Extending the Knowledge-Based Approach to Planning with Incomplete Information and Sensing*

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

In (Petrick and Bacchus 2002), a "knowledge-level" approach to planning under incomplete knowledge and sensing was presented. In comparison with alternate approaches based on representing sets of possible worlds, this higher-level representation is richer, but the inferences it supports are weaker. Nevertheless, because of its richer representation, it is able to solve problems that cannot be solved by alternate approaches.

In this paper we examine a collection of new techniques for increasing both the representational and inferential power of the knowledge-level approach. These techniques have been fully implemented in the PKS (Planning with Knowledge and Sensing) planning system. Taken together they allow us to solve a range of new types of planning problems under incomplete knowledge and sensing.

Introduction

Constructing conditional plans that employ sensing and must operate under conditions of incomplete knowledge is a challenging problem, but a problem that humans deal with on a daily basis. Although planning in this context is generally hard—both theoretically and practically—there are many situations where "common-sense" plans with fairly simple structure can solve the problem.

In (Petrick and Bacchus 2002), a "knowledge-level" approach to planning with sensing and incomplete knowledge was presented. The key idea of this approach is to represent the agent's knowledge state using a *first-order language*, and to represent actions by their effects on the agent's *knowledge*, rather than by their effects on the environment.

General reasoning in such a rich language is impractical, however. Instead, we have been exploring the approach of using a restricted subset of the language and a limited amount of inference in that subset. The motivation for this approach is twofold. First, we want to accommodate non-propositional features in our representation, e.g., functions and variables. Second, we are motivated more by the ability to automatically generate "natural" plans, i.e., plans that humans are able to find and that an intelligent agent should be able to generate, rather than by the ability to generate all possible plans—humans cope quite well with incomplete knowledge of their environment even with limited ability to generate plans. In this paper we present a collection of new techniques for increasing the representational and inferential power of the knowledge-based approach.

An alternate trend in work on planning under incomplete knowledge, e.g., (Bertoli *et al.* 2001; Anderson, Weld, and Smith 1998; Brafman and Hoffmann 2003), has concentrated on propositional representations over which complete reasoning is feasible. The common element in these works has been to represent the set of all possible worlds (i.e., the set of all states compatible with the agent's incomplete knowledge) using various techniques, e.g., BDDs (Bryant 1992), Graphplan-like structures (Blum and Furst 1997), or clausal representations (Brafman and Hoffmann 2003). These techniques yield planning systems that are able to generate plans requiring complex combinatorial reasoning. Because the representations are propositional, however, many natural situations and plans cannot be represented.

The difference in these approaches is well illustrated by Moore's classic open safe example (Moore 1985). In this example there is a closed safe and a piece of paper on which the safe's combination is written. The goal is to open the safe. Our planning system, PKS, based on the knowledge-level approach, is able to generate the obvious plan $\langle \mathit{readCombo}; \mathit{dial}(\mathit{combo}())\rangle$: first read the combination, then dial it. What is critical here is that the value of $\mathit{combo}()$ (a 0-ary function) is unspecified by the plan. In fact, it is only at execution time that this value will become known.[1] At plan time, all that is known is that the value will *become* known at this point in the plan's execution. The ability to generate parameterized plans containing run-time variables is useful in many planning contexts. Propositional representations are not capable of representing such plans, and thus, approaches based on propositional representations cannot generate such plans (at least not without additional techniques that go beyond propositional reasoning).

There are also a number of examples of plans that can be generated by "propositional possible worlds" planners that cannot be found by PKS because of its more limited inferential power. As mentioned above, we would not be so concerned if complex combinatorial reasoning was required to

*This research was supported by the Canadian Government through their NSERC program.

[1] The function $\mathit{combo}()$ acts as a run-time variable in the sense of (Etzioni, Golden, and Weld 1997).

discover these plans. However, some of these plans are quite natural. In this paper we present a collection of techniques for extending the inferential and representational power of PKS so that it can find more of these types of plans. The features we have added include extensive support for numbers, so that PKS can generate plans that deal with resources; a postdiction procedure so that PKS can extract more knowledge from its plans (Sandewall 1994); and a temporal goal language, so that PKS can plan for "hands-off," "restore," and other temporal goals (Weld and Etzioni 1994). With these techniques, all fully implemented in the new version of PKS, our planner can solve a wider and more interesting range of planning problems. More importantly, these techniques help us understand more fully the potential of the "knowledge-level" approach to planning under incomplete knowledge and sensing.

The rest of the paper is organized as follows. First, we present a short recap of the knowledge-based approach to planning embodied in the PKS system. Then, we discuss the new techniques we have developed to enhance PKS. A series of planning examples are then presented to help demonstrate the effectiveness of the PKS system and these techniques in general.

PKS

PKS (Planning with Knowledge and Sensing) is a knowledge-based planner that is able to construct conditional plans in the presence of incomplete knowledge (Petrick and Bacchus 2002). The PKS 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 and actions are represented as updates to these databases. Thus, actions are modelled as knowledge-level modifications to the agent's knowledge state, rather than as physical-level updates to the world state.

Modelling actions as database updates leads naturally to a simple forward-chaining approach to finding plans that is both efficient and effective (see (Petrick and Bacchus 2002) for empirical evidence). The computational efficiency of this approach is the result of restricting the types of knowledge that can be expressed, and by limiting the power of the inferential mechanism. In particular, only certain fixed types of disjunctive knowledge can be represented and the inferential mechanism is incomplete.

PKS uses four databases to represent an agent's knowledge, the semantics of which is provided by a translation to formulas of a first-order modal logic of knowledge (Bacchus and Petrick 1998). Thus, any configuration of the databases corresponds to a collection of logical formulas that precisely characterizes the agent's knowledge state. The four databases used are as follows.

$\mathbf{K_f}$: The first database is like a standard STRIPS database, except that both positive and negative facts are allowed and the closed world assumption is not applied. K_f can include any ground literal, ℓ; $\ell \in K_f$ means that we know ℓ. K_f can also contain knowledge of function values.

$\mathbf{K_w}$: The second database is designed to address plan-time reasoning about sensing actions. If the plan contains an action to sense a 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 f remains unknown. Hence, $\phi \in K_w$ means that the agent either knows ϕ or knows $\neg\phi$, and that at execution time this disjunction will be resolved.

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

$\mathbf{K_v}$: The third database stores information about function values that 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 the plan time modelling of sensing actions that return numeric values. For example, $\mathit{size}(\mathit{paper.tex}) \in K_v$ means the agent knows at plan time that the size of *paper.tex* will become known at execution time.

$\mathbf{K_x}$: The fourth database contains 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 ℓ_i is a ground literal. Such a formula represents knowledge of the fact that "exactly one of the ℓ_i is true." Hence, if one of these literals becomes known, we immediately come to know that the other literals are false. Similarly, if $n-1$ of the literals become false we can conclude that the remaining literal is true. This form of incomplete knowledge is common in many planning scenarios.

Actions in PKS are represented as updates to the databases (i.e., updates to the agent's knowledge state). Applying an action's effects simply involves adding or deleting the appropriate formulas from the collection of databases. An inference algorithm examines the database contents and draws conclusions about what the agent does and does not know or "know whether" (Bacchus and Petrick 1998). The inference algorithm is efficient, but incomplete, and is used to determine if an action's preconditions hold, what conditional effects of an action should be activated, and whether or not a plan achieves the stated goal.

For instance, consider a scenario where we have a bottle of liquid, a healthy lawn, and three actions: *pour-on-lawn*, *drink*, and *sense-lawn*,[2] specified in Table 1. Intuitively, *pour-on-lawn* pours some of the liquid on the lawn with the effect that if the liquid is poisonous, the lawn becomes dead. In our action specification, *pour-on-lawn* has two conditional effects: if it is not known that $\neg$*poisonous* holds, then $\neg$*lawn-dead* will be removed from K_f (i.e., the agent will no longer know that the lawn is not dead); if it is known that *poisonous* holds, then *lawn-dead* will be added to K_f (i.e., the agent will come to know that the lawn is dead). Drinking the liquid (*drink*) affects the agent's knowledge of

[2] This example was communicated to us by David Smith.

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		$\neg K(\neg poisonous2) \Rightarrow$ $del(K_f, \neg lawn\text{-}dead)$ $K(poisonous2) \Rightarrow$ $add(K_f, lawn\text{-}dead)$

Table 1: Actions in the poisonous liquid domain

being poisoned in a similar way. Finally, *sense-lawn* senses whether or not the lawn is dead and is represented as the update of adding *lawn-dead* to K_w.

If K_f initially 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 an action's effects provides a simple means of evolving a knowledge state, and it is the approach that PKS uses to search over the space of conditional plans.

In PKS, a conditional plan is a tree whose nodes are labelled by a knowledge state (a set of databases), and whose edges are labelled by an action or 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 n can also have two children, in which case each edge is labelled by a fluent F, such that $K_w(F)$ is entailed by n'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 n's K_f database, and the label for the other child by adding $\neg F$ to n's K_f database.

An existing plan may be extended by adding a new action (i.e., a new child) or a new branch (i.e., a new pair of children) to a leaf node. The inference algorithm computes whether or not an extension can be applied, generates the effects of the action or branch, and tests if the new nodes satisfy the goal; no leaf is extended it if already achieves the goal. The search terminates when a conditional plan is found in which all the leaf nodes achieve the goal. Currently, PKS performs only undirected search (i.e., no search control), but it is still able to solve a wide range of interesting problems (Petrick and Bacchus 2002).

Extensions to the PKS framework

Postdiction: Although PKS's forward-chaining approach is able to efficiently generate plans, there are situations where the resulting knowledge states fail to contain some "intuitive" conclusions. For instance, say we execute the sequence of actions $\langle$*pour-on-lawn*; *sense-lawn*$\rangle$ (Table 1) in an initial state where the lawn is alive, and then come to know that the lawn is dead. An obvious additional conclusion is that the liquid is poisonous. Similarly, if the lawn remained alive, we can conclude that the liquid is not poisonous. By reasoning about these two possible outcomes at plan time, prior to executing the plan, we should be able to conclude that the plan not only achieves K_w knowledge of *lawn-dead*, but that it also achieves K_w knowledge of *poisonous*.

It should be noted that a conclusion such as *poisonous* requires a non-trivial inference. Inspecting the states that result from the action sequence $\langle$*pour-on-lawn*; *sense-lawn*$\rangle$ reveals that neither *poisonous* nor $\neg$*poisonous* follows from the individual actions executed: *pour-on-lawn* 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 $\neg$*lawn-dead* holds. It is not hard to see that without this knowledge the conclusion *poisonous* is not justified.

We can capture these kinds of additional inferences at plan time by examining action effects and non-effects. Consider the two possible outcomes of *sense-lawn* in the action sequence $\langle$*pour-on-lawn*; *sense-lawn*$\rangle$: either it senses *lawn-dead* or it senses $\neg$*lawn-dead*. If we treat each outcome separately we can consider two sequences of actions, one for each outcome of *lawn-dead*. Each action sequence produces three world states: W_0 the initial world, W_1 the world after executing *pour-on-lawn*, and W_2 the world after executing *sense-lawn*.

In the first sequence, we know that $\neg$*lawn-dead* holds 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 since $\neg$*lawn-dead* held in W_0 and *lawn-dead* held in W_1, *pour-on-lawn* must have produced a change in *lawn-dead*. Since *lawn-dead* is only altered by a conditional effect of *pour-on-lawn*, it must be the case that the antecedent of the condition, *poisonous*, was true in W_0 when *pour-on-lawn* was executed. Furthermore, *poisonous* is not affected by *pour-on-lawn*, nor by *sense-lawn*. Hence, *poisonous* must be true in W_1 as well as in W_2.

Similarly, in the second sequence $\neg$*lawn-dead* holds in both W_0 and W_2. At the critical step we conclude that since $\neg$*lawn-dead* holds 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 have been false in W_0. That is, $\neg$*poisonous* must have been true in W_0. Since $\neg$*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 *poisonous* or knows $\neg$*poisonous*, and so, we can conclude that the plan allows us to know-whether *poisonous*. It should also be noted that all of the inferences described above provide updates to the agent's *current* knowledge state, that is, the agent's knowledge state arising from applying a given action sequence. For instance, the addition of *poisonous* or $\neg$*poisonous* to W_0 can only be made after the sequence of actions $\langle$*pour-on-lawn*; *sense-lawn*$\rangle$ is performed. Thus, we make a distinction between the agent's knowledge of W_0 after applying $\langle$*pour-on-lawn*; *sense-lawn*$\rangle$ (i.e., where the agent knows whether *poisonous* is true or not), compared with the agent's knowledge of the initial W_0 that we began with (i.e., where no actions have been performed and the agent did not know whether *poisonous* is true or not). In other words, the agent's knowledge maintains the most recent information about each state in the plan.

The inferences employed above are examples of *postdiction* (Sandewall 1994). Although the individual inferences are fairly simple, taken together they can add significantly to the agent's ability to deal with incompletely known environments. A critical element in these inferences is the Markov assumption as described in (Golden and Weld 1996): first, we must assume that we have complete knowledge of action effects and non-effects (our incomplete knowledge comes from a lack of information about precisely what state we are in when we apply the action); second, we must assume that the agent's actions are the only source of change in the world. The first assumption is not restrictive, since action non-determinism can always be converted into non-determinism about the state in which it is being executed (Bacchus, Halpern, and Levesque 1999). The second assumption is more restrictive, however, and needs to be examined carefully when dealing with other agents (or nature) that could be altering the world concurrently.

In PKS, reasoning about these kinds of inferences is implemented by manipulating *linearizations* of the tree-structured conditional 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 equal to 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 resolves itself during execution and in the manner in which that resolution affects the actions the agent subsequently executes. For each linear sequence, we apply a set of backward and forward inferences to draw additional conclusions along that sequence. When new branches are added to the conditional plan, new linearizations are incrementally constructed and the process is repeated.

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

1) If a cannot make ϕ false (e.g., ϕ is unrelated to any of the facts a makes true), then if ϕ becomes newly known in W make ϕ known in W^+. Similarly, if a cannot make ϕ true, then if ϕ becomes newly known in W^+ make ϕ known in W. In both cases a cannot have changed the status of ϕ between the two worlds W and W^+.

2) If ϕ becomes newly known in W and a has the conditional effect $\phi \rightarrow \psi$, make ψ known in W^+. ψ must be true in W^+ as either it was already true or a made it true.

3) If a has the conditional effect $\phi \rightarrow \psi$ and it becomes newly known that ψ holds in W^+ and $\neg\psi$ holds in W, make ϕ 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 $\phi \rightarrow \psi$ and it becomes newly known that $\neg\psi$ holds in W^+, make $\neg\phi$ 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 restricting disjunctions and, hence, we cannot use these rules to infer arbitrary new disjunctions.

To avoid this problem, we restrict ϕ and ψ to be literals, and further require that our actions cannot add or delete a fluent F with more than one conditional effect. For example, an action cannot contain the two conditional effects $a \rightarrow F$ and $b \rightarrow F$.[3] These restrictions do not prohibit ϕ and ψ from being parameterized, provided such parameters are among the parameters of the action. In certain cases, we can apply these rules to more complex formulas, without producing general disjunctions; implementing these extensions remains as future work.

A conditional plan is updated by applying these inference rules to each linearization of the 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. A successful application of one of the inference rules might allow other rules to fire. Nevertheless, even in the worst case we can still run the rules to closure (i.e., to a state where no rule can be applied) fairly efficiently.

Proposition 1 *On a conditional plan with n leaves and maximum height d, at most $O(nd^2)$ tests of rule applicability need be performed to reach closure.*

In practice, we have found that the number of effects applied at each state is often quite small.

We illustrate the operation of our postdiction algorithm on 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 Figure 1, along with the contents of the databases. The two linearizations of the plan are shown in (a) and (b). Applying the new inference rules produces the additional conclusions shown in bold; the number following the conclusion indicates the rule that was applied in each case. The net result is that we have proved that in every outcome of the plan the agent either knows *poisonous*

[3] If this was allowed, rule 3 above would be invalid. The correct inference from knowing $\neg F$ in W and F in W^+ would be $a \vee b$, which is a disjunction. This observation was pointed out to us by Tal Shaked.

or knows $\neg$*poisonous*, i.e., the plan achieves know-whether knowledge of *poisonous.*

Situation calculus encoding: We have also shown our new inference rules to be sound, with respect to an encoding in the language of the situation calculus. The situation calculus ((McCarthy 1963), and as presented in (Reiter 2001)) is a first-order language, with some second-order features, specifically designed to model dynamically changing worlds. A first-order term called a *situation* is used to represent a sequence of actions, also known as a *possible world history*. Predicates and functions are extended so that their values may be referenced with respect to a particular situation; the values of such *fluents* are permitted to change from situation to situation. Actions provide the means of change in a domain, and when applied to a given situation, generate a successor situation. A knowledge operator added to the basic language (Scherl and Levesque 2003) allows a formal distinction to be made between fluents that are true of a situation and fluents that are *known* to be true of a situation. Since a situation term denotes a history of action, it provides a temporal component that indexes all assertions made about the agent's knowledge.

Any planning problem represented by PKS can be recast formally in the language of the situation calculus. By doing so, we have been able to verify that the conclusions made by our inference rules are always entailed by an encoded theory, thus establishing the soundness of our inference rules. Furthermore, we have confirmed the soundness for a generalized version of our rules, rather than the restricted set of rules we implemented in PKS. For instance, since the situation calculus is a general first-order language, the inferences that generated disjunctions in rule 3 (and could not be represented in PKS) can be shown to be correct.

Our situation calculus encoding also allows us to correctly make the necessary formal distinction between what an agent knows about the previous state when it is in its current state, and what the agent knew about its previous state when it was in its previous state. In particular, in the situation calculus **Knows**$(F(\mathit{prev}(s)), s)$ and **Knows**$(F(\mathit{prev}(s)), \mathit{prev}(s))$ are distinct pieces of knowledge. **Knows**$(F(\mathit{prev}(s)), s)$ indicates that in situation s (now) it is known that F was true in the previous situation ($\mathit{prev}(s)$). **Knows**$(F(\mathit{prev}(s)), \mathit{prev}(s))$ on the other hand indicates that in the previous situation F is known to be true in that situation.

In conditional plans, when we reach a particular state as a result of having our sensing turn out a particular way, it is possible for us to know more now about the previous state than we did when we were in the previous state. Our inference rules are used to update our knowledge of the previous states so that knowledge is described with respect to the states labelling the leaf nodes in the conditional plan. Thus, the knowledge states represented in the conditional plans are always referenced with respect to s rather than with respect to $\mathit{prev}(s)$. Since the previous actions and outcomes leading up to a leaf node are fixed, we can never lose knowledge about the past. That is, we can never have the case that **Knows**$(F(\mathit{prev}(s)), \mathit{prev}(s))$ holds when we don't have that **Knows**$(F(\mathit{prev}(s)), s)$ also holds. Instead, the agent's knowledge about its past states is always non-decreasing, and the agent cannot lose knowledge about the past when the new inference rules are applied.

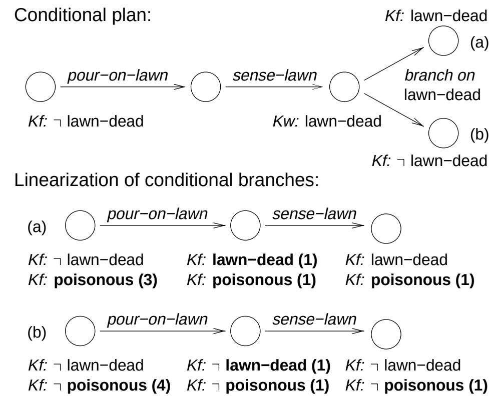

Figure 1: Postdiction in the poisonous liquid domain

Temporally extended goals: Consider again the plan illustrated in Figure 1. If we apply the postdiction algorithm as in branch (a), we can not only infer that *poisonous* held in the final state of the execution, but also that it held in the *initial state*. As we discussed above, postdiction can potentially update the agent's knowledge of any of the states visited by the plan. In other words, along this execution branch we can conclude that the liquid must have *initially* been poisonous. Similarly, along branch (b), we conclude that $\neg$*poisonous* held in the initial state. Thus, at plan time we could also infer that the conditional plan achieves know-whether knowledge of whether *poisonous* initially held.

Often, these kinds of temporally-indexed conclusions are needed to achieve certain goals. For instance, restore goals require that the final state returns a condition to the status it had in the initial state (Weld and Etzioni 1994). We might not know the initial status of a condition and, hence, it may be difficult for the planner to infer that a plan does in fact restore this status. However, with additional reasoning (as in the above example), we may be able to infer the initial status of the condition, and thus be in a position to ensure a plan properly restores it.

Since our postdiction algorithm requires the ability to inspect and augment any knowledge state in a conditional plan's tree structure, the infrastructure is already in place to let us solve more complex types of temporal goals that reference states other than the final state.

In PKS, goals are constructed from a set of primitive queries (Bacchus and Petrick 1998) that can be evaluated by the inference algorithm at a given knowledge state. A primitive query Q is specified as having one of the following forms: (i) $K(\ell)$: is a ground literal ℓ known to be true? (ii) $\mathit{Kval}(t)$: is term t's value known? (iii) $\mathit{Kwhe}(\ell)$: do we "know whether" a literal ℓ? Our enhancements to the goal language additionally allow a query Q to specify one of the following temporal conditions:

1. Q^N: the query must hold in the final state of the plan,

2. Q^0: the query must hold in the initial state of the plan, or
3. Q^*: the query must hold in every state that could be visited by the plan.

Conditions of type (1) can be used to express classical goals of achievement. Type (2) conditions allow, for instance, restore goals to be expressed. Conditions of type (3) can be used to express "hands-off" or safety goals (Weld and Etzioni 1994).

Finally, we can combine queries into arbitrary goal formulas that include disjunction,[4] conjunction, negation, and a limited form of existential and universal[5] quantification. When combined with the postdiction algorithm of the previous section, a goal is satisfied in a conditional plan provided it is satisfied in every linearization of the plan.

For instance, the plan in Figure 1 satisfies the goal $\mathit{Kwhe}^0(\mathit{poisonous}) \wedge \mathit{Kwhe}^N(\mathit{poisonous})$, i.e., we know whether *poisonous* is true or not in both the initial state and the final state of each linearization of the plan. The same plan also satisfies the stronger, disjunctive goal $(K^0(\mathit{poisonous}) \wedge K^N(\mathit{poisonous})) \vee (K^0(\neg\mathit{poisonous}) \wedge K^N(\neg\mathit{poisonous}))$. In this case, linearization (a) satisfies $K^0(\mathit{poisonous}) \wedge K^N(\mathit{poisonous})$, linearization (b) satisfies $K^0(\neg\mathit{poisonous}) \wedge K^N(\neg\mathit{poisonous})$, and so the conditional plan satisfies the disjunctive goal. Finally, the plan also satisfies the goal $\mathit{Kwhe}^*(\mathit{poisonous})$ since we know whether *poisonous* is true or not at every knowledge state of the plan.

Numerical evaluation: Many planning scenarios require the ability to reason about numbers. For instance, constructing plans to manage limited resources or satisfy certain numeric constraints requires the ability to reason about arithmetic expressions. To increase our flexibility to generate plans in such situations, we have introduced numeric expressions into PKS. Currently, PKS can only deal with numeric expressions containing terms that can be evaluated down to a number at plan time; expressions that can only be evaluated at execution time are not permitted. For example, a plan might involve filling the fuel tank of a truck t_1. If the numeric value of the amount of fuel subsequently in the tank, $\mathit{fuel}(t_1)$, is known at plan time, PKS can use $\mathit{fuel}(t_1)$ in further numeric expressions. However, if the amount of fuel added is known only at run time, so that PKS only K_v's $\mathit{fuel}(t_1)$ but does not know how to evaluate it at plan time, then it cannot use $\mathit{fuel}(t_1)$ in other numeric expressions.

Even though PKS can only deal with numeric expressions containing known terms, these expressions can be very complex: they are a subset of the set of expressions of the C language. Specifically, numeric expressions can contain all of the standard arithmetic operations, logical connective operators, and limited control structures (e.g., conditional evaluations and simple iterative loops). Temporary variables may also be introduced into calculations of an expression.

[4] As in the approach of (Petrick and Levesque 2002), disjunctions *outside* the scope of knowledge operators are not problematic.

[5] Queries containing universal quantifiers are currently restricted to quantify over the set of *known* objects, rather than the set of all objects in the domain.

Numeric expressions can also be used in queries, and we can update our databases with a formula containing numeric terms. For instance, a numeric term resulting from a queried expression can be utilized in additional calculations and then added to a database as an argument of a new fact. The only restriction, as noted above, is that PKS must be able to evaluate these expressions down to a specific numeric value before they are used to query or update the databases. Nevertheless, as we will demonstrate in the planning problems presented below, even with this restriction, the numeric expressions we allow are still very useful in modelling a variety of planning problems.

Exclusive-or knowledge of function values: PKS has a K_x database for expressing "exclusive-or" knowledge. A particularly useful case of exclusive-or knowledge arises when a function has a finite and known range. For example, the function $f(x)$ might only be able to take on one of the values *hi*, *med*, or *lo*. In this case, we know that for every value of x we have $(f(x) = \mathit{hi} | f(x) = \mathit{med} | f(x) = \mathit{lo})$. Previously, PKS could not represent such a formula in its K_x database, as the formula contains literals that are not ground. Because finite valued functions are so common in planning domains, we have extended PKS's ability to represent and reason with this kind of knowledge.

We can take advantage of this additional knowledge in two ways. First, we can utilize this information to reason about sets of function values and their inter-relationship. For example, say that $g(x)$ has range $\{d_1|d_2|\ldots|d_m\}$ while $f(x)$ has range $\{d_1|a_1|\ldots|a_m\}$. Then, from $f(c) = g(b)$ we can conclude that $f(c) = g(b) = d_1$.[6] Second, we have added the ability to insert multi-way branches into a plan when we have K_v knowledge of a finite-range function. In such situations, the planner will try to construct a plan branch for each of the possible mappings of the function.

For instance, in the open safe example it might be the case that we know the set of possible combinations. Such knowledge could be specified by including the formula $\mathit{combo}() = (c_1|c_2|\ldots|c_n)$ in our extended K_x database. In any plan state where $\mathit{combo}() \in K_v$ (we know the value of the combination) we could immediately complete the plan with an n-way branch on the possible values of *combo*, followed by the action $\mathit{dial}(c_i)$ to achieve an open safe along the i-th branch.

Planning problems

We now illustrate the extensions made to PKS with a series of planning problems. Our enhancements have allowed us to experiment with a wide range of problems PKS was previously unable to solve. We also note again that even though our planner employs blind search to find plans it is still able to solve many of the examples given below in times that are less than the resolution of our timers (1 or 2 milliseconds).

Poisonous liquid: When given the actions specified in Table 1, PKS can immediately find the plan $\langle$*pour-on-lawn*, *sense-lawn*$\rangle$ to achieve the goal $\mathit{Kwhe}^0(\mathit{poisonous}) \wedge$

[6] Constants that are syntactically distinct denote different objects in the domain.

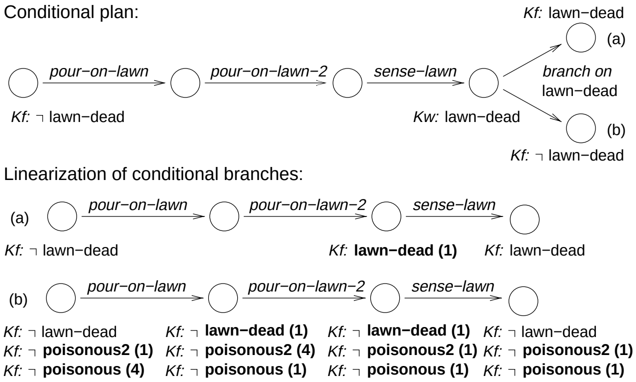

Figure 2: Poisonous liquid domain with two liquids

$\mathit{Kwhe}^N(\mathit{poisonous})$ (knowing whether *poisonous* held in both the initial and final states). It is also able to find the same plan when given the goal $(K^0(\mathit{poisonous}) \wedge K^N(\mathit{poisonous})) \vee (K^0(\neg\mathit{poisonous}) \wedge K^N(\neg\mathit{poisonous}))$, as well as the goal $\mathit{Kwhe}^*(\mathit{poisonous})$.

An interesting variation of the poisonous liquid domain includes the addition of the action *pour-on-lawn-2* (see Table 1). *pour-on-lawn-2* has the effect of pouring a second unknown liquid onto the lawn; its effects are similar to those of *pour-on-lawn*: the second liquid may be poisonous (represented by *poisonous2*) and, thus, kill the lawn. When presented with the conditional plan shown at the top of Figure 2, PKS is able to construct the linearizations (a) and (b), and augment the databases with the conclusions shown in bold in the figure. This plan is useful for illustrating our postdiction rules. In (a), 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. This reasoning is intuitively sensible: the agent is unable to determine which liquid killed the lawn and, thus, cannot conclude which of the liquids is poisonous. (The disjunctive conclusion that one of the liquids is poisonous cannot be represented by PKS). In (b), after applying the inference rules we are able to establish that $\neg$*lawn-dead*, $\neg$*poisonous*, and $\neg$*poisonous2* must hold in each state of the plan. Again these conclusions are intuitive: after sensing the lawn and determining that it is not dead, the agent can conclude that neither liquid is poisonous.

It should be noted that planners that represent sets of possible worlds (and thus deal with disjunction) are also able to obtain the conclusions obtained from our postdiction algorithm in the above examples (and some further disjunctions as well). In particular, these examples are all propositional, and do not utilize PKS's ability to deal with non-propositional features. What does pose a problem for many of these planners, however, is their inability to infer the temporally-indexed conclusions necessary to verify the temporal goal conditions; planners that only maintain and test the final states of a plan will be unable to establish the required conclusions.

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

Table 2: Painted door action specification

Action	Pre	Effects
$\mathit{dial}(x)$		$\mathit{add}(K_w, \mathit{open})$ $\mathit{del}(K_f, \neg\mathit{open})$ $K(\mathit{combo}() = x) \Rightarrow$ $\mathit{add}(K_f, \mathit{open})$

Table 3: Open safe action specification

Painted door: In the painted door domain we have the two actions given in Table 2. $\mathit{paint}(x)$ changes the colour of the door $\mathit{door\text{-}colour}()$ to an available colour x, while *sense-colour* senses the value of $\mathit{door\text{-}colour}()$. 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. This can be expressed by the temporally-extended formula $(\exists x)K^*(\mathit{door\text{-}colour}() = x)$.

Initially, the agent knows that the door is one of two possible colours, c_1 or c_2, represented by the formula $\mathit{door\text{-}colour}() = \{c_1|c_2\}$ in the K_x database.[7] During its search, PKS finds the single-step plan $\langle\mathit{sense\text{-}colour}\rangle$. This action has the effect of adding $\mathit{door\text{-}colour}()$ to the K_v database, indicating that the value of $\mathit{door\text{-}colour}()$ is known. Using this information, combined with its K_x knowledge of the possible values for $\mathit{door\text{-}colour}()$, PKS can construct a two-way branch that allows it to consider the possible mappings of $\mathit{door\text{-}colour}()$. Along one branch the planner asserts that $\mathit{door\text{-}colour}() = c_1$; along the other branch it asserts that $\mathit{door\text{-}colour}() = c_2$. Since *sense-colour* does not change $\mathit{door\text{-}colour}()$, after applying the postdiction algorithm we are able to conclude along each linearization that the value of $\mathit{door\text{-}colour}()$ is the same in every state. In each linearization $\mathit{door\text{-}colour}()$ has a different value in the initial state, but its value agrees with its value in the final state. Thus, we can conclude that the plan achieves the goal.

Note that if PKS examines a plan like $\langle\mathit{paint}(c_1)\rangle$ it will not know the value of $\mathit{door\text{-}colour}()$ in the initial state. Since *paint* changes the value of $\mathit{door\text{-}colour}()$, our postdiction algorithm will not allow facts about $\mathit{door\text{-}colour}()$ to be passed back through the *paint* action. Thus, PKS cannot conclude that $\mathit{door\text{-}colour}()$ remains the same throughout the plan, and so plans involving *paint* are rejected as not achieving the goal.

Opening a safe: Another interesting problem for PKS is the open safe problem. In this problem we consider a safe with a fixed number of known, possible combinations. In the initial state we know that the safe is closed and that one of the combinations will open the safe (represented as K_x knowledge). The actual combination is denoted by the 0-ary

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

Action	Pre	Effects
$dial(x)$		$add(K_w, open)$ $del(K_f, \neg open)$ $add(K_f, justDialled() = x)$ $K(combo() = x) \Rightarrow$ $add(K_f, open)$
Domain specific update rules		
$K(open) \wedge K(justDialled() = x) \Rightarrow$ $add(K_f, combo() = x)$ $K(\neg open) \wedge K(justDialled() = x) \Rightarrow$ $add(K_f, combo() \neq x)$		

Table 4: Open safe domain (Petrick and Bacchus 2002)

function, *combo*(). There is one action, $dial(x)$, which dials a combination x on the safe. *dial* has the effect of sensing whether or not the safe becomes open. If the combination dialled is known to be the combination of the safe, then it will become known that the safe is open. The action specification for *dial* is given in Table 3.

After a $dial(c_i)$ action is performed (where c_i is some combination), the planner is able to form a conditional branch on knowing-whether *open*. Along one branch the planner asserts that *open* is known to be true. Since *dial* changes the value of *open*, postdiction can conclude that *dial*'s conditional effect must have been applied, and so, $combo() = c_i$ must hold in the initial state. Since *combo* is not changed by *dial*, then $combo() = c_i$ must also be true in the state following the *dial* action. In this state, the goal is satisfied: the safe is known to be open and c_i is known to be the combination of the safe. Along the other branch the planner asserts that $\neg$*open* holds. Since $\neg$*open* is unchanged by *dial*, it must be the case that *dial*'s conditional effect was not applied, and so $combo() \neq c_i$ must hold in the initial state. Again, since *combo* is unchanged by *dial*, it must be the case that $combo() \neq c_i$ also holds in the state following *dial*. The planner's K_x knowledge of possible combinations is then updated, and the planner can try another *dial* action with a different combination.

An earlier PKS encoding of the open safe problem appeared in (Petrick and Bacchus 2002), and is shown in Table 4. The new inference rules, however, allow us to simplify this action specification. In our previous encoding, we required a special function, *justDialled*(), to explicitly track each combination c_i that was dialled. A pair of domain specific update rules[8] was then used to assert whether $combo() = c_i$ was true or not, depending on whether or not the safe was known to be open. Since postdiction automatically generates such conclusions from our action specification, we no longer require the extra function and update rules in our encoding.

UNIX domain: Our final examples are taken from the UNIX domain. The actions for the first example are given in Table 5. A directory hierarchy is defined by the relation $indir(x, y)$ (x is in directory y), the current working directory is specified by the 0-ary function *pwd*(), and there are two actions for moving around in the directory tree: $cd(x)$ moves down to a sub-directory of *pwd*() and $cd\text{-}up(x)$ moves to the parent directory of *pwd*(). Finally, the third action, *ls*, can sense the presence of a file in *pwd*().

[8] Update rules are simply a convenient way of specifying additional action effects that might apply to many different actions (Petrick and Bacchus 2002). Update rules are checked and conditionally fired after an action is applied or a plan branch is added.

Action	Pre	Effects
$cd(d)$	$K(dir(d))$ $K(indir(d, pwd()))$	$add(K_f, pwd() = d)$
$cd\text{-}up(d)$	$K(dir(d))$ $K(indir(pwd(), d))$	$add(K_f, pwd() = d)$
$ls(f, d)$	$K(pwd() = d)$ $K(file(f))$ $\neg K_w(indir(f, d))$	$add(K_w, indir(f, d))$
Domain specific update rules		
$\neg K(processed(f, d)) \wedge K(indir(f, d)) \wedge Kval(size(f, d)) \Rightarrow$ $t = [(size\text{-}max() > size(f, d))?\ size\text{-}max() : size(f, d)]$, $add(K_f, size\text{-}max() = t)$, $add(K_f, count() = count() + 1)$, $add(K_f, processed(f, d))$ $\neg K(processed(f, d)) \wedge K(indir(f, d)) \wedge \neg Kval(size(f, d)) \Rightarrow$ $add(K_f, size\text{-}unk() = size\text{-}unk() + 1)$, $add(K_f, processed(f, d))$ $\neg K(processed(f, d)) \wedge K(\neg indir(f, d)) \Rightarrow$ $add(K_f, processed(f, d))$		

Table 5: UNIX domain action specification (1)

Initially, the planner has knowledge of the current directory, $pwd() = root$, and of the directory tree's structure: $indir(icaps, root)$, $indir(kr, root)$, and $indir(planning, icaps)$. The planner also has the initial knowledge $file(paper.tex)$ (a precondition of *ls*).

In this example we consider the situation where multiple copies of the file *paper.tex* may exist, located in different directories with possibly different sizes. We would like to determine the number of instances of *paper.tex* that are in the directory tree, as well as the size of the largest copy (whose size is known). To do this, we introduce some additional functions. $size(f, d)$ specifies the size of file f in directory d, *size-max*() keeps track of the largest file size that has been found, *count*() simply counts the number of instances of *paper.tex* whose size is known, while *size-unk*() counts the number of copies whose size is not known. Initially, we have that *count*(), *size-max*(), and *size-unk*() are all known to be equal to zero.

Our domain encoding includes three domain specific update rules, rules that are conditionally fired in the initial state, or after actions or branches have been added to the plan. These rules handle the different cases when we have not yet "processed" a directory d, i.e., checked it for the presence of *paper.tex*. The first rule fires when *paper.tex* is in a directory d and its size is known. In this case, we can compare the size, $size(paper.tex, d)$ against the current maximum size, *size-max*(), and update *size-max*() if necessary. *count*() is also incremented. The second rule fires when *paper.tex* is in a directory d but we don't know its size. In this case we simple increment *size-unk*(). Finally, the third rule fires when *paper.tex* is not in a directory d. In this case, none of the functions are changed. After any of the update rules is fired, we mark directory d as being checked for *paper.tex* (i.e., *processed*$(\textit{paper.tex}, d)$ becomes known).

Our goal is to know that we have *processed* each directory in the directory tree. In the first example, we consider the case when we know the location and size of some copies of *paper.tex*: $\textit{indir}(\textit{paper.tex}, \textit{kr})$, $\textit{indir}(\textit{paper.tex}, \textit{icaps})$, $\textit{size}(\textit{paper.tex}, \textit{kr}) = 1024$, and $\textit{size}(\textit{paper.tex}, \textit{icaps}) = 4096$. Running PKS on this problem immediately produces the plan: $\langle \textit{ls}(\textit{paper.tex}, \textit{root}); \textit{cd}(\textit{icaps}); \textit{cd}(\textit{planning}); \textit{ls}(\textit{paper.tex}, \textit{planning}) \rangle$, following by a branch on knowing-whether $\textit{indir}(\textit{paper.tex}, \textit{planning})$: in each branch we branch again on knowing-whether $\textit{indir}(\textit{paper.tex}, \textit{root})$. The final plan has four leaf nodes. In each of these terminal states, $\textit{size-max}() = 4096$ and $\textit{count}() = 2$. The four branches of the plan track the planner's incomplete knowledge of *paper.tex* being in the directories *root* and *planning*: each final state represents one possible combination of knowing-whether $\textit{indir}(\textit{paper.tex}, \textit{root})$ and knowing-whether $\textit{indir}(\textit{paper.tex}, \textit{planning})$. Moreover, the value of the function $\textit{size-unk}()$ is appropriately updated in each of these states (by the second update rule). For instance, along the branch where $\textit{indir}(\textit{paper.tex}, \textit{root})$ and $\textit{indir}(\textit{paper.tex}, \textit{planning})$ holds, we would also know $\textit{size-unk}() = 2$. When $\neg\textit{indir}(\textit{paper.tex}, \textit{planning})$ and $\neg\textit{indir}(\textit{paper.tex}, \textit{root})$ is known, $\textit{size-unk}() = 0$. The remaining two branches would each have $\textit{size-unk}() = 1$.

PKS is also able to generate a plan if we don't have *any* information about the sizes or locations of *paper.tex*. In this case, the plan performs an *ls* action in each directory and produces a plan branch for each possibility of knowing-whether *paper.tex* is in that directory. With 4 directories to check, PKS produces a plan with $2^4 = 16$ branches. Our blind depth-first version of the planner is able to find this plan in 0.01 seconds; our breath-first version of PKS which ensures the smallest plan is generated, is able to do so in 30.1 seconds.

One final extension to this example is the addition of a goal that requires us to not only determine the size of the largest instance of *paper.tex*, but also to move to the directory containing this file (provided we have found a file whose size is known). To do this, we need simply add the additional "guarded" goal formula $(\exists d).K^N(\textit{count}() > 0) \Rightarrow K^N(\textit{pwd}() = d) \wedge K^N(\textit{size}(\textit{paper.tex}, d) = \textit{size-max}())$ to our goal list. If PKS has processed a file whose size is known (i.e., $\textit{count}() > 0$) then it also needs to ensure $\textit{pwd}()$ matches the directory containing a file size of $\textit{size-max}()$ for *paper.tex*. Otherwise, the goal is trivially satisfied.

Our second UNIX domain example uses the actions given in Table 6. Initially, we know about the existence of certain files and directories, specified by the $\textit{file}(f)$ and $\textit{dir}(d)$ predicates; some of their locations, specified by the $\textit{indir}(f, d)$ predicate; and that some directories are executable, specified by the $\textit{exec}(d)$ predicate. The action $\textit{ls}(d)$ senses the executability of a directory d; $\textit{chmod+x}(d)$ and $\textit{chmod-x}(d)$ respectively set and delete the executability of a directory; and $\textit{cp}(f, d)$ 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.

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

Table 6: UNIX domain action specification (2)

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

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

Since the executability of *icaps* is not known initially, the *ls* action is necessary to sense the value of $\textit{exec}(\textit{icaps})$. Postdiction establishes 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 $\textit{exec}(\textit{icaps})$ against its value in the final state(s) of the plan. By reasoning about the possible values of $\textit{exec}(\textit{icaps})$, 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, $\textit{cp}^+$ (also given in Table 6). Unlike *cp*, $\textit{cp}^+$ 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: $\textit{cp}^+(\textit{paper.tex}, \textit{icaps})$; branch on $\textit{indir}(\textit{paper.tex}, \textit{icaps})$: if $K(\textit{indir}(\textit{paper.tex}, \textit{icaps}))$ do nothing, otherwise $\textit{chmod+x}(\textit{icaps})$; $\textit{cp}(\textit{paper.tex}, \textit{icaps})$; $\textit{chmod-x}(\textit{icaps})$. In other words, PKS is able to reason from $\textit{cp}^+$ failing to achieve $\textit{indir}(\textit{paper.tex}, \textit{icaps})$ that *icaps* was not initially executable.

Conclusions

Our extensions to the PKS planner have served to increase both its representational and inferential power, enhancing our ability to plan in a variety of new situations. These extensions have also served to demonstrate the utility of the knowledge-based approach to planning under incomplete knowledge. We are currently working on several further extensions; we briefly mention two of these extensions here. The first enhancement is an improvement in our ability to deal with unknown numeric quantities (e.g., we were unable to evaluate expressions that could not be reduced to a number at plan time). Provided we have K_v knowledge of all the numeric terms involved in an expression, we should be able to track these (unevaluated) expressions and under certain conditions introduce plan branches that allow us to reason further using these expressions (e.g., when reasoning about the truth of an inequality such as $f(a) < c$ we could introduce conditional branches and reason about the cases when $f(a) < c$ and $f(a) \geq c$). The second enhancement is a natural generalization to the type of reasoning performed in the painted door domain. Instead of constructing a plan with a multi-way branch for each possible colour of the door, the planner could add a new assertion *door-colour*$() = c$ to K_f, where c is a new constant (essentially a Skolem constant). Our postdiction rules would then conclude that *door-colour*$()$ would be preserved no matter what colour c represented, thus ensuring that the goal is achieved even if the range of door colours is unknown or infinite. We are making progress solving these problems, and believe that the knowledge-based approach continues to have great potential for building powerful planners that can work under incomplete knowledge.

References

Anderson, C. R.; Weld, D. S.; and Smith, D. E. 1998. Extending Graphplan to handle uncertainty & sensing actions. In *Proceedings of the AAAI National Conference*, 897–904.

Bacchus, F., and Petrick, R. 1998. Modeling an agent's incomplete knowledge during planning and execution. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 432–443.

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

Bertoli, P.; Cimatti, A.; Roveri, M.; and Traverso, P. 2001. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 473–478.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Brafman, R., and Hoffmann, J. 2003. Conformant planning via heuristic forward search. In *Workshop on Planning Under Uncertainty and Incomplete Information*.

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

Etzioni, O.; Golden, K.; and Weld, D. 1997. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence* 89(1–2):113–148.

Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 174–185.

Levesque, H. J. 1996. What is planning in the presence of sensing? In *Proceedings of the AAAI National Conference*, 1139–1146. AAAI Press / MIT Press.

McCarthy, J. 1963. Situations, actions, and causal laws. Technical Report Memo 2, Stanford University. Reprinted in Marvin Minsky, ed., 1968, *Semantic Information Processing*. MIT Press.

Moore, R. C. 1985. A formal theory of knowledge and action. In Hobbs, J., and Moore, R. C., eds., *Formal Theories of the Commonsense World*. Norwood, NJ: Ablex Publishing Corp. 319–358.

Petrick, R., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, 212–222.

Petrick, R., and Levesque, H. J. 2002. Knowledge equivalence in combined action theories. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Sandewall, E. 1994. *Features and Fluents*, volume 1. Oxford University Press.

Scherl, R. B., and Levesque, H. J. 2003. Knowledge, Action, and the Frame Problem. *Artificial Intelligence* 144(1–2):1–39.

Weld, D., and Etzioni, O. 1994. The first law of robotics (a call to arms). In *Proc. of the AAAI National Conference*, 1042–1047.