquid domain; these actions are represented in Table 1. We have simplified the actions by omitting any preconditions (i.e., assuming that the liquid is available, we are near the lawn, etc.).

If the agent does not know for certain that the liquid is not poisonous, *pour-on-lawn* will remove $\neg$*lawn-dead* from $K_f$ (i.e., the agent will no longer know that the lawn is not dead). If the agent knows that the liquid is poisonous, *pour-on-lawn* will add *lawn-dead* to $K_f$ (i.e., the agent will come to know that the lawn is dead). *drink* has a similar kind of effect on the agent's knowledge of *poisoned*. *sense-lawn* on the other hand puts the agent in a knowledge state where it knows

| Action | Pre | Effects |
|---|---|---|
| *pour-on-lawn* | | $\neg K(\neg poisonous) \Rightarrow$ $del(K_f, \neg lawn\text{-}dead)$ $K(poisonous) \Rightarrow$ $add(K_f, lawn\text{-}dead)$ |
| *drink* | | $\neg K(\neg poisonous) \Rightarrow$ $del(K_f, \neg poisoned)$ $K(poisonous) \Rightarrow$ $add(K_f, poisoned)$ |
| *sense-lawn* | | $add(K_w, lawn\text{-}dead)$ |
| *pour-on-lawn-2* | | $\neg K(\neg poisonous\text{-}2) \Rightarrow$ $del(K_f, \neg lawn\text{-}dead)$ $K(poisonous\text{-}2) \Rightarrow$ $add(K_f, lawn\text{-}dead)$ |

Table 1: Actions in the poisonous liquid domain

whether *lawn-dead* is true or not.

If we start in an initial state where $K_f$ contains $\neg$*lawn-dead* and all of the other databases are empty, executing *pour-on-lawn* yields a state where $\neg$*lawn-dead* has been removed from $K_f$—the agent no longer knows that the lawn is not dead. If we then execute *sense-lawn* we will arrive at a state where $K_w$ now contains *lawn-dead*—the agent knows whether the lawn is dead. This forward application of the actions serves to capture their basic knowledge effects, and as shown in (Petrick and Bacchus 2002), a forward chaining engine utilizing this mechanism to search for conditional plans can very efficiently and effectively solve a range of interesting problems. However, we also see that the forward application of actions does not allow the agent to conclude that this plan also achieves know-whether knowledge of *poisonous*. As we illustrated above, such a conclusion can be reached by fairly simple inferences. In the next section we show how the PKS approach can be extended to realize these kinds of inferences.

## Reasoning about a PKS plan

PKS constructs conditional plans. A conditional plan is a tree, whose nodes are labelled by a knowledge state (represented by a set of databases), and whose edges are labelled by an action or by a sensed fluent. If a node $n$ has a single child $c$ the edge to that child is labelled by an action $a$, whose preconditions must be entailed by $n$'s knowledge state. The label for the child $c$ ($c$'s knowledge state) is computed by applying $a$ to $n$'s label. A node can also have two children, in which case each edge is labelled by a fluent $F$, such that $K(F) \vee K(\neg F)$ is entailed by the node's knowledge state (i.e., the agent must know-whether the fluent that the plan branches on). In this case the label for one child is computed by adding $F$ to the parent's $K_f$, and the label for the other child by adding $\neg F$ to the parent's $K_f$.

PKS searches over the space of conditional plans by using the forward application of actions to incrementally construct new plans. It uses the inference algorithm described in (Bacchus and Petrick 1998) to compute whether or not an action can be applied to a leaf node (to extend a conditional plan), to compute the effects of the action so as to generate the new child, and to test whether or not the leaf achieves the goal. It terminates its search when it has found a conditional plan in which all the leaf nodes achieve the goal. Furthermore, it does not extend any leaf that already achieves the goal.

Reasoning with these conditional plans is implemented by first building a set of linearizations of the tree structured plan. Each path to a leaf becomes a linear sequence of states and actions: the states and actions visited during that particular execution of the plan. The number of linearizations is simply the number of leaves in the conditional plan, so only a linear amount of extra space is required to convert the condition plan (tree) into a set of linear plans (the branches of the tree). Each path differs from other paths in the manner in which the agent's know-whether knowledge resolved itself during execution and in the manner in which that resolution affected the actions the agent subsequently executed. For each linear sequence, we then apply a set of backward and forward inferences to draw additional conclusions along that sequence.

Let $W$ be a knowledge state in the linear sequence, $W^+$ be its successor state, and $a$ be the label of the edge from $W$ to $W^+$. The basic inference rules we apply are:

1. If $a$ cannot make $\phi$ false (e.g., $\phi$ is unrelated to any of the facts $a$ makes true), then if $\phi$ becomes newly known in $W$ make $\phi$ known in $W^+$. Similarly, if $a$ cannot make $\phi$ true, then if $\phi$ becomes newly known in $W^+$ make $\phi$ known in $W$. In both cases $a$ cannot have changed the status of $\phi$ between the two worlds $W$ and $W^+$.
2. If $\phi$ becomes newly known in $W$ and $a$ has the conditional effect $\phi \rightarrow \psi$, make $\psi$ known in $W^+$. $\psi$ must be true in $W^+$ as either it was already true or $a$ made it true.
3. If $a$ has the conditional effect $\psi \rightarrow \phi$ and it becomes newly known that $\phi$ holds in $W^+$ and $\neg\phi$ holds in $W$, make $\psi$ known in $W$. It has become known that $a$'s conditional effect was activated, so the antecedent of this effect must have been true.
4. If $a$ has the conditional effect $\psi \rightarrow \phi$ and it becomes newly known that $\neg\phi$ holds in $W^+$, make $\neg\psi$ known in $W$. It has become known that $a$'s conditional effect was not activated, so the antecedent of this effect must have been false.

Although these rules are easily shown to be sound under the assumption that we have complete information about $a$'s effects, they are too general to implement efficiently. In particular, PKS achieves its efficiency by avoiding disjunctions, hence we cannot use these rules to infer new disjunctions.

In our implementation we restrict these rules to apply to literals (i.e., $\phi$ and $\psi$ are restricted to be literals), and further we restrict our actions so that they cannot add or delete a fluent $F$ with more than one conditional effect. For example, an action cannot contain two conditional effects $a \rightarrow F$ and $b \rightarrow F$.[3]

Note that this restriction does not prohibit $\phi$ and $\psi$ from being parameterized, provided that such parameters are

[3] If this was allowed, rule 3 above would be invalid. The correct inference from knowing $F$ in $W$ and $\neg F$ in $W^+$ would be $a \vee b$, which is a disjunction that we cannot represent. This observation was pointed out to us by Tal Shaked.

among the parameters of the action (i.e., they are not "free"). There are also cases where these rules can be applied to more complex formulas without yielding disjunctions. However, implementing such an extension remains as future work.

To apply these inference rules we simply need a scheme for controlling their activation, and a scheme for initiating the process. We use a stack to keep track of states that have been updated. Every time we pop a changed state from the stack we check to see if its predecessor or successor state should also be updated. If so, we push that state onto the stack. To initiate the process we first push onto the stack all of the states whose edges are labelled with fluents. These are fluents that were added by branching on know-whether knowledge. For simplicity we then delete the parents of these states from the sequence, since the parents are identical to the child except for the addition of the new fluent. This yields a sequence in which every edge is labelled with an action.

Applying the new inference procedure to a conditional plan means that we must apply the set of inference rules to each linearization of the plan, where the number of linear plans is equal to the number of leaf nodes in the conditional plan. To test whether or not one of the inference rules should be applied, the standard PKS inference algorithm is used to test the rule conditions against a given state in the plan. Thus, testing the inference rules has the same complexity as evaluating whether or not an action's preconditions hold.

Applying the effects of an inference rule is simply a constant time update of the databases—similar to applying the effects of an action, but each such update initiates a recursive application of the inference rules. Further successful firings of the inference rules may result in the same state being considered more than once, as the effects of the rules are propagated both forwards and backwards in the plan. However, no effect of any action can be applied more than once in the same linear plan. In the worst case all the conditional effects of the actions in a linear plan could be applied, with each application requiring us to examine all of the states in the linear plan to see if any of the inference rules can be fired. Hence, if the conditional plan contains $n$ leaves, and has height at most $d$, the inference procedure in the worst case might require $O(nd^2)$ testings of the inference rules: there are $n$ linear plans, each containing at most $O(d)$ different conditional effects ($d$ is the maximum number of actions in any linearization). Each of these conditional effects might require a backward and forward testing and application of the inference rules over the $d$ states in the linear plan.

In practice however, the number of effects that are applied at each state are quite small and the inference procedure can be applied to a plan quite efficiently.

We illustrate the operation of this process on two plans involving the poisonous liquid domain described above. In Figure 1, we consider the conditional plan $\langle$*pour-on-lawn*, *sense-lawn*$\rangle$ followed by a branch on knowing whether *lawn-dead*. This plan is shown at the top of the figure, along with the contents of the databases as described in the previous section. Applying the above reasoning procedure we obtain two linearizations, as shown in (a) and

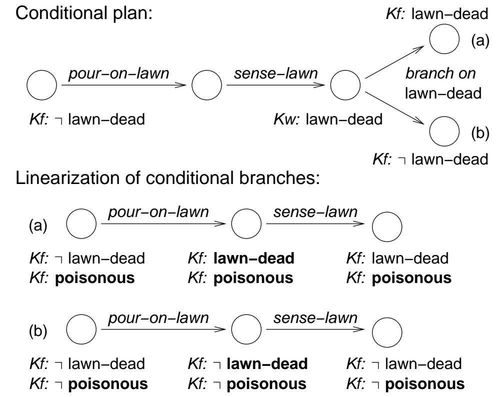


Figure 1: Reasoning in the poisonous liquid domain

(b). The additional conclusions achieved by applying the inference rules are shown in bold. The net result is that we have proved that in every outcome of the plan the agent either knows *poisonous* or knows $\neg$*poisonous*; i.e., the plan achieves know-whether knowledge of *poisonous*.

In Figure 2, we consider a variation of the previous plan that includes an additional action *pour-on-lawn-2* (see Table 1), occurring immediately after the *pour-on-lawn* action, but prior to *sense-lawn*. The action *pour-on-lawn-2* has the effect of pouring a second unknown liquid onto the lawn. The effects of *pour-on-lawn-2* are similar to those of *pour-on-lawn*; the second liquid may be poisonous and, thus, kill the lawn. In this case we also have two linearizations. In (a), we can only apply rule 1 to assert *lawn-dead*, as shown in bold in the figure. Since *pour-on-lawn-2* and *pour-on-lawn* both have conditional effects involving *lawn-dead*, we cannot make any additional conclusions about *lawn-dead* across these actions. As a result, no further reasoning rules are applied. Intuitively, this reasoning is correct: the agent is unable to determine which liquid killed the lawn and, therefore, cannot conclude which of the liquids is poisonous. In (b), after applying the inference rules we are able to establish the same conclusions as in Figure 1(b): $\neg$*lawn-dead* and $\neg$*poisonous* hold at each state. Furthermore, the inference rules also assert that $\neg$*poisonous-2* must hold in each state in the plan. Again, intuitively, these conclusions make sense: after sensing the lawn and determining it is not dead, the agent can conclude that neither liquid must be poisonous. Note that this plan would be rejected by the planner since linearization (a) fails to establish that the agent has know-whether knowledge of *poisonous* (or *poisonous-2*).

## Extending PKS's ability to plan

The above inference procedure allows us to extend PKS in a simple manner: a conditional plan generated by PKS during search is augmented by applying the effects of the inference procedure to the knowledge states of the plan. These changes enhance PKS's ability to generate plans in two ways. First, plans that PKS may previously have rejected as not achieving the goal may now be proven to satisfy the goal. For example, if PKS is searching for a plan that achieves

| Action | Pre | Effects |
|---|---|---|
| *paint*$(x)$ | $K(colour(x))$ | $add(K_f, (door\text{-}colour) = x)$ |
| *sense-colour* | | $add(K_v, (door\text{-}colour))$ |

Table 2: Painted door action specification

know-whether knowledge of *poisonous* it will now be able to conclude that ⟨*pour-on-lawn*, *sense-lawn*⟩ is a solution, as shown above. Previously this plan would have been rejected, and in fact PKS would not have been able to solve this problem at all. Second, since the inference procedure augments all of the knowledge states in the plan, we now have the potential of being able to solve more complicated temporal goals that reference states other than the final state. Since PKS uses search to find a plan, our ability to detect when a plan achieves a more complex goal allows us to determine when the search has succeeded.

We have enhanced PKS by implementing the inference procedure described above, and by extending the goal language to support the expression of more complex goals. In particular, the new implementation supports three types of goal conditions: (1) conditions that must hold in the final state of the plan, (2) conditions that make reference to the initial state, and (3) conditions that must hold of every state that could be visited by the plan. Goals may be specified as conjunctions of these conditions. Conditions of type (1) can be used to express classical goals of achievement. The addition of type (2) conditions allows, for instance, restore goals to be expressed. Conditions of type (3) can be used to express "hands-off" or safety goals (Golden and Weld 1996).[4]

We now illustrate PKS's enhanced planning abilities with a series of examples. First, however, we note that our current implementation is very much at the proof of concept stage. For example, it employs blind search to find plans. Nevertheless, it is able to solve *all* of the examples given below in time that is less than the resolution of our timers (less than 1 or 2 milliseconds). It should be noted that planners that represent all of the possible worlds (and thus deal with disjunction), are also able to obtain the conclusions of the examples above. In particular, the above examples are all propositional, and do not utilize PKS's ability to deal with non-propositional problems, e.g., those involving functions.

**Poisonous liquid:** When given the actions of the poisonous liquid domain (Table 1), PKS can immediately find the plan ⟨*pour-on-lawn*, *sense-lawn*⟩ to achieve the goal of knowing whether *poisonous*. It can also conclude *poisoned*, when presented with the execution sequence ⟨*drink*, *pour-on-lawn*, *sense-lawn*⟩ resulting from sensing a dead lawn. In both cases, it can conclude that *poisonous* held in the initial state. As mentioned above, prior to the extension, PKS was previously unable to solve this problem.

**Painted door:** Say that we have the two actions given in Table 2. *paint* changes the colour of a door (represented as a

[4]Our current implementation does not handle the quantified goals expressible in SADL (Golden and Weld 1996). However, it should be noted that SADL is a representation language, not a planning system. To our knowledge no implemented planner supported all of SADL.

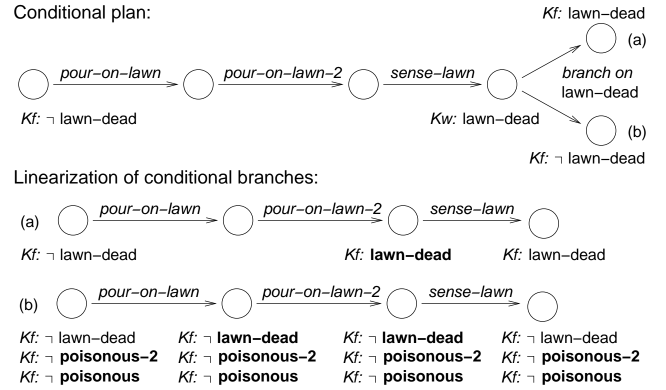


Figure 2: Poisonous liquid domain with two liquids

0-ary function (*door-colour*)) to an available colour $x$, while *sense-colour* senses the value of (*door-colour*). Our goal is a "hands-off" goal of coming to know the colour of the door while ensuring that the colour is never changed by the plan.

First, suppose that the door is known to be one of two colours $c_1$ or $c_2$.[5] This knowledge can be represented by including $((door\text{-}colour) = c_1 | (door\text{-}colour) = c_2)$ in the exclusive or $K_x$ database of the initial state. During its search PKS will consider the single step plan ⟨*sense-colour*⟩. After this action the planner has (*door-colour*) in its $K_v$ database, indicating that it knows the value of this function. The planner can then use its exclusive or knowledge and the fact that it can branch on $K_v$ knowledge (just as it can branch on know-whether knowledge) to construct a two-way branch on the possible values of (*door-colour*). Along one branch the planner asserts that $(door\text{-}colour) = c_1$; along the other branch it asserts that $(door\text{-}colour) = c_2$. Applying the inference procedure to the two branches of the conditional plan allows the planner to conclude that (a) in the last state of each branch it knows the value of (*door-colour*), and (b) (*door-colour*) has the same value in every state of each branch.[6] Thus, it can conclude that ⟨*sense-colour*⟩ achieves the goal.

On the other hand, when PKS examines a plan like ⟨*paint*$(c_1)$⟩[7] it will not know the value of (*door-colour*) in the initial state. Since *paint* changes the value of (*door-colour*), the inference rules will not allow facts about (*door-colour*) to be passed back through *paint*. Thus, PKS cannot conclude that (*door-colour*) remains the same throughout the plan, and plans involving *paint* are rejected as not achieving the goal.

Our current implementation solves this problem without difficulty, but there is a natural generalization that we have not yet implemented. In the above example, we had to assume a finite set of known colours for the door. In general,

[5]Any finite set of known colours will also work.

[6]*sense-colour* does not change (*door-colour*), so the value of this function is passed back to the initial state through the action. That is, in each linearization (*door-colour*) has a different value in the initial state, but its value agrees with its value in the final state.

[7]Say that *colour*$(c_1)$ and *colour*$(c_2)$ are both initially known.

| Action | Pre | Effects |
|---|---|---|
| $ls(d)$ | $K(dir(d))$ | $add(K_w, exec(d))$ |
| $chmod{+}x(d)$ | $K(dir(d))$ | $add(K_f, exec(d))$ |
| $chmod{-}x(d)$ | $K(dir(d))$ | $add(K_f, \neg exec(d))$ |
| $cp(f, d)$ | $K(file(f))$ $K(dir(d))$ $K(exec(d))$ | $add(K_f, indir(f, d))$ |
| $cp^+(f, d)$ | $K(file(f))$ $K(dir(d))$ | $K(exec(d)) \Rightarrow add(K_f, indir(f, d))$ $add(K_w, indir(f, d))$ |

Table 3: UNIX domain action specification

we might not know the range of possible colours. We can solve this problem by inserting into the plan an assumption that a $K_v$ function has an arbitrary new value, rather than inserting a multi-way branch on a set of known possible values for the function. That is, at the end of the plan ⟨*sense-colour*⟩, where we have (*door-colour*) in $K_v$, we can add the new assertion $(door\text{-}colour) = c$ to $K_f$, where $c$ is a new constant (essentially a Skolem constant). Our inference mechanism will then conclude that $(door\text{-}colour) = c$ in the initial state as well. Furthermore, since $c$ is an arbitrary constant about which nothing is known, we can conclude that the value of the function (*door-colour*) would be preserved no matter what colour it was, and thus that ⟨*sense-colour*⟩ achieves the goal even if the range of door colours is unknown or infinite.

**UNIX domain:** Say that we have the simplified UNIX actions given in Table 3. Initially, we know about the existence of certain files and directories, specified by the $file(f)$ and $dir(d)$ predicates, some of their locations, specified by the $indir(f, d)$ predicate, and that some directories are executable, specified by the $exec(d)$ predicate. The action $ls(d)$ senses the executability of a directory $d$; $chmod{+}x(d)$ and $chmod{-}x(d)$ respectively set and delete the executability of a directory; and $cp(f, d)$ copies a file $f$ into directory $d$, provided the directory is executable. The goal in this domain is to copy files into certain directories, while restoring the executability conditions of these directories.

Let the planner have the initial knowledge $dir(icaps)$, $file(paper.tex)$, and $\neg indir(paper.tex, icaps)$. The planner has no initial knowledge of the executability of the directory *icaps*. Let the goal be that we come to know $indir(paper.tex, icaps)$ and that we restore the executability status of *icaps* (i.e., that $exec(icaps)$ has the same value at the end and the beginning of the plan). The value of $exec(icaps)$ may change during the plan, provided it is restored to its original value by the end of the plan.

PKS finds the conditional plan: $ls(icaps)$; branch on $exec(icaps)$: if $K(exec(icaps))$ then $cp(paper.tex, icaps)$, otherwise $chmod{+}x(icaps)$; $cp(paper.tex, icaps)$; $chmod{-}x(icaps)$.

Since the executability of *icaps* is not known initially, the *ls* action is necessary to sense the value of $exec(icaps)$. The new inference rules establish that this sensed value must also hold in the initial state, since *ls* does not change the value of *exec*. The second goal can then be established by testing the initial value of $exec(icaps)$ against its value in the final state(s) of the plan. By reasoning about the possible values of $exec(icaps)$, appropriate plan branches can be built to ensure the first goal is achieved (the file is copied) and the executability permissions of the directory are restored along the branch where we had to modify these permissions.

We also consider a related example with a new version of the *cp* action, $cp^+$ (also given in Table 3). Unlike *cp*, $cp^+$ does not require that the directory be known to be executable, but returns whether or not the copy was successful. In this case PKS finds the conditional plan: $cp^+(paper.tex, icaps)$; branch on $indir(paper.tex, icaps)$: if $K(indir(paper.tex, icaps))$ do nothing, otherwise $chmod{+}x(icaps)$; $cp(paper.tex, icaps)$; $chmod{-}x(icaps)$. In other words PKS is able to reason from $cp^+$ failing to achieve $indir(paper.tex, icaps)$ that *icaps* was not initially executable.

## Conclusions

We have presented a simple mechanism for reasoning about the knowledge effects of conditional plans. This mechanism allows us to extend an existing planner so that it can solve a more interesting range of problems. In future work we plan to investigate extensions to deal with function terms whose range is unknown, to progress and regress more complex formulas, and to provide more sophisticated search control to allow PKS to scale to bigger problems.

## References


Corin R. Anderson, Daniel S. Weld, and David E. Smith. Extending graphplan to handle uncertainty & sensing actions. In AAAI-1998, pages 897–904.

Fahiem Bacchus and Ron Petrick. Modeling and agent's incomplete knowledge during planning and execution. In KR-1998, pages 432–443.

Fahiem Bacchus, Joseph Y. Halpern, and Hector J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence*, 111:171–208, 1999.

Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In IJCAI-2002, pages 473–478.

Blai Bonet and Héctor Geffner. Planning with incomplete information as heuristic search in belief space. In AIPS-2000, pages 52–61.

R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. In KR-1996, pages 174–185.

Hector J. Levesque. What is planning in the presence of sensing? In AAAI-1996, pages 1139–1146.

Ron Petrick and Fahiem Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In AIPS-2002, pages 212–222.

L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.