ALPprolog — A New Logic Programming Method for Dynamic Domains

CONRAD DRESCHER
Computing Laboratory, University of Oxford, UK
(*e-mail:* `Conrad.Drescher@comlab.ox.ac.uk`)

MICHAEL THIELSCHER
School of Computer Science and Engineering, The University of New South Wales, Australia
(*e-mail:* `mit@cse.unsw.edu.au`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Logic programming is a powerful paradigm for programming autonomous agents in dynamic domains, as witnessed by languages such as Golog and Flux. In this work we present ALPprolog, an expressive, yet efficient, logic programming language for the online control of agents that have to reason about incomplete information and sensing actions.

KEYWORDS: reasoning about actions, agent logic programs

1 Introduction

Programming autonomous agents that behave intelligently is one of the key challenges of Artificial Intelligence. Because of its declarative nature, and high level of abstraction, logic programming is a natural choice for this task. This is witnessed by e.g. the two major exponents of agent programming languages that are based on classical logic programming, namely Golog (Levesque et al. 1997) and Flux (Thielscher 2005a).

Both these languages combine a language for specifying the agent's behaviour with an axiomatic theory that describes the agent's environment. In the case of Golog the strategy language is procedural in nature (though implemented in Prolog), and the action theory is the classical Situation Calculus (McCarthy and Hayes 1969) in Reiter's version (Reiter 2001a). For Flux the strategy language is full classical logic programming, and the action theory is the more recent Fluent Calculus (Thielscher 1999).

In a recent work (Drescher et al. 2009) we have developed Agent Logic Programs (ALPs), a new declarative strategy language that is based upon a proof calculus in the style of classical SLD-resolution. Contrary to Golog and Flux the ALP framework is parametric in the action theory: any background theory that allows to infer when an action is applicable, and what the effects of the action are, can be used. Exploiting this generality we have recently (Thielscher 2010b) been able to give a semantics for the BDI-style language AgentSpeak (Bordini et al. 2007). Another distinctive feature of the theoretical framework is the elegant handling of incomplete information for offline planning via *disjunctive substitutions*. By default, ALPs are combined with our new Unifying Action Calculus (UAC) (Thielscher 2011) that encompasses the major logical action calculi, including both the Situation Calculus and the Fluent Calculus, as well as many planning domain description languages. The ALP formalism stays entirely within classical logic.

The implementation of any fragment of the ALPprolog framework consists of (1) an implementation of the proof calculus, and (2) an action theory reasoner. Existing mature Prolog technology can be used out of the box for (1) unless disjunctive substitutions enter the picture. For (2) we can also exploit existing technology: E.g. Golog implements a fragment of the Situation Calculus, and Flux handles a fragment of the Fluent Calculus. In (Drescher et al. 2009) the implementation of a Description Logic-based fragment of the Fluent Calculus is described.

In this work we present ALPprolog, where the underlying action theory is an essentially propositional version of the Fluent Calculus in the UAC that includes a simple, yet powerful model of sensing. ALPprolog is intended for the online control of agents, where actions are immediately executed. This starkly contrasts with offline reasoning, where agents may make assumptions to see where these are leading. ALPprolog was developed specifically for the efficient handling of large ground state representations, something that we consider to be practically useful. To this end ALPprolog combines strong-points of Golog and Flux:

- From Golog it takes the representation of the agent's state knowledge in full propositional logic via prime implicates; and
- From Flux it takes the principle of progression: The agent's state knowledge is updated upon the execution of an action. In standard Golog the agent's initial state knowledge is never updated. [1] Instead, queries referring to later time-points are rewritten until they can be evaluated against the initial state knowledge, something which becomes a hindrance to good performance as the sequence of executed actions grows.

We emphasise that ALPprolog is an agent programming language in the spirit of classical logic programming in Prolog: The straightforward operational semantics provides the programmer with a powerful means to actively determine the sequence of actions that an agent executes. ALPprolog [2] can be obtained at `alpprolog.sourceforge.net`.

The remainder of this paper is organised as follows: In Section 2 we recall the basics of the ALP framework, and in Section 3 we introduce ALPprolog. We evaluate the performance of ALPprolog in Section 4, and conclude in Section 5.

2 ALPs in a Nutshell

The purpose of agent logic programs is to provide high-level control programs for agents using a combination of declarative programming with reasoning about actions. The syntax of these programs is kept very simple: standard (definite) logic programs (see e.g. (J.W. Lloyd 1987)) are augmented with just two special predicates, one — written $\mathtt{do}(\alpha)$ —

[1] But there is a version of Golog where the initial state is periodically updated (Sardina and Vassos 2005).
[2] The name is a play on ALPs, propositional logic, and the implementation in plain Prolog.

to denote the execution of an action by the agent, and one — written `?(`φ`)` — to verify properties against (the agent's model of) the state of its environment. This model, and how it is affected by actions, is defined in a separate action theory. This allows for a clear separation between the agent's strategic behaviour (given by the agent logic program itself) and the underlying theory about the agent's actions and their effects. Prior to giving the formal definition, let us illustrate the idea by an example agent logic program.

Example 1

Consider an agent whose task is to find gold in a maze. For the sake of simplicity, the states of the environment shall be described by a single fluent *(i.e., state property):* $\mathrm{At}(u,x)$ *to denote that* $u \in \{\mathrm{Agent}, \mathrm{Gold}\}$ *is at location* x. *The agent can perform the action* $\mathrm{Go}(y)$ *of going to location* y, *which is possible if* y *is adjacent to the current location of the agent. The following ALP describes a simple search strategy via a given list of locations (choice points) that the agent may visit, and an ordered collection of backtracking points. We follow the Prolog convention of writing variables with a leading uppercase letter.*

```
explore(Choicepoints,Backtrack) :-         % finished, if
   ?(at(agent,X)), ?(at(gold,X)).          % gold is found

explore(Choicepoints,Backtrack) :-
   ?(at(agent,X)),
   select(Y,Choicepoints,NewChoicepoints), % choose a direction
   do(go(Y)),                              % go in this direction
   explore(NewChoicepoints,[X|Backtrack]). % store the choice

explore(Choicepoints,[X|Backtrack]) :-     % go back one step
   do(go(X)),
   explore(Choicepoints,Backtrack).

select(X,[X|Xs],Xs).
select(X,[Y|Xs],[Y|Ys]) :- select(X,Xs,Ys).
```

Suppose we are given a list of choice points `C`*, then the query* `:- explore(C,[])` *lets the agent systematically search for gold from its current location: the first clause describes the base case where the agent is successful; the second clause lets the agent select a new location from the list of choice points and go to this location (the declarative semantics and proof theory for* `do(`α`)` *will require that the action is possible at the time of execution); and the third clause sends the agent back using the latest backtracking point.* ■

The example illustrates two distinct features of ALPs: (1) The agent strategy is defined by a logic program that may use arbitrary function and predicate symbols in addition to the signature of the underlying action theory. (2) The update of the agent's belief according to the effects of its actions is not part of the strategy. Formally, ALPs are defined as follows.

Definition 1

Consider an action theory signature Σ that includes the pre-defined sorts ACTION and FLUENT, along with a logic program signature Π.

- *Terms* are from $\Sigma \cup \Pi$.
- If `p` is an n-ary relation symbol from Π and $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are terms, then $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ is a *program atom*.

- $\mathtt{do(}\alpha\mathtt{)}$ is a *program atom* if α is an ACTION term in Σ.
- $\mathtt{?(}\varphi\mathtt{)}$ is a *program atom* if φ is a *state property* in Σ, that is, a formula (represented as a term) based on the FLUENTs in Σ.
- Clauses, programs, and queries are then defined as usual for definite logic programs, with the restriction that the two special atoms cannot occur in the head of a clause. ■

2.1 Declarative Semantics: Program + Action Theory

The semantics of an ALP is given in two steps. First, the program needs to be "temporalised," making explicit the state change that is implicit in the use of the two special predicates, $\mathtt{do(}\alpha\mathtt{)}$ and $\mathtt{?(}\varphi\mathtt{)}$. Second, the resulting program is combined with an action theory as the basis for evaluating these two special predicates. The semantics is then the classical logical semantics of the expanded program together with the action theory.

Time is incorporated into a program through macro-expansion: two arguments of sort TIME [3] are added to every regular program atom $p(\bar{x})$, and then $p(\bar{x}, s_1, s_2)$ is understood as restricting the truth of the atom to the temporal interval between (and including) s_1 and s_2. The two special atoms receive special treatment: $\mathtt{?(}\varphi\mathtt{)}$ is re-written to $\mathit{Holds}(\varphi, s)$, with the intended meaning that φ is true at s; and $\mathtt{do(}\alpha\mathtt{)}$ is mapped onto $\mathit{Poss}(\alpha, s_1, s_2)$, meaning that action α can be executed at s_1 and that its execution ends in s_2. The formal definition is as follows.

Definition 2

For a clause $\mathtt{H :- B}_1\mathtt{, ..., B}_n$ $(n \geq 0)$, let $s_1, \ldots, s_{n+1}$ be variables of sort TIME.

- For $i = 1, \ldots, n$, if $\mathtt{B}_i$ is of the form
  - $\mathtt{p(t}_1\mathtt{, ..., t}_m\mathtt{)}$, expand to $P(t_1, \ldots, t_m, s_i, s_{i+1})$.
  - $\mathtt{do(}\alpha\mathtt{)}$, expand to $\mathit{Poss}(\alpha, s_i, s_{i+1})$.
  - $\mathtt{?(}\varphi\mathtt{)}$, expand to $\mathit{Holds}(\varphi, s_i) \wedge s_{i+1} = s_i$.
- The head atom $\mathtt{H=p(t}_1\mathtt{, ..., t}_m\mathtt{)}$ is expanded to $P(t_1, \ldots, t_m, s_1, s_{n+1})$.
- The resulting clauses are understood as universally quantified implications.

Queries are expanded exactly like clause bodies, except that

- a special constant S_0 — denoting the earliest time-point in the underlying action theory — takes the place of s_1 ;
- the resulting conjunction is existentially quantified. ■

Example 1 (cont.)

The example program of the preceding section is understood as the following axioms, which for notational convenience we have simplified in that all equations between TIME

[3] Which specific concept of time is being used depends on how the sort TIME is defined in the underlying action theory, which may be branching (as, e.g., in the Situation Calculus) or linear (as, e.g., in the Event Calculus).

variables have been applied and then omitted.

$$
\begin{array}{rcl}
(\forall)\text{Explore}(c, b, s_1, s_1) & \subset & \text{Holds}(\text{At}(\text{Agent}, x), s) \wedge \text{Holds}(\text{At}(\text{Gold}, x), s_1) \\
(\forall)\text{Explore}(c, b, s_1, s_4) & \subset & \text{Holds}(\text{At}(\text{Agent}, x), s_1) \wedge \text{Select}(y, c, c', s_1, s_2) \wedge \\
 & & \text{Poss}(\text{Go}(y), s_2, s_3) \wedge \text{Explore}(c', [x|b], s_3, s_4) \\
(\forall)\text{Explore}(c, [x|b], s_1, s_3) & \subset & \text{Poss}(\text{Go}(x), s_1, s_2) \wedge \text{Explore}(c, b, s_2, s_3)
\end{array}
$$

$$(\forall)\text{Select}(x, [x|x'], x', s_1, s_1) \subset \text{true}$$

$$(\forall)\text{Select}(x, [y|x'], [y|y'], s_1, s_2) \subset \text{Select}(x, x', y', s_1, s_2)$$

The resulting theory constitutes a purely logical axiomatisation of the agent's strategy, which provides the basis for logical entailment. For instance, macro-expanding the query `:- explore(C,[])` *from the above example results in the temporalised logical formula* $(\exists s)\,\text{Explore}(C, [\,], S_0, s)$ *. If this formula follows from the axioms above, then that means that the strategy can be successfully executed, starting at* S_0 *, for the given list of choice points* C *. Whether this is actually the case of course depends on the additional action theory that is needed to evaluate the special atoms* Holds *and* Poss *in a macro-expanded program.* ■

Macro-expansion provides the first part of the declarative semantics of an agent logic program; the second part is given by an action theory in form of a logical axiomatisation of actions and their effects. The overall declarative semantics of agent logic programs is given by the axiomatisation consisting of the action theory and the expanded program.

Let us next introduce the fragment of the UAC corresponding to the Fluent Calculus. The UAC that is used to axiomatise the action theory is based on many-sorted first order logic with equality and the four sorts TIME, FLUENT, OBJECT, and ACTION. By convention variable symbols s, f, x, and a are used for terms of sort TIME, FLUENT, OBJECT, and ACTION, respectively. Fluents are reified, and the standard predicate *Holds* : FLUENT $\times$ TIME indicates whether a fluent is true at a particular time. The predicate *Poss*(a, s_1, s_2) means that action α can be executed at s_1 and that its execution ends in s_2. The number of function symbols into sorts FLUENT and ACTION is finite.

Definition 3 (*Action Theory Formula Types*)
We stipulate that the following formula types are used by action theories:

- State formulas express what is true at particular times: A *state formula* $\Phi[\bar{s}]$ *in* $\bar{s}$ is a first-order formula with free variables $\bar{s}$ where
  - — for each occurrence of *Holds*(f, s) we have $s \in \bar{s}$;
  - — predicate *Poss* does not occur.

  A state formula is *pure* if it does not mention predicates other than *Holds*.
- A *state property* ϕ is an expression built from the standard logical connectives and terms $F(\bar{x})$ of sort FLUENT. With a slight abuse of notation, by *Holds*(ϕ, s) we denote the state formula obtained from state property ϕ by replacing every occurrence of a fluent f by *Holds*(f, s). In an expanded program Π we always treat *Holds*(ϕ, s) as atomic. State properties are used by agent logic programs in `?(Phi)` atoms.
- The *initial state axiom* is a state formula $\phi(S_0)$ in S_0, where S_0 denotes the initial situation.

- An *action precondition axiom* is of the form

$$(\forall)\mathit{Poss}(A(\bar{x}), s_1, s_2) \equiv \pi_A[s_1] \wedge s_2 = \mathit{Do}(A(\bar{x}), s_1),$$

where $\pi_A[s_1]$ is a state formula in s_1 with free variables among $s_1, \bar{x}$. This axiom illustrates how different actions lead to different situation terms $\mathit{Do}(A(\bar{x}), s_1)$. Situations constitute the sort TIME in the Fluent Calculus and provide a branching time structure.

- *Effect axioms* are of the form

$$\begin{aligned}&\mathit{Poss}(A(\vec{x}), s_1, s_2) \supset \\ &\quad \bigvee_k (\exists \vec{y}_k)(\Phi_k[s_1] \wedge (\forall f)[(\bigvee_i f = f_{ki} \vee (\mathit{Holds}(f, s_1) \wedge \bigwedge_j f \neq g_{kj})) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \equiv \mathit{Holds}(f, s_2)]).\end{aligned}$$

Such an effect axiom has k different cases that can apply — these are identified by the case selection formulas $\Phi_k[s_1]$ which are state formulas in s_1 with free variables among $s_1, \vec{x}, \vec{y}_i$. The f_{ki} (and g_{kj}, respectively) are fluent terms with variables among $\vec{x}, \vec{y}_k$ and describe the positive (or, respectively, negative) effects of the action, given that case k applies.
- *Domain constraints* are universally quantified state formulas $(\forall s)\delta[s]$ in s.
- *Auxiliary axioms* are domain-dependent, but time-independent, additional axioms such as e.g. an axiomatisation of finite domain constraints.

An action theory $\mathcal{D}$ is given by an initial state axiom $\mathcal{D}_{\text{Init}}$, finite sets $\mathcal{D}_{\text{Poss}}$ and $\mathcal{D}_{\text{Effects}}$ of precondition and effect axioms. Moreover domain constraints $\mathcal{D}_{\text{dc}}$ and auxiliary axioms $\mathcal{D}_{\text{aux}}$ may be included. For illustration, the following is a background axiomatisation for our example scenario as a basic Fluent Calculus theory in the UAC.

Example 1 (cont.)
Our example program can be supported by the following domain theory.

- *Initial state axiom*

$$\text{Holds}(\text{At}(\text{Agent}, 1), S_0) \wedge \text{Holds}(\text{At}(\text{Gold}, 4), S_0)$$

- *Precondition axiom*

$$\begin{aligned}\text{Poss}(\text{Go}(y), s_1, s_2) \equiv \quad &(\exists x)(\text{Holds}(\text{At}(\text{Agent}, x), s_1) \wedge (y = x+1 \vee y = x-1)) \\ &\wedge s_2 = \text{Do}(\text{Go}(y), s_1)\end{aligned}$$

- *Effect axiom*

$$\begin{aligned}&\text{Poss}(\text{Go}(y), s_1, s_2) \supset \\ &\quad (\exists x)(\text{Holds}(\text{At}(\text{Agent}, x), s_1) \wedge \\ &\quad [(\forall f)\text{Holds}(f, s_2) \equiv (\text{Holds}(f, s_1) \vee f = \text{At}(\text{Agent}, y)) \wedge f \neq \text{At}(\text{Agent}, x)]).\end{aligned}$$

Given this (admittedly very simple, for the sake of illustration) specification of the background action theory, the axiomatisation of the agent's strategy from above entails, for example, $(\exists s)\,\text{Explore}([2,3,4,5], [\,], S_0, s)$. *This can be shown as follows. First, observe that the background theory entails*

$$\text{Holds}(\text{At}(\text{Agent}, 4), S) \wedge \text{Holds}(\text{At}(\text{Gold}, 4), S),$$

where S *denotes the situation term* $\mathrm{Do}(\mathrm{Go}(4), \mathrm{Do}(\mathrm{Go}(3), \mathrm{Do}(\mathrm{Go}(2), S_0)))$ *. It follows that* $\mathrm{Explore}([5], [3, 2, 1], S, S)$ *according to the first clause of our example ALP. Consider, now, the situation* $S' = \mathrm{Do}(\mathrm{Go}(3), \mathrm{Do}(\mathrm{Go}(2), S_0))$ *, then action theory and strategy together imply*

$$\mathrm{Holds}(\mathrm{At}(\mathrm{Agent}, 3), S') \wedge \mathrm{Select}(4, [4, 5], [5], S', S') \wedge \mathrm{Poss}(\mathrm{Go}(4), S', S)$$

By using this in turn, along with $\mathrm{Explore}([5], [3, 2, 1], S, S)$ *from above, according to the second program clause we obtain* $\mathrm{Explore}([4, 5], [2, 1], S', S)$ *. Continuing this line of reasoning, it can be shown that*

$$\begin{array}{rl} & \mathrm{Explore}([3, 4, 5], [1], \mathrm{Do}(\mathrm{Go}(2), S_0), S) \\ \textit{and hence,} & \mathrm{Explore}([2, 3, 4, 5], [\,], S_0, S) \end{array}$$

This proves the claim that $(\exists s)\,\mathrm{Explore}([2, 3, 4, 5], [\,], S_0, s)$ *. On the other hand e.g. the query* $(\exists s)\,\mathrm{Explore}([2, 4], [\,], S_0, s)$ *is* not *entailed under the given background theory: Without location* 3 *among the choice points, the strategy does not allow the agent to reach the only location that is known to house gold.* ■

2.2 Operational Semantics: Proof Calculi

We have developed two sound and complete proof calculi for ALPs that both assume the existence of a suitable reasoner for the underlying action theory (Drescher et al. 2009).

The first proof calculus is plain SLD-resolution, only that *Holds* - and *Poss* -atoms are evaluated against the action theory. This calculus is sound and complete if the underlying action theory has the witness property: That is, whenever $\mathcal{D} \models (\exists x)\phi(x)$ then there is a substitution θ such that $\mathcal{D} \models (\forall)\phi(x)\theta$. Note that in general action theories may violate the witness property, as they may include disjunctive or purely existential information; consider e.g. the case $\mathit{Holds}(\mathit{At}(\mathit{Gold}, 4), S_0) \vee \mathit{Holds}(\mathit{At}(\mathit{Gold}, 5), S_0)$, where the exact location of the gold is unknown.

Hence the second proof calculus, intended for the general case, resorts to constraint logic programming, and the notion of a disjunctive substitution: Still assuming that the gold is located at one of two locations the query $\mathcal{D} \models (\exists x)\mathit{Holds}(\mathit{At}(\mathit{Gold}, x)$ can now be answered positively via the disjunctive substitution $x \rightarrow 4 \vee x \rightarrow 5$. Disjunctive substitution together with the respective principle of reasoning by cases are a powerful means for inferring conditional plans.

For the online control of agents, however, assuming a particular case is unsafe. But if we use the plain SLD-resolution-based ALP proof calculus on top of action theories that lack the witness property we obtain a nice characterisation of cautious behaviour in a world of unknowns (albeit at the cost of sacrificing logical completeness). For ALPprolog this is the setting that we use.

In both proof calculi we adopt the ”leftmost” computation rule familiar from Prolog. This has many advantages: First, it simplifies the implementation, as this can be based on existing mature Prolog technology. Second, state properties can always be evaluated against a description of the ”current” state. Last, but not least, this ensures that actions are executed in the order intended by the programmer — this is of no small importance for the online control of agents.

3 ALPprolog

We next present ALPprolog — an implementation of the ALP framework atop of action theories in a version of the Fluent Calculus that

- uses (a notational variant of) propositional logic for describing state properties;
- is restricted to actions with ground deterministic effects; and
- includes sensing actions.

The intended application domain for ALPprolog is the online control of agents in dynamic domains with incomplete information.

3.1 ALPprolog Programs

An ALPprolog program is an ALP that respects the following restrictions on the `?(Phi)` atoms in the program:

- All occurrences of non-fluent expressions in ϕ are positive.
- So called sense fluents $S(\vec{x})$ that represent the interface to a sensor may only occur in the form `?(s(X))`. Sense fluents are formally introduced below.

Because ALPprolog programs are meant for online execution the programmer must ensure that no backtracking over action executions occurs, by inserting cuts after all action occurrences. Observe that this applies to sensing actions, too. It is readily checked that — after the insertion of cuts — the ALP from example 1 satisfies all of the above conditions.

3.2 Propositional Fluent Calculus

In this section we introduce the announced propositional fragment of the Fluent Calculus. The discussion of sensing is deferred until section 3.3.

For ease of modelling we admit finitely many ground terms for fluents and objects, instead of working directly with propositional letters. An action domain $\mathcal{D}$ is then made propositional by including the respective domain closure axioms. For actions, objects, and fluents unique name axioms are included — hence we can avoid equality reasoning.

The basic building block of both the propositional Fluent Calculus and ALPprolog are the so-called prime implicates of a state formula $\phi(s)$:

Definition 4 (*Prime Implicate*)
A clause ψ is a *prime implicate* of ϕ iff it is entailed by ϕ, is not a tautology, and is not entailed by another prime implicate.

The prime implicates of a formula are free from redundancy — all tautologies and implied clauses have been deleted. For any state formula an equivalent *prime state formula* can be obtained by first transforming the state formula into a set of clauses, and by then closing this set under resolution, and the deletion of subsumed clauses and tautologies.

Prime state formulas have the following nice property: Let ϕ be a prime state formula, and let ψ be some clause (not mentioning auxiliary predicates); then ψ is entailed by ϕ if and only if it is subsumed by some prime implicate in ϕ, a fact that has already been exploited for Golog (Reiter 2001a; Reiter 2001b). This property will allow us to reduce reasoning about state knowledge in ALPprolog to simple list look-up operations.

Formally the propositional version of the Fluent Calculus is defined as follows.

Definition 5 (*Propositional Fluent Calculus Domain*)
We stipulate that the following properties hold in propositional Fluent Calculus domains:

- The initial state $\mathcal{D}_{\text{Init}}$ is specified by a ground prime state formula.
- The state formulas $\phi(s_1)$ in action preconditions $\mathit{Poss}(a, s_1, s_2) \equiv \phi(s_1) \wedge s_2 = \mathit{Do}(a, s_1)$ are prime state formulas.
- The effect axioms are of the form

$$\begin{aligned}
&\mathit{Poss}(A(\vec{x}), s_1, s_2) \supset \\
&\quad \bigvee_k (\Phi_k[s_1] \wedge (\forall f)[(\bigvee_i f = f_{ki} \vee (\mathit{Holds}(f, s_1) \wedge \bigwedge_j f \neq g_{kj})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \equiv \mathit{Holds}(f, s_2)]),
\end{aligned}$$

  where each $\Phi_k[s_1]$ is a prime state formula. This implies that existentially quantified variables that may occur in case selection formulas (cf. definition 3) have been eliminated by introducing additional cases.
- Only so-called *modular* domain constraints (Herzig and Varzinczak 2007) may be included. Very roughly, domain constraints are modular if they can be compiled into the agent's initial state knowledge, and the effect axioms ensure that updated states also respect the domain constraints. In the Fluent Calculus this holds if the following two conditions are met (Thielscher 2011): Condition (1), says that for a state that is consistent with the domain constraints and in which an action $A(\vec{x})$ is applicable, the condition $\Phi_i[S]$ for at least one case i in the effect axiom for A holds. Condition (2) requires that any possible update leads to a state that satisfies the domain constraints. Formally, let S, T be constants of sort TIME. $\mathcal{D}_{\text{dc}}$ the domain constraints, $\mathcal{D}_{\text{Poss}}$ the precondition axioms, and $\mathcal{D}_{\text{Effects}}$ the effect axioms. The following must hold for every action $A(\vec{x})$: There exists $i = 1, \ldots, n$ such that

$$\models \mathcal{D}_{\text{dc}}[S] \wedge \pi_A[S] \wedge (\exists \vec{y}_i)\Phi_i[S], \tag{1}$$

  and for every such i,

$$\models \mathcal{D}_{\text{dc}}[S] \wedge \pi_A[S] \wedge \Upsilon_i[S, T] \supset \mathcal{D}_{\text{dc}}[T]. \tag{2}$$

  Non-modular, fully general domain constraints greatly complicate reasoning.
- Auxiliary time-independent axioms may be included if they can faithfully be represented in the Prolog dialect underlying the implementation. This deliberately sloppy condition is intended to allow the programmer to use her favourite Prolog library. However, we stipulate that auxiliary predicates occur only positively outside of $\mathcal{D}_{\text{aux}}$ in the action domain $\mathcal{D}$ in order to ensure that they can safely be evaluated by Prolog. They also must not occur in the initial state formula at all. The update mechanism underlying ALPprolog can handle only ground effects. Hence, if auxiliary atoms are used in action preconditions, case selection formulas of effect axioms, then it is the burden of the programmer to ensure that these predicates always evaluate to ground terms on those variables that also occur in the action's effects.

On the one hand clearly every propositional Fluent Calculus domain can be transformed to this form. On the other hand it is well known that in general compiling away the quantifiers in a state formula can result in an exponential blow-up, as can the conversion to conjunctive normal form. We believe that the simplicity of reasoning with prime implicates outweighs this drawback.

Propositional action domains can still be non-deterministic. For example, for an applicable action two different cases may be applicable at the same time. The resulting state would then be determined only by the disjunction of the cases' effects. What is more, it would be logically unsound to consider only the effects of one of the cases. For the online control of agents in ALPprolog we stipulate that for an applicable action at most a single case applies, greatly simplifying the update of the agent's state knowledge.

Definition 6 (*Deterministic Propositional Fluent Calculus*)
A propositional Fluent Calculus domain is deterministic if the following holds: Let a be an applicable ground action. Then there is at most one case of the action that is applicable in the given state.

For example, an action theory is deterministic if for each effect axiom all the cases are mutually exclusive. Next assume we have an applicable deterministic action with e.g. two case selection formulas $\phi(s)$ and $\neg\phi(s)$, where neither case is implied by the current state. Here, instead of updating the current state with the disjunction of the respective effects, ALPprolog will employ incomplete reasoning.

3.3 Propositional Fluent Calculus with Sensing

We make the following assumptions concerning sensing: At any one time, a sensor may only return a single value from a fixed set $\mathcal{R}$ of ground terms, the *sensing results*. However, the meaning of such a sensing result may depend upon the concrete situation of the agent.

Example 1 (cont.)
Assume that now one of the cells in the maze contains a deadly threat to our gold-hunting agent. If the agent is next to a cell containing the threat she perceives a certain smell, otherwise she doesn't: She can sense whether one of the neighbouring cells is unsafe; but the actual neighbouring cells are only determined by the agent's current location. ■

Definition 7 (*Sensor Axiom*)
A sense fluent $S(x)$ is a unary fluent that serves as interface to the sensor. We assume the sort SENSEFLUENT to be a subsort of sort FLUENT. A sensor axiom then is of the form

$$(\forall s, x, \vec{y}) \mathit{Holds}(S(x), s) \equiv \bigvee_{R \in \mathcal{R}} x = R \wedge \phi(x, \vec{y}, s) \wedge \psi(x, \vec{y}, s),$$

for a ground set of sensing results $\mathcal{R}$. Here $\phi(x, \vec{y}, s)$ is a prime state formula that selects a meaning of the sensing result R, whereas the pure prime state formula $\psi(x, \vec{y}, s)$ describes the selected meaning. We stipulate that sensor axioms (which are a form of domain constraint) may only be included if they are modular.

Clearly $\phi(x, \vec{y}, s)$ should be chosen so as to be uniquely determined in each state. If auxiliary axioms are used in $\phi(x, \vec{y}, s)$ then again the programmer must ensure that these evaluate to ground terms in order that a ground state representation can be maintained.

Example 1 (cont.)
The following is the sensor axiom for our gold-hunter:

$$
\begin{aligned}
&(\forall)\mathrm{Holds}(\mathrm{PerceiveSmell}(x), s) \equiv \\
&\quad x = \mathrm{true} \wedge \mathrm{Holds}(\mathrm{At}(\mathrm{Agent}, y), s) \wedge \mathrm{Neighbours}(y, \vec{z}) \wedge \bigvee_{z \in \vec{z}} \mathrm{Holds}(\mathrm{ThreatAt}(z), s) \\
&\quad \vee \\
&\quad x = \mathrm{false} \wedge \mathrm{Holds}(\mathrm{At}(\mathrm{Agent}, y), s) \wedge \mathrm{Neighbours}(y, \vec{z}) \wedge \bigwedge_{z \in \vec{z}} \neg\mathrm{Holds}(\mathrm{ThreatAt}(z), s))
\end{aligned}
$$

■

Theoretically, the combination of sensing with the online control of an agent is quite challenging: It is logically sound to to consider the disjunction of all possible sensing results for offline reasoning. In the online setting, however, upon the observation of a sensing result we henceforth have to accept this result as being true; that is, at runtime we *add* the result to the action theory, something which is logically unsound. On the other hand, it also does not make sense to stipulate that the sensing result be known beforehand.

3.4 Action Theory Representation

We continue by describing how the underlying action theory is represented in ALPprolog. As basic building block we need a representation for prime state formulas. For notational convenience we will represent $(\neg)\mathit{Holds}(f, s)$ literals by the (possibly negated) fluent terms only, and, by an abuse of terminology, we will call such a term $(\neg)f$ a fluent literal. A convenient Prolog representation for such a state formula is a list, where each element is either a literal (i.e. a unit clause) or a list of at least two literals (a non-unit clause). In the following we call such a list a PI-list.

Definition 8 (*Action Theory Representation*)
Action theories as defined in definition 6 are represented in ALPprolog as follows:

- The initial state is specified by a Prolog fact `initial_state(PI-List).`, where `PI-List` mentions only ground fluent literals. Domain constraints other than sensor axioms have to be compiled into `PI-List`.
- a Prolog fact `action(A,Precond,EffAx).`, for each action a, has to be included, where
  - — `A` is an action function symbol, possibly with object terms as arguments;
  - — `Precond` is a PI-list, the action's precondition;
  - — `EffAx` is a list of cases for the action's effects with each case being a pair `Cond-Eff`, where the effect's condition `Cond` is a PI-list, and the effects `Eff` are a list of fluent literals; and
  - — all variables in `EffAx` also occur in `Precond`.
- If present, auxiliary axioms $\mathcal{D}_{\text{aux}}$ are represented by a set of Prolog clauses. The predicates defined in the auxiliary axioms must be declared explicitly by a fact `aux(Aux).`, where `Aux` denotes the listing of the respective predicate symbols.

The sensor axioms are represented as Prolog facts `sensor_axiom(s(X),Vals).`, where

- `s` is a sense fluent with object argument `X`; and
- `Vals` is a list of `Val-Index-Meaning` triples, where
  - — `Val` is a pair `X-result_i`, where `result_i` is the observed sensing result;
  - — `Index` is a PI-list consisting of unit clauses; and
  - — `Meaning` is a PI-list, mentioning only fluent literals and only variables from `Val` and `Index`.

The sense fluents have to be declared explicitly by a fact `sensors(Sensors).`, where `Sensors` is a listing of the respective function symbols. This is necessary in order to distinguish sense fluents, ordinary fluents, and auxiliary predicates in PI-lists.

3.5 Reasoning for ALPprolog

Reasoning in ALPprolog works as follows: For evaluating the program atoms we readily resort to Prolog. The reasoner for the action theory is based on the principle of progression. Setting out from the initial state, upon each successful evaluation of an action's precondition against the current state description, we update the current state description by the action's effects.

Reasoning about the action comes in the following forms:

- Given a ground applicable action a, from the current state description $\phi(s_1)$ and the action's positive and negative effects compute the description of the next state $\psi(s_2)$ (the update problem).
- Given a description $\phi(s)$ of the current state, check whether $\{\phi(s)\} \cup \mathcal{D}_{\text{aux}} \models \psi(s)$, where $\psi(s)$ is some state formula in s, but not a sense fluent (the entailment problem).
- For a sensing action, i.e. a query $\mathit{Holds}(S(x), s)$, integrate the sensing results observed into the agent's state knowledge (the sensing problem).

In the following we consider each of these reasoning problems in turn.

3.5.1 The Update Problem

It turns out that solving the update problem is very simple. Let `State` be a ground PI-List, and let `Update` be a list of ground fluents. The representation of the next state is then computed in two steps:

(1) First, all prime implicates in `State` that contain either an effect from `Update`, or its negation, are deleted, resulting in `State1`.
(2) The next state `NextState` is given by the union of `State1` and `Update`.

Starting from a ground initial state only ground states are computed.

The correctness of this procedure can be seen e.g. as follows: In (Liu et al. 2006; Drescher et al. 2009) algorithms for computing updates in a Fluent Calculus based upon Description Logics have been developed. The above update algorithm constitutes a special case of these algorithms.

3.5.2 The Entailment Problem

When evaluating a clause ψ against a ground prime state formula ϕ, ψ is first split into the fluent part ψ_1, and the non-fluent part ψ_2. It then holds that ψ is entailed by ϕ if there is a ground substitution θ such that

- $\psi_1\theta$ is subsumed by some prime implicate in ϕ; or
- some auxiliary atom $P(\vec{x})\theta$ from ψ_2 can be derived from its defining Prolog clauses.

Computing that the clause ψ_1 is subsumed by ϕ can be done as follows:

- If ψ_1 is a singleton, then it must be a prime implicate of ϕ (modulo unification).
- Otherwise there must be a prime implicate in ϕ that contains ψ_1 (modulo unification).

Hence the entailment problem for ALPprolog can be solved by `member`, `memberchk`, and `subset` operations on sorted, duplicate-free lists.

The following example illustrates how reasoning in ALPprolog can be reduced to simple operation on lists. It also illustrates the limited form of reasoning about disjunctive information available in ALPprolog:

Example 2 (Disjunctions and Substitutions in ALPprolog)
Assume that the current state is given by `[[at(gold,4),at(gold,5)]]`. *Then the query* `?([at(gold,X)])` *fails, because we don't consider disjunctive substitutions. However, on the same current state the query* `?([[at(gold,X),at(gold,Y)]])` *succeeds with* `X=4` *and* `Y=5`. ∎

3.5.3 The Sensing Problem

Sensing results have to be included into the agent's state knowledge every time a sensing action is performed, i.e. a literal `?(s(X))` is evaluated. This works as follows:

- First we identify the appropriate sensor axiom `sensor_axiom(s(X),Vals)`.
- Next we identify all the `[X-result_i]-Index-Meaning` triples in `Vals` such that `result_i` matches the observed sensing result, and unify `X` with `result_i`.
- We then locate the unique `Index-Meaning` s.t. the current state entails `Index`.
- Finally, we adjoin `Meaning` to the current state and transform this union to a PI-list.

3.6 Soundness of ALPprolog

At the end of section 3.3 we have already mentioned that adding sensing results to the action theory at runtime makes the subsequent reasoning logically unsound wrt. the original program plus action theory. If we add the set of sensing results observed throughout a run of an ALPprolog program, however, then we can obtain the following soundness result:

Proposition 1 (Soundness of ALPprolog)
Let Π be a ALPprolog program on top of an action domain $\mathcal{D}$. Let Σ be the union of the sensor results observed during a successful derivation of the ALPprolog query Γ with computed answer substitution θ. Then $\mathcal{D} \cup \Pi \cup \Sigma \models \Gamma\theta$.

Proof (Sketch)
It is well-known that SLD-resolution is sound for any ordinary program atom. A query $(\exists)\mathit{Holds}(\phi, s)$, where ϕ is not a sense fluent, is only evaluated successfully if there is a substitution θ such that $\mathcal{D} \models (\forall)\mathit{Holds}(\phi, s)\theta$. Assume we observe the sensing result $R_i \in \mathcal{R}$ for a sense fluent $S(x, s)$. In general we have (cf. Definition 7):

$$\mathcal{D} \models (\forall)\mathit{Holds}(S(x,s)) \wedge \bigvee_{R \in \mathcal{R}} x = R, \text{ but } \mathcal{D} \not\models (\forall)\mathit{Holds}(S(x,s)) \wedge x = R_i.$$

For soundness, we have to add the observed sensing result as an additional assumption to the theory: $\mathcal{D} \cup \{\mathit{Holds}(S(R_i, s))\} \models (\forall)\mathit{Holds}(S(x, s)) \wedge x = R_i$. $\square$

4 Evaluation

We have evaluated the performance of ALPprolog via the so-called Wumpus World (Russell and Norvig 2003) that is a well-known challenge problem in the reasoning about action community. Essentially, the Wumpus World is an extended version of the gold-hunter domain from example 1. The main features that make it a good challenge problem are incomplete information in the form of disjunctions and unknown propositions, and reasoning about sensing results.

We have used both Flux and ALPprolog to solve Wumpus Worlds of size up to 32×32.[4] We have done this using three different modellings:

(1) In (Thielscher 2005b) a Flux model is described that uses quantification over variables — this is beyond ALPprolog.
(2) We have evaluated both languages on a ground model.
(3) We have artificially increased the size of the ground model by making the connections between cells part of the state knowledge.

A first observation is that both languages roughly scale equally well in all models. Using (1) Flux is slightly faster than ALPprolog using (2). Let us then point out that on ground models Flux and ALPprolog maintain the same state representation: Flux also computes the prime implicates. On the encoding (2) ALPprolog is roughly one order of magnitude faster than Flux, whereas on (3) the difference is already two orders of magnitude. The key to the good performance of ALPprolog then is that it handles large state representations well: By encoding states as *sorted* lists (of lists) some of the search effort necessary in Flux can be avoided. If, however, we use Flux' capability of handling quantified variables in the state knowledge for a more concise encoding, then ALPprolog and Flux are again on par, with Flux even having slightly the edge. In general, we expect ALPprolog to excel on problem domains that feature large state representations that are not easily compressed using quantification.

It has already been established that Flux gains continuously over standard Golog the more actions have to be performed (Thielscher 2005a). As ALPprolog scales as well as Flux the same holds for ALPprolog and Golog. The version of Golog with periodically progressed state knowledge is slightly slower than Flux (Sardina and Vassos 2005).

[4] The distribution of ALPprolog contains the Wumpus World example for both ALPprolog and Flux.

Let us also compare ALPprolog, Flux, and Golog from a knowledge representation perspective: Both ALPprolog and Flux allow the programmer to define new auxiliary predicates for the agent strategy that are not present in the action theory, a practically very useful feature that is missing from Golog. Also, the propositional variables used in Golog instead of the finitely many ground terms used in ALPprolog make it hard for the programmer to fully exploit the power of Prolog's unification mechanism. In this regard Flux, on the other hand, excels in that the programmer can include fluents containing (possibly quantified) variables in the agent's state knowledge. Contrary to ALPprolog and Golog, however, Flux does not support arbitrary disjunctions.

5 Conclusion and Future Work

In this work we have presented ALPprolog, an efficient logic programming language for the online control of autonomous agents in domains that feature incomplete information and sensing. On the one hand, it can be argued that the state-of-the-art languages Golog and Flux already successfully address this application domain. On the other hand, we have shown that ALPprolog excels because of its efficient reasoning with large ground state representations, something that we expect to be quite useful in practice.

For future work, there are two interesting directions: On the one hand it would be nice to extend ALPprolog to offline planning. The disjunctive substitutions in the general ALP proof calculus provide a powerful form of reasoning about conditional plans, or planning in the presence of sensing in the sense of (Levesque 1996).

On the other hand we plan to fruitfully apply ALPprolog in the domain of General Game Playing. General Game Playing (Genesereth et al. 2005) is a new exciting AI research challenge aiming at the integration of manifold AI techniques: A program (also called a player) is given an axiomatisation of the rules of a game. The player then computes a strategy/heuristic that it uses to play and hopefully win the game. The main challenge of General Game Playing consists of constructing suitable heuristics.

However, at its base the player also needs a means to represent, and reason about, the state of the game. Up to now the games played in General Game Playing have been restricted to complete information (Love et al. 2008) — but clearly games with incomplete information constitute a bigger challenge (Thielscher 2010a). We intend to include techniques from ALPprolog into the successful Flux-based Fluxplayer (Schiffel and Thielscher 2007).

Acknowledgements. We appreciate the helpful comments by the reviewers. This work was partially supported by DFG Grant TH 541/14. C. Drescher wishes to acknowledge support by EPSRC Grant EP/G055114/1. M. Thielscher is the recipient of an Australian Research Council Future Fellowship (project number FT 0991348). He is also affiliated with the University of Western Sydney.

References

BORDINI, R., HÜBNER, J., AND WOOLDRIDGE, M. 2007. *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley.

DRESCHER, C., LIU, H., BAADER, F., GUHLEMANN, S., PETERSOHN, U., STEINKE, P., AND THIELSCHER, M. 2009. Putting abox updates into action. In *Proceedings of the Seventh International Symposion on Frontiers of Combining Systems (FroCoS 2009)*. Trento, Italy.

DRESCHER, C., SCHIFFEL, S., AND THIELSCHER, M. 2009. A declarative agent programming language based on action theories. In *Proceedings of the Seventh International Symposion on Frontiers of Combining Systems (FroCoS 2009)*. Trento, Italy.

GENESERETH, M. R., LOVE, N., AND PELL, B. 2005. General game playing: Overview of the AAAI competition. *AI magazine 26,* 2, 62–72.

HERZIG, A. AND VARZINCZAK, I. 2007. Metatheory of actions: Beyond consistency. *Artificial Intelligence 171,* 16–17, 951–984.

J.W. LLOYD. 1987. *Foundations of Logic Programming*. Springer.

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,* 1–3, 59–83.

LEVESQUE, H. J. 1996. What is planning in the presence of sensing? In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996)*. Portland, Oregon, USA, 1139–1146.

LIU, H., LUTZ, C., MILICIC, M., AND WOLTER, F. 2006. Updating description logic ABoxes. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR 06)*. Lake District of the UK.

LOVE, N., HINRICHS, T., HALEY, D., SCHKUFZA, E., AND GENESERETH, M. 2008. General game playing: Game description language specification. Tech. rep., Stanford University.

MCCARTHY, J. AND HAYES, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4,* B. Meltzer and D. Michie, Eds. Edinburgh University Press, 463–502.

REITER, R. 2001a. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, MA.

REITER, R. 2001b. On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic 2,* 4, 433–457.

RUSSELL, S. J. AND NORVIG, P. 2003. *Artificial Intelligence: a modern approach,* 2nd international edition ed. Prentice Hall, Upper Saddle River, N.J.

SARDINA, S. AND VASSOS, S. 2005. The wumpus world in IndiGolog: A preliminary report. In *Proceedings of the Workshop on Nonmonotonic Reasoning, Action and Change at IJCAI*.

SCHIFFEL, S. AND THIELSCHER, M. 2007. Fluxplayer: A successful general game player. In *Proceedings of the Twenty-second National Conference on Artificial Intelligence (AAAI 2007)*. AAAI Press, Menlo Park, CA, 1191–1196.

THIELSCHER, M. 1999. From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence 111,* 1–2, 277–299.

THIELSCHER, M. 2005a. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming 5,* 4–5, 533–565.

THIELSCHER, M. 2005b. A FLUX agent for the Wumpus World. In *Proceedings of the Workshop on Nonmonotonic Reasoning, Action and Change at IJCAI*, L. Morgenstern and M. Pagnucco, Eds. Edinburgh, UK, 104–108.

THIELSCHER, M. 2010a. A general game description language for incomplete information games. In *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI 2010)*. AAAI Press, Atlanta, 994–999.

THIELSCHER, M. 2010b. Integrating action calculi and AgentSpeak: Closing the gap. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*. Toronto, 79–89.

THIELSCHER, M. 2011. A unifying action calculus. *Artificial Intelligence Journal 175,* 120–141.