Declarative Strategies for Agents with Incomplete Knowledge

Gerhard Brewka and **Hannes Strass**
Computer Science Institute
University of Leipzig
{brewka, strass}@informatik.uni-leipzig.de

Michael Thielscher
School of Computer Science and Engineering
The University of New South Wales
mit@cse.unsw.edu.au

Abstract

Definite Agent Logic Programs (definite ALPs), recently introduced by Drescher, Schiffel and Thielscher, provide an interesting alternative to imperative languages like GOLOG for specifying agent behavior. The main advantages of ALPs is that they are fully declarative and independent of the underlying action theory. In this paper we extend the expressiveness of ALPs by introducing nonmonotonic negation in rule bodies. This allows us to handle incomplete knowledge, and to represent preferences among different agent behaviors. The semantics of ALPs with negation is based on a variant of answer set semantics.

Introduction

Knowledge representation languages for actions and change are traditionally concerned with formalizing preconditions and effects of actions. Agents can then use this knowledge to reason about the consequences of their actions and to plan. A variety of approaches have been developed in the past, including the classical Situation Calculus (McCarthy 1968), the Event Calculus (Mueller 2006), and Action Languages (Gelfond and Lifschitz 1998), to mention but a few.

In practice, however, intelligent agents do not just rely on action knowledge. While for example Action Languages like $\mathcal{A}$ and $\mathcal{C}$ and their straightforward implementations in answer set programming (Gelfond 2008) can represent the dynamics of a given domain, they offer no possibility to express heuristic strategies of an agent acting in this domain. A strategy, intuitively, is a specification of how an agent should proceed in order to achieve a goal. For this purpose, knowledge-based agents additionally employ *behavioral* knowledge, which resides on top of the domain knowledge and guides agents' actions under different circumstances.

The need to specify strategies led to the development of GOLOG, a procedural language for agents that use a Situation Calculus theory to reason about their actions (Levesque et al. 1997). As an alternative, (Drescher, Schiffel, and Thielscher 2009) recently developed (definite) Agent Logic Programs, a declarative language for providing agents with strategies using the syntax of Horn clauses. The basic underlying idea is the following: a strategy can be viewed as a way of splitting a goal into sequences of actions and subgoals, possibly depending on certain conditions that need to hold in a particular situation. This can conveniently be represented using rules with goals in the heads and a sequence of actions/subgoals, possibly augmented by additional tests, in the body.[1] As an example, consider this simple strategy for going to the airport:

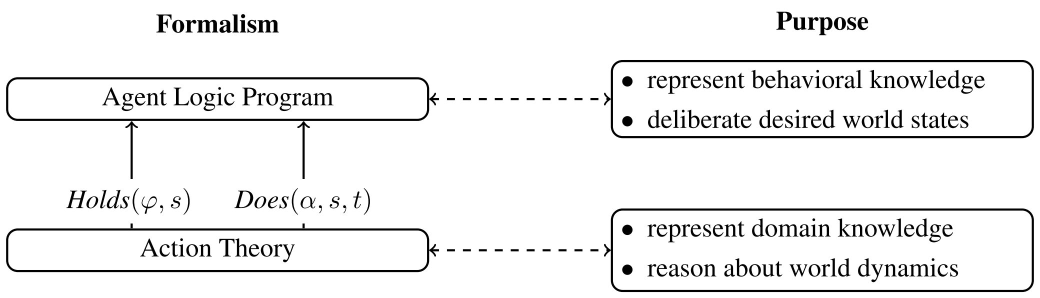
$$\begin{aligned}\texttt{goToAirport} \leftarrow\ & \mathit{Holds}(\mathsf{At}(\mathsf{Car}, \mathsf{Home})),\\ & \mathit{Does}(\mathsf{Drive}(\mathsf{Airport}))\\ \texttt{goToAirport} \leftarrow\ & \mathit{Holds}(\mathsf{At}(\mathsf{Car}, \mathsf{Office})),\ \texttt{goByTaxi}\end{aligned}$$

Intuitively, the rules say: the goal *go to the airport* can be achieved by the action *drive to the airport* in case the car is at home. If the car is at the office it can be achieved by *going by taxi*. Going by taxi here is a subgoal for which the Agent Logic Program needs to have further strategies, that is, rules specifying possible ways of achieving this subgoal.

The declarative semantics of definite Agent Logic Programs is given by the standard interpretation of definite logic programs in combination with an action theory to evaluate the special predicates *Holds* and *Does*. The operational semantics is obtained by incorporating a reasoner for the agent's background action knowledge into standard SLD-resolution, whereby the action theory is effectively seen as a black box and reasoning about the domain is out-sourced. (See also Figure 1 on the next page.) This entails in particular that Agent Logic Programs are completely independent of the underlying action formalism, which allows to use them in combination with (almost) any action calculus (Drescher, Schiffel, and Thielscher 2009). Given the vast number of different action formalisms in the literature, this independence is a significant advantage in comparison to other languages for specifying agent behavior. The aforementioned GOLOG, for example, is restricted to action theories formulated in the language of the Situation Calculus (Levesque et al. 1997).

The restriction to Horn clauses, however, severely limits the usability of definite Agent Logic Programs as they can be meaningfully applied only for agents with complete knowledge of all relevant aspects of their environments. If, say, in the above example the agent does not know where the

[1] Thus, the order of elements in the rule body does matter. The semantics of the rules will later be defined in terms of expansions of the rules which contain additional variables. The expanded rules indeed are order-independent and thus fully declarative.



Formalism
Purpose
Agent Logic Program
• represent behavioral knowledge
• deliberate desired world states
$Holds(\varphi, s)$
$Does(\alpha, s, t)$
Action Theory
• represent domain knowledge
• reason about world dynamics

Figure 1: Agent logic programs serve to express strategic information used by an agent to guide its actions. Action theories express information about the domain and how it changes in response to actions. Agent logic programs employ action theories to reason about domain dynamics, where the information flow is strictly from action theory to agent logic program. Through a carefully designed interface of two special predicates, ALPs can be used with a wide variety of existing action theories.

car is, then neither of the behavioral rules can be applied. Moreover, it does not help to use the extended expressiveness described in (Drescher, Schiffel, and Thielscher 2009) of querying an action theory with arbitrary formulas using the special predicate *Holds*, as in

$$\texttt{goToAirport} \leftarrow \mathit{Holds}(\neg\mathsf{At}(\mathsf{Car},\mathsf{Home})),\ \texttt{goByTaxi} \quad (1)$$

To apply this clause, it is not enough for the agent not to know that the car is at home; rather, the agent's background theory must explicitly entail that the car is *not* at home.

In this paper, we therefore substantially extend the concept of (definite) Agent Logic Programs by nonmonotonic negation. This will allow us to replace a rule like (1) with

$$\texttt{goToAirport} \leftarrow \text{not } \mathit{Holds}(\mathsf{At}(\mathsf{Car},\mathsf{Home})),\ \texttt{goByTaxi}$$

The operational meaning of this rule is very different from (1), as the new clause will be applicable as soon as the background theory does not entail $\mathit{Holds}(\mathsf{At}(\mathsf{Car},\mathsf{Home}))$. Our extended Agent Logic Programs are thus suitable for agents with incomplete knowledge.

Interestingly, we will also allow negated actions and subgoals in bodies of rules. Intuitively, a literal of the form not $\mathit{Does}(\mathsf{Drive}(\mathsf{Airport}))$ reads: the action $\mathsf{Drive}(\mathsf{Airport})$ is impossible, and not `goByTaxi` reads: goal `goByTaxi` is not achievable. We will later see how this feature can be used to specify preferences among different strategies.

Of course, all this requires a new declarative semantics, which we will develop in this paper. We will show that these extended Agent Logic Programs are suitable for agents with incomplete knowledge. Moreover, we will demonstrate that our nonmonotonic negation allows to specify preferences among different behaviors. From the original concept of definite Agent Logic Programs we will inherit the generality of the entire framework, keeping it independent of a particular action calculus. In fact, we will show that this allows us to combine features from different formalisms and use branching time for planning and linear time for plan verification.

The paper is organized as follows. The next section provides the necessary background on action theories and definite Agent Logic Programs. We then define syntax and semantics of Agent Logic Programs with nonmonotonic negation, the main contribution of the paper, and we show that the new declarative semantics is a proper generalization of the semantics for definite programs under complete information. Thereafter, we illustrate how different time structures can be used for different reasoning problems, and we also present an extension of Agent Logic Programs by the concept of static predicates. Finally, we illuminate the expressiveness of ALPs in comparison to GOLOG and provide a complexity result for reasoning with Agent Logic Programs.

Background

Action Theories

Action theories are used to formalize knowledge of actions in order to enable an agent to reason about their preconditions and effects. Many different representation languages exist, and we intend to make this paper as general as possible by retaining the independence of Agent Logic Programs from the specifics of an underlying action calculus. We therefore just require a background axiomatization to provide a signature and to implicitly define an entailment relation. The signature will be used by the agent logic program to access the vocabulary used to describe the domain. The entailment relation will be used to determine whether a (possibly complex) world property holds at a particular time point and whether an action is executable.

Definition 1. An *action theory* consists of the following.

- A *domain signature* Σ that includes sorts ACTION, FLUENT, and TIME along with
  - a constant S_0 : TIME (the *initial* time-point);
  - a predicate *Holds* : FLUENT $\times$ TIME;
  - a predicate *Does* : ACTION $\times$ TIME $\times$ TIME.

  A *fluent formula* φ in Σ is a first-order formula whose atoms are terms of sort FLUENT. With a slight abuse of notation, we denote by $\mathit{Holds}(\varphi, s)$ the formula obtained from fluent formula φ by replacing every occurrence of a fluent f by $\mathit{Holds}(f, s)$.
- An *entailment relation* $\vdash$ for
  - $\mathit{Holds}(\varphi, s)$, where φ is a fluent formula in Σ;
  - $\mathit{Does}(\alpha, s_1, s_2)$, where α is an ACTION term in Σ.

Fluent formulas serve to specify properties of the world while abstracting away from specific time points. For example, the fluent formula $\varphi = \mathsf{At}(\mathsf{Home}) \wedge \mathsf{At}(\mathsf{Car}, \mathsf{Office})$ says that the agent is at home but the car is at the office. The entailment relation provided by an action theory now can be used to check whether the world property expressed in the fluent formula holds at a specific time point. For example, the notation $\mathit{Holds}(\varphi, S_0)$ expands to the formula $\mathit{Holds}(\mathsf{At}(\mathsf{Home}), S_0) \wedge \mathit{Holds}(\mathsf{At}(\mathsf{Car}, \mathsf{Office}), S_0)$ stating that *initially* the agent is at home and the car at the office. Such an expansion is merely a technically convenient way of querying the action theory for general properties while remaining able to treat expressions like $\mathit{Holds}(\varphi, S_0)$ as atomic in the agent logic program.

Prominent examples of action formalisms are the Situation- and the Fluent Calculus, both of which are based on branching time. In these instances, $\mathit{Does}(\alpha, s_1, s_2)$ means that action α is possible in situation s_1 and leads to situation s_2. Formalisms like the Event Calculus, on the other hand, use linear time, in which case $\mathit{Does}(\alpha, t_1, t_2)$ means that action α is actually executed starting at time t_1 and ending at time t_2.

Example 1 (Elevator Domain in Situation/Event Calculus)**.** An elevator can move to different floors, and open and close its doors to serve requests. The branching time structure of situations can be used to reason about different possible futures of this domain. A future situation is considered reachable if there is an executable sequence of actions leading to it. For example, for the action of moving the elevator to floor n to be executable, the elevator needs to be currently at another floor with its doors closed:[2]

$$\mathit{Poss}(\mathsf{Move}(n), s) \supset (\exists m)(\mathit{Holds}(\mathsf{At}(m), s) \wedge m \neq n \wedge \mathit{Holds}(\mathsf{Closed}, s)) \quad (2)$$

The effects of the action are stated by, "after moving from m to n, the elevator is at n and no longer at m":

$$\mathit{Holds}(\mathsf{At}(n), \mathit{Do}(\mathsf{Move}(n), s)) \quad (3)$$

$$\mathit{Holds}(\mathsf{At}(m), s) \supset \neg \mathit{Holds}(\mathsf{At}(m), \mathit{Do}(\mathsf{Move}(n), s)) \quad (4)$$

Formulas of the above forms enable us to reason about action executability at certain time points (axiom (2)) and to make predictions about action effects (axioms (3,4)). Agent Logic Programs use these features for plan generation.

Unlike the hypothetical nature of situations, linear time is used to reason about effects of actions that actually occurred. We will illustrate this by axiomatizing the action $\mathsf{TurnOff}(n)$ for deactivating a request after it has been carried out:

$$\mathit{Happens}(\mathsf{TurnOff}(n), t_1, t_2) \supset \neg \mathit{Holds}(\mathsf{Request}(n), t_2) \quad (5)$$

Linear time structures can be used to reason about given sequences of actions, e.g. to verify executability of plans.

In this paper, we do not want to commit ourselves to a particular time structure. Rather, different time structures can be used for different purposes – e.g., linear time for plan *verification* and branching time for plan *generation*. To ease presentation, we unify notation from different action calculi:

- $\mathit{Happens}(a, s, t)$ becomes $\mathit{Does}(a, s, t)$;
- $\mathit{Poss}(a, s)$ also becomes $\mathit{Does}(a, s, t)$, and for a branching time structure we tacitly assume given the axiom

$$\mathit{Does}(a, s, t) \supset t = \mathit{Do}(a, s)$$

- $(\neg)\mathit{Holds}(f, \mathit{Do}(a, s))$ in situation-based effect axioms becomes $\mathit{Does}(a, s, t) \supset (\neg)\mathit{Holds}(f, t)$.

Hence $\mathit{Does}(a, s, t)$ denotes hypothetical or actual occurrence of action a from s to t. This allows us to treat action theories with different notions of time in a uniform way.

Example 1 (Continued)**.** Different action formalisms use different solutions to the frame problem. Reiter (2001), for example, applies a form of completion to Situation Calculus-style precondition and effect axioms, with the latter being transformed into so-called *successor state axioms*. For Event Calculus, Shanahan (1997) embeds domain axioms in a circumscribed theory.

Since we are not concerned with the frame problem here, we just assume action theories to include a solution to the frame problem that is suitable for the underlying time structure. In the following, we will slightly overload the symbol $\Delta_{\mathtt{Elevator}}$ to contain axioms (2–5), where *Poss* and *Happens* have been replaced by *Does* as indicated above, augmented by any solution to the frame problem that suits the time structure (branching or linear) that is given in the context in which we use the theory $\Delta_{\mathtt{Elevator}}$.

Hence the technical specifics of the underlying action theory are quite irrelevant for agent logic programs, as long as the action theory provides the interface predicates *Holds* and *Does* along with the entailment relation $\vdash$. An underlying action formalism could even be nonmonotonic *itself* – that is, provide a nonmonotonic entailment relation $\vdash$ –, like the modal logic approach of (Lakemeyer and Levesque 2009), the default theories of (Baumann et al. 2010) or the deductive argumentation frameworks of (Michael and Kakas 2011). This generous abstraction is possible because the underlying action formalism is not fully embedded into Agent Logic Programs, but only accessed through the predicates *Holds* and *Does*. The details will be provided next.

Definite Agent Logic Programs

(Drescher, Schiffel, and Thielscher 2009) have introduced Agent Logic Programs as a declarative knowledge representation formalism to provide agents with strategies. As such, they allow to supply *behavioral* knowledge in addition to action knowledge. In the following, we briefly recapitulate the basic definition of definite Agent Logic Programs, henceforth abbreviated DALPs.

Definition 2. Consider an action theory signature Σ with sorts ACTION and FLUENT, and let Π be a logic program signature.

- *Terms* are from $\Sigma \cup \Pi$.
- If P is an n-ary relation symbol from Π and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is an *(ordinary) program atom.*
- $\mathit{Does}(\alpha)$ is a *(special) program atom* if α is an ACTION term in Σ.

[2]We denote variables by lowercase letters. Unbound variables in formulas are implicitly assumed to be universally quantified.

- *Holds*(φ) is a *(special) program atom* if φ is a *fluent formula* in Σ, that is, a formula (represented as a term) based on the FLUENTS in Σ.
- Clauses and programs are then defined as usual for definite logic programs, with the restriction that only ordinary program atoms can occur as head of a clause.

We will utilize a well-known agent programming domain (Levesque et al. 1997) to familiarize the reader with how we use Agent Logic Programs to specify behavior.

Example 2 (Elevator Control)**.** We present a declarative agent program for controlling an elevator. Where GOLOG uses *procedures* to encapsulate complex behaviors, in ALPs behaviors are expressed by predicates that are defined as usual in logic programming. For instance, to serve a floor the elevator must first move to the floor, then open its doors, deactivate the request and lastly close the doors.

$$\texttt{serve}(n) \leftarrow \texttt{moveTo}(n),\ \textit{Does}(\mathsf{Open}),\ \textit{Does}(\mathsf{TurnOff}(n)),\ \textit{Does}(\mathsf{Close}) \quad (6)$$

The strategy for moving to a floor succeeds if the elevator is at the desired floor or if it can move there.

$$\texttt{moveTo}(n) \leftarrow \textit{Holds}(\mathsf{At}(n)) \quad (7)$$
$$\texttt{moveTo}(n) \leftarrow \textit{Does}(\mathsf{Move}(n)) \quad (8)$$

The main strategy for the control program now consists of two clauses: the first clause uses a substrategy to determine the next floor to serve, another substrategy to then serve the floor and finally continues to follow the same main strategy.

$$\texttt{elevatorControl} \leftarrow \texttt{nextFloor}(n),\ \texttt{serve}(n),\ \texttt{elevatorControl} \quad (9)$$

The second clause is applicable when there is definitely no request on any floor, in which case the elevator is parked.

$$\texttt{elevatorControl} \leftarrow \textit{Holds}((\forall n)\neg\mathsf{Request}(n)),\ \textit{Does}(\mathsf{Park}) \quad (10)$$

In this simple initial version, the strategy to determine the next floor to serve just checks for a respective request.

$$\texttt{nextFloor}(n) \leftarrow \textit{Holds}(\mathsf{Request}(n)) \quad (11)$$

From an operational semantics point of view, rule (9) for serving a floor above constitutes an infinite loop. From a strictly logical point of view, taking $\leftarrow$ to be implication, it is a tautology. The reader might be asking, “What is the intended meaning of this clause, or ALPs in general?” The detailed answer will be given later; for now, it suffices to view rule bodies as sequentially executed substrategies.

It is more important to see that in the example above, the specified elevator control strategy succeeds only if all non-requested floors are *known*. Alas, this is not due to bad modeling or poor programming: it is an inherent restriction of allowing only definite Horn clauses. In the next section, we shall generalize ALPs to overcome this restriction.

Agent Logic Programs with Negation

We now move from definite Horn logic programs to *normal* logic programs, where the rule body may contain atoms and default-negated negated atoms.

Definition 3. Consider an action theory signature Σ with sorts ACTION and FLUENT, and let Π be a logic program signature. An *agent logic program with negation* (ALP) is a set of rules $H \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$ where the *head* H is an ordinary program atom (cf. Definition 2); and each *body literal* B_i is a program atom possibly preceded by “not”.

Extending Agent Logic Programs with negation opens up a whole new range of features for agent programming: we can deal with incomplete knowledge and we can express preferences between different behaviors. We demonstrate these capabilities extending the ALP seen so far.

Example 2 (Continued)**.** Imagine that call buttons frequently get broken, and as a result thereof knowledge about requests becomes incomplete. Instead of serving only floors for which there definitely is a request, the following rule also chooses floors for which the action theory does not entail that there is no request (i.e., there *might* be a request):

$$\texttt{nextFloor}(n) \leftarrow \text{not } \textit{Holds}(\neg\mathsf{Request}(n)) \quad (12)$$

With the two rules (11,12), the strategy $\texttt{nextFloor}(n)$ selects possibly requested floors in no particular order. Alternatively, clause (11) together with (13,14) below express a strategy with a qualitative preference: first, all of the definite requests are handled; after that, the requests on speculation (e.g. due to a broken button) are considered.

$$\texttt{definiteRequest} \leftarrow \textit{Holds}(\mathsf{Request}(n)) \quad (13)$$
$$\texttt{nextFloor}(n) \leftarrow \text{not } \texttt{definiteRequest},\ \text{not } \textit{Holds}(\neg\mathsf{Request}(n)) \quad (14)$$

The next definition now takes the first step towards defining a formal, declarative semantics for Agent Logic Programs with negation. It determines how the concept of time that passes with action execution is treated by the program.

We first distinguish between progressing and non-progressing literals. Intuitively, progressing literals are those which consume time when being executed. For the time being we assume all non-negated atoms except *Holds* atoms to be progressing (static atoms will be discussed later). In contrast, negated atoms and all *Holds* atoms are not progressing. For *Does* and (sub)strategy atoms this reflects the intuition that negation here represents impossibility of an action or failure of a strategy, which do not consume time.

Consider the rule $r = H \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$. The progression index of B_1, $pi(B_1)$ is 1. For $1 < i \leq n$ define $pi(B_i) = pi(B_{i-1})$ if B_i is not progressing, and $pi(B_i) = pi(B_{i-1}) + 1$ otherwise. The progression index of rule r as a whole is 1 if $n = 0$, $pi(B_n) + 1$ if $n > 0$ and B_n is progressing, $pi(B_n)$ otherwise.

To treat quantification of time points correctly, we introduce new (internal) predicates: *Doable*(a, s) says that action a can be executed in s leading to some (at this point irrelevant) resulting time-point; for a predicate $P \in \Pi$, the atom $\textit{Succ}_P(\vec{t}, s)$ means that strategy $P(\vec{t})$ succeeds when executed starting at s.

Definition 4. Let P be an ALP, $r = H \leftarrow B_1, \ldots, B_n$ ($n \geq 0$) a rule in P with progression index j, and let $s_1, \ldots, s_j$ be pairwise distinct variables of sort TIME. The *expansion* of r, denoted $Exp(r)$, is the rule

$$Exp(H) \leftarrow Exp(B_1), \ldots, Exp(B_n)$$

where

- For $H = P(\vec{t})$, $Exp(H) = P(\vec{t}, s_1, s_j)$.
- For B_i $(i = 1, \ldots, n)$ with $pi(B_i) = k$:
  - if $B_i = P(\vec{t})$ for some $P \in \Pi$ then: $Exp(B_i) = P(\vec{t}, s_k, s_{k+1})$;
  - if $B_i = \mathit{Does}(\alpha)$ then: $Exp(B_i) = \mathit{Does}(\alpha, s_k, s_{k+1})$;
  - if $B_i = (\text{not})\mathit{Holds}(\varphi)$ then: $Exp(B_i) = (\text{not})\mathit{Holds}(\varphi, s_k)$;
  - if $B_i = \text{not } \mathit{Does}(\alpha)$ then: $Exp(B_i) = \text{not } \mathit{Doable}(\alpha, s_k)$;
  - if $B_i = \text{not } P(\vec{t})$ for some $P \in \Pi$ then: $Exp(B_i) = \text{not } \mathit{Succ}_P(\vec{t}, s_k)$.

The expansion of P, $Exp(P)$, is the program

$$\{Exp(r) \mid r \in P\} \cup \{\mathit{Doable}(a, s) \leftarrow \mathit{Does}(a, s, s')\} \cup \{\mathit{Succ}_P(\vec{t}, s_i) \leftarrow P(\vec{t}, s_i, s') \mid P \in \Pi\}$$

Example 2 (Continued)**.** Applying the above definition to clause (9) shows how to actually make sense of this program rule for the main loop of serving requests:

$$\texttt{elevatorControl}(s_1, s_4) \leftarrow \texttt{nextFloor}(n, s_1, s_2), \texttt{serve}(n, s_2, s_3),\ \texttt{elevatorControl}(s_3, s_4)$$

It says: the strategy to serve requests from s_1 to s_4 is composed of sequentially executed sub-strategies, one for determining the next floor to serve and the next for actually dealing with the request. The recursive call in the last atom concludes the declarative equivalent of a procedural loop.

We are now in a position to finalize the definition of the semantics of ALPs. The basic idea is to take the expansion of an ALP and to apply answer set semantics to it. However, contrary to regular answer set semantics, in our case some of the literals in the bodies of rules have a meaning which is determined outside the program itself, namely in the underlying action theory.

Definition 5. Let P be an ALP, P_{exp} the ground instantiation of its expansion. Let Δ be an action theory. A set of atoms S is an *answer set of* P *under* Δ iff it is the least model of the program $P_{exp}^{S,\Delta}$ that is obtained from P_{exp} by

1. deleting each rule with a body literal L satisfying one of the following conditions:
   - $L = \text{not } A$ for some $A \in S$,
   - $L = \mathit{Holds}(\varphi, s)$ or $L = \mathit{Does}(a, s, s')$ and $\Delta \not\vdash L$,
   - or $L = \text{not } \mathit{Holds}(\varphi, s)$ and $\Delta \vdash \mathit{Holds}(\varphi, s)$.
2. deleting all default negated literals and all literals with predicate *Holds* or *Does* from the remaining rules.

Note that the ground instantiation P_{exp} actually depends on the action theory as different ground terms may be used for representing time points (expressions built using the Do function in Situation Calculus, numerical time stamps in the Event Calculus). In any case, the interesting information about a strategy $\texttt{p}$ can be read off atoms of the form $\texttt{p}(\vec{o}, s_1, s_2)$ in the answer set(s). For instance, in Situation Calculus the term s_2 will specify an action sequence corresponding to a successful strategy execution starting in s_1.

Based on the definition of answer sets, we can introduce a skeptical and a credulous notion of consequence, where a formula Q is a skeptical (credulous) consequence of a program P under Δ iff Q follows from $S \cup \Delta$ for all answer sets S (some answer set S) of P under Δ.

Example 3. Let $\Delta_{\texttt{Elevator}}$ contain a specification of an initial state where the elevator is at floor 0, there is a request for floor 2 and definitely no requests for floors other than the first or second (that is, there might be a request for the first floor). This amounts to the formulas

$$\mathit{Holds}(\mathsf{At}(0), S_0), \mathit{Holds}(\mathsf{Request}(2), S_0), (n \neq 2 \wedge n \neq 1) \supset \neg \mathit{Holds}(\mathsf{Request}(n), S_0)$$

Now let the Agent Logic Program $P_{\texttt{Elevator}}$ consist of the rules (6)–(12) from Example 2. Under the action theory $\Delta_{\texttt{Elevator}}$, there is an answer set S of $P_{\texttt{Elevator}}$ which contains the plan of first serving floor 1 and then floor 2. Technically, the existence of this plan is witnessed by the atom $\texttt{elevatorControl}(S_0, Do(\alpha, S_0)) \in S$, where $\alpha = [\mathsf{Move}(1), \mathsf{Open}, \mathsf{TurnOff}(1), \mathsf{Close}, \mathsf{Move}(2), \ldots, \mathsf{Park}]$.[3] When we replace clause (12) of $P_{\texttt{Elevator}}$ by (13,14), however, this plan will not be contained in any answer set: the strategy encoded by clauses (11) and (13,14) explicitly forbids dealing with default requests before definite requests. This shows how the nonmonotonic semantics of ALPs handles preferences among behaviors.

The semantics defined in this paper is indeed a proper generalization of the existing semantics for definite ALPs and complete knowledge:

Proposition 1. *Let P be an ALP without negation* not *in the body of any rule. Then, for each action theory Δ, P possesses a single answer set S_Δ under Δ. Moreover, if Δ is complete, then the set of ordinary program atoms in the answer set coincides with the set of ground instances of ordinary program atoms entailed by P given Δ according to the definition of the declarative semantics in (Drescher, Schiffel, and Thielscher 2009).*

Proof. If P is an ALP without negation not, then $P_{exp}^{S,\Delta}$ is the same for any S, and by Definition 5 the least model of this unique clause set is the only answer set of P under Δ. The expansion defined in (Drescher, Schiffel, and Thielscher 2009) coincides with $Exp(P)$ in case of definite ALPs; hence, the existing declarative semantics for P under Δ is given by the theory $Exp(P) \cup \Delta$. Moreover, completeness of Δ means that $\Delta \vdash L$ or $\Delta \vdash \neg L$ for any ground

[3] $Do([a_1, \ldots, a_n], s)$ abbreviates $Do(a_n, \ldots Do(a_1, s) \ldots)$.

instance L of the special atoms *Holds* and *Does*. Consequently, a ground instance of an ordinary program atom is logically entailed by $Exp(P) \cup \Delta$ if, and only if, it is in the least model of the program obtained from P_{exp} by

1. deleting each rule with a body literal $L = \mathit{Holds}(\varphi, s)$ or $L = \mathit{Does}(a, s, s')$ such that $\Delta \vdash \neg L$;
2. deleting all literals with *Holds* or *Does* from the remaining rules.

In case of definite programs this is equivalent to the construction in Definition 5, which proves the claim. □

It is noteworthy that the equivalence of the two semantics does not generalize to arbitrary background theories. A simple counter-example is the expanded definite ALP

$$\begin{aligned} \mathtt{p}(s_1, s_1) &\leftarrow \mathit{Holds}(\mathsf{F}, s_1) \\ \mathtt{p}(s_1, s_1) &\leftarrow \mathit{Holds}(\neg\mathsf{F}, s_1) \end{aligned}$$

These implications alone logically entail $\mathtt{p}(S_0, S_0)$. If, however, the agent's background theory Δ is incomplete in that it does not entail $\mathit{Holds}(\mathsf{F}, S_0)$ nor $\neg\mathit{Holds}(\mathsf{F}, S_0)$, then the (unique) answer set of the program under Δ does not include $\mathtt{p}(S_0, S_0)$. (Drescher, Schiffel, and Thielscher 2009) also presented an operational semantics for ALPs which is incomplete but sound wrt. the declarative first-order semantics. We want to mention that our answer set semantics coincides with this operational semantics for negation-free programs also in the case of incomplete action theories.

Branching vs. Linear Time

The entire concept of Agent Logic Programs is defined independently of the specifics of the underlying action formalism. In particular, the framework can be combined with action calculi that use a branching time structure, like the Situation Calculus, or linear time, like the Event Calculus. In this section, we will show that this flexibility allows us to combine features from different formalisms in that Agent Logic Programs can be employed for both plan generation using branching time, as well as plan verification using linear time.

ALPs and Branching Time

Action theories based on branching time allow to reason about different possible futures, represented by different action sequences that make up the underlying time structure. This class of action formalisms can be used to underpin strategy specifications for the purpose of planning: Consider an answer set S for an Agent Logic Program under an action theory Δ and some ground program atom $Q \in S$, e.g. $Q = \mathtt{elevatorControl}(S_0, T)$. If Δ uses branching time, T encodes a sequence of actions representing a plan that achieves the goals encoded with the atom Q. In this regard, ALPs are the declarative counterpart to procedural GOLOG programs, whose successful executions also determine plans (Levesque et al. 1997). A specific advantage of Agent Logic Programs in comparison is to support specifications of strategies that depend on the failure of other strategies, that is, the non-existence of alternative (sub-)plans.

Example 2 (Continued)**.** In complex domains, it may at any time happen that things do not work any more as they normally do. For example if one of the elevator's engines breaks down, it must switch to a backup engine. This strategy is only necessary if moving fails, which is easily expressed:

$$\begin{aligned} \mathtt{moveTo}(n) \leftarrow \text{not } &\mathit{Does}(\mathsf{Move}(n)), \\ &\mathit{Does}(\mathsf{SwitchEngine}),\ \mathit{Does}(\mathsf{Move}(n)) \end{aligned} \quad (15)$$

Note that success of an action simply hinges on whether the underlying action theory *entails* that the action is executable. Conversely, an action fails if the action theory does not entail its executability. This can be generalized to strategies in that a strategy fails if and only if the strategy is not provably successful given the agent's knowledge.

ALPs and Linear Time

Action theories based on linear time allow to reason about a given set of action occurrences, or a *narrative* (Mueller 2006). If this class of formalisms serves as background theory, then an ALP can be used to verify that a given plan or set of events follows a desired structure.

Example 2 (Continued)**.** Consider another variation of $\Delta_{\mathtt{Elevator}}$ where we use linear time and extend it with the narrative $\Delta'_{\mathtt{Elevator}}$ which entails there are initially no requests for floors different from 2 and contains the only action occurrences $\mathit{Does}(\mathsf{Move}(2), S_0, 1)$,[4] $\mathit{Does}(\mathsf{Open}, 1, 2)$, $\mathit{Does}(\mathsf{TurnOff}(2), 2, 3)$, $\mathit{Does}(\mathsf{Close}, 3, 4)$ and $\mathit{Does}(\mathsf{Park}, 4, 5)$. This narrative is easily verified to conform to the elevator control strategy lined out by clause (9): atom $\mathtt{elevatorControl}(S_0, 5)$ is among the skeptical consequences of $P_{\mathtt{Elevator}}$ under $\Delta_{\mathtt{Elevator}} \cup \Delta'_{\mathtt{Elevator}}$.

Planning problems in action theories with linear time are standardly solved using abduction (Shanahan 1989). We can introduce this principle into ALPs as follows.

Definition 6. Let P be an ALP, Δ an action theory, and Q a formula (representing a planning goal) over the signature of P. A set Δ' of ground $\mathit{Does}(\alpha, s_1, s_2)$ instances is a *solution* to Q under P, Δ iff

- $\Delta \cup \Delta'$ is consistent, and
- Q is a skeptical consequence of P under $\Delta \cup \Delta'$.

Static Predicates

Agent Logic Programs as introduced until now allow the user to write normal logic programs to specify strategies for knowledge-based agents. The atoms of these logic programs are then extended with time arguments to give the intended time-dependent meaning of the specification. Sometimes, however, we wish to express properties that do *not* vary over time; properties that express time-independent features of actions, action sequences or behaviors. For example, moving to a floor on speculation of a request bears the risk of wasting time, irrespective of the current state of the world. To express this in an ALP, it is convenient to have access to predicates whose extensions do not vary over time. With these *static predicates* (which are treated as non-progressing

[4]Recall that this is read as $\mathit{Happens}(\mathsf{Move}(2), 0, 1)$.

in computing the progression index, cf. Definition 4), we can assign time-independent properties in a declarative way. The answer sets of an Agent Logic Program can then be inspected for behaviors satisfying particular properties.

Definition 7. Let Π be a logic program signature and $\Pi^{[]} \subseteq \Pi$. For an n-ary relation symbol P from $\Pi^{[]}$ and a vector $\vec{t}$ of n terms, we call $[P(\vec{t})]$ a *static program atom*. Its expansion in a program rule is

$$Exp((\text{not})[P(\vec{t})]) = (\text{not})P(\vec{t}).$$

The following further extension to our example ALP will demonstrate the intended usage of static predicates.

Example 2 (Continued)**.** We enrich the signature of our agent logic program for elevator control by the static predicates `quick` and `slow` with their natural-language meanings. Now we can say that it is quick behavior if the next floor to serve, n, is servable and all definitely requested floors n' are at least as far away as n:

$$\begin{aligned}&\texttt{nextFloor}(n) \leftarrow [\texttt{quick}],\ \texttt{servable}(n),\ \mathit{Holds}(\mathsf{At}(n_c)),\\ &\qquad \mathit{Holds}((\forall n')(\mathsf{Request}(n') \supset |n_c - n| \leq |n_c - n'|))\end{aligned}$$

As before, a floor is servable if it cannot be excluded that there possibly is a request for it. However, it is slow behavior to serve a floor purely on speculation:

$$\begin{aligned}&\texttt{servable}(n) \leftarrow \mathit{Holds}(\mathsf{Request}(n))\\ &\texttt{servable}(n) \leftarrow [\texttt{slow}],\ \text{not } \mathit{Holds}(\mathsf{Request}(n)),\\ &\qquad\qquad\qquad \text{not } \mathit{Holds}(\neg\mathsf{Request}(n))\end{aligned}$$

Finally, we say that quick and slow are antonyms and thus their respective meta-behaviors are mutually exclusive:

$$[\texttt{quick}] \leftarrow \text{not } [\texttt{slow}] \qquad [\texttt{slow}] \leftarrow \text{not } [\texttt{quick}]$$

The ALP containing clauses (6–10) together with the ones above will have at least two different answer sets, each of which describes a meta-behavior: e.g., an answer set containing the atom `quick` means that all strategies therein are in accordance with the meta-behavior of being quick.

From GOLOG to Agent Logic Programs

This section clarifies the relative expressiveness of GOLOG and Agent Logic Programs by providing a translation of one into the other. We first define the relevant GOLOG constructs; for technical ease, we treat procedure names as function symbols into a new sort PROC of the underlying Situation Calculus signature.

Definition 8. Let Σ be a Situation Calculus signature with R a function into sort PROC, A a function into sort ACTION, φ a fluent formula, x a variable and $\vec{v}$ a sequence of terms. A *complex action* δ is inductively defined by having one of the following forms:

1. primitive action $A(\vec{v})$
2. test action $\varphi?$
3. procedure call $R(\vec{v})$
4. sequence $\delta_1; \delta_2$
5. nondeterministic choice of two actions $\delta_1 | \delta_2$
6. nondeterministic choice of action arguments $(\pi x)\delta(x)$
7. nondeterministic iteration δ^*

Let $\vec{x}$ be a sequence of variables of sort OBJECT. For a function symbol R into sort PROC and a complex action $\delta(\vec{x})$ with free variables in $\vec{x}$, a *procedure declaration* ρ is of the form **proc** $R(\vec{x})\delta(\vec{x})$ **endProc**. A *program* γ is of the form $\rho_1; \ldots; \rho_n; \delta_0$ where $\rho_1, \ldots, \rho_n$ are procedure declarations and δ_0 is a complex action, the *main program body* of γ.

We now translate the procedural GOLOG programs into declarative Agent Logic Programs. The function $\mathfrak{P}(\cdot)$ creates the ALP rules for a given GOLOG construct and will naturally be defined by structural induction. The translation follows the straightforward form that one would expect given the logic programming implementation of the GOLOG interpreter (Levesque et al. 1997).

Definition 9. Consider a fixed Situation Calculus signature Σ and let δ be a complex action with parameters $\vec{v}$. Denote by Q_δ the program atom built from a fresh predicate symbol and the arguments $\vec{v}$. The complex action δ gives rise to the Agent Logic Program $\mathfrak{P}(\delta)$ as follows.

$$\begin{aligned}\mathfrak{P}(A(\vec{v})) &\stackrel{\text{def}}{=} \{Q_{A(\vec{v})} \leftarrow \mathit{Does}(A(\vec{v}))\}\\ \mathfrak{P}(\varphi?) &\stackrel{\text{def}}{=} \{Q_{\varphi?} \leftarrow \mathit{Holds}(\varphi)\}\\ \mathfrak{P}(R(\vec{v})) &\stackrel{\text{def}}{=} \{Q_{R(\vec{v})} \leftarrow R(\vec{v})\}\\ \mathfrak{P}(\delta_1; \delta_2) &\stackrel{\text{def}}{=} \{Q_{\delta_1;\delta_2} \leftarrow Q_{\delta_1},\ Q_{\delta_2}\} \cup \mathfrak{P}(\delta_1) \cup \mathfrak{P}(\delta_2)\\ \mathfrak{P}(\delta_1|\delta_2) &\stackrel{\text{def}}{=} \{Q_{\delta_1|\delta_2} \leftarrow Q_{\delta_1}\} \cup\\ &\qquad \{Q_{\delta_1|\delta_2} \leftarrow Q_{\delta_2}\} \cup \mathfrak{P}(\delta_1) \cup \mathfrak{P}(\delta_2)\\ \mathfrak{P}((\pi x)\delta(x)) &\stackrel{\text{def}}{=} \{Q_{(\pi x)\delta(x)} \leftarrow Q_{\delta(x)}\} \cup \mathfrak{P}(\delta(x))\\ \mathfrak{P}(\delta^*) &\stackrel{\text{def}}{=} \{Q_{\delta^*}\} \cup \{Q_{\delta^*} \leftarrow Q_\delta,\ Q_{\delta^*}\} \cup \mathfrak{P}(\delta)\end{aligned}$$

The Agent Logic Program corresponding to a procedure declaration $\rho = \textbf{proc}\ R(\vec{x})\delta(\vec{x})\ \textbf{endProc}$ is given by

$$\mathfrak{P}(\rho) \stackrel{\text{def}}{=} \{R(\vec{x}) \leftarrow Q_{\delta(\vec{x})}\} \cup \mathfrak{P}(\delta(\vec{x}))$$

Finally, for a GOLOG program $\gamma = \rho_1; \ldots; \rho_n; \delta_0$ we set

$$\mathfrak{P}(\gamma) \stackrel{\text{def}}{=} \mathfrak{P}(\delta_0) \cup \bigcup_{i=1}^{n} \mathfrak{P}(\rho_i)$$

The logic program signature of $\mathfrak{P}(\gamma)$ thus contains predicates for all functions of sort PROC in Σ, and the predicates Q_δ for all complex actions δ occurring in γ.

Note that the translation yields a definite program, which obviates the fact that definite Agent Logic Programs are already as expressive as GOLOG. It is straightforward from Example 2 how to syntactically manipulate any Situation Calculus axiomatisation Δ such that it complies with our definition of an action theory. In this special case, the action theory's entailment relation $\vdash$ will be instantiated to standard first-order logical entailment $\models$.

Given this immediate correspondence between the action theories underlying a given GOLOG program γ and its associated ALP $\mathfrak{P}(\gamma)$, we will identify these theories in the sequel and can now assess the correctness of our translation.

Intuitively, we would have something like a one-to-one correspondence of the plans accepted by the programs γ and $\mathfrak{P}(\gamma)$. More formally, for a GOLOG program γ with main program body δ_0 and the unique answer set S for $\mathfrak{P}(\gamma)$ under Δ, we need that for all ground situations σ, we have

$$\Delta \models Do(\gamma, S_0, \sigma) \quad \text{iff} \quad Q_{\delta_0}(S_0, \sigma) \in S$$

where the ternary macro $Do(\gamma, s, s')$ defines in second-order logic the semantics of GOLOG program γ executed from s to s' (Levesque et al. 1997). For a full-fledged formal proof however, we would face the same hurdles as the authors of the original GOLOG paper, who already recognized the difficulty of formally pinning down the exact sense in which their Prolog-based GOLOG interpreter is correct. But given that our translation in effect defines for each GOLOG program an ALP-based interpreter, it should be straightforward to adapt the proof of correctness they refer to (Levesque, Lin, and Reiter 1997) for our purpose.

Computational Complexity

In this section we briefly focus on the complexity of deciding whether a given set of atoms is an answer set for an ALP under an action theory and if such a set exists. Since ALPs are parametric in action theories Δ and their associated entailment relation $\vdash$, the complexity of these problems depends on the complexity of deciding whether $\Delta \vdash A$ for an atom A over Δ's signature.

Theorem 2. *Let Δ be an action theory with entailment relation $\vdash$ and let C be the complexity class of the problem to decide whether $\Delta \vdash A$ for an atom A. Let Q be a finite instantiation of an expanded agent logic program with negation.*

1. *Deciding if a given set S of atoms is an answer set of Q under Δ is in* P^C.
2. *Deciding if Q has an answer set under Δ is in* NP^C.

Proof sketch. Intuitively, we use the candidate set S and a C-oracle to compute the reduct according to Definition 5 in polynomial time. It then remains to verify that the candidate set is the least model of this reduct (a definite logic program). To decide existence of an answer set, we guess an answer set candidate S and proceed as before. $\square$

As a lower bound, NP-hardness of answer set existence immediately follows from the respective complexity for propositional normal logic programs. Whenever deciding $\Delta \vdash A$ is in P, then answer set existence for agent logic programs with negation is thus NP-complete. Likewise, if deciding $\Delta \vdash A$ is PSPACE-complete, then answer set existence for NALPs is also PSPACE-complete.

Conclusion

In this paper we introduced nonmonotonic agent logic programs, an expressive agent specification language possessing two significant advantages: (a) the language is fully declarative, and (b) apart from some mild assumptions regarding the predicates used to assess relevant knowledge, it is independent of the underlying action theory. Both aspects make the ALP approach easy to use and widely applicable.

Existing approaches to strategy specification, on the other hand, are bound to their respective particular action formalisms. The procedural language GOLOG is restricted to Situation Calculus action theories (Levesque et al. 1997). (Son et al. 2006) provide an approach for adding domain-dependent knowledge to planning problems which is limited to domain knowledge expressed in the Action Language $\mathcal{B}$ (Gelfond and Lifschitz 1998). (Hindriks et al. 1998) presented an operational semantics for a fairly complex agent programming language, which later gave rise to a language offering declarative goals (Hindriks et al. 2001). They however use their own tailor-made propositional modal action theory to represent domain knowledge, whereas we can draw upon decades of research into action theories. Furthermore, Agent Logic Programs have a declarative semantics and are conceptually much simpler.

Our approach relies on basic ideas and concepts from answer set programming (ASP). Although we cannot cover this in detail in the present paper, we would like to point out another obvious benefit: many of the extensions of the basic ASP paradigm developed over the last decade can be adapted for ALPs without much effort. This includes programs with strong negation and disjunctive programs (Gelfond and Lifschitz 1991), cardinality and weight constraints (Simons, Niemelä, and Soininen 2002), aggregates (Faber, Pfeifer, and Leone 2011) and the like. We believe that in particular the latter will play an important role whenever the behavior of an agent depends on available resources.

Most relevant, and directly applicable to ALPs, is also the body of work on extending ASP with preferences, see (Delgrande et al. 2004) for a survey. Adding preferences to ALPs will allow for more fine-grained distinctions and more flexible specifications of potential agent behaviors. Finally, existing work in ASP on accessing external knowledge sources (e.g. the work on HEX-programs (Eiter et al. 2005)) provides a promising basis for implementing ALPs.

An interesting line of future work is to investigate the use of more expressive action theories in combination with ALPs. Of particular interest are formalisms like (Scherl and Levesque 2003), which include the axiomatization of sensing actions and how these affect what an agent knows about its environment. Background theories of this expressiveness could be used as the basis for *conditional* strategy execution in analogy to conditional planning (Rintanen 1999).

References

Baumann, R.; Brewka, G.; Strass, H.; Thielscher, M.; and Zaslawski, V. 2010. State Defaults and Ramifications in the Unifying Action Calculus. In *Proc. KR*, 435–444.

Delgrande, J. P.; Schaub, T.; Tompits, H.; and Wang, K. 2004. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence* 20(2):308–334.

Drescher, C.; Schiffel, S.; and Thielscher, M. 2009. A declarative agent programming language based on action theories. In Ghilardi, S., and Sebastiani, R., eds., *FroCoS*, volume 5749 of *LNCS*, 230–245. Trento, Italy: Springer.

Eiter, T.; Ianni, G.; Schindlauer, R.; and Tompits, H. 2005. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *Proceedings of IJCAI-05*, 90–96.

Faber, W.; Pfeifer, G.; and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence* 175(1):278–298.

Gelfond, M., and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9(3/4):365–386.

Gelfond, M., and Lifschitz, V. 1998. Action Languages. *Electronic Transactions on Artificial Intelligence* 3.

Gelfond, M. 2008. Answer sets. In van Harmelen, F.; Lifschitz, V.; and Porter, B., eds., *Handbook of Knowledge Representation*, 285–316. Elsevier.

Hindriks, K. V.; de Boer, F. S.; van der Hoek, W.; and Meyer, J.-J. C. 1998. Formal Semantics for an Abstract Agent Programming Language. In *Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL'97)*, volume 1365 of *Lecture Notes in Computer Science*, 215–229. Springer.

Hindriks, K. V.; de Boer, F. S.; van der Hoek, W.; and Meyer, J.-J. C. 2001. Agent Programming with Declarative Goals. In Castelfranchi, C., and Lespérance, Y., eds., *Proceedings of the Seventh International Workshop on Agent Theories, Architectures and Languages (ATAL'00)*, volume 1986 of *Lecture Notes in Computer Science*, 228–243. Springer.

Lakemeyer, G., and Levesque, H. 2009. A Semantical Account of Progression in the Presence of Defaults. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI-09)*, 842–847.

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.; Lin, F.; and Reiter, R. 1997. Defining Complex Actions in the Situation Calculus. Technical report, Department of Computer Science, University of Toronto.

McCarthy, J. 1968. Programs with common sense. In Minsky, M., ed., *Semantic Information Processing*, 403–418. MIT Press.

Michael, L., and Kakas, A. 2011. A Unified Argumentation-Based Framework for Knowledge Qualification. In Davis, E.; Doherty, P.; and Erdem, E., eds., *Proceedings of the Tenth International Symposium on Logical Formalizations of Commonsense Reasoning*.

Mueller, E. 2006. *Commonsense Reasoning*. Morgan Kaufmann.

Reiter, R. 2001. *Knowledge in Action*. MIT Press.

Rintanen, J. 1999. Constructing conditional plans by a theorem-prover. *JAIR* 10:323–352.

Scherl, R., and Levesque, H. 2003. Knowledge, action, and the frame problem. *Artificial Intelligence* 144(1):1–39.

Shanahan, M. 1989. Prediction is deduction but explanation is abduction. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1055–1060.

Shanahan, M. 1997. *Solving the Frame Problem: A Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press.

Simons, P.; Niemelä, I.; and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1–2):181–234.

Son, T. C.; Baral, C.; Tran, N.; and Mcilraith, S. 2006. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic (TOCL)* 7(4):613–657.