Planning with Execution and Incomplete Information

Keith Golden, Oren Etzioni and Daniel Weld

Technical Report UW-CSE-96-01-09

Department of Computer Science and Engineering
University of Washington

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350

Planning with Execution and Incomplete Information

Keith Golden Oren Etzioni Daniel Weld*
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{kgolden, etzioni, weld}@cs.washington.edu

April 16, 1996

Abstract

We are motivated by the problem of building agents that interact in complex real-world domains, such as UNIX and the Internet. Such agents must be able to exploit complete information when possible, yet cope with incomplete information when necessary. They need to distinguish actions that return information from those that change the world, and know when each type of action is appropriate. They must also be able to plan to obtain information needed for further planning. They should be able to represent and exploit the richness of their domains, including universally quantified causal (*e.g.*, UNIX `chmod *`) and observational (*e.g.*, `ls`) effects, which are ubiquitous in real-world domains such as the Internet.

The XII planner solves the problems listed above by extending classical planner representations and algorithms to deal with incomplete information. XII represents and reasons about local closed world information, information preconditions and postconditions and universally quantified observational effects. Additionally, it interleaves planning with execution and handles universally quantified preconditions in the presence of incomplete information. We present XIIL(the XII action language) and the XII planning algorithm, which are the major contributions of this paper.

*We thank Denise Draper, Steve Hanks, Marc Friedman, Terrance Goan, Nick Kushmerick, Cody Kwok, Neal Lesh, Rich Segal, and Mike Williamson for helpful discussions. This research was funded in part by Office of Naval Research Grants 90-J-1904 and 92-J-1946, and by National Science Foundation Grants IRI-8957302, IRI-9211045, and IRI-9357772. Golden is supported in part by a UniForum Research Award and by a Microsoft Graduate Fellowship.

Contents

1 Introduction

Classical planners such as NONLIN [45], TWEAK [5], or UCPOP [38, 47] presuppose correct and complete information about the world. Having complete information facilitates planning since the planning agent need not obtain information from the external world — all relevant information is present in the agent's world model (this is the infamous *closed world assumption* [42]). However, in many cases, an agent does not have complete information about its world. For instance, a robot may not know the size of a bolt or the location of an essential tool [35]. Similarly, a software agent, such as the Internet Softbot [15] cannot be familiar with the contents of *all* the bulletin boards, FTP sites, and files accessible through the Internet.[1] An agent 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 concerning representational abilities and planning algorithms:

- How should one represent actions whose primary function is to obtain information rather than to change the world's state (*e.g.* SENSE-COLOR or the UNIX command `ls`)? What about goals of obtaining information? (*e.g.*, "where is my wrench?" or "do I own a file called `core-dump`?") Are "information goals" different from the standard goals used in classical planning?

Section 2 of this paper answers these questions by introducing an expressive language for representing causal and sensing actions as well as goals that combine information-gathering aspects with state-attainment. We demonstrate the expressive power of this representation language with examples from the UNIX domain.

- How does the presence of incomplete information impact planning algorithms? Can a planner satisfy universally quantified goals in the absence of complete information? Goals of the form "Move all bolts into the drawer" or "Make all files in `/tex` write-protected" are common in real-world domains. Classical planners such as PRODIGY [31] or UCPOP [38] reduce a universally quantified goal to the set of ground instances of the goal, and satisfy each instance in turn. But how can a planner compute this set in the absence of complete information? How can the planner be certain that it has moved *all* the bolts or protected *all* the relevant files?

Section 3 of this paper describes the fully-implemented XII planner which explicitly represents and reasons about information goals, sensory actions, and incomplete information about the world's state. XII uses a variety of techniques to handle universally quantified goals in the presence of incomplete information.

- What is the right balance between planning an execution? Planning for all contingencies may involve much extra work, but interleaving planning with execution forces the planner to adjust to a dynamically changing information state.

[1] Because our work is motivated by the Softbot, most of our examples are drawn from software domains. However, we emphasize that our results are general and corresponding examples are easily found in physical domains as well.

Section 4 explains a how the choice to execute an action can be treated as a search problem, and describes how to backtrack over the decision to execute.

While XII addresses a number of issues related to incomplete information, there are many issues that it does not address – including noisy sensors, continuous change, and more. These are important issues, which we ignore in our focus on incomplete information. Fortunately, this focus doesn't restrict us to working on toy problems. XII is operational in a real world domain where we can demonstrate its capabilities with concrete examples and assess its limitations. XII is at the core of Internet Softbot. The Softbot is a software agent that uses a Unix shell and the World Wide Web to interact with a wide range of Internet facilities. The inputs to the Softbot include action models of commands such as `ftp`, `telnet`, `mail` and information gathering commands such as `archie`, `gopher`, `netfind`, and many more. The Softbot relies on XII to dynamically generate plans, or sequences of these commands, to satisfy user goals. XII not only generates the commands, but also executes them, backtracking from one facility to another based on information collected at runtime.

Our focus on the Internet domain enables us to buy out of some questions, but forces us to address others. For example, sensors in the Internet domain are relatively noise-free; thus, we do not concern ourselves with that issue. However, redundant sensing is an extremely important issue, and so we have developed a solution to this problem [13, 12].

XII is an extension of UCPOP [38, 3], a partial-order planner that handles universal quantification and conditional effects. In addition to the expressiveness of UCPOP, XII supports the UWL [14] language for representing sensory actions and information goals. The combined action language, XIIL, is described in Section 2. Section 3 describes the partial-order planning algorithm used by XII, while Sections 3.3 and 3.4 detail our extensions relating to handling open goals and resolving threats. Section 4 describes how we interleaving planning with execution. In Section 5, we summarize our results and discuss related work. Appendices A and B present our representation and algorithms in more detail.

2 Representing Sensing Actions & Information-Gathering Goals

In real-world domains, it is impossible to have complete information about the entire world. For example, the Internet is just too vast for a Softbot to know everything out there, and even if it did, there would be no way to keep up to date. This presents the twin challenges of representing incomplete information about the world, and representing the actions required to obtain more information.

We present an action representation language, XIIL, that confronts both of these challenges. To represent incomplete information, XIIL supports an explicit notion of a proposition's state being unknown. It also supports a limited form of circumscription, called Local Closed World information (see Section 2.1) for circumstances in which limited instances of complete information are available. To represent the combination of sensing and causation, XIIL uses annotations to distinguish causal effects from observational effects and goals of information from goals of satisfaction. It also explicitly represents information that is unknown at plan time but will be provided at run time by sensing actions (see Section 2.2).

XIIL melds elements of both ADL [37] and UWL [14]. ADL gives us logical expressiveness including universal quantification, conditional effects, disjunction and negation. UWL provides a means of representing information goals and information-gathering actions. As explained below,

the combination is synergistic; for example, universally quantified observational actions can return an unbounded amount of information.

XIIL adopts a three valued logic: propositions can be either true `T`, false `F`, or "unknown" `U`. Any proposition that cannot be inferred to be `T` or `F`, defaults to `U`. However, it is sometimes possible to infer that a ground literal is false, even when neither it nor it's negation appears explicitly in the agent's model (See Section 2.1).

The XIIL language allows one to describe partially specified *action schemata* (also known as operators) in terms of *parameters*, *preconditions* and *effects*. Appendix A provides a complete specification of the language. Note that nested universal and existential quantification is allowed (with some restrictions) in both the effects and preconditions of schemata and in goals. Conditional effects and disjunctive goals are also supported, since algorithms for planning with these constructs have become well understood [47].

2.1 Local Closed World Information

Our agent's model of the world is represented as a set of ground literals stored in a database $\mathcal{D}_{\mathbf{M}}$. Since $\mathcal{D}_{\mathbf{M}}$ is incomplete, the closed world assumption is invalid — the agent cannot automatically infer that any sentence absent from $\mathcal{D}_{\mathbf{M}}$ is false. Thus, the agent is forced to represent false facts explicitly — as $\mathcal{D}_{\mathbf{M}}$ sentences with the truth value `F`.

In practice, many sensing actions return exhaustive information which warrants limited or "local" closed world information. For example, the UNIX `ls -a` command lists *all* files in a given directory. After executing `ls -a`, in directory `mydir`, it is not enough for the agent to record that `paper.tex` and `proofs.tex` are in `mydir` because, in addition, the agent knows that *no other* files are in that directory. Note that the agent is not making a closed world *assumption*. Rather, the agent has executed an action that yields closed world *information*.

Although the agent now knows that `parent.dir(foo, mydir)` is false, it is impractical for the agent to store this information explicitly in $\mathcal{D}_{\mathbf{M}}$, since there is an infinite number of such sentences. Instead, the agent represents closed world information explicitly in a meta-level database, $\mathcal{D}_{\mathbf{C}}$, containing formulas of the form LCW(Φ) that record *where* the agent has closed world information. LCW(Φ) means that for all variable substitutions θ, if the ground sentence $\Phi\theta$ is true in the world then $\Phi\theta$ is represented in $\mathcal{D}_{\mathbf{M}}$. For instance, we represent the fact that $\mathcal{D}_{\mathbf{M}}$ contains all the files in `mydir` with LCW(`parent.dir(`f`,mydir)`) and that it contains the length of all such files with LCW(`parent.dir(`f`,mydir)`$\wedge$`length(`f`,`l`)`).

When asked whether an atomic sentence Φ is true, the agent first checks to see if Φ is in $\mathcal{D}_{\mathbf{M}}$. If it is, then the agent returns the truth value (`T` or `F`) associated with the sentence. However, if $\Phi \notin \mathcal{D}_{\mathbf{M}}$ then Φ could be either `F` or `U` (unknown). To resolve this ambiguity, the agent checks whether $\mathcal{D}_{\mathbf{C}}$ entails LCW(Φ). If so, Φ is `F`, otherwise it is `U`.

2.2 Observational Effects

We divide the primitive effects of action schemata into those that change the world (annotated by **cause**), and those that merely change the agent's *model* of the world (annotated by **observe**). Causational effects should be familiar to most readers since classical planners (such as TWEAK) have only this type of effect. In XII, we assume that if execution of an action causes `P` to become

true,[2] then the agent knows this fact. However, **cause** effects can also result in information loss. For instance, compressing a file, makes the length of the file unknown: **cause** `(length (`*f*`, `*l*`), U)`.

Observational postconditions come in two forms, corresponding to the two ways the planner can gather information about the world at run time. One, it can observe the truth value of a proposition, **observe** (P (c), *!tv*). For example, the effect of `ping`, the UNIX command for checking whether a machine is accessible, is represented as follows:

observe `(machine.alive (`*machine*`), `*!tv*`)`

A symbol starting with an exclamation point, such as *!t*v, is a *run-time* variable, used to refer to a piece of information that will determined by executing the action and returned to the agent by the execution system at that time. Variables that are not run-time variables are referred to as plan-time variables. The other form of observational effect identifies an object that has a particular property: **observe** `(P (`*!val*`))`. For example, the effects of the UNIX command `wc` are represented as follows:

observe `(character.count (`*file*`, `*!char*`))` $\wedge$
observe `(word.count (`*file*`, `*!word*`))` $\wedge$
observe `(line.count (`*file*`, `*!line*`))`

Some actions are best modeled with universally quantified effects. For example, `chmod +r *` adds read permission to all files in the current directory; supposing that *d* is bound to the current directory, then the effect can be encoded in XIIL as:

$\forall$ `(file (`*f*`)` $\in$ `parent.directory (`*f*`, `*d*`))`
  cause `(readable (`*f*`))`

The notation provides a convenient way to specify a *universe of discourse* (*i.e.*, the set of objects being quantified over [17, p. 10]) based on the extension of a predicate in the world. For example, `parent.directory (`*f*`, /bin)` means the set of files actually in directory `/bin`. The universe of discourse may be specified by a conjunction of predicates, but disjunction is not allowed. Section 3.3.5 explains how an agent can reason about the effects of universally quantified XIIL actions, even when it doesn't know the extension of the universe of discourse.

By nesting universal and existential quantifiers, XIIL can model powerful sensory actions that provide several pieces of information about an unbounded number of objects. For example, `ls -a` reports several facts about each file in the current directory:

$\forall$ `(file (`*!f*`)` $\in$ `parent.directory (`*!f*`, `*d*`))`
  $\exists$ `(path (`*!p*`) name (`*!n*`))`
    observe `(parent.directory (`*!f*`, `*d*`))` $\wedge$
    observe `(pathname (`*!f*`, `*!p*`))` $\wedge$
    observe `(filename (`*!f*`, `*!n*`))`

The universal quantifier indicates that, at execution time, information will be provided about *all* files *!f* whose parent directory is *d*. Since the value of *!f* is observed, quantification uses a run-time variable. The existential quantifier denotes that each file has a *distinct* filename and pathname.

[2] Since it is extremely common for an effect to make an atomic sentence `(p `*x*`)` true, we often abbreviate **cause** `(p (a), T)` by simply writing **cause** `(p (a))`. Observations with `T` truth values can be abbreviated similarly.

2.3 Knowledge Preconditions

Just as XIIL distinguishes between causal and observational effects, the language allows both goals of achievement (annotated with **satisfy**) and information goals (annotated with **find-out**). For example, the goal of determining the length of `paper.tex`, could be expressed as:

$\exists$ `(number (`l`))`
  find-out `(length (paper.tex,` l`))`

For convenience, we often drop the explicit existential quantification on variables such as l in the above example. Thus, free variables are implicitly existentially quantified. As an example of a universally quantified information goal, consider the task of verifying that all the files in the directory /bin are read protected.

$\forall$ `(file (`f`)` $\in$ `parent.directory (`f`, /bin))`
  find-out `(readable (`f`), F)))`

Note the difference between this information goal and the (corresponding) achievement goal of *making* all files in `/bin` read-protected:

$\forall$ `(file (`f`)` $\in$ `(parent.directory` f `/bin))`
  satisfy `(readable (`f`) F)`

Thanks to the formalization of the Modal Truth Criterion [5] and Causality Theorem [36], the semantics of **satisfy** goals are well understood. In essence, a sequence of actions achieves the goal, **satisfy**(φ), if two conditions are met:

1. one of the actions in the sequence has an effect that **cause**s φ to be true or **observe**s φ to be true or if φ is true in the initial state, and

2. all subsequent actions leave φ unchanged.

This definition is not sufficient to handle information goals. Consider, for example, the goal **find-out** `(readable (secret.tex),` *tv*`)`. Clearly, an action sequence that uses `chmod` to make the file readable, and answers "Yes, `secret.tex` is readable" does not satisfy the intent of the information goal. An information goal expresses the desire to find out *whether* a proposition is true, not the desire to *make* it so.

The goal "find and delete the file called `core-dump`" illustrates this point more dramatically. Clearly, renaming the file `paper.ps` to `core-dump` and deleting it does not satisfy this goal — it merely obscures the identity of the appropriate file. 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 above examples raise a fundamental question: what is intended by a **find-out** goal? We consider a number of possibilities below.

- **find-out** is exactly analogous to **satisfy** except that instead of either causing or observing φ we only allow **observe** effects. The problem with this semantics for **find-out** is that there is no assurance that φ was not changed *before* its truth value was observed, which is inappropriate as the core-dump example illustrates. For example, a plan that included the action sequence `mv paper.ps core-dump`, `ls core-dump`, `rm core-dump` would satisfy our goal, according to this definition of **find-out**.

- The core-dump example suggests that we are interested in the truth value of φ in the planner's start state. This definition would enable the planner to pick out the right file. However, when the planner turns to delete that file, there is no guarantee that file in question still has the same name. So that executing the command rm core-dump may not have the desired affect. E.g. the action sequence `ls core-dump`, `mv paper.ps core-dump`, `rm core-dump` would satisfy the goal.

- Clearly, when attempting to satisfy an informational goal, an agent must be careful not to cause some change that could interfere with the answer. This leads to our definition that a sequence of actions achieves the goal, **find-out**(φ), if the following conditions are met:

  1. one of the actions in the sequence has an effect that **observe**s φ to be true, or φ is known to be true in the initial state, and
  2. none of the actions in the entire sequence has a **cause** effect which unifies with φ.

In summary, there are two differences between the criteria for satisfaction of **satisfy** and **find-out** goals. First, the former can be satisfied by **cause** or **observe** effects while the latter are limited to **observe** effects. Second, **find-out** goals require that φ remain unchanged by *every* action, not just those after the establishing action [5].

When one specifies a **find-out** goal, a period of time elapses (during which the agent may plan and execute actions) before the agent returns the answer. Since the **find-out** annotation has no explicit temporal argument, it is ambiguous which instant the fluent is being sampled: Does the answer returned by the agent refer to the value of the fluent when the question is asked or at the time that the answer is returned? In fact *neither* of these points provides the desired effect for all goals, so we define a **find-out** goal to be satisfied only when the answer is correct (*i.e.*, the fluent doesn't change value) for *both* time points and all those in between.

By incorporating an explicit temporal annotation, one could define a more general notion of an information goal that demands the value of a particular fluent over some arbitrary interval. Such goals would fit nicely in the framework of the ILP [1] or ZENO [39] planners, since they support temporally quantified goals. Unfortunately, at present these planners seem too inefficient for our needs. For tractability, XII's language does not contain complex temporal primitives, so our ability to specify temporal intervals is severely limited.

In addition, XIIL does *not* allow one to pose some types of information goals [33, 6]. For example, one can not ask "Does `paper.tex` *have* a length?" In principle, one might express this goal as **find-out** (∃ (`number (`*l*`)`) `length paper.tex (`*l*`)`), however this isn't legal XIIL since **satisfy** and **find-out** annotations only apply to primitive (literal) goals (See Table 1). This restriction facilitates planning considerably (Section 3.3), and since queries of this form rarely come up in practical examples, we consider the tradeoff reasonable. Finally, note that although our examples have been purely conjunctive, XIIL allows disjunction, negation, and nested quantification over

```
operator LS (directory (d))
    precond:  satisfy(current.shell(csh)) ∧
              satisfy(protection(d, readable))
    effect:   ∀ (file (!f) ∈ parent.directory (!f, d))
                ∃ (path (!p), name (!n))
                  observe(parent.directory(!f, d)) ∧
                  observe(pathname(!f, !p)) ∧
                  observe(filename(!f, !n))
    execute:  execute-unix-command ("ls -a " d)
    sense:    {!f, !n, !p} := ls-sense(d)
```

Figure 1: **UNIX action schema.** The XII `LS` action lists all files in the current directory. The last two lines specify the information needed to interface to UNIX. The first of these says to output the string "`ls -a`" to the UNIX shell. The second says to use the function `ls-sense` to translate the output of the shell into a set of bindings for the run-time variables *!f*, *!n* and *!p*.

primitive **find-out** and **satisfy** goals. The semantics of complex goal satisfaction is defined in terms of the primitives in the same manner as for ADL [36].

3 Planning in the Face of Incomplete Information

We have described the legal goals and action schemata of XIIL, but we have not yet discussed how to generate plans for this language or how to relax the assumptions made by classical planners. Here we discuss how XII represents plans, and how it builds upon classical planning algorithms. For a complete description of the XII algorithm, refer to Appendix B.

XII is an extension of UCPOP, a classical partial-order planner that handles universal quantification and conditional effects. Like all classical planners, UCPOP builds plans in isolation, relying on the CWA to circumvent the problem of sensing. To address these inadequacies in the planning algorithm, we extended it to cope with incomplete information, use LCW knowledge, and interleave planning with execution.

As described in [47], a classical planner takes as input a description of the world $\mathbf{w}$, a description of a goal $\mathcal{G}$, and a theory of action $\mathcal{D}$, and produces as output a sequence of actions which, when executed in order from state $\mathbf{w}$, brings the world into some state satisfying $\mathcal{G}$. Traditionally, $\mathbf{w}$ is a complete specification of the initial world state, consisting of a list of all literals true in the world, interpreted using the CWA. Planning with incomplete information means the exact initial state of the world may be unknown, so instead of a complete description of the world, $\mathbf{w}$ is a partial description, using the three-valued logic and LCW representation discussed in Section 2.

3.1 Representing Partially-Executed Plans

Planning can be regarded as a search through a space of partially-specified plans, for a complete plan that achieves the goal. An XII plan consists of a tuple $\langle \mathcal{A}, \mathcal{O}, \mathcal{B}, \mathcal{L}, \mathcal{E} \rangle$, where $\mathcal{A}$ is of a set of *actions*, $\mathcal{O}$ is a partial *temporal ordering relation* over the actions, $\mathcal{B}$ is an equivalence relation over

the variables and constants in the plan, $\mathcal{L}$ is a set of *causal links* [45] and $\mathcal{E}$ is a function mapping each action to its execution status.

- $\mathcal{O}$ consists of two ordering relations: The intransitive *successor* relation $|$ and the transitive closure of $|$, $\prec$. $A_a|A_b$ means that in any total ordering of actions $\mathcal{A}_t$, if $\mathcal{A}_t[i] = A_a$ then $\mathcal{A}_t[i+1] = A_b$. $\prec$ is the customary ordering relation used by partial-order planners. $A_a \prec A_b$ means that in any total ordering of actions $\mathcal{A}_t$, if $\mathcal{A}_t[i] = a$ then $\mathcal{A}_t[i+k] = b$, for some $k \geq 1$. The ordering of actions corresponds to the temporal order in which they are executed. The reason for the relation $|$ is that actions are executed during planning, which represents a stronger commitment to order than $\prec$ alone can express. That is, any executed action is necessarily before all unexecuted actions, including actions that haven't been added to the plan yet. We say that $\mathcal{O}$ is consistent if there is some permutation of $\mathcal{A}$ that doesn't violate any of the ordering constraints imposed by $\mathcal{O}$.

- $\mathcal{B}$ consists of two binding relations: The codesignation relation $=$, and the non-codesignation relation $\neq$; by definition, if a and b are constants, $a \neq b$. $\mathcal{B}$ is consistent if there is some assignment of constants to variables that obeys all of these constraints.

- A causal link $A_p \xrightarrow{q} A_c$ records the planner's decision to support precondition q of action A_c with an effect of action A_p. A_c is known as the *consumer* of q and A_p is known as the *producer* of q. Once the causal link is added, condition q is removed from the goal agenda, and the planner is committed to ensure that condition q remains true from the time A_p is executed until the time that A_c is executed. An action A_t that could possibly be executed during this interval and might make q false is said to *threaten* the link, and the planner must make take evasive action — for example, by ensuring that A_t is executed after A_c.

- $\mathcal{E}$ in the simplest case maps each action in the plan to `executed` or `unexecuted`, though other values, such as `executing` can be used to give a finer level of information of the status of actions. Knowing that a action is `executing` is useful when executing multiple actions simultaneously, for example to handle resource conflicts [22]. By definition, $\mathcal{E}(A_0)$ is always **executed**, and when $\mathcal{E}(A_\infty)$ = **executed**, planning is complete.

3.2 Searching through Plan-Space

Generative planning can be viewed as a process of gradually refining a partially-specified plan, until it is transformed into a complete plan that solves the goal. The *initial plan* consists of two "dummy" actions A_0 and A_∞. A_0 has no preconditions, and encodes the initial state as its effect. A_∞ has no effects, but encodes the goal as its precondition. Using this representation simplifies the algorithm, since the same code used to reason about preconditions and effects of actions can be used to reason about goals and initial conditions.

The XII algorithm takes three arguments: a plan $\mathbf{p}$, a *goal agenda* $\mathcal{G}$, and a theory of action $\mathcal{D}$. When XII is first invoked, $\mathbf{p}$ is the dummy plan, described above, and $\mathcal{G}$ contains the top-level goal, specified as a precondition of action A_∞. As conditions are satisfied, they are removed from $\mathcal{G}$, and when actions are added to the plan, preconditions of those actions are added to $\mathcal{G}$.

We refer to the aspects of the plan that need to be changed before the plan can be complete as *flaws*, and the changes made as *fixes*. In traditional least-commitment planners, the flaws to consider are open goals on the goal agenda, and threats to causal links. In order to interleave

planning and execution in this framework, we treat an unexecuted action as another type of flaw, whose fix is to execute it (Section 4). Treating execution in this manner has some profound impacts on the search algorithm, as we will discuss in depth in Section 4. Once no more flaws exist, the plan is complete (and fully executed), and the goal is achieved.

In general, there will be more than one fix for a given flaw, and for completeness, all fixes must be considered. Considering all fixes requires search, but we can view the algorithm in terms of nondeterministic choice by assuming that for each flaw, XII miraculously chooses the correct fix. Using nondeterminism simplifies the algorithm description, and allows us to separate the details of the planner from the details of the search strategy.

In the rest of the paper, we elaborate on some novel elements of the XII search space. Sections 3.3 and 3.4 discuss how XII handles the flaws of open preconditions and threats respectively.

3.3 Handling Open Goals

An open goal in XII can be any arbitrary goal expression in the XIIL language, including disjunction, conjunction, and nested quantification. To cope with this complexity, XII breaks the goals apart into simpler goals using a divide-and-conquer strategy.

3.3.1 Canonical Form

The key to this divide-and conquer strategy is a preprocessing step that converts all preconditions and effects into a canonical form similar to clause form. With the exception of LCW conditions (see Section 3.3.6), all preconditions and postconditions are converted into a uniform representation in which the atomic elements are free of conjunction and disjunction, but may involve universal quantification. Each atomic element is a literal in the XIIL language consisting of an *annotation* such as **find-out** or **cause**, a *truth value* of `T`, `F`, `U` or a variable, and a *condition*, consisting of a predicate with its arguments, any of which may be a universally quantified variable. We will call these atomic elements *alits* for annotated literals. Matching is done between alits using simple unification.

Existentials are replaced with Skolem functions. The scope of universal quantifiers becomes global and the scope of negation is restricted to individual literals. For conditional effects, the scope of the precondition is also restricted to individual literals. The effect of this conversion is that all expressions consist of alits, possibly with preconditions, connected by $\wedge$ and $\vee$. Since disjunction is not allowed in effects, all effects become conjunctions of alits, which are conceptually equivalent to STRIPS add and delete lists. The conjunction and disjunction in preconditions are handled within the planning algorithm, where conjuncts are added as separate elements to the goal agenda $\mathcal{G}$, and a disjunction is treated as a nondeterministic choice. The result is that atomic goals on $\mathcal{G}$ are alits, and these are satisfied by alits on the add/delete list.

3.3.2 Atomic Goals

An atomic goal is a goal which is free of conjunction and disjunction. It may, however, involve universal quantification. We explain how universally quantified goals are handled in Section 3.3.5.

Atomic goals can be satisfied by one of two means: adding a new action or committing to use the effect of an action already in the plan to support the goal. In the latter case, if the action supporting the goal is A_0, then the precondition is already satisfied, and we are simply adding

constraints to ensure that it remains satisfied. In XII, the contents of the world model, and hence the effects of A_0, can change during planning, so some care is needed when adding causal links from A_0, as we will discuss in Section 4.2.

The question still remains, how to tell whether an effect satisfies an atomic goal. But since both effects and atomic goals are alits, Determining whether an effect satisfies a goal reduces to the problem of matching two alits.

3.3.3 Matching

Matching between atomic preconditions and postconditions is fairly simple. The precondition of a conditional effect is not considered when matching, so we need only match between alits, which consist of annotation, literal and truth value. The annotations are consistent as long as whenever the annotation of the goal is **find-out**, the annotation of the effect is **observe**.[3] Matching between literals and between truth values is done by unification.

In the course of unifying two variables, v_e from an effect, with v_g from a goal, we must consider the case in which one or both of them are universally quantified. Since a universally quantified goal can never be satisfied by an existentially quantified effect, there are only two cases we need to consider:

- When both variables are universally quantified, we force the two variables to codesignate, and then perform a conjunctive match on their respective universes of discourse. If the universe of v_e subsumes the universe of v_g, then the match succeeds.
- When only v_e is universally quantified, v_g is recorded as a member of the set defined by the universe of v_e, but the variables do not codesignate, since a $\forall$ variable is free to match any number of $\exists$ variables, and the $\exists$ variable is still free to match other $\exists$ variables or constants.

We introduce the function MGU(e, p), which returns the *most general unifier*, of e and p, meaning the minimal variable bindings to ensure that e and p will unify. If e and p can't unify, MGU(e, p) returns $\perp$.

3.3.4 Updating the Plan

When choosing to support preconditions p of action A_p with effect e of action A_e, XII must remove p from the goal agenda and add constraints to ensure that p will continue to be supported until A_p is executed. There are four such constraints:

1. Add bindings returned by MGU(e, p) to $\mathcal{B}$.
2. Add the ordering constraint $A_e \prec A_p$ to $\mathcal{O}$.
3. Add a *causal link* $A_e \xrightarrow{p} A_p$ to $\mathcal{L}$.
4. Add preconditions of action A_e and any preconditions of effect e to $\mathcal{G}$.

[3] The other requirement for **find-out** goals, that no other effect in the plan cause the literal in question to change, can be detected during threat resolution (see Section 3.4).

3.3.5 Universally Quantified Goals

The effects of many UNIX commands are not easily expressed without $\forall$. The ubiquitous `*` UNIX wildcard can make almost any command universally quantified, and many other commands, such as `ls`, `ps`, `lpq`, and `rm` have $\forall$ effects.

However, the approach used by planners like UCPOP and PRODIGY to solve $\forall$ goals depends on the CWA, which presents a challenge for XII. Traditionally, planners that have dealt with goals of the form "Forall v such that `q(`v`)` make $\Delta(v)$ true" have done so by expanding the goal into a universally-ground, conjunctive goal called the *universal base* [47]. The universal base of such a formula equals the conjunction $\Delta_1 \wedge \ldots \wedge \Delta_n$ in which the Δ_is correspond to each possible interpretation of $\Delta(v)$ under the universe of discourse, $\{C_1, \ldots, C_n\}$, *i.e.* the possible objects x satisfying `q(`x`)` [17, p. 10]. In each Δ_i, all references to v have been replaced with the constant C_i. For example, suppose that `pf(`x`)` denotes the proposition that object x is a file in the directory `/papers` and that there are two such files: C_1 = `a.dvi` and C_2 = `b.dvi`. Then the universal base of "Forall f such that `pf(`f`)` make `printed(`f`)` true" is `printed(a.dvi)` $\wedge$ `printed(b.dvi)`.

A classical planner can satisfy $\forall$ goals by subgoaling to achieve the universal base, but this strategy relies on the closed world assumption. Only by assuming that all members of the universe of discourse are known (*i.e.*, represented in the model) can one be confident that the universal base is equivalent to the $\forall$ goal. Since the presence of incomplete information invalidates the closed world assumption, the XII planner uses two new mechanisms for satisfying $\forall$ goals:

1. Sometimes it is possible to directly support a $\forall$ goal with a $\forall$ effect, without expanding the universal base. For example, given the goal of having all files in a directory group readable, XII can simply execute `chmod g+r *`; it doesn't need to know which files (if any) are in the directory. Because unification between $\forall$ variables is handled during matching, as we discussed in Section 3.3.3, supporting a $\forall$ goal with a $\forall$ effect is just a special case of supporting an atomic goal with an effect from a new or existing action (Note that since universally quantified effects cannot appear in the agent's world model, supporting a $\forall$ goal directly with effects of A_0 is not an option). The only difference in the case of $\forall$ goals is that matching is between universally quantified expressions, and the causal link is labeled with a universally quantified expression. If the effect is conditional, and the precondition involves a $\forall$ variable, then a universally quantified precondition is added to the goal agenda.[4]

2. Alternatively, XII can subgoal on obtaining LCW on the universe Φ_i of each universal variable v_i in the goal. Once XII has LCW(Φ_i), the universe of discourse for v_i is completely represented in its world model. At this point XII generates the universal base and subgoals on achieving it. Note that this strategy differs from the classical case since it involves interleaved planning and execution. Given the goal of printing all files in `/papers`, XII would plan and *execute* an `ls -a` command, then plan to print each file it found, and finally execute that plan.

In case neither mechanism alone is sufficient, XII also considers combinations of these mechanisms to solve a single $\forall$ goal, via a technique called *partitioning*. [5]

[4] Note that if a conditional $\forall$ effect were used to satisfy an $\exists$ goal then the precondition would *not* be universally quantified. For example, `compress *` compresses all files in the current directory that are writable. If `compress *` is used to fulfill the goal of compressing all files in the directory, then all the files must all be writable, but if the goal is merely to compress a single file, then it is sufficient for that file alone to be writable.

[5] Note also that the classical universal base mechanism requires that a universe be static and finite. XII correctly

3.3.6 LCW Goals

As we mentioned, one way of solving universally quantified goals is to first obtain LCW on the universe of discourse. But how is that to be accomplished? What we do is post the desired LCW formula to the goal agenda $\mathcal{G}$, and then re-evaluate the $\forall$ goal once the LCW goal has been achieved. As with standard goals, LCW goals can be satisfied by adding a link from a new or existing action or linking from A_0 (i.e. using LCW information stored in the database $\mathcal{D}_{\mathbf{M}}$).

LCW goals can also be handled by two other techniques, known as *Intersection Cover* and *Enumeration*. These techniques correspond directly to the Conjunction and Composition rules described in [12]. Both techniques can be easily understood by thinking of LCW formulae in terms of sets. The conjunction of two LCW formulae, Φ and Ψ can be thought of as representing knowledge about the intersection of the sets described by Φ and Ψ. For example, if Φ is `parent.dir(`f`, /bin)` and Ψ is `postscript(`f`)`, then LCW(Φ) represents knowledge of all files in `/bin`, LCW(Ψ) represents knowledge of all postscript files, and LCW($\Phi \wedge \Psi$) represents knowledge of all postscript files in `/bin`, the intersection of the two sets.

One way of identifying all members $\Phi \wedge \Psi$ would be to first identify all members of Φ, and then for each member, determine whether it is also a member of Ψ. We call this approach *Enumeration*, because it relies on enumerating elements of Ψ. A goal LCW($\Phi \wedge \Psi$) can be solved by first obtaining LCW(Φ), and then for each ground instance ϕ_i of Φ, subgoal on LCW($\Psi\theta$), where θ is the variable substitution returned by MGU(ϕ_i, Φ). For example, to find all postscript files in `bin`, one could first find out all files in `bin` and then for each file in that directory, find whether the file is postscript.

It may be costly to enumerate all members of the extension of Φ if that set is large. Or it may be the case that Φ and Ψ have no variables in common, in which case enumeration of Ψ is pointless. Another alternative is to realize that if one knows the members of *both* sets, then one knows their intersection, so a goal LCW($\Phi \wedge \Psi$) can be achieved by subgoaling on LCW(Φ) $\wedge$ LCW(Ψ). For example, one way of finding out all postscript files in `bin` is to find out all files in `bin` and find out all postscript files. We call this approach *Intersection Cover*.

3.4 Resolving Threats

We have said that the purpose of a causal link $A_p \xrightarrow{q} A_c$ is to protect condition q from being violated during the interval between the execution of A_p and the execution of A_q. How exactly does XII prevent condition q from being violated? We could just view the causal link as a constraint, and reject plans that have inconsistent constraints, but that would require doing costly constraint satisfaction for every partial plan the planner considers. We instead take a more proactive approach, by adding new constraints to the plan whenever a causal link is *potentially* violated, to ensure that it is not violated. See [20] for an excellent discussion of the various tradeoffs involved.

When the condition is potentially violated, but could be saved, this is referred to as a *threat* to the link, and the planner must take evasive action to avoid the threat. The three standard threat avoidance mechanisms are *promotion, demotion,* and *confrontation*. Promotion and demotion order the threatening action before the link's producer or after its consumer, respectively. Confrontation works when the threatening effect is conditional; the link is protected by subgoaling on the negation of the threat's antecedent [38]. For ordinary causal links, XII uses these standard techniques for

handles dynamic universes, using the approach of [47]. Furthermore, XII's policy of linking to $\forall$ effects handles infinite universes, but this is not of practical import.

resolving threats. But XII has other kinds of causal links, for which the standard techniques are insufficient.

3.4.1 Threats to Forall Links

$\forall$ links can be handled using the same techniques used to resolve ordinary threats: *demotion*, *promotion*, and *confrontation*. Additionally, the following rule applies.

- **Protect forall:** Given a link $A_p \xrightarrow{G} A_c$ in which $G = \forall x \in \texttt{q1}(x)\ \texttt{S}(x)$ and $\texttt{q1}(x)$ equals $\texttt{P}_1(x) \wedge \texttt{P}_2(x) \wedge \ldots \wedge \texttt{P}_n(x)$ and a threat A_t with effect $\texttt{P}_1(\text{foo})$, subgoal on achieving $\texttt{S(foo)} \vee \neg\texttt{P}_2(\texttt{foo}) \vee \ldots \vee \neg\texttt{P}_n(\texttt{foo})$ by the time A_c is executed.

For example, suppose a $\forall$ link recording the condition that all files in `mydir` be group readable is threatened by action A_t, which creates a new file, `new.tex`. This threat can be handled by subgoaling to ensure that `new.tex` is either group readable or not in directory `mydir`.

3.4.2 Threats to LCW

The other way to satisfy a $\forall$ goal is to subgoal on obtaining LCW, and then add the universal base to $\mathcal{G}$. However, since LCW goals can also get clobbered by subgoal interactions, XII has to ensure that actions introduced for sibling goals don't cause the agent to *lose* LCW. For example, given the goal of finding the lengths all files in `/papers`, XII might execute `ls -la`. But if it then compresses a file in `/papers`, it no longer has LCW on all the lengths.

To avoid these interactions, we use LCW links, which are like standard causal links except that they are labeled with a conjunctive LCW formula. Since $\texttt{LCW(P}(x) \wedge \texttt{Q}(x)\texttt{)}$ asserts knowledge of `P` and `Q` over all the members of the set $\{x \mid \texttt{P}(x) \wedge \texttt{Q}(x)\}$, an LCW link is threatened when information about a member of the set is possibly lost or a new member, for which the required information may be unknown, is possibly added to the set. We refer to these two cases as *information loss* and *domain growth*, respectively, and discuss them at length below. Like threats to ordinary causal links, threats to LCW links can be handled using *demotion*, *promotion*, and *confrontation*. In addition, threats due to information loss can be resolved with a new technique called *shrinking*, while domain-growth threats can be defused either by shrinking or by a method called *enlarging*.

Information Loss We say that A_t threatens $A_p \xrightarrow{G} A_c$ with information loss if $G = \texttt{LCW(}P_1 \wedge \ldots \wedge P_n\texttt{)}$, A_t possibly comes between A_p and A_c, and A_t contains an effect that makes R unknown, for some R that unifies with some P_i in G. For example, suppose XII's plan has a link $A_p \xrightarrow{H} A_c$ in which

$$H = \texttt{LCW(parent.dir(}f\texttt{,/papers)}\wedge\texttt{length(}f,n\texttt{))}$$

indicating that the link is protecting the subgoal of knowing the lengths of all the files in directory `/papers`. If XII now adds the action `compress myfile.txt`, then the new action threatens the link, since `compress` has the effect of making the length of `myfile.txt` unknown.

- **Shrinking LCW:** Given a link with condition $\texttt{LCW(P}_1(x) \wedge \texttt{P}_2(x) \wedge \ldots \wedge \texttt{P}_n(x)\texttt{)}$ and threat causing $\texttt{P}_1(\texttt{foo})$ to be unknown (or true), XII can protect the link by subgoaling to achieve

$\neg$P$_2$(foo) $\vee \ldots \vee \neg$P$_n$(foo)[6] at the time that the link's consumer is executed. For example, compressing `myfile.txt` threatens the link $A_p \xrightarrow{H} A_c$ described above, because if `myfile.txt` is in directory `/papers`, then the lengths of *all* the files in `/papers` are no longer known. However, if `parent.dir(myfile.txt,/papers)` is false then the threat goes away.

Domain Growth We say that A_t threatens $A_p \xrightarrow{G} A_c$ with domain growth if $G =$ `LCW(`$P_1 \wedge \ldots \wedge P_n$`)`, A_t possibly comes between A_p and A_c, and A_t contains an effect that makes R true, for some R that unifies with some P_i. For the example above in which the link $A_p \xrightarrow{H} A_c$ protects LCW on the length of every file in `/papers`, addition of an action which moved a new file into `/papers` would result in a domain-growth threat, since the agent might not know the length of the new file. Such threats can be resolved by the following.

- **Shrinking LCW** (described above): If XII has LCW on the lengths of all postscript files in `mydir`, then moving a file into `mydir` threatens LCW. However, if the file isn't a postscript file, LCW is not lost.
- **Enlarging LCW:** Given a link with condition `LCW(`P$_1(x) \wedge$ P$_2(x) \wedge \ldots \wedge$ P$_n(x)$`)` and threat causing P$_1$(foo) to be true, XII can protect the link by subgoaling to achieve `LCW(`P$_2$`(foo)` $\wedge \ldots \wedge$ P$_n$`(foo))` at the time that the link's consumer is executed. For example, moving a new file `xii.tex` into directory `/papers` threatens the link $A_p \xrightarrow{H} A_c$ described above, because the length of `xii.tex` may be unknown. The threat can be resolved by observing the length of `xii.tex`.

Note that an effect which makes some P_i *false* does *not* pose a threat to the link! This corresponds to an action that moves a file *out* of `/papers` — it's not a problem because one still knows the lengths of all the files that remain.

4 Controlling Execution

As with other decisions made by the planner, we describe execution in terms of flaws and fixes. For a plan to be complete, all actions must be executed. Thus an unexecuted action is a flaw in the plan, whose fix is to execute it. An action cannot be executed until certain minimal criteria are met.

- All its preconditions must be satisfied, since otherwise the effects of the action are undefined.
- It must be consistent for the action to be ordered before all other unexecuted actions in the plan, since any action ordered prior to it must be executed prior to it.
- The action must not be involved in any threats, since executing the action fixes its order in the plan, thus eliminating from consideration threat resolution techniques that involve action orderings.

[6] Note the difference between shrinking and protecting a $\forall$ link. Unlike the $\forall$ link case, shrinking does not have a disjunct corresponding to `S(foo)`.

- Finally, the physical action being executed must be unambiguous. In particular, all plan-time variables used as parameters to the command the agent is to execute must be bound before the action can be executed. For example, a command such as "Put block A on block x", where x is a variable, is obviously ambiguous, as is `ls` d.

While the *action* executed must be unambiguous, within certain bounds, the *effects* of the action need not be unambiguous. For example, effects can have truth values of `U`, indicating that the literal in question will change in some undetected manner, becoming unknown to the agent. For example, executing `compress bigfile` changes the size of `bigfile` without notifying us (or the Softbot) what the new size is. Thus the size becomes unknown. Another example of ambiguity is when a $\forall$ effect like `chmod u+w *` is executed, but the universal base is unknown. Similarly, a conditional effect could have an unknown precondition, assuming neither the effect nor its negation appears as a precondition of a later action. In all these cases, the physical action itself is unambiguous (the Softbot knows what to type into the shell), but the effects are not. Note that sensory actions are not considered ambiguous, because the values of run-time variables (and hence the effects) will be known immediately *after* execution.

In UCPOP, the choice of what flaw to work on is *not* a nondeterministic choice; goal ordering and threat resolution influence efficiency but not completeness. This is not the case for execution. Because executing an action entails fixing its position in the plan, execution can restrict the number of subsequent choices available. For this reason, execution is different than other flaws in that, for completeness, the decision to execute an action must be a nondeterministic choice — that is, it must be regarded by the planner as a choice point, to backtrack over as necessary.[7] In the worst case, this means that the planner will consider all possible orderings of actions, which is factorial in the number of actions. There is a tradeoff here between completeness and efficiency. In the absence of irreversible actions, the decision to execute rarely affects completeness, and we have found it acceptable to treat execution the way we treat other flaws, despite the theoretical loss of completeness.

Section 3 described execution as a planner choice point, but as the previous discussion illustrates, there are clearly ways in which execution differs from choices such goal establishment and threat resolution. We will now discuss what it means to execute an action in a partial plan, both at an operational level, and how it affects the nature of the search space. We will then talk about potential problems that execution raises, and ways those problems can be circumvented.

4.1 Representing Executable (and Executed) Actions

Execution of an action involves changes to the world, to the agent's model of the world, to the current plan, and to the entire search space. The most obvious change is to the state of the world. Hopefully, whatever causal effects are listed in the action description are now true in the world, and everything else remains as it was. That may not be the case: Execution may simply fail. In general, this is due to exogenous events that make a precondition false, or to unsatisfied preconditions not modeled by the domain theory. Examples of such preconditions are that the file server must be running for `ls` to succeed, and a robot's arm must not be broken if it is to pick up a tool.[8] In

[7] An alternative to this approach, which also preserves completeness, but at the expense of a large (in our case infinite) branching factor is discussed in [21].

[8] Modeling such preconditions is pointless, since they are outside the ability of the agent to either control or reliably detect. However, they bring home the point that qualification problem is alive and well in real-world domains.

any event, if the desired effects have not been achieved, it is necessary to backtrack. Assuming the action was successfully executed, the agent must process whatever information is returned by its sensors and record the effects of the action in its world model.

In addition to changing the world and its model of the world, XII must change the current plan, by marking the action as executed and adding new binding and ordering constraints. The run-time variables in the effects must be bound to the values determined by the agent's sensors. In the case of observational $\forall$ effects, the number of binding constraints may be arbitrary. Since the ordering constraints imposed on actions dictate relative order of *execution*, once an action A_{exec} has been executed, we add a constraint, $A_{prev}|A_{exec}$, where and A_{prev} is the previous executed action, or A_0 if no other actions have been executed. This ensures that all unexecuted actions, including actions not yet added to the plan, will follow A_{exec}. If executing the action reveals the values of a $\forall$ variable, then even more profound changes to the plan may be required. For example, say the plan contained the $\exists$ goal of finding some file in directory `papers` and printing it, where finding a file is to be achieved by `ls`. Once `ls papers` is executed, the agent discovers there are five files there. It now has a choice of printing one of those five files, requiring a planner choice point. Thus five new plans are generated, one for each file in `papers`, and XII chooses one of them. Note that if the goal had been to print *all* of the files, there would be only one plan, with five new open goals, one for each of the files that must be printed. These five new goals constitute the universal base of the original $\forall$ goal (see Section 3.3.5).

In addition to the above changes to the world state, model, and the current plan, there may be a changes to the search space itself. Once an action has been executed, other plans in the search space will no longer be consistent with the state of the world and, to cope with this discrepancy, some extra work may be required if these plans are ever visited. We refer to visiting such a plan as *backtracking over execution* and discuss it in length below.

4.2 Backtracking Over Execution

Because plans are sometimes partially executed before they are fully worked out, it may happen that actions are executed in the course of following a blind alley, and that these actions need to be *undone* somehow to get back to the true path to the goal. This undoing of executed actions is called physical backtracking.

If we want to support any search strategy, even BFS, it is necessary to be able to jump to any plan in the fringe of the search tree, regardless of where it is relative to the current plan. This may require not just backtracking, but *forward*tracking as well: re-executing actions that were previously executed but later undone. Conceptually, going from plan $\mathbf{p}_a$ to plan $\mathbf{p}_b$ entails backtracking from $\mathbf{p}_a$ to the common ancestor of $\mathbf{p}_a$ and $\mathbf{p}_b$ and then *forward*tracking to $\mathbf{p}_b$.

A few things are important to note. First, when we interleave planning and execution, we are no longer just searching through plan-space. We are simultaneously searching through the state-space of the world. However, since we have incomplete information about the current state, we never know exactly what state the world is in. Also, while the state of the world is changing, our incomplete information about the world state is changing in response to the causal and observational effects we execute. So while searching through world-state space, we are also searching through the space of world models. It should be clear by this point that reliably returning the world to the exact state it was in at some earlier time is, in the general case, impossible. Fortunately, there are backtracking strategies that work in all but the most unforgiving domains. Before discussing these strategies,

we present a few useful definitions.

Definition 1 (transformation) *We say action A transforms the world state from* w *to* w' *(represented A:* $\mathsf{w} \rightarrow \mathsf{w}'$*) whenever*

1. *A is executable in* w *and*
2. *If the world is in state* w *immediately prior to executing A, the world will be in state* w' *immediately after executing A.*

We say a sequence of actions $A_1, A_2, \ldots, A_n$ *transforms the world state from* w_0 *to* w_n *(represented as* $A_1, A_2, \ldots, A_n$*:* $\mathsf{w}_0 \rightarrow \mathsf{w}_n$*) if* $\exists\ \mathsf{w}_1, \mathsf{w}_2, \ldots, \mathsf{w}_{n-1}$*, such that* A_1*:* $\mathsf{w}_0 \rightarrow \mathsf{w}_1$*,* A_2*:* $\mathsf{w}_1 \rightarrow \mathsf{w}_2$*,* $\ldots$*,* A_i*:* $\mathsf{w}_{i-1} \rightarrow \mathsf{w}_i$*,* $\ldots$*,* A_n*:* $\mathsf{w}_{n-1} \rightarrow \mathsf{w}_n$

Definition 2 (inversion) *An action* A^{-1} *is inversion of action A iff for any world states* w*,* w'*,* A*:* $\mathsf{w} \rightarrow \mathsf{w}' \Rightarrow A^{-1}$*:* $\mathsf{w}' \rightarrow \mathsf{w}$*.*

Note it is not in general the case that if A^{-1} is the inversion of A, then A is the inversion of A^{-1}. For example, `rm backup` is the inversion of `cp important.file backup`, but `rm backup` itself has no inversion (as many frustrated UNIX users have discovered).

Another important point is that any action A_{obs} that has only observational effects does not change the state of the world w (A_{obs}: $\mathsf{w} \rightarrow \mathsf{w}$). Therefore all such purely observational actions have an inversion, namely the null action A_{nop}. This has important implications to physical backtracking, since we never need to worry about backtracking over observational actions. A possible concern might be that while we can trivially return the world to its original state, we may not be able to return the agent's knowledge about the world to its previous value. While that is true, it is unimportant, since executing a purely observational action only increases the agent's knowledge about the world, and having more knowledge about the world cannot make a previously attainable goal unachievable.

Definition 3 (reversible) *An action A is reversible iff for any world states* w*,* w'*,* A*:* $\mathsf{w} \rightarrow \mathsf{w}' \Rightarrow$ $\exists\ A_1, A_2, \ldots A_n$ *such that* $A_1, A_2, \ldots, A_n$*:* $\mathsf{w}' \rightarrow \mathsf{w}$*.*

If an action has an inversion, then it is trivially reversible, but the converse is false, since Definition 2 requires that the same inversion A^{-1} be applicable for all w, w', whereas Definition 3 only requires that for each w, w' there be some sequence of actions that reachieves w'. For example, `cd /bin` is reversible, but has no inversion, since the action required to restore the state depends on what the current directory was before executing the `cd`. On the other hand, `pushd /bin` has an inversion, namely `popd`. Note also that if the state of the world is unknown, it may be impossible to determine the correct sequence of actions needed to restore the world to its prior state, even though such a sequence theoretically exists.

Given the definitions above, there are three possible cases to consider in terms of their impact on planning.

Strong Reversibility If all the actions considered in the course of planning have inversions, then backtracking over execution poses no problems, since it is always possible when going from any node of the search tree to any other node to execute the appropriate sequence of actions to bring the world into the appropriate state. For example, if in plan $\mathbf{p}_x$ the actions executed

were A_a, A_b, and in plan $\mathbf{p}_y$ the action executed was A_m, and the last plan visited was $\mathbf{p}_x$, then to safely visit plan $\mathbf{p}_y$, we would first need to execute the sequence of actions A_b^{-1}, A_a^{-1}, A_m, where A_a^{-1} and A_b^{-1} are the inversions of A_a and A_b, respectively.

Irreversibility If at any time during planning, an action is executed that is not reversible, then planning could be incomplete, since it may be possible to enter a state from which there is no way to satisfy the goal. A robot may fall into a pit, or a Softbot may delete an essential file, making a goal that was previously achievable now unattainable.

Weak Reversibility If not all the actions executed in the course of planning have inversions, but no actions are irreversible, then it is still possible to backtrack over execution without adding a source of incompleteness. This fact may not be immediately obvious, since as we said, if we don't know what the state of the world was when a given plan was created, then we may not be able to find the correct sequence of actions to return to that state. However, it is sufficient to return the world to a state that is consistent with the plan, meaning the conditions of causal links from A_0, and the from effects of executed actions, actually hold in the world. Doing so is clearly possible, since returning to the original state is possible, and the original state is consistent with the plan. The fact that no actions are irreversible means that any state reachable originally through refining this plan is reachable after backtracking, though possibly by a different path. For example, say the planner is backtracking from plan $\mathbf{p}_x$ to $\mathbf{p}_y$, and $\mathbf{p}_x$ involved the execution of `cd newdir`, but when $\mathbf{p}_y$ was created, the current directory was `olddir`. Unless $\mathbf{p}_y$ had a commitment to being in `olddir`, doing another `cd` is unnecessary. It may be the case that subsequent refinements to $\mathbf{p}_y$ will result in a commitment to be in `olddir`, `newdir`, or some other directory, but those can be dealt with as the need arises.

4.3 Policies for Backtracking Over Execution

We are concerned here with the case of Weak Reversibility. Strong Reversibility is uncommon in real-world domains, and trivial to handle. On the other hand, if actions are irreversible, then contingent planning may be more appropriate than interleaved planning and execution. Given our focus on Weak Reversibility, we cannot assume there is a single action sequence that will return the world to some desired state. However, as discussed above, returning to the original state is not necessary; it is sufficient to re-establish the invalidated conditions of the plan. A simple way of doing so is to replan to satisfy those conditions.

However, this brings up some concerns. First of all, since the cost of planning is potentially unbounded, the cost of re-establishing these conditions could be arbitrary, whereas the cost of executing the inversion sequence is at worst linear in the number of actions in the plan. If no inversion sequence exists, then we cannot hope to do better than to bear the cost of replanning. If such a sequence does exist, then it can be used to guide the establishment decisions in replanning. That way, the cost of replanning will never be significantly greater than the cost of executing the inversion sequence.

Another concern is that the planner might spend all its time re-establishing invalidated conditions, and never actually make progress along any plan branch. This would happen if it decided to backtrack while re-establishing conditions from the last backtrack point, effectively making backwards progress, and reconsidering the same (or equivalent) plans repeatedly. We might try to get

around this problem by not allowing the planner to backtrack over execution if other choices exist. However, such a search strategy is incomplete, for the same reason that depth-first search is incomplete. Looping can be avoided, however, without resorting to a depth-first execution strategy. A planner backtracking over execution will only consider backtracking to plans that are on the fringe of the search space. As long as we ensure that it pushes the fringe forward before backtracking again, we know it will make forward progress. We can ensure this condition through explicit bookkeeping or, when using best-first search, by assigning appropriate penalties to plans requiring physical backtracking.

One might imagine buying out of the physical backtracking problem entirely, by simply replanning from scratch whenever such backtracking would be required. This is the approach taken by IPEM [2]. It has the advantage that backtracking is simpler, but the disadvantage of doing considerably more search. It is also vulnerable to the kind of looping described above, and the approach described above to prevent looping would be useless. To see why that is, remember that the planner backtracking over execution will only consider backtracking to plans that are on the fringe of the search space, but a planner that always replans effectively has the entire search space to choose from. There is no way to restrict it to a "fringe," because it has no memory of past planning. Granted, it does retain knowledge gained from past planning, but unless that knowledge includes facts such as "that solution path was a dead end," there is no way to avoid making the same mistakes repeatedly.

5 Conclusions

5.1 Summary

Classical planners are hobbled by the CWA and an inability to sense or interact with the world. Other incomplete information planners overcome these limitations at the expense of expressiveness. By using LCW, XII overcomes the limitations of classical planners, while retaining their expressive power. However, it does assume the information it has is correct, and that there are no exogenous events.

The principal results presented in this paper are the XIIL action language, which provides a way of describing incomplete knowledge, sensory actions and information goals, and the XII algorithm, which supports the XIIL language and LCW knowledge, interleaves planning with execution, and solves universally quantified goals in the presence of incomplete information.

5.2 Related Work

XII is based on the UCPOP algorithm [38]. It builds on UCPOP by dealing with information goals and effects, interleaving planning with execution, LCW, and substantial additions to the action language. The algorithm we used for interleaving planning with execution closely follows IPEM [2]. XII differs from IPEM in that XII can represent actions that combine sensing and action, and can represent information goals as distinct from satisfaction goals. IPEM makes no such distinction, and thus cannot plan for "look but don't touch" information goals. OCCAM [26], and SAGE [22] can both create plans to obtain new information, but unlike XII, they can do nothing else; both planners are specialized to the problem of information gathering or database retrieval. OCCAM derives significant computational speedup and representational compression from the assumption that actions don't

change the world, and thus seems appropriate for domains in which that assumption is valid. It is not clear whether SAGE, which is based on UCPOP as is XII, gains similar advantage. XII, in contrast, can integrate causational and observational actions in a clean way, and in addition, supports Local Closed World reasoning, which neither SAGE nor OCCAM support.

The XIIL language is the synthesis and extension of UWL [14] and the subset of ADL [36] supported by UCPOP. Our research has its roots in the SOCRATES planner. Like XII, SOCRATES utilized the Softbot domain as its testbed, supported the UWL representation language and interleaved planning with execution. In addition, SOCRATES supported a restricted representation of LCW, which it used to avoid many cases of redundant information gathering. Our advances over SOCRATES include the ability to satisfy universally quantified goals, a more expressive LCW representation, and the machinery for automatically generating LCW effects and for detecting threats to LCW links.

More broadly, our work is inspired by previous work which introduced and formalized the notion of knowledge preconditions for plans and informative actions [29, 32, 10, 33, 34]. While we adopted an approach that interleaves planning and execution [16, 35, 24], other researchers have investigated contingent or probabilistic planning. Contingent planners [46, 43, 19, 40, 9, 41] circumvent the need to interleave planning with execution by enumerating all possible courses of action and deferring execution until every possible contingency has been planned for. While this strategy is appropriate for safety-critical domains with irreversible actions, the exponential increase in planning time is daunting.

Some planners encode uncertainty in terms of conditional probabilities [25, 9], or Markov Decision Processes [23, 7]. Our approach sacrifices the elegance of a probabilistic framework for an implemented system capable of tackling practical problems.

Robotics researchers have also addressed the problem of planning with actions whose effects are uncertain, exploring combinations of sensory actions and compliant motion to gain information [27, 11, 4, 8]. Recent complaints about "Sensor abuse" [28, 30] suggest that the robotics community is aware of the high cost of sensing and is interested in techniques for eliminating redundant sensing.

XII's search space can be understood in terms of Kambhampati's Universal Classical Planner (UCP) [21], which unifies plan-space planning with forward-chaining and backward-chaining planning. When XII works on open conditions or threats, it is following a version of the Refine-plan-plan-space algorithm with bookkeeping constraints and conflict resolution. When XII executes an action, it is following the Refine-plan-forward-state-space algorithm, though XII actually executes the action and obtains sensory information, whereas Kambhampati's algorithm only modifies the plan. Another difference is that, to preserve completeness, Kambhampati's algorithm considers adding and "executing" all applicable new actions, whereas XII will only execute actions that were previously added to the plan.

An approach to planning similar in spirit to XII's use of LCW is to count the number of relevant ground propositions in the model, before inserting information-gathering actions into the plan, to check whether the desired information is already known [44, 35]. However, this heuristic is only effective when the number of sought-after facts is known in advance. LCW reasoning is more general in that it can also deal with cases when the size of a set (e.g. the number of files in a directory) is unknown.

5.3 Future Work

XII shares the plight of other expressive AI systems in being highly dependent on well-crafted search control knowledge. We are investigating domain-independent search control strategies, and a more intuitive language for supplying domain-dependent search control knowledge. We also hope to apply speedup learning techniques to automatically acquire much of this knowledge. Learning search control in the context of incomplete information and execution poses some interesting new research questions.

One of the key restrictions of XII is that it assumes there are no exogenous events. We can partially relax that assumption by associating expiration times with beliefs, so beliefs about rapidly changing propositions don't persist, and by recovering from errors that result from incorrect information about action preconditions. However, these techniques are fairly primitive, and much work remains to be done in learning appropriate expiration times and in revising beliefs when assumptions are found to be valid.

A XIIL Action Language

The key to XII's expressiveness is the XIIL language, which merges UWL with the subset of ADL supported by UCPOP. Table 1 provides an EBNF representation of the XIIL language. Some of the less important details of XIIL are omitted from the table and the following discussion, for the sake of clarity. Below, we elaborate on some of the unique features of the language.

A.1 Variables and Types

XIIL variables are considerably more complex than variables in UCPOP and other planners. They are always typed, as are constants. Additionally, variables may be designated as *plan-time* or *run-time*, and may be universally or existentially quantified, or parameters.[9] Run-time variables are variables whose values will not be known until an action is executed; plan-time variables are the standard variables used in classical planners. Plan-time variables appearing as parameters of an action schema must be bound before the action can be executed. Run-time variables that appear as parameters are treated as existentially quantified. Run-time variables may be bound or unbound prior to execution, but if a run-time variable is bound, and sensing reveals the correct value to be different than the expected value, then the plan will be rejected. This is a departure from [14], where run-time variables are treated as constants, and cannot be bound prior to execution.

Plan-time or run-time variables may appear in quantified expressions, and will be labeled as universally or existentially quantified. In addition to a type, universally quantified variables must have a specified *universe*, a conjunctive expression that designates the set of values over which the variable ranges. The universe is used to compute LCW preconditions and effects, and to restrict the universal base. Formally, the universe is equivalent to a precondition introduced by the **when** clause of a universally quantified effect, or to a material conditional (univ $(x) \Rightarrow$ goal(x)) in a universally quantified goal, but the universe can be used more efficiently than can additional preconditions, since it is restricted to conjunctions and since it is often unnecessary to explicitly subgoal on the

[9]Parameters facilitate a *lifted* representation, in which one action schema, such as MOVE(x, y), stands in for many specific actions, such as MOVE(A, TABLE) In planners such as SNLP, which don't support quantification, all variables in an action schema are parameters.

action-schema	::=	`(defaction` *name* `([`*parmlist*`])`
		`[precond:` *GD*`]`
		`effect:` *effect*`)`
		`execute:` *function-symbol*`(`*arglist*`)`
		`sense:` {*rtvlist*} `:=` *function-symbol*`(`*arglist*`)`

effect	::=	`cause (`*literal*`)` | `observe (`*literal*`)`
effect	::=	*effect* ∧ *effect* | `(`*effect*`)` | `when (`*GD*`)` *effect*
effect	::=	∀ `(`*uvarlist*`)` *effect* | ∃ `(`*evarlist*`)` *effect*
GD	::=	`satisfy (`*literal*`)` | `find-out (`*literal*`)` | `hands-off (`*literal*`)` | `contemplate (`*literal*`)` |
GD	::=	*GD* ∧ *GD* | *GD* ∨ *GD* | *(GD)* | ¬*GD*
GD	::=	*GD* ⇒ *GD* | *vc* `=` *vc* | *vc* ≠ *vc*
GD	::=	∀ `(`*uvarlist*`)` *GD* | ∃ `(`*gvarlist*`)` *GD*
literal	::=	*predicate-symbol* `([`*arglist*`])`, `[`*truth-value*`]`
truth-value	::=	`T` | `F` | `U` | *var*
vc	::=	*var* | *constant-symbol*
var	::=	*ptvar* | *rtvar*
ptvar	::=	*variable-symbol*
rtvar	::=	*!variable-symbol*
uvar	::=	*type-symbol* `(`*var*`)` ∈ *scope*
scope	::=	*predicate-symbol* `(`*arglist*`)` | *scope* ∧ *scope*
uvarlist	::=	*uvar* | *uvar*, *uvarlist*
parmlist	::=	*var* | *var*, *parmlist*
rtvlist	::=	*rtvar* | *rtvar*, *rtvlist*
arglist	::=	*vc* | *vc*, *arglist*
evarlist	::=	*type-symbol* `(`*rtvar*`)` | *type-symbol* `(`*rtvar*`)`, *evarlist*
gvarlist	::=	*type-symbol* `(`*ptvar*`)` | *type-symbol* `(`*ptvar*`)`, *gvarlist*

Table 1: EBNF specification of XIIL.

universe. A universe may be specified as well for existentially quantified variables, but it is not required.

Variables, constants, and the parameters of predicates and action schemas, are all typed, employing a simple, tree-structured type system. Types are formally equivalent to unary predicates, and are implemented as such in UCPOP. XII treats types differently, using them to restrict options during unification, rather than explicitly subgoaling on them. Each type has a single parent type, except for the root type, `thing`, which has no parent. In general, two variables are consistent if one is an ancestor of the other in the inheritance tree. XII performs type inference and type-checking during unification and belief update. These operations are fast. In the worst case, they are linear in the number of types, but given a bushy inheritance tree, they are logarithmic. In addition to providing a useful mechanism for checking common errors in goals and action schemas, this type system can substantially speed up planning, by cutting down on the number of effects that unify with a given goal.

A.2 Action Schemas

An action schema is a template for an action. Each schema definition starts with a name and a parameter list. Every variable appearing in the schema must be declared either in the parameter list, or within a quantifier. An action is simply a copy of a schema, with all the variables replaced with unique copies.

As Figure 1 illustrates, a XIIL action schema also contains the following fields:

`precond` The preconditions of the action. Preconditions consist of *alits* and *codesignation constraints*, connected by conjunction, disjunction, and quantifiers. The alits may have annotations **find-out** or **satisfy**, introduced in section 2.3. There are other annotations, such as **hands-off** [14], which means the truth value of the literal is not to be modified, and **contemplate**, which means only the agent's beliefs about the literal are to be queried. Codesignation constraints are of the form $x = y$, or $x \neq y$, where x and y are variables or constants appearing in the schema. Preconditions cannot contain run-time variables.

`effect` The effects, or postconditions, of the action. Postconditions consist of *alits*, connected by conjunction and quantifiers. Postconditions may themselves have preconditions, designated by the keyword `when`. In this case, the postconditions are called *conditional effects*. Such effects apply only when the associated precondition is true. The preconditions of effects are of the same form as the preconditions of actions. The alits in an effect may have annotations of **cause** or **observe**. When the **observe** annotation is used, the variables may be run-time.

`execute` This specifies what the agent needs to do in order to realize the effects of the action. In the case of `ls`, the agent must send the string `"ls -a"` to a UNIX shell, by means of the procedure `execute-unix-command`.

`sense` This specifies how the agent is to interpret any results it gets back from execution. The values to the left of the `:=` are the run-time variables in the action to be bound to values after execution. All run-time variables must be listed. The function to the right of the `:=` computes the bindings for the variables from the output returned after execution and any additional arguments passed to the function. In the case of `ls`, the pathname of the current directory is passed as an argument to `ls-sense`, in order to compute the pathnames of files in that directory.

B Algorithm

Algorithm: XII($\langle \mathcal{A}, \mathcal{O}, \mathcal{B}, \mathcal{L}, \mathcal{E} \rangle$, $\mathcal{G}$)
Until $\mathcal{G} = \emptyset \wedge \mathcal{E} = \emptyset \wedge \forall \ell \in \mathcal{L}$ ℓ not threatened, do one of

1. HandleGoal($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{L}$, $\mathcal{G}$, $\mathcal{D}$)
2. HandleThreats($\mathcal{O}$, $\mathcal{B}$, $\mathcal{L}$, $\mathcal{G}$)
3. HandleExecution($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{L}$, $\mathcal{E}$)

B.1 Open Goals

Procedure: HandleGoal($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{L}$, $\mathcal{G}$, $\mathcal{D}$)
if $\mathcal{G} \neq \emptyset$ then pop <g, S_{cons}, Context> from $\mathcal{G}$ and select case:

1. g $= g_1 \wedge g_2 \wedge \ldots \wedge g_n$: Add all g_1, g_2, ..., g_n to $\mathcal{G}$
2. g $= g_1 \wedge g_2 \wedge \ldots \wedge g_n$: Nondeterministically add one of g_1, g_2, ..., g_n to $\mathcal{G}$
3. otherwise: If Context $\models$ g then g is trivially satisfied.
   Else nondeterministically choose
   (a) Choose an action S_{prod} from $\mathcal{D}$ (new action), or from $\mathcal{A}$ (existing action), and an effect of S_{prod}, e, such that MGU(e, g) $\neq \perp$. If selecting a new action S_{prod}, add S_{prod} to $\mathcal{A}$. Add new link $S_{prod} \xrightarrow{g} S_{cons}$ to $\mathcal{L}$, $S_{prod} \prec S_{cons}$ to $\mathcal{O}$, MGU(e, g) to $\mathcal{B}$. If S_{prod} is new, add preconditions of S_{prod} to $\mathcal{G}$. If e is conditional, add its precondition to $\mathcal{G}$.
   (b) if g $= (\forall\ x \in \text{scope}(x)\ \text{P}(x))$:
   - Support universal base: Add <g, S_{cons}, Context> back to $\mathcal{G}$, tagged to WAIT. Add <LCW(scope (x) $\wedge$ Context), S_{cons}, $\emptyset$> to $\mathcal{G}$
   - Expand universal goal: if g tagged to WAIT and LCW obtained, expand goal by replacing universal variable with each instance in the universal base that is consistent with Context.

   (c) Partition:[10] if g $= (\forall\ x \in \text{scope}(x)\ \text{P}(x))$:
   Nondeterministically choose some expression Π.
   Add <g, S_{cons}, Context $\wedge\ \Pi$> to $\mathcal{G}$.
   Add <g, S_{cons}, Context $\wedge\ \neg\Pi$> to $\mathcal{G}$.

B.2 Threats

Procedure: HandleThreats($\mathcal{O}$, $\mathcal{B}$, $\mathcal{L}$, $\mathcal{G}$)
if $\ell \in \mathcal{L}$ is threatened by S_{threat}then either

1. Promote: Add $S_{cons} \prec S_{threat}$ to $\mathcal{O}$.
2. Demote: Add $S_{threat} \prec S_{prod}$ to $\mathcal{O}$.
3. Confront: If threatening effect has a precondition, add its negation to $\mathcal{G}$.

[10] The choice of the partition, Π is obviously crucial to avoid arbitrary blowup of the search space. Fortunately, the number of viable candidates for partitioning is generally small, and can be obtained by a search through the set of $\forall$ effects that partially satisfy the goal.

4. Enlarge LCW.
5. Shrink LCW.
6. Protect forall.

B.3 Execution

Procedure: HandleExecution($\mathcal{A}$, $\mathcal{O}$, $\mathcal{B}$, $\mathcal{L}$, $\mathcal{E}$)
if $A_E \in \mathcal{A}$ is executable and $\mathcal{E}(A_E) =$ `unexecuted` then execute it. A_E is executable if:

- All preconditions of A_E are satisfied
- All actions necessarily before A_E have been executed
- No pending action has an effect that clobbers any effect of A_E.
- A_E does not threaten any links
- All parameters are bound.

To execute A_E:

- Add $A_l | A_E$ to $\mathcal{O}$, where A_l is the last executed action in the plan, or A_0, if no actions have been executed.
- Execute the procedure associated with the action
- Set $\mathcal{E}(A_E) =$ `executed`
- Update the model
- Update the plan with effects asserted in model.
- If execution fails or bindings inconsistent, then backtrack.

References

[1] J. Allen. Planning as temporal reasoning. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 3–14, 1991.

[2] J Ambros-Ingerson and S. Steel. Integrating planning, execution, and monitoring. In *Proc. 7th Nat. Conf. on A.I.*, pages 735–740, 1988.

[3] A. Barrett, K. Golden, J.S. Penberthy, and D. Weld. UCPOP user's manual, (version 2.0). Technical Report 93-09-06, University of Washington, Department of Computer Science and Engineering, September 1993. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[4] R. Brost. Automatic grasp planning in the presence of uncertainty. *International Journal of Robotics Research*, 7(1):3–17, February 1988.

[5] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[6] E. Davis. *Representations of Commonsense Knowledge.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[7] T. Dean, L. P. Kaelbling, J. Kirman, and A. Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76, 1995.

[8] B. Donald. A geometric approach to error detection and recovery for robot motion planning with uncertainty. *Artificial Intelligence*, 37:223–271, 1988 1988.

[9] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. In *Proc. 2nd Intl. Conf. on A.I. Planning Systems*, June 1994.

[10] M. Drummond. A representation of action and belief for automatic planning systems. In M. Georgeff and A. Lansky, editors, *Reasoning about Actions and Plans*. Morgan Kaufmann, 1986. Proceedings of the 1986 Timberline Workshop.

[11] M. Erdmann. On Motion Planning with Uncertainty. AI-TR-810, MIT AI LAB, August 1984.

[12] O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*. (To appear).

[13] O. Etzioni, K. Golden, and D. Weld. Tractable closed-world reasoning with updates. In *Proc. 4th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 178–189, San Francisco, CA, June 1994. Morgan Kaufmann.

[14] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An Approach to Planning with Incomplete Information. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, San Francisco, CA, October 1992. Morgan Kaufmann. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[15] O. Etzioni and D. Weld. A softbot-based interface to the internet. *CACM*, 37(7):72–76, July 1994. See `http://www.cs.washington.edu/research/softbots`.

[16] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4), 1971.

[17] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[18] M. Ginsberg, editor. *Readings in Nonmonotonic Reasoning*. Morgan Kaufmann, San Mateo, CA, 1987.

[19] Leslie Pack Kaelbling. Goals as parallel program specifications. In *Proc. 7th Nat. Conf. on A.I.* Morgan Kaufmann, 1988.

[20] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence*, 76:167–238, 1995.

[21] S. Kambhampati and B. Srivastava. Universal classical planner: An algorithm for unifying state-space and plan-space planning. In *Proc. 2nd European Planning Workshop*, 1995.

[22] C. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *Proc. 15th Int. Joint Conf. on A.I.*, pages 1686–1693, 1995.

[23] S. Koenig. Optimal probabilistic and decision-theoretic planning using markovian decision theory. UCB/CSD 92/685, Berkeley, May 1992.

[24] K. Krebsbach, D. Olawsky, and M. Gini. An empirical study of sensing and defaulting in planning. In *Proc. 1st Intl. Conf. on A.I. Planning Systems*, pages 136–144, June 1992.

[25] N. Kushmerick, S. Hanks, and D. Weld. An Algorithm for Probabilistic Planning. *Artificial Intelligence*, 76:239–286, 1995.

[26] C. Kwok and D. Weld. Planning to gather information. Technical Report 96-01-04, University of Washington, Department of Computer Science and Engineering, January 1996. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[27] T. Lozano-Perez, M. Mason, and R. Taylor. Automatic synthesis of fine motion strategies for robots. *International Journal of Robotics Research*, 3(1):3–24, Spring 1984.

[28] M. Mason. Kicking the sensing habit. *AI Magazine*, 14(1):58–59, Spring 1993.

[29] 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.

[30] D. Miller. A twelve-step program to more efficient robotics. *AI Magazine*, 14(1):60–63, Spring 1993.

[31] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989. Available as technical report CMU-CS-89-103.

[32] R. Moore. A Formal Theory of Knowledge and Action. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World*. Ablex, Norwood, NJ, 1985.

[33] Leora Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of IJCAI-87*, pages 867–874, 1987.

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

[35] 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.

[36] E. Pednault. *Toward a Mathematical Theory of Plan Synthesis*. PhD thesis, Stanford University, December 1986.

[37] E. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[38] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 103–114, October 1992. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[39] J.S. Penberthy and D. Weld. Temporal planning with continuous change. In *Proc. 12th Nat. Conf. on A.I.*, July 1994.

[40] M. Peot and D. Smith. Conditional Nonlinear Planning. In *Proc. 1st Intl. Conf. on A.I. Planning Systems*, pages 189–197, June 1992.

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

[42] R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978. Reprinted in [18].

[43] M. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of IJCAI-87*, pages 1039–1046, August 1987.

[44] D. Smith. Finding all of the solutions to a problem. In *Proc. 3rd Nat. Conf. on A.I.*, pages 373–377, 1983.

[45] A. Tate. Generating project networks. In *Proc. 5th Int. Joint Conf. on A.I.*, pages 888–893, 1977.

[46] D. Warren. Generating Conditional Plans and Programs. In *Proceedings of AISB Summer Conference*, pages 344–354, University of Edinburgh, 1976.

[47] D. Weld. An introduction to least-commitment planning. *AI Magazine*, pages 27–61, Winter 1994. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.