GOLOG and PDDL: What is the Relative Expressiveness?

Patrick Eyerich[1], Bernhard Nebel[1], Gerhard Lakemeyer[2], and Jens Claßen[2]

[1] Department of Computer Science, University of Freiburg, 79110 Freiburg, Germany
{eyerich,nebel}@informatik.uni-freiburg.de
[2] Department of Computer Science, RWTH Aachen, 52056 Aachen, Germany
{gerhard,classen}@informatik.rwth-aachen.de

Abstract. Action formalisms such as GOLOG or FLUX have been developed primarily for representing and reasoning about change in a logical framework. For this reason, expressivity was the main goal in the development of these formalisms. In another line of research, efficiency of planning methods was the topmost goal resulting in the basic STRIPS language, which has only moderate expressivity. The planning language PDDL developed since 1998 is an extension of basic STRIPS with many expressive features. Now the interesting question is how PDDL compares to GOLOG or other action languages from an expressivity point of view. We will show that a GOLOG fragment, which we call Restricted Basic Action Theories, is as expressive as the ADL fragment of PDDL. To prove this equivalence we use the compilation framework. From a practical point of view, this result can be used for employing efficient planners inside a GOLOG interpreter.

1 Introduction

While action formalisms and planning methods have the same roots [5], the research within these two fields has been developed independently from each other since the introduction of STRIPS [6]. While in the area of action formalisms expressivity was the main goal and efficiency only played a minor role, in the area of planning systems special emphasis was put on efficiently generating action plans. For this reason, only comparatively simple languages such as the basic variant of STRIPS (not containing any background theory) have been used for a long time.

However, in recent years one could observe some convergence. In particular, the development of the *Planning Domain Definition Language (PDDL)* with its extensions [1, 8], which can handle, among other things, conditional effects, time and continuous effects, has led to a state, where action formalisms and planning languages approach similar expressiveness, at least if we focus on linear sequences of actions.

In order to compare the expressiveness of PDDL and GOLOG, we will use the compilation approach [4], in which two planning formalisms are considered as equally expressive if planning domains and plans in the two formalisms can be translated into each other without creating an unreasonable blowup.

As a first step, we will restrict so-called Basic Action Theories, which are formulated in the situation calculus—the base of GOLOG—in a way such that they have the same expressive power as the ADL fragment of PDDL. That the two formalisms are indeed identical in expressive power is shown by giving a compilation scheme from Restricted Basic Action Theories to the ADL fragment of PDDL and vice versa. From a practical point of view, this result can be used for employing efficient planners inside a GOLOG interpreter.

The rest of the paper is structured as follows: In Section 2, we briefly introduce PDDL and its ADL fragment, which forms the relevant part for this work. In Section 3, we introduce the situation calculus and Basic Action Theories. We define the so-called *Restricted Basic Action Theories* and give compilation schemes for both directions in Section 4. Finally we summarize the obtained results in Section 5 and give an outlook of future work.

2 PDDL

PDDL was designed by Drew McDermott in 1998 [8] and since then has been continuously further developed in order to meet the needs of the bi-annual *International Planning Competition*. The most recent version is version 3.0 [2]. PDDL is based on the STRIPS formalism, but also offers many additional constructs like numeric fluents, durative actions and derived predicates.

A planning task in PDDL is divided into a domain description, which describes the behavior and a problem description, which contains specific objects, the initial state, the goal description and a metric which can be used to specify how to measure the quality of a plan. In the following we restrict ourselves to PDDL 2.1 Level 1 [1], which essentially corresponds to the ADL fragment [7] of the language, i.e., in addition to STRIPS, we have universally quantified conditional effects and arbitrary first-order formulas in preconditions, effect condition, and goal conditions. In the following, we will use PDDL to denote just the ADL fragment of PDDL.

A PDDL *planning task* is a tuple $P = \langle P', P_{Objects}, P_{Init}, P_{Goal} \rangle$ with the *domain description* $P' = \langle P_{Types}, P_{Predicates}, P_{Actions} \rangle$. We use the common state transition semantics of PDDL [1]. This means that *states* are truth assignments to the set of atoms formed by instantiating all predicate symbols with all possible objects, i.e., constant symbols, meeting the typing restrictions. Alternatively, we sometimes identify a state with the set of atoms that are true in this state. *Ground actions* are actions which have all their parameters instantiated. They are applicable in a state, if their preconditions are satisfied and their application changes the state according to the effects of the action.

A sequence of actions $a_1, \ldots, a_n$ is called a *plan* for P iff there exists a sequence of states $s_0, \ldots, s_n$ such that $s_0 = P_{Init}$, $s_n \models P_{Goal}$, and each action a_i is applicable in state s_{i-1} and produces state s_i.

3 Situation Calculus

GOLOG is an action formalism developed at the University of Toronto [3, 10]. It has been formalized using the the situation calculus. The situation calculus is a second-order language specifically designed for representing dynamically changing worlds. A *situation* in this formalism is nothing else than the history of actions [3]. The initial situation or empty history is denoted by s_0. Using the special binary function symbol do and an action term α, the situation after executing α is denoted by $do(\alpha, s)$.

Relations whose truth values vary from situation to situation are called relational fluents. Besides their regular arguments, fluents have one additional argument which by convention is the last argument. $At(x, y, s)$ for example means that object x is at position y in situation s. Furthermore, there are two binary predicates: $Poss$ and $\sqsubset$. $Poss(a, s)$ states under which conditions action a is applicable in situation s; $s \sqsubset s'$ defines an ordering relation on situations (which are interpreted as sequences of actions) and states that s is a proper subsequence of s'.

Dynamic domains are described by giving a precondition axiom for every action which is directly applicable by an agent (these are the so-called primitive actions), defining under which circumstances a situation dependent predicate (fluent) is true in a situation (this is done through the so-called successor state axioms (SSA)) and defining the initial situation of the world. Furthermore, there are some general situation calculus axioms and unique name axioms. All these together form a Basic Action Theory (BAT).

GOLOG also offers the possibility to define complex actions, e.g. test actions, sequences, nondeterministic choice of two actions, nondeterministic choice of action arguments as well as nondeterministic iteration. Additional constructors allow for defining procedures. All these constructs are interpreted as macros and for a given initial situation lead to unfolding the complex action to a sequence of primitive actions. We will ignore this possibility here for two reasons. First, it obviously lead to much conciser representations than PDDL plans. Furthermore, since we are interested in embedding a planner in GOLOG in order to generate a plan, we are only interested to consider linear sequences of primitive actions.

The starting point of our work are Basic Action Theories as defined by Reiter [3, page 58]. A Basic Action Theory T in its general form consists of five parts:

$$T = \Sigma \cup T_{SSA} \cup T_{PA} \cup T_{UNA} \cup T_{s_0}$$

where Σ is a set of foundational axioms for situations, T_{SSA} is a set of successor state axioms for functional and relational fluents, T_{PA} is a set of precondition axioms for actions, T_{UNA} is the set of unique names axioms for actions, and T_{s_0} is a set of first-order sentences that are uniform in s_0, so that s_0 is the only term of the sort *situation* mentioned by the sentences of T_{s_0}. Thus, no sentence of T_{s_0} quantifies over situations, or mentions $Poss$, $\sqsubset$ or the function symbol do. Finally, T_{s_0} describes the initial situation.

Basic Actions Theories in their general form are much more powerful constructs than PDDL (e.g. BAT's are not forced to represent a complete theory, and potentially there are infinitely many objects). At this point the question arises whether it can be shown for a useful part of the set of all Basic Action Theories that they are as expressive as PDDL.

We call a BAT without T_{s_0} a BAT domain $D = \langle \Sigma, T_{SSA}, T_{PA}, T_{UNA} \rangle$, and such a BAT domain together with T_{s_0}, and a situation calculus formula $G(s)$ with only free variable s a BAT task $I = \langle D, T_{s_0}, G(s) \rangle$. Similar to a PDDL plan, a plan for a BAT task I, is a variable-free situation term σ, for which it holds that $T \models executable(\sigma) \wedge G(\sigma)$, whereby $executable(s)$ means that all the actions occurring in the action sequence s can be executed one after the other.

In the following we give some restrictions which define such a part of this set and show that all tasks meeting these restrictions may be compiled into PDDL. In order to show that these restrictions are not too strict, we also present a compilation scheme for the other direction.

3.1 Restricted Basic Action Theories

In the following, we call a Basic Action Theory T *restricted* (RBAT) if it satisfies the following conditions:

E1 The arity of functional terms is constrained to be zero. So no functional terms apart from constants are allowed.

E2 All successor state axioms are in a special syntactic form. The SSA of a fluent F looks as follows:

$$F(x_1, ..., x_n, do(a, s)) \equiv \bigvee_{l=1}^{n} \phi_l \tag{1}$$

for a finite number n. Exactly one ϕ_l is of the form

$$\begin{array}{l} F(x_1, ..., x_n, s) \wedge \neg(\\ (\exists ...)([\varphi_1 \wedge](a = A_1(y_{11}, ..., y_{1m_1}))) \\ \vee(\exists ...)([\varphi_2 \wedge](a = A_2(y_{21}, ..., y_{2m_2}))) \\ \vee(\exists ...)([\varphi_3 \wedge](a = A_3(y_{31}, ..., y_{3m_3}))) \\ \vee ..) \end{array} \tag{2}$$

and all the other ϕ_l are of the form

$$(\exists ...)([\varphi_l \wedge](a = A(y_1, ..., y_m))) \ . \tag{3}$$

It is existentially quantified over the variables y_i for which it holds that $y_i \neq x_j$ for all j. These are exactly those variables which appear as a parameter of the action but not as an argument of the fluent. The formulas within brackets are optional. The φ_l are first order formulas without functional terms. Any free variables in the ϕ_l are implicitly universally quantified. Each SSA has to have exactly one term of form (2). Furthermore, each action has to appear at most one time in a SSA.
It can be seen that fluent F becomes true in situation $do(a, s)$ iff one of the terms ϕ_l is true. Each term ϕ_l is of one of the forms (2) or (3). In order to describe how F can become true, form (3) is used. F is true in situation $do(a, s)$ if the action a is equal to A. Optionally further conditions for the truth of F can be given by φ_l.
Form (3) states that F will be true in $do(a, s)$ if it was already true in s and the performed action was none of the A_i. As in (2) there can be some more conditions by a φ_l. The φ_l correspond to conditional effects in PDDL. Each SSA has to have exactly one term of form (2) because in PDDL predicates can change their truth value only through actions. So a fluent which is true has to remain true as long as it is not made false by an action, otherwise it has to stay true forever.

Example 1. In an example domain we assume to have trucks and humans. The fluent $At(x, y, s)$ describes that x is at position y in situation s whereby x can be a

truck or a human. In the following we assume that $In(x, y, s)$ was already defined to be true exactly when human x is inside truck y in situation s. Furthermore, we assume that there is an action $drive(t, p_1, p_2)$ which describes that truck t drives from position p_1 to position p_2.
The SSA of $At(x, y)$ can be defined in the following way:

$$\begin{aligned}
&At(x, y, do(a, s)) \equiv \\
&(\exists p_1)(a = drive(x, p_1, y)) \vee \\
&(\exists t, p_1)(In(x, t, s) \wedge a = drive(t, p_1, y)) \vee \\
&At(x, y, s) \wedge \neg(((\exists p_2)(a = drive(x, y, p_2))) \\
&\quad \vee ((\exists t, p_2)(In(x, t, s) \wedge a = drive(t, y, p_2))))
\end{aligned}$$

The first and the second disjunct of this example are of form (3) while the third is of form (2).

E3 T_{s_0} consists *only* of the following sentences:

1.) For each relational fluent F with $n + 1$ arguments there is a sentence of the form

$$\begin{aligned}
&F(\boldsymbol{x}, s_0) \equiv \\
&x_1 = d_{11} \wedge ... \wedge x_n = d_{1n} \\
&\vee ... \vee x_1 = d_{m1} \wedge ... \wedge x_n = d_{mn}
\end{aligned} \tag{4}$$

with m finite.

2.) For each situation independent predicate P with n arguments there is a sentence of the form

$$\begin{aligned}
&P(\boldsymbol{x}) \equiv \\
&x_1 = d_{11} \wedge ... \wedge x_n = d_{1n} \\
&\vee ... \vee x_1 = d_{m1} \wedge ... \wedge x_n = d_{mn}
\end{aligned} \tag{5}$$

with m finite.

3. There is a *domain closure axiom*:

$$\forall x(x = d_1 \vee ... \vee d_n)$$

for each constant d_i in T.

E4 There are unique names axioms for all functions of arity zero (constants).

4 Compilation Schemes

4.1 Introduction

To analyze the expressive power of different planning formalisms, Nebel introduced the so-called compilation scheme framework [4]. This is a formal tool which can be used to measure the relative expressive power of planning formalisms. The intuition behind it is that a formalism X is as expressive as a second Y if all domain descriptions of Y can be expressed concisely in X and the resulting plans are not blown up too much. Compilation schemes are solution-preserving mappings with polynomially sized results from Y domain structures to X domain structures. While we restrict the size of the result of a compilation scheme, we do not require any bounds on the computational resources for the compilation. In fact, for measuring the expressiveness, it is irrelevant whether the mapping is polynomial-time computable, exponential-time computable, or even non-recursive.[3]

Furthermore, it is required that the operators of the planning task can be translated without considering the initial state and the goal. This restriction guarantees that compilations are non-trivial. If the entire planning task could be transformed, a compilation scheme could decide the existence of a plan for the source task and then generate a small solution-preserving task in the target formalism, which would lead to the unintuitive conclusion that all planning formalisms have the same expressive power.

In addition to the resource requirements on the compilation process, one can distinguish between compilation schemes which preserve the size of the plans *exactly*, *linearly* and *polynomially*. *Exactly* means, that the plan in the target formalism should only be longer by a constant number of steps; *linearly* means that the plan should only be longer by a constant factor, and *polynomially* means that the plan length in the target formalism should be bounded by a polynom in the size of the source plan and the size of the source domain description [4].

Because we do not want to compare the expressive power of two planning formalisms but those of two quite different theories, we have slightly adjusted the definition of compilation schemes for our purposes.

Definition 1. *Assume a tuple of functions* ***f***$=\langle f_{Types}, f_{Predicates}, f_{Actions}, f_{Objects}, f_{Init}, f_{Goal}\rangle$ *that induces a function F from RBAT tasks $I = \langle D, T_{s_0}, G(s)\rangle$ to PDDL planning tasks $F(I)$ as follows:*

$$\begin{aligned} F(I) = \langle & f_{Types}(), f_{Predicates}(D), f_{Actions}(D), \\ & f_{Objects}(D, T_{s_0}), f_{Init}(T_{s_0}), f_{Goal}(G(s))\rangle \end{aligned}$$

If the following two conditions are satisfied, we call ***f*** *a compilation scheme from RBAT's to PDDL:*

K1 there exists a plan for I iff there exists a plan for $F(I)$.
K2 the size of the results of $f_{Types}, f_{Predicates}, f_{Actions}, f_{Objects}, f_{Init}$ and f_{Goal} is polynomial in the size of their arguments.

Definition 2. *Assume a tuple of functions* ***g***$=\langle g_{\Sigma}, g_{T_{SSA}}, g_{T_{PA}}, g_{T_{UNA}}, g_{T_{s_0}}, g_{Goal}\rangle$ *that induces a function G from PDDL planning tasks $P = \langle P', P_{Init}, P_{Goal}\rangle$ to RBAT tasks $G(P)$ as follows:*

$$\begin{aligned} G(P) = \langle & g_{\Sigma}, g_{T_{SSA}}(P'), g_{T_{PA}}(P'), g_{T_{UNA}}(P'), \\ & g_{T_{s_0}}(P_{Init}, P_{Objects}, P_{Types}), g_{Goal}(P_{Goal})\rangle \end{aligned}$$

If the following two conditions are satisfied, we call ***g*** *a compilation scheme from PDDL to RBAT's:*

[3] Of course, if one wants to capitalize on a positive result, the computational resources should be bounded. However, until now, a positive result has always implied a computationally cheap translation.

K3 *there exists a plan for P iff there exists a plan for $G(P)$.*
K4 *the size of the results of $g_{\Sigma}, g_{T_{SSA}}, g_{T_{PA}}, g_{T_{UNA}}, g_{T_{s_0}}$ and g_{Goal} is polynomial in the size of the arguments.*

One can see that if there exists a compilation scheme from RBAT's to PDDL, it follows that for every RBAT task there exists a solution-preserving PDDL planning task, that is only polynomially larger. Similarly, for the other direction. So if we can find two compilation schemes from RBAT's to PDDL and back such that the plan size is preserved exactly, one can indeed claim that these two formalisms have the same expressiveness.

Such a result would have practical relevance, because of the ability to transform any planning problem which occurs during the evaluation of a GOLOG program which is based on RBAT's into PDDL, search for a solution with an efficient planning system based on PDDL, and transform it back to a RBAT. Prerequisite for this proceeding is that the compilation schemes are polynomial-time computable. Furthermore, one would also need a one-to-one correspondence between PDDL and RBAT plans. However, this is the case for both compilation schemes which we present in the following.

One essential difference between PDDL and the situation calculus is that changes in the domain are described in different ways. While in the situation calculus successor state axioms are used for this request, in PDDL such changes are noted as effects of actions. So one has to find a mapping from the fluent-based notation in GOLOG to the action-centered one in PDDL.

4.2 A compilation scheme from RBAT's to PDDL

Because of the lack of space, we cannot present the whole scheme at this point. Instead we only deal with the critical parts of the scheme and only sketch the other parts briefly.

We create an PDDL planning task from a RBAT task $I = \langle D, T_{s_0}, G(s)\rangle$. For that purpose we have to specify the functions $f_{Types}()$, $f_{Predicates}(D)$, $f_{Actions}(D)$, $f_{Objects}(T_{s_0})$, $f_{Init}(T_{s_0})$ and $f_{Goal}(G(s))$.

Because the first five functions are relatively straightforward, we omit them here to save space and elaborate on $f_{Actions}(D)$, which is the most interesting function. Note that in the compilation we never use the typing feature of PDDL because there are no types in the RBAT's.

$N = \{n_i | n_i$ is a name of a primitive action in $T\}$ be the set of all names of primitive actions in T. For any primitive action A there is a precondition axiom of this form: $Poss(A(x_1, ..., x_n), s) \equiv \Pi_A(x_1, ..., x_n, s)$. It is

$$\begin{aligned} f_{Actions}(D) = (&: action\ n_i \\ &: parameters(h_i^{Parameters}) \\ &: precondition(\Pi_A(x_1, ..., x_n, s)) \\ &: effect(h_i^{Effect}))... \end{aligned}$$

for each action name n_i where

1. $h_i^{Parameters}$ states the parameters of the function which are all untyped and taken directly from the $Poss$-predicate.

2. h_i^{Effect} states the effects of the action. It is

$$h_i^{Effect} = (and\ E_1\ E_2\ ...)$$

where for every ϕ_l in which the expression $a = n_i$ occurs an E_i is contained, which is build as follows. We distinguish between the forms (2) and (3) of ϕ_l:

(a) $(\exists...)([\varphi_l \wedge](a = A(y_1, ..., y_m)))$
$F(x_1, ..., x_n, do(a, s))$ is true if $a = A(y_1, ..., y_m)$ and additionally the optional φ_l is true. First of all, we create the PDDL-formula $\Xi^{PDDL}(\varphi_l)$ which is simply the first-order formula φ_l expressed in PDDL syntax. This formula then is used in a conditional effect in PDDL-syntax:

$$(when\ \Xi^{PDDL}(\varphi_l)(F\ x_1 ...\ x_n))$$

Let $A(v_1, ..., v_n)$ be the PDDL action A with its parameter variables v_i, $F(x_1, ..., x_m)$ the fluent with its parameter variables x_i, and $a = A(y_1, ..., y_n)$ the parameterized action with its parameters y_i. We replace each x_i in $\Xi^{PDDL}(\varphi_l)$ and $(F\ x_1 ...\ x_n)$ by v_j if $x_i = y_j$.
Now we quantify universally over variables x_i for which there is no y_j with $y_j = x_i$. So we get an effect:

$$\begin{aligned}&(forall\ (X_i\ ...)\\ &\quad (when\ \Xi^{PDDL}(\varphi_l)\ (F\ x_1\ ...\ x_n)))\end{aligned}$$

(b) $F(x_1, ..., x_n, s) \wedge \neg$
$((\exists...)([\varphi_1 \wedge](a = A_1(y_{11}, ..., y_{1m_1})))$
$\vee(\exists...)([\varphi_2 \wedge](a = A_2(y_{21}, ..., y_{2m_2})))$
$\vee(\exists...)([\varphi_3 \wedge](a = A_3(y_{31}, ..., y_{3m_3})))$
$\vee..)$
here we proceed for each action A_i like in (a); in the effect we use additionally not:

$$\begin{aligned}&(forall\ (X_i\ ...)\\ &\quad (when\ \Xi^{PDDL}(\varphi_l)\ (not\ (F\ x_1\ ...\ x_n))))\end{aligned}$$

Example 2. Assume we have $At(x, y, do(a, s)) = (\exists p_1)(a = drive(x, p_1, y))$ and the PDDL action $drive(truck\ from\ to)$.

We start with $At(x, y)$ and then replace x by $truck$ and y by to. Because there is no x_i for which there is no y_j with $y_j = x_i$, there is no need to quantify. We get: $(At\ truck\ to)$

Example 3. Assume we have $At(x, y, do(a, s)) = At(x, y, s) \wedge \neg(\exists t, p_2)(In(x, t) \wedge a = drive(t, y, p_2))$ and the PDDL action $drive(truck\ from\ to)$.
We originate in $At(x, y)$ and then replace y by $from$ and t by $truck$. We get ***(when(In x truck)(not(At x from)))***. At last we quantify universally over x because no argument of $drive(t, y, p_2))$ is equal to x. We get:
$(forall(x)(when(In\ x\ truck)(not(At\ x\ from))))$

In the following, we will write s_{i+1} in order to denote $do(a_i, ..., do(a_1, do(a_o, s_0)))$.

Now we prove that **f** is indeed a compilation scheme. In order to show this, we need the following lemma.

Lemma 1. *In RBAT's, the truth value of each ground fluent $F(\boldsymbol{x}, s_i)$ is determined (so we have a complete theory in each situation).*

Proof Sketch: From restrictions E3 and E4 the claim follows for s_0. Using E2 and E1 on top of that, we can prove the claim inductively for all s_i. ■

In the following, we will use the expression $PosAtms(s_i)$ to denote all ground fluents $F(\boldsymbol{x}, s_i)$ for which it holds that $T \models F(\boldsymbol{x}, s_i)$. No other ground fluent is contained in $PosAtms(s_i)$. Because of Lemma 1, the next proposition is obvious.

Proposition 1. *For all ground fluents $F(\boldsymbol{x}, s_i)$ not contained in $PosAtms(\sigma)$ it holds that: $T \not\models F(\boldsymbol{x}, s_i)$.*

Theorem 1. ***f** is a compilation scheme from RBAT's to PDDL that preserves plan size exactly.*

Proof Sketch: In order to prove this theorem we have to prove K1 and K2 in Definition 1. Since K2 is straight-forward, we omit the proof for this condition here. The proof for K1 is done inductively over the length of the plan.

Let $I = \langle D, T_{s_0}, G(s) \rangle$ be an RBAT task. Then $\mathbf{f}(I) = \langle P', P_{Init}, P_{Goal} \rangle$ is the planning task in PDDL syntax created by **f**. Let s_0 be the initial situation of I and let z_0 be the initial state of $\mathbf{f}(I)$ (represented by the set of atoms true in this state).

$f_{Init}(T_{s_0})$ is simply another encoding of T_{s_0} and so does not change anything on the fluents truth values, so $z_0 = PossAtms(s_0)$ follows immediately. Assume now that $i - 1$ actions were applied, z_i be the state of $\mathbf{f}(I)$ after the application of this actions and it holds that $z_i = PossAtms(s_i)$.

On the one hand we now examine the state z_{i+1}, which is reached by applying a, and on the other hand we look at $PossAtms(s_{i+1}) = PossAtms(do(a, s_i))$ for the same action a.

Let F be a ground fluent (F is in z_i iff it is in $PossAtms(s_i)$).

1. $F \in PosAtms(s_i)$ and $F \notin PossAtms(s_{i+1})$
   Because of E2, this can only be the case if the SSA of F contains a ϕ_l which is of the form (2) and one of the actions in ϕ_l is action a. **f** contains a function h_i^{Effect}, which translates this scheme so that with each execution of a fluent F will be asserted false. So F is not in z_{i+1}.
2. $F \notin PosAtms(s_i)$ and $F \in PossAtms(s_{i+1})$
   Because of E2, this can only be the case if the SSA of F contains a ϕ_l which is of the form (3) and whose action is a. **f** contains a function h_i^{Effect}, which translates this scheme so that with each execution of a fluent F will be asserted true. So F is in z_{i+1}.
3. $F \notin PosAtms(s_i)$ and $F \notin PossAtms(s_{i+1})$
   Because of E2, this can only be the case if the SSA of F does not contain a ϕ_l which is of the form (3) and whose action is a. Then **f** does not create any effect regarding F for a. Because truth values of fluents may be changed in PDDL only by actions, F is not in z_{i+1}.

4. $F \in PosAtms(s_i)$ and $F \in PossAtms(s_{i+1})$
   Because of E2, this can only be the case if the SSA of F does not contain a ϕ_l which is of the form (2) and whose action is a. Then **f** does not create any effect regarding F for a. Because truth values of fluents may be changed in PDDL only by actions, F will stay in z_{i+1}.

So F is in z_{i+1} iff it is in $PossAtms(s_{i+1})$ and therefore it holds that $z_{i+1} = PossAtms(s_{i+1})$.

Hence, if there is a plan s_{n+1} in I, it holds that $I \models ((\forall a, s^*).do(a, s^*) \sqsubseteq s_n \supset Poss(a, s^*)) \wedge G(s_n)$. This means that $I \models Poss(a_i, s_i)$ for $1 \leq i \leq n-1$. As just shown $PosAtms(s_i) = z_i$ holds so $z_i \models Pre(a_i)$ holds too.

Also because of $I \models G(s_n)$ and $z_n = PosAtms(s_n)$, $z_n \models f_{Goal}(G(s_n))$ holds as well.

So $a_1, ..., a_n$ is a plan for the PDDL task **f**(I) too.

If there is no such plan, there cannot be a plan in **f**(I) with a similar argumentation either.

As is obvious from the construction, **f** also preserves plan size exactly.

From this the claim follows. ■

4.3 A compilation scheme from PDDL to RBAT's

In order to create a RBAT task from a PDDL planning task $P = \langle P', P_{Init}, P_{Goal} \rangle$, we have to specify the functions $g_{\Sigma}, g_{T_{SSA}}(P'), g_{T_{PA}}(P'), g_{T_{UNA}}(P')$, $g_{T_{s_0}}(P_{Init}, P_{Objects}, P_{Types})$ and $g_{Goal}(P_{Goal})$. Again, because of the lack of space, we only present $g_{T_{SSA}}(P')$ here.

All fluents occurring in P' are declared in a $(: predicates)$-clause. $g_{T_{SSA}}(P')$ creates for each of these fluents an successor state axiom. During this process each effect of each PDDL action is "built" into the SSA by creating a ϕ_l in which the action together with optional conditions is coded. One expression of form (2) for all actions setting the fluent to false and one expression of form (3) for each action setting the fluent to true is used.

So $g_{T_{SSA}}(P')$ creates an SSA for each PDDL predicate $(F\ v_1 - T_1\ ...\ v_n - T_n)$ (where the T_i's are PDDL types):

$$\begin{aligned} F(v_1, ..., v_n, do(a, s)) \equiv\ & (T_1(v_1) \wedge ... \wedge T_n(v_n)) \\ & \wedge(\xi_1 \vee \xi_2 \vee \xi_3 \vee ... \\ & \vee(F(v_1, ..., v_n, s) \wedge \neg(\zeta_1 \vee \zeta_2 \vee \zeta_3 \vee ...)) \end{aligned}$$

The ξ_l and ζ_l for the SSA's are generated by decomposing the effect E of each PDDL action $(A\ x_1\ x_2\ x_3\ ...)$ until the predicate constrained by the SSA is reached.

During the decomposition each occurring part effect E' has an assigned set $\Phi_{E'}$ with additional conditions. The decomposition takes place recursively and distinguishes between the following cases:

1. If E is of the form $(and\ A\ B\ C\ ...)$, decompose A, B and C separately. It is $\Phi_A = \Phi_B = \Phi_C = \Phi_{(and\ A\ B\ C\ ...)}$.

2. If E is of the form $(forall\ (x - (either\ T1\ T2...)...)\ E')$, decompose E'. It is $\Phi_{E'} = \Phi_E \cup ((T_1(x) \vee T_2(x)) \wedge ...)$.
3. If E is of the form $(when\ \Pi\ E')$, decompose E'. It is $\Phi_{E'} = \Phi_E \cup \Xi^{FOL}(\Pi)$.
4. If E is of the form $(P\ y_1\ y_2\ ...\ y_n)$, break off this branch of the decomposition and create a proper formula $(a = A(x_1, x_2, x_3...)) \wedge \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge ...$ for all $\phi_i \in \Phi_E$. Then we have to adjust the variables. $F(v_1, v_2, ..., do(a, s))$ be the fluent with its parameters in the SSA. We proceed for $1 \leq i \leq n$ as follows:
   (a) if $x_i = y_j$, replace x_i by v_j
   (b) if $y_i \neq x_j$ for all j, replace y_i by v_i in all ϕ_k. Finally, we quantify existentially over all remaining x_i and get ξ_l for P.

$$((\exists ...)(a = A(x_1, x_2, x_3...)) \wedge \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge ...)$$

5. If E is of the form $(not\ (P\ y_1\ y_2\ ...\ y_n))$, we proceed as in 4. and get ζ_l for P.

Example 4. Assume we have the following PDDL action *A=(:action Drive :parameters (car from to))*
with effect E=

```
:effect (and (At car to)
             (not (At car from))
           (forall
              (x - (either Men Women))
              (when (In x car)
                (and (At x to)
                     (not (At x from))
                )
              )
           )
        )
```

If we apply the decomposition above, we get the following effects for $At(v_1, v_2)$:

$$\xi_1 = \exists p_1(a = drive(v_1, p_1, v_2))$$

$$\zeta_1 = \exists p_2(a = drive(v_1, v_2, p_2))$$

$$\xi_2 = \exists car, p_2(a = drive(car, p_2, v_2) \wedge (Man(v_1) \vee Women(v_1)) \wedge In(v_1, car))$$

$$\zeta_2 = \exists car, p_2(a = drive(car, v_2, p_2) \wedge (Man(v_1) \vee Women(v_1)) \wedge In(v_1, car))$$

Under the assumption that $drive$ is the only action affecting $At(x, y)$ and that all parameters of $At(x, y)$ are untyped, the SSA of $At(x, y)$ looks as follows:

$$At(v_1, v_2, do(a, s)) \equiv \xi_1 \vee \xi_2 \vee (At(v_1, v_2, s) \wedge \neg(\zeta_1 \vee \zeta_2))$$

We state the next theorem without proof, which in fact is similar to the proof of Theorem 1.

Theorem 2. ***g*** *is a compilation scheme from PDDL planning tasks to RBAT's that preserves plan size exactly.*

5 Conclusion and Outlook

We have defined a subset of Basic Action Theories, the so-called *Restricted Basic Action Theories*, which have the same expressive power as the ADL fragment of PDDL (PDDL 2.1 Level 1). To show this equivalence, we have presented compilation schemes for both directions.

From a practical point of view, one can use this result to embed efficient planning engines in GOLOG for generating plans when the GOLOG non-deterministic choice operator appears and the language restrictions are met. A semantic base for that and first experiments are described in a companion paper [11].

On the theoretical side, we have for the first time identified a syntactic fragment of GOLOG that is expressively equivalent with the ADL fragment of PDDL and have with this bridged the gap between planning languages and action formalisms. Of course, such a result triggers a number of related questions. For example, how far can we extend the expressiveness of RBAT's without invalidating the result? Can we find more correspondences between GOLOG and PDDL fragments? Which features of GOLOG are provably not compilable to PDDL? We will address some of these questions in the future with the intention to extend the applicability of planning techniques in action formalisms even further.

References

1. Fox, M., Long, D.: An extension to pddl for expressing temporal planning domains. Journal of Artificial Intelligence Research (2003)
2. Gerevini, A., Long, D.: Plan constraints and preferences in PDDL 3. Technical Report, Univ. Brescia, Italy (2005)
3. Reiter, R.: Knowledge in Action. MIT Press (2001)
4. Nebel, B.: On the compilability and expressive power of propositional planning formalisms. Journal of Artificial Intelligence research (2000)
5. Green, C.: Application of theorem proving to problem solving. Proceedings of the 1st International Joint Conference on Artificial Intelligence, pages 219-240 (1996)
6. Fikes, R.E., Nilsson, N.J.: Strips: a new approach to the application of theorem proving to problem solving. Proceedings of the Australian Joint Conference on Artificial Intelligence, 2:189-208 (1971)
7. Pednault, E.P.D.: ADL: Exploring the middle ground between STRIPS and the situation calculus. Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (1989)
8. McDermott, D.: PDDL - The Planning Domain Definition Language, Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
9. Lifschitz, E.: On the Semantics of STRIPS. Proceedings of 1986 Workshop: Reasoning about actions and plans (1986)
10. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming, 31:59-84 (1997)
11. Claßen, J., Eyerich, P., Lakemeyer, G., Nebel, B.: Towards an Integration of GOLOG and Planning. Proceedings of the International Joint Conference on Artificial Intelligence (to appear)