

A 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 this paper we present a new approach to the problem of planning with incomplete information and sensing. Our approach is based on a higher level, “knowledge-based”, representation of the planner’s knowledge and of the domain actions. In particular, in our approach we use a set of formulas from a first-order modal logic of knowledge to represent the planner’s incomplete knowledge state. Actions are then represented as updates to this collection of formulas. Hence, actions are being modelled in terms of how they modify the knowledge state of the planner rather than in terms of how they modify the physical world. We have constructed a planner to utilize this representation and we use it to show that on many common problems this more abstract representation is perfectly adequate for solving the planning problem, and that in fact it scales better and supports features that make it applicable to much richer domains and problems.

Introduction

We present a new approach to the problem of planning with incomplete information and sensing. The intuition behind our approach is that a planning agent operating under conditions of incomplete knowledge (and without a model of uncertainty) can only build plans based on what it knows and on how its knowledge will change as it executes actions—it has access to no other information at plan time. Hence, it should be possible to build plans by considering only the agent’s knowledge state and the way that knowledge state is changed by actions, rather than by having to consider the various ways the physical world can be configured and the ways in which actions change the physical world. In fact, as we will demonstrate, planning at the level of the agent’s knowledge state has significant benefits. In particular, on many problems, planning at the knowledge level scales up much better since it abstracts away from a number of irrelevant distinctions that previous approaches are often forced to make. Furthermore, there are a number of features that we can model at the knowledge level that would be very difficult or even impossible with previous approaches. For example, we are not limited to the propositional case, and

we can deal with functions and run-time variables, features that are essential in a number of domains.

A planner that can operate at the knowledge level requires a tractable representation of its (incomplete) knowledge and a tractable way of updating that representation so as to model the effects actions have on its knowledge state. In previous work (Bacchus & Petrick 1998) we have developed most of such a representation. In this paper we make a key addition to the representation developed in this previous work, an addition that makes it possible to generate plans for a much wider range of problems.

Our representation supports a number of features that are currently unique to our approach, features that we feel are essential for planning in richer domains. Nevertheless, there are some things that our “higher-level” representation cannot handle that previous approaches can. In particular, our representation is not able to handle problems that require complex case analysis to distinguish various possible configurations of the world. However, as we will demonstrate, some of these other approaches are forced to engage in this kind of potentially computationally expensive case analysis even when it is not necessary for generating a plan.

Using our representation we have constructed a simple forward chaining planner that is able to generate interesting plans in a number of different domains. Our current implementation does not as yet have any sophisticated forms of search control. Nevertheless, even using blind search, it is able to demonstrate impressive performance, performance that clearly demonstrates the potential of our approach.

In the paper we first present a short comparison between our approach and the approach most commonly utilized in previous work. Then we explain the representation of incomplete knowledge we have developed, along with the way in which actions, including sensing actions, are represented. How planning problems are specified and a planning algorithm for generating conditional plans are presented next. We close with empirical results on a few different domains and some concluding remarks.

Previous Approaches

A number of works have addressed the problem of planning under incomplete information, e.g., (Pryor & Collins 1996; Bertoli *et al.* 2001; Bonet & Geffner 2000; Anderson, Weld, & Smith 1998). The most recent of these approaches

*This research was supported by the Canadian Government through their NSERC and NCE-IRIS programs.

Copyright © 2002, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

have worked at the propositional level using sets of possible worlds to represent the planner's incomplete knowledge. These sets contain all the possible worlds that are compatible with the planner's knowledge. When an action is to be applied, a check must be performed that its preconditions hold in all of the possible worlds (i.e., that its preconditions are *known*), and then the action is applied to each world in the set to yield a new set of possible worlds. That is, actions are modelled in terms of how they change the world, and the planner's new incomplete knowledge state is computed by considering how that action would behave in each of the different possible worlds characterizing the current knowledge state.

Of course the number of possible worlds needed to represent any knowledge state grows exponentially with the number of propositions. And since each new object in a planning domain generates many new propositions, these sets become very large, very quickly. In (Bertoli *et al.* 2001), which is probably the most efficient of the current planners for incomplete knowledge, this problem is addressed by using BDDs (Bryant 1992) to represent and manipulate these sets compactly. Nevertheless, BDDs are limited to propositional representations and are not always compact.

For example, softbots (Weld & Etzioni 1994) are agents that have to generate and execute plans in software environments like the UNIX operating system or the world wide web. In such domains the set of objects is either unknown (e.g., the files available on the web) or very large, and thus propositional representations are not suitable.

As a result we have taken a completely different approach to the problem. In particular, we use finite sets of formulas of a first-order modal logic of knowledge to represent incomplete knowledge, and we model actions as updates to these sets of formulas (thus actions update the agent's knowledge state rather than the world state). A set of formulas is a much more general and more compact representation of a set of possible worlds than a BDD.¹

The problem, of course, is that it can be computationally intractable to reason with a set of logical formulas. Hence, we are forced to restrict our representation and the kinds of reasoning we can perform with it. In particular, we cannot reason at the level of individual possible worlds with our representation, and thus some planning problems that require this kind of low-level world by world analysis cannot be modelled in our approach whereas they can be handled with BDDs. Nevertheless, we feel that the additional features we gain more than make up for this deficiency.

Representing the Agent's Knowledge

In this section we describe our formalism for representing incomplete knowledge. Our representation is based on extending the STRIPS idea of utilizing a database that can be easily updated by actions. Instead of a single database we utilize a collection of databases, each representing a different kind of knowledge. Actions can then be modelled by

¹Every BDD can be represented with a similarly sized propositional formula but not vice-versa. Furthermore, with first-order formulas the possible worlds can have a relational structure.

specifying the updates they make to these databases.

We want to provide a formal specification of the knowledge state any particular configuration of this collection of databases represents. We accomplish this by providing a fixed translation from the database contents to a set of formulas of a first-order modal logic of knowledge.² Thus the formal specification we want is automatically provided by the formal semantics of this logic. In the sequel we use **DB** to represent the agent's databases, and **KB** to represent the set of logical formulas that formalize the contents of **DB**.

Briefly, a standard modal logic of knowledge adds a modal operator K to an ordinary first-order language, extending the language's syntax by adding the rule: if ϕ is a formula then so is $K(\phi)$. Semantically, the language is interpreted over a collection of possible worlds W , each of which is an ordinary first-order model. Any non-modal formula ϕ is true at a particular world w iff it is true according to the standard rules for interpreting first-order formulas. A formula of the form $K(\phi)$ is true at w iff ϕ is true at every world in W .³

Semantically, the agent's knowledge is being captured by the set W . The agent does not know which world in W is the real world, and considers all of these worlds as possible versions of the way the real world could be configured. If it does not know whether or not ϕ is true, then there will be worlds in W where ϕ is true and worlds where ϕ is false. We assume that although the agent's knowledge is incomplete, it is correct. This assumption is captured by requiring that the real world be a member of W . Hence, knowing ϕ , i.e., that ϕ is true in every world in W , will also imply that ϕ is true in the real world (since the real world is a member of W). For example, $K(\text{readable}(\text{paper.tex}))$ means that the agent knows file *paper.tex* is readable, and that *paper.tex* is in fact readable (it must be true in the real world). However, a formula like $\text{writable}(\text{paper.tex})$ means that *paper.tex* is writable, but this is not necessarily known by the agent.⁴

As mentioned above, previous works have tended to work directly with the set of possible worlds W , rather than with a set of formulas that characterize W as we do here.

Rigidity

The agent's knowledge will include atomic facts about various terms, like the above knowledge about the term *paper.tex*. We also allow functions in our representation. For example, the agent might know various function values like $K(\text{size}(\text{paper.tex}) = 1024)$, i.e., that *paper.tex* is 1024 bytes in length.

Terms composed from functions and constants, like *paper.tex*, 1024, and $\text{size}(\text{paper.tex})$, can be problematic when dealing with knowledge. In particular, terms may be rigid or non-rigid. Non-rigid terms are terms whose denotation (i.e., meaning) varies from world to world, while rigid terms have a fixed denotation (i.e., the same meaning) across worlds. To avoid certain complications that arise when these

²See (Fagin *et al.* 1995) for an introduction to such logics.

³That is, the accessibility relation is such that every world is accessible from every world.

⁴We will always interpret non-modal formulas at the real world.

two types of terms are mixed⁵ we impose the restriction that all constants must be rigid. Thus, a term like *paper.tex* will always denote the same object in every world. On the other hand, we allow functions to generate non-rigid terms. Thus, a term like *size(paper.tex)* can denote a different value in different worlds. We assume that numeric functions, like “+”, or numeric predicates like “<” have their standard interpretation in every world and are therefore rigid.

Formally, our rigidity assumption means that for every constant c , the agent’s knowledge (the set **KB**) includes the formula:

$$(\exists x).K(x = c). \quad (1)$$

This formula asserts that there is a particular object in the real world such that in every possible world the constant c denotes that object. We will also assume for simplicity that all worlds have identical domains of discourse (we do not assume that we know the domain).

Databases

We represent the agent’s knowledge by a collection of four databases, the contents of each having a fixed translation to a collection of formulas in the modal logic of knowledge. These databases are presented next.

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. K_f is further restricted so that all the terms that appear in any literal must be constants. So, for example, an atomic formula like *readable(parent(f))*, where the function *parent* specifies the parent directory of a file f , cannot appear in K_f .

In addition to literals, K_f can also contain specifications of function values. In particular, we allow formulas of the form $f(c_1, \dots, c_n) = c_{n+1}$ or $f(c_1, \dots, c_n) \neq c_{n+1}$, where f is an n -ary function and the c_i are all constants. An equality formula specifies that f ’s value on this particular set of arguments is or is not the constant c_{n+1} . In effect, our restriction means that function values in K_f are considered to be known by the agent only if they can be “grounded” out as constant values.

For every formula $\ell \in K_f$, **KB** includes the formula:

$$K(\ell). \quad (2)$$

For example, if the formula *readable(paper.tex)* is in K_f , **KB** includes the formula $K(\text{readable}(\text{paper.tex}))$ (the agent knows *paper.tex* is readable).

K_w : The second database contains a collection of formulas every instance of which the agent either knows or knows the negation. In particular, K_w can contain any formula that is a conjunction of atomic formulas. By adding simple ground atomic facts to K_w we can model the effects of sensing actions. For example, if the agent at plan time hypothesizes applying an action that senses some fact like *readable(paper.tex)*, all the agent will know at plan time is that after executing the action it will know whether or not this fact is true. Only at execution time will there be a resolution of which of these disjunctions actually holds.

⁵See Garson (Garson 1977) for a good discussion of these issues.

By adding formulas containing variables to K_w we can model the plan time effects of actions that generate universal effects. For example, the UNIX command *ls(d)* yields universal knowledge about the contents of directory d : after applying *ls(d)* we will know, for every file, whether or not that file is in directory d . The actual names of the files in d , however, will only be known at run time.⁶

For every formula $\phi(\vec{x}) \in K_w$ (a conjunction of atomic formulas in which the variables in \vec{x} appear free), **KB** includes the formula

$$(\forall \vec{x}).K(\phi(\vec{x})) \vee K(\neg\phi(\vec{x})). \quad (3)$$

For instance, if *bomb(B₁)* is in K_w , **KB** includes the formula $K(\text{bomb}(B_1)) \vee K(\neg\text{bomb}(B_1))$: the agent is in a state where in every possible world B_1 is a bomb or in every possible world B_1 is not a bomb. At run time, when the agent actually executes the sensing action that added this fact to K_w , the agent will have definite knowledge about *bomb(B₁)*. The value of K_w knowledge at plan time is that the agent can legitimately construct a conditional branch in its plan at this point, branching on the value of *bomb(B₁)*: it is assured that it will have sufficient knowledge at run time to decide which of these branches needs to be executed. By utilizing the contents of K_w in this manner the agent can ensure that it is building a correct plan since it will know which facts it will legitimately be able to branch on at run time.

Some predicates, e.g., numeric predicates like “<” and equality “=”, have the same denotation in every world. Such “rigid” predicates are considered to be implicitly in K_w . For example, the formula $c < 1024$, where c is a constant, is implicitly in K_w : since c is rigid the agent knows that this formula is true in all worlds or false in all worlds.

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. In particular, K_v can contain any unnested function term. For example, $f(x, a)$ would be a legal entry in K_v but $f(g(a), c)$ would not be. Like K_w , the entries in K_v are used to model sensing actions, except in this case the sensors are returning constants (e.g., numbers) not truth values. The value returned by the sensor will not be known until execution time, but at plan time the agent will know that such a value will become known.

For every formula $f(\vec{x}) \in K_v$, where \vec{x} is the set of variables appearing in the term, **KB** includes the formula

$$(\forall \vec{x})(\exists v).K(f(\vec{x}) = v). \quad (4)$$

Formulas of this type are a standard way of specifying that the agent knows a function value, see, e.g., (Scherl & Levesque 1993). For example, if *size(paper.tex)* is in K_v , **KB** includes the formula $(\exists v).K(\text{size}(\text{paper.tex}) = v)$ (the agent knows that *paper.tex* has the same size in every possible world).

K_x : The fourth database is new to this paper. It contains information about a particular type of disjunctive knowledge,

⁶Actions such as *ls* generate local closed world (LCW) information at run time (Etzioni, Golden, & Weld 1997). K_w can be thought of as a plan time analog of the approach of Etzioni *et al.* for modelling such information. See (Bacchus & Petrick 1998) for more on the distinction between plan time and run-time knowledge.

namely “exclusive or” knowledge of literals. Entries in K_x are of the form $(\ell_1|\ell_2|\dots|\ell_n)$, where each ℓ_i is a ground literal (ground functional equalities are permitted). Intuitively, such a formula represents knowledge of the fact that “exactly one of the ℓ_i is true.” For each formula $\phi \in K_x$, **KB** includes the formula

$$K\left(\bigvee_{i=1}^n \ell_i \wedge (\neg\ell_1 \wedge \dots \wedge \neg\ell_{i-1} \wedge \neg\ell_{i+1} \wedge \dots \wedge \neg\ell_n)\right). \quad (5)$$

For example, if $(\textit{infected}(I_1)|\textit{infected}(I_2)) \in K_x$ then the formula $K((\textit{infected}(I_1) \wedge \neg\textit{infected}(I_2)) \vee (\neg\textit{infected}(I_1) \wedge \textit{infected}(I_2)))$ is in **KB**. That is, the agent knows that one and only one of $\textit{infected}(I_1)$ or $\textit{infected}(I_2)$ is true. This form of incomplete knowledge is common in planning scenarios.

Knowledge States

Given a set of these four databases (a particular **DB**), the translation specified by the set of formulas 1–5 defines the agent’s knowledge state (the corresponding **KB**).⁷ Thus, the contents of the databases have a fixed formal interpretation in a first-order logic of knowledge.

Because the database contents can be interpreted at the knowledge level, the restrictions we have placed on the database contents result in restrictions in the types of knowledge that can be modelled. At the possible world level, this means there are certain configurations of possible worlds that cannot be modelled.

Say we have worlds w_1, w_2, w_3 , so that at w_1 : $P(a), Q(b, c)$ holds, at w_2 : $P(a), \neg Q(b, c)$ holds, and at w_3 : $\neg P(a), Q(b, c)$ holds. Since our representation does not allow us to model knowledge of general (non-exclusive) disjunctions such as $K(P(a) \vee Q(b, c))$, we cannot represent the knowledge state characterized by the set $\{w_1, w_2, w_3\}$ of possible worlds. Of course this also means that we cannot generate plans from such an initial state.

Querying a Knowledge State

Planning requires an ability to query the current knowledge state. For example, to determine the set of actions applicable to the initial state the planner must be able to determine which of these action’s preconditions holds in the initial state. Computing an action’s conditional effects also requires querying the knowledge state.

As mentioned above, to retain tractability we must restrict the kinds of reasoning we can perform on **KB** (the knowledge state). To this end we have developed a simple language for representing a set of useful primitive queries about a knowledge state. Let α be any ground atomic formula, and t be any variable free term. The primitive queries we allow are the following.

1. $K(\alpha)$, is α known to be true?
2. $K(\neg\alpha)$, is α known to be false?

⁷The agent’s knowledge state is actually characterized by the set of all models that satisfy the formulas in **KB** in the style of Levesque’s characterization of All I Know (Levesque 1990). See (Bacchus & Petrick 1998) for a discussion.

3. $K_w(\alpha)$, is the agent in a state where α is true in all worlds or α is false in all worlds?
4. $K_v(t)$, is t known to have a fixed value in every world?
5. The negation of any of the above four queries.

To evaluate such queries, an inference algorithm, **IA**, has been developed. The truth of a primitive query is determined by checking the status of the databases. In (Bacchus & Petrick 1998), **IA** is specified and shown to be sound. That is, every inference is correct and is entailed by the knowledge state **KB**. **IA** is, however, incomplete. That is, there are conclusions that can be entailed from **KB** that can’t be established by **IA**. We do not have a formal characterization of when **IA** is complete, but in all of the planning domains we have experimented with **IA**’s incompleteness has not proved to be an impediment to finding plans.

Planning Problems

A planning problem is a four tuple $\langle I, G, \mathcal{A}, \mathcal{U} \rangle$, where I is the initial state, G is the set of goal conditions, \mathcal{A} is a non-empty set of action specifications, and \mathcal{U} is a set of domain specific knowledge update rules.

The initial state I is specified by describing the initial contents of each database. This initial configuration of the databases defines the agent’s initial knowledge state. The goal conditions G are specified as a conjunctive set of primitive queries. For a goal to be satisfied by a plan P , every primitive query in the goal set must be satisfied in every knowledge state that could arise from executing P . (P has conditional branches so its execution could give rise to different knowledge states.) \mathcal{A} and \mathcal{U} are discussed below.

Representing Actions

As mentioned above, actions are represented as updates to the collection of databases. Hence, actions specify how they change the agent’s knowledge state, rather than how they change the state of the world. More specifically, actions have three components: parameters, preconditions, and effects.⁸

Parameters: The parameters are a set of variables that are bound to produce a particular instance of the action. Action parameters may be used in the action’s preconditions and effects.

Preconditions: The preconditions are a conjunctive set of primitive queries about **KB**. For a precondition to be satisfied, each primitive query in the set must evaluate to true.

Effects: An action’s effects are specified by a list of conditional effect statements of the form $C \Rightarrow E$. Each condition C is a conjunctive set of primitive queries. Each effect E is specified as a set of additions or deletions to the four databases. For example, effects such as $\textit{add}(K_f, \textit{size}(\textit{project.tex}) = 1024)$ specifies adding the function value $\textit{size}(\textit{project.tex}) = 1024$ to the K_f database. In general, updates can be applied to any of the databases.

⁸In our approach, an action’s effects are actually divided into *plan time* effects and *execution time* effects. In this paper, however, we will only focus on plan time effects. See (Bacchus & Petrick 1998) for a discussion of the distinction between these two types of effects.

Action	Precondition	Effects
$drop(x)$	$K(holding(x))$ $K(\neg broken(x))$	$del(K_f, holding(x))$ $add(K_f, onfloor(x))$ $add(K_f, dropped(x))$ $del(K_f, \neg broken(x))$ $K(fragile(x)) \Rightarrow$ $add(K_f, broken(x))$
$inspect(y)$	$\neg K_w(broken(y))$	$add(K_w, broken(y))$

Table 1: *drop* and *inspect* actions

Modelling action effects as database updates means that it is easy to compute the new knowledge state that arises from the application of an action. This is in contrast to the situation where incomplete knowledge is represented as a set of possible worlds. In that situation one needs to compute the effect of the action on each of the worlds in the set, which can be difficult even when done symbolically via a BDD.

An Example: Consider the definition of the actions *drop* and *inspect* in Table 1. For an agent to consider applying the *drop*(x) action it must know that it is holding x and that x is not broken. *drop*(x)’s effects on the real world are that it always causes x to be on the floor and to be dropped, and breaks x if x is fragile. At the knowledge level, however, things are somewhat different. Since *onfloor*(x) and *dropped*(x) are always caused, the agent will know that they are true after performing the action. However, if the agent does not know that x is fragile, it will not come to know that x is broken: in some worlds x will be fragile and broken, and in others not fragile and not broken. Thus it will lose knowledge that x is not broken ($del(K_f, \neg broken(x))$), and it will only gain knowledge of broken if it knows that x is fragile.

Notice, however that after a drop the agent does in fact come to know something about fragile, $K((broken(x) \wedge fragile(x)) \vee (\neg broken(x) \wedge \neg fragile(x)))$, and that if we were representing knowledge as a set of possible worlds we would capture this information. However, our representation cannot capture this information as it does not fit our syntactic restrictions. This is an example of where our representation is abstracting away from fine distinctions: it can be computationally expensive to track these distinctions and in many planning domains it is not necessary to keep track of knowledge of this form.

Consider the initial state defined by $K_f = \{\neg broken(vase), holding(vase), \neg broken(box), holding(box), fragile(vase)\}$. The precondition of *drop*($vase$) is satisfied in this initial knowledge state, and applying it yields the new knowledge state defined by $K_f = \{onfloor(vase), holding(box), fragile(vase), broken(vase), dropped(vase)\}$. In particular, the condition *fragile*($vase$) was known, so *broken*($vase$) will be known after the action is performed.

The precondition of *drop*(box) is also satisfied in the above initial knowledge state. However, the effect precondition $K(fragile(box))$ is not satisfied, so the conditional effect cannot be applied. The new knowledge state in this case is $K_f = \{holding(vase), onfloor(box), fragile(vase), dropped(box)\}$. If we now apply the action *inspect*(box),

we obtain the database $K_w = \{broken(box)\}$, with K_f unchanged. That is, after the *inspect* the agent will have sensed the value of *broken*(box).

Domain Specific Update Rules

A planning problem may also include a set \mathcal{U} of domain specific update rules. These rules have the same form as those for conditional effects ($C \Rightarrow E$), where C is a conjunctive set of primitive queries and E is a set of database updates. These updates correspond to state invariants at the knowledge level and may be triggered in any knowledge state where the preconditions for a specific update are satisfied.

Strictly speaking domain specific update rules are not necessary. Rather, one could always elaborate the action effects so as to capture these invariants. This approach is common in classical planning. For example, in the blocks world the stack action requires that the block to be stacked on be clear and it deletes clear, thus maintaining the state invariant that a block can have only one block on top of it. Dealing with knowledge level invariants this way tends, however, to be more cumbersome. Thus we have added the notion of domain specific update rules to ease the task of writing knowledge level domain specifications.

For example, in the above example we could add the following update rules:

- $K(broken(x)) \wedge K(dropped(x)) \Rightarrow add(K_f, fragile(x))$.
- $K(\neg broken(x)) \wedge K(dropped(x)) \Rightarrow add(K_f, \neg fragile(x))$.

Since *drop* only operates on objects known to be unbroken, if we know that an object was dropped and was broken it must have been broken by the drop, and thus it must have been fragile. Similarly, if the object was dropped and remains unbroken it must not be fragile. These update rules would allow knowledge level conclusions about fragile to be drawn when we obtain knowledge about broken.

Consistency Rules

The standard STRIPS formalism involves only a single database. In our case, however, since we have distinct databases we must ensure that their contents remain mutually consistent. As with domain update rules, we have found that we can facilitate the specification of actions by providing a collection of standard, domain independent consistency rules. These rules allow the actions to specify an update to one database with the necessary updates to the other databases being performed automatically. Our consistency rules maintain the following conditions:

1. There is no formula α such that α and $\neg\alpha$ are both in K_f .
2. No function $f(c_1, \dots, c_n)$ in K_f can map to more than one distinct constant.
3. If a literal ℓ is added or deleted from K_f as the result of a non-sensing action, remove all formulas from K_x that mention ℓ or $\neg\ell$.
4. If a literal ℓ is added to K_f as the result of making a conditional branch in the plan, and there exists a formula $\phi \in K_x$ such that $\phi = (\ell_1 | \dots | \ell_m)$, then

- (a) if $\ell \equiv \ell_i$ for some i , delete ϕ from K_x and add $-\ell_j$ to K_f for each $j \neq i$.
- (b) if $\ell \equiv -\ell_i$ for some i , delete ϕ from K_x and add the formula $(\ell_1 | \dots | \ell_{i-1} | \ell_{i+1} | \dots | \ell_m)$ to K_x .

Rules 1 and 2 are simple consistency conditions for K_f . In the specification of a planning problem, the initial state must include a K_f database that meets the consistency requirements of rules 1 and 2.

Rules 3 and 4 maintain consistency between K_f and K_x . In general, sensing actions and ordinary (non-sensing) actions are different: sensing actions observe the state of the world, whereas ordinary actions make changes to the state of the world. In terms of our representation, sensing actions add formulas to K_v and K_w , while ordinary actions add formulas to K_f . K_f , however, may also be updated by adding assumptions arising from adding conditional branches to the plan (see Knowledge-based planning, below). This distinction is important for ensuring the correct interaction between K_f and K_x : additions to K_f resulting from making assumptions may allow K_x formulas to be resolved, adding additional facts to K_f (the agent's knowledge is refined as a result of making an assumption about the outcome of an observation), whereas additions to K_f resulting from ordinary actions may cause K_x knowledge to be lost (an action may change the world so that a K_x formula is no longer valid).

For example, if the formula $\phi = (\text{infected}(I_1) | \text{infected}(I_2))$ is in K_x , adding $\text{infected}(I_2)$ to K_f by assumption along a conditional branch would delete ϕ from K_x and add $-\text{infected}(I_1)$ to K_f . We could add such a conditional branch if a sensing action had added $\text{infected}(I_2)$ to K_w . Along the branch where we assume $\text{infected}(I_2)$ we could also conclude $-\text{infected}(I_1)$ since our K_x knowledge tells us that only one of these holds. If, on the other hand, $\text{infected}(I_2)$ was added to K_f by an action that caused the infection we would simply remove ϕ from K_x —it would no longer be necessarily true that only one of $\text{infected}(I_1)$ or $\text{infected}(I_2)$ holds.

Knowledge-Based Planning

In this section we describe the operation of PKS,⁹ a forward-chaining planner that is able to construct conditional plans in the presence of incomplete knowledge. The planning algorithm *PlanPKS* is given in Table 2.

The planning algorithm *PlanPKS*(**DB**, P , G) takes as input a database collection **DB**, an initial plan P , and a set of goals G . *PlanPKS* returns a plan that achieves the goals, or failure. A plan is a nested list of action sequences, where the nesting indicates conditional branches in the plan. Given a problem specification $\langle I, G, \mathcal{A}, \mathcal{U} \rangle$, the planning algorithm is initially invoked as *PlanPKS*(I, \emptyset, G), where I is the initial database collection, \emptyset indicates an empty plan, and G is the goal set.

The algorithm attempts to grow the plan by non-deterministically choosing to add either a new action or a new conditional branch.

Action Addition: Adding an action is itself a non-deterministic choice from among the set of actions whose

```

begin PlanPKS(DB,  $P$ ,  $G$ )
  if goalsSatisfied(DB,  $G$ ) then
    return  $P$ 
  else Choose
    pick( $A$ ) : precondsSatisfied( $A$ , DB) ;
    applyEffects( $A$ , DB, DB') ;
    return PlanPKS(DB', ( $P$ ,  $A$ ),  $G$ )
  or
    pick( $\alpha$ ) :  $\alpha$  is a ground instance of an entry in  $K_w$  ;
    branch(DB,  $\alpha$ , DB1, DB2) ;
     $C := \{ \text{PlanPKS}(\text{DB}_1, \emptyset, G), \text{PlanPKS}(\text{DB}_2, \emptyset, G) \}$  ;
    return  $P, C$ 
  endChoose
end

```

Table 2: PKS planning algorithm

preconditions hold in **DB** (the preconditions are tested using the inference algorithm **IA**). Once the action is chosen, its effects are applied to update **DB**, after which all of the domain specific updates are applied and the consistency rules enforced. This yields a new database **DB'**. Planning continues from **DB'**.

Conditional Branching: Adding a conditional branch first involves making a non-deterministic choice α of a ground instance from the set of formulas in K_w . In our implementation we restrict ourselves to branches on atomic formulas from K_w . If α is a ground atomic formula from K_w then the planner will know whether or not α is true at this point in the plan when the plan is being executed. Thus it will have sufficient knowledge to choose which branch direction it must take at execution time. Since at plan time the planner does not know which branch will be taken it must build a plan to solve for both contingencies.

Once α is chosen, two new databases **DB**₁ and **DB**₂ are created by modifying **DB**. In particular, for both **DB**₁ and **DB**₂ α is removed from K_w , in **DB**₁ α is added to K_f (i.e., α is known to be true in **DB**₁), and in **DB**₂ $\neg\alpha$ is added to K_f (i.e., α is known to be false in **DB**₂). These additions to K_f may trigger various update rules to further augment **DB**₁ or **DB**₂.

PlanPKS is then invoked recursively to achieve the goal set G from each of the new initial databases **DB**₁ and **DB**₂. These new plans are added to the current plan P as a conditional branch point.

Goal Testing: The *goalsSatisfied* step simply evaluates the set of goals G in the current **DB**. If the goal is true, the current plan P is returned. This means that for a conditional branch, *PlanPKS* must satisfy the goals along each branch.

Search: Search is used to implement the non-determinism in *PlanPKS*. We have implemented both a breadth-first search and a depth-first search version of *PlanPKS*, but have not yet implemented any search heuristics or other forms of search control. That is we are employing blind search, with some forms of cycle checking, to find plans. As can be expected breadth-first search does not scale up well. However, the blind depth-first search does surprisingly well, which indicates that there is a significant advantage to our approach of modelling planning problems at the knowledge level.

Plan Correctness: When planning with knowledge precon-

⁹PKS stands for Planning with Knowledge and Sensing.

Action	Precondition	Effects
$dunk(x)$		$add(K_f, disarmed(x))$

Table 3: BT action specification

Action	Precondition	Effects
$dunk(x)$	$K(\neg clogged)$	$add(K_f, disarmed(x))$ $add(K_f, clogged)$
$flush$		$add(K_f, \neg clogged)$

Table 4: BTC action specification

Action	Precondition	Effects
$dunk(x, y)$	$K(package(x))$ $K(toilet(y))$ $K(\neg clogged(y))$	$add(K_f, disarmed(y))$ $add(K_f, clogged(y))$
$flush(y)$		$add(K_f, \neg clogged(y))$

Table 5: BMTC action specification

ditions, plan correctness relies on two criteria. As Levesque (Levesque 1996) points out, not only is it necessary at plan time for the planner to know that the plan will achieve its desired goals, but at run time the planner must have sufficient knowledge at every step of the plan to execute it.

The *goalsSatisfied* step of the planning algorithm ensures that PKS meets the first criterion: plans are constructed so that the goals are satisfied in the knowledge states along every conditional branch of the plan. The second criterion places an important condition on plan generation: a plan should be built so that it doesn't depend on information that will be unknown to the planner at the required time. This requirement is essential for constructing correct conditional plans, where deciding what branch to execute must depend on knowledge obtained earlier in the plan.

Our planner also meets this second criterion. Conditional branches are built based on formulas in K_w , where such formulas are the result of sensing actions. The semantics of K_w , however, ensure the planner will have definite knowledge obtained from the results of the sensing action. Thus, at run time the planner will have sufficient knowledge to determine which branch of a conditional plan should be executed (see (Bacchus & Petrick 1998) for a more detailed discussion).

Empirical Results

Problems Simple at the Knowledge Level

The first set of experiments involve two problem domains which show the value of modelling problems at the knowledge level. These domains become almost trivial at the knowledge level. The first consists of 3 versions of the bomb in the toilet domain, and the second is the medicate domain. **Bomb in the Toilet:** In the standard BT version (Table 3) there is one toilet and p packages, one of which is a bomb. At the knowledge level it is irrelevant which of the p packages is the bomb: we must generate a plan that allows us to *know* that every package is disarmed, and the dunk action is the only action that provides such knowledge. Thus we can only achieve our goal by dunking every package. Note that this abstracts from the distinctions maintained by those approaches that use sets of possible worlds to represent incomplete knowledge. Such approaches will keep track of the individual possibilities as to which package is a bomb. Then they must implicitly reason about the sequence of ac-

tions they will employ to ensure that these actions cover all of these possibilities.

The next version is the BTC version (Table 4) where the toilet becomes clogged on every dunk, we must know that the toilet is unclogged before we can do a dunk, and we have an additional flush action that always unclogs the toilet (thus providing us knowledge of this fact). This version forces the solution to be sequential and thus the results provide a better comparison of our approach with other planners since our planner is not currently capable of generating parallel plans.

The last version is the multiple toilets version with high uncertainty (BMTC) where we do not initially know whether or not the toilets are clogged. At the knowledge level this domain is almost the same as the simpler BTC version. The only change is that the *dunk* action must also specify which toilet to dunk into, thus it has an extra y parameter. Again at the knowledge level all that matters is to gain knowledge that each package is disarmed, and the only way to know that a package is disarmed is to dunk it. To perform a dunk the agent must know that the toilet is not clogged, and the only way to achieve that knowledge is to flush the toilet. Thus at the knowledge level it is irrelevant which particular collection of toilets are initially clogged: to *know* that a toilet is unclogged we have to flush it. In this version, the sets of possible worlds approach faces an explosion in the number of initial states. In particular, with k toilets such approaches would have to represent all 2^k possible sets of clogged and unclogged toilets.¹⁰ That is, it is forced to maintain a distinction between all possible ways that the toilets could be initially clogged, when in fact this is irrelevant. All that matters is that the agent does not know they are not clogged.

The performance of our planner on these problems is shown in Tables 6 and 7. All times are reported in CPU seconds, and we ran our planner on a 450 MHz Sun with 4GB of memory.¹¹ The plans produced are the ones expected: dunk all of the packages one by one, flushing the toilet at each stage to ensure it is unclogged. The performance of our planner on these problems is at least as good as those reported in (Bertoli *et al.* 2001) for their HSCP system, and considerably better than those reported for the CMBP (Cimatti & Roveri 2000) and GPT (Bonet & Geffner 2000) systems. However, results for the HSCP system were only given for very small problems (10 packages in the BT domain, 16 in the BTC domains, and 10 packages/6 toilets for the BMTC domain) which makes a proper comparison impossible.

The results demonstrate that these problems are very easy at the knowledge level. Breadth-first search can also be applied to these domains, but it is much slower since it must investigate all permutations of package dunkings. It is worth noting that our approach can solve the (100,60) BMTC problem in less than 12 seconds. This problem would have over

¹⁰It could be that BDDs can represent this set compactly, however, the point remains the same: the set of possible worlds approach is forced to make many more distinctions that might be necessary for solving the problem.

¹¹This machine is a general compute server, our planner in fact had very modest memory requirements.

#P	BT	BTC	#P	BT	BTC
10	0.00	0.00	60	0.00	0.01
20	0.00	0.00	70	0.00	0.01
30	0.00	0.00	80	0.01	0.02
40	0.00	0.00	90	0.01	0.02
50	0.00	0.01	100	0.02	0.03

Table 6: Results for BT and BTC with #P packages using depth-first search

BMTC – DFS					
(#P, #T)	Time	(#P, #T)	Time	(#P, #T)	Time
(10, 10)	0.00	(10, 20)	0.01	(10, 30)	0.01
(20, 10)	0.02	(20, 20)	0.03	(20, 30)	0.04
(40, 10)	0.15	(40, 20)	0.20	(40, 30)	0.24
(60, 10)	0.57	(60, 20)	0.68	(60, 30)	0.81
(80, 10)	1.63	(80, 20)	1.97	(80, 30)	2.41
(100, 10)	4.78	(100, 20)	5.68	(100, 30)	6.92
(10, 40)	0.02	(10, 50)	0.02	(10, 60)	0.03
(20, 40)	0.05	(20, 50)	0.06	(20, 60)	0.08
(40, 40)	0.29	(40, 50)	0.35	(40, 60)	0.43
(60, 40)	0.97	(60, 50)	1.19	(60, 60)	1.47
(80, 40)	3.05	(80, 50)	3.66	(80, 60)	4.71
(100, 40)	8.17	(100, 50)	9.64	(100, 60)	11.54

Table 7: Results for BMTC with #P packages, #T toilets using depth-first search

Medicate					
#I	Time	#I	Time	#I	Time
20	0.08	50	1.61	80	9.52
30	0.28	60	3.13	90	13.68
40	0.74	70	5.71	100	20.39

Table 8: Results for Medicate with #I infections

2^{60} possible worlds in the initial state.

Medicate: Medicate is another domain that becomes trivial at the knowledge level. In this domain the patient has one of I possible infections, or no infections at all. The plan that must be found involves performing a *stain* action to diagnose the infection the patient has and then conditionally applying the appropriate *medicate* action to cure the infection. Applying the wrong medication will kill the patient.

Once again at the knowledge level all that we need to do is to achieve knowledge that the patient is cured, so our plan simply moves through a sequence of knowledge states where the possible infections are eliminated one by one. We never need to track the different possibilities reflecting which of the uneliminated illnesses might be the actual one. The performance of our planner on this domain is shown in Table 8.

The Knowledge Level: It could be argued that our results for these two domains are incomparable with those reported in (Bertoli *et al.* 2001) since our model of the problem is different. To some extent this is true. But the point we are making here is that we are able to solve the same problem using a more abstract model of the domain. In particular, we are using the same intuitive description of the domain and we are generating the same plans as those generated by HSCP. Hence these results provide evidence of the utility of modelling problems at the knowledge level.

Action	Precondition	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 9: OSMC action specification

Opening a Safe

A far more interesting domain for our planner is the open safe problem. We consider two versions of this problem. In the first version (OSMC) we consider a safe and a fixed number of possible combinations. In the initial state we know that the safe is closed and that one of these combinations is the combination for the safe (this is represented as K_x knowledge in the initial state). The actual combination of the safe is represented as a 0-ary function (*combo*). Since the actual combination is unknown, semantically, this function has different values in different possible worlds. The goal is to know that the safe is open.

There is only one action for this domain which dials a combination x , given in Table 9. If we dial x we no longer know with certainty that the safe is closed ($del(K_f, \neg open)$), but we also know whether the safe is open ($add(K_w, open)$). That is, when this action is actually executed the agent will detect if the safe is open, but at plan time it only acquires K_w knowledge. We also come to know that x was the combination we just tried, and if we know with certainty that x is the combination, $K(x = (combo))$ then we will come to know that the safe is open.

We also have two domain specific update rules which allow us to conclude (1) that the combination just dialled is the safe's combination if it opened the safe, and (2) the combination just dialled is not the safe's combination if it failed to open the safe.

Using blind depth-first search the planner is able to solve fairly large instances of this problem (100 combinations in less than 900 sec.). However, as to be expected with undirected depth-first search, the solutions generated are lengthy and contain many irrelevant instances of the *dial* operator. Eventually, however, the plan does succeed opening the safe.

We are still investigating alternate methods for controlling search in PKS. However, a particularly simple and effective approach to search control is to use extra preconditions to block attempting an action in contexts where it is not useful to perform it (Bacchus & Ady 1999). Given that the goal is to open the safe, there are two obvious ways to use precondition control. First, if we know whether the safe is open, it is clearly better to branch on this possibility rather than trying to dial another combination: on one side of the branch we will have achieved the goal and on the other side we would have achieved more knowledge. This control can be achieved by adding the precondition $\neg K_w(open)$ to *dial*.

OSMC					
#C	Orig.	Pre.	#C	Orig.	Pre.
10	0.07	0.00	60	80.48	0.08
20	0.94	0.01	70	162.95	0.12
30	4.62	0.01	80	289.23	0.18
40	14.59	0.03	90	516.58	0.25
50	36.88	0.05	100	863.25	0.34

Table 10: Results for OSMC with #C combinations

Second, if we know that x is not the combination of the safe, then there is no point in dialling x . This control can be achieved by adding the precondition $\neg K((combo) \neq x)$.

With these two preconditions the performance of the planner is significantly improved. Its performance on the original domain (Orig.) and when control preconditions are added (Pre.), is shown in Table 10. This example illustrates that as in classical planning (Bacchus & Ady 1999) search control can be applied with great effect in the incomplete knowledge situation.

Besides speeding up the planner, search control also greatly improves the quality of the plans generated. The plans constructed consist of a sequence of *dial* actions, one for each possible combination. After each dial the planner has K_w knowledge of *open* and the planner can insert a conditional branch after each dial. On one side of the branch we have that the safe is open. Furthermore, the first domain specific update rule allows us to conclude that x is in fact the combination of the safe (i.e., it just opened the safe), and the domain independent consistency rules use the K_x knowledge to conclude that all of the other combinations are not the safe's combination. This side of the branch achieves the goal of knowing that the safe is open as well as knowledge of what the combination is.

On the other side of the branch the safe is open, and we also know, via the second domain specific update rule, that x is not the combination of the safe. The consistency rules allow us to conclude that only one of the untried combinations can be the combination. This branch will then be extended by dialling another combination. Thus the plan generated will try all of the combinations in sequence stopping when one of them opens the safe.

Run-Time Variables: The second open safe problem is much easier to solve, but it illustrates a very interesting feature of our approach. In this version we have two actions available as shown in Table 11. The extra action is a *readCombo* action, which can be executed if we know *haveCombo*, i.e., we have the safe's combination written on a piece of paper. In the initial state we know *haveCombo*, but we know nothing else about the safe's combination. In particular, we do not have a finite set of different possibilities for its combination. Again the goal is to know that the safe is open.

readCombo is a sensing action that senses the value of the function (*combo*) at execution time. At plan time it adds (*combo*) to K_v : at plan time we will come to know that this term's value is known. Once we K_v the term (*combo*) the preconditions of *dial*((*combo*)) hold, as we $K_v((combo))$, and we know (*combo*) = (*combo*) (and thus

Action	Precondition	Effects
<i>dial</i> (x)	$K_v(x)$ $\neg K((combo) \neq x)$	$add(K_w, open)$ $K((combo) = x) \Rightarrow$ $add(K_f, open)$
<i>readCombo</i>	$K(haveCombo)$	$add(K_v, (combo))$

Table 11: OSSC action specification

that $\neg K((combo) \neq (combo))$). The effects of this action are to know whether the safe is open, and since the precondition of the conditional effect is true (we $K((combo) = (combo))$ since these two terms are syntactically identical), we also come to know that the safe is open.¹²

Thus from an initial state where we know *haveCombo* our planner constructs the plan *readCombo; dial*((*combo*)): first read the combination and then dial it to open the safe. This plan is constructed in time that is below the resolution of our measurements. It should also be noted that the single action *dial*((*combo*)) is not a plan: prior to *readCombo* we do not know the value of (*combo*).

An important feature of this plan is that the value for (*combo*), and thus the parameter that instantiates *dial*, is not known until run time. The term (*combo*) acts as a run-time variable (Etzioni, Golden, & Weld 1997), its value is only determined at run time. However, at plan time we know that the value it will take on at run time will allow the plan to achieve its goal. (That is, no matter what value (*combo*) takes at run time, that value will open the safe.) The ability to generate a plan whose parameters are only filled in at run time is essential when dealing with environments where not all objects are known at plan time. In this case, in contrast to the previous example, we do not know the range of different combinations that could be dialled. Thus the only plan that will work is one with a run-time variable.

UNIX Domain

Our final example is taken from the UNIX domain. The actions for this domain are given in Table 12. A directory hierarchy is defined by the relation *indir*(x, y) (x is in directory y), the current working directory is specified by a 0-ary function (*pwd*), and there are two actions for moving around in this hierarchy and one for gathering information. The first action is a change directory *cd-down*(x) that can move down to a sub-directory of (*pwd*). It requires that x is known to be a sub-directory of the current directory, and its effect is to change the current directory. The second action is *cd-up*(x) that moves up to the parent of the current directory. It requires that it be known that x , the directory to be moved to, contains the current working directory. Finally, the third action is an *ls* action that can sense the presence of a file in the current working directory. Note that the change directory actions operate by modifying the value of the function (*pwd*).

Initially, the planner has knowledge of the value of the current directory, (*pwd*) = *root*, and knowledge of the directory structure: *indir*(*papers*, *root*),

¹²It is not inconsistent to K_w that the safe is open as well as know it (K_f). The K_w information is, however, made redundant by this K_f information.

Action	Precondition	Effects
$cd-down(x)$	$K(directory(x))$ $K(indir(x, (pwd)))$	$add(K_f, (pwd) = x)$
$cd-up(x)$	$K(directory(x))$ $K(indir((pwd), x))$	$add(K_f, (pwd) = x)$
$ls(x, y)$	$K(file(x))$ $K((pwd) = y)$	$add(K_w, indir(x, y))$

Table 12: UNIX domain action specifications

$indir(mail, root)$, $indir(kr, papers)$, $indir(aips, papers)$, and $indir(planning, aips)$. The goal is to move the current working directory to the directory containing the file *paper.tex*. This goal can be specified as the condition $K(indir(paper.tex, (pwd)))$. That is, achieve a knowledge state where it is known that *paper.tex* is in directory *(pwd)*. Initially, the planner has incomplete knowledge of the location of *paper.tex*. In particular, $(indir(paper.tex, planning) | indir(paper.tex, kr))$ is in K_x (file *paper.tex* is in directory *planning* or *kr*).

One solution is the conditional plan: $cd-down(papers)$; $cd-down(kr)$; $ls(paper.tex, kr)$; $cd-up(papers)$; branch on $indir(paper.tex, kr)$: if $K(indir(paper.tex, kr))$ then $cd-down(kr)$, otherwise $cd-down(aips)$, $cd-down(planning)$.

The BFS version of our planner finds this plan in 0.16 seconds. This is the first plan found, but other plans can be found, including one that branches right after the *ls* rather than this one which moves back up to *papers* before branching. Note that the K_w knowledge obtained by *ls* is not destroyed by the $cd-up(papers)$ action, so it is a valid plan to branch after this action rather than before. The planner is also able to solve other more elaborate problems with these actions. If a “move file” action was defined in the domain, another way the goal could be achieved would be by moving the file *paper.tex* to the present directory (whatever it was). Since we do not have such an action, we must move to the file instead.

Conclusions

We have presented a new approach to planning with incomplete knowledge and sensing. Our approach works directly at the knowledge level, modelling how the agent’s knowledge evolves as actions are added to a plan. The advantage of our approach is that it is able to abstract from many irrelevant distinctions that occur at the level of possible worlds but not at the level of “compact” knowledge assertions. As a result we are able to model many additional features, like functions and run-time variables, that appear to be essential for many interesting domains. On the other hand, our approach is limited in its inferential power: plans that could be discovered by reasoning at the level of possible worlds might not be found by our approach. Nevertheless, we consider the trade off between the power of low-level case analysis and the power of richer representations to be worth making, at least in the domains that we are most interested in like the UNIX domain.

There are many issues for future study, including improving the searching capacity of our planner and increasing the

power of the representation so that we can handle additional types of planning problems. One issue that our work raises is the question of just how easy is it to specify domains at the knowledge level. We are currently studying methods for automatically converting a set of actions described by their effects on the world into a set of knowledge-level actions described by their effects on the agent’s knowledge. Some progress has been made on this problem, and solving it would remove this particular concern (Petrick & Levesque 2002).

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 Ady, M. 1999. Precondition control. Available at <http://www.cs.toronto.edu/~fbacchus/on-line.html>.
- Bacchus, F., and Petrick, R. 1998. Modeling and agent’s incomplete knowledge during planning and execution. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 432–443.
- 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.
- Bonet, B., and Geffner, H. 2000. Planning with incomplete information as heuristic search in belief space. In *Proceedings of the International Conference on Artificial Intelligence Planning*, 52–61.
- Bryant, R. E. 1992. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys* 24(3):293–318.
- Cimatti, A., and Roveri, M. 2000. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research* 13:305–338.
- Etzioni, O.; Golden, K.; and Weld, D. 1997. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence* 89(1–2):113–148.
- Fagin, R.; Halpern, J. Y.; Moses, Y.; and Vardi, M. Y. 1995. *Reasoning About Knowledge*. MIT Press, Cambridge, Massachusetts.
- Garson, J. W. 1977. Quantification in modal logic. In Gabbay, D., and Guenther, F., eds., *Handbook of Philosophical Logic, Vol. II*. Dordrecht, Netherlands: Reidel. 249–307.
- Levesque, H. J. 1990. All I Know: A study in autoepistemic logic. *Artificial Intelligence* 42:255–287.
- 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.
- Petrick, R. P. A., and Levesque, H. J. 2002. Knowledge equivalence in combined action theories. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, 303–314. Morgan Kaufmann.
- Pryor, L., and Collins, G. 1996. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research* 4:287–339.
- Scherl, R. B., and Levesque, H. J. 1993. The frame problem and knowledge-producing actions. In *Proceedings of the AAAI National Conference*, 689–695. AAAI Press / MIT Press.
- Weld, D., and Etzioni, O. 1994. A softbot-based interface to the internet. *Communications of the ACM* July:72–76.