Appears in KR-92

An Approach to Planning with Incomplete Information

Oren Etzioni, Steve Hanks, Daniel Weld,
Denise Draper, Neal Lesh, Mike Williamson*
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

Classical planners presuppose complete and correct information about the world. This paper provides the syntax and semantics for UWL, a representation for goals and actions that facilitates planning with incomplete information about the world's state. While the expressive power of UWL is limited compared to previous work on logics of knowledge and belief, UWL has the advantage of being easily incorporated into planning algorithms. We describe a provably correct planning algorithm based on UWL. To demonstrate UWL's expressive power we encode a subset of the UNIX[1] domain (planning to achieve UNIX goals, using UNIX shell commands as primitive actions), which is difficult to capture using existing planning languages.

1 Introduction and Motivation

Classical planners (*e.g.* [Chapman 1987,Fikes and Nilsson 1971]) presuppose correct and complete information about the world. Assuming correct information means that every proposition entailed by the planner's world model is in fact true in the world. The converse of this assumption is the assumption of complete information: every proposition that is true in the world is entailed by the planner's world model. This paper provides the syntax and semantics of UWL, a representation for goals and actions that does not assume complete information on the part of the planner.[2] A planner might have incomplete information about the world state, its own actions, or exogenous events (*e.g.* the actions of other agents). We focus on incomplete but correct information about the world state.

This departure from classical planning raises a number of fundamental questions:

- How do we represent the goal of obtaining information? (*e.g.* "determine the color of `door-1`") Are "information goals" different from the standard goals used in classical planning?
- How do we represent actions whose primary function is to obtain information rather than to change the world's state (*e.g.* SENSE-COLOR)?
- Actions that change the world provide us with information as well. For example, one way of determining the color of a door is to paint it blue. How do we decide which action to use in order to satisfy an information goal (*e.g.* SENSE-COLOR *versus* PAINT)?
- Can we extend classical planning algorithms to allow for information goals and sensing actions?

*Authors' names are listed alphabetically within two status-oriented equivalence classes. Our research was funded in part by National Science Foundation Grants IRI-8957302 and IRI–9008670, Office of Naval Research Grants 92-J-1946, 90-J-1904, a NASA Graduate Student Researcher's Fellowship, an Office of Naval Research Graduate Fellowship, and a grant from the Xerox corporation. We thank Ernie Davis, Mark Drummond, Keiji Kanazawa, Craig Knoblock, Leora Morgenstern, Alicia Pérez, Rich Segal, and the anonymous reviewers for helpful comments on previous drafts.

[1] UNIX is a trademark of AT&T Bell Labs.

[2] UWL stands for the University of Washington Language.

1.1 A Motivating Example

Consider the problem of satisfying the goal `(color door-1 blue)`. A classical planner will check whether its current state satisfies the goal and, if necessary, create a plan that changes the state to one in which the goal is satisfied. If the planner's world model is incomplete, however, the goal may actually be satisfied in the world but this fact will not be true in the world model. Thus the planner has two options: try to change the world in order to satisfy the goal, or try to elaborate the model by *sensing* whether `(color door-1 blue)` is true in the world.

Is sensing necessary? Painting the door blue (with no sensing) seems to satisfy the goal, so why bother with sensory operations? There are a number of possible reasons: the planner may have no blue paint, or the blue paint may be needed to achieve a different goal; painting an already-painted door may have a harmful side effect (*e.g.* the door will be covered with noxious wet paint), *etc.* There is a deeper reason, though.

Suppose that the planner is told that the hidden treasure it is seeking is located behind "the blue door." Painting a door blue does not satisfy the goal of finding "the blue door"— it merely obscures the identity of the appropriate door. In general, if a planner is given a definite description that is intended to identify a *particular* object, then changing the world so that another object meets that description is a mistake. The appropriate behavior is to scan the world, leaving the relevant properties of the objects unchanged until the desired object is found.

The goal `(color door-1 blue)` is ambiguous when provided to a planner with an incomplete world model. Should the planner change the world to make this goal true, or should it sense whether it is true, leaving `door-1`'s color unchanged? UWL provides a way to represent the types of goals and actions involved.

Goals and actions of this sort frequently appear in the UNIX domain (planning to achieve UNIX goals, relying on UNIX commands as primitive actions). An agent's model of its UNIX environment is invariably incomplete, giving rise to information goals such as `(active.on neal june.cs.washington.edu)` sensory actions (*e.g.*, `finger, ls, pwd`, *etc.*), and actions that change the world state (`rlogin, mv, cd`, *etc.*) Thus, the UNIX domain provides a natural test of the expressiveness of UWL.[3]

[3] In fact, the design of UWL was originally motivated by the project of building a *softbot* (software robot) for UNIX [Etzioni and Segal 1992].

1.2 Contributions

The research contribution of the UWL language stems from the following novel features:

- UWL allows for an incomplete world model and for conditional plans.
- UWL actions may change the state of the world, the state of the agent's knowledge, or a combination of the two.
- UWL goals may include specifications of both desired states of knowledge, desired states of the world, and injunctions against changing certain aspects of the world state.
- The language is tractable; we describe an implemented planner for UWL.
- UWL is expressive enough to encode a nontrivial subset of the UNIX domain.

1.3 Organization of the Paper

Section 2 provides the syntax and semantics of UWL. Section 3 describes extensions to the SNLP algorithm [McAllester and Rosenblitt 1991,Barrett and Weld 1992] allowing it to generate plans involving sensory operations. Section 4 demonstrates the expressive power of UWL by encoding a subset of the UNIX domain. The paper concludes with a discussion of related work, the limitations of UWL, and directions for future work.

2 The Language

UWL is an extension of the STRIPS language [Fikes and Nilsson 1971]. Section 2.1 describes the syntax of UWL, Section 2.2 follows with a specification of its semantics. Section 2.3 discusses how to use the language to formalize the notion of an information goal.

2.1 Syntax

A BNF syntax for UWL appears in Table 1. The novel features of the language are: using annotations to distinguish causal effects from observational effects and goals of information from goals of satisfaction, an explicit notion of a proposition's state being neither true nor false, the explicit representation of information that is unknown at plan time but will be provided at run time by sensing actions, and conditional plan steps that exploit this run-time information.

The definitions for planning problems and for a plan that solves a planning problem are typical of STRIPS-like systems: a planning problem consists of an initial state, a goal state, and a set of operator schemas. A plan is a totally ordered set of *steps*, which are ground instances of the operators. Every plan will contain an

planning-problem	::=	`Goal:` *goals* `Initial:` *literal*$*$ `Actions:` *operator*$*$
plan	::=	*pcond* | *vcond* | *step plan* | ϵ (the empty plan)
truth-value	::=	`T`| `U`| `F`
pred	::=	a constant symbol designating a predicate name
const	::=	a constant symbol designating something in the world
var	::=	a symbol of the form "?c" designating a variable
rvar	::=	a symbol of the form "!c" designating a run-time variable
vc	::=	*var* | *const*
rvc	::=	*rvar* | *var* | *const*
vtv	::=	*var* | *truth-value*
rvtv	::=	*rvar* | *var* | *truth-value*
content	::=	(*pred vc*$*$)
rcontent	::=	(*pred rvc*$*$)
literal	::=	(*content* . *vtv*)
rliteral	::=	(*rcontent* . *rvtv*)
goals	::=	((`satisfy` *literal*) | (`hands-off` *content*) | (`find-out` *literal*))$*$
postconditions	::=	((`cause` *literal*) | (`observe` *rliteral*))$*$
operator	::=	`Name:` *name* `Preconds:` *goals* `Postconds:` *postconditions*
step	::=	an instance of an *operator* with no unbound variables
pcond	::=	(`pcond` *content* (*plan*) (*plan*))
vcond	::=	(`vcond` (*rvar* = *const*) (*plan*) (*plan*))

Table 1: BNF specification of UWL.

initial step and a goal step. The initial step's precondition set is empty and its postcondition set asserts the problem's initial state. The goal step's precondition set consists of the problem's goal state, and it has no postconditions. Defining plans in this way means that we don't need to talk about a problem's goals being satisfied; instead we talk about every step's preconditions being met. This is what we mean by a *correct* plan, and defining correctness is the focus of our discussion of a plan's semantics.

Before moving to the discussion of a plan's semantics we should give intuitive definitions for the syntactic constructs that extend STRIPS: the distinction between causal and observational postconditions, the concept of a run-time variable, the additional truth value for propositions, and the conditional plan constructs.

The first extension involves dividing an operator's effects into those that change the world (the `cause` annotation), and those that change the planner's state of information (the `observe` annotation)[4]. Causal postconditions correspond to STRIPS' adds and deletes.

Observational postconditions come in two forms, corresponding to the two ways the planner can gather information about the world at run time: it can observe the truth value of a proposition, `(observe ((P c) . !v))`, or it can identify an object that has a particular property, `(observe ((P !x) . T))`.

The variables `!v` and `!x` are *run-time* variables, used to refer to a piece of information that will not be available until execution time. We require that each `observe` postcondition contain *exactly one* run-time variable and that each `cause` postcondition contain *no* run-time variables.

Conditional plan steps use this run-time information. We introduce two conditional constructs corresponding to the two forms for `observe` postconditions: if the planner observes the truth value of a proposition `P` using a run-time variable `!v`, it can then build a plan conditional on this information using the construct (`pcond` `P` $\mathcal{P}_1$ $\mathcal{P}_2$) where the plan $\mathcal{P}_1$ will be executed if the proposition turns out to be true at execution time, and $\mathcal{P}_2$ will be executed otherwise.

The second form for `observe` postconditions binds its run-time variable to an object that satisfies a particular property. The planner can condition on *which* object the variable will be bound to using the construct (`vcond` ($!x = K$) $\mathcal{P}_1$ $\mathcal{P}_2$) where $\mathcal{P}_1$ will be executed if the execution system binds variable `!x` to constant `K`; otherwise $\mathcal{P}_2$ will be executed.

Our next extension to STRIPS involves annotating preconditions (and thus goals) with `satisfy`, `hands-off`, or `find-out`. We will discuss these annotations in more detail, but the intuition is that a `satisfy` pre-

[4]Causal postconditions imply a corresponding observation: we assume that if an operator causes `P` to become true the planner *knows* that `P` becomes true.

condition can be achieved by any means, causal or observational. The precondition `(find-out (P . T))` means roughly that the planner wants to determine that `P` is true, but does not want to change `P`'s state in doing so. (We may want to discourage the planner from finding out that a chair is blue by painting it blue, for example.)

A precondition of the form `(hands-off P)` is a different sort of constraint: it says nothing about `P`'s truth value, but demands that the plan do nothing to change `P`'s state. Section 2.3 discusses how these annotations relate to the intuitive notion of an information or knowledge goal.

Our final extension to the language extends the truth values a proposition can take on: propositions can be either true `T`, false `F`, or "unknown" `U`. Truth values of `U` apply to propositions about which the planner has incomplete information: those that are not mentioned in the initial state and which no subsequent plan step has changed.

2.2 Semantics

Our discussion of the meaning of UWL plans centers around a definition of what it means for a plan to be *correct*—informally, that it will actually achieve the goals that it was constructed to achieve. Intuitively a *correct* plan is one in which every precondition is *true*. In a totally ordered ground plan without conditionals or run-time variables, this definition is straightforward:

Definition 1 (Correctness (STRIPS version)) *A plan $\mathcal{P}$ is correct just in case every precondition of every one of its steps is true. A step $\mathcal{S}_j$'s precondition* `P` *is true just in case there is some step $\mathcal{S}_i$, $i < j$ that has* `(P . T)` *as a postcondition, and there is no intervening step $\mathcal{S}_k$, $i < k < j$ that has* `(P . F)` *as a postcondition.*

We need to extend this definition to include conditional plans, explicit truth values, and precondition and postcondition annotations.

2.2.1 Plan branches

The STRIPS definition of correctness admits only one course of execution for the plan: $\mathcal{S}_1$, $\mathcal{S}_2$, ... Introducing conditionals means that the actual course of execution may not be known at plan time. We therefore introduce the idea of a plan's *branches*. A branch through a plan is a sequence of non-conditional steps, representing one possible course of execution. Associated with a plan $\mathcal{P}$ is a set of branches, branches($\mathcal{P}$), representing *all possible* courses of execution. We define a plan's branch set using the four possible definitions of a plan from Table 1:

1. branches(ϵ) = $\emptyset$
2. branches($\mathcal{S}$ $\mathcal{P}$) =
   $\{\mathcal{S}; b \mid b \in \text{branches}(\mathcal{P})\}$
3. branches((`pcond P` $\mathcal{P}_1$ $\mathcal{P}_2$)) =
   $\{\mathcal{S}_{\mathtt{P}}; b \mid b \in \text{branches}(\mathcal{P}_1)\} \cup$
   $\{\mathcal{S}_{\overline{\mathtt{P}}}; b \mid b \in \text{branches}(\mathcal{P}_2)\}$
   where $\mathcal{S}_{\mathtt{P}}$ is a step with no preconditions and the single postcondition `(observe (P . T))` and where $\mathcal{S}_{\overline{\mathtt{P}}}$ is a step with no preconditions and the single postcondition `(observe (P . F))`.
4. branches((`vcond` (!x = K) $\mathcal{P}_1$ $\mathcal{P}_2$)) =
   $\{\mathcal{S}; b \mid b \in \text{branches}(\mathcal{P}_1)\} \cup \text{branches}(\mathcal{P}_2)$
   where $\mathcal{S}$ is a step with no preconditions and the single postcondition `(observe ((!x = K) . T))`.[5]

Now each branch of the plan is a totally ordered ground sequence of steps; we will say that a plan $\mathcal{P}$ is correct just in case every branch in branches($\mathcal{P}$) is correct.

2.2.2 Matching and truth values

We still cannot apply the STRIPS definition of correctness to a plan's branches for three reasons:

1. Preconditions and postconditions have explicit truth values: an operator can require that `P` be *false* or cause its truth value to change to *unknown*.
2. A precondition can be satisfied through the binding of a run-time variable: a step $\mathcal{S}_k$'s precondition `(P K)` may be satisfied by a prior step $\mathcal{S}_i$ with postcondition `(P !x)` and an intervening step $\mathcal{S}_j$ with postcondition `(!x = K)`.
3. Preconditions can have annotations that restrict the form of the steps that can appear in the plan. `find-out` preconditions will generally be satisfied by `observe` postconditions, for example.

We address the second complication by defining what it means for two propositions (presumably one step's precondition and another step's postcondition) to *match*:

Definition 2 (Matching) *Suppose that $\mathcal{S}_i$ has a postcondition whose propositional content is* $\mathtt{P}_1$*, and $\mathcal{S}_k$ has a precondition whose propositional content is* $\mathtt{P}_2$*, and $i < k$.* $\mathtt{P}_1$ matches $\mathtt{P}_2$ *if*

- $\mathtt{P}_1$ *and* $\mathtt{P}_2$ *are identical: the predicates are the same and in every argument place they have exactly the same constant or run-time variable, or*
- $\mathtt{P}_1$ *and* $\mathtt{P}_2$ *are identical except that* $\mathtt{P}_1$ *has a run-time variable* `!x` *in an argument place, and* $\mathtt{P}_2$ *has a constant* `K` *in the corresponding argument place, and there is a step $\mathcal{S}_j$, $i < j < k$ with a*

[5] Steps with postconditions of this form occur *only* as a result of "unfolding" conditionals.

postcondition of the form `((!x = K) . T)`*, and there is no step occurring between* j *and* k *with a postcondition of the form* `((!x = L) . T)` *for any constant* `L`*.*

We now deal with the problem of defining what it means for a plan containing explicit truth values and annotations to be correct. Recall that a precondition will be annotated with one of `satisfy`, `hands-off`, or `find-out`, and a postcondition will be annotated with `cause` or `observe`. Our precondition annotations are intended to convey the following information:

- `(satisfy (P . v))` Make `P` have truth value `v` by any means—causal, observational, or some combination.
- `(hands-off P)` Do not change the value of proposition `P`.
- `(find-out (P . v))` Ascertain whether or not `P`'s truth value is `v`. This can be accomplished in one of two ways:
  - By using a step that has a *observational* postcondition that matches `P`, or
  - By using a step that has a *causal* postcondition that matches `P`, as long as that step serves some other purpose in the plan.

The tricky part is to formalize the meaning of "serves some other purpose." To do so we start with the idea of a postcondition *supporting* a precondition. The first part of the definition is similar to the STRIPS notion of a postcondition making a precondition true.

Definition 3 (Postcond. supports precond.) *Suppose that step* i *has a postcondition with annotation* a_i*, propositional content* p_i *and truth value* t_i *and that step* k *has a precondition with annotation* a_k*, propositional content* p_k*, and truth value* t_k*. Step* i*'s postcondition* supports *step* k*'s precondition only if*

1. $i < k$*,*
2. p_i *matches* p_k *(in the sense defined above),*
3. $t_i = t_k$*, and*
4. *there is not step* j*—*$i < j < k$*—that has a postcondition whose propositional content matches* p_k*.*

There is one way in which a precondition can be supported without a supporting step i: a truth value of `U` can be satisfied if there is *no* step that affects the proposition.

Definition 4 (Precondition supported) *A step* k*'s precondition is* supported *just in case either*

1. *There is a step* i *with a postcondition that supports the precondition (as defined above), or*
2. *The precondition's truth value* t_k *is* `U` *and there is* no *step* $i < k$ *with a postcondition that matches the precondition's proposition* p_k*.*

The definitions so far do not mention the precondition's annotations, so we will define a "valid" precondition to be one that is supported in a manner that respects its annotation. We define what acceptable support is for each of the three precondition annotations.

Definition 5 (Valid precondition) *Consider a precondition of step* k *with annotation* a_k*, proposition* p_k*, and truth value* t_k[6]*.*

1. *If* $a_k =$ `satisfy` *then the precondition is valid just in case it is supported.*
2. *If* $a_k =$ `hands-off` *then the precondition is valid just in case there is no step* $i < k$ *with a postcondition with annotation* `cause` *that matches* p_k*.*
3. *If* $a_k =$ `find-out` *then the precondition is valid just in case either*
   (a) *it is supported by a postcondition whose annotation is* `observe`*, or*
   (b) *it is supported by a postcondition of some step* i *whose annotation is* `cause`*, but step* i *also has a postcondition that supports some precondition by satisfying either item 1 or item 3a.*

The idea behind the definition for `find-out` preconditions is to disallow a step appearing in a plan if its only purpose is to support `find-out` preconditions using its causal postconditions. If the step is in the plan for some other reason—either because it causes some other proposition that needs to be satisfied or because it observes some proposition that needs to be satisfied or found out—it can then validate the `find-out` precondition as well.

The definition of a correct plan follows directly from the definition of a plan's branches and the definition of a valid precondition:

Definition 6 (Correct plan) *A plan's branch is correct just in case every precondition of every step in the branch is valid. A plan is correct just in case every one of its branches is correct.*

2.3 Discussion

One of the most fundamental questions we address is what is the precise meaning of the informal notion of an "information goal." UWL provides two alternatives amenable to precise specification and simple implementation, but there are certainly many more.

[6] If $a_k =$ `hands-off` then t_k is undefined.

Our first alternative is to represent the information goal "determine that the proposition `P` is true" as the conjunction `(satisfy (P . T))` and `(hands-off P)`. This encoding can be too restrictive, in some cases, in that it disallows plans in which `P` is fortuitously achieved. Suppose that a plan contains a step $\mathcal{S}_i$ that was inserted into the plan because it achieved goal `G`, and that step also caused `P` to be true as a side effect. Defining information goals in terms of `hands-off` means that this plan does *not* satisfy the goal `(satisfy (G . T))` and `(satisfy (P . T))` and `(hands-off P)`, even though at the end of the plan it is the case that `G` is true and that `P`'s truth value is known to be `T`. The problem is that $\mathcal{S}_i$ *causes* `P` to be true, even though the step served to satisfy the other goal as well, and that violates the `hands-off` annotation.

To illustrate this sort of difficulty, consider the goal of printing the file `paper.tex`. The final step in a plan to achieve this goal is `lpr`, the UNIX command that sends a postscript file to the printer. One of the preconditions to `lpr` determines that the argument to the command is indeed a file. If we express this precondition as the conjunction `(satisfy (isa file.object ?file) . T)` and `(hands-off (isa file.object ?file))`, then we effectively disallow the creation of the postscript file at an earlier step. Yet, that is precisely what we need to do (via the commands `latex` and `dvi-ps`). A more appropriate encoding of the precondition is `(find-out (isa file.object ?file) . T)`.

In general, we can represent an information goal simply as `(find-out (P . v))`. This encoding is less restrictive since, as in the above example, `P`'s truth value can change as a side effect of achieving a different goal. Neither alternative is guaranteed to be satisfactory in all cases, and UWL is not committed to using either one exclusively. In the UNIX domain we have found that `find-out` is useful for writing operator preconditions and that the `hands-off` and `satisfy` combination is useful for expressing top-level goals (see Table 3). Developing a complete taxonomy of information goals is an area of future research.

3 Partial-Order Planning

We have worked to keep UWL close to the familiar STRIPS representation in order to build on the long-standing body of work on planning algorithms using that representation (see [Allen *et al.* 1990] for a survey). In this section we present SENS_P a provably sound partial-order planning algorithm for UWL based on SNLP [McAllester and Rosenblitt 1991,Barrett and Weld 1992,Hanks and Weld 1992]. We believe SENS_P is complete subject to one simplifying assumption.

The basic operation of SENS_P follows that of SNLP, but a number of important extensions are made to handle the features of UWL. In this paper, we focus on two issues: extending the notion of a *causal link* to planning with incomplete information, and generating conditional plans.

3.1 Causal Links

SENS_P inherits the notion of a causal link from SNLP and earlier planners (*e.g.* [Warren 1974,Tate 1977]). Causal links are used to record why a step was introduced into a plan and to prevent other steps from interfering with that purpose. If a step S_i achieves a proposition p to satisfy a precondition of step S_j, that dependency is recorded by the causal link $S_i \xrightarrow{p} S_j$.

Following McAllester, we say that a link $S_i \xrightarrow{p} S_j$ is *threatened* if some step S_k might be ordered between S_i and S_j, and S_k has a `cause` postcondition that matches p according to the plan's variable-binding constraints. A precondition p of a plan-step S_j is an *open condition* of the plan if there is no causal link $S_i \xrightarrow{p} S_j$. A plan is said to be *complete* if it has no open conditions and no threatened links. SNLP is sound, systematic, and complete (as long as backtracking explores all nondeterministic choice points using a strategy such as iterative deepening) [McAllester and Rosenblitt 1991].

The SENS_P algorithm extends the notion of a casual link to handle the UWL's annotated preconditions and postconditions. The postconditions of a UWL operator are split into a `cause` list and an `observe` list. Since an `observe` postcondition does not change that proposition's state in the world, these conditions cannot pose a threat to any causal links; thus threat detection considers only `cause` postconditions, which correspond to STRIPS add and delete lists.

3.2 Hands-off Goals

SENS_P handles goals of the form `(hands-off (P ?x))` by adding a causal link between the plan's initial step and its goal step. The link's "content" is `(P ?x)`, but the variable `?x` is interpreted as universally quantified. As a result, any step that might be added to the plan that has a `cause` postcondition `(P ?y)` (for any variable `?y` or any constant) will be considered a threat to that link, and will not be added to the plan. An `observe` postcondition of `(P ?y)`, on the other hand, is not considered a threat to the link, and can therefore be added to the plan. Thus `hands-off` preconditions are implemented trivially by exploiting standard techniques for detecting and resolving threats on links.

3.3 Find-out Goals

`Find-out` preconditions can be achieved by a step's `observe` postcondition or, when the step has some other purpose, by a `cause` postcondition. The actual algorithm to achieve this is moderately complex, but

the two basic ideas are:

1. SENS_P branches and separately considers the two ways of establishing the `find-out` goal, backtracking to ensure completeness.
2. To achieve `find-out` goals using a `cause` postcondition the algorithm first tries to add causal links to support all preconditions without `find-out` annotations (adding new steps to the plan if necessary). Then it tries to add links supporting all remaining goals without adding new steps to the plan.

3.4 Run-time Variables

For the purpose of matching preconditions and postconditions, SENS_P treats run-time variables as constants whose values are not yet known: they are not allowed to match with other constants or with different run-time variables (*cf.* [Olawsky and Gini 1990]). Run-time variables *can* match ordinary variables, but SENS_P adds ordering constraints which ensure that the value of a run-time variable is not used until it has been `observed`.

For example, suppose we wish our kitchen table and chair to share the same color. Further, suppose that while we don't know the color of the table, we don't want to change it. We can express this goal as follows:

```
Goal:
  (satisfy ((color chair ?c) . T))
  (satisfy ((color table ?c) . T))
  (handsoff (color table ?tc))
```

Let there be only three possible actions: we can obtain paint of any color, we can paint any object with any color that we have, and we can sense the color of any object.

```
Name:  (SENSE-COLOR ?obj !color)
Preconds:
Postconds: (observe ((color ?obj !color) . T))

Name:  (GET-PAINT ?color)
Preconds:
Postconds: (cause ((have-color ?color) . T))

Name:  (PAINT ?obj ?color)
Preconds: (satisfy ((have-color ?color) . T))
Postconds: (cause ((color ?obj ?color) . T))
```

SENS_P will immediately construct a causal link $S_0 \overset{C}{\rightarrow} \underline{\text{table}} S_\infty$ to enforce the `hands-off` goal.[7] Next, SENS_P explores the two ways to satisfy the goal that the table have color `?c`, i.e. by introducing either a PAINT or SENSE-COLOR operation. If SENS_P decides to paint the table, it will quickly realize the threat to the `hands-off` link and backtrack. Thus a `(sense-color table !color)` step will be added (which we shall denote as step S_s) along with the causal link $S_s \overset{C}{\rightarrow} \underline{\text{table}} S_\infty$. Since S_s has only `observe` postconditions, it does not threaten the earlier link.

Next, SENS_P might try to achieve the fact that the chair has color `?c`. Since the variable `?c` is also mentioned in the goal proposition, `(color table ?c)`, which has been achieved, SENS_P has recorded that `?c` must codesignate with the run-time variable `!color` which is bound by the SENSE-COLOR step. In other words, SENS_P needs to make the chair have the color corresponding to the value of `!color`. As before, there are two possibilities, PAINT and SENSE-COLOR. If SENS_P attempts to use SENSE-COLOR, it will realize that it must construct a conditional plan—we cover this case in the next section, so for now, assume that SENS_P chooses to paint the chair. Thus, SENS_P adds a PAINT step (S_p) and the causal link $S_p \overset{C}{\rightarrow} \underline{\text{chair}} S_\infty$. Since this PAINT step uses the run-time variable, SENS_P constrains $S_s < S_p$.

At this point all the original goals have been supported, but the `paint` step has an unsatisfied precondition, so SENS_P needs to achieve `(have-color !color)`. The only way to do this is with a GET-PAINT step, so GET-PAINT is added to the plan as step S_g along with the causal link $S_g \overset{HC}{\rightarrow} S_p$. Since this GET-PAINT step also uses the run-time variable, SENS_P constrains $S_s < S_g$. The final plan is thus:[8]

```
(sense-color table !color)
(get-paint !color)
(paint chair !color)
```

3.5 Generating Conditional Branches

SENS_P uses a variant of Warren's WARPLAN-C technique for generating conditional plans [Warren 1976]. The basic idea is that conditionals are inserted into the plan only when SENS_P needs to constrain the value of a run-time variable. The algorithm must be careful as to how it makes the constraint, however, since it has to obey the requirement that a run-time variable be `observed` before it is used. The algorithm first chooses one value `k` for the run-time variable `!v`, generates a conditional step `(vcond (!v = k) ...)`, then continues planning. It later generates a plan for the other branch (without the equality constraint), then combines the two.[9]

[7] Clarity demands abbreviations in this example; *e.g.*, C_{table} denotes the proposition `(color table ...)`, *etc.*

[8] For brevity, the steps in a plan are indicated by their names only.

[9] Since writing this paper, we have learned of independent work on the synthesis of conditional plans which could

For example, consider a small modification to example above: instead of a completely general GET-PAINT operator, suppose that getting green paint requires a distinct operator, MAKE-GREEN-PAINT, which mixes blue paint and yellow paint to create green. The GET-PAINT operator suffices to get all other colors.[10]

```
Name:  (MAKE-GREEN-PAINT)
Preconds:
  (satisfy ((have-color blue) . T))
  (satisfy ((have-color yellow) . T))
Postconds: (cause ((have-color green) . T))

Name:  (GET-PAINT ?c)
Preconds:
Postconds: (cause ((have-color ?c) . T))
Equals: (<> ?c green)
```

As before, SENS_P will construct the `hands-off` link, and add a sensing operation to determine the color of the table, and as before, let us assume that it adds a PAINT step to paint the table. Now, however, when it attempts to support the precondition `(have-color !color)` for the paint, it will attempt to constrain the run-time variable !color with the constant green. Thus, it will create two copies of the plan, one in which !color is constrained to equal green, and one in which it is constrained not to equal green. The two plans will be independently completed in a manner analogous to the first example, and the merged plan is created by extracting the sensing operation and adding the conditional step (Table 2).

It is tricky to ensure soundness *and* completeness for an algorithm that generates conditional plans. So far we are confident of completeness subject to an important simplifying assumption: that the operator schemata for sensing actions (those with `observe` postconditions) have no preconditions.[11] At this point producing correct plans requires us to have operators, like GET-PAINT, which will work for all cases not covered by more specific operators. In future work, we will consider ways of weakening this restriction, either by producing 'conditionally correct' plans, or by declaring ranges on the possible values of run-time variables.

4 UWL and the UNIX Domain

In [Etzioni and Segal 1992] we describe the project of building *softbots* (*soft*ware ro*bots*): programs that interact with software environments by issuing commands and interpreting the environments' response. We believe that software environments such as operating systems or databases are a pragmatically convenient yet intellectually challenging substrate for AI research. To support this claim, we have developed a softbot for the UNIX operating system that uses UWL to represent its actions (UNIX commands such as `finger` or `cd`) and goals. The softbot generates and executes plans to achieve the UWL goals it receives as input. The design and development of UWL validates our claim that softbots provide fertile testbeds for AI research. In addition, the ability to encode UNIX commands and goals in UWL demonstrates the language's expressive power (see Table 3 for an illustration). We have represented over twenty UNIX commands as UWL operators, and are in the process of encoding many more.

Models of some UNIX commands (*e.g.*, `cd, rm`) can actually be encoded as plain STRIPS operators. However, due to the dynamic nature and sheer size of the UNIX environment, information about it is necessarily incomplete. For example, users continually log in and out, and the number of files accessible through the Internet is staggering. Consequently, many of the most routine UNIX commands (*e.g.*, `ls, pwd, finger, lpq, grep`) are used to gather information. Such commands cannot be represented by STRIPS operators but are naturally encoded in UWL (see Table 3).

Information goals arise frequently in the UNIX domain. Consider, for example, the goal of removing a file named `core-dump`. There are several ways to satisfy this goal. One is to find a file named `core-dump`, using `ls`, and remove it. Another is to rename an arbitrary file to `core-dump`, using `mv`, and remove that file. The first alternative is obviously the one intended. UWL forces the correct interpretation by representing the goal as:

(and (`find-out` ((file.name !file core-dump) . `T`))
(`satisfy` ((file.object !file) . `F`)).

The `observe` postcondition of `ls` satisfies the `find-out` goal, whereas the `cause` postcondition of `mv` does not. This goal will only be satisfied when some file named `core-dump` exists. Naturally, stipulating that *all* `core-dump` files should be removed would require universal quantification.

While UWL has turned out to be a convenient language for our softbot, it does not address a number of issues that arise in representing UNIX commands including the need for universally quantified preconditions and postconditions, and the fact that other agents (particularly humans) are continually changing the world's state by logging in and out, creating new files, *etc.* We are currently extending UWL to include universal quantification. We plan to address the dynamic nature of the UNIX domain by allowing the softbot to detect

improve our algorithm; see the discussion of [Peot and Smith 1992] in section 5.

[10] The 'Equals' slot in the last operator description is an extension to UWL that allows codesignation and noncodesignation constraints on variables to be asserted.

[11] This assumption is powerful because it frees SENS_P from subgoaling to achieve observation steps. Subgoaling complicates completeness because the sensing actions can interfere with the steps to be taken after the conditional.

Plan P_T

```
(get-paint yellow)
(get-paint blue)
(sense-color table !color)
(make-green-paint)
(paint chair green)
```

Plan P_F

```
(sense-color table !color)
(get-paint ?c)
(paint chair ?c)
```

Final Plan

```
(sense-color table !color)
(vcond (!color = green)
  (  (get-paint yellow)
     (get-paint blue)
     (make-green-paint)
     (paint chair green))
  (  (get-paint !color)
     (paint chair !color)))
```

Table 2: Subplans for both branches and the final, conditional plan.

and update its incorrect beliefs about the world. In addition, we are developing learning algorithms that will enable the softbot to model the variability of its world over time. Based on this learned model, some of the softbot's observations will persist (*e.g.*, a new workstation has been added to the network) whereas others (*e.g.*, the printer is out of paper) will be forgotten quickly. See [Etzioni *et al.* 1992] for a more comprehensive discussion.

5 Related Work

[McCarthy and Hayes 1969] argues for a formalization of the notion of "knowledge" as the basis for a theory of plans. This approach has led to the development of a rich body of logical work within AI [Morgenstern 1988,Moore 1985].[12] [Moore 1985] introduces a first-order modal logic of knowledge and action, developing the idea of *informative* actions that supply an agent with additional information about the world. [Morgenstern 1987] develops a more expressive theory of action and planning, and specifically addresses the problem of *knowledge preconditions* for the performance of actions and plans. She considers the expression of complex plans using sequential, conditional, iterative, and concurrent constructs, and axiomatizes their knowledge preconditions. She does not, however, address the problem of how these plans might be generated.

[Drummond 1986] presents a framework (plan nets) that allows for the representation of sensory actions. The occurrence of an *event* entails a set of *beliefs*, which is partitioned into two subsets. The *external-results* describe the external, physical effects of the event, while the *internal-results* describe those changes that affect only the agent's world model. However, the precondition and goal languages presented do not provide any way of distinguishing between internal and external results, so the distinction between sensory and non-sensory actions is of limited use. Again, the problem of plan generation is not considered.

[Drummond 1989] presents another version of the plan-net formalism. This version does not make the distinction between sensory and non-sensory effects, but it does provide a rich language for expressing goals. A goal may be any arbitrary boolean combination of propositions, and may also include the meta-predicates *maint(p)* (p must be true throughout the plan), *ach(p)* (p must become true at some point during the plan), and *obt(p)* (p must become true at some point and remain true until the end of the plan). The language is used only for the specification of goals. Although the top-level goals can be annotated, operator preconditions are restricted to be conjunctions of simple propositions.

Our work is close to that of [Olawsky and Gini 1990], which specifically considers the problem of planning with an incomplete initial world-state description. Their approach uses deferred planning: when a proposition's truth value is needed but unknown they suspend the planning process, execute a plan to learn its value, then resume planning. Their action representation is essentially like STRIPS. Although they discuss sensory actions, there is no distinction in their representation between sensory and non-sensory actions. The examples they give are greatly simplified by the fact that their domain contains a single operator that achieves every proposition, which allows them to avoid the complex issues that arise when some proposition is a sensory effect of one action and a causal effect of another.

[Peot and Smith 1992] presents a variant of SNLP for the construction of conditional non-linear plans. Their action representation also uses a three-valued logic, and allows actions to have multiple, mutually-exclusive sets of outcomes. However, they do not provide for a distinction between observational and causal effects of actions, or treat sensing explicitly. It is possible that their planning algorithm could be extended to work with UWL.

Universal plans [Schoppers 1987] and Gapps [Kaelbling 1988] provide methods for constructing exhaustive conditional plans. However, the representation languages employed do not allow explicit description of sensing actions. Rather, it is assumed that complete sensory information is available at every point during the execution of the plan in order to select the correct

[12] Logics of knowledge also appear in the philosophical literature (*e.g.*, [Hintikka 1962]).

UNIX goals:

Determine if neal is on june.cs.washington.edu:

```
Goal:      (find-out ((active.on neal june.cs.washington.edu) .  T))
```

Determine if the file with name "paper.tex" contains the word "theorem:"

```
Goal:      (satisfy ((name ?somefile paper.tex) .  T))
           (hands-off (name ?somefile paper.tex))
           (satisfy ((file.contains.string ?somefile theorem) .  T))
           (hands-off (file.contains.string ?somefile theorem))
```

UNIX operators:

```
Name:      FINGER
Preconds:  (find-out ((isa machine ?machine) .  T))
           (find-out ((isa person ?person) .  T))
Postconds: (observe ((active.on ?person ?machine) !boolean))

Name:      MV
Preconds:  (find-out ((isa file.object ?file) .  T))
           (find-out ((isa directory.object ?dir1) .  T))
           (find-out ((isa directory.object ?dir2) .  T))
           (find-out ((name ?file ?name) .  T))
           (find-out ((parent.directory ?file ?dir1) .  T))
           (satisfy ((protection ?dir1 readable)) .  T)
           (satisfy ((protection ?dir2 writable)).  T)
           (satisfy ((current.directory softbot ?file ?dir1) .  T))
Postconds: (cause ((parent.directory ?file ?dir1) .  F))
           (cause ((parent.directory ?file ?dir2) .  T))

Name:      GREP
Preconds:  (find-out ((isa file.object ?file) .  T))
           (find-out ((isa directory.object ?dir) .  T))
           (find-out ((name ?file ?name) .  T))
           (find-out ((parent.directory ?file ?dir) .  T))
           (satisfy ((protection ?file readable) .  T))
           (satisfy ((current.directory softbot ?dir) .  T))
Postconds: (observe ((file.contains.string ?file ?string) !boolean))

Name:      WC
Preconds:  (find-out ((isa file.object ?file) .  T))
           (find-out ((isa directory.object ?dir) .  T))
           (find-out ((name ?file ?name) .  T))
           (find-out ((parent.directory ?file ?dir) .  T))
           (satisfy ((protection ?file readable) .  T))
           (satisfy ((current.directory softbot ?dir) .  T)) ))
Postconds: (observe ((character.count ?file !char) .  T))
           (observe ((word.count ?file !word) .  T))
           (observe ((line.count ?file !line) .  T))
```

Table 3: Sample representations of UNIX goals and operators.

action.

[Ram and Hunter 1992] discusses the application of *knowledge goals* as a means of controlling inference. They develop a theoretical framework for describing the explicit desire for knowledge, and illustrate it on examples from natural language understanding and machine learning.

6 Conclusion

This paper contains two fundamental observations. First, we showed that information goals and information-gathering actions can be represented in a simple and elegant manner by annotating the preconditions and postconditions to standard STRIPS operators. The proximity of UWL to the STRIPS language enabled us to extend the SNLP planning algorithm to one that generates correct plans in the presence of incomplete information. In addition, we have developed a softbot that relies on UWL to represent a subset of the UNIX domain (see Table 3 for an illustration). The softbot generates and executes plans to achieve a wide range of UNIX goals, demonstrating the utility and expressiveness of UWL [Etzioni *et al.* 1992]. Second, we showed that posing an information goal to a planner implicitly dictates that the state of the proposition in question should be protected in the process of planning to achieve that goal. This observation is critical to choosing actions appropriately in domains (such as the UNIX domain) that contain both information-gathering actions and actions that change the world's state. We discussed two ways to encode information goals in UWL; future work will explore other ways to do so.

References

[Allen *et al.* 1990] J. Allen, J. Hendler, and A. Tate, editors. *Readings in Planning*. Morgan Kaufmann, San Mateo, CA, August 1990.

[Barrett and Weld 1992] A. Barrett and D. Weld. Partial Order Planning: Evaluating Possible Efficiency Gains. Technical Report 92-05-01, University of Washington, Department of Computer Science and Engineering, July 1992.

[Chapman 1987] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32(3):333–377, July 1987.

[Drummond 1986] M. Drummond. A representation of action and belief for automatic planning systems. In *Proceedings of the 1986 workshop on Reasoning about Actions and Plans* , San Mateo, CA, 1986. Morgan Kaufmann.

[Drummond 1989] M. Drummond. Situated Control Rules. In *Proceedings of the First International Conference on Knowledge Representation and Reasoning*, May 1989.

[Etzioni and Segal 1992] Oren Etzioni and Richard Segal. Softbots as testbeds for machine learning. In *Working Notes of the AAAI Spring Symposium on Knowledge Assimilation*, Menlo Park, CA, 1992. AAAI Press.

[Etzioni *et al.* 1992] Oren Etzioni, Neal Lesh, and Richard Segal. Building softbots for UNIX. In preparation, 1992.

[Fikes and Nilsson 1971] R. Fikes and N. Nilsson. STRIPS: A new Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3/4), 1971.

[Hanks and Weld 1992] Steven Hanks and Daniel Weld. Systematic adaptation for case-based planning. In *Proceedings of the First International Conference on AI Planning Systems*, June 1992.

[Hintikka 1962] Jaako Hintikka. *Semantics for Propositional Attitudes*. Cornell University Press, 1962.

[Kaelbling 1988] Leslie Pack Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*. Morgan Kaufmann, 1988.

[McAllester and Rosenblitt 1991] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. In *Proceedings of AAAI-91*, pages 634–639, July 1991.

[McCarthy and Hayes 1969] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[Moore 1985] R.C. Moore. A Formal Theory of Knowledge and Action. In *Formal Theories of the Commonsense World*. Ablex, 1985.

[Morgenstern 1987] Leora Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of IJCAI-87*, 1987.

[Morgenstern 1988] Leora Morgenstern. *Foundations of a Logic of Knowledge, Action, and Communication*. PhD thesis, New York University, 1988.

[Olawsky and Gini 1990] D. Olawsky and M. Gini. Deferred planning and sensor use. In *Proceedings, DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*. Morgan Kaufmann, 1990.

[Peot and Smith 1992] Mark A. Peot and David E. Smith. Conditional nonlinear planning. In *Proceedings of the First International Conference on AI Planning Systems*, June 1992.

[Ram and Hunter 1992] Ashwin Ram and Lawrence Hunter. The use of explicit goals for knowledge to guide inference and learning. Git-cc-92/04, Georgia Institute of Technology, 1992.

[Schoppers 1987] M. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proceedings of IJCAI-87*, pages 1039–1046, August 1987.

[Tate 1977] A. Tate. Generating Project Networks. In *Proceedings of IJCAI-77*, pages 888–893, 1977.

[Warren 1974] D. Warren. WARPLAN: A system for generating plans. Memo No. 76, Univerity of Edinburgh, Department of Computational Logic, 1974.

[Warren 1976] D. Warren. Generating copnditional plans and programs. In *Proceedings of AISB Summer Conference*, pages 344–354, University of Edinburgh, 1976.