Modeling an Agent's Incomplete Knowledge during Planning and Execution

Fahiem Bacchus
Department of Computer Science
University of Waterloo
Waterloo, Canada N2L 3G1
fbacchus@logos.uwaterloo.ca

Ron Petrick
Department of Computer Science
University of Waterloo
Waterloo, Canada N2L 3G1
rpapetrick@logos.uwaterloo.ca

Abstract

In many domains agents must be able to generate plans even when faced with incomplete knowledge of their environment. We provide a model to capture the evolution of the agent's knowledge as it engages in the activities of planning (where the agent must attempt to infer the effects of hypothesized actions) and execution (where the agent must update its knowledge to reflect the actual effects of actions). The effects (on the agent's knowledge) of a planned sequence of actions are very different from the effects of an executed sequence of actions, and one of the aims of this work is to clarify this distinction. The work is also aimed at providing a model that is not only rigorous but can also be of use in developing planning systems.

1 Introduction

In this paper we address the problem of how agents who must operate in incompletely known environments can generate and execute plans. In particular, we examine the case where an agent has correct but incomplete knowledge of its environment. A representation scheme for incomplete knowledge is developed that is specifically oriented towards the development of actual planning systems. In particular, we focus on representing and updating the kinds of incomplete knowledge that would be useful to a planning agent capable of sensing and manipulating its environment, and we ensure that the representation can be used in a straightforward manner in an actual planning system.

When planning, the agent must reason about the effects of actions. When the agent has complete knowledge of its environment, there is no need to distinguish between what the agent knows and what is true in its environment. Hence, in classical planning systems there is no explicit separation between the agent's knowledge and facts about the world. For example, when a STRIPS database is employed to model the world state it is only implicit that the agent knows the contents of the database.

When faced with incomplete knowledge, however, we do require an explicit model of the agent's knowledge and the manner in which this knowledge is affected by the actions executed by the agent. In fact, for the purposes of planning it is the action's effects on the agent's knowledge that are most important: at plan time the agent must *know* that the plan will achieve its desired effects, and at execution time the agent must have sufficient *knowledge* at every step of the plan to execute it [Lev96].

A major complication, when having to reason about how actions affect the agent's knowledge, arises from the fact that the plan time effects of such actions are quite different from their execution time effects. For example, say that the agent is operating in the UNIX domain and that it is considering the action of listing a directory. At plan time all that it will know is that after the action it will know all of the files in the directory: the actual identity of those files will not become known until the action is actually executed.

In many domains generating plans that operate correctly no matter how the world is configured is impossible—such conditional plans [PS92, PG93] end up being too large. Instead, the agent must often commit by actually executing some actions so as to avoid having to plan for contingencies that never occur. However, execution also has its pitfalls, as executing an action might change the world in such a way that the agent's ultimate goal becomes impossible to achieve.

Understanding how to manage these tradeoffs so that we can effectively interleave planning and execution remains an important open problem in the area. We believe that our work makes a contribution to this problem. In particular, our representation of actions provides a clear separation between their plan time and execution time effects. We can project the agent's knowledge state through both planned actions sequences and executed action sequences. This provides useful information about the differences between plan time and execution time and leads to a deeper understanding of both plan time and execution time effects. It also opens up a wider range of possibilities for interleaving planning and execution.

The general approach we adopt is much like the traditional STRIPS representation. In particular, we use a collection of databases to represent the agent's knowledge. However, we provide a formal semantics for the items in each database. We do this by translating each of these items into formulas of a modal logic of knowledge. Actions operate much like STRIPS actions do: they modify the contents of the various databases. Through examples we show that a useful range of actions can be represented as update operations to these databases.

Our approach allows us to project the agent's knowledge through a sequence of planned actions: we simply apply the actions' plan time effects to the agent's initial knowledge state to produce a sequence of intermediate knowledge states. This means that a straightforward forward chaining search could in principle be used to generate plans. We can also project the agent's knowledge state through sequences of action executions, and this means that an plan execution module can also be supported by our formalism.[1].

In the rest of the paper we will present the method we use for representing the agent's knowledge, discuss how inferences can be made from this knowledge, and briefly discuss how actions are represented and how they update the agent's knowledge. Finally, we will close with some simple examples that show how our approach models the plan time and execution time effects of various actions and plans. But first we discuss some related work.

[1]Some work would have to be done to modify our approach to support partial order planning or backwards chaining planning. This should be possible as such planning technologies were initially developed from "projective" action semantics like ours. However, such approaches are not a major interest of ours, as we are pessimistic about their ultimate future. We are much more optimistic about the future of forward chaining planners [BK96, McD96]

1.1 Related work

The general issue of planning with correct but incomplete knowledge has received a great deal of attention recently. There are many domains that can be usefully modeled under this paradigm. For example, Etzioni, Golden and Weld have been engaged in ongoing research into software agents that operate in the UNIX and Internet environments [EGW97, GW96, EGW94, GEW94]. As they point out, these domains are reasonably approximated by the assumption of correct but incomplete knowledge. The main feature of their work has been to develop methods for providing such agents with planning capabilities: exactly the issue we address here. Their work, particularly their work on locally closed worlds [EGW97] has been very influential in our work.

There are two main differences between their work and that presented here. First, much of their approach is tied to the technology of partial order planning. We feel that this often has the detrimental effect of making the semantics of their representations and algorithms much more difficult to understand. The projective semantics we use here gives a clearer separation between the issues that involve the semantics of actions and the agent's knowledge and the issues that involve the implementation and semantics of partial order planning. The second difference is that their work is intimately tied to execution time effects. For example, the algorithms they develop for reasoning about locally closed world conditions [EGW97] assume that the actions achieving such conditions have been executed. This means that the planning system they construct is forced to interleave planning and execution in an inflexible manner. There is limited scope for alternative ways of interleaving planning and execution, to deal, e.g., with domains where executing actions can produce irreversible changes.

As pointed out by Levesque, there are a number of subtle issues involved in planning in the face of incomplete knowledge. In [Lev96] Levesque provides a formal specification of plan correctness in the face of incomplete knowledge. He points out that plans have knowledge preconditions, and that it must be known at plan time that these conditions will be achieved at execution time. Although Levesque's work provides vital insights into the problem, his work does not directly address the issue of generating plans. In particular, his model of actions and knowledge is specified in the situation calculus. Hence, to reason about the effects of actions one would, in general, have to employ full first-order inference. In our work we have used some of Levesque's ideas about plan correctness, but have focused on more limited representations that can be implemented more effectively in real planning systems.

2 Representing the Agent's Knowledge

The first issue we address is that of representing the agent's knowledge. As mentioned in the introduction we are assuming that the agent has correct, albeit incomplete, information about its environment. This kind of information is conveniently formalized using a standard modal logic of knowledge (see [FHMV95] for an introduction).

One of our aims, however, is to develop an approach that can facilitate the development of effective planning systems, and we do not know, at this time, how to deal with a fully general logic of knowledge. Instead we adopt a STRIPS like approach where by the agent's knowledge is represented as a collection of databases each of which maintains a particular type of knowledge. We formally characterize the agent's knowledge by providing a translation from the database contents to a set of logical formulas. Thus we utilize the logic's semantics as the underlying semantics of our representation.[2] We use **DB** to represent the agent's databases, and **KB** to represent the set of logical formulas that characterize the agent's knowledge.

In brief, the 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 worlds W, each of which is an ordinary first-order model. These worlds are related to each other by an accessibility relation. In this case every world is accessible from every other world. Any non-modal formula ϕ is interpreted to be true at a particular world w (written $w \models \phi$) iff it is true according to the standard rules for interpreting first-order formulas. A formula of the form $K(\phi)$ is interpreted to be true at w iff ϕ is true at every world accessible from w, which means that ϕ must be true at every world in W (since at every world all worlds are accessible).

Intuitively, the agent's knowledge is being modeled by the set W. The agent does not know which of the worlds in W is the real world, and considers all of these worlds to be possible versions of the way the real world is 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. Knowing ϕ to be true means that ϕ is true in every world in W. Our assumption that the agent's knowledge is correct is modeled by the fact that the real world is a member of W. Thus, if the agent knows ϕ, ϕ is in fact true in the real world. For convenience, we use the notation w^* to represent the real world. Furthermore, when we write a logical formula we always interpret it at w^*. Thus, a formula like $K(\mathit{readable}(\mathit{kr.tex})) \wedge \mathit{writable}(\mathit{kr.tex})$ means that the agent knows that file *kr.tex* is readable (and by the semantics of K, *kr.tex* is in fact readable) and that it is in fact writable (but this is not necessarily known by the agent). A useful notation is $K_{\mathrm{whe}}(\phi)$ which is defined to be the formula $K(\phi) \vee K(\neg\phi)$: either ϕ or its negation is known to hold.

2.1 Rigid Terms and Constant Domains of Discourse

The agent's knowledge will include atomic facts about various terms. For example, knowing that the file *kr.tex* is readable might be represented by the atomic formula $K(\mathit{readable}(\mathit{kr.tex}))$, where *kr.tex* is a term of the language. We also allow functions. For example, the agent might know various function values like $K(\mathit{size}(\mathit{kr.tex}) = 1024)$, i.e., *kr.tex* is 1024 bytes in length.

Terms composed from functions and constants, like *kr.tex*, 1024, and *size*(*kr.tex*), pose potential problems when dealing with knowledge. In particular, the terms they generate may be rigid or non-rigid. Non-Rigid terms are terms whose denotation varies from world to world, while rigid terms have a fixed denotation across worlds. For example, the agent might not know the size of the file *kr.tex*, so the term *size*(*kr.tex*) may have a different denotation (i.e., a different value) in the different worlds the agent considers possible. On the other hand a number like 1024 would have the same denotation (i.e., the same meaning) in every world.

When terms can be of either type reasoning about facts like *readable*(*kr.tex*) becomes more complex.[3] For example, it is not immediately obvious what it would mean for the agent to know this fact if the term *kr.tex* had potentially a different denotation in every world. Since there does not seem to be a good reason to have this level of generality, we impose the restriction that all constants must be rigid. Thus, a term like *kr.tex* will always denote the same object in every world.[4] On

[2] In essence we are simply restricting ourselves to a particular subset of the logic.

[3] See Garson [Gar77] for a good discussion of these issues.

[4] There may be many files in the agent's environment called *kr.tex*. In practice, we would have to use a distinct constant for each file. For example, we could use a unique

the other hand, we allow functions to generate non-rigid terms. Thus, a term like *size*(*kr.tex*) can denote a different value in different worlds.

Formally, this means that for every constant c in the language describing any particular planning domain, the agent's knowledge (the set **KB**) includes the formula:

$$\exists x.K(x = c). \tag{1}$$

This says that there is a particular object in the real world such that in every possible world the constant c denotes that object.

We assume that numeric functions, like "+", or numeric predicates like "<" have their standard interpretation in every world (hence they also are rigid).

Another complication that we wish to avoid are those that arise when different worlds w can have different domains of discourse.[5] So we restrict our semantics to only consider models in which all worlds have an identical domain of discourse.[6]

2.2 The Databases

We represent the agent's knowledge by a collection of four databases, each of which is discussed below.

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 (atomic formula or negation of an atomic formula). 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*(..(*dir*)), where the function ".." specifies the parent directory of a direction file, cannot appear in K_f. To include such information we would have to know the name of *dir*'s parent directory.

In addition to literals K_f can also contain specifications of function values. In particular, formulas of the form $f(c_1, \ldots, c_n) = c_{n+1}$, where f is an n-ary function and the c_i are all constants. This formula specifies that f's value on this particular set of arguments is 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.

identifier for each file and have a function *name* that maps this identifier to the file's "common" name. The function *name* may thus map many different files to the same common name. However, for readability we will continue to use common names in our examples, leaving it to the reader to remember that all such names are intended to be unique.

[5] Again see [Gar77] for a discussion.

[6] We have not found that this poses any practical problems. In particular, this assumption does not mean that we know the identity of all the objects in the real world.

We specify what the contents of K_f means in terms of the agent's knowledge by specifying that for every formula $\ell \in K_f$, **KB** includes the formula:

$$K(\ell). \tag{2}$$

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 at plan time. In particular, at plan time if the agent hypothesizes executing a sensing action that senses some fact like *readable*(*kr.tex*), all the agent will know is that after sensing it will know whether or not this fact is true. Only at execution time will there be a resolution of this disjunction.

In a similar manner by adding formulas containing variables to K_w we can model the plan time effects of actions that generate universal effects like local closed world information [EGW97]. For example, the UNIX "*ls dir*" command yields local closed world information about the contents of directory *dir*. Yet at plan time the agent will not know the actual contents of the directory. The contents will only become known after the *ls* action is executed.

We specify what the contents of K_w means in terms of the agent's knowledge by specifying that 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

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

Note that in the case where $\vec{x}$ is the empty set (i.e., ϕ is a conjunction of ground atomic formulas), this reduces to the formula $K_{\text{whe}}(\phi)$.

Some predicates, e.g. numeric predicates like $<$ and equality $=$, have the same denotation in every world in W. Such "rigid" predicates are considered to be implicitly in K_w. For example, $x > y$ and $x = y$ are implicit members of K_w. The inference algorithm presented below has access to these implicit members of K_w.

K_v: The third database is simply a specialized version of K_w designed to store information about various function values the agent will come to know. 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 can be 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

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

Formulas of this type are a standard way of specifying that the agent knows a function value, see, e.g., [SL93].

More general information about knowing function values can be specified by entries in K_w. For example, if we will come to know the sizes of all the files in a particular directory *dir*, we could place $\textit{in-dir}(x, \textit{dir}) \wedge \textit{size}(x) = y$ in K_w, where $\textit{in-dir}(x,y)$ means that x is in directory y. This formula says that for every file x that is in directory *dir* we know all values of y such that $\textit{size}(x) = y$. Of course since *size* is a function there is only one such y.

***LCW*:** The fourth database is a database of local closed world information. The innovative concept of locally closed worlds comes from the work of Etzioni et al. [EGW97]. *LCW* represents the execution time analog of K_w, and basically asserts that the agent's K_f database contains a complete list of all items satisfying a particular conjunction of atomic formulas. In most cases such a list can only be added to the K_f database by actually executing an action.

LCW can contain formulas of exactly the same form as K_w: conjunctions of atomic formulas. We specify the semantics of the *LCW* database as follows. Let $\phi(\vec{x}) = \alpha_1(\vec{x}) \wedge \ldots \wedge \alpha_k(\vec{x})$ be a conjunction of atomic formulas in which the vector of variables $\vec{x} = \langle x_1, \ldots, x_n \rangle$ appear free. Say that $\phi \in LCW$.[7] Let $C = \{\vec{c} : \alpha_i(\vec{x}/\vec{c}) \in K_f, 1 \leq i \leq k\}$. C is the set of tuples of constants explicitly listed in K_f as satisfying ϕ. For every such formula $\phi \in LCW$, **KB** includes the formula

$$\forall \vec{x}. \bigwedge_{\vec{c} \in C} \neg(x_1 = c_1 \wedge \ldots \wedge x_n = c_n) \Rightarrow K(\neg\phi(\vec{x}/\vec{c})). \tag{5}$$

For example, if $P(x) \wedge Q(x,y) \in LCW$, and $P(a)$, $P(c)$, $Q(a,b)$ and $Q(a,c)$ are all in K_f, (which means that the pairs (a,b), and (a,c) are explicitly listed as satisfying $P(x) \wedge Q(x,y)$ in K_f), then the formula

$$\begin{aligned}&\forall x,y.\neg(x = a \wedge y = b) \wedge \neg(x = a \wedge y = c)\\ &\quad \Rightarrow K\Big(\neg(P(x) \wedge Q(x,y)\Big),\end{aligned}$$

is in **KB**. This formula says that the pairs (a,b) and (a,c) are in fact the only pairs satisfying $P(x) \wedge Q(x,y)$. Thus it entails, e.g., that $K(\neg(P(b) \wedge Q(b,c)))$.

This formula makes explicit the notion utilized by Etzioni et al. that if we have local closed world information and we don't have an instance explicitly listed in the database then we can conclude that the property does not hold.

2.3 The semantics of LCW and K_w

We have provided a semantics for the *LCW* and K_w databases by translating their contents to modal logic formulas. In doing this we are using the well understood semantics of the modal logic to provide a final grounding for the entries in these databases. It is useful to point out that when we convert entries in K_w to formulas of the form $\forall \vec{x}.K(\phi(\vec{x})) \vee K(\neg\phi(\vec{x}))$ this corresponds to the agent knowing that the set of satisfying instances of $\phi(\vec{x})$ is invariant across worlds. That is, a tuple of constants $\vec{c}$ satisfies $\phi(\vec{x}/\vec{c})$ in the real world if and only if it satisfies the formula in every world the agent considers possible.

The presence of such a formula in K_w does not mean, however, that the agent knows the truth value of $\phi(\vec{x}/\vec{c})$, since the action that will resolve this has not yet been executed. When the formula is in *LCW* the action has already been executed and all of the satisfying instances of ϕ have been added to the agent's K_f database by the action. Hence, the agent will know the truth value of $\phi(\vec{x}/\vec{c})$ for every $\vec{c}$. Thus a typical action specification will include a plan time addition to K_w and an execution time addition to *LCW*.

The concept of locally closed worlds as a generalization of the closed world assumption is due to Etzioni et al. who develop the concept in detail in [EGW97]. In our approach, however, we have carefully separated local closed world information into plan time effects and execution time effects. The inference algorithm developed in [EGW97] is an execution time algorithm that requires the actions executed to actually add all of the satisfying instances to the K_f database. At plan time the satisfying instances are not yet known, yet we still want to perform "local closed world" reasoning at plan time. Our approach gives us that ability.

[7]Note that not every variable in $\vec{x}$ need appear free in every literal.

2.4 The Knowledge State

Given a particular set of these four databases, i.e., a particular **DB**, the agent's knowledge state is defined by the set of formulas in **KB** as specified by the formulas 1–5 above. In particular, the agent's knowledge state is characterized by the set of models (in which every possible world has the same domain of discourse) that satisfy all of the formulas in **KB**.

It can be shown that subject to obvious consistency requirements any **DB** specifies a consistent **KB**.

Theorem 2.1 *Let* **DB** *be any set of these four databases subject to the two conditions*

1. *there is no atomic formula α with both α and $\neg\alpha$ in K_f and*

2. *no function $f(c_1, \ldots, c_n)$ is specified to have two distinct values in K_f.*

Then the **KB** *corresponding to* **DB** *is consistent. That is,* **KB** *has a model.*

Proof: In general **KB** will have many models. We show how an arbitrary model can be constructed. First, we let the domain of discourse be the set of all constants appearing in **DB**. Then we construct a single first-order model w by starting with the set of ground literals (and function values) contained in K_f. Then we add to K_f a set of negative facts sufficient to satisfy all of the formulas arising from LCW. Let $\forall\vec{x}. \bigwedge_{\vec{c}\in C} \neg(x_1 = c_1 \wedge \ldots \wedge x_n = c_n) \Rightarrow K(\neg\phi(\vec{x}/\vec{c}))$ be a formula in **KB** arising from a formula $\phi \in LCW$. For every $\vec{c} \notin C$ we pick a conjunct of $\phi(\vec{x}/\vec{c})$, $\alpha_i(\vec{x}/\vec{c})$, that is not in K_f: one such conjunct must exist by the definition of C. In fact, more than one such conjunct may exist, in which case we make an arbitrary choice. We add $\neg\alpha_i(\vec{x}/\vec{c})$ to K_f, thus satisfying that negative instance of ϕ. We do this for every negative instance of every $\phi \in LCW$.

Note that since no positive facts are added to K_f, our additions do not affect what we can infer from LCW. (The sets C of satisfying instances do not change.) Hence, the addition of negative facts to K_f in order to satisfy a formula $\phi \in LCW$ will not affect the additions required to satisfy any other formula $\phi' \in LCW$.

Clearly, the resulting set of facts in K_f continues to satisfy the above two conditions, and thus this set of facts has at least one first-order model. We pick an arbitrary model, w. Finally, we build a model for the modal logic by setting the collection of models W to be simply the set $\{w\}$. It is not difficult to see that this set of worlds W satisfies any formula of the form $\forall\vec{x}.K(\phi(\vec{x})) \vee K(\neg\phi(\vec{x}))$ that could arise from entries in K_w and K_v. ∎

Corollary 2.2 *If actions are specified as additions and deletions to these databases and these updates maintain the obvious consistency conditions, then no sequence of actions can give rise to an inconsistent* **KB***.*

Intuitively, this theorem says that our representational formalism remains much like the classical STRIPS representation. In STRIPS any database is logically consistent and any sequence of actions maintains this consistency. This is true for our representation as well (except we must outlaw obvious inconsistencies). Like STRIPS this has both positive and negative features. On the positive side, a user of our representation need not worry about "breaking" the representation by generating an inconsistent state. On the negative side, the onus is on the user to build an accurate domain model. As with STRIPS the user must ensure that the **KB** represented by the databases makes sense in the domain being modeled, and that the actions update **KB** in an sensible manner. For example, as with STRIPS, if there are state constraints (e.g., the agent can't be carrying an object and have its hands empty at the same time), then the user must ensure that the databases representing the initial world satisfies those constraints and that the actions properly update the databases so as to maintain those constraints.

3 Inference from DB

From its collection of databases the agent can infer various things. An inference procedure is sound if whenever it infers a formula ϕ from **DB** we have that **KB** $\models \phi$; the procedure is complete if **KB** $\models \phi$ implies that ϕ can be inferred by the procedure from **DB**. Unfortunately, complete inference is impractical, as the set of things that follow from **KB** includes all logical truths (this is the famous problem of logical omniscience [Hin75]).

Fortunately planning applications typically do not require particularly complex reasoning. The major requirement is usually to decide whether or not an atomic formula is true or false at a particular point in a plan. When dealing with incomplete knowledge the requirements become more complex, e.g., we may need to determine whether or not the agent will K_{whe} some fact at a particular point in a plan. In Table 1 we present a simple procedure for answering queries about atomic formulas from the databases.

Procedure $\mathbf{IA}(\varepsilon)$
Inputs: Either a ground atomic formula containing the terms $(t_1, \ldots, t_k)$, or a single term. The terms in ε can contain functions but no variables.
Output: $\mathbf{T}$, $\mathbf{F}$, $\mathbf{W}$, or $\mathbf{U}$ subject to the conditions: (1) $\mathbf{T}$ implies $\mathbf{KB} \models K(\varepsilon)$, (2) $\mathbf{F}$ implies $\mathbf{KB} \models K(\neg\varepsilon)$, (3) $\mathbf{W}$ implies $\mathbf{KB} \models K_{\text{whe}}(\varepsilon)$ (know whether) when ε is a formula and $\mathbf{KB} \models \exists x.K(x = \varepsilon)$ when ε is a term, and (4) $\mathbf{U}$ implies the algorithm is unable to conclude anything about ε.

```
1. Simplify all terms by replacing each $t_i$ in $\varepsilon$ by $\mathbf{EvalT}(t_i)$.
2. If $\varepsilon$ is the term $t$ and either (1) $t$ is a constant or (2) there exists a $t' \in K_v$ and a substitution $\theta$ such that $t'\theta = t$, then $\mathbf{return}(\mathbf{W})$. Else $\mathbf{return}(\mathbf{U})$.
3. If $\varepsilon$ is of the form $t_1 = t_2$, then if these two terms are syntactically identical $\mathbf{return}(\mathbf{T})$. Else if $t_1$ and $t_2$ are both constants then $\mathbf{return}(\mathbf{F})$. Else $\mathbf{return}(\mathbf{U})$.
4. If $\varepsilon \in K_f$, then $\mathbf{return}(\mathbf{T})$.
5. If $\neg\varepsilon \in K_f$, then $\mathbf{return}(\mathbf{F})$.
6. If there exists a $\phi(\vec{x}) = \alpha_1(\vec{x}) \wedge \ldots \wedge \alpha_k(\vec{x}) \in LCW$ and a ground instance of $\phi$, $\phi(\vec{x}/\vec{a})$, such that (1) $\vec{a}$ are constants appearing in $K_f$, (2) $\alpha_i(\vec{x}/\vec{a}) = \varepsilon$ for some $i$, and (3) $\mathbf{IA}(\alpha_j(\vec{x}/\vec{a})) = \mathbf{T}$ for all $j \neq i$, then $\mathbf{return}(\mathbf{F})$.
7. If there exists a $\phi(\vec{x}) = \alpha_1(\vec{x}) \wedge \ldots \wedge \alpha_k(\vec{x}) \in K_w$ and a ground instance of $\phi$, $\phi(\vec{x}/\vec{a})$, such that (1) $\vec{a}$ are either constants appearing in $K_f$ or terms $t_i$ appearing in $\varepsilon$ for which $\mathbf{IA}(t_i) = \mathbf{W}$ , (2) $\alpha_i(\vec{x}/\vec{a}) = \varepsilon$ for some $i$, and (3) $\mathbf{IA}(\alpha_j(\vec{x}/\vec{a})) = \mathbf{T}$ for all $j \neq i$, then $\mathbf{return}(\mathbf{W})$.
8. Else return $\mathbf{U}$.
```

Procedure $\mathbf{EvalT}(t)$
Inputs: A variable free term.
Output: t' the simplest term known to be equal to t.

```
1. If $t$ is a constant then $\mathbf{return}(t)$.
2. If $t = f(t_1, \ldots, t_k)$ and $f(\mathbf{EvalT}(t_i), \ldots, \mathbf{EvalT}(t_k)) = c \in K_f$ or we can compute that $f$ on these arguments is equal to $c$ (e.g., when $f$ is an arithmetic function) then $\mathbf{return}(c)$, else $\mathbf{return}(f(\mathbf{EvalT}(t_i), \ldots, \mathbf{EvalT}(t_k)))$.
```

Table 1: Inference Algorithm

This algorithm can be shown to be sound. Its complexity is dominated by the search for ground instances of $\phi(\vec{x})$ in steps 4 and 5. Potentially the number of ground instances of $\phi(\vec{x})$ can be exponential in the number of variables in $\vec{x}$. However, we do not feel that this will be an issue in practice.

As an example of the operation of **IA** consider the query $\mathbf{IA}(\mathit{size}(\mathit{kr.tex}) > 1000)$ when $\mathit{size}(\mathit{kr.tex}) \in K_v$ is the only entry in any of the databases. In this case **IA** will return **W**. Intuitively, since the agent will come to know the value of $\mathit{size}(\mathit{kr.tex})$ it will also come to know whether or not that size is larger than 1000. First **IA** tries to reduce the function term $\mathit{size}(\mathit{kr.tex})$, but no reduction is known as this term is not in K_f. There are no entries in LCW so the algorithm progresses to step 7. The predicate $>$ is rigid and thus $\phi = x > y$ is an implicit entry in K_w (see discussion of K_w above). Since $\mathit{size}(\mathit{kr.tex}) \in K_v$, $\mathbf{IA}(\mathit{size}(\mathit{kr.tex})) = \mathbf{W}$ and the ground substitution $\{x = \mathit{size}(\mathit{kr.tex}), y = 1000\}$ satisfies condition (1). Under this substitution condition (2) is satisfied and (3) is trivially satisfied as ϕ has no other conjunctions.

4 Representing Actions

The previous sections have provided a mechanism for representing an agent's knowledge state in a STRIPS like manner as a collection of databases. We have also provided a mechanism for answering some simple queries from these databases. In this section we show how we can model actions in a very STRIPS like manner as well. In particular, the preconditions of actions involve testing the contents of the various databases, and the action effects bottom out on a set of adds and deletes to the databases. This means that starting at some initial configuration of the agent's knowledge state we can decide what actions can be applied and we can compute what the agent's new knowledge state will be after the action has been applied.

A major theme throughout the paper has been the separation between plan time and run time. This separation is maintained in our action descriptions. Every action has a specified set of plan time effects and a set of run time effects. Both plan time and run time effects are encoded as database updates. This means that we can compute the plan time effects of a sequence of actions or track their execution time effects in the same formalism. This will be illustrated by the examples presented in Section 5, but first we specify more formally the representation of actions.

Actions are specified by four components: the parameters, the preconditions, the plan time effects, and the run time effects.

The action's parameters. This is simply a set of variables that can be bound to produce a particular instance of the action.

The action's precondition. Since it is the agent that is executing or planning the actions a decision on whether or not an action can be executed must be based on the agent's knowledge state: the agent has no direct access to the state of its environment. To this end it is possible to develop a query language for querying the status of its databases. However, to keep things simple we will specify preconditions to be a conjunctive set of primitive queries. All queries in the set must evaluate to true to satisfy the precondition. The primitive queries all utilize the above inference algorithm and they are listed below. In this listing α is any ground atomic formula, and t is any variable free term.

1. $K(\alpha)$, true iff $\mathbf{IA}(\alpha)$ returns $\mathbf{T}$.
2. $K(\neg\alpha)$, true iff $\mathbf{IA}(\alpha)$ returns $\mathbf{F}$.
3. $K_w(\alpha)$, true iff $\mathbf{IA}(\alpha)$ returns $\mathbf{W}$, $\mathbf{T}$, or $\mathbf{F}$.
4. $K_v(t)$, true iff $\mathbf{IA}(t)$ returns $\mathbf{W}$.
5. The negation of any of the above four queries.

The action's plan time effects. These are specified by a list of condition effect statements of the form $C \Rightarrow E$. Each condition C is a conjunctive set of primitive queries, and each effect E is a set of additions or deletions to the four databases.

The action's run time effects. We assume a simple interface between the planner and the execution module. In particular, when an action instance is executed the name of that action is passed to the execution module along with a list of "run-time" variables [GW96]. The execution module binds the run-time variables with information it obtains while executing the action.[8] The execution module may generate a sequence of bindings for the run-time variables. The effects of the action are specified using a list of condition effect statements, $C \Rightarrow E$, as before. For run-time effects, however, C and E may contain any of the run-time variables. Furthermore, C may contain tests on the run-time variables. If $C \Rightarrow E$ contains a run-time variable then this condition effect statement will be evaluated once for every distinct binding of the run-time variables generated by the execution module. On the other hand, when $C \Rightarrow E$ has no runtime variables it is only executed once.

Additions and deletions to the four databases are specified by formulas like $add(K_f, size(kr.tex) = 33000)$, which adds this function value to the K_f database. We assume that *add* and *delete* have been configured so as to maintain the obvious consistency conditions mentioned in Theorem 2.1. For example, when we add the function value to K_f we delete any previous function values.

5 Examples

Our first example is that of opening a safe, due originally (we believe) to Moore [Moo85]. There are two actions available: *readComb* and *dialComb*. Formal descriptions of these actions are given in Table 2. We consider two different plans to see if they achieve the goal of opening the safe.

Consider the situation where the agent's initial knowledge state I is described by $K_f = \{haveComb(safe)\}$, i.e., the object "*safe*" has a combination lock. The agent might try dialing a random combination on the safe, for instance, taking the action *dialComb*(*safe*, *15-42-7*). In I it is easy to see that $\mathbf{IA}(haveComb(safe)) = \mathbf{T}$. Furthermore, $\mathbf{IA}(\textit{15-42-7}) = \mathbf{W}$ since "*15-42-7*" is a constant (step 2 of the algorithm) and all constants are known. Hence

[8]The run-time variables are positional just as in a procedure call. The user has to know what information is returned by the execution module at each position in order to properly specify the action.

Command	Precondition	Effects
$readComb(x)$	$K(haveComb(x))$	**Plan Time:** $add(K_v, combo(x))$ **Run Time:** $exec(readComb(x), !val)$ $delete(K_v, combo(x)), add(K_f, combo(x) = !val)$
$dialComb(x, y)$	$K(haveComb(x))$, $K_v(y)$	**Plan Time:** $K(y = combo(x)) \Rightarrow add(K_f, (open(x)))$ **Run Time:** $exec(dialComb(x), !safeopen)$ $!safeopen = \mathbf{True} \Rightarrow$ $add(K_f, (open(x))), add(K_f, (y = combo(x)))$

Table 2: Open Safe Domain Actions

the agent knows at plan time that the action's preconditions are satisfied.

Since the action's preconditions are satisfied, the action can be simulated[9] on I to yield an updated **DB**, I'. In this case however $I' = I$ since the action has no plan time effects on I. *dialComb* has a conditional plan time effect, but in this case **IA** cannot deduce the condition $K(y = combo(safe))$ from I and so the effect $add(K_f, open(safe))$ is not activated. Intuitively, the agent does not know if dialing a random combination will cause the safe to open.

When we execute the action from the initial state I, however, we get a different set of effects. The combination *15-42-7* is passed to the execution module along with the run time variable *!safeopen* (this is the $exec(dialComb(x, y), !safeopen)$ component of the action where x is bound to *safe* and y is bound to *15-42-7*). The execution module will set *!safeopen* to **True** or **False** dependent on whether or not the action succeeded in opening the safe. At run time, if *!safeopen* is set to **True** by the execution module, the action's conditional effect will be activated resulting in both $open(safe)$ and $combo(safe) = $ *15-42-7* being added to K_f to create a new state I'. Intuitively, if the safe opens the agent comes to know it and also comes to know that the combination dialed was in fact the right combination. So we see that the act of dialing a arbitrary combination does not allow the agent to conclude at plan time that the safe will be opened. However, at run time the agent may in fact be lucky and cause the safe to open.

Now consider the action sequence $readComb(safe)$ followed by $dialComb(safe, combo(safe))$, again from initial state I. The precondition to the first action, $readComb(safe)$, is satisfied in I. At plan time this action updates I by adding $combo(safe)$ to K_v. Intuitively, this action will cause the agent to come to know the combination of the safe. Let the updated state be I'.

In I', $K(haveComb(safe))$ holds as this fact was not deleted from K_f. Furthermore, $K_v(combo(safe))$ also holds as this term was added to K_v by the previous action. Thus, we can conclude that the preconditions of the second action $dialComb(safe, combo(safe))$ hold in I'. When we simulate the action in I' we must determine if the conditional of *dialComb*'s plan time effect holds in I'. For this action instance the conditional is $K(combo(safe) = combo(safe))$. I' has nothing in it to allow the inference algorithm to simplify these terms, but the algorithm is still able to return **T** as the two terms are syntactically identical (step **3** of the **IA** algorithm). Hence, $open(safe)$ is added to the K_f database of I'. Intuitively, the agent knows at plan time that these two actions will open the safe, even though it does not currently know what combination will be dialed.

At run time, $readComb(safe)$ has the effect of determining what the actual value of the combination is. The execution module binds this value to the run time variable $!val$. Suppose that this value is *15-42-7*. Then $combo(safe) = $ *15-42-7* will be added to K_f. In addition, the term $combo(safe)$ is deleted from K_v.[10] These changes will be made to the initial state I to yield a new state I'. Now $dialComb(safe, combo(safe))$ is executed in I'. Prior to passing information to the execution module we must reduce all terms to their simplest form using the **EvalT** algorithm. This means that the run time call to the execution module will be $exec(dialComb(safe, \textit{15-42-7}), !safeopen)$: the second argument of the action $combo(safe)$ will have been reduced to *15-42-7* by the function value added

[9]We use the term "simulated" when talking about projecting the action's effects at plan time, and "executed" when talking about projecting the action's effects at run time.

[10]This deletion is not strictly necessary. It "cleans up" K_v by removing redundant information.

Command	Effects
drink	**Plan Time:** $add(K_f, hydrated)$
medicate	**Plan Time:** $K(hydrated) \Rightarrow add(K_f, \neg infected)$ $K(\neg hydrated) \Rightarrow add(K_f, dead)$ $\neg K_w(hydrated) \Rightarrow delete(K_f, \neg dead)$ **Run Time:** $exec(medicate, !alive)$ $!alive = \textbf{False} \Rightarrow add(K_f, dead)$ $!alive = \textbf{True} \Rightarrow add(K_f, \neg infected)$
stain	**Plan Time:** $add(K_w, blue), add(K_w, infected)$ **Run Time:** $exec(stain, !stainblue)$ $delete(K_w, blue), delete(K_w, infected)$ $!stainblue = \textbf{True} \Rightarrow add(K_f, blue), add(K_f, infected)$ $!stainblue = \textbf{False} \Rightarrow add(K_f, \neg blue), add(K_f, \neg infected)$

Table 3: Medical Domain Actions

by the previous action. This reduction is important, and is the reason we need a $K_v(y)$ precondition on the *dialComb* action: the execution module cannot be expected to take complex terms whose value is unknown as arguments. If the execution module is successful it will return **True** in the run time variable !*safeopen*, which will cause *open*(*safe*) to be added to K_f in I'. The other addition is redundant as the value of *combo*(*safe*) is already in I'.

Our second example is due to Smith and Weld. Three actions are available: *drink*, *medicate*, and *stain*. The goal is to cure a patients' infection, without killing them. *drink* has the effect of hydrating the patient. *medicate* has the ability to cure the infection, but only if the patient is hydrated. Otherwise, it kills the patient. *stain* can be used to test if the patient is infected: the stain becomes blue if the patient is infected. These actions are described in Table 3. None of these actions have preconditions that need to be satisfied, so we are only concerned with their effects.

Suppose that the agent's initial knowledge state is described by $K_f = \{\neg dead\}$. One possible plan is the action sequence *drink* followed by *medicate*. *drink* has the plan time effect that the agent knows that the patient is hydrated. The second action, *medicate*, has a conditional plan time effect. Since the agent knows *hydrated*, it will also come to know $\neg$*infected*. Furthermore, $K(hydrated)$ implies $K_w(hydrated)$ so the third conditional is not activated. Hence, neither of these actions removes $\neg$*dead* from K_f, so the agent also knows the patient will be alive after these two actions. Thus, the agent is able to construct to plan that it knows will achieve its goals. Furthermore, it knows this at plan time.

Another possible plan is to perform the action *medicate* without first hydrating. Since initially the agent does not have any knowledge about hydration the third conditional effect is activated and the agent loses its knowledge that the patient is not dead. So at plan time the agent can conclude that the medicate action has an unknown effect on *dead* and hence that this plan is not safe.

Finally consider the plan *stain* followed by the conditional action if $K(infected)$ then *drink* followed by *medicate*. The action *stain* has the plan time effect of adding *infected* to K_w. In other words, the agent knows at plan time that after executing *stain* it will either be in a state where it knows *infected* or it knows $\neg$*infected*. It is not difficult to extend the planner so that at plan time it can add a conditional branch for any fact in K_w, like *infected*. Along one of the branches it adds *infected* to K_f, assuming *infected* to be true, and along the other it adds $\neg$*infected* to K_f assuming *infected* to be false. It then proceeds to complete the plan along both branches ensuring that all branches achieve the goal. At execution time the K_w fact that conditions any branch will be resolved and the plan executor will know which branch to take.

In this example, after the *stain* action one branch will start in a state where $K_f = \{\neg dead, infected\}$. In this state it is not difficult to see that the actions *drink* then *medicate* achieve the agent's goal. The other branch starts in a state where $K_f = \{\neg dead, \neg infected\}$. No additional actions are needed along this branch to achieve the agent's goal.

So we see that the agent is able to determine at plan time that the above conditional plan achieves its goal.

Command	Precondition	Effects
ls -al z	$K(\mathit{readable}(z))$	**Plan Time:**
		$\mathit{add}(K_w, \mathit{in\text{-}dir}(x,z))$
		$\mathit{add}(K_w, \mathit{in\text{-}dir}(x,z) \wedge \mathit{readable}(x))$
		$\mathit{add}(K_w, \mathit{in\text{-}dir}(x,z) \wedge \mathit{size}(x) = y)$
		Run Time:
		$\mathit{exec}(\mathit{ls\ \text{-}al\ z}, !\mathit{file}, !\mathit{readable}, !\mathit{size})$
		$\mathit{add}(K_f, \mathit{in\text{-}dir}(!\mathit{file}, z))$
		$!\mathit{readable} \Rightarrow \mathit{add}(K_f, \mathit{readable}(!\mathit{file}))$
		$\mathit{add}(K_f, \mathit{size}(!\mathit{file}) = !\mathit{size})$
		$\mathit{add}(\mathit{LCW}, \mathit{in\text{-}dir}(x,z))$
		$\mathit{add}(\mathit{LCW}, \mathit{in\text{-}dir}(x,z) \wedge \mathit{readable}(x))$
		$\mathit{add}(\mathit{LCW}, \mathit{in\text{-}dir}(x,z) \wedge \mathit{size}(x) = y)$
gzip x	$K(\mathit{readable}(x))$	**Plan Time:**
		$\mathit{delete}(K_v, \mathit{size}(x))$
		Run Time:
		$\mathit{exec}(\mathit{gzip\ x})$
		$\mathit{delete}(K_f, \mathit{size}(x)), \mathit{delete}(K_v, \mathit{size}(x))$

Table 4: UNIX Domain Actions

At run time when the *stain* action is executed, the execution module determines if the colour of the stain is blue and binds the result to the run time variable *!stainblue*. The truth value of this variable will then determine whether or not *infected* or $\neg$*infected* is added to K_f. In either case, the plan executor will have sufficient information to correctly execute the rest of the conditional plan (cf. [Lev96]).

Notice that at plan time the agent is able to guarantee that the goal of curing the infection is achieved, by considering the possible consequences of the first action and planning appropriately. But, it is not until run time that the actual branch of the plan to execute in order to achieve the goal (either medicating or doing nothing) becomes known.

We close the paper with a final example taken the UNIX domain. The actions used in the example are given in Table 4.[11] This example uses a mechanism for posting exceptions to K_w and LCW information: specifying particular instances for which a K_w or LCW formula no longer holds. This mechanism will be explained in full in a later paper.

Say that in the real world we have $\mathit{readable}(\mathit{1.ps})$, $\mathit{readable}(\mathit{2.ps})$, $\mathit{readable}(\mathit{old})$, $\mathit{in\text{-}dir}(\mathit{1.ps}, \mathit{old})$, $\mathit{size}(\mathit{1.ps}) = 10{,}000$, and $\mathit{in\text{-}dir}(\mathit{2.ps}, \mathit{new})$. The following conditional plan is intended to achieve the goal "If the file *1.ps* is in directory *old* and readable then compress it, and if *2.ps* is in directory *old* and readable compress it:" (1) *ls -al old*; (2) if $\mathit{in\text{-}dir}(\mathit{1.ps}, \mathit{old})$ and $\mathit{readable}(\mathit{1.ps})$ execute *gzip 1.ps*; (3) if $\mathit{in\text{-}dir}(\mathit{2.ps}, \mathit{old})$ and $\mathit{readable}(\mathit{2.ps})$ execute *gzip 2.ps*.

[11]We have simplified these UNIX actions somewhat for ease of presentation.

Say that the agent's initial knowledge state is $K_f = \{\mathit{readable}(\mathit{1.ps}), \mathit{readable}(\mathit{2.ps}), \mathit{readable}(\mathit{old})\}$, with all of the other databases empty. Using the above action specifications we can project this conditional plan forward to determine what the agent's knowledge state would be at the various steps of the plan.

From the initial state we can conclude that the preconditions of *ls -al old* hold. Simulating this action we generate the new knowledge state where $K_w = \{\mathit{in\text{-}dir}(x, \mathit{old}), \mathit{in\text{-}dir}(x, \mathit{old}) \wedge \mathit{readable}(x), \mathit{in\text{-}dir}(x, \mathit{old}) \wedge \mathit{size}(x) = y\}$, and everything else is unaffected. From this knowledge state we have that $K_w(\mathit{in\text{-}dir}(\mathit{1.ps}, \mathit{old}))$, and $K(\mathit{readable}(\mathit{1.ps}))$. This entails that we know whether the branch condition of step 2 at this point in the plan, and hence the branch is legitimate.

Along the false branch we can conclude that $K(\neg(\mathit{in\text{-}dir}(\mathit{1.ps}, \mathit{old}))$ and $K(\mathit{readable}(\mathit{1.ps}))$, which is sufficient to show that the first goal is achieved on this branch. Along the true branch, K_f still contains $\mathit{readable}(\mathit{1.ps})$ which is sufficient to conclude that the preconditions of *gzip 1.ps* hold.

After simulating this action we obtain a new K_w in which the entry $\mathit{in\text{-}dir}(x, \mathit{old}) \wedge \mathit{size}(x) = y$ has been replaced by the entry $\mathit{in\text{-}dir}(x, \mathit{old}) \wedge \mathit{size}(x) = y \wedge (x \neq \mathit{1.ps})$ to reflect the fact that we no longer know the value of $\mathit{size}(\mathit{1.ps})$. The mechanism that handles this update is part of an extension we have developed to deal with exceptions to K_w (and LCW) facts. This mechanism recognizes that the delete specified by *gzip* , $\mathit{delete}(K_v, \mathit{size}(\mathit{1.ps}))$, should not mean the simple removal of this item from the K_v database (in this case it is not even present in K_v). Rather, in this situation K_w allows us to conclude that we know this value, and so we must also update K_w. The mechanism we have developed posts exceptions to K_w and LCW facts. This allows us to update such facts without loosing excessive amounts of information (cf. [EGW97]).

The third step of the plan can be simulated in a similar manner to show that both of its branches also succeed in achieving the second goal (irrespective of the branch we took for step 2).

Turning now to execution time, the effects of the first and second steps of the plan are fairly straightforward. It is the third step that is interesting. At this stage of execution we would have executed the true branch of step 2 and would have $K_f = \{\mathit{readable}(\mathit{1.ps}), \mathit{readable}(\mathit{2.ps}), \mathit{readable}(\mathit{old}), \mathit{in\text{-}dir}(\mathit{1.ps}, \mathit{old})\}$. At execution time a size fact for *1.ps* would have been added by step 1, but deleted by the execution of *gzip* . There are no facts in K_f about the file *2.ps* as it was not found to be in the listed directory, but we will have that $\mathit{in\text{-}dir}(x, \mathit{old}) \in LCW$. Now the inference algorithm can infer that $K(\neg(\mathit{in\text{-}dir}(\mathit{2.ps}, \mathit{old})))$, and the execution module can correctly realize that it should execute the false (null) branch of step 3's conditional.

References

[BK96] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani, editors, *New Directions in Planning*, pages 141–153. IOS Press, 1996.

[EGW94] O. Etzioni, K. Golden, and D. Weld. Tractable closed-world reasoning with updates. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 178–189, 1994.

[EGW97] O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 1997. To appear, preprint available at ftp.cs.washington.edu.

[FHMV95] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge.* MIT Press, Cambridge, Mass., 1995.

[Gar77] J. W. Garson. Quantification in modal logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Vol. II*, pages 249–307. Reidel, Dordrecht, Netherlands, 1977.

[GEW94] K. Golden, O. Etzioni, and D. Weld. Omnipotence without omniscience: Efficient sensor management in planning. In *Proceedings of the AAAI National Conference*, pages 1048–1054, 1994.

[GW96] K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 174–185, 1996.

[Hin75] J. Hintikka. Impossible possible worlds vindicated. *Journal of Philosophical Logic*, 4:475–484, 1975.

[Lev96] H. Levesque. What is planning in the presense of sensing? In *Proceedings of the AAAI National Conference*, pages 1139–1146, 1996.

[McD96] D. McDermott. A heuristic estimator for means-end analysis in planning. In *Proceedings of the Third International Conference on A.I. Planning Systems*, 1996.

[Moo85] R. C. Moore. A formal theory of knowledge and action. In J. Hobbs and R. C. Moore, editors, *Formal Theories of the Commonsense World*, pages 319–358. Ablex Publishing Corp., Norwood, NJ, 1985.

[PG93] L. Pryor and Collins G. Cassandra: Planning for contingencies. Technical Report 41, Northwestern University, The Institute for the Learning Sciences, June 1993.

[PS92] M. Peot and D. Smith. Conditional nonlinear planning. In *Proceedings of the First International Conference on A.I. Planning Systems*, pages 189–197, 1992.

[SL93] R. B. Scherl and H. J. Levesque. The frame problem and knowledge-producing actions. In *Proceedings of the AAAI National Conference*, pages 689–695, 1993.