To appear in KR '96

Representing Sensing Actions: The Middle Ground Revisited

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

Abstract

To build effective planning systems, it is crucial to find the right level of representation: too impoverished, and important actions and goals are impossible to express; too expressive, and planning becomes intractable. Within the classical framework, Pednault's ADL [24] provided a happy compromise between the impoverished STRIPS representation and the expensive situation calculus.

Among languages handling sensing actions and information goals, there is a similar spectrum of expressiveness. UWL, an extension of STRIPS, can't express goals like "Rename the file `paper.tex` to `kr.tex`." Nor can it represent universally quantified goals or effects. At the other extreme are elegant languages [22, 21, 17] for which effective planners do not exist.

In this paper, we combine elements of UWL and ADL, to define SADL: a middle-ground representation for sensing actions. Underlying our language are two insights, missing from UWL: 1) Knowledge goals are inherently temporal. 2) Knowledge preconditions are unnecessary for an important class of domains (those obeying a Markov property). SADL is expressive enough to encode the rich domain theory of the Internet Softbot, including hundreds of UNIX and Internet operators; yet it supports tractable inference by planners such as XII [11, 10].

*Many thanks to Mark Boddy, Bob Doorenbos, Oren Etzioni, Marc Friedman, Robert Goldman, Neal Lesh, Greg Linden, Mike Perkowitz, Rich Segal, Jonathan Shakes and Ellen Spertus for helpful comments. This research was funded in part by Office of Naval Research Grant N00014-94-1-0060, by National Science Foundation Grant IRI-9303461, by ARPA / Rome Labs grant F30602-95-1-0024, by a gift from Rockwell International Palo Alto Research, and by a Microsoft Graduate Fellowship

1 INTRODUCTION

One of the stumbling blocks to past research in planning with incomplete information has been inadequate or imprecisely defined languages for representing information goals and sensing actions. Many researchers have devised formalisms for reasoning about knowledge and action [21, 22, 23, 5, 3, 32, 17], but those languages are too expressive to be used in practical planning algorithms. UWL [9] offered a more tractable representation (based on STRIPS) that was tailored to current planning technology, but as Levesque [17] observes, the semantics of UWL are unclear — the definitions were made relative to a specific planning algorithm. In our efforts to define a semantics for UWL, we determined that UWL confused information goals with maintenance goals, and conflated knowledge goals with knowledge preconditions. Furthermore, years of experience with UWL convinced us that it wasn't expressive enough to fully handle the real-world domains (*e.g.*, UNIX and the Internet) for which it was intended. Since UWL didn't support universal quantification or conditional effects, it could not correctly represent the UNIX command `ls`, which lists all files in a directory, or `rm *`, which deletes all writable files.

Information	Expressiveness ⟶		
Complete	STRIPS	ADL	Situation Calculus
Incomplete	UWL	**SADL**	Moore *et al*

In this paper, we define a new action representation language, SADL,[1] that combines ideas from UWL with those from Pednault's ADL [26, 24]. Just as ADL marked the "middle ground" on the tractability spectrum between STRIPS and the situation calculus, SADL offers an advantageous combination of expressiveness and efficiency. Since SADL supports universally quantified information goals and universally quantified, conditional, observational effects, it is expressive enough to represent hundreds of UNIX and Internet commands.

[1] SADL (pronounced "Saddle") stands for "Sensory Action Description Language."

Indeed, four years of painful experience writing and debugging the Internet Softbot [7] knowledge base forced us to uncover and remedy some subtle confusions about information goals:

- In a dynamic world, knowledge goals are inherently *temporal* — If proposition P is true at one time point and false in another, which time point do we mean when we ask about P's truth value? Since UWL has limited provision to make temporal distinctions, it cannot encode an important class of goals. In particular, UWL cannot express goals that require causal change to attributes used to designate objects, *e.g.*"Rename the file `paper.tex` to `kr.tex`." (See Sections 2.2 and 2.3 for the SADL solution)
- We identify a large class of domains, called *Markov domains*, and argue that actions in these domains are best encoded *without knowledge preconditions*. The multiagent scenarios that inspired Moore, Morgenstern, and others are not Markov, but UNIX and much of the Internet are. While SADL discourages knowledge *preconditions* it recognizes the need for knowledge *subgoals*. (Section 2.4 elaborates).

1.1 ROADMAP

Section 2 describes problems with the UWL formulation of knowledge goals and presents the SADL solution. In Section 3 we discuss observational effects of actions, and causal effects, which can decrease the agent's knowledge about the world. We also demonstrate the representational adequacy of SADL by presenting an encoding of the UNIX `ls -a` command. In Section 4 we discuss temporal projection in SADL. In Section 5 we demonstrate that the SADL formalism is expressive enough to represent many interesting actions. Section 6 argues that SADL's expressive power comes at a reasonable price — reasoning is tractable. We conclude with a discussion of related work in Section 7 and a summary in Section 8.

2 KNOWLEDGE GOALS AND PRECONDITIONS

In UWL, preconditions and goals were limited to conjunctions of literals, each annotated with one of three tags: **satisfy**, **hands-off**, and **find-out**. The SADL action language is based on UWL, but uses a different set of annotations: **satisfy**, **hands-off**, and **initially**, which provide a cleaner semantics for information goals and greater expressive power; additionally, SADL uses unannotated literals to designate preconditions that don't depend on the agent's knowledge. Furthermore, SADL supports universal quantification and conditional effects, both of which have interesting ramifications in the context of incomplete information. We proceed by reviewing UWL, uncovering some confusions, presenting the SADL solution, and sketching the formal semantics.

In UWL (and in SADL) individual literals have truth values expressed in a three-valued logic: `T`, `F`, `U` (unknown). Free variables are implicitly existentially quantified, and the quantifier takes the widest possible scope.[2] For example, **satisfy**(in.dir (f, `tex`), `T`)[3] means "Ensure that there's at least one file in directory `tex`." Truth values can also be represented by variables. For example, **satisfy**(in.dir (`myfile`, `tex`), tv) means "Find out whether or not `myfile` is in `tex`."

Although the semantics of UWL was defined procedurally [9], we provide SADL's semantics in terms of the situation calculus. The situation calculus [19] is a first-order logic used to capture changes to the world that come about by the execution of actions. A *fluent* is a proposition whose truth value changes over time. Every fluent, $\varphi(x)$, takes an additional argument, namely a situation, s. $\varphi(x, s)$ represents the statement that $\varphi(x)$ holds in situation s. By convention, s is always the last argument of φ, so we will freely add or drop the s, depending on whether we are referring to φ in a particular situation. Thus, if $\text{in.dir}(f, d)$ means file f is in directory d, $\text{in.dir}(f, d, s)$ means this fact holds in situation s. All state changes are assumed to result from the execution of actions. The special function DO is used to describe these changes: $\text{DO}(a, s)$ returns the situation resulting from executing action a in situation s. We use $\{a\}_1^n$ to represent the sequence of actions $a_1; a_2; \ldots; a_n$. $\text{DO}(\{a\}_1^n, s)$ denotes nested application $\text{DO}(a_n, \text{DO}(a_{n-1}, \ldots, \text{DO}(a_1, s)))$, *i.e.*, the result of executing the entire sequence, starting in situation s. We use s_n as a shorthand for $\text{DO}(\{a\}_1^n, s_0)$.

Our formulation of SADL is based on Scherl and Levesque's [32] solution to the frame problem for knowledge-producing actions. We adopt their completeness assumptions, and their formulation of incomplete knowledge, and thus their results (i.e. the persistence of knowledge and of ignorance) hold for us as well. Incomplete knowledge is defined in terms of the standard possible-worlds semantics, where $\text{K}(s', s)$ means that if the situation is s, then it is consistent with the agent's knowledge to believe that the situation could in fact be s'. In other words, $\{s' | \text{K}(s', s)\}$ denotes the set of all possible worlds consistent with the agent's knowledge in situation s. We assume that an agent's knowledge is correct, so the actual situation is always considered possible by the agent ($\forall s.\text{K}(s, s)$), and we assume that situations only change when the agent executes an action. We define $\text{KNOW}(\varphi, s) \stackrel{\text{def}}{=} \forall s'.\text{K}(s', s) \Rightarrow \varphi(s')$, *i.e.*, φ is true in all worlds consistent with the agent's knowledge.

[2] Explicit quantifiers can be used to indicate a narrower scope.

[3] For notational convenience, an omitted truth value defaults to `T`, so this could be rewritten as **satisfy**(in.dir (f, `tex`)). We use this shorthand in the remainder of the paper. Italicized lower-case symbols, such as f, denote variables. Symbols in `typewriter` font denote constants. Annotations are in **bold**.

As we mentioned, SADL uses a three-valued logic (T, F, U) to represent knowledge. The relation between these truth values and KNOW is straightforward. If φ has the truth value T, then KNOW(φ). If φ has the truth value F, then KNOW($\neg\varphi$). If the truth value is U, then $\neg$KNOW(φ) $\wedge$ $\neg$KNOW($\neg\varphi$).

2.1 SATISFACTION AND MAINTENANCE GOALS

The goal **satisfy**(P) indicates a traditional goal (as in ADL): achieve P by whatever means possible. In the presence of incomplete information, we make the further requirement that the agent knows that P is true. We define GOAL($G, s_0, \{a\}_1^n$) to mean that goal G is achieved in the situation resulting from executing plan $\{a\}_1^n$ in situation s_0; since we assume the agent's knowledge is correct, it is sufficient to state that the agent knows P:

$$\text{GOAL}(\textbf{satisfy}(P, \texttt{T}), s_0, \{a\}_1^n) \stackrel{\text{def}}{=} \text{KNOW}(P, s_n) \quad (1)$$

$$\text{GOAL}(\textbf{satisfy}(P, \texttt{F}), s_0, \{a\}_1^n) \stackrel{\text{def}}{=} \text{KNOW}(\neg P, s_n) \quad (2)$$

$$\text{GOAL}(\textbf{satisfy}(P, tv), s_0, \{a\}_1^n) \stackrel{\text{def}}{=} \begin{array}{l}\text{KNOW}(P, s_n) \vee \\ \text{KNOW}(\neg P, s_n)\end{array} \quad (3)$$

Note that when given an (existentially quantified) variable as truth value, a **satisfy** goal requires that the agent learn whether the proposition is true or false (which could be achieved by making it true or false). Equation 3 is a slight simplification; if several fluents in a goal use the *same* variable, tv, then they should all have the same truth value. The above definition fails to capture such correlations. We don't discuss correlated truth values in this paper, so for clarity, we omit these variable constraints in the remainder of the paper. However, we show them below for **satisfy** goals. Variable constraints in the other definitions follow the same form:

$$\text{GOAL}(\textbf{satisfy}(P, tv), s_0, \{a\}_1^n) \stackrel{\text{def}}{=}$$
$$\text{KNOW}(P \wedge tv = \texttt{T}, s_n) \vee \text{KNOW}(\neg P \wedge tv = \texttt{F}, s_n) \quad (4)$$

The **hands-off** annotation indicates a maintenance goal that prohibits the agent from changing the fluent in question.

$$\text{GOAL}(\textbf{hands-off}(P), s_0, \{a\}_1^n) \stackrel{\text{def}}{=}$$
$$\forall s \in \text{State-History}.[P(s) \Leftrightarrow P(s_0)] \quad (5)$$

By State-History we mean the set of $n+1$ situations produced during execution of DO($\{a\}_1^n, s_0$) (including both s_0 and s_n). Thus, the definition of **hands-off** requires that P not change value during execution of the plan. Etzioni *et al* [9] noted that together, **satisfy**+**hands-off** can be used to indicate a "look but don't touch" goal: the agent may sense the fluent's value, but is forbidden to change it. While **hands-off** goals are clearly useful, we argue that they are an overly restrictive way of specifying *knowledge* goals. In particular, they outlaw changing the value of a fluent after it has been sensed.

2.2 KNOWLEDGE GOALS ARE INHERENTLY TEMPORAL

Before explaining the SADL approach to knowledge goals, we discuss the UWL **find-out** annotation. **find-out** is problematic because the original definition was in terms of a particular planning algorithm [9]. The motivation for **find-out** was the existence of goals for which **hands-off** is too restrictive, but **satisfy** alone is too permissive. For example, given the goal "Tell me what files are in directory `tex`," executing `rm tex/*` and reporting "None" would clearly be inappropriate. But what about the conjunctive goal "Free up some disk space and tell me what files are in directory `tex`"? In this case *excluding* the `rm` seems inappropriate, since it may be necessary in service of freeing disk space. Yet the knowledge that the directory is now empty is relevant to the information goal. Proponents of **find-out** argued that `rm` was unacceptable for the first goal, but acceptable in service of the conjunction [9]. We contend that this definition is unclear and unacceptable; a plan that satisfies the conjunction $A \wedge B$ should also be a solution to A.

While the examples used to justify the original **find-out** definition are evocative, their persuasive powers stem from *ambiguity*. At what *time point* do we wish to know the directory contents? Before freeing disk space, afterward, or in between? Since fluents are always changing, a general information goal requires two temporal arguments: the time a fluent is sensed, and the time the sensed value is to be reported. *E.g.*, one can ask "Who was president in 1883," or "Tell me tomorrow who was president today."

Since planning with an explicit temporal representation is slow, our quest for the "middle ground" along the expressiveness / tractability spectrum demands a minimal notion of time that captures most common goals. We limit consideration to two time points: the time when a goal is given to the agent, and the time the agent gives his reply. Note that **satisfy**(P, tv) (Equation 3) allows one to specify the goal of knowing P's truth value at this latter time point. To specify the goal of sensing a fluent at the time the goal is given, we introduce the annotation **initially**.

$$\text{GOAL}(\textbf{initially}(P, tv), s_0, \{a\}_1^n) \stackrel{\text{def}}{=}$$
$$[\forall s \in \text{ORIG}_n.\ P(s)] \vee [\forall s \in \text{ORIG}_n.\ \neg P(s)] \quad (6)$$

We use ORIG$_n$ (Figure 1) to represent the agent's knowledge in s_n *about the past situation* s_0, *i.e.*, the set of situations indistinguishable from s_0 after execution of the plan: ORIG$_n$ = $\{s \mid \text{K}(\text{DO}(\{a\}_1^n, s), \text{DO}(\{a\}_1^n, s_0))\}$. Thus the definition of **initially** states that when the agent has finished executing the plan, he will know whether P was true or false when he started. **initially**(P) is not achievable by an action that changes the fluent P, since such an action only obscures the initial value of P. However, changing P after determining its initial value

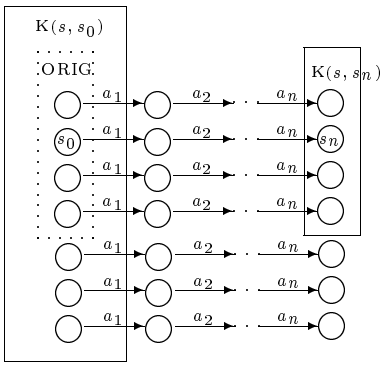

Figure 1: The region surrounded by dotted lines represents the set ORIG_n, the set of states indistinguishable from s_0, based on the agent's knowledge in state s_n. ORIG_n is a subset of $\{s \mid \text{K}(s, s_0)\}$, the states that were consistent with the agent's knowledge in s_0, since the agent has learned more about what originally held, but has not forgotten anything it knew originally.

is fine. By combining **initially** with **satisfy** we can express "tidiness" goals (modify P at will, but restore its initial value by plan's end) [35]. Furthermore, we can express goals such as "Find the the file currently named `paper.tex`, and rename it to `kr.tex`," which are impossible to express in UWL. Since UWL can't make temporal distinctions, there is no way to ask for the past value of a fluent without also requiring that the fluent have the same value when the reply is given, so any goal of the form "Find some x such that $P(x)$, and make $P(x)$ false" is inexpressible in UWL.

2.3 UNIVERSALLY QUANTIFIED GOALS

When defining universally quantified goals, one must again be specific with respect to time points: does the designator specifying the Herbrand universe refer to s_0 or s_n? Since SADL allows an arbitrary goal description to be used to scope a universally quantified goal, one can specify a wide range of requests. For example, suppose an agent is given the goal of seeing to it that all files in directory `tex` are compressed. What plans satisfy the goal? It depends on what the request really means. In SADL, one can write one of the following precise versions, thus eliminating the ambiguity.

1. Ensure that all files, which were initially in `tex`, end up being compressed: $\forall f$ **initially**(in.dir (f, `tex`)) $\Rightarrow$ **satisfy**(compressed (f)). Executing `compress tex/*` solves this goal, as does executing `mv tex/* temp` then `compress temp/*`.
2. Ensure that all files, which end up in `tex`, end up being compressed: $\forall f$ **satisfy**(in.dir (f, `tex`)) $\Rightarrow$ **satisfy**(compressed (f)). Executing `compress tex/*` solves this goal, but so does `rm tex/*`!
3. Determine if all files, initially in `tex`, were initially compressed: $\forall f$ **initially**(in.dir (f, `tex`)) $\Rightarrow$ **initially**(compressed (f)).
4. Determine if all files, in `tex` at the end of execution, were initially compressed: $\forall f$ **satisfy**(in.dir (f, `tex`)) $\Rightarrow$ **initially**(compressed (f)). This is equivalent to $\forall f$ **initially**(compressed (f), `F`) $\Rightarrow$ **satisfy**(in.dir (f, `tex`), `F`), *i.e.* ensure that all files *not* initially compressed do *not* end up in `tex`.

The first example seems the most likely interpretation of the goal in this case, but it still leaves something to be desired, since the user may not want the files moved from `tex`. We can easily state the additional requirement that the files not be moved (**hands-off**(in.dir (f, `tex`))), or that they be returned to `tex` by the end (**satisfy**(in.dir (f, `tex`))). We should be careful not to make goals overly restrictive, though. If the desire is that the agent should fail if there's no way to compress the files without moving them, then adding such restrictions is correct. If the desire is merely that the agent should avoid moving the files unnecessarily, then we want the original solution, with some background preference to minimize unnecessary changes. Such background preferences could be expressed in terms of a utility function over world states [30], a measure of plan quality [28, 36], or an explicit notion of harm [35].

Note that even if we decide to forbid moving the files from `tex`, there are still other actions, such as deleting all the files in `important/papers`, or sending threatening email to `president@whitehouse.gov` that haven't been excluded. This is a general problem with satisficing plans: anything goes as long as the goal is achieved. Specifying all the undesired outcomes with every goal would be tedious and error-prone. A better solution is to separate the criteria of goal satisfaction from background preferences, as is done in [37, 13, 35].

Given the appropriate annotations on fluents, which provide temporal information, the semantics of $\forall$ goals is straightforward:

$$\text{GOAL}(\forall \vec{x}.P, s_0, \{a\}_1^n) \stackrel{\text{def}}{=} \forall \vec{x}.\text{GOAL}(P, s_0, \{a\}_1^n) \quad (7)$$

$$\text{GOAL}(P \Rightarrow Q, s_0, \{a\}_1^n) \stackrel{\text{def}}{=} \begin{array}{l}\text{GOAL}(P, s_0, \{a\}_1^n) \Rightarrow \\ \text{GOAL}(Q, s_0, \{a\}_1^n)\end{array} \quad (8)$$

Logical operators such as $\wedge$, $\vee$, and $\exists$ follow the same form as above.

2.4 KNOWLEDGE PRECONDITIONS CONSIDERED HARMFUL

Moore [21] identified two kinds of knowledge preconditions an agent must satisfy in order to execute an action in support of some proposition P: First, the agent must know a rigid designator (*i.e.*, an unambiguous, executable description) of the action. Second, the agent must know that executing the action will in fact achieve P. Subsequent work, *e.g.* [22], generalized this framework to handle scenarios where multiple agents reasoned about each other's knowledge.

In the interest of tractability, we take a much narrower view, assuming away Moore's first type of knowledge precondition and refuting the need for his second type. Our argument occupies the remainder of this section, but the summary is that there is a large class of domains, those obeying a *Markov property*, for which actions are best encoded without knowledge preconditions. While the multiagent scenarios considered by Moore and Morgenstern are not Markov, UNIX and much of the Internet are.

We start the argument by assuming away Moore's first type of knowledge precondition. We define actions as programs that can be executed by a robot or softbot, without the need for further reasoning. In this view, *all* actions are rigid designators. `dial` (combination(`safe`)) is not an admissible action, but `dial(31-24-15)` is. Lifted action schemas, *e.g.* `dial(`x`)`, are not rigid designators, but it is easy to produce one by substituting a constant for x. Thus Moore's first type of knowledge precondition vanishes.

Moore's second type of knowledge precondition presupposes that an action in a plan must provably succeed in achieving a desired goal. This is a standard assumption in classical planning, but is overly restrictive given incomplete information about the world; enforcing this assumption by adding knowledge preconditions to actions is inappropriate. For example, if knowledge of the safe's combination is a precondition of the `dial` action, then it becomes impossible for a planner to solve the goal "find out whether the combination is 31-24-15" by dialing that number, since before executing the `dial` action, it will need to satisfy that action's precondition of finding out whether 31-24-15 is the right combination![4]

On the other hand, it is often necessary for an agent to plan to obtain information, such as the combination of a safe, either to reduce search or to avoid dangerous mistakes. These knowledge *subgoals*, naturally, have a temporal component, but the only time point of interest is the moment the action is executed. For example, the goal of knowing the safe's combination could be satisfied by watching another agent open the safe, but it might also be satisfied by changing the combination to some known value (for instance, at some earlier time when the safe is open).

We say that an action is *Markov* if its effects depend only on the state of the world at the time of execution. Note that simple mechanical and software systems are naturally encoded as Markov, while multiagent systems are typically *not*, because it is useful to endow one's model of another agent with state (*i.e.*, I know that Bill knew ...). If all actions in a domain are Markov, then all knowledge sub-goals will be of the same form: 1) The agent needs to know the value of some fluent at the time the action is to be executed, and 2) it doesn't matter if the agent affects the fluent while obtaining its value.[5] These requirements for knowledge sub-goals are met by the SADL definition of **satisfy** (Equation 3),[6] if we regard the action sequence $\{a\}_1^n$ as a plan to achieve the preconditions of action a_{n+1}.

The Markov assumption for actions yields a substantially simpler representation of change than those defined by Moore and Morgenstern. While their theories are more appropriate for complex, multi-agent domains, SADL gains tractability while retaining enough expressive power to model many important domains.

3 EFFECTS

Like UWL, SADL divides effects into those that change the world, annotated by **cause**, and those that merely report on the state of the world, annotated by **observe**. Because it lacked universal quantification, UWL couldn't even correctly model UNIX `ls`. SADL goes beyond UWL by allowing both observational and causal effects to have universal quantification and secondary preconditions.

3.1 OBSERVATIONAL EFFECTS

Executing actions with observational effects assigns values to *runtime* variables that appear in those effects. By using a runtime variable as a parameter to a later action (or to control contingent execution), information gathered by one action can affect the agent's subsequent behavior. Inside an effect, runtime variables (syntactically identified with a leading an exclamation point, *e.g.* *!tv*) can appear as terms or as truth values. For example, `ping twain` has the ef-

[4] Note that eliminating the knowledge precondition from the `dial` action also allows the unhurried agent to devise a plan to enumerate the possible combinations until he finds one that works. Indeed, the Internet Softbot [7] follows an analogous strategy when directed to find a particular user, file or a web page, whose location is unknown. If `finger` and `ls` included knowledge preconditions, then the actions would be useless for locating users and files.

[5] The reader may object that (nonrigid) indexical references could appear as preconditions to actions. For example, suppose that running Netscape requires that *the* file `netscape.bookmarks` be in a given directory. It is not sufficient that *a* file of that name be there, because renaming `paper.tex` to `netscape.bookmarks` would cause Netscape to fail. But this example makes it clear that the proposed preconditions of Netscape are simply underspecified. They should be "The directory contains a file named `netscape.bookmarks`, which is a valid bookmarks file, and ..." This is just the qualification problem [18] in disguise. Granted, it will usually be impossible (or undesirable) to model all such preconditions.

[6] A justification that might be given for **initially** or **hands-off** preconditions is to minimize destructive actions used by an agent to satisfy a goal (*i.e.* don't use `mv` to find out the name of a file). We agree on the need for reasoning about *plan quality*, but an accurate theory of action should distinguish action preconditions from *user preferences*.

fect of **observe**(machine.alive(`twain`), *!tv*), *i.e.* determining whether it is true or false that the machine named `twain` is alive, and `wc myfile` has the effect **observe**(word.count(`myfile`, *!word*)), *i.e.* determining the number of words in `myfile`.

Before we define individual effects, we discuss what it means to execute an action, with all its effects. Let $\mathrm{EFF}(E, a, s)$ denote the fact that E becomes true after action a is executed in s, let π^a be the precondition of action a, and let ε^a be the effects. An action's effects will only be realized if the action is executed when its preconditions are satisfied. Furthermore, the agent always knows when it executes an action, and it knows the effects of that action. Following Moore [21]:

$$\forall s.\mathrm{GOAL}(\pi^a, s, \{\}) \Rightarrow \forall s''.[K(s'', \mathrm{DO}(a, s)) \Leftrightarrow \exists s'.\mathrm{K}(s', s) \wedge s'' = \mathrm{DO}(a, s') \wedge \mathrm{EFF}(\varepsilon^a, a, s)] \quad (9)$$

The fact that the agent knows the effects of a doesn't imply that effects are always certain. As we discuss in Section 3.3, actions with conditional effects can result in uncertainty.

We now define the semantics of **observe** in terms of primitive situation calculus expressions:

$$\mathrm{EFF}(\mathbf{observe}(P, \mathtt{T}), a, s) \stackrel{\mathrm{def}}{=} \forall s'.\mathrm{K}(s', \mathrm{DO}(a, s)) \Rightarrow \exists s_i.\mathrm{K}(s_i, s) \wedge s' = \mathrm{DO}(a, s_i) \wedge P(s_i) \quad (10)$$

$$\mathrm{EFF}(\mathbf{observe}(P, tv), a, s) \stackrel{\mathrm{def}}{=} \forall s'.\mathrm{K}(s', \mathrm{DO}(a, s)) \Rightarrow \exists s_i.\mathrm{K}(s_i, s) \wedge s' = \mathrm{DO}(a, s_i) \wedge (P(s_i) \Leftrightarrow P(s)) \quad (11)$$

In other words if action a has an **observe** effect and is executed in situation s, then in the resulting situation, the agent knows more about the value that P had in s. For example, if in s the agent observes that the sky is blue, we would say that in situation $s' = \mathrm{DO}(\mathtt{look}, s)$, the agent knows that the sky was blue in situation s. The double use of the K operator in Equations 9 and 10 is a trifle redundant given only a single observational effect. Indeed, if we assume positive introspection (*i.e.* K is transitive), as in the S4 logic, the resulting equation can be greatly simplified. However, in more complex effects, we wish to distinguish between the agent knowing that the effect as a whole took place, and knowing the value of a single fluent.

SADL supports universally quantified run-time variables. By nesting universal and existential quantifiers, SADL can model powerful sensory actions that provide several pieces of information about an unbounded number of objects. For example, `ls -a`, (Figure 2), reports several facts about each file in the current directory. The universal quantifier indicates that, at execution time, information will be provided about *all* files *!f* which are in directory *d*. Since the value of *!f* is observed, quantification uses a run-time variable. The nested existential quantifier denotes that each file has

```
action ls(d)
    precond:  satisfy(current.shell(csh)) ∧
              satisfy(protection(d, readable)) ∧
    effect:   ∀ !f when in.dir(!f, d)
                ∃ !p, !n
                  observe(in.dir(!f, d)) ∧
                  observe(pathname(!f, !p)) ∧
                  observe(name(!f, !n))
```

Figure 2: **UNIX action schema.** The SADL `ls` action (UNIX `ls -a`) to list all files in the a directory.

a *distinct* filename and pathname. The conditional **when** restricts the files sensed to those in directory *d*. The fact that the in.dir relation appears in two places may seem odd, but as we shall explain, the first use of in.dir refers to the actual situation s, whereas the second refers to the agent's knowledge (*i.e.*, all possible situations).

It is useful to note that after executing `ls -a tex`, the agent not only knows all files in `tex`; she knows that she knows all files (*i.e.*, she has closed world knowledge on the contents of `tex`). Because of the $\forall$ in the effects of `ls`, and since she knows the effects of `ls`, the agent can infer closed-world knowledge. Such inference would be costly if it were done using first-order theorem-proving in the situation calculus. We have devised efficient algorithms for doing this reasoning, which we describe in [6, 8].

The translation of $\forall$ effects into the situation calculus is straightforward (Other logical operators follow the same form):

$$\mathrm{EFF}(\forall \vec{x}.E, a, s) \stackrel{\mathrm{def}}{=} \forall \vec{x}.\mathrm{EFF}(E, a, s) \quad (12)$$

This definition of $\forall$ effects may seem anticlimactic. The magic, however, stems from the way in which **when** introduces secondary preconditions; these are required for $\forall$ effects, where the **when** clause restricts the universe of discourse to a finite set, and indicates precisely the range of the quantifier.

3.2 CONDITIONAL EFFECTS

A secondary precondition, *i.e.* one associated with an effect [26], defines the conditions under which action execution will achieve that effect. Unlike primary preconditions, secondary preconditions need not be true for the action to be executed. If p is the secondary precondition of effect e, then the resulting conditional effect is defined as:

$$\mathrm{EFF}(\mathbf{when}(p, e), a, s) \stackrel{\mathrm{def}}{=} \mathrm{GOAL}(p, s, \{\}) \Rightarrow \mathrm{EFF}(e, a, s)$$

We use GOAL in our definition of **when**, but we have only defined GOAL for annotations **satisfy**,

hands-off and **initially**. How should we define **when** preconditions? Since they need to hold, if at all, when the action is executed, they are different from **initially** preconditions. But **satisfy** requires that the agent know that the condition is true, which would lead to the faulty conclusion that the effect only occurs if the agent *knows* that the secondary preconditions hold. So we add a new type of precondition, without any annotation at all, to represent conditions that must hold at the time of execution, with or without knowledge of the agent:

$$\text{GOAL}(P, s_0, \{a\}_1^n) \stackrel{\text{def}}{=} P(s_n) \tag{13}$$

This ensures that whether the effects occur depends only on the state of the world. It also makes it clear what is being quantified over in `ls`: The files *really* in d, at the time of execution.

3.3 UNCERTAIN EFFECTS

In some cases, executing actions with causal effects can decrease the agent's knowledge about the world. SADL provides two ways of encoding these actions: as conditional effects whose secondary precondition is unknown, or by explicitly specifying the `U` truth value. As an example of the former, executing `rm tex/*` deletes all writable files in `tex`; if the agent doesn't know which files are writable, then she won't know which files remain in `tex` even if she knew the contents before executing the action. As an example of explicit creation of uncertainty, we encode `compress myfile` with the effect $\forall n$ **cause** `(size (myfile,` n`), U)`.[7] We define causal effects for `T` and `U` truth values as follows:

$$\text{EFF}(\textbf{cause}(P, \texttt{T}), a, s) \stackrel{\text{def}}{=} P(\text{DO}(a, s)) \tag{14}$$

$$\text{EFF}(\textbf{cause}(P, \texttt{U}), a, s) \stackrel{\text{def}}{=} Unk_P(a, \text{DO}(a, s)) \Leftrightarrow P(\text{DO}(a, s)) \tag{15}$$

where, Unk_P is a predicate such that

$$\neg KNOW(Unk_P(a), \text{DO}(a, s)) \wedge \neg KNOW(\neg Unk_P(a), \text{DO}(a, s)) \tag{16}$$

In other words, we represent an uncertain effect as a deterministic function of hidden state. $Unk_P(a)$ denotes a *unique* unknown predicate, which represents the hidden state responsible for the change in truth value of P. It must be unique to avoid biasing correlation of independent unknown effects.

It is clear from the above definition how a **cause** effect may make P unknown. What may not be clear is how a **cause** effect can make P known. In fact, it wouldn't, if not for the fact that the agent knows all the effects of an action (Equation 9). However, knowledge of a conditional effect does not necessarily mean knowledge of the consequent. For example, if an agent executes `compress myfile`, she only knows that *if* she had write permission prior to executing `compress`, then `myfile` is compressed afterward.

[7] In principle, we could represent all uncertain effects as conditional effects with unknown preconditions, but doing so would be cumbersome. However, we define the *semantics* of uncertain effects in precisely this manner.

4 TEMPORAL PROJECTION & REGRESSION

We have discussed the function DO, which maps a situation and an action (or sequence of actions) to a new situation, but we haven't yet said how the two situation terms relate to each other. If $s' = \text{DO}(\{a\}_1^n, s)$, we want to answer the following questions.

- Progression: What can we say about s', given knowledge of the conditions that hold in s?
- Regression: What must be true in s, to guarantee some desired condition in s'?

We treat each in turn.

4.1 PROJECTION & THE FRAME PROBLEM

The definitions for preconditions and effects that we have given are insufficient to solve the temporal projection problem. SADL effects only list fluents that an action affects, but what about fluents it doesn't affect? Explicitly stating everything that *doesn't* change would be tedious — this is the well-known frame problem. The standard approach to the frame problem, and the one we adopt, is to make the STRIPS assumption: anything not explicitly said to change remains the same. To fully specify the SADL semantics, it is necessary to express the STRIPS assumption in terms of the situation calculus. We use the formulation introduced in [31], and augmented in [32] to account for sensing actions. This strategy consists of providing a formula for each fluent, called a *successor state axiom*, that specifies the value of the fluent in terms of 1) the action executed, and 2) the conditions that held before the action was executed. By quantifying over actions, we can produce a single, concise formula for each fluent that includes only the relevant information.

Specifying update axioms for each fluent independently requires fluents to be logically independent of each other, so disjunction is not allowed. Effects consist of conjunctions of terms, each term being equivalent to one of the following

$$\textbf{when } \gamma_P^{\texttt{T}}(a) \textbf{ cause}(P, \texttt{T}) \tag{17}$$

$$\textbf{when } \gamma_P^{\texttt{F}}(a) \textbf{ cause}(P, \texttt{F}) \tag{18}$$

$$\textbf{when } \gamma_P^{\texttt{U}}(a) \textbf{ cause}(P, \texttt{U}) \tag{19}$$

$$\textbf{when } \kappa_P^{tv}(a) \textbf{ observe}(P, tv) \tag{20}$$

where a is an action and P is a fluent, which may contain universally quantified variables or constants,[8] and $\gamma_P^{tv}(a)$ and $\kappa_P^{tv}(a)$ represent arbitrary goal expressions.[9] For example, if `compress tex/*` changes the size of all writable files in directory `tex`, then $\gamma^{\mathtt{U}}_{\mathrm{size}(f)}(\texttt{compress tex/*}) = \mathrm{indir}(f, \texttt{tex}) \wedge \mathrm{writable}(f)$. Clearly, all actions can be represented by specifying the γ and κ preconditions for each fluent in the domain theory. If a has a non-conditional effect, **cause**(P, tv), then $\gamma_P^{tv}(a) = \mathtt{T}$. We can express the fact that action a doesn't affect P at all by saying $\forall tv.\gamma_P^{tv}(a) = \mathtt{F}$. We don't list **observe**$(P, \mathtt{T})$ above, since it is subsumed by the conjunction **observe**$(P, v) \wedge v = \mathtt{T}$ (similarly for `F`).

Given these definitions, we can state the conditions under which an action changes or preserves a fluent's truth value. Following Pednault [24], we define Σ^a_φ to be the conditions under which an executable action a will establish φ, and Π^a_φ to be the conditions under which a will preserve φ. We have the following establishment conditions:

$$\Sigma^a_\varphi \Leftrightarrow \gamma^{\mathtt{T}}_\varphi(a) \vee (Unk_\varphi(a) \wedge \gamma^{\mathtt{U}}_\varphi(a)) \quad (21)$$

$$\Sigma^a_{\neg\varphi} \Leftrightarrow \gamma^{\mathtt{F}}_\varphi(a) \vee (\neg Unk_\varphi(a) \wedge \gamma^{\mathtt{U}}_\varphi(a)) \quad (22)$$

where $Unk_\varphi(a)$ is the unknown predicate introduced in Equations 15 and 16. The presence of an effect with a `U` truth value will make φ true or false, depending on the value of $Unk_\varphi(a)$. Since $Unk_\varphi(a)$ is unknown by definition, effects with `U` truth values aren't generally useful for goal establishment. We also have the following preservation conditions:

$$\Pi^a_\varphi \Leftrightarrow \neg\gamma^{\mathtt{F}}_\varphi(a) \wedge (\neg Unk_\varphi(a) \vee \neg\gamma^{\mathtt{U}}_\varphi(a)) \quad (23)$$

$$\Pi^a_{\neg\varphi} \Leftrightarrow \neg\gamma^{\mathtt{T}}_\varphi(a) \wedge (Unk_\varphi(a) \vee \neg\gamma^{\mathtt{U}}_\varphi(a)) \quad (24)$$

For each fluent, we can then generate an expression that specifies precisely when it is true or false, by quantifying over actions. For each fluent P, there is a *successor state axiom*, which combines update axioms and frame axioums for P. The successor state axioms are straightforward statements of the STRIPS assumption: a fluent is true if and only if it was made true, or it was true originally and it wasn't made false:

$$\mathrm{GOAL}(\pi^a, s, \{\}) \Rightarrow [P(\mathrm{DO}(a, s)) \Leftrightarrow \Sigma^a_P(s) \vee P(s) \wedge \Pi^a_P(s)] \quad (25)$$

Similarly, there is a successor state axiom for K.

$$\mathrm{GOAL}(\pi^a, s, \{\}) \Rightarrow [K(s'', \mathrm{DO}(a, s)) \Leftrightarrow \exists s'.\mathrm{K}(s', s) \wedge (s'' = \mathrm{DO}(a, s')) \wedge \forall P.([\kappa^v_P(a, s) \wedge P(s)] \Rightarrow [P(s') \wedge v =_{s'} \mathtt{T}] \wedge ([\kappa^v_P(a, s) \wedge \neg P(s)]) \Rightarrow [\neg P(s') \wedge v =_{s'} \mathtt{F}])] \quad (26)$$

[8] Including variables that will resolve to constants.

[9] with the restriction that effects must be consistent, so, for example, $\gamma^{\mathtt{T}}_P(a) \wedge \gamma^{\mathtt{F}}_P(a)$ must always be false.

We have stated this formula in second-order logic, but only because the formula depends on all of the actual fluents in the domain theory. Given any specific domain, this second-order formula could be replaced with an equivalent first-order formula by replacing P with each fluent in the domain.

The above definition only specifies when information is gained, and seems to say nothing about when it is lost. However, information loss is indeed accounted for, through the successor state axiom for P. If P becomes true in some situations accessible from s, and false in others, then by definition, P is unknown. For example, `compress myfile` compresses `myfile` if it is writable. If it is unknown whether `myfile` is writable, then in some accessible worlds, `myfile` is writable and will be compressed. In other worlds, `myfile` is not writable and won't be compressed. The result is that it becomes unknown whether `myfile` is compressed. Similarly, if P was known previously and not changed, then by the successor state axioms for P and K, P will continue to be known. [32].

The above formula correctly describes how K changes, but it is a little unwieldy if what we want to know about is KNOW(φ). Intuitively, KNOW(φ) becomes true if φ is known to become true, or φ is observed. Additionally, φ continues to be known true until it possibly becomes false. The following formulas follow from the successor state axioms for φ and K.

$$\Pi^a_{\mathrm{KNOW}(\varphi)} \Leftrightarrow \mathrm{KNOW}(\Pi^a_\varphi) \quad (27)$$

$$\Sigma^a_{\mathrm{KNOW}(\varphi)} \Leftrightarrow KNOW(\gamma^{\mathtt{T}}_\varphi(a)) \vee (\kappa^{tv}_\varphi(a) \wedge \varphi \wedge \Pi^a_{\mathrm{KNOW}(\varphi)})^{10} \quad (28)$$

4.2 REGRESSION

Most modern planners build plans using goal regression — starting with a goal and successively adding actions that achieve either part of the goal or preconditions of previously added actions. Once no preconditions remain that aren't true in the initial state, the plan is complete. It is therefore useful to have a formal specification of what conditions must be true for a

[10] The additional requirement $\Pi^a_{\mathrm{KNOW}(\varphi)}$ may come as a surprise, since an action that simultaneously observes φ and causes φ to become false or unknown would seem to violate our rule against inconsistent actions. However, such effects aren't inconsistent, since the observation pertains to situation s, whereas the update is to situation $\mathrm{DO}(a, s)$. Such *destructive sensing actions* are commonplace. By the Heisenberg Uncertainty Principle, they are inevitable, but examples can be found in macroscopic domains as well. Biologists find out the number of insects living in a tree by placing containers under the tree and then fogging the tree with poison. The number of insects that fall into the containers provides an estimate of the number that were originally living there.

given action sequence to achieve a given goal. Let a^{-1} be a *regression operator* for action a. $a^{-1}(\varphi)$ is a condition that, if true immediately before the execution of a, results in φ being true after a is executed. We define $(\{a\}_1^n)^{-1}(\varphi)$ to be $a_n{}^{-1}(a_{n-1}{}^{-1}(\ldots(a_1{}^{-1}(\varphi))))$. Naturally, regression on an action sequence of zero length is the identity function: $\{\}^{-1}(\varphi) = \varphi$.

Let α be an axiomatization of the initial conditions, and let Γ be some goal expression. The objective of planning is to produce an executable sequence of actions, $\{a\}_1^n$, such that $\alpha \models (\{a\}_1^n)^{-1}(\Gamma)$. We discuss executability in Section 4.3.

We specify regression operators for **satisfy**, **initially** and **hands-off** goals below. Since some conditions could be true in the initial state, we also must specify when a condition is true after executing a plan of zero length. Since **initially** indicates something that must be true before the plan is executed, and **satisfy** indicates things true afterwards, it follows that if there is no plan, then **initially** and **satisfy** have the same interpretation: For all φ,

$$\alpha \models \text{KNOW}(\varphi, S_0) \Leftrightarrow \{\}^{-1}(\textbf{initially}(\varphi)) = \mathtt{T} \quad (29)$$

$$\alpha \models \text{KNOW}(\varphi, S_0) \Leftrightarrow \{\}^{-1}(\textbf{satisfy}(\varphi)) = \mathtt{T} \quad (30)$$

hands-off is always true in the initial state, since it can only be violated by changing the proscribed fluent:

$$\{\}^{-1}(\textbf{hands-off}(\varphi)) = \mathtt{T} \quad (31)$$

We now consider how to regress a SADL goal formula through an action. A goal **satisfy**(φ) is achieved if the agent knows that φ is true; *i.e.*, φ just became true, was just observed to be true, or was previously known to be true and wasn't subsequently affected. The first two conditions are captured by $\Sigma^a_{\text{KNOW}(\varphi)}$. The latter holds when **satisfy**(φ) held in the previous state, and knowledge of φ was preserved:

$$a^{-1}(\textbf{satisfy}(\varphi)) = \Sigma^a_{\text{KNOW}(\varphi)} \vee (\textbf{satisfy}(\varphi) \wedge \Pi^a_{\text{KNOW}(\varphi)}) \quad (32)$$

A **hands-off** goal holds if the state of φ always remains the same as it was in the initial state. **hands-off**(φ) doesn't forbid actions that *affect* φ — just actions that *change* φ. For example, an action `compress myfile` doesn't violate the goal **hands-off**(compressed(`myfile`)) if `myfile` was already compressed initially.[11]

$$a^{-1}(\textbf{hands-off}(\varphi)) = (\Pi^a_{\neg\varphi} \vee \textbf{initially}(\varphi)) \wedge (\Pi^a_{\varphi} \vee \textbf{initially}(\neg\varphi)) \wedge \textbf{hands-off}(\varphi) \quad (33)$$

[11] This is a departure from UWL's notion of **hands-off**, in which the `compress` would be a violation. However, uncompressing the file and then recompressing it does violate the goal, since the uncompress changes the fluent.

initially(φ) is satisfied after action a if it was already satisfied, or if φ was observed by action a, and wasn't affected by any previous actions. Unlike other goals, we are interested in the *first* time point at which an **initially** goal is achieved, as opposed to the last. The disjunct **initially**(φ) ensures that the first occurrence is considered, because it is always regressed back.

$$a^{-1}(\textbf{initially}(\varphi)) = \textbf{initially}(\varphi) \vee (\kappa^{tv}_{\varphi}(a) \wedge \varphi \wedge \textbf{hands-off}(\varphi)) \quad (34)$$

This definition doesn't rule out using destructive sensing actions. All that matters is that φ be undisturbed *before* it is sensed. It's fine if the act of sensing the value of φ itself affects φ.

Unannotated preconditions merely need to be satisfied in the final state, and it isn't necessary that they be known true.

$$a^{-1}(\varphi) = \Sigma^a_{\varphi} \vee (\varphi \wedge \Pi^a_{\varphi}) \quad (35)$$

Logical operators are simply regressed back to the initial state, since their interpretation is the same across all situations, as detailed in [25].

With these definitions, we can show that regression is correct — that is, if the conditions returned by $a^{-1}(\Gamma)$ are true, and $\{a\}_1^n$ is successfully executed, then Γ will indeed be true.

Theorem 1 (Soundness of Regression) *Let $\{a\}_1^n$ be an executable action sequence. Let Γ be a goal formula, and let α be an axiomatization of the initial state, s_0. Then $\alpha \models (\{a\}_1^n)^{-1}(\Gamma) \Rightarrow GOAL(\Gamma, s_0, \{a\}_1^n)$*

We believe that the reverse is also true — *i.e.*, if Γ is true after $\{a\}_1^n$ is executed, then $a^{-1}(\Gamma)$ must have been true.

4.3 EXECUTABILITY

Regression operators alone only tell part of the story about when an action, or sequence of actions, can achieve a goal. $a^{-1}(\varphi)$ consists of the conditions under which a will achieve φ *assuming it is successfully executed*. So to ensure that a brings about φ, we must also ensure that a can be executed. Action a is executable in situation s iff the preconditions of a are true in s. A sequence of actions, $\{a\}_1^n$, is executable in s iff a_1 is executable in s, a_2 is executable in $\text{DO}(a_1, s)$, a_3 is executable in $\text{DO}(a_2, \text{DO}(a_1, s))$, and so on.

5 EXPRESSIVENESS

Although SADL is appropriate for any Markov domain (*e.g.*, transportation logistics, manufacturing, mobile robotics, *etc.*), the language is best at modeling domains with accurate (low noise) sensors. We have

concentrated our efforts on UNIX and the Internet, encoding hundreds of commands. Examples of sensory actions include `finger, wc, grep`, the `netfind` and `inspec` Internet sites, and actions to traverse the Web; causal actions include `cp, rm`, and `compress`. Universal quantification allows us to model actions that return an unbounded amount of information, such as `ls` (Figure 2).

As an illustration, consider the goal, Γ, of finding a file named `old` and renaming it to `new`: $\Gamma = \exists f.$**initially**(name(f, `old`)) $\wedge$ **satisfy**(name(f, `new`)). Recall that this goal is inexpressible in UWL. It can be achieved by executing `ls` in various directories until the desired file is found, and then executing `mv` to change the name to `new`. There is no single action sequence that will work in all situations, because the location of `old` is not necessarily known. Let's assume that `old` resides in the directory `tex`, and that its location is unknown. We also assume that the agent knows that `tex` is readable, and that current.shell(`csh`) is true. The shortest possible action sequence that would achieve the goal is `ls tex` then `mv tex/old tex/new`. For brevity, we abbreviate these actions as `ls` and `mv`, respectively. We show that this action sequence is executable, and that it achieves the goal.

For the sake of this example, we won't consider `mv` in its full glory. Rather, we assume a simplified version of `mv`, with the precondition $\pi^{\mathtt{mv}} =$ **satisfy**(name(f, `old`)) $\wedge$ **satisfy**(in.dir(f, `tex`)), and the single effect $\varepsilon^{\mathtt{mv}} =$ **cause**(name(f, `new`)). This representation ignores many details, such as whether `tex` is writable, `old` is readable, there is already a file named `new`, etc.

To show that the plan is executable, we must first show that the preconditions of `ls` hold in S_0, *i.e.*, $S_0 \models \pi^{\mathtt{ls}}$, and then show that the preconditions of `mv` hold after `ls` is executed, *i.e.*, $\mathrm{DO}(\mathtt{ls}, S_0) \models \pi^{\mathtt{mv}}$. To show that the plan achieves the goal, we need show that $\mathrm{DO}(\mathtt{mv}, \mathrm{DO}(\mathtt{ls}, S_0)) \models \Gamma$. We use regression to show that these results hold.

We first regress the two conjuncts of Γ through `mv`. `mv` achieves the **satisfy** goal, with no secondary preconditions: $\mathtt{mv}^{-1}$(**satisfy**(name(f, `new`))) $\Leftarrow$ $\Sigma^{\mathtt{mv}}_{\mathrm{KNOW}(\mathrm{name}(f,\mathtt{new}))} \Leftarrow \mathrm{KNOW}(\gamma^{\mathtt{T}}_{\mathrm{name}(f,\mathtt{new})}(\mathtt{mv}))$ $\Leftarrow \mathrm{KNOW}(\mathtt{T}) \Leftarrow \mathtt{T}$.

`mv` has no effect on the **initially** goal: $\mathtt{mv}^{-1}$(**initially**(name(f, `old`) $\Leftarrow$ **initially**(name(f, `old`)).

Now we regress $\mathtt{mv}^{-1}(\Gamma) \wedge \pi^{\mathtt{mv}}$ through `ls`. That is, we regress **initially**(name(f, `old`)) $\wedge$ **satisfy**(name(f, `old`)) $\wedge$ **satisfy**(in.dir(f, `tex`)). We regress the first two conjuncts through `ls`. The final conjunct, **satisfy**(in.dir(f, `tex`)), follows the same pattern.

The action `ls tex` has the effect $\forall f \exists n$ **observe**(name(f, n)) $\wedge$ **observe**(in.dir(f, `tex`)), with the secondary precondition in.dir(f, `tex`). This precondition does not require knowledge on the part of the agent. So $\mathtt{ls}^{-1}$(**initially**(name(f, `old`)) $\wedge$ **satisfy**(name(f, `old`))) $\Leftarrow \Sigma^{\mathtt{ls}}_{\mathrm{KNOW}(\mathrm{name}(f,\mathtt{old}))} \wedge$ **hands-off**(name(f, `old`)) $\wedge\ \Pi^{\mathtt{ls}}_{\neg\mathrm{name}(f,\mathtt{old})} \Leftarrow$ $\kappa^{\mathtt{T}}_{\mathrm{KNOW}(\mathrm{name}(f,\mathtt{old}))}(\mathtt{ls}) \wedge \mathrm{KNOW}(\Pi^{\mathtt{ls}}_{\mathrm{name}(f,\mathtt{old})})$ $\wedge$ name(f, `old`) $\wedge$ **hands-off**(name(f, `old`)) $\wedge$ $\neg\gamma^{\mathtt{T}}_{\mathrm{name}(f,\mathtt{old})}(\mathtt{ls}) \wedge \neg\gamma^{\mathtt{U}}_{\mathrm{name}(f,\mathtt{old})}(\mathtt{ls}) \Leftarrow$ in.dir(f, `tex`) $\wedge \mathrm{KNOW}(\neg\gamma^{\mathtt{T}}_{\mathrm{name}(f,\mathtt{old})}(\mathtt{ls})) \wedge$ $\neg\gamma^{\mathtt{U}}_{\mathrm{name}(f,\mathtt{old})}(\mathtt{ls})) \wedge$ name(f, `old`) $\Leftarrow$ in.dir(f, `tex`) $\wedge \mathrm{KNOW}(\mathtt{T}) \wedge$ name(f, `old`).

This last formula is entailed by S_0. All that remains is to show that $S_0 \models \{\}^{-1}(\pi^{\mathtt{ls}})$. By the definition of $\{\}^{-1}$ for **satisfy** goals, that follows iff $S_0 \models$ KNOW(current.shell(`csh`) $\wedge$ KNOW(protection `tex, readable`), which is true by assumption.

6 TRACTABILITY

SADL is implemented by XII [11, 10], a partial-order planner whose performance is comparable to the UCPOP/SNLP family of classical planners. We analyze its performance in terms of the refinement paradigm described in [15] — XII has three refinement operations: goal establishment, conflict resolution and action execution. Goal establishment involves possibly adding an action to the plan, and adding an *interval protection constraint* (IPC) to prevent the goal from being clobbered. In SADL, there are three possible intervals to consider. If s_p is the situation in which the action will be executed, and s_c is the situation in which the goal is to be fulfilled, the intervals are $[s_{p+1}, s_c]$, $[s_0, s_p]$ or $[s_0, s_c]$, corresponding to **satisfy**, **initially** or **hands-off**, respectively. Ensuring that no actions violate the IPC requires $O(n)$ time, where n is the number of steps in the plan, but maintaining a consistent ordering of actions requires $O(n^2)$ time. Conflict resolution and action execution also take $O(n^2)$ time. In contrast, note that goal establishment and conflict resolution are undecidable in the situation calculus.

7 RELATED WORK

McCarthy and Hayes [19] first argued that an agent needs to reason about its ability to perform an action. Moore [21] devised a theory of knowledge and action, based on a variant of the situation calculus with possible-worlds semantics. He provided an analysis of knowledge preconditions, which we discussed earlier, and information-providing effects. Morgenstern [22] generalized Moore's results to express partial knowledge that agents have about the knowledge of other agents (e.g. "John knows what Bill said"), using a substantially more expressive logic, which

is syntactic rather than modal. Davis [3] extended Moore's theory to handle contingent plans, though, like Moore, he doesn't discuss actions with indeterminate effects. Levesque [17] offers an elegant theory of when a plan, with conditionals and loops, achieves a satisfaction goal in the presence of incomplete information. However, Levesque doesn't discuss knowledge goals, and his sensory actions can return only `T` or `F`, and can't change the state of the world. Goldman and Boddy [12] present a clean language for contingent plans with context-dependent effects and nondeterminism. However, like Levesque, they don't allow variables in sensing actions: possible outcomes are represented as a disjuction. Shoham [33] presents a language, with explicit time, for representing beliefs and communication among multiple agents. Agents can request other agents to perform actions, which can include (nested) communicative actions, but not arbitrary goals. A discrete temporal logic, without $\forall$, is used to represent beliefs. PRS [14] is a procedural language that can represent a similar class of goals as SADL, but lacks temporal goals such as **initially**. PRS has annotation **achieve** corresponding to SADL **satisfy**, **preserve** corresponding to **hands-off**, and **test** corresponding to **satisfy+hands-off**, as well as several procedural constructs that have no corresponding terms in the declarative SADL language.

Partially-observable Markov Decision Processes [20, 2] provide an elegant representation of sensing actions and actions with uncertain outcomes in Markov domains. However, they don't lend themselves to efficient algorithms. With few exceptions, such as [1], work in MDPs assumes that reward functions (goals) are Markov as well, so temporal goals like **initially** are inexpressible.

A number of contingent planning systems have introduced novel representations of uncertainty and sensing actions. Warplan-C [34] tags actions as conditional, meaning they have two possible outcomes: P or $\neg P$. C-BURIDAN [16, 4] uses a probabilistic action language that can represent conditional, observational effects, including noisy sensors, and effects that cause information loss. Unlike SADL, the C-BURIDAN language is propositional, and makes no distinction between knowledge goals and goals of satisfaction. C-BURIDAN and Cassandra [29] (and WCPL [12]) can represent and reason with uncertain outcomes of actions as disjunctions, allowing them to deal with correlations between multiple unknown variables (e.g. either it is raining and Fido is wet, or it is sunny and Fido is dry). By using the `U` truth value, SADL gives up the ability to represent these correlations (*i.e.* as far as the agent knows, it is raining and fido is dry). However, reasoning with `U` truth values is more efficient than the possible-worlds representation used to handle disjunction. CNLP [27], like SADL, uses a three-valued logic to represent uncertainty. Another limitation of these other languages is an inability to represent actions, like `ls` that return information about an unbounded number of objects.

8 CONCLUSIONS

We introduced SADL, a language for representing sensing actions and information goals, which embodies the lessons learned during four years of building and debugging Internet Softbot domain theories: 1) Since knowledge goals are temporal, SADL supports the temporal annotation **initially**. 2) In *Markov domains*, such as UNIX, knowledge preconditions for actions are inappropriate, but subgoaling to obtain knowledge is often necessary; SADL handles this paradox by eliminating knowledge preconditions from actions, but using secondary preconditions to clearly indicate when subgoaling to acquire knowledge could be useful. SADL is expressive enough to represent real-world domains, such as UNIX and the World Wide Web, yet restricted enough to be used efficiently by modern planning algorithms, such as XII [10].

References

[1] Fahiem Bacchus, Craig Boutilier, and Adam Grove. Rewarding behaviors. In *Proc. 14th Nat. Conf. on AI*, 1995.

[2] A. R. Cassandra, L. P. Kaebling, and M. L. Littman. Algorithms for partially observable markov decision processes. Technical report 94-14, Brown University, Providence, Rhode Island, 1994.

[3] E. Davis. Knowledge preconditions for plans. Technical Report 637, NYU Computer Science Department, May 1993.

[4] D. Draper, S. Hanks, and D. Weld. A probabilistic model of action for least-commitment planning with information gathering. In *Proc. 10th Conf. on Uncertainty in Artifical Intelligence*, 1994.

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

[6] O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 1997. (To appear).

[7] O. Etzioni and D. Weld. A softbot-based interface to the Internet. *CACM*, 37(7):72–76, 1994.

[8] Oren Etzioni, Keith Golden, and Dan Weld. Tractable closed-world reasoning with updates. In *Proc. 4th Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 178–189, 1994.

[9] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information.

In *Proc. 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 115–125, 1992.

[10] K. Golden, O. Etzioni, and D. Weld. Planning with execution and incomplete information. Technical Report 96-01-09, University of Washington, Department of Computer Science and Engineering, February 1996. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[11] Keith Golden, Oren Etzioni, and Dan Weld. Omnipotence without omniscience: Sensor management in planning. In *Proc. 12th Nat. Conf. on AI*, pages 1048–1054, 1994.

[12] Robert P. Goldman and Mark S. Boddy. Expressive Planning And Explicit Knowledge. In *Proc. 3rd Intl. Conf. on AI Planning Systems*, May 1996.

[13] Peter Haddawy and Steve Hanks. Utility Models for Goal-Directed Decision-Theoretic Planners. Technical Report 93–06–04, Univ. of Washington, Dept. of Computer Science and Engineering, September 1993. Submitted to *Artificial Intelligence*. Available via FTP from `pub/ai/` at `ftp.cs.washington.edu`.

[14] F. Ingrand, R. Chatila, R. Alami, and F Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *Proceedings of the 1996 IEEE International Conference On Robotics and Automation*, 1996.

[15] 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.

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

[17] Hector Levesque. What is planning in the presence of sensing? In *Proc. 14th Nat. Conf. on AI*, 1996.

[18] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1,2):27–39, April 1980.

[19] 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.

[20] G. E. Monahan. A survey of partially observable markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[21] 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.

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

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

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

[25] E. Pednault. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence*, 4(4):356–372, 1988.

[26] 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.

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

[28] Martha Pollack. The uses of plans. *Artificial Intelligence*, 57(1), 1992.

[29] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 1996.

[30] Howard Raiffa. *Decision Analysis: Introductory Lectures on Choices Under Uncertainty*. Addison-Wesley, 1968.

[31] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.

[32] R. Scherl and H. Levesque. The frame problem and knowledge producing actions. In *Proc. 11th Nat. Conf. on AI*, pages 689–695, July 1993.

[33] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, March 1993.

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

[35] Dan Weld and Oren Etzioni. The first law of robotics (a call to arms). In *Proc. 12th Nat. Conf. on AI*, pages 1042–1047, 1994.

[36] D. E. Wilkins. *Practical Planning*. Morgan Kaufmann, San Mateo, CA, 1988.

[37] M. Williamson and S. Hanks. Optimal planning with a goal-directed utility model. In *Proc. 2nd Intl. Conf. on AI Planning Systems*, June 1994.