Expressiveness of ADL and Golog: Functions Make a Difference

Gabriele Röger and **Bernhard Nebel**

Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Germany

{roeger,nebel}@informatik.uni-freiburg.de

Abstract

The main focus in the area of action languages, such as GOLOG, was put on expressive power, while the development in the area of action planning was focused on efficient plan generation. An integration of GOLOG and planning languages would provide great advantages. A user could constrain a system's behavior on a high level using GOLOG, while the actual low-level actions are planned by an efficient planning system. First endeavors have been made by Eyerich et al. by identifying a subset of the situation calculus (which is the basis of GOLOG) with the same expressiveness as the ADL fragment of PDDL. However, it was not proven that the identified restrictions define a maximum subset. The most severe restriction appears to be that functions are limited to constants. We will show that this restriction is indeed necessary in most cases.

Introduction

While action formalisms and planning techniques share the same origin, the emphasis in their development lies on different aspects. The main focus in the area of action formalisms was put on expressive power, whereas the development of planning techniques was focused on an efficient generation of action plans. The languages used in this field – such as the basic variant of STRIPS – had only a comparatively limited expressive power. This changed in 1998 with the introduction of PDDL (McDermott 1998), whose expressive power has been more and more extended but whose development has always been oriented on the capabilities of state-of-the-art planning systems. In recent years one could observe some convergence of the expressive power of PDDL and of the expressiveness of action formalisms like GOLOG (Levesque *et al.* 1997), at least if one focuses on linear sequences of actions.

This generates the idea of integrating the concepts from both areas, which would provide great advantages. A user could write a program in GOLOG and benefit from the flexibility in describing a system's (e.g. a robot's) behaviour, while, during execution, the upcoming planning tasks could be solved with efficient planning systems. Therefore a common semantic basis and a comparison of the expressiveness of the two languages GOLOG and PDDL is required.

As a first step Eyerich *et al.* (2006) presented restrictions on basic action theories (Reiter 2001), which are formulated in the situation calculus, which is in turn the basis of GOLOG. With these restrictions the formalism has the same expressive power as ADL, a subset of PDDL. Eyerich *et al.* show this by giving a compilation scheme (Nebel 2000) from these restricted basic action theories to ADL and vice versa.

Another step towards a common semantic basis is the work of Claßen *et al.* (2007): They presented a mapping of ADL to a variant of the situation calculus and related the update of an ADL world to progression in the situation calculus. Moreover, they presented first experimental results and demonstrated that a GOLOG interpreter which uses a state-of-the-art planning system can handle much bigger instances and takes less time (except for the smallest instances) than a system which uses the built-in GOLOG planner.

But it is still an open problem what a *maximum* subset of the situation calculus with the same expressive power as ADL is. The restrictions of Eyerich *et al.* provide a good starting point in search for such a subset, but it is still open whether it is possible to shift a wider part of GOLOG to a highly efficient ADL planning system. We will address this issue for the restriction that appears to be most severe: The limitation of functions to constants. We will prove that as soon as functions lead to additional actions or objects, the formalism would have more expressive power than ADL. Also we will present restrictions which maintain the same expressiveness as ADL.

The paper is structured as follows: First we will introduce the framework used to compare the expressiveness of planning formalisms. In the following two sections, we will briefly sketch the two considered formalisms, namely ADL and basic action theories. Subsequently we will first present our results concerning situation-independent functions before we turn to the functions that can change their value after each action. We will close with a brief conclusion and an outlook on our future work.

Compilation Schemes

In order to compare the expressive power of different planning formalisms, we use a technique introduced by Nebel (2000) – the so-called *compilation schemes*, which are *solution preserving mappings* with *polynomially sized results*

from one formalism $\mathcal{X}$ to another formalism $\mathcal{Y}$.

A planning instance $\Pi = \langle \Xi, \mathbf{I}, \mathbf{G} \rangle$ consists of the *domain structure* Ξ which contains primarily the description of the possible actions, the *initial state specification* $\mathbf{I}$, and the *goal specification* $\mathbf{G}$. A *plan* is a sequence of actions that leads from the start situation to a goal. A compilation scheme maps such a planning instance Π of formalisms $\mathcal{X}$ to an instance $F(\Pi)$ of formalism $\mathcal{Y}$.

Definition 1. *Let* $\mathbf{f}$ *be a tuple* $\langle f_\xi, f_i, f_g, t_i, t_g \rangle$ *of functions that induces a function F from $\mathcal{X}$-instances $\Pi = \langle \Xi, \mathbf{I}, \mathbf{G} \rangle$ to $\mathcal{Y}$-instances $F(\Pi)$:*

$$F(\Pi) = \langle f_\xi(\Xi), f_i(\Xi) \cup t_i(\mathbf{I}), f_g(\Xi) \cup t_g(\mathbf{G}) \rangle.$$

We call $\mathbf{f}$ *a compilation scheme from $\mathcal{X}$ to $\mathcal{Y}$ iff*

1. *there exists a plan for Π iff there exists a plan for $F(\Pi)$,*
2. *the state translation functions t_i and t_g are polynomial-time computable,*
3. *and the size of the results of f_ξ, f_i, and f_g is polynomial in the size of the arguments.*

Note that there are no restrictions on the computational resources being used to compile the domain structure Ξ.

To compare the expressive power of two formalisms we additionally have to measure the size of corresponding plans. If a compilation scheme $\mathbf{f}$ has the property that for each plan P of instance Π there is a plan P' solving $F(\Pi)$ such that $\|P'\| \leq c \times \|P\| + k$ for some positive integers c and k, we say that $\mathbf{f}$ is a *compilation scheme preserving plan size linearly*, and conclude that the target formalism is at least as expressive as the source formalism. If there is at most a polynomial blow-up required, i.e. $\|P'\| \leq p(\|P\|, \|\Pi\|)$ for some polynomial p, we call $\mathbf{f}$ a *compilation scheme preserving plan size polynomially*. In this case the source formalism has more expressive power than the target formalism. If we even need a super-polynomial blow-up, there is a huge difference in the expressiveness of the two formalisms.

Let us now turn to the first of the formalisms considered in this article.

The ADL fragment of PDDL

In 1998 McDermott published the Planning Domain Description Language PDDL, which, with some revisions (Fox & Long 2003; Gerevini & Long 2005), has since become a standard for the representation of planning domains. In this article we are interested in the ADL fragment of PDDL, i.e. the language we get if exactly the `:adl` requirement is set. Beyond the definition of standard STRIPS operators, preconditions may contain negation, disjunction and quantification. Effects may contain conditional effects. Furthermore, equality is a built-in predicate and variables and objects may be typed.

For the proofs in this article we do not need to go into the details of ADL. It is only necessary to know that the state space of an ADL planning task is always finite and, thus, the general planning problem, i.e. the problem whether there is a plan for a given problem, is decidable.

The second formalism we consider in this paper is formulated in the situation calculus.

Basic Action Theories

The situation calculus is a second-order language which is specially designed for representing dynamically changing worlds. All changes to the world are the result of *actions* and each action leads to a new *situation*, which hence can be identified with a sequence of actions. The empty sequence corresponds to the *initial situation* which is denoted by s_0. Besides *actions* and *situations* there is a third sort *object* which is used for everything else. In the following, we use variables a for actions and s for situations (each with subscripts and superscripts). There are two special predicates: $s \sqsubset s'$ means that situation s is a proper subsequence of s', and $Poss(a, s)$ means that it is possible to perform action a in situation s. All situations except s_0 are formed with a function $do(a, s)$ meaning that action a is performed in situation s. Functions and predicates whose values vary from one situation to the next are called *fluents* and take a situation term as their last argument, e.g. $switched\_on(lamp, s)$.

In the following we will often omit leading universal quantifiers in sentences. The convention will be that any free variables in such expressions are implicitly universally quantified.

For the definition of the notion of basic action theories we need some further concepts (Reiter 2001).

Unique names axioms for actions tell us whether two actions are equal: Distinct action names A and B define distinct actions.

$$A(\bar{x}) \neq B(\bar{y}).$$

Identical actions have identical arguments:

$$A(x_1, \ldots, x_n) = A(y_1, \ldots, y_n) \supset x_1 = y_1 \wedge \ldots \wedge x_n = y_n.$$

A formula is called *uniform* in situation s if it does not mention the predicates $\sqsubset$ or $Poss$ and the only permitted occurrence of a situation term is the occurrence of situation s in the situation argument position of a fluent.

Whether it is possible to perform an action is stated by so-called *action precondition axioms* (APA), which are of the form

$$Poss(A(x_1, \ldots, x_n), s) \equiv \Pi_A(x_1, \ldots, x_n, s),$$

where A is an action function symbol with arity n and $\Pi_A(x_1, \ldots, x_n, s)$ is a formula that is uniform in s and whose free variables are among $x_1, \ldots, x_n, s$.

The value of a relational fluent after performing an action a is given by a *successor state axiom* (SSA), which is a sentence of the form

$$F(x_1, \ldots, x_n, do(a, s)) \equiv \Phi_F(x_1, \ldots, x_n, a, s),$$

where $\Phi_F(x_1, \ldots, x_n, a, s)$ is a formula uniform in s whose free variables are among $a, s, x_1, \ldots, x_n$. Similarly, a successor state axiom for a functional fluent is of the form

$$f(x_1, \ldots, x_n, do(a, s)) = y \equiv \phi_f(x_1, \ldots, x_n, y, a, s)$$

with conditions analogous to those of the relational fluents.

The starting point of Eyerich *et al.*'s work is the definition of basic action theories (BATs) from Reiter (2001):

Definition 2. *A BAT T is a theory of the form*

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

where

- *Σ are some foundational axioms for situations,*
- *T_{SSA} is a set of successor state axioms for functional and relational fluents,*
- *T_{APA} is a set of action precondition axioms,*
- *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, the initial database.*

The state successor axiom for a functional fluent f must actually define a value for f in the next situation, and this value must be unique.

Such a BAT T together with a situation calculus formula $G(s)$ whose only free variable is s form a planning task. A situation s is a plan for G (relative to T) iff

$$T \models executable(s) \land G(s),$$

where $executable(s)$ means that the action sequence s can be executed with respect to $Poss$.

Starting from this definition Eyerich *et al.* added several restrictions to gain the same expressiveness as ADL.

Restricted Basic Action Theories

A restricted basic action theory (RBAT) is a BAT with the following restrictions:

R1 The usage of functions is restricted to those of sort $\epsilon \to object$, which are in fact constants, and to functions of sort $object^n \to action$, i.e. action functions take only objects as arguments.

R2 All successor state axioms are of a certain form. The SSA of a relational fluent F has to fit the schema

$$F(x_1, \ldots, x_n, do(a,s)) \equiv \bigvee_{l=1}^{p} \psi_l \qquad (1)$$

for a finite $p > 0$, where exactly one ψ_l is of the form

$$F(x_1, \ldots, x_n, s) \Big[\land$$
$$\neg \Big([\exists \ldots] (a = A_1 (y_{11}, \ldots, y_{1m_1}) [\land \phi_1]) \lor \ldots \lor$$
$$[\exists \ldots] (a = A_q (y_{q1}, \ldots, y_{qm_q}) [\land \phi_q]) \Big) \Big]. \quad (2)$$

All the other ψ_l are of the form

$$[\exists \ldots] (a = A (y_1, \ldots, y_m) [\land \phi_l]). \qquad (3)$$

The existential quantification ranges over all y_i for which there is no x_j with $x_j = y_i$, and thus over all variables that are parameters of the action but not parameters of the fluent. The parts between square brackets are optional. Each action may be contained in at most one $a = A_i (y_{i1}, \ldots, y_{im_i})$ in (2) and in at most one $a = A_i (y_{i1}, \ldots, y_{im_i})$ in the expressions of form (3).

R3 The initial database must consist exactly of the following sentences:

1. For each $n + 1$-ary relational fluent F there is either an expression

$$\neg F(x_1, \ldots, x_n, s_0) \qquad (4)$$

or an expression

$$F(x_1, \ldots, x_n, s_0) \equiv ((x_1 = d_{11}) \land \ldots \land (x_n = d_{1n}))$$
$$\lor \ldots \lor ((x_1 = d_{m1}) \land \ldots \land (x_n = d_{mn})). \quad (5)$$

2. There are analogous expressions for all situation independent predicates.

3. There is a *domain closure axiom* $(x = d_1) \lor \ldots \lor (x = d_n)$ for constants.

4. There are unique names axioms $c_i \neq c_j$ for each two constants c_i and c_j.

Eyerich *et al.* have shown that these restrictions lead to the same expressive power as ADL, but it is still open whether they all are really necessary. In the following two sections we will consider this question for restriction R1. We will start with an examination of situation-independent functions.

Situation-independent Functions

The restriction to constants is intimately connected with restriction R3.3, which excludes the existence of other, unknown objects. Usually it should be possible that a function denotes also new objects which cannot be accessed by the constant symbols. If we want to permit such functions of sort $object^n \to object$, we also have to abandon this domain closure axiom. The following result is not due to this, but also keeps its validity if we suitably generalize the domain closure axiom.

Theorem 1. *There is no compilation scheme from RBAT to ADL if functions of sort $object^n \to object$ are permitted for $n > 0$, even if they are restricted to $n = 1$.*

Proof. Let $(\Sigma, Q, \delta, q_0, Q_{acc})$ be a deterministic Turing machine (TM) with $\Sigma = \{*, \square\}$ that starts on an empty tape.

We will formulate a BAT in such a way that there is a plan iff the TM halts, and all information concerning a certain TM is encoded in the initial database and the goal. As the state-translation functions of a compilation scheme must be polynomial-time computable and the plan existence problem is decidable in ADL, we could decide the halting problem if there were a compilation scheme.

We will exploit the absence of the restriction to represent infinitely many tape cells. Therefore, we use a single constant $\mathbf{p_0}$ to denote the first tape cell and a function $succ : object \to object$ which yields the successor of a cell. For a reliable simulation of the TM it is required that $succ(x) = succ(y) \supset x = y$. Due to the requirements for the initial database we cannot simply add this sentence there. Thus, we use an additional action $init : \epsilon \to action$ which is necessary in each plan and makes the goal formula true for all models where the desired condition is not fulfilled and the tape is flawed. As a plan must work for all

models, the existence of a plan is finally determined only by the models where the structure interpreting $succ()$ contains an infinite, linear sequence starting at $\mathbf{p_0}$.

We require the following constants of sort *object*:

- $*$ and $\square$ encodes the alphabet of the Turing machine,
- $\mathbf{q} \in Q$ to encode the states of the Turing machine,
- ◄,▼, and ► to encode the movements of the Turing machine, and
- $\mathbf{p_0}$ to denote the first tape cell.

In the following we will denote the set of these constants by $\mathcal{C}$. We will use the following predicates:

- $transition(c, q, c', q', d)$ to encode transition function δ,
- $state(q, s)$ denotes the state of the TM in situation s,
- $scan(p, s)$ denotes the tape cell the R/W-head stands on in a situation s,
- $star(p, s)$ encodes whether cell p contains a $*$,
- $initialized(s)$ is used to enforce action $init$, and
- $flawed(s)$ means that the structure interpreting $succ()$ does not contain an infinite, linear sequence starting at $\mathbf{p_0}$.

In the initial database the values of these predicates represent the initial status of the TM:

Predicate $transition$ encodes the transition function δ of the Turing machine:

$$transition(c, q, c', q', d) \equiv \bigvee_{\delta(\mathbf{c},\mathbf{q})=(\mathbf{c'},\mathbf{q'},\mathbf{d})} (c = \mathbf{c} \wedge$$
$$q = \mathbf{q} \wedge c' = \mathbf{c'} \wedge q' = \mathbf{q'} \wedge d = \mathbf{d}) \quad (6)$$

The R/W head stands on the first tape cell, the TM is in state $\mathbf{q_0}$, and all cells contain a blank.

$$scan(p, s_0) \equiv (p = \mathbf{p_0}) \quad (7)$$
$$state(q, s_0) \equiv (q = \mathbf{q_0}) \quad (8)$$
$$\neg star(p, s_0) \quad (9)$$

At the beginning the task is not initialized and we assume that the tape is not flawed. The latter will be set timely to the correct value by the action $init()$.

$$\neg initialized(s_0) \quad (10)$$
$$\neg flawed(s_0) \quad (11)$$

There should be a plan iff the tape is not flawed and the TM halts or if the tape is flawed. This leads to the goal formula

$$\exists q \left(state(q, s) \wedge \bigvee_{\mathbf{q} \in Q_{acc}} (q = \mathbf{q}) \right) \vee flawed(s). \quad (12)$$

An action $step(q, c, p, q', c', d, p')$ simulates one step of the TM: Previously the TM is in state q and reads character c on cell p. Then, it changes to state q', writes a c' and moves

in direction d to cell p'. This results directly in the following action precondition axiom:

$$Poss(step(q, c, p, q', c', d, p'), s) \equiv flawed(s) \vee$$
$$\big(state(q, s) \wedge scan(p, s) \wedge transition(q, c, q', c', d) \wedge$$
$$((c = *) \wedge star(p, s) \vee (c = \square) \wedge \neg star(p, s)) \wedge$$
$$((d = ◄) \wedge p = succ(p') \vee (d = ▼) \wedge p = p' \vee$$
$$(d = ►) \wedge p' = succ(p)) \wedge initialized(s)) \quad (13)$$

Action $init$ can only be used once.

$$Poss(init(), s) \equiv \neg initialized(s) \quad (14)$$
$$initialized(do(a, s)) \equiv a = init() \vee initialized(s) \quad (15)$$

This action diagnoses whether the tape is flawed:

$$flawed(do(a, s)) \equiv flawed(s) \vee a = init() \wedge$$
$$\neg \big(\forall x, y(succ(x) = succ(y) \supset x = y) \wedge$$
$$\forall x (\bigwedge_{\mathbf{c} \in \mathcal{C}} \neg succ(x) = \mathbf{c})) \quad (16)$$

In the cases where the tape is not flawed, the following successor state axioms provide a reliable simulation:

$$state(q', do(a, s)) \equiv state(q', s) \wedge \neg \exists q, c, p, c', d, p'$$
$$(a = step(q', c, p, q, c', d, p') \wedge \neg q = q') \vee$$
$$\exists q, c, p, c', d, p' (a = step(q, c, p, q', c', d, p')) \quad (17)$$

$$scan(p', do(a, s)) \equiv scan(p', s) \wedge \neg \exists q, c, p, q', c', d$$
$$(a = step(q, c, p', q', c', d, p) \wedge \neg p = p') \vee$$
$$\exists q, c, p, q', c', d (a = step(q, c, p, q', c', d, p')) \quad (18)$$

$$star(p, do(a, s)) \equiv star(p, s) \wedge$$
$$\neg \exists q, q', d, p' (a = step(q, *, p, q', \square, d, p')) \vee$$
$$\exists q, q', d, p' (a = step(q, \square, p, q', *, d, p')) \quad (19)$$

As there are no axioms constraining $succ()$, there can be arbitrary structures interpreting $succ()$. If the structure is "flawed" (not containing an infinite, linear structure starting at $\mathbf{p_0}$), the action $init()$ will force $flawed(s)$ to become true, which will make formula (12) true. Since there are *always* non-flawed structures, formula (12) can only become true in all models if s is a halting computation on the non-flawed structures and, thus, the simulated TM halts. Conversely, it is obvious that a halting computation translates into a sequence of actions making formula (12) true.

We have presented a reduction where all information concerning a certain TM is encoded in the initial database and the goal. Thus, according to the argumentation above, there cannot be any compilation scheme at all – not even with an exponential blow-up of the plan size. $\square$

Reiter (2001) also allows functions of sort $(action \cup object)^n \rightarrow object$ and $(action \cup object)^n \rightarrow action$ in basic action theories. Let us start with the functions of sort $(action \cup object)^n \rightarrow object$. The case $object^n \rightarrow object$ has been addressed in Theorem 1. Thus, we only have to consider $action^n \rightarrow object$.

Theorem 2. *There is no compilation scheme from RBAT to ADL if functions of sort $action^n \rightarrow object$ are permitted, even if they are restricted to $n = 1$.*

Proof. Basically, this can be prooved the same way as Theorem 1. We only need some minor changes to represent the infinitely many tape cells.

Action $succ$ now takes one argument of sort $action$. An additional action $obj2act(x) : object \rightarrow action$ is used to do the necessary conversion of the arguments. This action may never appear in a plan and, hence, its APA is always false. The correlation between the movements of the R/W head and the position on the tape has been formulated in the action precondition axiom of $step$. Thus, one has to change this precondition axiom and the SSA of $flawed$ (because $succ$ takes now an $action$ as argument) accordingly.

With these modifications the basic action theory behaves as in the proof of Theorem 1 and, hence, the same argumentation holds. □

Theorem 1 and 2 lead to the following corollary.

Corollary 1. *There is no compilation scheme from RBAT to ADL if functions of sort $(action \cup object)^n \rightarrow object$ are permitted for $n > 0$.*

The only remaining situation-independent functions are those of sort $(action \cup object)^n \rightarrow action$. As functions of sort $object^n \rightarrow action$ are permitted (see restriction R1), it suffices to consider only those of sort $action^n \rightarrow action$.

Theorem 3. *There is no compilation scheme from RBAT to ADL if functions of sort $action^n \rightarrow action$ are permitted, even if they are restricted to $n = 1$.*

Proof. This is again a variation of the proof of Theorem 1. We only need to change the representation of the tape cells and associate them with actions. Due to the unique names axioms for actions we can omit everything concerning models where the tape is flawed.

Constant $\mathbf{p_0}$ is now of sort $action$ instead of sort $object$ and function $succ$ changes to $succ : action \rightarrow action$ but keeps its meaning. Also all variables denoting tape cells, e.g. $p, p', \ldots$, change their sort to $action$. Additional action precondition axioms for these new functions declare them to be unexecutable. With these modifications of the reduction we can use the same argumentation as before. □

We have seen that situation-independent functions which can denote other objects than the constants add additional expressive power to RBATs and, thus, lead to a different expressiveness than ADL. If we had not abandoned the domain closure axiom, the functions had only been synonyms for the known constants. As this case is only a special case of the considerations in the next section, we do not discuss this further.

Functional Fluents

As opposed to the previous section we require for functional fluents, i.e. functions of sort $(action \cup object)^n \times situation \rightarrow (object \cup action)$ that they always have known constants or situation-independent actions as values. We

have several reasons for this: First of all, if these functions could introduce new objects (this implies that we use the relaxed domain closure axiom from the previous section) and we pass on SSAs for these functions, we would face the same difficulties as before. If they, in principle, can introduce new objects, but there must be a SSA for each fluent, this would lead to the following situation: Assume a SSA which in a certain situation assigns a previously unknown object to a functional fluent. As in each situation the value of a functional fluent must be unique and the unknown objects cannot be distinguished by a uniform expression, there can be at most one additional object. In fact, this is not a matter of functions, but rather a matter of the initial database. If we would drop restriction R3.3, we could easily compile these tasks by means of one additional constant. Thus, it makes sense to restrict the values of the functional fluents to constants and situation-independent actions. Furthermore, in the book of Reiter we have found only examples that corroborate this assumption.

Why do we accept such a restriction for functional fluents, but not for situation-independent functions? Situation-independent functions that are restricted to such an extent are in practice only synonyms for constants or other known actions, and, thus, add only marginal benefit. By contrast, functional fluents, which can change their value from situation to situation, are more like pointers in programming languages and can provide the possibility of a more brief and elegant formulation of a planning task.

Before we consider functional fluents, we have to determine what a reasonable extrapolation of the restrictions of Eyerich *et al.* to functional fluents looks like. If we omit such restrictions, the basic action theories are undecidable with similar proofs as above.

Obviously the following is a reasonable extension of restriction R2: There is a successor state axiom for each functional fluent. The SSA of a functional fluent f has to fit the schema

$$f(x_1, \ldots, x_n, do(a, s)) = y \equiv \bigvee_{l=1}^{p} \psi_l \qquad (20)$$

for a finite $p > 0$, where exactly one ψ_l is of the form

$$f(x_1, \ldots, x_n, s) = y \Big[\wedge$$
$$\neg \Big([\exists \ldots] (a = A_1 (y_{11}, \ldots, y_{1m_1}) [\wedge \phi_1]) \vee \ldots \vee$$
$$[\exists \ldots] (a = A_q (y_{q1}, \ldots, y_{qm_q}) [\wedge \phi_q]) \Big) \Big]. \quad (21)$$

All the other ψ_l are of the form

$$[\exists \ldots] (a = A (y_1, \ldots, y_m) [\wedge \phi_l]). \qquad (22)$$

The conditions that hold for the SSAs of relational fluents must also hold for those of functional fluents.

Restriction R3 of Eyerich *et al.* leads to a unique model in the initial database, in which the truth-values of the predicates are explicitly specified. Analogously we require that the value of each functional fluent in s_0 is explicitly specified:

For each $(n + 1)$-ary functional fluent f there is an expression

$$f(x_1, \ldots, x_n, s_0) = y \equiv$$
$$((x_1 = d_{11}) \wedge \ldots \wedge (x_n = d_{1n}) \wedge (y = d_1)) \vee \ldots \vee$$
$$((x_1 = d_{m1}) \wedge \ldots \wedge (x_n = d_{mn}) \wedge (y = d_m)). \quad (23)$$

We begin our considerations with the functions of sort $(action \cup object)^n \times situation \rightarrow object$.

Theorem 4. *There is a compilation scheme from RBATs which are extended by fluents of sort*

$$(action \cup object)^n \times situation \rightarrow object \quad (24)$$

to ADL preserving plan size exactly.

Proof. We will sketch a compilation scheme from the extended RBATs to ordinary RBATs. If one combines this compilation scheme with the one presented by Eyerich *et al.*, one receives a compilation scheme from extended RBATs to ADL.

For each function $f : (action \cup object)^n \times situation \rightarrow object$ we introduce a predicate

$$P_f : (action \cup object)^n \times object \times situation. \quad (25)$$

We substitute the sentence $f(x_1, \ldots, x_n, s_0) = y \equiv \phi_f$ in the initial database by an expression

$$P(x_1, \ldots, x_n, y, s_0) \equiv \phi_f. \quad (26)$$

The SSA $f(x_1, \ldots, x_n, do(a, s)) = y \equiv \psi_f$ analogously becomes $P_f(x_1, \ldots, x_n, y, do(a, s)) \equiv \psi_f$.

To eliminate all remaining occurrences of function f in a formula, we proceed as follows:

- If f occurs as $f(x_1, \ldots, x_n, s) = y$, where y is a constant or a variable, we substitute it by $P_f(x_1, \ldots, x_n, y, s)$.
- If f occurs as $f(x_1, \ldots, x_n, s) = g(y_1, \ldots, y_m, s)$, we introduce a new variable y and substitute it by $\exists y (P_f(x_1, \ldots, x_n, y, s) \wedge P_g(y_1, \ldots, y_m, y, s))$.
- While f still occurs as an argument in a predicate, we can substitute it by the introduction of a new variable. For example, $P(f(y_1, \ldots, y_n, s), x_2, \ldots, x_m, s)$ becomes $\exists y (P_f(y_1, \ldots, y_n, y, s) \wedge P(y, x_2, \ldots, x_m, s))$.

The result obviously satisfies the requirements of a compilation scheme. $\square$

Before we consider the functional fluents whose value is of sort $action$, we have to clarify some aspects.

The first aspect is the unique names axioms for fluent actions. According to Reiter, identical actions shall have identical arguments and actions with distinct action names are not equal. Unfortunately, Reiter does not go into whether there are UNAs for fluents, too. In our opinion it does not make sense to postulate that, because it shall be possible that two action fluents represent the same situation-independent action.

The second aspect is the action precondition axioms. The APAs for the situation-independent action functions declare for each situation whether a certain action is executable. As each ground fluent action represents in each situation only a ground situation-independent action function, it is in each situation perfectly determined whether the action is executable. Thus, we can omit additional APAs for fluent actions.

Having these considerations in mind, it is obvious that the compilation scheme of Theorem 4 works likewise for functions whose value is of sort $action$.

Theorem 5. *There is a compilation scheme from RBATs which are extended by fluents of sort*

$$(action \cup object)^n \times situation \rightarrow action \quad (27)$$

to ADL preserving plan size exactly.

A general functional fluent of sort $(action \cup object)^n \times situation \rightarrow (object \cup action)$ can return both $action$s and $object$s. As we suppose that this was not intended by Reiter rather being a side effect of a condensed notation, we abstain from analysing this case.

Conclusion and Outlook

We have proven that functions whose values are not restricted to the constants extend the expressive power of RBATs and, thus, have a higher expressiveness than ADL. Further, we have shown that such a restriction preserves the same expressive power as ADL by presenting a compilation scheme that is efficiently computable. Therefore we have extrapolated the restrictions of Eyerich *et al.* to functional fluents. If we omit this extrapolation, the resulting problems arise not due to the usage of functions, but would also arise if we eased the restrictions R2 and R3. What happens in these cases, e.g. if the truth-values of predicates in the initial database are not specified explicitly but by a fixed point iteration, or if they are unspecified for some predicates, will be the topic of our future work. Having clarified this issue, we will consider PDDL fragments beyond ADL, which add features such as functions, constraints, durative actions, and preferences. The overall goal is to use the entire power of efficient, state-of-the-art planning techniques within a GOLOG interpreter.

References

Claßen, J.; Eyerich, P.; Lakemeyer, G.; and Nebel, B. 2007. Towards an Integration of Golog and Planning. In *Proc. IJCAI-07*.

Eyerich, P.; Nebel, B.; Lakemeyer, G.; and Claßen, J. 2006. GOLOG and PDDL: What is the Relative Expressiveness? In *Proc. PCAR-06*.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR* 20:61–124.

Gerevini, A., and Long, D. 2005. Plan Constraints and Preferences in PDDL3. Technical report, Univ. of Brescia, Italy.

Levesque, H. J.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A Logic Programming Language for Dynamic Domains. *J. Log. Prog.* 31(1-3):59–83.

McDermott, D. 1998. PDDL—the planning domain definition language. Tech. report, Yale Center f. Comp. Vision and Control.

Nebel, B. 2000. On the Compilability and Expressive Power of Propositional Planning Formalisms. *JAIR* 12:271–315.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press.