Adding Knowledge to the Action Description Language $\mathcal{A}$

Jorge Lobo*
Department of EECS
University of Illinois at Chicago
jorge@eecs.uic.edu

Gisela Mendez
Departamento de Matemáticas
Universidad Central de Venezuela
gmendez@ciens.ucv.ve

Stuart R. Taylor
Department of EECS
University of Illinois at Chicago
staylor@eecs.uic.edu

Abstract

We introduce $\mathcal{A}_k$ an extension of the action description language $\mathcal{A}$ (Gelfond & Lifschitz 1993) to handle actions which affect knowledge. We use sensing actions to increase an agent's knowledge of the world and non-deterministic actions to remove knowledge. We include complex plans involving conditionals and loops in our query language for hypothetical reasoning. Finally, we present a translation of descriptions in $\mathcal{A}_k$ to epistemic logic programs.

Introduction

The action description language $\mathcal{A}$ was designed as a minimal core of a high level language to represent and reason about actions and their effects. Extensions of $\mathcal{A}$ have been developed to study and reason about the concurrent execution of actions, the non-deterministic effects of some actions and to study many instances of the qualification, ramification and frame problems.

In this paper, we propose a new action description language called $\mathcal{A}_k$. $\mathcal{A}_k$ is a minimal extension of $\mathcal{A}$ to handle *sensing* actions. A sensing action is an action that does not have any effect in the world. Instead, a sensing action will increase the agent's knowledge about the current state of the world. We also consider actions with non-deterministic effects since these actions may cause the agent to lose knowledge.[2]

It has been shown that sensing actions give an agent the ability to reason about complex plans that include conditionals and iterations (Levesque 1996). We will show how we can extend the query language of $\mathcal{A}$ to prove properties of complex plans in $\mathcal{A}_k$. We will also show how, similar to $\mathcal{A}$ domains, domain descriptions in $\mathcal{A}_k$ can be translated into logic programs. Because of our manipulation of knowledge, our translation will be into epistemic logic programs (Gelfond 1994).

*Partially funded by Argonne National Laboratory under Contract No. 963042401

[2]If we assume that time or errors do not affect the memory of the agent.

We will start with the description of the syntax and semantics of the deterministic part of $\mathcal{A}_k$. Then, we will introduce the query language for verifying properties of complex plans. Next, we will extend $\mathcal{A}_k$ with non-deterministic actions and present the translation into epistemic logic programs. Finally, we will discuss the relation to other work and present some concluding remarks.

Syntax of $\mathcal{A}_k$

The language of $\mathcal{A}_k$ consists of two non-empty disjoint sets of symbols F of *fluents* and A of *actions*. The set A consists of two disjoint sets of actions, *sensing* actions and *non-sensing* actions. Generic elements from F will be denoted by f, possibly indexed. Generic elements from A will be denoted by a possibly indexed too. A *fluent literal* is an element from the set of fluents that are possibly preceded by a $\neg$ sign. A fluent literal is negative when preceded by $\neg$, otherwise it is positive. There are two kinds of propositions in $\mathcal{A}_k$, *object effect* propositions and *value* propositions. Object effect propositions are expressions of the form

$$a \textbf{ causes } f \textbf{ if } \varphi \qquad (1)$$

where a is a non-sensing action, f a fluent literal, and φ a propositional formula built using fluent literals. This proposition intuitively means that in a situation where φ is true the execution of a causes f to become true.

A *value proposition* is an expression of the form

$$\textbf{initially } \varphi \qquad (2)$$

where φ is a formula. Value propositions describe the initial knowledge the agent has about the world.

There are also *knowledge* laws. Knowledge laws are expressions of the form

$$a \textbf{ causes to know } f \textbf{ if } \varphi \qquad (3)$$

where a is a sensing action, f is a fluent and φ a formula as in (1). Intuitively, this proposition says that in a situation where φ is true, the execution of a causes the agent to realize the current value of f in the world. We do not allow sensing actions to occur in object effect propositions.

At this point, we should remark that we are assuming the agent may have incomplete but always correct knowledge about the world. Propositions and laws in $\mathcal{A}_k$ describe how the knowledge of the agent changes. If these changes are the result of propositions like (1), we assume that the effects in the world would be the same as if the world were in a state where φ was true.

If φ is a tautology in (1) or (3) we will write the propositions as

$$a \textbf{ causes } f;\ \text{and } a \textbf{ causes to know } f \qquad (4)$$

A collection of the above propositions and laws is called a *domain description.*

Example 1 A robot is instructed to replace the halogen bulb of lamp and it needs to find a sequence of actions to complete the task without burning out its circuits. Assume that its world is described by the following domain description.

$$D_1 \begin{cases} r_1 : \textbf{initially } \neg burnOut \\ r_2 : \textbf{initially } \neg bulbFixd \\ r_3 : changeBulb \textbf{ causes } burnOut \textbf{ if } swtchOn \\ r_4 : changeBulb \textbf{ causes } bulbFixd \textbf{ if } \neg swtchOn \\ r_5 : turnSwtch \textbf{ causes } swtchOn \textbf{ if } \neg swtchOn \\ r_6 : turnSwtch \textbf{ causes } \neg swtchOn \textbf{ if } swtchOn \end{cases}$$

It follows from D_1 that in the initial state the robot does not know the state of the switch in the lamp. The robot is at risk of being burned out if the action *changeBulb* is executed. The robot must first check the state of the switch. This can be described with a knowledge law such as:
r_7 : *checkSwtch* **causes to know** *swtchOn* **if** $\neg burnOut$
Sensing gives the robot that extra knowledge it would need to accomplish the task without burning out and provides a branching point in its hypothetical reasoning. If the switch is on, it will turn the switch and replace the bulb. If the switch is off, it will directly replace the bulb. □

Semantics of $\mathcal{A}_k$

The semantics of $\mathcal{A}_k$ must describe how knowledge changes according to the effects defined by a domain description. We will represent the knowledge of an agent by a set of worlds in which the agent believes it can be. We call these worlds *situations* and a collection of worlds an *epistemic state.* A situation, since it could be an incomplete description of the world, will be represented by a collection of sets of fluents. A set of fluents will be called a *state.* If a formula is true in an epistemic state of an agent (to be defined later), we assume it means that the agent knows the formula is true in the real world. Epistemic states will also allow us to distinguish when the agent only knows that the formula $f_1 \vee f_2$ is true as opposed to knowing that f_1 is true or knowing that f_2 is true.[3]

[3]Note the similarity with a collection of beliefs in (Gelfond 1994).

We will say that a fluent f is true (or holds) in a state σ iff $f \in \sigma$. A fluent f is false (or does not hold) in a state σ iff $f \notin \sigma$. For more complex formulas, their truth value can be recursively defined as usual. A formula is true in a situation if it is true in every state in the situation; it is false if it is false in every state of the situation; otherwise its truth value is *unknown.* A situation is *consistent* if it is non-empty; otherwise it is *inconsistent.* An epistemic state is inconsistent if it is empty or contains an inconsistent situation; otherwise it is consistent.

Interpretations for $\mathcal{A}_k$ are functions that map pairs of actions and situations into situations. To define when an interpretation models a domain description, we will define an auxiliary function that will interpret the effect of actions at the state level. Hence, *0-interpretations* are functions that map an action and state into state. A 0-interpretation Φ_0 is a *0-model* of a domain description D iff for every state σ

1. For a fluent f of any effect proposition of the form "a **causes** f **if** φ" in D, the fluent f holds in $\Phi_0(a, \sigma)$ if its formula φ holds in σ,
2. For a fluent literal $\neg f$ of any effect proposition of the form "a **causes** $\neg f$ **if** φ" in D, the fluent f does not hold in $\Phi_0(a, \sigma)$ if its formula φ holds in σ,
3. For a fluent f, if there are no effect propositions of the above types, then $f \in \Phi_0(a, \sigma)$ if and only if $f \in \sigma$.

Before we define when an interpretation Φ is a *model* of a domain description D, we need the following definitions that will allow us to interpret knowledge laws.

Definition 2 A state σ is called an *initial state* of a domain description D iff for every value proposition of the form "**initially** φ" in D, φ is true in σ. The *initial situation* Σ_0 of D is the the set of all the initial states of D.

Sensing of an unknown fluent f will split a situation into situations where the fluent is known if the precondition φ of the action holds.[4] This is stated below.

Definition 3 When the formula is true, the fluent's value will be known. Let Σ be a consistent situation, f a fluent and φ a propositional formula. A consistent situation Σ' is "*f, φ-compatible*" with Σ iff $\Sigma' = \Sigma$ whenever f is either true or false in Σ; otherwise Σ' must satisfy one of the following conditions:

1. $\Sigma' = \{\sigma \in \Sigma \mid \varphi \text{ is not true in } \sigma\}$
2. $\Sigma' = \{\sigma \in \Sigma \mid \varphi \text{ is true in } \sigma,\ f \notin \sigma\}$
3. $\Sigma' = \{\sigma \in \Sigma \mid \varphi \text{ is true in } \sigma,\ f \in \sigma\}$

Definition 4 Given an interpretation Φ of $\mathcal{A}_k$, Φ is a *model* of a domain description D, if and only if for any consistent situation Σ.

[4]Unknown preconditions are also learned during sensing.

1. There exists a 0-model Φ_0 of D, such that for any non-sensing action a, $\Phi(a,\Sigma) = \bigcup_{\sigma\in\Sigma}\{\Phi_0(a,\sigma)\}$.
2. For each sensing action a, Let

   a **causes to know** f_1 **if** φ_1

   $\vdots$

   a **causes to know** f_n **if** φ_n

   be all the knowledge laws in which a occurs. Then, $\Phi(a,\Sigma)$ must be consistent and if $n = 0$, $\Phi(a,\Sigma) = \Sigma$ otherwise $\Phi(a,\Sigma) = \bigcap_{i\in[1..n]}\Sigma_i$, such that each Σ_i is a situation f_i, φ_i – *compatible* with Σ.

$\Phi(a,\Sigma) = \emptyset$ for any action a if $\Sigma = \emptyset$.

Example 5 Assume that when an agent assigned to clean white-boards looks into a room(denoted by *lknRm*) it will know whether the white-board in that room is clean(denoted by *bdCl*). It will also learn whether it is raining outside(denoted by *rnOs*) if the curtains in the room are open(denoted by *ctOp*). We assume that because of building policy, the agent will initially know that the lights of the room will be on(denoted by *ltOn*) and that the curtains of the room will be open. A simple domain description for this situation will be:

$$D_2 \begin{cases} r_1: \textbf{ initially } ctOp \\ r_2: \textbf{ initially } ltOn \\ r_3: lknRm \textbf{ causes to know } rnOs \textbf{ if } ctOp \\ r_4: lknRm \textbf{ causes to know } bdCl \textbf{ if } ltOn \end{cases}$$

The initial situation Σ_0 of D_2 has four states.

$$\Sigma_0 = \{\{ctOp, ltOn\}, \{rnOs, ctOp, ltOn\}, \{bdCl, ctOp, ltOn\}, \{rnOs, bdCl, ctOp, ltOn\}\}$$

There is only one action in D_2, and any model of D_2, given the initial situation Σ_0, may behave in one of the following forms:

$\Phi_1(lknRm, \Sigma_0) = \{\{ctOp, ltOn\}\}$
$\Phi_2(lknRm, \Sigma_0) = \{\{rnOs, ctOp, ltOn\}\}$
$\Phi_3(lknRm, \Sigma_0) = \{\{bdCl, ctOp, ltOn\}\}$
$\Phi_4(lknRm, \Sigma_0) = \{\{rnOs, bdCl, ctOp, ltOn\}\}$

Models may differ in how they behave when they are applied to situations different than Σ_0, but for Σ_0 they must be equal to one of the Φ_i. To verify that each of the Φ_i can be a partial description of a model of r_3 and r_4, let

$\Sigma_1 = \{\{ctOp, ltOn\}, \{bdCl, ctOp, ltOn\}\}$
$\Sigma_2 = \{\{rnOs, ctOp, ltOn\}, \{rnOs, bdCl, ctOp, ltOn\}\}$
$\Sigma_3 = \{\{ctOp, ltOn\}, \{rnOs, ctOp, ltOn\}\}$
$\Sigma_4 = \{\{bdCl, ctOp, ltOn\}, \{rnOs, bdCl, ctOp, ltOn\}\}$

Note that Σ_1 and Σ_2 are *rnOs, ctOp*-compatible with Σ_0, Σ_3 and Σ_4 are *bdCl, ltOn*-compatible with Σ_0, and

$\Phi_1(lknRm, \Sigma_0) = \Sigma_1 \cap \Sigma_3$
$\Phi_2(lknRm, \Sigma_0) = \Sigma_2 \cap \Sigma_3$
$\Phi_3(lknRm, \Sigma_0) = \Sigma_1 \cap \Sigma_4$
$\Phi_4(lknRm, \Sigma_0) = \Sigma_2 \cap \Sigma_4$ □

Answering Queries

In $\mathcal{A}$ (Gelfond & Lifschitz 1993), queries are of the form

$$\varphi \textbf{ after } [a_1, \ldots, a_n] \qquad (5)$$

The answer to this query will be *yes* (or true) in a domain D if for every model Φ of D the formula φ is true in the situation $\Phi(a_n, \Phi(a_{n-1}, \ldots \Phi(a_1, \Sigma_0)\cdots))$, i.e. the situation that results after the execution of $a_1, \ldots, a_n$ from the initial situation. The answer will be *no* if φ is false in $\Phi(a_n, \Phi(a_{n-1}, \ldots \Phi(a_1, \Sigma_0)\cdots))$ for every model Φ of D. Otherwise the answer will be unknown. With this notion we can define an entailment relation between domain descriptions and queries. We say that a domain D entails a query Q, denoted by $D \models Q$, if the answer for Q in D is yes. For example, if we add **initially** *swtchOn* to D_1, it can be easily shown that

$D_1 \models bulbFixd$ **after** $[turnSwtch, changeBulb]$

However, from the original D_1 (even including r_7) there is no sequence of actions α such that $D_1 \models bulbFixd$ **after** α. The inferences from D_1 are conditioned to the output of the sensing action: if the switch is on then the sequence $[turnSwtch, changeBulb]$ will cause the light to be fixed, else the single action $[changeBulb]$ will fix it. Reasoning in the presence of sensing actions requires the projections to be over plans more complex than a simple sequence of actions. We recursively define a *plan* as follows,

1. An action is a plan.
2. If α_1 and α_2 are plans then the sequence α_1, α_2 is a plan.
3. If φ is a formula and α_1 and α_2 are plans then **if** φ **then** α_1 **else** α_2 and **if** φ **then** α_1 are (conditional) plans.
4. If φ is a formula and α is a plan then **while** φ **do** α is also a (routine) plan.
5. Nothing else is a plan.

Now we redefine a query to be a sentence of the form

$$\varphi \textbf{ after } [\alpha] \qquad (6)$$

Where φ is a formula and α a plan. Now we can ask if D_1 entails the query

bulbFixd **after** [*checkSwtch*,
  if $\neg$*swtchOn* **then** *changeBulb*
   else [*turnSwtch*, *changeBulb*]].

The evaluation of a plan is defined in terms of interpretations. We define a new function that takes a plan and a situation and returns the situation that may result after the execution of the plan. We adapt the denotational semantics of conditionals and while loops from conventional programming languages to define the evaluation function of plans. We will start by defining a collection of continuous transformations of functions that map situations into situations. Let $\mathcal{E}$ be the set of all situations and $\mathcal{P} = \{f \mid f : \mathcal{E} \rightarrow \mathcal{E}\}$.

Definition 6 Let α be a plan and $\Gamma \in \mathcal{P}$. Let φ be a formula. Then, we define the function $\mathcal{F}^{\Gamma}_{\alpha,\varphi} : \mathcal{P} \rightarrow \mathcal{P}$ such that for any function $f \in \mathcal{P}$,

$$\mathcal{F}^{\Gamma}_{\alpha,\varphi}(f)(\Sigma) = \begin{cases} \Sigma & \text{if } \varphi \text{ is false in } \Sigma \\ f(\Gamma(\alpha,\Sigma)) & \text{if } \varphi \text{ is true in } \Sigma \\ \emptyset & \text{otherwise} \end{cases}$$

Let $f_1, f_2 \in \mathcal{P}$. We say that $f_1 \leq f_2$ if and only if for any $\Sigma \in \mathcal{E}$ if $f_1(\Sigma) \neq \emptyset$, then $f_1(\Sigma) = f_2(\Sigma)$. $\leq$ is a complete semi-lattice and the transformations $\mathcal{F}^{\Gamma}_{\alpha,\varphi}$ are continuous. We will denote the bottom element by $f_{\emptyset}$. (i.e. the function which output is always the empty situation).
Hence, we can define the powers of $\mathcal{F}^{\Gamma}_{\alpha,\varphi}$ as follows:

1. $\mathcal{F}^{\Gamma}_{\alpha,\varphi} \uparrow 0 = f_{\emptyset}$.
2. $\mathcal{F}^{\Gamma}_{\alpha,\varphi} \uparrow n+1 = \mathcal{F}^{\Gamma}_{\alpha,\varphi}(\mathcal{F}^{\Gamma}_{\alpha,\varphi} \uparrow n)$.
3. $\mathcal{F}^{\Gamma}_{\alpha,\varphi} \uparrow \omega = \bigsqcup\{\mathcal{F}^{\Gamma}_{\alpha,\varphi} \uparrow n \mid n \leq \omega\}$.

Note the least fix-point of $\mathcal{F}^{\Gamma}_{\alpha,\varphi}$ is defined by item (3).[5]

Definition 7 The plan evaluation function Γ_Φ of an interpretation Φ is a function such that for any situation Σ

1. $\Gamma_\Phi(a,\Sigma) = \Phi(a,\Sigma)$ for any action a.
2. $\Gamma_\Phi([\alpha_1,\alpha_2],\Sigma) = \Gamma_\Phi(\alpha_2, \Gamma_\Phi(\alpha_1,\Sigma))$.
3. $\Gamma_\Phi(\textbf{if } \varphi \textbf{ then } \alpha, \Sigma) = \begin{cases} \Gamma_\Phi(\alpha,\Sigma) & \text{if } \varphi \text{ is true in } \Sigma \\ \Sigma & \text{if } \varphi \text{ is false in } \Sigma \\ \emptyset & \text{otherwise} \end{cases}$
4. $\Gamma_\Phi(\textbf{if } \varphi \textbf{ then } \alpha_1 \textbf{ else } \alpha_2, \Sigma) = \begin{cases} \Gamma_\Phi(\alpha_1,\Sigma) & \text{if } \varphi \text{ is true in } \Sigma \\ \Gamma_\Phi(\alpha_2,\Sigma) & \text{if } \varphi \text{ is false in } \Sigma \\ \emptyset & \text{otherwise} \end{cases}$
5. $\Gamma_\Phi(\textbf{while } \varphi \textbf{ do } \alpha, \Sigma) = \mathcal{F}^{\Gamma_\Phi}_{\textbf{if}\varphi\textbf{then}\alpha,\varphi} \uparrow \omega$

We will present an example of routines in the next section after we introduce actions that remove knowledge.

Knowledge Removing Actions

A non-deterministic action is an action in which the outcome cannot be predicted beforehand as with the toss of a coin. Once the action *toss* occurs, we no longer know whether heads or tails will show. *Non-deterministic actions* remove knowledge. We describe the removal of knowledge as no longer knowing the truth value of a fluent. A non-deterministic effect proposition is of the form

$$a \textbf{ may affect } f \textbf{ if } \varphi \qquad (7)$$

where a is non-sensing action. The formula φ is defined as in (1). Intuitively, the proposition says that the truth value of f may change if a is executed in a situation where φ is true. When φ is a tautology, (7) becomes

$$a \textbf{ may affect } f \qquad (8)$$

[5]$\bigsqcup$ denotes the least upper bound.

We now re-define 0-interpretations to be functions that map an action and a state to a set of states. A 0-interpretation Φ_0 is a *0-model* of a domain description D iff for every state σ, $\Phi_0(a,\sigma)$ contains every state σ' such that

1. For a fluent f of any effect proposition of the form "a **causes** f **if** φ" in D, $f \in \sigma'$ if φ holds in σ,
2. For a fluent literal $\neg f$ of any effect proposition of the form "a **causes** $\neg f$ **if** φ" in D, $f \notin \sigma'$ if φ holds in σ,
3. For a fluent f such that there are no effect propositions of the above types, $f \in \sigma'$ if and only if $f \in \sigma$, unless there is a non-deterministic effect proposition of the form a **may affect** f **if** φ for which φ holds in σ.

Interpretations and models of domain descriptions are still defined as in Section 3.

Example 8 Our agent is ordered this time to put ice into cups. The ice cannot fit in the cups. The agent decides to drop the ice to produce smaller pieces of ice that will fit.

$$D_3 \begin{cases} t1 : \textbf{initially } inHandIce \\ t2 : \textbf{initially } solidIce \\ t3 : \textbf{initially } dropNo \\ t4 : pickUp \textbf{ causes } inHandIce \textbf{ if } \neg inHandIce \\ t5 : drop \textbf{ causes } \neg inHandIce \textbf{ if } inHandIce \\ t6 : drop \textbf{ may affect } solidIce \textbf{ if } dropNo \\ t7 : drop \textbf{ may affect } solidIce \textbf{ if } dropAFew \\ t8 : drop \textbf{ causes } dropAFew \textbf{ if } dropNo \\ t9 : drop \textbf{ causes } dropEnough \textbf{ if } dropAFew \\ t10 : drop \textbf{ causes } \neg solidIce \textbf{ if } dropEnough \\ t11 : checkIce \textbf{ causes to know } solidIce \\ t12 : putIceIn \textbf{ causes } iceInCups \textbf{ if } \neg solidIce \end{cases}$$

Rules t1 - t3 establish what is initially known in the world. Rules t4 and t5 describes the effect that *Drop* and *pickUp* have on *inHandIce*. Rules t6 and t7 describe the non-deterministic effect of the action *drop* on the ice. Rules t8 - t10 are object effect propositions which ensure that the ice will break after *drop* occurs at most three times. This condition ensures termination and it will be explained later in this section. Rule t11 is the sensing action which allow the agent to know whether the ice is broken or not after the execution of the non-deterministic action *drop*. Rule t12 is the goal of the task the agent is to perform.

It can be proved that D_3 entails the following query,
iceInCups **after**
[**while** $\neg$*solidIce* **do** [*drop*, *pickUp*, *checkIce*],
putIceIn] □

The above query illustrates the use of a routine. Notice, however, that the plan in the query could have been written using three nested conditionals. Ideally, we would like to use a routine which could solve any type of repetitive task, but then we are faced with the problem of termination verification. Each task has its own condition for termination. For example, breaking the ice differs from filling the volume of a cup with ice. Do we really know if the ice will break? Or how do we know that the cup is filling up? With time the ice will either melt or break, and if we do not place infinitesimally small amounts of ice in the cup the cup eventually will fill up or we will run out of ice. To ensure termination (either with success or failure) we need to add to our domain descriptions general axioms or constraints of this sort. We do not have constraints in $\mathcal{A}_k$ but we may be able to add them by using other extensions of $\mathcal{A}$ such as the one in (Baral, Gelfond, & Provetti 1995). However, analysis of termination of plans is outside the scope of this paper.

Translation into Epistemic Logic Programs

We will translate domain descriptions into epistemic logic programs (Gelfond 1994). In epistemic logic programs the language of extended logic programs is expanded with two modal operators K and M. KF is read as "F is known to be true" and MF is read as "F may be believed to be true."

In this section, we will assume that all formulas φ appearing in a domain or a query are always conjunctions of fluent literals. The translation, as in domains of $\mathcal{A}$, can be partitioned into a set of *domain dependent* and set of *domain independent* rules. In addition we will also have a domain independent set of rules to evaluate queries.

Our logic program will use variables of three sorts: *situation* variables denoted by S or S' possibly indexed, *fluent* variables denoted by F or F' possibly indexed, and *action* variables denoted by A or A' possibly indexed, and the special situation constant s_0 that represents the initial situation.

The Domain independent translation

We start by first giving the rules for inertia. These rules encode that a fluent remains unchanged if no action that affects the fluent is executed.

$holds(F, res(A,S)) \leftarrow holds(F,S), \textbf{not } ab(\overline{F}, A, S).$
$\neg holds(F, res(A,S)) \leftarrow$
$\quad \neg holds(F,S), \textbf{not } ab(F,A,S).$

Any fluent not found in a value proposition generates an or-classicalization[6] rule of the form
$\quad holds(f, s_0) \; or \; \neg holds(f, s_0)$

The above rule captures the incomplete description of the world. When no information is given about a fluent, one cannot say that the fluent's value is true or false, but rather unknown.

The ignorance rule is

$unk(F,S) \leftarrow \neg K holds(F,S), \neg K \neg holds(F,S).$

Rules required to process queries will be given later.

The Domain dependent translation

Value propositions of the form "**initially** f" and "**initially** $\neg f$" are translated into

$holds(f, s_0)$, and $\neg holds(f, s_0)$.

Object effect propositions of the form "a **causes** f **if** $p_1, \ldots, p_n$" become

$holds(f, res(a,S)) \leftarrow holds(p_1,S), \ldots, holds(p_n,S).$
$ab(f,a,S) \leftarrow holds(p_1,S), \ldots, holds(p_n,S).$

Any time a precondition p_i in an effect proposition is of the form $\neg q$ (i.e. a negative fluent literal) then $holds(p_i, S)$ in the translation represents $\neg holds(q,S)$. We will assume the same representation in the translation of the other propositions.

"a **causes** $\neg f$ **if** $p_1, \ldots, p_n$" becomes

$\neg holds(f, res(a,S)) \leftarrow$
$\quad holds(p_1,S), \ldots, holds(p_n,S).$
$ab(\overline{f},a,S) \leftarrow holds(p_1,S), \ldots, holds(p_n,S).$

Knowledge laws of the form "a **causes to know** f **if** $p_1, \ldots, p_n$" become

$holds(f, res(a,S)) \leftarrow \neg M \neg holds(f, res(a,S)),$
$\quad holds(p_1,S), \ldots, holds(p_n,S), unk(f,S).$
$ab(f,a,S) \leftarrow \neg M \neg holds(f, res(a,S)),$
$\quad holds(p_1,S), \ldots, holds(p_n,S), unk(f,S).$
$\neg holds(f, res(a,S)) \leftarrow \neg M \; holds(f, res(a,S)),$
$\quad holds(p_1,S), \ldots, holds(p_n,S), unk(f,S).$
$ab(\overline{f},a,S) \leftarrow \neg M \; holds(f, res(a,S)),$
$\quad holds(p_1,S), \ldots, holds(p_n,S), unk(f,S).$

The first rule plus the abnormality rule capture that the value of the fluent f will be known to be true after a is executed. The subjective formula in the body of the rules state "There will be no state in the resulting situation in which f will be known to be false". The other two rules capture that the value of the fluent f will be known to be false. The subjective formula in the body of these rules state "There will be no state in the resulting situation in which f will be known to be true". The unknown literal(unk) ensures that these rules will fire only when the value of the fluent is not already known.

"a **may affect** f **if** $p_1, \ldots, p_n$" is translated into

$holds(f, res(a,S)) \leftarrow \textbf{not } \neg holds(f, res(a,S)),$
$\quad holds(p_1,S), \ldots, holds(p_n,S).$
$\neg holds(f, res(a,S)) \leftarrow \textbf{not } holds(f, res(a,S)),$
$\quad holds(p_1,S), \ldots, holds(p_n,S).$
$ab(f,a,S) \leftarrow \textbf{not } \neg holds(f, res(a,S)),$
$\quad holds(p_1,S), \ldots, holds(p_n,S).$
$ab(\overline{f},a,S) \leftarrow \textbf{not } holds(f, res(a,S)),$
$\quad holds(p_1,S), \ldots, holds(p_n,S).$

[6]Recall that since the semantics of logic prog. is based on minimal models, the meaning of "or" differs form the classical $\vee$ of prop. logic.

The recursion through default negation provides the desired effect of two possible interpretations for the effect of a in f.

Query translation

Queries involving plans and fluents as in "f **after** $[\alpha]$" are represented as $holds(F, res(\alpha, s_0))$. When the object fluent is negative as $\neg f$ then the query becomes $\neg holds(F, res(\alpha, s_0))$.

Plans which are more complex than actions are evaluated with the following rules that are part of every translation:

$holds(F, res([\alpha_1, \alpha_2], S)) \leftarrow$
$\quad holds(F, res(\alpha_2, res(\alpha_1, S)))$.
$holds(F, res(\textbf{if } \varphi \textbf{ then } \alpha_1 \textbf{ else } \alpha_2, S) \leftarrow$
$\quad holds(F, res(\alpha_2, S)), \neg holds(\varphi, S)$.
$holds(F, res(\textbf{if } \varphi \textbf{ then } \alpha_1 \textbf{ else } \alpha_2, S) \leftarrow$
$\quad holds(F, res(\alpha_1, S)), holds(\varphi, S)$.
$holds(F, res(\textbf{while } \varphi \textbf{ do } \alpha, S) \leftarrow$
$\quad holds(F, S)), \neg holds(\varphi, S)$.
$holds(F, res(\textbf{while } \varphi \textbf{ do } \alpha, S) \leftarrow$
$holds(F, res(\textbf{while } \varphi \textbf{ do } \alpha, res(\alpha, S))), \neg holds(\varphi, S)$.

Let Π_D be the epistemic logic program corresponding to the translation of a domain description D. Then we can show:

Theorem 9 $D \models F$ **after** $[\alpha]$ iff $\Pi_D \models holds(F, res(\alpha, s_0))$.

Final Discussion

In (Levesque 1996) there is a programming language based on the situation calculus which also uses sensing actions. This work is based on previous work from (Scherl & Levesque 1993), in which knowledge is represented using two levels. There is a representation of the actual situation (called s) in which the agent is in, and there are situations accessible from s (called s') which the agent thinks it might be in. Something is known to the agent as being true(false) if it is true(false) in all situations s' which are accessible from the actual situation s and is unknown otherwise. We remark that their situations are analogous to what we call states. Our representation of knowledge is done with only one level: the world in the "mind" of the agent. Something is known in our representation if its value is the same throughout the states in a situation and unknown otherwise. The agent will act and make decisions based on what it knows in its "mind". In (Levesque 1996; Scherl & Levesque 1993) the authors use preconditions which are executability conditions for an action's execution. Our preconditions differ in that they are conditions on the effects. Extending $\mathcal{A}_k$ to include executability conditions is desirable for future work. The use of conditions on effects however allows us to represent a phenomenon of sensing in which the value of previously unknown preconditions are learned along with the fluent we are trying to gain knowledge about. In (Levesque 1996) once knowledge is gained it is never lost. In this paper, we introduce the use of non-deterministic actions as a mechanism to remove knowledge.

We use non-deterministic actions to make a fluent true or false, but exactly which is indeterminate. As might be expected, there are cases where the possible outcome is not intuitive. These cases are prevented with integrity constraints as in (Kartha & Lifschitz 1994). Our language could be easily extended to include constraints as in (Baral, Gelfond, & Provetti 1995), but our interest in non-determinism comes from its effect on knowledge.

Most translations for dialects of $\mathcal{A}$ are to extended logic programs. Our translation is to epistemic logic programs because of its ability to represent knowledge and incomplete information. To the best of our knowledge this is the first use of epistemic logic programs in a translation from action languages.

Two possible directions of research are: First, the ability of an agent to query itself about what it knows (i.e introspection). This is useful when the cost of executing a series of plans is expensive (maybe in terms of time). Allowing an agent to query whether it knows that it knows something may be a cheaper alternative and cost effective than executing an action to gather information about the world. This type of introspection cannot be done directly in Levesque's formalism. In our formalism we can add a modal operator to the query language to accomplish this. Second, we could investigate expanding the initial epistemic state. At present, a domain may start from a situation with only one initial epistemic state. For more states or to represent multiple initial situations the language to describe domains must also be extended with modal operators.

References

Baral, C.; Gelfond, M.; and Provetti, A. 1995. Representing actions: Laws, observations and hypotheses. Technical report, University of Texas at El Paso.

Gelfond, M., and Lifschitz, V. 1993. Representing actions and change by logic programs. *Journal of Logic Programming* 17(2,3,4):301–323.

Gelfond, M. 1994. Logic programming and reasoning with incomplete information. *Annals of Mathematics and Artificial Intelligence* 98–116.

Kartha, G. N., and Lifschitz, V. 1994. Actions with indirect effects. In *Proc. of the Fourth Int'l conf. on Principles of Knowledge Representation and Reasoning*, 341–350.

Levesque, H. 1996. What is planning in the presence of sensing? In *Proc. of AAAI-96*, 1139–1146.

Scherl, R., and Levesque, H. 1993. The frame problem and knowledge-producing actions. In *Proc. AAAI-93*, 689–695.