A Semantics for ADL as Progression in the Situation Calculus

Jens Claßen and **Gerhard Lakemeyer**
Department of Computer Science
RWTH Aachen
52056 Aachen
Germany
⟨classen|gerhard⟩@cs.rwth-aachen.de

Abstract

Lin and Reiter were the first to propose a purely declarative semantics of STRIPS by relating the update of a STRIPS database to a form of progression in the situation calculus. In this paper we show that a corresponding result can be obtained also for ADL. We do so using a variant of the situation calculus recently proposed by Lakemeyer and Levesque. Compared to Lin and Reiter this leads to a simpler technical treatment, including a new notion of progression.

Introduction

Lin and Reiter (Lin & Reiter 1997) were the first to propose a purely declarative semantics of STRIPS (Fikes & Nilsson 1971) by relating the update of a STRIPS database to a form of progression of a corresponding situation-calculus theory. More precisely, they show that when translating STRIPS planning problems into basic action theories of Reiter's situation calculus (Reiter 2001), then the STRIPS mechanism of adding and deleting literals after an action A is performed is correct in the sense that the conclusions about the future that can be drawn using the updated theory are the same as those drawn from the theory before the update.

Given that today's planning languages like PDDL (Fox & Long 2003) go well beyond STRIPS, it seems natural to ask whether Lin and Reiter's results can be extended along these lines. One advantage would be to have a uniform framework for specifying the semantics of planning languages. Perhaps more importantly, as we will argue in more detail at the end of the paper, this would also provide a foundation to bring together the planning and action language paradigms, which have largely developed independently after the invention of STRIPS.

In this paper we propose a first step in this direction by considering the ADL fragment of PDDL (Pednault 1989; 1994). In contrast to Lin and Reiter, we use a new variant of the situation calculus called $\mathcal{ES}$ recently proposed by Lakemeyer and Levesque (Lakemeyer & Levesque 2004). This has at least two advantages: for one, there is no need to switch the language when translating formulas of the planning language into the new situation calculus because there are no situation terms to worry about (in $\mathcal{ES}$, situations occur only in the semantics); for another, semantic definitions like progression become simpler as it is no longer necessary to consider arbitrary first-order structures but only certain ones over a fixed universe of discourse. As Lakemeyer and Levesque recently showed (Lakemeyer & Levesque 2005), these simplifications do not lead to a loss of expressiveness. In fact, they show that second-order $\mathcal{ES}$ captures precisely the non-epistemic fragment of the situation calculus and the action language Golog (Levesque *et al.* 1997).[1]

The main technical contributions of this paper are the following: we show how to translate an ADL problem description into a basic action theory of $\mathcal{ES}$; we develop a notion of progression, which is similar to that of Lin and Reiter but also simpler given the semantics underlying $\mathcal{ES}$; finally, we establish that updating an ADL database (called a *state*) after performing an action is correct in the sense that the resulting state corresponds precisely to progressing the corresponding basic action theory. The result is obtained for both closed and open-world states.

With the exception of Lin and Reiter (Lin & Reiter 1997), the approaches to giving semantics to planning languages have all been meta-theoretic. When Pednault introduced ADL (1989; 1994), he provided a semantics that defined operators as mappings between first-order structures that are defined by additions and deletions of tuples to the relations and functions of that structures. He presented a method of deriving a situation calculus axiomatization from ADL operator schema, but did not show the semantic correspondence between the two. Despite the fact that PDDL was built upon ADL, it was not until PDDL2.1 that a formal semantics was provided. The focus in (Fox & Long 2003) was more on formalizing the meaning of the newly introduced temporal extensions and concurrent actions; nonetheless, the predicate-logic subset of Fox and Long's semantics represents a generalization of Lifschitz' state transition semantics for STRIPS (1986). However, they compile conditional effects into the preconditions of the operators, propositionalize quantifiers and only consider the case of complete state descriptions. An exhaustive study of the expressiveness and compilability of different subsets of the propositional version of ADL is given in (Nebel 2000).

The paper proceeds as follows. We first introduce $\mathcal{ES}$ and show how basic action theories are formulated in this logic.

[1]The correspondence with the full situation calculus is close but not exact.

Next, we define ADL problem descriptions and provide a formal semantics by mapping them into basic action theories. We then define progression and establish the correctness of updating an ADL state with respect to progression. Before concluding, we give an outlook on applying the results to combine planning and the action language Golog.

The Logic $\mathcal{ES}$

For the purpose of this paper, we only need the objective (i.e. non-epistemic), first-order subset of $\mathcal{ES}$.

The Language

The language consists of formulas over symbols from the following vocabulary:

- variables $V = \{x_1, x_2, \ldots, y_1, y_2, \ldots, a_1, a_2, \ldots\}$;
- fluent predicates of arity k: $F^k = \{F_1^k, F_2^k, \ldots\}$; for example, *At*;
  we assume this list includes the distinguished predicate *Poss*;
- rigid functions of arity k: $G^k = \{g_1^k, g_2^k, \ldots\}$; for example, *paycheck*, *moveB*;
- connectives and other symbols: $=$, $\wedge$, $\neg$, $\forall$, $\Box$, round and square parentheses, period, comma.

For simplicity, we do not include rigid (non-fluent) predicates or fluent (non-rigid) functions. The *terms* of the language are the least set of expressions such that

1. Every first-order variable is a term;
2. If $t_1, \ldots, t_k$ are terms and $g \in G^k$, then $g(t_1, \ldots, t_k)$ is a term.

We let R denote the set of all ground terms. For simplicity, instead of having variables of the *action* sort distinct from those of the *object* sort as in the situation calculus, we lump both of these together and allow ourselves to use any term as an action or as an object. Finally, the *well-formed formulas* of the language form the least set such that

1. If $t_1, \ldots, t_k$ are terms and $F \in F^k$, then $F(t_1, \ldots, t_k)$ is an (atomic) formula;
2. If t_1 and t_2 are terms, then $(t_1 = t_2)$ is a formula;
3. If t is a term and α is a formula, then $[t]\alpha$ is a formula;
4. If α and β are formulas, then so are $(\alpha \wedge \beta)$, $\neg\alpha$, $\forall x.\alpha$, $\Box\alpha$.

We read $[t]\alpha$ as "α holds after action t" and $\Box\alpha$ as "α holds after any sequence of actions". As usual, we treat $\exists x.\alpha$, $(\alpha \vee \beta)$, $(\alpha \supset \beta)$, and $(\alpha \equiv \beta)$ as abbreviations. We call a formula without free variables a *sentence*.

In the following, we will call a sentence *fluent*, when it does not contain $\Box$ and $[t]$ operators and does not mention *Poss*. In addition, we introduce the following special notation: A *type* τ is a symbol from F^1, i.e. a unary predicate. Then we define:

$$\forall x{:}\tau.\,\phi \stackrel{def}{=} \forall x.\,\tau(x) \supset \phi$$

We will often use the vector notation to refer to a tuple of terms ($\vec{t}$) or types ($\vec{\tau}$). If $\vec{r}$ denotes $r_1, \ldots, r_k$ and $\vec{t}$ stands for $t_1, \ldots, t_k$, then $(\vec{r} = \vec{t})$ means $(r_1 = t_1) \wedge \cdots \wedge (r_k = t_k)$. Further, $\vec{\tau}(\vec{t})$ serves as an abbreviation for $\tau_1(t_1) \wedge \cdots \wedge \tau_k(t_k)$.

The semantics

Intuitively, a world w will determine which fluents are true, but not just initially, also after any sequence of actions. Formally, let P denote the set of all pairs $\sigma{:}\rho$ where $\sigma \in R^*$ is considered a sequence of actions, and $\rho = F(r_1, \ldots, r_k)$ is a ground fluent atom from F^k. A *world* then is a mapping from P to truth values $\{0, 1\}$.

First-order variables are interpreted substitutionally over the rigid terms R, that is, R is treated as being isomorphic to a fixed universe of discourse. This is similar to $\mathcal{L}$ (Levesque & Lakemeyer 2001), where standard names are used as the domain.

Here is the complete semantic definition: Given a world w, for any formula α with no free variables, we define $w \models \alpha$ as $w, \langle\rangle \models \alpha$ where $\langle\rangle$ denotes the empty action sequence and

$w, \sigma \models F(r_1, \ldots, r_k)$ iff $w[\sigma{:}F(r_1, \ldots, r_k)] = 1$;
$w, \sigma \models (r_1 = r_2)$ iff r_1 and r_2 are identical;
$w, \sigma \models (\alpha \wedge \beta)$ iff $w, \sigma \models \alpha$ and $w, \sigma \models \beta$;
$w, \sigma \models \neg\alpha$ iff $w, \sigma \not\models \alpha$;
$w, \sigma \models \forall x.\, \alpha$ iff $w, \sigma \models \alpha_r^x$, for every $r \in R$;
$w, \sigma \models [r]\alpha$ iff $w, \sigma \cdot r \models \alpha$;
$w, \sigma \models \Box\alpha$ iff $w, \sigma \cdot \sigma' \models \alpha$, for every $\sigma' \in R^*$;

The notation α_t^x means the result of simultaneously replacing all free occurrences of the variable x by the term t; $\sigma_1 \cdot \sigma_2$ denotes the concatenation of the two action sequences.

When Σ is a set of sentences and α is a sentence, we write $\Sigma \models \alpha$ (read: Σ logically entails α) to mean that for every w, if $w \models \alpha'$ for every $\alpha' \in \Sigma$, then $w \models \alpha$. Finally, we write $\models \alpha$ (read: α is valid) to mean $\{\} \models \alpha$.

Basic Action Theories

As shown in (Lakemeyer & Levesque 2004), we are able to define basic action theories in a way very similar to those originally introduced by Reiter:

Given a set $\mathcal{F}$ of fluent predicates, a set of sentences Σ is called a *basic action theory* over $\mathcal{F}$ iff it only mentions the fluents in $\mathcal{F}$ and is of the form $\Sigma = \Sigma_0 \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}}$, where

- Σ_0 is a finite set of fluent sentences,
- Σ_{pre} is a singleton of the form[2] $\Box Poss(a) \equiv \pi$, where π is fluent with a being the only free variable;
- Σ_{post} is a finite set of successor state axioms of the form[3] $\Box[a]F(\vec{x}) \equiv \gamma_F$, one for each fluent $F \in \mathcal{F} \setminus \{Poss\}$, where γ_F is a fluent sentence whose free variables are among $\vec{x}$ and a.

[2] We follow the convention that free variables are universally quantified from the outside. We also assume that $\Box$ has lower syntactic precedence than the logical connectives, so that $\Box Poss(a) \equiv \pi$ stands for $\forall a.\Box(Poss(a) \equiv \pi)$.

[3] The $[t]$ construct has higher precedence than the logical connectives. So $\Box[a]F(\vec{x}) \equiv \gamma_F$ abbreviates $\forall a.\Box([a]F(\vec{x}) \equiv \gamma_F)$.

The idea is that Σ_0 represents the initial database, Σ_{pre} is one large precondition axiom and Σ_{post} the set of successor state axioms for all fluents in $\mathcal{F}$ (incorporating Reiter's solution (1991) to the frame problem).

The ADL subset of PDDL

ADL was proposed by Pednault (1989) as a planning formalism that constitutes a compromise between the highly expressive situation calculus on the one hand and the computationally more beneficial STRIPS language on the other. Recently, it has been used as the basis for the definition of a general planning domain definition language called PDDL (Ghallab *et al.* 1998; Gerevini & Long 2005a).

Here, we are interested in the ADL subset of PDDL, i.e. the language we obtain by only allowing the `:adl` requirement to be set. This implies that, beyond the definition of standard STRIPS operators, equality is supported as built-in predicate and preconditions may contain negation, disjunction and quantification (therefore they are normal first-order formulas using the domain's predicates together with the action's object parameters and the domain objects as the only function symbols). Further, conditional effects are allowed and objects may be typed.

ADL Operators: The General Form

Formally, an *ADL operator* A is given by a triple $(\vec{y}{:}\vec{\tau}, \pi_A, \epsilon_A)$, where $\vec{y}{:}\vec{\tau}$ is a list of variable symbols with associated types[4], π_A is a precondition formula with free variables among $\vec{y}$ and ϵ_A is an effect formula with free variables among $\vec{y}$. The $\vec{y}$ are the action's *parameters*, π_A is called the *precondition* and ϵ_A the *effect* of A, for short. The name of the operator A has to be a symbol from G^p (the function symbols of arity p), where p is the number of parameters $\vec{y}$ of A (possibly zero).

A *precondition formula* is of the following form: An atomic formula $F(\vec{t})$ is a precondition formula, if each of the t_i is either a variable or constant (i.e. terms are not nested). Similarly, an equality atom $(t_1 = t_2)$ is a precondition formula, if each t_i is a variable or a constant. If ϕ_1 and ϕ_2 are precondition formulas, then so are $\phi_1 \land \phi_2$, $\neg\phi_1$ and $\forall x{:}\tau.\phi_1$[5].

The *effect formulas* are defined as follows: An atomic formula $F(\vec{t})$ is an effect formula, if each of the t_i is either a variable or a constant. Similarly, a negated atomic formula $\neg F(\vec{t})$ is an effect formula, if each t_i is a variable or a constant. If ψ_1 and ψ_2 are effect formulas, then $\psi_1 \land \psi_2$ and $\forall x{:}\tau.\psi_1$ are as well. If γ is a precondition formula and ψ is an effect formula not containing "$\Rightarrow$" and "$\forall$", then $\gamma \Rightarrow \psi$ is an effect formula.

Therefore, effect formulas are always conjunctions of single effects. An effect of the form $\gamma \Rightarrow \psi$ is called a *conditional effect*. Nesting of conditional effects is disallowed.

[4] $\vec{y}{:}\vec{\tau}$ is to be understood as a list of pairs $y_i{:}\tau_i$.

[5] Recall that $\lor$, $\exists$ etc. are treated as abbreviations, therefore disjunction and existential quantification is allowed as well.

ADL Operators: The Normal Form

We say that an ADL operator A is in *normal form*, if its effect ϵ_A looks as follows:

$$\begin{array}{l} \bigwedge_{F_j} \forall \vec{x_j}{:}\vec{\tau_{F_j}}.\ (\gamma^+_{F_j,A}(\vec{x_j}) \Rightarrow F_j(\vec{x_j})) \land \\ \bigwedge_{F_j} \forall \vec{x_j}{:}\vec{\tau_{F_j}}.\ (\gamma^-_{F_j,A}(\vec{x_j}) \Rightarrow \neg F_j(\vec{x_j})) \end{array} \qquad (1)$$

We mean here that for each F_j, there is *at most* one conjunct $\cdots \Rightarrow F_j(\vec{x})$ and also at most one conjunct $\cdots \Rightarrow \neg F_j(\vec{x})$; neither is it required that there are conjuncts for all predicates of a theory nor is the ordering important.

ADL Problem Descriptions

A problem description for ADL now is given by:

1. a finite list of types $\tau_1, \ldots, \tau_l$, *Object* (*Object* is a special type that has to be always included);
   - along with this a finite list of statements of the form
   $$\tau_i{:}(\text{either } \tau_{i_1} \cdots \tau_{i_{k_i}}) \qquad (2)$$
   defining some of the types as compound[6]types; a *primitive type* is one that does not appear on the left-hand side of such a definition and is distinct from *Object*;
2. a finite list of fluent predicates $F_1, \ldots, F_n$;
   - associated with each F_j a list of types $\tau_{j_1}, \ldots, \tau_{j_{k_j}}$ (one for each argument of F_j)
3. a finite list of objects with associated primitive types $o_1{:}\tau_{o_1}, \ldots, o_k{:}\tau_{o_k}$ (object symbols are taken from G^0);
4. a finite list of ADL operators $A_1, \ldots, A_m$ (with associated descriptions in the above general form);
5. an initial state I (see below) and
6. a goal description G in form of a precondition formula.

I and G may only contain the symbols from items 1 to 3; the formulas in the descriptions of the A_i have to be constructed using only these symbols and the respective operator's parameters. We further require that all the symbols are distinct. In particular, this forbids using a type also as an F_j and using an object also as an A_i.

The purpose of the *Object* type is to serve as a dummy whenever an action argument, predicate argument or object is not required to be of any specific type. All objects o_i are implicitly of this type; *Object* is a super-type of all other types. Therefore, it is not allowed to appear anywhere in an "either" statement.

In the case of closed-world planning, the initial state description I is simply given by a finite set of ground fluent atoms $F(\vec{r})$; the truth value of non-appearing atoms is assumed to be FALSE. When we are doing open-world planning, I is defined by a finite set of ground atoms $F(\vec{r})$ and negated ground atoms $\neg F(\vec{r})$ (literals); non-appearing atoms are assumed to have an initially unknown truth value (I is a *belief state*, i.e. a representation of a *set* of possible world states).

[6] τ_i is to be understood as the union of the τ_{i_j}. For simplicity, we assume that these definitions do not contain cycles, although one can think of examples where this would make sense (e.g. defining two types as equal).

Example

For illustration, let's consider a variant of Pednault's well-known briefcase example (Pednault 1988; Ghallab *et al.* 1998) dealing with transporting objects between home and work using a briefcase. We have the following problem description:

1. Types:

   Object, *Item*, *Location*

   The *Object* type is the general superclass introduced above. *Item*s are objects that may be transported. A *Location* is a place where we may move objects to.

2. Predicates with associated types of arguments:

   $At(Item, Location), In(Item)$

   $At(x_1, x_2)$ denotes that item x_1 currently is at location x_2, $In(x_1)$ means that x_1 is in the briefcase.

3. Objects with associated types:

   briefcase:*Item*, *paycheck*:*Item*, *dictionary*:*Item*
   office:*Location*, *home*:*Location*

4. Operators:

   $$\begin{aligned}&moveB = \\ &\quad(\langle y_1{:}Location,\ y_2{:}Location\rangle, \\ &\quad At(briefcase, y_1) \wedge \neg(y_1 = y_2), \\ &\quad At(briefcase, y_2) \wedge \neg At(briefcase, y_1) \wedge \\ &\qquad \forall z{:}Item.\ In(z) \Rightarrow At(z, y_2) \wedge \neg At(z, y_1))\end{aligned}$$

   The briefcase can be moved from y_1 to y_2 if it is at the starting location y_1 and y_1 is not identical to the destination y_2. After moving, the briefcase is at the destination and no longer at the starting location, which equally holds for everything that is in the briefcase.

   $$\begin{aligned}&putInB = \\ &\quad(\langle y_1{:}Item,\ y_2{:}Location\rangle, \\ &\quad \neg(y_1 = briefcase), \\ &\quad At(y_1, y_2) \wedge At(briefcase, y_2) \Rightarrow In(y_1))\end{aligned}$$

   This operator allows to put something into the briefcase, if it is not the briefcase itself. When applied to an object that is not at the same location as the briefcase, the action has no effect.

   $takeOutOfB =$

   $(\langle y_1{:}Item\rangle, In(y_1), \neg In(y_1))$

   Something is removed from the briefcase.

   $emptyB =$

   $(\langle\rangle, \text{TRUE}, \forall z{:}Item.\ In(z) \Rightarrow \neg In(z))$

   Everything is removed from the briefcase.

5. Initial State (in a closed world):

   $$\begin{aligned}I = \{&At(briefcase, home),\ At(paycheck, home), \\ &At(dictionary, home),\ In(paycheck)\}\end{aligned}$$

6. Goal description:

   $$\begin{aligned}G = &At(briefcase, office) \wedge At(dictionary, office) \wedge \\ &At(paycheck, home)\end{aligned}$$

Mapping ADL to $\mathcal{ES}$

In this section, we generalize the approach of (Lin & Reiter 1997) for STRIPS to show that applying ADL operators can as well be expressed as a certain form of first-order progression in the situation calculus $\mathcal{ES}$. Below, we will construct, given an ADL problem description in normal form, a corresponding basic action theory Σ. The restriction to normal-form ADL is no loss of generality, as the following theorem shows.

Theorem 1 *The operators of an ADL problem description can always be transformed into an equivalent normal form.*

Here are the operators from the example, put into normal form:

$$\begin{aligned}&moveB = \\ &\quad(\langle y_1{:}Location,\ y_2{:}Location\rangle, \\ &\quad At(briefcase, y_1) \wedge \neg(y_1 = y_2), \\ &\quad \forall x_1{:}Item.\ \forall x_2{:}Location. \\ &\qquad ((x_1 = briefcase \vee In(x_1)) \wedge x_2 = y_2) \\ &\qquad\quad \Rightarrow At(x_1, x_2) \wedge \\ &\quad \forall x_1{:}Item.\ \forall x_2{:}Location. \\ &\qquad ((x_1 = briefcase \vee In(x_1)) \wedge x_2 = y_1) \\ &\qquad\quad \Rightarrow \neg At(x_1, x_2))\end{aligned}$$

$$\begin{aligned}&putInB = \\ &\quad(\langle y_1{:}Item,\ y_2{:}Location\rangle, \\ &\quad \neg(y_1 = briefcase), \\ &\quad \forall x_1{:}Item.\ At(x_1, y_2) \wedge At(briefcase, y_2) \Rightarrow In(x_1))\end{aligned}$$

$$\begin{aligned}&takeOutOfB = \\ &\quad(\langle y_1{:}Item\rangle, \\ &\quad In(y_1), \\ &\quad \forall x_1{:}Item.\ (x_1 = y_1) \Rightarrow \neg In(x_1))\end{aligned}$$

$$\begin{aligned}&emptyB = \\ &\quad(\langle\rangle, \\ &\quad \text{TRUE}, \\ &\quad \forall x_1{:}Item.\ In(x_1) \Rightarrow \neg In(x_1))\end{aligned}$$

The Successor State Axioms Σ_{post}

It is not a coincidence that the normal form (1) resembles Reiter's (1991) normal form effect axioms which are combined out of individual positive and negative effects and which he then uses to construct his successor state axioms as a solution to the frame problem. Generalizing his approach (also applied in (Pednault 1994)), we transform a set of operator descriptions to a set of successor state axioms as follows, assuming (without loss of generality by Theorem 1) that all operators are given in normal form. Let

$$\gamma^+_{F_j} \stackrel{def}{=} \bigvee_{\gamma_{F_j,A_i} \in NF(A_i)} \exists \vec{y_i}.a = A_i(\vec{y_i}) \wedge \gamma^+_{F_j,A_i} \qquad (3)$$

By "$\gamma_{F_j,A_i} \in NF(A_i)$" we mean that there only is a disjunct for A_i, $1 \leq i \leq m$ if there really exists a γ_{F_j,A_i} in the normal form of the effect of A_i. Recall that the normal form did not require that there is a $\forall \vec{x_j}{:}\tau_{\vec{F_j}}.\ (\gamma^+_{F_j,A}(\vec{x_j}) \Rightarrow$

$F_j(\vec{x_j}))$ for every F_j of the domain. We only obtain ones for F_j that did already appear[7] positively in the original effect of A.

Using a similar definition for $\gamma^-_{F_j}$, we get the successor state axiom for F_j:

$$\Box[a]F_j(\vec{x_j}) \equiv \gamma^+_{F_j} \wedge \tau_{\vec{F_j}}(\vec{x_j}) \vee F_j(\vec{x_j}) \wedge \neg\gamma^-_{F_j} \quad (4)$$

Differing from the usual construction, we introduced the conjunct $\tau_{\vec{F_j}}(\vec{x_j})$ to ensure that F_j can only become true for instantiations of the $\vec{x_j}$ that are consistent with the type definitions for F_j's arguments.

For each type τ_i, we additionally include a successor state axiom

$$\Box[a]\tau_i(x) \equiv \tau_i(x) \quad (5)$$

to define it as a situation-independent predicate (recall that by the definition of the semantics, all predicates are initially assumed to be fluent).

In the example, we get

$$\begin{aligned}
\gamma^+_{\mathit{At}} &= \exists y_1.\exists y_2.a = \mathit{moveB}(y_1, y_2) \wedge && (6)\\
&\quad ((x_1 = \mathit{briefcase} \vee \mathit{In}(x_1)) \wedge x_2 = y_2) &&\\
\gamma^-_{\mathit{At}} &= \exists y_1.\exists y_2.a = \mathit{moveB}(y_1, y_2) \wedge && (7)\\
&\quad ((x_1 = \mathit{briefcase} \vee \mathit{In}(x_1)) \wedge x_2 = y_1) &&\\
\gamma^+_{\mathit{In}} &= \exists y_1.\exists y_2.a = putInB(y_1, y_2) \wedge && (8)\\
&\quad (\mathit{At}(x_1, y_2) \wedge \mathit{At}(\mathit{briefcase}, y_2)) &&\\
\gamma^-_{\mathit{In}} &= \exists y_1.a = \mathit{takeOutOfB}(y_1) \wedge (x_1 = y_2) \vee && (9)\\
&\quad a = \mathit{emptyB} \qquad \wedge \mathit{In}(x_1) &&
\end{aligned}$$

Notice that, as stated above, not all operators are mentioned in γ^+_{At}, but only those that possibly cause a positive truth value for *At*. Therefore, the construction presented here still incorporates a solution to the frame problem. Our Σ_{post} now consists of the following sentences:

$$\begin{aligned}
\Box[a]\mathit{At}(x_1, x_2) &\equiv \gamma^+_{\mathit{At}} \wedge \mathit{Item}(x_1) \wedge \mathit{Location}(x_2)\\
&\quad \vee \mathit{At}(x_1, x_2) \wedge \neg\gamma^-_{\mathit{At}}\\
\Box[a]\mathit{In}(x_1) &\equiv \gamma^+_{\mathit{In}} \wedge \mathit{Item}(x_1)\\
&\quad \vee \mathit{In}(x_1) \wedge \neg\gamma^-_{\mathit{In}}\\
\Box[a]\mathit{Object}(x) &\equiv \mathit{Object}(x)\\
\Box[a]\mathit{Item}(x) &\equiv \mathit{Item}(x)\\
\Box[a]\mathit{Location}(x) &\equiv \mathit{Location}(x)
\end{aligned}$$

The Precondition Axiom Σ_{pre}

Further, a precondition axiom can be obtained in a similar fashion, that is a case distinction for all operators of the problem domain:

$$\pi \stackrel{def}{=} \bigvee_{1 \le i \le m} \exists \vec{y_i}\mathbf{:}\vec{\tau_i}.a = A_i(\vec{y_i}) \wedge \pi_{A_i} \quad (10)$$

The types $\vec{\tau_i}$ are those stated in the parameter list of A_i, and π_{A_i} simply is the unmodified precondition for the operator A_i. In our example, we obtain:

$$\begin{aligned}
\pi = \;& \exists y_1\mathbf{:}\mathit{Location}.\exists y_2\mathbf{:}\mathit{Location}.a = \mathit{moveB}(y_1, y_2) \wedge\\
&\quad \mathit{At}(\mathit{briefcase}, y_1) \wedge \neg(y_1 = y_2) \vee\\
& \exists y_1\mathbf{:}\mathit{Item}.\exists y_2\mathbf{:}\mathit{Location}. \quad a = \mathit{putInB}(y_1, y_2) \wedge\\
&\quad \neg(y_1 = \mathit{briefcase}) \vee\\
& \exists y_1\mathbf{:}\mathit{Item}. \quad a = \mathit{takeOutOfB}(y_1) \wedge\\
&\quad \mathit{In}(y_1) \vee\\
&\quad a = \mathit{emptyB} \wedge\\
&\quad \text{TRUE}
\end{aligned} \quad (11)$$

The Initial Description Σ_0

Finally, we are left with defining the initial description Σ_0. Here, we not only have to encode the information about the initial state of the world, but also everything that is concerned with the typing of objects. For all "either" statements of the form (2), we need a sentence

$$\tau_i(x) \equiv \tau_{i_1}(x) \vee \cdots \vee \tau_{i_{k_i}}(x) \quad (12)$$

in Σ_0. Further, we include

$$F_j(x_{j_1}, \ldots, x_{j_{k_j}}) \supset \tau_{j_1}(x_{j_1}) \wedge \cdots \wedge \tau_{j_{k_j}}(x_{j_{k_j}}) \quad (13)$$

for each type declaration of predicate arguments. Next, for each primitive type τ_i such that $o_{j_1}, \ldots, o_{j_{k_i}}$ are all objects declared to be of type τ_i, we include the sentence

$$\tau_i(x) \equiv x = o_{j_1} \vee \cdots \vee x = o_{j_{k_i}} \quad (14)$$

The final sentence needed for translating the type definitions is

$$\mathit{Object}(x) \equiv \tau_1(x) \vee \cdots \vee \tau_l(x) \quad (15)$$

Although the above sentences in themselves only establish type consistency in the initial situation (there are no $\Box$ operators here), the special form of Σ_{post} defined earlier ensures that these facts will remain true in successor situations. More precisely, we have here an example where state constraints are resolved by compiling them into successor state axioms (Lin & Reiter 1994).

We now come to the encoding of the actual initial world state. In the case of a closed world, we include for each F_j the sentence

$$F_j(\vec{x_j}) \equiv \vec{x_j} = \vec{o_1} \vee \cdots \vee \vec{x_j} = \vec{o_{k_o}} \quad (16)$$

assuming that $F_j(\vec{o_1}), \ldots, F_j(\vec{o_{k_o}})$ are all the atoms in I mentioning F_j. If we are however dealing with an open-world problem, we instead include the sentence

$$\vec{x_j} = \vec{o_1} \vee \cdots \vee \vec{x_j} = \vec{o_{k_o}} \supset F_j(\vec{x_j}), \quad (17)$$

where $F_j(\vec{o_1}), \ldots, F_j(\vec{o_{k_j}})$ are all the positive literals in I using F_j; and the sentence

$$\vec{x_j} = \vec{o_1} \vee \cdots \vee \vec{x_j} = \vec{o_{k_o}} \supset \neg F_j(\vec{x_j}) \quad (18)$$

when $\neg F_j(\vec{o_1}), \ldots, \neg F_j(\vec{o_{k_o}})$ are all the negative literals in I using F_j.

[7] more precisely for those F_j appearing positively in ϵ_A outside of the antecedent γ of a conditional effect $\gamma \Rightarrow \psi$

In our closed-world example, we end up with a Σ_0 consisting of:

$$
\begin{array}{rcl}
At(x_1, x_2) & \supset & (Item(x_1) \wedge Location(x_2)) \\
In(x_1) & \supset & Item(x_1) \\
Item(x) & \equiv & ((x = briefcase) \vee (x = paycheck) \vee \\
& & \quad (x = dictionary)) \\
Location(x) & \equiv & ((x = office) \vee (x = home)) \\
Object(x) & \equiv & (Item(x) \vee Location(x)) \\
At(x_1, x_2) & \equiv & ((x_1 = briefcase \wedge x_2 = home) \vee \\
& & (x_1 = paycheck \wedge x_2 = home) \vee \\
& & (x_1 = dictionary \wedge x_2 = home)) \\
In(x_1) & \equiv & (x_1 = paycheck)
\end{array}
$$

Correctness

Finally, we will show the correspondence between the state-transitional semantics for ADL of adding and deleting literals and first-order progression in $\mathcal{ES}$. The following definition is derived from Lin and Reiter's original proposal of progression, but is simpler due to the fact that we do not need to consider arbitrary first-order structures.

A set of sentences Σ_r is a *progression* of Σ_0 through a ground term r (wrt Σ_{pre} and Σ_{post}) iff:

1. all sentences in Σ_r are fluent in $\langle r \rangle$;
2. $\Sigma_0 \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}} \models \Sigma_r$;
3. for every world w_r with $w_r \models \Sigma_r \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}}$, there is a world w with $w \models \Sigma_0 \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}}$ such that:

   $$w_r, r \cdot \sigma \models F(\vec{t}) \quad \text{iff} \quad w, r \cdot \sigma \models F(\vec{t})$$

   for all $\sigma \in R^*$ and all primitive formulas $F(\vec{t})$ such that $F \in \mathcal{F}$ (including *Poss*).

A formula that is *fluent in* $\langle r \rangle$ is one which is equivalent to $[r]\phi$ for some fluent[8] formula ϕ, i.e. it only talks about the fluents' values in the situation $\langle r \rangle$. Intuitively, for an observer standing in the situation after r was performed (and only looking "forward" in time), it is impossible to distinguish between a world w satisfying the original theory Σ and a world w_r that satisfies $\Sigma_r \cup \Sigma_{\text{pre}} \cup \Sigma_{\text{post}}$.

We now want to address the issue of how to obtain such a progression. The result will be that for a basic action theory that is a translation from an ADL problem (and therefore the member of a restricted subclass of the general form of action theories), it is quite easy to construct such a set. Given an ADL problem description and an action $A(\vec{p})$ (i.e. an operator and object symbols as instantiations for A's parameters), we make, under the condition that $\Sigma_0 \cup \Sigma_{\text{pre}} \models Poss(A(\vec{p}))$, the following modifications to the state description I:

- in the case of closed-world planning:
  - for all objects $\vec{o}$ and all fluent predicates F_j such that $F_j(\vec{o})$ is type-consistent
    if $\Sigma_0 \models \gamma^{+\ \vec{x_j}\ a}_{F_j\ \vec{o}\ A(\vec{p})}$: add $F_j(\vec{o})$
  - for all objects $\vec{o}$ and all fluent predicates F_j such that $F_j(\vec{o})$ is type-consistent
    $\Sigma_0 \models \gamma^{-\ \vec{x_j}\ a}_{F_j\ \vec{o}\ A(\vec{p})}$: delete $F_j(\vec{o})$
- in the case of open-world planning:
  - for all objects $\vec{o}$ and all fluent predicates F_j such that $F_j(\vec{o})$ is type-consistent
    $\Sigma_0 \models \gamma^{+\ \vec{x_j}\ a}_{F_j\ \vec{o}\ A(\vec{p})}$: add $F_j(\vec{o})$, delete $\neg F_j(\vec{o})$
  - for all objects $\vec{o}$ and all fluent predicates F_j such that $F_j(\vec{o})$ is type-consistent
    $\Sigma_0 \models \gamma^{-\ \vec{x_j}\ a}_{F_j\ \vec{o}\ A(\vec{p})}$: add $\neg F_j(\vec{o})$, delete $F_j(\vec{o})$

If we denote the set of literals to be added by $Adds$ and the ones to be deleted by $Dels$, then the new state description is

$$I' = (I \setminus Dels) \cup Adds.$$

Here it is assumed that we only consider symbols (objects, fluents, operators) that appear in the given problem description, which yields only finitely many combinations. The fact that we only have to check type-consistent atoms further restricts the number of atoms to be treated. Formally, $F_j(\vec{o})$ being *type-consistent* means that $\Sigma_0 \cup \{\tau_{\vec{F_j}}(\vec{o})\}$ is satisfiable.

Theorem 2 *Let I' be the set obtained in the above construction applied to a given (closed- or open-world) ADL problem description and a ground action $r = A(\vec{p})$. Further let*

$$\Sigma_r = \{[r]\psi \mid \psi \in \Sigma_0(I')\},$$

where $\Sigma_0(I')$ means the result of applying the constructions in (12)-(18) to I' instead of I. For all fluent predicates F_j in the problem description, let the consistency condition

$$\models (\gamma^{+}_{F_j} \wedge \gamma^{-}_{F_j})^{a}_{r}$$

hold. Then Σ_r is a progression of Σ_0 through r

- *in the closed-world case;*
- *in the open-world case only under the additional condition that whenever for some $\gamma^{*}_{F_j}$ (where $* \in \{+,-\}$) it holds that $\Sigma_0 \cup \{\gamma^{*\ \vec{x_j}\ a}_{F_j\ \vec{o}\ r}\}$ is satisfiable, then*

  $$\Sigma_0 \models \gamma^{*\ \vec{x_j}\ a}_{F_j\ \vec{o}\ r}.$$

For space reasons, we will not present a proof here. The main reason for being able to establish the result lies in the finiteness of the domain to be considered. Whereas $\mathcal{ES}$'s semantics assumes a domain with countably infinite many objects and actions, PDDL, as a language that is used in practical implementations, only allows problem domains with a finite number of operators and items. We utilize the typing constructs to reconcile these two views.

The additional condition for open-world problems can be illustrated with a small example: consider an operator $A = (\emptyset, \text{TRUE}, P \Rightarrow Q_1 \wedge \neg P \Rightarrow Q_2)$ and an open world initial state description of $I = \emptyset$. Applying A to I leads to a situation whose state is described by $Q_1 \vee Q_2$, since it is both possible that P holds and that it does not hold. Obviously, the resulting state is not representable by a set of literals,

[8] Recall that our terminology contains both the notions of fluent predicates (like *At*) as well as that of fluent formulas (e.g. $\exists x.At(x, home) \wedge In(x)$); they should not be confused.

therefore we cannot apply the above progression scheme. In fact, mainly because of such undefined states, the open-world requirement is not included in the PDDL language definition anymore since version 2.1 (Fox & Long 2003), restricting its application to purely closed-world planning.

Notice that in the closed-word case, our special form of basic action theories constitutes a proper subclass of what is called "relatively complete databases" in (Lin & Reiter 1997). It is therefore not surprising that a progression of such theories exists. Theorem 2 however additionally establishes that our class of action theories is also *closed under progression*, since the progression result Σ_r is of the same form as the original Σ_0. Progression steps may thus be applied iteratively.

On the other hand, in both the open- and closed-world case, Lin and Reiter's progression for "strongly context-free" theories (for which they show the correspondence to STRIPS) is a special case of our result. This agrees with what one would expect from the fact that ADL action descriptions can be viewed as a generalization of STRIPS operators.

Now let us return to our example again to see how the closed-world progression works in this case. We assume that we want to progress through the action *moveB*$(home, office)$ (abbreviated as m). The first thing to notice is that

$$\Sigma_0 \cup \Sigma_{\text{pre}} \models Poss(m)$$

iff, using (11), unique names for actions and the fact that *home* and *office* are both *Location*s,

$$\Sigma_0 \cup \Sigma_{\text{pre}} \models At(briefcase, home) \wedge \neg(home = office)$$

iff, with unique names for objects (recall that our semantics does not distinguish between objects and actions)

$$\Sigma_0 \cup \Sigma_{\text{pre}} \models At(briefcase, home)$$

which is obviously the case, therefore we may proceed. The reader may verify (considering (6) and (7)) that

- $\Sigma_0 \models \gamma^{+}{}_{At}{}^{a}_{m}{}^{x_1}_{briefcase}{}^{x_2}_{office}$
- $\Sigma_0 \models \gamma^{+}{}_{At}{}^{a}_{m}{}^{x_1}_{paycheck}{}^{x_2}_{office}$
- $\Sigma_0 \models \gamma^{-}{}_{At}{}^{a}_{m}{}^{x_1}_{briefcase}{}^{x_2}_{home}$
- $\Sigma_0 \models \gamma^{-}{}_{At}{}^{a}_{m}{}^{x_1}_{paycheck}{}^{x_2}_{home}$

and that these are all type-consistent instantiations for x_1, x_2 such that $\gamma^{+}{}_{At}{}^{a}_{m}$ respectively $\gamma^{-}{}_{At}{}^{a}_{m}$ are entailed by Σ_0. Because there are no disjuncts for *moveB* in (8) and (9), $\gamma^{+}{}_{In}{}^{a}_{m}$ and $\gamma^{-}{}_{In}{}^{a}_{m}$ are not entailed for any instantiation of x_1. The new initial state then is

$$\begin{aligned} I' = \{&At(dictionary, home),\ In(paycheck),\\ &At(briefcase, office),\ At(paycheck, office)\}. \end{aligned}$$

We obtain the progression Σ_m consisting of

$$\begin{array}{ll}
[m](At(x_1,x_2) & \supset (Item(x_1) \wedge Location(x_2))) \\
[m](In(x_1) & \supset Item(x_1)) \\
[m](Item(x) & \equiv ((x = briefcase) \vee (x = paycheck) \vee \\
 & \quad (x = dictionary))) \\
[m](Location(x) & \equiv ((x = office) \vee (x = home))) \\
[m](Object(x) & \equiv (Item(x) \vee Location(x))) \\
[m](At(x_1,x_2) & \equiv ((x_1 = dictionary \wedge x_2 = home) \vee \\
 & \quad (x_1 = briefcase \wedge x_2 = office) \vee \\
 & \quad (x_1 = paycheck \wedge x_2 = office))) \\
[m](In(x_1) & \equiv (x_1 = paycheck))
\end{array}$$

Notice that the only changes, compared to Σ_0, are the "$[m]$" in front of each formula (to denote the situation after m has been performed) and the new instances for *At*.

Outlook: Embedding ADL planning in Golog

The situation calculus (and, as (Lakemeyer & Levesque 2005) showed, also $\mathcal{ES}$) constitutes the foundation[9] on which the semantics of the agent programming language Golog (Levesque *et al.* 1997) is defined. The language gives a programmer the freedom to on the one hand specify the agent's behaviour only roughly by using nondeterministic constructs and where it is the system's task to find a deterministic strategy to achieve its goal. On the other hand, the programmer may utilize deterministic constructs known from imperative programming languages. Nonetheless there is some drawback with this general purpose approach which can be illustrated best by considering the following completely nondeterministic Golog program:

$$achieve(Goal) := \textbf{while}\ (\neg Goal)\ \textbf{do}\ (\pi a)\ a\ \textbf{endWhile}$$

The program corresponds to the task description of finding sequential plans: As long as the condition $Goal$ is not fulfilled, nondeterministically pick some action a and execute it. Although it is thus possible to do sequential planning in Golog, the performance of the Golog system can usually not compete with current state-of-the-art planners like FF (Hoffmann & Nebel 2001; Hoffmann 2003; Hoffmann & Brafman 2005), LPG (Gerevini *et al.* 2005), HSP2 (Bonet & Geffner 2001a), Fast Downward (Helmert & Richter 2004) or TLPlan (Bacchus & Ady 2001). The reason is that current Golog implementations resolve nondeterminism by a simple backtracking mechanism, whereas planners resort to efficient techniques like heuristic search, e.g. (Bonet & Geffner 2001b).

The idea now is, using the results presented here, to embed ADL-based planners (more precisely planners that take the ADL subset of PDDL as an input language) into Golog, to combine the benefits of both systems. We envision that whenever, during the execution of a Golog program, a planning problem arises (i.e. an $achieve(G)$ subgoal has to be solved), the necessary parts of the current situation and the subgoal are transformed into a planning problem instance and handed over to the planner. Once a solution (a sequence of actions) is found, it is transformed back into Golog and the program execution continues, where PDDL serves in

[9]Valid executions of Golog programs are expressed by a situation calculus (respectively $\mathcal{ES}$) formula that is entailed by the underlying basic action theory.

both cases as an interface language. The results in this paper show that, as long as the action theory underlying the Golog program is obtained by a translation from an ADL problem description, this method is semantically well-founded.

Conclusion

We presented an alternative definition for the semantics of ADL operators as progression in first-order $\mathcal{ES}$ knowledge bases. This establishes the basis for embedding existing state-of-the-art planners that take ADL as an input language, into an interpreter for the robot programming language Golog, to obtain a powerful language that is equally suited for autonomously constructing complete plans (utilizing the planner) and letting the programmer specify preconstructed plans with residual nondeterminism (by means of the usual Golog constructs) to be resolved by the system. Such an embedding into an $\mathcal{ES}$-based Golog interpreter (currently under development) is the focus of future work, as well as giving semantics to larger fragments of PDDL (Edelkamp & Hoffmann 2004; Gerevini & Long 2005b) with features such as numeric fluents, time, or preferences.

References

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *Proc. IJCAI-2001*, 417–424.

Bonet, B., and Geffner, H. 2001a. Heuristic Search Planner 2.0. *AI Magazine* 22(3):77–80.

Bonet, B., and Geffner, H. 2001b. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 195, Institut für Informatik, Universität Freiburg.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.* 2(3/4):189–208.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)* 20:61–124.

Gerevini, A., and Long, D. 2005a. BNF description of PDDL3.0. `http://zeus.ing.unibs.it/ipc-5/bnf.pdf`.

Gerevini, A., and Long, D. 2005b. Plan constraints and preferences in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia, Italy.

Gerevini, A.; Saetti, A.; Serina, I.; and Toninelli, P. 2005. Planning with derived predicates through rule-action graphs and relaxed-plan heuristics. Technical report, Universita degli Studi di Brescia, Dipartimento di Elettronica per l Automazione, Brescia, Italy.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL—the planning domain definition language. `ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz`.

Helmert, M., and Richter, S. 2004. Fast Downward – making use of causal dependencies in the problem representation. `http://ls5-www.cs.uni-dortmund.de/~edelkamp/ipc-4/Proc/downward.pdf`.

Hoffmann, J., and Brafman, R. 2005. Contingent planning via heuristic forward search with implicit belief states. In *Proc. ICAPS-05*, 71–80.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Lakemeyer, G., and Levesque, H. J. 2004. Situations, si! situation terms, no! In *Proc. KR2004*. AAAI Press.

Lakemeyer, G., and Levesque, H. J. 2005. Semantics for a useful fragment of the situation calculus. In *Proc. IJCAI-05*.

Levesque, H. J., and Lakemeyer, G. 2001. *The Logic of Knowledge Bases*. MIT Press.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31:59–84.

Lifschitz, V. 1986. On the semantics of STRIPS. In Georgeff, M. P., and Lansky, A. L., eds., *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, 1–9. Timberline, Oregon: Morgan Kaufmann.

Lin, F., and Reiter, R. 1994. State constraints revisited. *Journal of Logic and Computation* 4(5):655–678.

Lin, F., and Reiter, R. 1997. How to progress a database. *Artif. Intell.* 92(1-2):131–167.

Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *J. Artif. Intell. Res. (JAIR)* 12:271–315.

Pednault, E. P. D. 1988. Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence* 4:356–372.

Pednault, E. P. D. 1989. ADL: Exploring the middle ground between STRIPS and the Situation Calculus. In *Proc. KR1989*, 324–332. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Pednault, E. P. D. 1994. ADL and the state-transition model of action. *J. Log. Comput.* 4(5):467–512.

Reiter, R. 1991. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy* 359–380.

Reiter, R. 2001. *Knowledge in action : logical foundations for specifying and implementing dynamical systems*. Cambridge, Mass.: MIT Press. The frame problem and the situation calculus.